Journal of Logical and Algebraic Methods in Programming 134 (2023) 100875

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp

Verification of mutable linear data structures and iterator-based algorithms in Dafny ☆

Jorge Blázquez, Manuel Montenegro *, Clara Segura

Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain

ARTICLE INFO

Article history:
Received 10 January 2022
Received in revised form 27 April 2023
Accepted 28 April 2023
Available online 5 May 2023

Keywords:
Program verification
Data structures
Abstract data types
Iterators
Dafny

ABSTRACT

We address the verification of mutable, heap-allocated abstract data types (ADTs) in Dafny, and their traversal via iterators. For this purpose, we devise a verification methodology that makes it possible to implement ADTs based on already existing ones, while maintaining proper encapsulation. Then, we apply this methodology to the specification and implementation of linear collections such as stacks, queues, deques, and lists with iterators. The approach introduced in this paper allows one to progressively refine some aspects of the specification such as iterator invalidation, so that clients of the library can reason about how structural changes to a list affect existing iterators. Finally, we extend our methodology to the verification of client code (i.e., code that makes use of the implemented ADTs) and identify the boilerplate conditions common to all methods that receive and manipulate ADTs.

1. Introduction

Verification platforms, such as Dafny [1,2], allow a programmer to specify the intended behavior of their programs, so that the platform checks that the actual implementation conforms to that behavior. However, this process is not fully automatic; the programmer has to guide the verifier. This task is far from trivial when dealing with a self-contained algorithm, but the task of verifying *Abstract Data Types* (ADTs) is even more challenging.

An ADT is a set of operations that manipulate elements of a given *abstract model*. This model can be realized in several ways, each of them leading to a specific implementation. In this paper we focus on mutable, heap-allocated ADTs, which are those that can be usually found in mainstream imperative languages. In this context, the specification of the operations becomes more complex, since such specifications have to be expressed in terms of the abstract model, while there is a *representation invariant* that delimits well-formed instances and an *abstraction function* that maps those instances to the model values they denote. Moreover, some ADT operations modify the heap as a side effect, so their specification should also demarcate which parts of the heap might be affected. This is known as *framing*, and it is an alternative to other approaches based on separation logic [3]. All these challenges have been addressed in the context of Dafny [4–6], JML [7], Eiffel [8], and Ada [9], among others. In the case of Dafny, an ADT instance usually comes with two *ghost fields* containing, respectively, the model represented by the ADT and the heap region that is owned by that instance (its *footprint*), the latter

☆ Work partially supported by the Spanish MINECO project CAVI-ART-2 (TIN2017-86217-R), MICINN project ProCode (PID2019-1085288RB-C22) and by Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM), co-funded by EIE Funds of the European Union.

* Corresponding author.
E-mail address: montenegro@fdi.ucm.es (M. Montenegro).

https://doi.org/10.1016/j.jlamp.2023.100875

of which might be affected by mutator methods. A ghost field is a field that is used only for verification purposes, but it does not exist at runtime. The ghost fields of the ADT have to be manually updated in each method by means of ghost code. Moreover, the specification of every operation in the ADT has to include boilerplate conditions involving the representation invariant of the instance and the growth of its footprint.

Besides the choice of the model, another challenge in the specification of a hierarchy of ADTs is finding the right level of specification detail in the different abstraction levels. If a given ADT operation is overspecified at the most abstract level, the concrete levels are forced to comply with the overspecification as well. If it is underspecified at the most abstract level, then the ADT might not be suitable to the client's use case since they do not have complete information.

In this paper we devise a novel methodology for verifying mutable heap-allocated ADTs and apply it to the specification and implementation of linear collections such as stacks, queues, deques, and lists with iterators. We explain how to progressively refine our specifications into the different levels of abstraction, and identify the new boilerplate conditions that are shared among all the specifications.

Refined specifications can be applied to solve the issue of iterator invalidation on lists. Iterators allow us to traverse a list, but modifications to a list might make existing iterators *invalid*. This means that their behavior is undefined if they are used afterwards. In some languages (such as C++) list modifications might invalidate all iterators, or only a subset of them. We define a generic invalidation policy at the most abstract level, which consists in invalidating all the existing iterators when a list is modified. However, we also refine this policy in order to achieve more fine-grained policies that specify which iterators are invalidated and which are not.

The main contributions of this paper are:

1. The development of progressively refined specifications and the identification of common specification patterns, achieving at the same time proper information hiding and reuse of specification and implementation code.
2. A twofold interpretation of an ADT's footprint, both as a generic unstructured set, and also as an implementation-dependent structured representation (*spine*). The latter eases the task of verification, and allows us to provide function-based definitions of the model and the footprint of a data structure, so we can omit the corresponding ghost fields and their explicit update.
3. A stratified approach to the representation of an instance's footprint in order to support hierarchies of data structures without exposing their private representation details.
4. Examples of specifications and implementations of list iterators with different invalidation policies. Our verified implementations specify how list modifications affect the existing iterators.

This paper builds upon an informal presentation at a Spanish conference [10] and Jorge Blázquez BS Thesis [11]. The main difference of this paper with respect to the previous version is that we also extend our methodology to the clients of the library. This involves the verification of some well-known algorithms. A common pitfall when verifying ADTs is the underspecification of the heap-related side-effects that might be produced as a consequence of a given operation. This usually results in ADT implementations that are flawlessly accepted by the verifier, but hardly usable in practice. In this paper we prove the feasibility of our specifications by implementing and verifying some non-trivial algorithms on lists. We specify how existing iterators are affected by the execution of those algorithms, and we introduce a novel technique (ghost maps) for this purpose. Besides this, we have given more detailed explanations in some sections, and extended our section on related work.

In this paper we start by giving a brief summary of Dafny's features and syntax in Section 2, and explain the standard approach to ADT verification [4]. In Section 3 we explain how we deviate from this approach and explain our decisions in specifying the model and the stratified footprint, and then we show in Section 4 how to split our ADTs into different levels of abstraction and identify the boilerplate conditions to be added to the ADT's interface. In Section 5 we address the specification and implementation of iterators, with several invalidation policies. Section 6 describes the verification methodology to be used at the client's side. This methodology is applied to several examples in Section 7. Finally, in Section 8 we present some related work and in Section 9 we assess the benefits of our methodology and outline future work.

2. A brief overview of Dafny

Dafny [5] is a verification-aware programming language that includes support for program specifications, the latter of which are checked by a static program verifier. Dafny methods and functions are annotated with preconditions, postconditions, invariants and assertions, and the verifier generates verification conditions that are sent to an SMT solver that checks whether they are valid. In Dafny, assertions are checked at compile time, unlike other languages (such as C, C++, and Java) in which they are checked at runtime. This means that we can mathematically prove that our programs are correct for all inputs, not only for those that are tested.

In Dafny, programs contain implementation and specification constructs. Implementations are defined through methods, and specifications through functions and predicates (that is, boolean functions). Methods are subroutines that can be executed at runtime and can have side effects, such as modifying heap objects. Functions, on the other hand, are only present at compile-time and cannot have side-effects; they are used for verification purposes.

```
method Fill(v: array<int>, c: int)
  modifies v
  ensures ∀ k | 0 ≤ k < v.Length • v[k] = c
{
  var i := 0;
  while i < v.Length
    invariant i ≤ v.Length ∧ ∀ k | 0 ≤ k < i • v[k] = c
  {
    v[i] := c;
    i := i + 1;
  }
}
```

Fig. 1. Method that fills all the positions of an array with a given value.

```
class SNode {
  var data: int
  var next: SNode?

  constructor(data: int,
              next: SNode?)
  { ... }
}
```

```
class SinglyLinkedList {
  var head: SNode?    // first node
  var last: SNode?    // last node

  // Create an empty list
  constructor() { ... }
  // Append an element at the end
  method PushBack(elem: int) { ... }
}
```

Fig. 2. Definition of singly linked lists.

Both functions and methods may have `requires` and `ensures` clauses that respectively specify the preconditions and postconditions of a definition. These clauses are specially important in methods, since methods are opaque, in the sense that they do not expose their implementation to their callers. The only information available to the callers about a method's behavior is its specification. Functions, on the other hand, expose their implementation to the callers.[1] Consequently, the latter can prove properties on functions beyond those appearing in their specifications.

Methods can also have a `modifies` clause that tells Dafny which memory locations might be modified during the execution of a method. Functions do not have such a clause, since they cannot have side-effects. Instead, functions may have a `reads` clause that tells Dafny on which memory locations the function depends. If the set of locations in the `modifies` clause of a method is disjoint from that in the `reads` clause of a function, we can ensure that the function, when applied to the same arguments, yields the same result before and after calling the method.

In Fig. 1 we have an example of a method that sets all the elements of an array to a given number. The `modifies` clause specifies that the input array is going to be mutated and the postcondition specifies exactly how. The invariant helps Dafny to verify the postcondition.

So far we have shown the verification of a single method. Now we shall focus on the verification of ADTs. In order to implement them, Dafny provides classes, which may contain fields and methods. For example, in Fig. 2 we show two classes that define a singly linked list, where the specifications and method bodies have been omitted for brevity. We assume that the elements of the list are integer numbers. If `T` is a reference type (e.g. a class) then `T?` denotes the type of possibly null references to objects of type `T`.

Besides the implementation, we have to specify the behavior of an ADT. For this purpose, some additional definitions are needed inside a class [4]:

- *Model*. It is the formal interpretation of an ADT. It is usually a mathematical entity, such as a sequence or a set, but it can also be any other datatype, as long as it is immutable and non-heap based, since immutable types are suitable to formal reasoning. In Dafny, the model is usually represented by a *ghost* field (i.e., a field used only for verification purposes) in the class definition. In the case of singly linked lists, the model is a *sequence*, an immutable data type provided by Dafny, so we add the following ghost field to the `SinglyLinkedList` class:

  ```
  ghost var model: seq<int>
  ```

  The interface of the ADT's methods is described in terms of the model. In particular, the `PushBack` method in `SinglyLinkedList` would have, among others, the following postcondition:

  ```
  ensures model = old(model) + [elem]
  ```

  where `old(model)` represents the state of the `model` field before the method's execution.

[1] It is possible to use attribute `{:opaque}` in a function definition to hide it by default, but this does not prevent client code from unfolding it when necessary by means of statement `reveal`.

```
class SinglyLinkedList {
  ...
  predicate Valid()
    reads this, this.repr
  {
      this ∉ repr ∧ ValidAux(head, model, repr) ∧
        (last ≠ null ⟹ last in repr) ∧ (head = null ⟺ last = null)
  }

  // Declared as 'static', since it does not depend on the fields of the class
  static predicate ValidAux(node: SNode?, model: seq<int>, repr: set<object>)
    reads repr
  {
    (node = null ⟹ model = [] ∧ repr = {}) ∧
    (node ≠ null ⟹ node in repr ∧ |model| > 0 ∧ model[0] = node.data
                      ∧ ValidAux(node.next, model[1..], repr - {node})) ∧
    (node ≠ null ∧ node.next = null ⟹ node = last)
  }
}
```

Fig. 3. `Valid` predicate of singly linked lists.

```
class SinglyLinkedList {
  ...
  method PushBack(elem: int) {
    if (head = null) {
      head := new SNode(elem, null); last := head;
      model := [elem]; repr := {head};
      assert Valid();   // OK
    } else {
      // Create a new node with no successor
      var newSNode := new SNode(elem, null);
      // Link the last node to newSNode, which now becomes the last node
      last.next := newSNode; last := newSNode;
      model := model + [elem]; repr := repr + {current.next};
      assert Valid();   // Error: cannot prove Valid()
    }
  }
}
```

Fig. 4. Implementation of `PushBack` (specification and invariants omitted).

- *Representation* (footprint). It is the set of heap objects owned by each instance of the ADT. It is used for specifying how the heap is affected by the execution of each method. This set is usually represented in Dafny as a ghost field:

  ```
  ghost var repr: set<object>
  ```

  In our example, the representation of a singly linked list is the set of nodes that make up the list. The methods of the class have a `modifies this, repr` clause, which states that it cannot modify any object in the heap beyond those of the representation. However, a method can extend the representation by adding newly allocated objects to it. In particular, this is what `PushBack` does, since it creates a new `SNode` that is connected to the end of the list. As a consequence, `PushBack` has the following postcondition:

  ```
  ensures fresh(repr - old(repr))
  ```

  In this way, the caller of `PushBack` knows that, although the representation may grow after the execution of the method, it will not overlap with the representations of other instances of `SinglyLinkedList`.
- *`Valid` predicate*. This predicate takes on the role of the *representation invariant*, that is, it tells whether an instance is well-formed. However, in this context is also relates the `model` and `repr` fields to the current state of the instance. In this sense, we could say that `Valid` also defines the *abstraction function*,[2] which specifies the model represented by the instance. This predicate depends on the ADT's representation. In Fig. 3 we show the `Valid` predicate of singly linked lists. The relation between the model, the representation, and the nodes of the list is delegated to a recursive predicate `ValidAux`, which traverses the linked list and states that the nodes must belong to the representation and that their values must coincide with the corresponding positions of the model.
  Regarding methods specifications, each method has `Valid()` as a precondition and also `Valid()` as a postcondition, which means that the invariant is preserved by any execution of the method.

The methods that modify the state of the ADT are required to update the `repr` and `model` fields accordingly in order to preserve the representation invariant. This is shown in the implementation of `PushBack` in Fig. 4, where the specification is

[2] provided that `Valid` constrains the `model` field such as it is uniquely determined.

omitted for the sake of brevity. Unfortunately, Dafny cannot prove that `Valid()` holds at the end of `PushBack`. This is because `PushBack` modifies the `next` attribute of the last node in order to connect it to the newly created node, but the node being modified is part of the representation, and since `Valid()` depends on this representation, the representation invariant has to be reestablished after the modification. According to the definition of `ValidAux` (Fig. 3), the validity of a list that starts at a given `node` depends on the validity of the sublist that starts at `node.next`. Therefore, in Fig. 4, after linking `last` to `newSNode`, we can easily establish that `ValidAux` holds for the list that starts from `last`, but then we have to traverse the list backwards in order to reestablish the validity of the rest of the nodes up to the head of the list. Unfortunately, we cannot traverse a singly linked list backwards, since each node only contains a pointer to the next one. Instead, we need a `while` loop in `PushBack` that builds a sequence of nodes in the order in which they are linked.

```
ghost var spine := [];
while (current.next ≠ null) {spine := spine + [current]; current := current.next}
```

This is a *ghost loop*, in the sense that it only accesses ghost variables, so it will not be executed at runtime.

After linking the last node to the newly created one and updating `repr` and `model` accordingly, we would have another ghost loop that traverses the `spine` backwards and reestablishes `ValidAux()` for each node:

```
while (|spine| > 0) { // While the spine is non-empty
  ghost var current := spine[|spine| - 1]; // last element of the spine
  // Prove ValidAux() for the sublist that starts from 'current'
  spine := spine[..|spine| - 1] // Remove last element from spine
}
```

The full code of `PushBack` is not shown here for the sake of brevity (see Appendix A), but the ghost loops and their invariants make the verification far more convoluted; each loop contains at least nine invariants that involve four ghost variables. An alternative to the spine would be to implement `PushBack` by traversing the list with a recursive function instead of a loop, but we would still need ghost parameters to denote the segment of the model and the subset of the representation that are being worked on at each recursive call. The structured representation introduced in Section 3 allows us to maintain the imperative style in `PushBack` without having to resort to ghost loops before and after each modification.

3. Representing ADT verification concepts in Dafny

During the development of the data structures in this work we diverged several times from Leino's approach to verification of ADTs [4]. In this section we explain the differences to his approach and incrementally present our ADT verification methodology.

The footprint of an ADT is usually defined as a set of objects. This is adequately abstract for the interface of the ADT (and we maintain this interface), but implementing the representation under the hood as a set is problematic since we do not have any information about the shape of that representation. To gain more information when verifying list operations we have internally replaced the `repr` field with a *spine*, a sequence of nodes that is ordered as the nodes are linked: first the head, then the next node and so on. Note that the spine is meant as a private field, so that client code is unaware of its existence. From the client's point-of-view, the footprint of an ADT is still an unstructured set of objects, but now it is defined as a `Repr` function instead of a ghost field. In Fig. 5 we implement a singly linked list with such a spine. The `Valid` predicate ensures that the nodes in memory are linked in the same order as in the spine, as pictured in the diagram of Fig. 6.

We call this technique *structured representation*, as opposed to using a set that has no structure. This representation gives useful information that would not be available otherwise. For example, in a singly linked list we cannot directly access the previous node of a given node. We could solve this like we did in Section 2: in essence, by building the spine at the beginning of the method and then using it at the end to finish the verification. By contrast, with a structured representation we do not need to build the spine each time we need it, as it is permanently stored in a ghost field. Additionally, we define the representation invariant in terms of the spine (a sequence), so we can leverage all the properties that Dafny knows about sequences. This is exemplified in Fig. 7, where we define `PushBack` again but using the structured representation. The code needed to verify that method is minimal: update the spine and call a simple lemma that is used across the library, `LastHasLastIndex`.[3]

When we use the spine, obtaining the representation is just a matter of collecting all the nodes stored in it. We produce that set in the `Repr` function, as shown in Fig. 5. Using a function to define the representation, however, does not come without drawbacks: some boilerplate is needed to use this methodology. We leave discussion to Section 4.1.

Once we have a spine we can obtain the model from it and remove the `model` field, as shown in Fig. 5. We call this way of defining the model *calculated model* since it is computed on-demand each time the `Model` function is used in specification

[3] This lemma proves by contradiction that, for every `node` such that `node.next = null`, that node is in fact the last node of the list (that is, `spine[|spine|-1] = node`). It is a class lemma so `spine` is implicit.

```
class SinglyLinkedList {
  var head: SNode?;
  ghost var spine: seq<SNode>;

  function Repr(): set<object>
    reads this
  {
    set x | x in spine
  }

  predicate Valid()
    reads this, Repr()
  {
    ∧ (∀ i | 0 ≤ i < |spine|-1  •  spine[i].next = spine[i+1])
    ∧ (if head = null
         then spine = []
         else spine ≠ [] ∧ spine[0] = head ∧ spine[|spine|-1].next = null)
  }

  function Model() : seq<int>
    reads this, Repr()
    requires Valid()
  {
    seq(|spine|, i ⇒ spine[i].data)
  }
}
```

Fig. 5. Definition of `SinglyLinkedList` class.

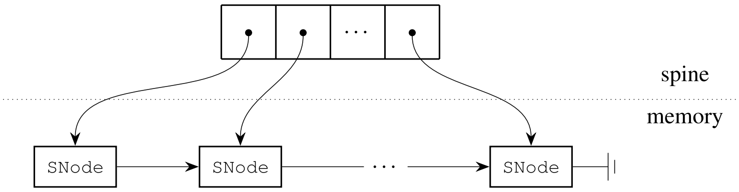

Fig. 6. Spine and nodes.

```
lemma LastHasLastIndex(last: SNode)
  requires last.next = null ∧ last in Repr() ∧ Valid()
  ensures spine[|spine|-1] = last
{
  var i :| 0 ≤ i < |spine| ∧ spine[i] = last;
  assert i ≠ |spine|-1 ⟹ last.next = spine[i].next = spine[i+1] ≠ null;
}

method PushBack(x: int) {
  var n := new SNode(x, null);
  if last = null {
    head := n;
    spine := [n];
    last := n;
  } else {
    LastHasLastIndex(last);
    last.next := n;
    spine := spine + [n];
    last := last.next;
  }
}
```

Fig. 7. `PushBack` method on a singly linked list with spine.

or verification. By using the spine we have gone from a `repr` field and a `model` field in the list to just one field in the list that can produce both the representation and the model. This way we express the abstraction function explicitly with the `Model` function, unlike what we discussed in Section 2, where the abstraction function is defined inside the `Valid` predicate. Additionally, we also make the `Valid` predicate only contain the representation invariant.

Now that we have an implementation of singly linked lists, we are going to implement a stack ADT with it. First, we define the specification of the ADT in Fig. 8. We do so with a trait, a definition that is similar to the interfaces of other languages that lets us specify an ADT without giving an implementation. Then, we try to implement that trait by using the list previously defined but Dafny complaints in the `Repr` function with an error: the `reads` clause should include the `list` field. But, if we add that field to the `reads` clause then we will not comply with the specification, since `Repr` only

```
trait Stack {
  function Repr(): set<object>
    reads this

  predicate Valid()
    reads this, Repr()

  function Model(): seq
    reads this, Repr()
    requires Valid()
}
```

```
class StackImpl {
  var list: SinglyLinkedList;

  function Repr(): set<object>
    reads this
  {
    // error: insufficient reads clause
    list.Repr()
  }
}
```

Fig. 8. Specification and implementation of a stack.

```
type pos = n: int | n > 0

trait Stack {
  ghost const ReprDepth: pos

  function ReprFamily(n: nat): set<object>
    decreases n
    ensures n > 0 ⟹
      ReprFamily(n) ≥ ReprFamily(n-1)
    reads this, if n = 0
                 then {}
                 else ReprFamily(n-1)

  function Repr(): set<object>
    reads this, ReprFamily(ReprDepth-1)
  {
    ReprFamily(ReprDepth)
  }

  predicate Valid()
    reads this, Repr()

  function Model(): seq<int>
    reads this, Repr()
    requires Valid()
}
```

```
class StackImpl {
  var list: SinglyLinkedList;

  function ReprFamily(n: nat): set<object>
    decreases n
    ensures n > 0 ⟹
      ReprFamily(n) ≥ ReprFamily(n-1)
    reads this, if n = 0
                 then {}
                 else ReprFamily(n-1)
  {
    if n = 0 then
      {list}
    else
      {list} + list.Repr()
  }

  predicate Valid()
    reads this, Repr()
  {
    ∧ ReprDepth = 1
    ∧ list.Valid()
  }

  constructor()
    ensures Valid()
    ensures Model() = []
    ensures fresh(Repr())
  {
    ReprDepth := 1;
    list := new List();
  }
}
```

Fig. 9. Specification and implementation of a stack using the stratified representation.

reads `this`. To solve this problem we devise a new technique that we call *stratified representation*. Instead of defining the representation as just one set, we will define a family of representations that is incrementally built, level by level. Each level reads from the previous level. In Fig. 9 we show how we use this technique to specify and implement a stack. We define two levels, the first one (`ReprFamily(0)`) holds the `list` internal field and the second (`ReprFamily(1)`) adds all the representation of the underlying list. In general, the set `ReprFamily(n)` contains the set `ReprFamily(n-1)`, which in Dafny is expressed by `ReprFamily(n) ≥ ReprFamily(n-1)`. The field `ReprDepth` holds the highest level of the family (in this case 1) and it is used to define the `Repr` function that reads from the previous level (`ReprDepth-1`).

Using this technique we can reuse code from other ADTs while hiding the implementation details to the user. As an example of this, in Fig. 10 we show an implementation of a queue using two stacks (an inbox and an outbox). We will use these stacks in a completely abstract way, without access to any of the implementation details, by referencing the `Stack` trait and only using the implementations in the constructor. To do so we need to define a family of representations that includes the representation of both stacks level by level, making sure at each level that we can read everything from the previous levels. If we use the stacks implemented with a linked list that we defined in Fig. 9, the family of representations can be illustrated as in Fig. 11. However, we could also use different implementations of stacks that would result in a different representation without changing the code of `ReprFamily` in the queue. The model is defined by concatenating the model of outbox with the reverse of the model of inbox, and the rest of the implementation and verification is standard.

4. Overview of our ADT library

It is common for library designers to divide their library components into several levels of abstraction. For example, in the case of ADTs that denote collections of elements (such as stacks, lists, maps, sets, etc.), many standard libraries provide

```
class QueueImpl extends Queue {
  var inbox: Stack
  var outbox: Stack

  function ReprFamily(n: nat): set<object>
    decreases n
    ensures n > 0 ⟹ ReprFamily(n) ≥ ReprFamily(n-1)
    reads this, if n = 0 then {} else ReprFamily(n-1)
  {
    if n = 0 then
      {inbox, outbox}
    else if n-1 ≤ inbox.ReprDepth ∧ n-1 ≤ outbox.ReprDepth then
      {inbox, outbox} + inbox.ReprFamily(n-1) + outbox.ReprFamily(n-1)
    else if n-1 > inbox.ReprDepth ∧ n-1 ≤ outbox.ReprDepth then
      {inbox, outbox} + inbox.Repr() + outbox.ReprFamily(n-1)
    else if n-1 ≤ inbox.ReprDepth ∧ n-1 > outbox.ReprDepth then
      {inbox, outbox} + inbox.ReprFamily(n-1) + outbox.Repr()
    else
      {inbox, outbox} + inbox.Repr() + outbox.Repr()
  }

  predicate Valid()
    reads this, Repr()
  { ReprDepth = max(inbox.ReprDepth, outbox.ReprDepth) + 1 ∧ ... }

  function Model(): seq<int>
    reads this, Repr()
    requires Valid()
  { outbox.Model() + rev(inbox.Model()) }

  constructor ()
    ensures Valid() ∧ Model() = [] ∧ fresh(Repr())
  {
    inbox := new StackImpl();
    outbox := new StackImpl();
    ReprDepth := max(inbox.ReprDepth, outbox.ReprDepth) + 1;
  }

  method Enqueue(x: int)
  method Dequeue() returns (x: int)
}
```

Fig. 10. Specification and implementation of a queue using two stacks.

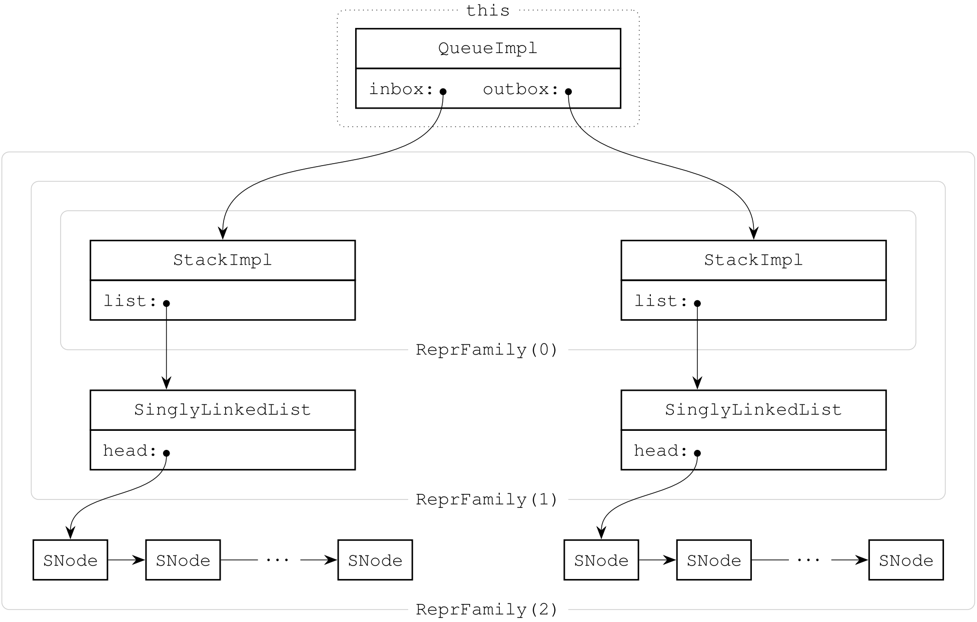

Fig. 11. Stratified representation of a queue that uses two stacks.

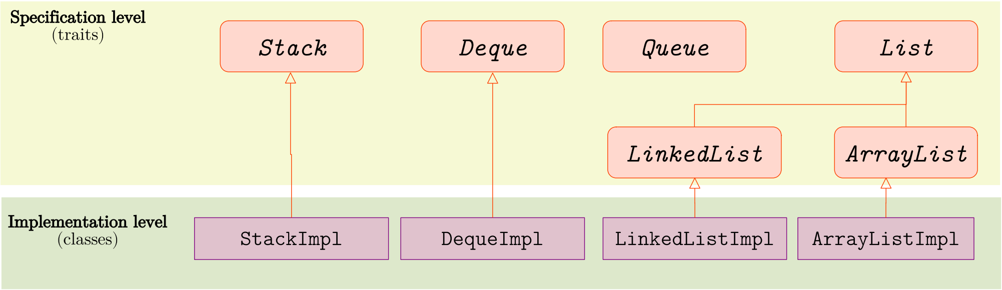

Fig. 12. Specification and implementation levels.

abstract classes that define the *interface* of the corresponding ADTs, and also provide concrete classes with *implementations* of these interfaces. In fact, a single interface could have several implementations. This is the case of Java's `Map` interface and its implementations `TreeMap`, `HashMap`, `LinkedHashMap`, etc.

We maintain the usual distinction between interface and implementation, encapsulating implementation-specific definitions of the ADTs, so that client code can be verified without relying on such definitions. The specification level only contains Dafny traits with interfaces and pre/post specifications, whereas the implementation level contains actual implementations using Dafny classes. We have followed this architecture to develop a library of data structures in Dafny 3.9. In Fig. 12 we depict the specification and implementation levels currently defined in our ADTs library. The source code can be found in our version control repository.[4]

The specification level contains the following *traits* (i.e., abstract classes) with the interfaces of ADTs: `Stack`, `Deque`, `Queue` and `List`. They correspond to the above-mentioned interface level that can be found in Java standard library, for example. The difference between our library and that of Java is that we do not only include type signatures in our methods, but also specifications, which should be general enough to be applicable to any foreseeable implementation. Each trait in this level should declare the `ReprDepth` field, the `Model` and `ReprFamily` functions, and the `Valid` predicate, but without body. An example of such a trait is `Stack`, shown in Fig. 9. A trait also includes methods that are specific to the ADT (`Push`, `Pop`, etc. in the case of stacks) with their specifications, but without implementations. The only exception is the `Repr` function, which is defined in terms of `ReprDepth` and `ReprFamily`, as shown in Fig. 9.

The specification level also contains traits refining other specifications in this level, provided that behavioral subtyping [12] holds: the precondition (resp. postcondition) of a method in the refining trait has to be weaker (resp. stronger) than its counterpart in the refined trait. Such refinements are useful, for example, for specifying ADTs with different iterator invalidation policies. For example, as it will be explained in Section 5, structural modifications to a list (i.e., inserting and removing elements) could invalidate all iterators, or just some of them. For example, in Fig. 12 we represent the `LinkedList` and `ArrayList` traits, whose refined specifications capture a behavior similar to that of `list` and `vector` classes from C++ respectively.[5] For example, the `LinkedList` trait specifies that removing an element from a list does not affect existing iterators, except those pointing to the element being deleted, whereas `ArrayList` specifies that a removal affects those iterators that point to elements whose indices are greater or equal than that of the element being removed.

The implementation level contains classes which implement the abstract methods, functions and predicates of their corresponding traits in the specification level. This includes concrete definitions for `Valid`, `ReprDepth` and `ReprFamily`. For example, the `StackImpl` class of Fig. 9 belongs to this level. In general, clients should not directly access these classes, as this would break modularity. This is because, unlike methods, functions and predicates are not opaque; they reveal their definition to their callers, as explained in Section 2.[6] A client must know that the predicate `Valid` holds before and after calling a method, but it should not know how `Valid` is defined in order to reason about the ADT. This is why a client should only deal with traits, and only mention the classes of the implementation level when creating an instance of the trait. In fact, instance creation could be delegated to factory methods:

```
class StackImpl {
  ...
  static method New() returns (s: Stack)
    ensures s.Valid() ∧ s.Model() = [] ∧ fresh(s.Repr())
  {  s := new StackImpl(); }
}
```

[4] https://github.com/jorge-jbs/adt-verification-dafny/tree/JLAMP22.

[5] We could have used other names which do not include the underlying implementation names, but we preferred to maintain them because many readers will be accustomed to Java and C++ iterators and because these names provide an intuitive view of how iterators are affected.

[6] There we explained that even if we use attribute {:opaque} to hide a definition, we can still unfold it using a statement `reveal`. This would imply, for example, that the verification of client code could rely on the internal definition representation invariant of an instance `x` of an ADT just by executing `reveal x.Valid()`, so breaking encapsulation.

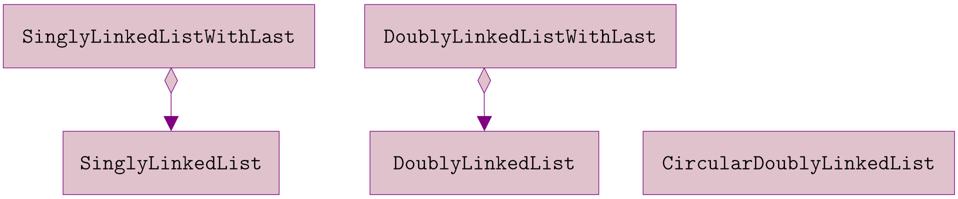

Fig. 13. Underlying data structures.

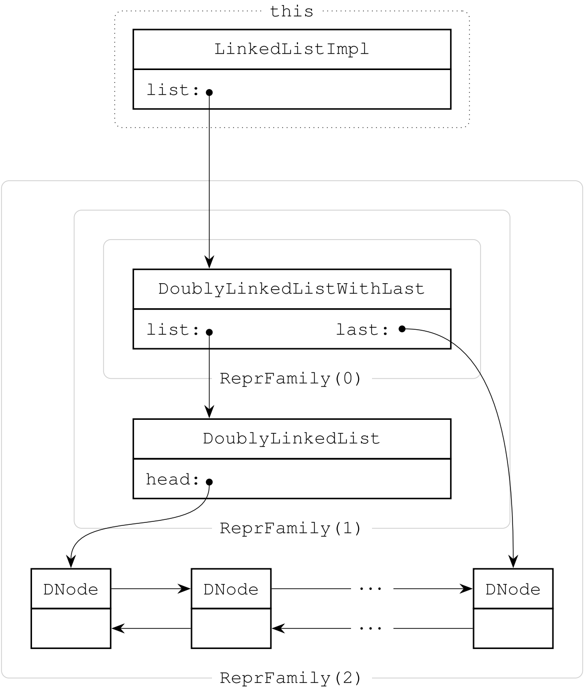

Fig. 14. Stratified representation of `LinkedList`.

In order to facilitate code reuse we have implemented some common underlying data structures, shown in Fig. 13, so that most of the methods in the classes shown in Fig. 12 just delegate their implementations to those defined in a particular data structure. This is the case of the stack implementation of Section 3, which relies on a singly linked list. There is also code reuse among the data structures. For example, the implementation of `DoublyLinkedListWithLast` consists of an instance of a `DoublyLinkedList` with a pointer to the last node. The `LinkedListImpl` is based, in turn, on these two classes. In Fig. 14 we depict the stratified representation of the latter.

Notice that it is possible that the implementation of an ADT may rely on another ADT (for example, the queue of Section 3 was implemented with two stacks). In this case, the class in the implementation level will reference other traits in the specification level.

4.1. Interface of ADT methods

After explaining how the traits and classes are distributed among the different levels, now we focus on the specification of the methods of an ADT. In Fig. 15 we show the specification of the `PushBack` method of the `List` ADT. That method follows the same structure as all the other methods in our methodology, dividing the specification in the following five parts:

1. The framing `modifies`-clause. All methods have this clause, even if they do not alter the model, since they could modify internal state to implement their behavior.
2. The boilerplate pre/postconditions concerning the invariant of the representation: all the ADT methods require as a precondition the validity of the object and ensure its validity in the postcondition.
3. The pre/postconditions concerning the model of the ADT. Methods describe the mutations (or absence of mutations, in case of an observer method) that they do to the model.

```
method PushBack() returns (x: int)
  // 1. Framing clause
  modifies this, Repr()

  // 2. Representation invariant
  requires Valid()
  ensures Valid()

  // 3. Model
  ensures Model() = old(Model()) + [x]

  // 4. Memory allocation
  requires ∀ x | x in Repr() • allocated(x)
  ensures ∀ x | x in Repr() • allocated(x)
  ensures fresh(Repr()-old(Repr()))

  // 5. Iterators
  ensures Iterators() ≥ old(Iterators())
```

Fig. 15. Specification of list's push-back method.

4. The boilerplate pre/postconditions concerning memory allocation, explained below, which are essential when invoking methods on several ADTs in the same context as we will see in Section 6.
5. If the ADT allows iterators, either to traverse it or to modify it, this part is concerned with describing their state after the execution of the method. We will explain this further in Section 5.

The representation set cannot grow by capturing already allocated memory locations as they could be owned by another structure; it can only grow with newly allocated pointers inside the method. So, a postcondition is always added to the mutators saying that all the new elements included in `Repr()` (if any) are fresh. It is not necessary to specify how the footprint is modified, as this depends on the particular implementation, e.g. in the pop method of a stack implemented with an array the set does not change, but when using a linked list of nodes it decreases.

One drawback of defining the representation as a function instead of a field is that, in order to invoke ADT methods, it is essential that Dafny can prove that the objects of the representation are heap-allocated. This is always true, since all objects in Dafny are heap-allocated, but because of the way the axioms of Dafny are defined it is not proved automatically,[7] and we are forced to manually state that fact. One alternative consists in proving in a lemma that all the objects in `Repr()` are heap-allocated and invoke it wherever it is needed. The problem of this option is that the user of the ADT is constantly forced to invoke explicitly the lemma after each method call, which would be annoying. We have decided instead to systematically add that property as a precondition and a postcondition, both in the library code and in the user code. We expect that in the future all these boilerplate properties can be automatically added, but currently the only feature that could help in that front is the `autocontracts` facility, a feature that enforces the application of the verification concepts shown in Section 2, which are incompatible with our methodology.

4.2. Example: reordering numbers

In this section we show an example that shows the usage of linear ADTs. The code is available in Fig. 16. The verification of the example requires boilerplate conditions both in the specification and in the loop invariant but they are omitted in the figure because they are not necessary to understand the example. We delay to Section 6 the explanation of the boilerplate code needed in methods that use ADTs.

Given a sequence of integer numbers stored in an array that is increasingly ordered by their absolute value we want to return the sequence ordered by value. For example, if we receive $0, -1, 2, -4, -8, 9$ we should return $-8, -4, -1, 0, 2, 9$. As the sequence is ordered by absolute value, positive numbers are already ordered, while negative ones are in reverse order. The algorithm traverses the elements of the array and splits them using a method called `Split`: the negative ones are stored on a stack and the positive ones on a queue. Then the stack (negative) values are popped back to the array (in reverse order) and finally the queue (positive) values are dequed back to the array (in the same order). This is done by two auxiliary methods, called `FillFromStack` and `FillFromQueue`.

The main method, called `Reorder`, receives as argument the input array of integers and the two ADTs used to solve the problem, i.e. a stack and a queue, so that we can invoke it with any implementation of stacks and queues that meet the interfaces, and the method is verified just by using those interfaces:

```
...//in main method
var v := ScanArray(n);        //read the data from input
var neg: Stack := new ArrayStack();  //also LinkedStack
var pos: Queue := new LinkedQueue(); //also ArrayQueue
```

[7] https://github.com/dafny-lang/dafny/issues/1152.

```
method Reorder(neg: Stack, pos: Queue, v: array<int>)
  requires ∀ i | 0 ≤ i < v.Length - 1 • abs(v[i]) ≤ abs(v[i+1])
  requires neg.Empty() ∧ pos.Empty()
  ensures multiset(v[..]) = old(multiset(v[..])) //sorted permutation
  ensures ∀ i | 0 ≤ i < v.Length - 1 • v[i] ≤ v[i+1]
{
  Split(v, neg, pos);
  ghost var negm := neg.Model();
  ghost var posm := pos.Model();
  var i := 0;
  i := FillFromStack(v, i, neg); //put back the elements
  i := FillFromQueue(v, i, pos);
    //proves ∀ i | 0 ≤ i < |v|-1 • v[i] ≤ v[i+1]
  LastLemma(negm, posm, v[..]);
}
method Split(v: array<int>, neg: Stack, pos: Queue)
  requires neg.Empty() ∧ pos.Empty()
  requires ∀ i | 0 ≤ i < v.Length-1 • abs(v[i]) ≤ abs(v[i+1])
  ensures ∀ x | x in neg.Model() • x < 0
  ensures ∀ x | x in pos.Model() • x ≥ 0
  ensures ∀ i | 0 ≤ i < |neg.Model()|-1 •
          abs(neg.Model()[i]) ≥ abs(neg.Model()[i+1])
  ensures ∀ i | 0 ≤ i < |pos.Model()|-1 •
          abs(pos.Model()[i]) ≤ abs(pos.Model()[i+1])
  ensures multiset(neg.Model()) + multiset(pos.Model()) = multiset(v[..])
{
  var i := 0;
  while i < v.Length
    /* ...more invariants from the postconditions of the method... */
    invariant i ≤ v.Length
    invariant multiset(neg.Model()) + multiset(pos.Model()) = multiset(v[..i])
  { if v[i] < 0
     { neg.Push(v[i]); }
    else
      { pos.Enqueue(v[i]); }
    i := i + 1; }
}
```

Fig. 16. Reordering numbers example.

```
method  ListToArray(l: List, v: array<int>)
{
  var it := l.Begin();
  var i := 0;
  while it.HasNext()
  { var x := it.Next();
    v[i] := x;
    i := i + 1;
  }
}
```

Fig. 17. Traversal of an abstract list with an iterator to fill an array.

```
Reorder(neg, pos, v);
```

5. Iterators

Iterators are an important part of library software [13,14]. At the most abstract level they allow us to traverse a collection regardless of its implementation. We show an example in Fig. 17 in which an iterator traverses an abstract list in order to fill an array with the elements of the list.

Iterators can be used to observe the collection (i.e., as readers) but also to modify it (i.e., as writers). When a collection is modified it is necessary to specify not only how the model of the collection has been modified but also if it is still safe to use the iterators pointing to the collection (i.e., if they are valid), and which properties they meet after the modification. For example, if we delete the last element of a list the specification should ensure that the new model of the list is the old model without the last element, but it could also say that those iterators which were not pointing to the deleted element are still valid and its distance to the beginning of the list (i.e., their index in the sequence) has not changed. If the verifier had that information, we could keep using those iterators in the program. However, it will be impossible to invoke the methods of those iterators that cannot be proved to be valid after the modification because validity is a precondition in all of them. We will say that they have been invalidated. More accurate information about iterators in the specifications allows us to have multiple readers and writers on the same collection.

Verification of iterators implementations and of programs that use them has been addressed in different ways, as we explain in more detail in Section 8, but in most cases their use is constrained to a single writer, i.e., when an iterator modifies

```
method DeleteDup(l: LinkedList)
{
 var it2 := l.Begin();  var it1 := it2.Copy();
 if it1.HasNext() {
  it2.Next();
  while it2.HasNext()
  {
   if it1.Peek() = it2.Peek() {
     it2 := l.Erase(it2);
    } else {
     it2.Next();
     it1.Next();
    }
  }
 }
}
```

Fig. 18. Deletion of repeated elements in a sorted list.

the collection the rest of iterators cannot be proved valid, which in a verification context means that they can no longer be used. As an example, a situation in which we would like to reason about the rest of iterators is the following: assume we want an ADT for representing a queue of people (and their associated information) with an additional operation to delete any person from the queue; and that we implement it with a linked list of people and a map from the person identifier to an iterator pointing to its corresponding information in the list. By using the iterator we can delete the information of one person in constant time but we need to prove that the iterators pointing to the rest of the people are valid if we want to keep using them to delete the information they point to.

Our specifications include different levels of knowledge about iterators after a modification of the collection. The specification of iterators on abstract lists allows us to have either multiple readers or only one writer, as it happens for example in Java [15]. However, the specification of iterators on linked lists and array lists allows us to have multiple readers and writers at the same time in a similar way to C++ [16].

In Fig. 18 we show an example of a method that deletes duplicated elements in a sorted linked list using two iterators on it, pointing to the current element (iterator `it2`) and to the previous one (iterator `it1`), in order to compare consecutive elements. Method `Erase` receives one iterator which is used to delete from the list the element pointed by it, and returns a new iterator pointing to the next element in the list. In the example, iterator `it2` is used to delete a duplicated element and, after that, it points to the following element in the list. The more refined specification of linked lists with iterators will allow us to prove (see Section 7) that both iterators are valid along the loop execution. However, this method cannot be verified on abstract lists as their specification does not allow us to prove that iterator `it1` is valid.

In Section 7 we will also explain quicksort on lists, where iterators are used to delimit the sublist of the argument list on which the algorithm acts.

5.1. Iterators specification

As we have explained before, we define traits `List` and `ListIterator` for specifying abstract lists with iterators that have no invalidation policy. At this abstraction level we do not know which iterators remain valid if the list is modified by inserting or deleting elements, so the postconditions of those methods only establish that the set of iterators may grow.

In Fig. 19, we show the trait `ListIterator`. Each iterator has a representation invariant defined by predicate `Valid`; a model defined by function `Index`, which returns the position of the list it is pointing to; and the function `Parent`, which returns the list to which the iterator belongs, such that `Index() <= |Parent().Model()|`. Valid past-the-end iterators meet that `Index() == |Parent().Model()|`.

We use predicate[8] `HasNext` in the precondition to the `Next` method, which returns the current element and advances the iterator by one; it is also a precondition to the `Peek` method, which returns the current element, and to the `Set` method, which allows us to modify the current element. Method `Copy` adds a new iterator to the parent list by copying an existing one. We add to mutator methods postconditions ensuring that the rest of iterators pointing to the parent list remain valid and maintain their index. They allow us to use simultaneously several valid iterators on the same list in order to read or replace the elements of the list.

In Fig. 20 we show the signatures of the trait `List`, where methods `Front`, `PushFront`, `PopFront`, and their corresponding counterparts applied at the back of the list, are omitted. As explained before, a list has a representation invariant, defined by predicate `Valid`, a model returned by `Model`, and the set of objects owned by the list, returned by `Repr`. The iterators pointing to the list are part of that representation set and they are returned by function `Iterators`. The set of iterators is increased by calling the `Begin` method (which returns a new iterator with index 0) and it never decreases. This method also ensures that the rest of iterators pointing to the same list remain valid and maintain their index.

[8] If we want to implement a method with no side effects, we can define a `function method` in Dafny, which has all the properties of functions except for not being ghost code. Dafny does not allow calling a method in an expression, but it allows calling function methods. Function methods that return a boolean are in fact predicates.

```
trait ListIterator {
  function Parent(): List
  predicate Valid()
  function Index(): nat
    ensures Index() ≤ |Parent().Model()|

  function method HasNext(): bool
      ensures HasNext() ⟺ Index() < |Parent().Model()|

  method Next() returns (x: int)
    requires HasNext()
    ensures Parent().Iterators() = old(Parent().Iterators())
    ensures x = Parent().Model()[old(Index())]
    ensures Index() = 1 + old(Index())
    ensures ∀ it | it in Parent().Iterators() ∧ old(it.Valid()) •
        it.Valid() ∧ (it ≠ this ⟹ it.Index() = old(it.Index()))

  function method Peek(): int
    requires HasNext()
    ensures Peek() = Parent().Model()[Index()]

  method Copy() returns (it: ListIterator)
    ensures fresh(it)
    ensures Parent().Iterators() = {it} + old(Parent().Iterators())
    ensures Parent() = it.Parent()
    ensures Index() = it.Index()
    ensures ∀ it | it in old(Parent().Iterators()) ∧ old(it.Valid()) •
         it.Valid() ∧ it.Index() = old(it.Index())

  method Set(x: int)
    requires HasNext()
    ensures Parent().Model()[Index()] = x
    ensures ∀ i | 0 ≤ i < |Parent().Model()| ∧ i ≠ Index() •
         Parent().Model()[i] = old(Parent().Model()[i])
    ensures Index() = old(Index())
    ensures ∀ it | it in old(Parent().Iterators()) ∧ old(it.Valid()) •
         it.Valid() ∧ it.Index() = old(it.Index())
}
```

Fig. 19. Specification of abstract iterators.

```
trait List {
  function Repr(): set<object>
  predicate Valid()
  function Model(): seq<int>

  function Iterators(): set<ListIterator>
   ensures ∀ it | it in Iterators() • it in Repr() ∧ it.Parent() = this

  method Begin() returns (it: ListIterator)
   ensures fresh(it)
   ensures Iterators() = {it} + old(Iterators())
   ensures it.Index() = 0
   ensures it.Parent() = this
   ensures ∀ it | it in old(Parent().Iterators()) ∧ old(it.Valid()) •
           it.Valid() ∧ it.Index() = old(it.Index())

  method Insert(mid: ListIterator, x: int) returns (newt: ListIterator)
   requires mid.Parent() = this
   requires mid in Iterators()
   ensures Model() = insert(x, old(Model()), old(mid.Index()))
   ensures fresh(newt)
   ensures newt.Valid() ∧ newt.Parent() = this ∧ newt.Index() = old(mid.Index())
   ensures Iterators() = {newt} + old(Iterators())

  method Erase(mid: ListIterator) returns (next: ListIterator)
   requires mid.Parent() = this
   requires mid.HasNext()
   requires mid in Iterators()
   ensures Model() = remove(old(Model()), old(mid.Index()))
   ensures fresh(next)
   ensures next.Valid() ∧ next.Parent() = this ∧ next.Index() = old(mid.Index())
   ensures Iterators() = {next} + old(Iterators())
}
```

Fig. 20. Specification of abstract lists with iterators.

```
trait LinkedList extends List {
  method Begin() returns (it: ListIterator)
   ensures ∀ it | it in old(Iterators()) ∧ old(it.Valid()) • it.Valid() ∧
     it.Index() = old(it.Index())

  method Insert(mid: ListIterator, x: int) returns (newt: ListIterator)
   ensures ∀ it | it in old(Iterators()) ∧ old(it.Valid()) • it.Valid() ∧
     if old(it.Index()) < old(mid.Index())  then
        it.Index() = old(it.Index())
     else
        it.Index() = old(it.Index()) + 1

  method Erase(mid: ListIterator) returns (next:ListIterator)
   ensures ∀ it | it in old(Iterators()) ∧ old(it.Valid())
                ∧ old(it.Index()) ≠ old(mid.Index()) • it.Valid() ∧
     if old(it.Index()) < old(mid.Index())  then
        it.Index() = old(it.Index())
     else
        it.Index() + 1 = old(it.Index())
}
```

Fig. 21. Specification of linked lists with iterators.

```
trait ArrayList extends List {
  method Begin() returns (it: ListIterator)
   ensures ∀ it | it in old(Iterators()) ∧ old(it.Valid()) • it.Valid() ∧
     it.Index() = old(it.Index())

  method Insert(mid: ListIterator, x: int) returns (newt: ListIterator)
   ensures ∀ it | it in Iterators() ∧ old(it.Valid()) • it.Valid() ∧
     it.Index() = old(it.Index())

  method Erase(mid: ListIterator) returns (next:ListIterator)
   ensures ∀ it | it in old(Iterators()) ∧ old(it.Valid())
                ∧ old(it.Index()) ≠ old(|Model()|) • it.Valid() ∧
     it.Index() = old(it.Index())
}
```

Fig. 22. Specification of array lists with iterators.

Method `Insert` receives an iterator and an element, inserts the element before the iterator (specified by the function `insert` on the model), and returns a fresh valid iterator pointing to the inserted element. Method `Erase` also receives an iterator to the list, deletes the element pointed to by the iterator (specified by the function `remove` on the model) and returns a fresh valid iterator pointing to the next element. In both methods, the returned iterator can be used to keep traversing the list. None of these methods ensures anything about the validity or the index of the rest of iterators, not even the parameter iterator. This is the reason why the use of iterators on an abstract list is limited to have multiple simultaneous iterators that either read or modify values (via method `Set`) on the list, or just only one iterator that can both read and modify the list by inserting or deleting elements.

More specific iterator behavior policies in `LinkedList` and `ArrayList` allow the use of one or several valid iterators on the same list to modify its structure by inserting and deleting elements. In Fig. 21 and 22 we show the signatures of the linked list trait and the array list trait. We only show the properties concerning the iterators: in every mutator method we have to specify exactly how the validity and the index of the iterators is altered. In a linked list, the iterator behaves as attached to the node it is pointing to, while in the array list it behaves as attached to the index of the list. While method `Begin` behaves the same in both types of lists, the rest of methods differ both in the validity and the index properties.

With respect to the iterators validity, all the iterators remain valid after insertion in both types of lists. But linked lists and array lists differ in the methods that delete elements:

- In an array list all the iterators except the past-the-end iterators are valid. This is related to the fact that all the indices stay unchanged: as there is one element less, the index of past-the-end iterators becomes strictly greater than the model length, so those iterators are no longer valid.
- In a linked list all the iterators except those that point to the deleted element are valid.

With respect to the iterators indices change, linked lists and array list differ in all methods:

- In an array list, the indices of the valid iterators remain unchanged. This implies for example that, when inserting an element, old past-the-end iterators may end up pointing to the last element.
- In a linked list, when inserting an element, the indices of the iterators that are pointing to the left of the inserted element remain unchanged, while those to the right and including the index where the element is inserted are increased

```
method DupElements(l: LinkedList)
  ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid())  • it.Valid() ∧
     if (old(it.Index()) = |old(l.Model())|) then it.Index() = 2*old(it.Index())
     else it.Index() = 2*old(it.Index())+1
{
  var it := l.Begin();
  while it.HasNext()
    //invariant omitted...
  {
    var x := it.Peek();
    var _ := l.Insert(it, x);
    it.Next();
  }
}
method DupElements(l: ArrayList)
  ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) •  it.Valid() ∧
   it.Index()=old(it.Index())
{
  var it := l.Begin();
  while it.HasNext()
    //invariant omitted...
  {
    var x := it.Peek();
    var _ := l.Insert(it, x);
    it.Next();
    it.Next();
  }
}
```

Fig. 23. Duplicating elements of a list.

by one. When deleting an element, the indices of the iterators pointing to the left of the deleted element remain unchanged, while those that point to the right of the deleted element are decreased by one.

We show the different behavior of both types of iterators with a method that duplicates the elements of a list, shown in Fig. 23. If we use a linked list, we can duplicate each element by inserting a copy of the currently visited element (pointed to by `it`) before itself. Notice that we can specify that the previous existing iterators remain valid and which are their new indices. However, in an array list the code is different because after inserting an element, the iterator `it` stays on the inserted element so it has to be moved forward twice in order to reach the following element. The indices of the already existing iterators do not change.

If we wanted to implement this method on an abstract list we would have to replace the sentence that inserts the element by `it := l.Insert(it, x)`, which of course would also be correct for the two types of lists.

5.2. Iterators implementation

We outline in Fig. 24 the implementation of iterators on array lists. An iterator consists of the `parent` array list and an `index` which must be either the length of the parent list or the index of an element to be valid. The list contains as a ghost variable the set of iterators pointing to it, information that is used only for verification purposes.

In linked lists however, shown in Fig. 25, one iterator is just a (possibly null) pointer to a `node` in the parent list (which is of type `DoublyLinkedListWithLast`). In this case the parent properties are available through a ghost variable `parent`, so the doubly linked sequence of nodes is accessed via `parent.list.list` (as we depicted in Fig. 14).

6. Methodology for the verification of client code

In this section we enumerate some methodological aspects to be considered when implementing algorithms that use the ADTs described in the previous section. To illustrate them we revisit the example `ListToArray` of Fig. 17 but now we define it on a `LinkedList` in order to show the additional information we know about iterators in this kind of lists. In Fig. 26 we show the full specification and the invariant of that method.

First of all, in each example we have to add (by hand up to now) all the boilerplate code sections already mentioned in Section 4.1 for the ADTs methods.

In the `modifies` clause we express that the method might modify its parameters and their representations. We will always do this, as part of our methodology, when the parameter of a method is an ADT. This means that the internal state of the ADT may have changed even if the model has not. So we are always forced to write explicitly if the model of the ADT changes or not.

```
modifies l, l.Repr(), v
ensures l.Model() = old(l.Model())
```

```
class ArrayListIteratorImpl extends ListIterator {
  var parent: ArrayListImpl
  var index: nat

  predicate Valid()
    reads this, parent, parent.Repr()
  {
    parent.Valid()
    ∧ 0 ≤ index ≤ parent.size
  }...
}

class ArrayListImpl extends ArrayList {
  var elements: array<int>;
  var size: nat;
  ghost var iterators: set<ArrayListIteratorImpl>

  predicate Valid()
    reads this, Repr()
  {
    0 ≤ size ≤ elements.Length
      ∧ elements.Length ≥ 1
      ∧ ∀ it | it in iterators • it.parent = this
  }...
 }
```

Fig. 24. Implementation of array lists with iterators.

```
class LinkedListIteratorImpl extends ListIterator {
 ghost var parent: LinkedListImpl
 var node: DNode?

 predicate Valid()
   reads this, parent, parent.Repr()
 {
    parent.Valid() ∧
    (node ≠ null ⟹ node in parent.list.list.spine)
 }...
}

class LinkedListImpl extends LinkedList {
 var list: DoublyLinkedListWithLast;
 var size: nat;
 ghost var iters: set<LinkedListIteratorImpl>;

 predicate Valid()
   reads this, Repr()
 {
    size = |list.list.spine| ∧
    list.Valid() ∧
    ∀ it | it in iters • it.parent = this ∧ {it} ∩ {this} = {}
 }...
}
```

Fig. 25. Implementation of linked lists with iterators.

We also specify that ADTs' representation invariants hold on exit provided that they hold on entrance. The parameters are required to be valid, and the results and the mutated parameters should also be valid if we want to use them later.

```
requires l.Valid()
ensures l.Valid()
```

We also have to express that ADTs' representations are heap-allocated on entrance and exit and that any new pointers in them are freshly allocated.

```
requires ∀ x | x in l.Repr() • allocated(x)
ensures fresh(l.Repr()-old(l.Repr()))
ensures ∀ x | x in l.Repr() • allocated(x)
```

When using several ADTs in the same context we additionally have to ensure that the parameters are pairwise disjoint from the point of view of memory allocation on entrance, and they remain disjoint on exit. This way, if two ADTs objects' footprints are disjoint and they can only grow with fresh memory locations, when we invoke a mutator on one of the objects we can prove that the footprints remain disjoint. This is essential to prove that when one of the objects is modified the validity of the other object has not been affected so we can invoke its methods. These conditions can also be considered boilerplate code.

```
method  ListToArray(l: LinkedList, v: array<int>)
  modifies l,l.Repr(), v

  requires l.Valid()
  ensures l.Valid()                                             //invariant

  requires ∀ x | x in l.Repr()  •  allocated(x)
  ensures fresh(l.Repr()-old(l.Repr()))                         //invariant
  ensures ∀ x | x in l.Repr()  •  allocated(x)                  //invariant

  requires {v} ∩ ({l}+l.Repr()) = {}
  ensures {v} ∩ ({l}+l.Repr()) = {}                             //invariant
  requires v.Length = |l.Model()|
  ensures v[..] = l.Model()
  ensures l.Model() = old(l.Model())                            //invariant

  ensures l.Iterators() ≥ old(l.Iterators())                    //invariant
  ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid())  •  //invariant
                 it.Valid() ∧ it.Index()=old(it.Index())
{
  var it := l.Begin();  var i := 0;
  while it.HasNext()
    decreases |l.Model()| - it.Index()
    invariant it.Valid() ∧ it.Parent() = l
    invariant i ≤ |l.Model()| ∧ v[..i] = l.Model()[..i] ∧ it.Index() = i
    //the rest of properties marked as invariant in the specification
  {
    var x := it.Next();
    v[i] := x;
    i := i + 1;
  }
}
```

Fig. 26. Full specification and verification of method `ListToArray`.

```
requires {v} ∩ ({l} + l.Repr()) = {}
ensures  {v} ∩ ({l} + l.Repr()) = {}
```

If the ADT uses iterators we must add a postcondition saying that the set of iterators may grow. Those iterators used inside the method to traverse or modify the ADT are added to the set of iterators of the list but they will no longer be accessible because they go out of scope when the method ends, so they need not be declared as valid in the postcondition. We can also add conditions about the validity of previously defined iterators and how their indices (their model) are changed. In the `ListToArray` example iterators do not change as the list is only traversed:

```
ensures l.Iterators() ≥ old(l.Iterators())
ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid())  •
        it.Valid() ∧ it.Index() = old(it.Index())
```

All the described conditions are also required in the invariants of the loops. Only the part of the invariant concerning the model and the changes in the iterators are specific to the loop. In the `ListToArray` example, the non-boilerplate part of the invariant says that from the beginning up to the `i`-th index the elements of the list have been copied onto the array:

```
invariant l.Model() = old(l.Model())
invariant it.Valid() ∧ it.Parent() = l
invariant i ≤ |l.Model()| ∧  v[..i] = l.Model()[..i] ∧ it.Index() = i
invariant ∀ it | it in old(l.Iterators()) ∧ old(it.Valid())  •
          it.Valid() ∧ it.Index() = old(it.Index())
```

In this example the validity and the indices of the old iterators are fixed along the execution of the loop but in some algorithms they may change between iterations (e.g. in the method for duplicating elements), and even more that change may depend on the values of the elements of the list (e.g. in a method for filtering a list).

We have explored different alternatives for specifying the functionality of the methods:

1. Define the functionality through a pure function on the model and prove that the method behaves according to that function. The properties of the method results are then proved apart by lemmas that the client must invoke when needed.
2. Define the properties of the method results directly in the postconditions without using a functional specification. The user of the method does not need to invoke any lemmas to reason with them.
3. Combine 1 and 2.

As an example, let us review the `DupElements` method of Fig. 23 for linked lists. We could define a recursive function `dup` on sequences and use it to specify the method:

```
method DupElements(l: LinkedList)
  ensures l.Model() = old(dup(l.Model()))
```

and then prove the properties that function `dup` satisfies:

```
lemma {:induction xs} SetDup<A>(xs: seq<A>)
ensures ∀ x • x in xs ⟺ x in dup(xs)
ensures |dup(xs)| = 2 * |xs|
ensures ∀ i | 0 ≤ i < |xs| • xs[i] = dup(xs)[2 * i] = dup(xs)[2 * i + 1]
```

A disadvantage of this alternative is that the client has to invoke the lemma to get the properties of the method. One advantage is that it is enough to use the function in the invariant of the loop to verify `DupElements`:

```
invariant l.Model()[..2*i] = old(dup(l.Model()[..i]))
invariant l.Model()[2*i..] = old(l.Model()[i..])
```

Another advantage is that the caller could prove by himself other properties that the method specifier did not consider but that are deductible from the function definition. Finally, as we will see in Section 7, another advantage of using a functional specification is that it helps in the verification of the iterators properties when they depend on the values of the model.

The second alternative, i.e., to provide only the properties, has the advantage that the caller does not need to invoke any lemmas but the disadvantage that the verifier of the method has to foresee which properties may be needed by a user and maybe leave the method underspecified from the user's point of view.

The third alternative, both a functional specification and the properties in the postconditions, has the disadvantage that specifications get larger, but it is the most flexible option from the user's point of view.

With respect to the iterators behavior, we have explored two alternatives. First, we can provide the iterators properties directly, as we did in Section 5 in the postcondition. In the `DupElements` method another postcondition is:

```
ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) • it.Valid() ∧
  if (old(it.Index()) = |old(l.Model())|) then
      it.Index() = 2 * old(it.Index())
  else
      it.Index() = 2 * old(it.Index()) + 1
```

Also in the invariant we have to capture the values of the indices of all the iterators, those that point to the already processed part of the list, and those that point to the not-yet processed part of the list:

```
invariant ∀ iter | iter in old(l.Iterators()) ∧ old(iter.Valid()) •
  iter.Valid() ∧
  (if (old(iter.Index()) < i) then
       iter.Index() = 2 * old(iter.Index()) + 1
   else
       iter.Index() = i + old(iter.Index()))
```

Alternatively, and considering that the indices are numbers, we can provide a map representing how the indices of the iterators have changed and return it as a ghost result of the method:

```
method DupElements(l: LinkedList) returns (ghost mit: map<int, int>)
 ensures mit = map i |
    i in (set it | it in old(l.Iterators()) ∧ old(it.Valid()) • old(it.Index()))
  • if i = |old(l.Model())|) then 2 * i else 2 * i + 1
 ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
        old(it.Index()) in mit • it.Valid() ∧ mit[old(it.Index())] = it.Index()
```

The ghost result `mit` is a map whose domain is the set of indices of the input list that stay in the output list and whose range are the new positions of the elements. So, the input list iterators whose index belong to the map domain remain valid and their new indices are given by the map. The user of the method can then reason in the program using the map:

```
ghost var validSet :=
         (set it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
                   old(it.Index()) < |old(l.Model())|);
var mit1 := DupElements(l);
var mit2 := DupElements(l);
mit := map it | it in mit1 • mit2[mit1[it]];
assert ∀ it | it in validSet • it.Valid() ∧
       mit[old(it.Index())] = 4 * old(it.Index()) + 3;
```

```
method DeleteDup(l: LinkedList)
 requires sorted(l.Model())
 ensures l.Model() = delDup(old(l.Model()), |old(l.Model())|)
 ensures (set x | x in old(l.Model())) = (set x | x in l.Model())
 ensures strictSorted(l.Model())

 ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
               validIt(old(l.Model()), old(it.Index())) •
        it.Valid() ∧ it.Index() = |delDup(old(l.Model()), old(it.Index()))|
{
 var it2 := l.Begin();  var it1 := it2.Copy();
 if (it1.HasNext())  {
  it2.Next();      ghost var j := 1;     ghost var p := 0;
  while (it2.HasNext())
    decreases |l.Model()| - it2.Index()
    invariant 1 ≤ j ≤ |old(l.Model())|
    invariant j + (|l.Model()| - it2.Index()) = |old(l.Model())|
    invariant it2.Index() = |delDup(old(l.Model()), j)| ≤ j
    invariant it2.Index() = it1.Index() + 1
    invariant l.Model()[..it2.Index()] = delDup(old(l.Model()), j)
    invariant l.Model()[it2.Index()..] = old(l.Model())[j..]
    invariant ∀ k | p ≤ k < j  • old(l.Model())[k] = old(l.Model())[j - 1]
    invariant old(l.Model())[j - 1] = l.Model()[it1.Index()]

    invariant ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
       validIt(old(l.Model()), old(it.Index())) ∧ old(it.Index()) < j  •
        it.Valid() ∧
        it.Index() = |delDup(old(l.Model()), old(it.Index()))| < it2.Index()
     invariant ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
       old(it.Index()) ≥ j  •
       it.Valid() ∧
       it.Index() = old(it.Index()) - (j - |delDup(old(l.Model()), j)|)
  {
   if (it1.Peek() = it2.Peek())
    {  it2 := l.Erase(it2); }
   else
    {  it2.Next();
       it1.Next();
       p := j;
    }
   j := j + 1;
  }
 PropDelDup(old(l.Model()), l.Model(), |old(l.Model())|, |l.Model()|);
} }
```

Fig. 27. Deletion of repeated elements in a sorted list.

When using maps to represent iterators behavior, the burden of verification moves from the loop invariant verification to proving properties about the map. This reduces the verification time of the method although increases the human-guided verification effort. In Section 7.3 we show some examples.

7. Examples using iterators

In this section we show in detail two examples on linked lists using iterators: a method for deleting repeated elements in a sorted list and the quicksort algorithm. As the boilerplate code has already been explained in Section 6 we will omit it here. We will also omit in the invariants those properties that are postconditions and some intermediate assertions. In Appendix B you can find more examples: a method that returns an iterator pointing to the maximum element, a method for filtering even numbers and a method that merges two sorted lists into a sorted list.

7.1. Deleting repeated elements

We resume the example shown in Fig. 18 in Section 5 for deleting repeated elements in a sorted list. We show in Fig. 27 its specification and the invariant of the loop and explain them in detail below. The refined specification of the method `Erase` in the trait `LinkedList` is essential to verify this method.

The part of the specification concerning the model requires that the list is sorted and ensures that this method computes the recursive function `delDup` on the model of the input list:

```
function delDup(xs: seq<int>, i: int): seq<int>
   //deletes repeated elements in xs[..i]
```

The properties that the set of elements does not change and that each of them appears just once (i.e., the list is strictly sorted at the end) are proved apart with the following lemma, which is invoked only at the end of the method so that in the invariant we do not need to mention these properties.

```
lemma PropDelDup(oxs: seq<int>, xs: seq<int>)
 requires  xs = delDup(oxs, |oxs|) ∧ sorted(oxs)
 ensures  (set x | x in oxs) = (set x | x in xs)
 ensures strictSorted(xs[..|delDup(oxs, |oxs|)|])
```

The part of the specification concerning iterators express that the validity of an iterator depends on the values of the model; those that were pointing to the first occurrence of an element remain valid:

```
function validIt(xs: seq<int>, i: int): bool
requires 0 ≤ i ≤ |xs|
{
 if (i = 0) then true
 else if (i = |xs|) then true
 else xs[i] ≠ xs[i - 1]
}
```

The new index of a valid iterator `it` is the number of distinct elements before `it`, i.e., `|delDup(old(l.Model()), old(it.Index()))|`.

In the code, two iterators `it2` and `it1` point respectively to the current element and to the previous one, in order to compare them and delete the one pointed by `it2` in case they are equal. We use ghost variables `j` to traverse the old model, and `p` to represent the smallest index on the old model such that all the elements in segment `[p..j]` are equal. The element at index `p` is the first occurrence of that element and all the elements between `p + 1` and `j - 1` have been deleted in the new model.

In the invariant we express the relation between the old and the new model, and the indices `j`, `p`, `it2.Index()` and `it1.Index()`:

- In the part of the list which has already been traversed (`[0..j]` in the old model) duplicates have been deleted (`[0..it2.Index()]` in the new model). The number of deleted elements is: `j - |delDup(old(l.Model()), j)|`.

```
invariant 1 ≤ j ≤ |old(l.Model())|
invariant j + (|l.Model()| - it2.Index()) = |old(l.Model())|
invariant it2.Index() = |delDup(old(l.Model()), j)| ≤ j
invariant it2.Index() = it1.Index() + 1
invariant l.Model()[..it2.Index()] = delDup(old(l.Model()), j)
```

- The part of the list which has not been traversed yet has not changed:

```
invariant l.Model()[it2.Index()..] = old(l.Model())[j..]
```

- In the old model, all the elements in `[p..j]` are equal. This guarantees that although maybe the element at `j - 1` has been deleted, `it1` points to an equal element and we can use it to compare with the current element pointed by `it2`.

```
invariant ∀ k | p ≤ k < j • old(l.Model())[k] = old(l.Model())[j - 1]
invariant old(l.Model())[j - 1] = l.Model()[it1.Index()]
```

Concerning the iterators, we have to distinguish between the iterators pointing to the left of `j` and those pointing to the right and including `j`:

- To the left of `j` only iterators with index that meet property `validIt` are valid, and their new index is the number of distinct elements before each, i.e., `|delDup(old(l.Model()), old(it.Index()))|`. These indices will not change any more, as they are less than `it2.Index()`:

```
invariant ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
      validIt(old(l.Model()), old(it.Index())) ∧ old(it.Index()) < j •
    it.Valid() ∧
    it.Index() = |delDup(old(l.Model()), old(it.Index()))| < it2.Index()
```

- To the right of `j` all are valid, and their index has been decreased by the number of deleted elements up to now, i.e., `j - |delDup(old(l.Model()), j)|`:

```
invariant ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
      old(it.Index()) ≥ j •
    it.Valid() ∧
    it.Index() = old(it.Index()) - (j - |delDup(old(l.Model()), j)|)
```

```
method Quicksort(l: List, c: ListIterator, n: int)
decreases n
requires 0 ≤ c.Index() ≤ |l.Model()| ∧
         0 ≤ n ≤ |l.Model()| ∧ c.Index() + n ≤ |l.Model()|
ensures multiset(l.Model()[c.Index()..c.Index() + n])=
        multiset(old(l.Model())[c.Index()..c.Index() + n])
ensures sorted(l.Model(), c.Index(), c.Index() + n)
ensures l.Model()[..c.Index()] = old(l.Model()[..c.Index()]) ∧
        l.Model()[c.Index() + n..] = old(l.Model()[c.Index() + n..])
{ var p, q: ListIterator; var nL, nG: int;
  if (n>1)
  {
   p, nL, q, nG := Partition(l, c, n, c.Peek());

   Quicksort(l, c, nL);
   assert sorted(l.Model(), c.Index(), c.Index() + nL);

   Quicksort(l, q, nG);
   assert sorted(l.Model(), q.Index(), c.Index() + n);
   assert sorted(l.Model(), c.Index(), c.Index() + nL);
}}

method  Partition(l: List, c: ListIterator, n: int, x: int)
  returns (p: ListIterator, nL: int, q: ListIterator, nG: int)
requires 0 ≤ n ≤ |l.Model()| ∧ 0 ≤ c.Index() ≤ c.Index() + n ≤ |l.Model()|
ensures 0 ≤ c.Index() ≤ p.Index() ≤ q.Index() ≤ c.Index() + n ≤ |l.Model()|

ensures multiset(l.Model()[c.Index()..c.Index() + n])=
        multiset(old(l.Model()[c.Index()..c.Index() + n]))
ensures l.Model()[..c.Index()] = old(l.Model()[..c.Index()]) ∧
        l.Model()[c.Index() + n..] = old(l.Model()[c.Index() + n..])
ensures ∀ z | c.Index() ≤ z < p.Index()  •  l.Model()[z] < x
ensures ∀ z | p.Index() ≤ z < q.Index()  •  l.Model()[z] = x
ensures ∀ z | q.Index() ≤ z < c.Index() + n  •  l.Model()[z]>x
ensures nL = p.Index() - c.Index() ∧ nG = c.Index() + n - q.Index()
ensures x in multiset(old(l.Model()[c.Index()..c.Index() + n]))
        ⟹ p.Index() < q.Index()
{ p := c.Copy(); q := c.Copy(); var r := c.Copy();
  var i := 0;nL := 0;nG := 0;

  while i < n
  { if r.Peek()>x {
      nG := nG + 1;
    } else if r.Peek() = x {
      Swap(l, q, r, c, n);
      q.Next();
    } else {
      Swap(l, q, r, c, n);
      Swap(l, p, q, c, n);
      p.Next();
      q.Next();
      nL := nL + 1; }
    r.Next();
    i := i + 1;
}}
```

Fig. 28. Quicksort algorithm.

7.2. Quicksort algorithm

Quicksort is a good example for illustrating the relevance of specifying the behavior of iterators when several functions manipulate them. In Fig. 28 we show the most important parts. The method receives an iterator `c` and a length `n` to express the sublist on which quicksort acts: `[c.Index()..c.Index()+n]`.

Given an element `x`, the partition algorithm returns two iterators, `p` and `q` and two numbers, `nL` and `nG`. The numbers represent respectively the number of elements that are strictly lower and greater than `x` in `[c.Index()..c.Index()+n]`. All the elements in the sublist that are equal to `x` are between `p` and `q`, the smaller ones to the left of `p` and the bigger ones to the right of (and including) `q`. We only show the specification and the code of this method. Predicate `sorted(xs,i,j)` determines if the sequence `xs[i..j]` is sorted.

In order to verify this method, we need to prove that after the recursive calls iterators `c`, `p` and `q` remain valid and still have the same index as before. The postcondition guarantees that. The validity of the iterators allows us to invoke the second recursive call after the first one and also to prove that after the second one both the smaller and bigger elements are already sorted, and so is the whole sublist:

```
assert sorted(l.Model(), q.Index(), c.Index() + n); //elements smaller than x
assert sorted(l.Model(), c.Index(), c.Index() + nL);//elements bigger than x
```

```
method Swap(l: List, p: ListIterator, q: ListIterator,
            ghost c: ListIterator, ghost n: int)
requires p.HasNext() ∧ q.HasNext()
requires c in l.Iterators() ∧ c.Valid() ∧ c.Parent() = l
requires 0 ≤ c.Index() < |l.Model()| ∧ 0 ≤ n ≤ |l.Model()| ∧
         c.Index() + n ≤ |l.Model()|
requires c.Index() ≤ p.Index() < c.Index() + n ∧
         c.Index() ≤ q.Index() < c.Index() + n

ensures l.Model()[old(q.Index())] = old(l.Model())[old(p.Index())]
ensures l.Model()[old(p.Index())] = old(l.Model())[old(q.Index())]
ensures ∀ i | 0 ≤ i < |l.Model()| ∧ i ≠ old(p.Index()) ∧ i ≠ old(q.Index()) •
        l.Model()[i] = old(l.Model())[i]
ensures multiset(l.Model()) = multiset(old(l.Model()))

ensures l.Iterators()≥old(l.Iterators())
ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) • it.Valid() ∧
        it.Index() = old(it.Index())

//ADDITIONAL GHOST INFORMATION
ensures c in l.Iterators() ∧ c.Valid() ∧ c.Parent() = l ∧
        c.Index() = old(c.Index())
ensures |l.Model()| = |old(l.Model())| ∧
        0 ≤ c.Index() ≤ c.Index() + n ≤ |l.Model()|
ensures multiset(l.Model()[c.Index()..c.Index() + n]) =
        multiset(old(l.Model()[c.Index()..c.Index() + n]))
{
     var auxq :=q.Peek();
     var auxp:= p.Peek();
     p.Set(auxq);
     q.Set(auxp);
}
```

Fig. 29. Swap elements pointed by two iterators.

```
   assert sorted(l.Model(), c.Index(), c.Index() + n);
```

The method `Swap`, shown in Fig. 29 swaps two elements in the list pointed by two iterators `p` and `q`. When it is invoked from partition we need to prove that the multiset of elements in the sublist remains unchanged after the swap. So we add two ghost parameters, `c` and `n`, to identify the sublist from which the method is invoked. The postconditions ensuring that the iterators are valid and maintain their indices are essential to verify the whole program.

7.3. *Maps for describing iterators behavior*

In this subsection we show some examples of specifying iterators behavior in a functional way using maps, as we explained in Section 6.

In those methods where the iterators do not change, the map returned by the method is the identity map:

```
function idMap(xs: seq<int>): map<int, int>
{map i | 0 ≤ i ≤ |xs| • i}
```

As an example, recall `ListToArray` method shown in Fig. 26. Using maps in the specification we have:

```
method  ListToArray(l: LinkedList, v: array<int>) returns (ghost mit:map<int,int>)
 ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
         old(it.Index()) in mit • it.Valid() ∧ mit[old(it.Index())] = it.Index()
 ensures mit = idMap(old(l.Model()))
```

Let us consider now the `DupElements` method introduced in Fig. 23 of Section 5, where the map changes along the execution of the loop. We define a function for computing the map at each iteration of the loop `i`:

```
function dupMap(xs: seq<int>, i: int): map<int, int>
{map it | 0 ≤ it ≤ |xs| • if (it < i) then 2 * it + 1 else i + it}
```

So, in the specification we write:

```
method DupElements(l: LinkedList) returns (ghost mit: map<int, int>)
ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
        old(it.Index()) in mit • it.Valid() ∧ mit[old(it.Index())] = it.Index()
ensures mit= dupMap(old(l.Model()), |old(l.Model())|)
```

```
method DeleteDupMap(l: LinkedList) returns (ghost mit: map<int, int>)
ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
        old(it.Index()) in mit  •  it.Valid() ∧
        mit[old(it.Index())] = it.Index()
ensures mit = delMap(old(l.Model()), |old(l.Model())|)
{
  mit := delMap(old(l.Model()), 1);

  while (it2.HasNext())
     invariant ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) ∧
       old(it.Index()) in mit  •  it.Valid() ∧
       mit[old(it.Index())] = it.Index() ≤ old(it.Index())
     invariant mit = delMap(old(l.Model()), j)
  {
    if it1.Peek() = it2.Peek() {
      assert ¬validIt(old(l.Model()), j + 1, j);
      DelMapRange(old(l.Model()), j);

      it2 := l.Erase(it2);

      UpdateMapNV(old(l.Model()), j);
      j := j + 1;

      mit := delMap(old(l.Model()), j);
    } else {
      assert validIt(old(l.Model()), j + 1, j);
      UpdateMapV(old(l.Model()), j);

      it2.Next();
      it1.Next();

      p := j;
      j := j + 1;
      mit := delMap(old(l.Model()), j);
    }
  }
}
```

Fig. 30. Eliminating repeated elements with maps.

and in the loop we just build the appropriate map at each loop iteration:

```
mit := dupMap(old(l.Model()), 0);

while it.HasNext()
 invariant ∀ iter | iter in old(l.Iterators()) ∧ old(iter.Valid()) ∧
           old(iter.Index()) in mit  •  iter.Valid() ∧
           mit[old(iter.Index())] = iter.Index()
 invariant mit = dupMap(old(l.Model()), i)

{ //the rest of the body loop
     mit := dupMap(old(l.Model()), i);
}
```

In these two examples, either returning a map or directly writing the iterators properties is just a matter of style, because such properties depend at most on the loop iteration, but not on the values of the list as it happens in method `DeleteDup` of Fig. 18. In Fig. 30 we show the specification and verification of iterators in this method using maps. In this case, the map also depends on the iteration `j` of the loop, but there are some elements to the left of `j` that have been deleted. The predicate `validIt` establishes which indices correspond to the remaining elements, and the function `nIndex` specifies which are their new indices:

```
function  delMap(xs: seq<int>, j: int): map<int, int>
requires 0 ≤ j ≤ |xs|
{ if (j = 0) then map[]
  else map it | 0 ≤ it ≤ |xs| ∧ validIt(xs, j, it)  •  nIndex(xs, j, it)}
```

In this case predicate `validIt` says that all iterators except those pointing to the left of `j` to repeated elements are valid:

```
predicate validIt(xs: seq<int>, j: int, it: int)
requires  1 ≤ j ≤ |xs| ∧ (0 ≤ it ≤ |xs|)
{((it = 0) ∨ (it = |xs|) ∨ (1 ≤ it < j ∧ (xs[it] ≠ xs[it - 1])) ∨ (j ≤ it < |xs|))}
```

and function `nIndex` follows the ideas we already explained in Section 7.1 about the new indices:

```
function nIndex(xs: seq<int>, j: int, it: int): int
requires 1 ≤ j ≤ |xs| ∧ 0 ≤ it ≤ |xs|
{ if (it = 0) then 0
  else if (it≥j) then it - (j - |delDup(xs, j)|)
  else |delDup(xs, it)|
```

In the loop, see Fig. 30, we need to invoke some lemmas about the relation between `delMap(old(l.Model()), j)` and `delMap(old(l.Model()), j + 1)`. We distinguish two cases, when index `j` is valid and when it is not.

As an example, assume the list values are 2, 2, 3, 3, 3, 4, 4, 5, 6 and that `j` is 5. Currently the map is `[0 := 0, 2 := 1, 5 := 2, 6:= 3, 7 := 4, 8 := 5, 9 := 6]`. As index 5 is valid because $4 \neq 3$, the map does not change. However, if the index `j` were 6, which is not valid, the map `[0 := 0, 2 := 1, 5 := 2, 6:= 3, 7 := 4, 8 := 5, 9 := 6]` turns into `[0 := 0, 2 := 1, 5 := 2, 7 := 3, 8 := 4, 9 := 5]` after deleting the element. Notice that the map to the left of 6 has not changed but to its right the values have been decreased by one. Additionally value 6 is not anymore in the map.

So, in the alternative when `it1.Peek() != it2.Peek`, i.e., index `j` is valid, we need the following property:

```
lemma UpdateMapV(xs: seq<int>, j: int)
requires 0 < j < |xs| ∧ validIt(xs, j + 1, j)
ensures delMap(xs, j + 1) = delMap(xs, j)
```

and when `it1.Peek() == it2.Peek` we need this one:

```
lemma UpdateMapNV(xs: seq < int>, j: int)
requires 0 < j < |xs| ∧ ¬validIt(xs, j + 1, j) ∧ sorted(xs)
ensures ∀ it | it in delMap(xs, j) ∧ it < j  •
        it in delMap(xs, j + 1) ∧ delMap(xs, j + 1)[it] = delMap(xs, j)[it] < j
ensures ∀ it | it in delMap(xs, j) ∧ j < it ≤ |xs|  •
        it in delMap(xs, j + 1) ∧ delMap(xs, j + 1)[it] = delMap(xs, j)[it] - 1
ensures j ∉ delMap(xs, j + 1)
```

In the latter alternative we also need to prove a monotonicity property:

```
lemma DelMapRange(xs: seq<int>, j: int)
requires 0 < j < |xs| ∧ sorted(xs)
ensures ∀ it | it in delMap(xs, j) ∧ it < j  •  delMap(xs, j)[it] < |delDup(xs, j)|
ensures ∀ it | it in delMap(xs, j) ∧ j < it ≤ |xs|  •
        delMap(xs, j)[it]>|delDup(xs, j)|
ensures ∀ it | it in delMap(xs, j) ∧ it = j  •  delMap(xs, j)[it] = |delDup(xs, j)|
```

that we invoke before deleting elements to prove that the indices strictly to the left or right of `j` (which is going to be deleted) stay respectively to the left or the right of `it2.Index()` so they are not going to be erased.

8. Related work

Verification of abstract data types is a well-studied topic in the area of automated verification. There are several ways in which one can map the concrete state of an ADT to an abstract value. One of them is based on *model fields* [17], which are ghost (i.e., specification-only) fields that depend on the concrete state, and are automatically updated whenever the state changes. There are some soundness-related issues when models are specified from constraints on the concrete fields [18]. In particular, the constraints defining the model could be unsatisfiable under a given state. This could happen, for example, when the invariant does not hold at a given program point, so the instance is in an inconsistent state. When these unsatisfiable constraints are assumed, the verification process become unsound, since any formula can be proved from them. Model fields also present some issues regarding modularity: if the model of an ADT depends on fields contained within other classes, the mutator methods of these fields must have a `modifies` clause involving those classes, which breaks modularity. In order to address these issues, Leino and Müller [18,19] build on the *Boogie methodology* which makes distinction between *valid* and *mutable* states of a given object. An object's invariant is only assumed in a valid state. A *pack* statement transitions an object from a mutable to a valid state, provided its invariant can be proved at that point. Besides that, the model field is decoupled from the concrete state, so when the latter is modified, the model has to be updated accordingly when the object is being packed. The calculated model used in our work is closer to model fields, since the `Model()` function is bound to the state. A difference with respect to model fields is that `Model()` requires the object to be `Valid()` and is proved to be terminating. Moreover, if the model depends on another object's field, that object belongs to the representation of the ADT. Therefore, when that field is modified externally, the model is also assumed to change, because of the `reads Repr()` declaration in the latter. That is why our approach does not suffer the modularity issue of model fields explained above; the implementation of mutator methods in model fields is unaware of the owner of the instance being modified.

Some verification systems support explicit class invariants, which are implicitly checked at method boundaries. However, the validity of these invariants can be decoupled from method boundaries by adding an explicit ghost attribute that determines whether the invariant is expected to hold when calling a method [20,19]. In our case, the boilerplate code shown in Section 4.1 contains `Valid()` assertions in the precondition and postcondition of each method in the ADT. If we do not need them to hold at a specific method boundary, these assertions can be removed or weakened.

One of the challenges in the verification of ADTs lies in specifying how objects are related. Our stratified `Repr()` function is based on a layered ownership model. This is similar to Müller's work [21], which introduces the concept of layered object structures. In this scheme, objects are classified into contexts, depending on the ownership level they belong to. If an object references another one living in a different context, there must be an ownership relation between them. Ownership-based relations are somewhat limited when implementing mutually-dependent data structures, in which one component has direct access to the fields of another one. The *visibility technique* [21,19] allows an object to modify the fields of another one, provided the invariant of the latter is restored after such modifications. Unfortunately, this requires the invariant of the object being modified to be visible to the one modifying it. This modularity issue is also present in our approach, since if an object directly modified the fields of another one, it would have to access those fields via the implementation level, where the definition of the invariant is also visible, thus breaking modularity. However, in our library, the only relation between objects that is not ownership-based is the one between iterators and lists and, in our case, iterators do not directly modify the fields of the lists they belong to, so they need not know how the representation invariant of lists is defined; they only need to know that the invariant holds.

Beyond ownership relations, the specification of collaboration between objects has been carried out in the context of the *Observer* and *Iterator* design patterns [22]. Both have been addressed with a subject-observer scheme involving *history invariants* [23]. A history invariant is a binary predicate that specifies how the subject (i.e., the thing being observed) may evolve. An observer's invariant is allowed to depend on the subject, assuming that this invariant is preserved when the subject is modified in accordance to what the history invariant dictates. Another methodology to specify the Observer and Iterator patterns is *semantic collaboration* [24], which does not rely on history invariants.

One of the most comprehensive efforts to specify and verify a full container library is that of Polikarpova et al. [8] in the context of the Eiffel language. This library (called EiffelBase2) includes not only lists, stacks and queues, but also non-linear TADs, such as sets and maps, and an abstract class `Collection` above all of these. In their work, semantic collaboration is applied in order to specify iterations. A difference with respect to our work is that a collection in EiffelBase2 is unaware of the types of iterators traversing it. However, in our library, although iterators can be operated by the client in an abstract way via the `Iterator` interface, each collection is associated with a specific type of iterator. We believe that this is not a limitation in practice, since in many collection libraries (such as those of Java, C++, and C#) iterators are not created directly by the client, but they are obtained from the collection (e.g. by `iterator()` method in Java). Moreover, having refined specifications for iterators is what allows us to specify fine-grained invalidation policies that depend on each implementation.

The specification of iterators was proposed as a verification challenge in the SAVCBS'06 conference [25]. Non-interference properties of iterators have been addressed in several ways: by adding a version field to collections [14], by using separation logic [13], by combining typestates with linear logic [26], in the context of coroutines [27], or in a language with value-based semantics [28]. The invalidation policies of these approaches are limited in the sense that either they prevent the collection from being modified when an iterator is active, or they invalidate all active iterators when the list is modified. Iterators have also been formalized by means of separation logic in a functional style [29,30], where iterators never get invalidated because of immutability, and also in an imperative style [31]. In the latter case, sharing is addressed using fractional permissions, which allows multiple readers and a single writer. Separation logic has also been applied to the specification of *views* in linked-lists [32]. Views are a generalization of iterators, in the sense that they represent a slice of a list, instead of a single position. Views are increasingly popular in some programming languages (e.g. ranges in C++20, read-only slices in Rust). Again, separation logic with fractional permissions can be used to capture updates through a view such that the changes are reflected not only in the whole linked list, but also in the overlapping views. Our methodology is based on method framing rather than on separation logic, since framing is directly supported by Dafny via `modifies` and `reads` clauses. As future work, we plan to study how to adapt our approach to separation logic-based tools.

9. Conclusions and future work

We have provided a new methodology for the specification and implementation of data structures using Dafny aimed at proper abstraction of implementation details, even at the verification level, and at the reuse of code. We have applied this methodology to the standard linear ADTs from the lower level heap-allocated linked data structures up to the highest level of abstraction in such a way that we have been able to use those ADTs in real programming examples without needing the particular properties of the representation invariant. In particular, we have addressed lists with iterators, where accurate information about iterators behavior after insertion or deletion in the list is crucial to reason when several writers are pointing to the same list.

The architecture of our library incrementally hides the details of the lower levels to the ADT user: the specific underlying data structure (e.g. singly or doubly linked list); the shape of the footprint (e.g. a set or a sequence of objects) and the specific representation invariant (e.g linked list or dynamically growing array).

In linked data structures we have used a structured footprint, called spine, which is a sequence of heap-allocated nodes. The advantages of using this kind of footprint make highly recommendable to use it in other linked data structures. The general case is having a functional version of the data structure where each slot contains a pointer to a node making sure that this auxiliary datatype correctly represents the shape of the nodes. For example, if we are verifying a tree ADT, the representation would also be a tree but defined in the functional section of Dafny, and each node of that tree representation would be a pointer to its corresponding heap-allocated node [33].

The reuse of code is achieved by means of a stratified definition of the footprint that allows both to include in the footprint objects that are not available at the top-level class but in internal ones and at the same time to hide such details to the user. This strategy is more general than the usual ghost field and is applicable also in other ADTs.

We have used functions instead of the usual ghost fields to define both the model and the footprint and described the impact this decision has on the specification and verification at the different levels. In return we are forced to add new pre/postconditions that help Dafny to verify properties about heap allocation but all of them are boilerplate, and would not be necessary if Dafny's axioms included explicitly a true property about the allocation of function results. We consider that the advantages beat this drawback.

As future work we plan to apply this methodology to other families of ADTs and data structures, e.g. maps, trees, graphs; and complete and explore other alternatives to iterators formalization. Additionally, we want to explore the possibility of maintaining in the set only the valid iterators, and in that case the set should decrease when iterators are invalidated.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The manuscript contains a link to the source code repository.

Appendix A. Implementing `PushBack` without spine

In the following we show the full code of the `PushBack` method in singly linked lists, as implemented in Section 2.

```
method PushBack(elem: A)
  requires Valid()
  modifies this, this.repr
  ensures model = old(model) + [elem]
  ensures fresh(repr - old(repr))
  ensures Valid()
{
  if (last = null) {
    head := new SNode(elem, null);
    last := head;
    model := [elem] + model;
    repr := { head };
    assert Valid();
  } else {

    // Ghost loop 1: building the spine
    ghost var current := head;
    ghost var spine: seq<SNode> := [];
    ghost var i := 0;
    ghost var curRepr: set<object> := repr;
    while current.next ≠ null
      invariant 0 ≤ i = |spine| < |model|
      invariant ValidAux(current, last, model[i..], curRepr)
      invariant ∀ j | 0 ≤ j < |spine| • spine[j].data = model[j]
      invariant ∀ j | 0 ≤ j < |spine| - 1 • spine[j].next = spine[j + 1]
      invariant ∀ j, k | 0 ≤ j < k < |spine| • spine[j] ≠ spine[k]
      invariant current.data = model[i]
      invariant i = 0 ⟹ current = head
      invariant |spine| > 0 ⟹ spine[0] = head ∧ spine[|spine| - 1].next = current
      invariant repr = curRepr + (set n | n in spine)
      invariant curRepr ∩ (set n | n in spine) = {}
      decreases |model| - i
    {
      curRepr := curRepr - { current };
      spine := spine + [current];
      i := i + 1;
      current := current.next;
    }

    var newSNode := new SNode(elem, null);
    last.next := newSNode;
    last := newSNode;
    model := model + [elem];
    repr := repr + {newSNode};
    curRepr := curRepr + { newSNode };

    // Ghost loop 2: traversing the spine backwards
    while |spine| > 0
      invariant 0 ≤ i = |spine| < |model|
      invariant ValidAux(current, last, model[i..], curRepr)
      invariant repr = curRepr + (set n | n in spine)
      invariant curRepr ∩ (set n | n in spine) = {}
      invariant ∀ j | 0 ≤ j < |spine| - 1 • spine[j].next = spine[j + 1]
      invariant |spine| > 0 ⟹ spine[0] = head ∧ spine[|spine| - 1].next = current
      invariant |spine| = 0 ⟹ current = head
      invariant ∀ j | 0 ≤ j < |spine| • spine[j].data = model[j]
      invariant ∀ j, k | 0 ≤ j < k < |spine| • spine[j] ≠ spine[k]
    {
      current := spine[|spine| - 1];
      spine := spine[..|spine| - 1];
      curRepr := curRepr + { current };
      i := i - 1;
    }

    assert i = 0 ∧ curRepr = repr;
    assert ValidAux(head, last, model, repr);
    assert Valid();
  }
}
```

Appendix B. Code of examples

In this appendix we show the code of examples mentioned in Section 7.

B.1. Maximum element

Here we show a linear algorithm returning a new valid iterator `max` pointing to the maximum element of a list. The rest of the valid iterators are not affected, as shown in the postcondition. We use a different iterator `it` to traverse the list. The set of iterators grows: it contains `max` and `it` but the latter will not be valid when the method ends.

```
method FindMax(l: LinkedList) returns (max: ListIterator)
 requires l.Model() ≠ []
 ensures l.Model() = old(l.Model())

 ensures fresh(max) ∧ max in l.Iterators()
 ensures max.Valid() ∧ max.Parent() = l ∧ max.HasNext()
 ensures ∀ x | x in l.Model() • l.Model()[max.Index()] ≥ x

 ensures ∀ it | it in old(l.Iterators()) ∧ old(it.Valid()) •
    it.Valid() ∧ it.Index() = old(it.Index())
{
  max := l.Begin();
  var it := l.Begin();
  while it.HasNext()
    decreases |l.Model()| - it.Index()
    invariant it in l.Iterators() ∧ it.Valid() ∧ max ≠ it
    invariant it.Index() ≤ |l.Model()|
    invariant ∀ k | 0 ≤ k < it.Index() • l.Model()[max.Index()] ≥ l.Model()[k]
  {
    if it.Peek() > max.Peek() {
      max := it.Copy();
    }
    var x := it.Next();
  }
}
```

B.2. Filtering even numbers

Filtering of a list is another example in which elements are deleted and iterators behavior depends on the values of the list. We define a higher order and polymorphic function to filter the elements of a sequence according to a given boolean function `f`:

```
function FilterR<A>(xs: seq<A>, f: A -> bool): seq<A>
{ if xs = [] then []
  else if f(xs[|xs| - 1]) then FilterR(xs[..|xs| - 1], f) + [xs[|xs| - 1]]
  else FilterR(xs[..|xs| - 1], f)
}
```

and use it both to specify the functionality of the method and the iterators behavior:

```
method filterEven(l: LinkedList)
 ensures l.Model() = FilterR(old(l.Model()), x ⇒ x % 2 = 0 )
 ensures
  ∀ iter | iter in old(l.Iterators()) ∧ old(iter.Valid()) ∧
   validIt(old(l.Model()), old(iter.Index()), x ⇒ x % 2 = 0) •
  iter.Valid() ∧
  iter.Index() = |FilterR(old(l.Model())[..old(iter.Index())], x ⇒ x % 2 = 0)|
```

The following lemma, invoked at the end of the method, proves that all the elements in the output list meet the property, that all the elements in the input list that met the property are in the output list, and that both the multiplicity of each element and their relative order is maintained, i.e., the resulting list is a subsequence of the input list:

```
lemma allProps<A>(oxs: seq<A>, xs: seq<A>, f: A -> bool)
requires xs = FilterR(oxs, f)
ensures ∀ i • 0 ≤ i < |xs| ⟹ f(xs[i])
ensures ∀ i • 0 ≤ i< |oxs| ∧ f(oxs[i]) ⟹ oxs[i] in xs
ensures isSubSec(xs, oxs)
ensures ∀ i • 0 ≤ i < |xs| ⟹ multiset(xs)[xs[i]] = multiset(oxs)[xs[i]]
```

The subsequence property is proved by providing a map from the output list indices to the input list indices such that the relative order is respected:

```
predicate subSec<A>(xs1: seq<A>, xs2: seq<A>, f: map<int, int>)
  {
   ∀ i • (0 ≤ i < |xs1| ⟺ i in f) ∧
   ∀ i | i in f • 0 ≤ i < |xs1| ∧ 0 ≤ f[i] < |xs2| ∧ xs2[f[i]] = xs1[i] ∧
   ∀ i, j | i in f ∧ j in f ∧ i < j • f[i] < f[j]
  }
predicate isSubSec<A>(xs1: seq<A>, xs2: seq<A>)
  {∃ f: map<int, int> • subSec(xs1, xs2, f)}
```

This is the map and the lemma that proves the property:

```
function FilterRMap<A>(xs: seq<A>, f: A -> bool): map<int, int>
{
 if (xs = []) then map[]
 else if f(xs[|xs| - 1]) then
    FilterRMap(xs[..|xs| - 1], f)[|FilterR(xs, f)| - 1 := |xs| - 1]
 else FilterRMap(xs[..|xs| - 1], f)
}
lemma subSecFilter<A>(xs: seq<A>, f: A -> bool)
  ensures subSec(FilterR(xs, f), xs, FilterRMap(xs, f))
```

The ideas for providing the invariant are similar to the previous example. We use variable `i` to traverse the old model and divide the iterators properties to the left and the right of `i`.

```
method filterEven(l: LinkedList)
 ensures l.Model() = FilterR(old(l.Model()), x ⇒ x % 2 = 0 )
 ensures
  ∀ iter |
   iter in old(l.Iterators()) ∧ old(iter.Valid()) ∧
   validIt(old(l.Model()), old(iter.Index()), x ⇒ x % 2 = 0) •
    iter.Valid() ∧
    iter.Index() = |FilterR(old(l.Model())[..old(iter.Index())], x ⇒ x % 2 = 0)|
{
 var it := l.Begin();
 ghost var i := 0;

 while it.HasNext()
  decreases |l.Model()| - it.Index()
  invariant 0 ≤ i ≤ |old(l.Model())|
  invariant |l.Model()| - it.Index() = |old(l.Model())| - i
  invariant l.Model()[it.Index()..] = old(l.Model())[i..]
  invariant l.Model()[..it.Index()] =
               FilterR(old(l.Model())[..i], x ⇒ x % 2 = 0)

  invariant it.Index() = |FilterR(old(l.Model())[..i], x ⇒ x % 2 = 0)| ≤ i
  invariant ∀ iter | iter in old(l.Iterators()) ∧ old(iter.Valid()) ∧
   validIt(old(l.Model()), old(iter.Index()), x ⇒ x % 2 = 0) ∧
   old(iter.Index()) < i  •  iter.Valid() ∧
   iter.Index() =
      |FilterR(old(l.Model())[..old(iter.Index())], x ⇒ x % 2 = 0)| < it.Index()
  invariant ∀ iter |
    iter in old(l.Iterators()) ∧ old(iter.Valid()) ∧ old(iter.Index()) ≥ i  •
      iter.Valid() ∧
      iter.Index() =
      old(iter.Index()) - (i - |FilterR(old(l.Model())[..i], x ⇒ x % 2 = 0)|)
 {
  if (it.Peek() % 2 ≠ 0)
  { assert ¬validIt(old(l.Model()), i, x ⇒ x % 2 = 0);
     it := l.Erase(it);  }
  else
  { assert validIt(old(l.Model()), i, x ⇒ x % 2 = 0);
    assert ∀ iter | iter  in old(l.Iterators()) ∧ old(iter.Valid()) ∧
          old(iter.Index()) = i
         •  iter.Index() = |FilterR(old(l.Model())[..i], x ⇒ x % 2 = 0)| ≤ i;

    var x := it.Next();
  }
  i := i + 1;
 }
 assert it.Index() = |l.Model()| ∧ i = |old(l.Model())|;
 allProps(old(l.Model()), l.Model(), x⇒x % 2 = 0);
}
```

B.3. Merging sorted lists

Merging of two sorted lists into a sorted list requires a big amount of boilerplate code because there are several data structures in scope and three loops, about four times the code shown here. We show the most important parts of the invariants, which are almost the same in the three loops. Predicate `smaller` express that all the elements of a sequence are less or equal than all the elements of another sequence. We do not show the iterators behavior as in this case the input lists are only traversed so their validity and indices do not change:

```
method merge(l1: LinkedList, l2: LinkedList, ml: LinkedList)
 requires ml.Empty()
 requires Sorted(l1.Model()) ∧ Sorted(l2.Model())
 ensures Sorted(ml.Model())
 ensures multiset(ml.Model()) = multiset(l1.Model()) + multiset(l2.Model())
{
 var it1 := l1.Begin(); var it2 := l2.Begin(); var x;
 while (it1.HasNext() ∧ it2.HasNext())
  decreases |l1.Model()| + |l2.Model()|- (it1.Index() + it2.Index())
  invariant l1.Model() = old(l1.Model()) ∧ l2.Model() = old(l2.Model())
  invariant it1.Index() + it2.Index() = |ml.Model()|
  invariant Sorted(ml.Model()) ∧ Sorted(l1.Model()) ∧ Sorted(l2.Model())
  invariant smaller(ml.Model(), l1.Model()[it1.Index()..]) ∧
            smaller(ml.Model(), l2.Model()[it2.Index()..])
  invariant
    multiset(ml.Model())=
    multiset(l1.Model()[..it1.Index()]) + multiset(l2.Model()[..it2.Index()])
 {
  if it1.Peek() ≤ it2.Peek()
    { x := it1.Next(); }
  else
    { x := it2.Next(); }
  ml.PushBack(x);
 }
 assert it1.Index() = |l1.Model()| ∨ it2.Index() = |l2.Model()|;

 while (it1.HasNext())
   invariant it1.HasNext() ⟹ it2.Index() = |l2.Model()|
 { x := it1.Next();
   ml.PushBack(x);
 }

 while (it2.HasNext())
    invariant it2.HasNext() ⟹ it1.Index() = |l1.Model()|
   { x := it2.Next();
     ml.PushBack(x);
   }
 assert it1.Index() = |l1.Model()| ∧ it2.Index() = |l2.Model()|;
}
```

References

[1] K.R.M. Leino, Dafny: an automatic program verifier for functional correctness, in: LPAR-16, in: LNCS, vol. 6355, Springer, 2010, pp. 348–370.

[2] K.R.M. Leino, D. Matichuk, Modular verification scopes via export sets and translucent exports, in: Principled Software Development: Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of His 60th Birthday, Springer, 2018, pp. 185–202.

[3] J. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, 2002, pp. 55–74.

[4] R. Leino, Specification and Verification of Object-Oriented Software, Marktoberdorf International Summer School, 2008, pp. 1–36.

[5] K. Leino, Developing verified programs with Dafny, in: VSTTE 2012, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, p. 82.

[6] R.R. Rubio Cuéllar, Verificación de algoritmos y estructuras de datos en Dafny, TFG en Ingeniería Informática y Matemáticas (Fac. de Informática, UCM), 2016, https://eprints.ucm.es/id/eprint/38702/.

[7] J. Hatcliff, G.T. Leavens, K.R.M. Leino, P. Müller, M. Parkinson, Behavioral interface specification languages, ACM Comput. Surv. 44 (3) (2012).

[8] N. Polikarpova, J. Tschannen, C.A. Furia, A fully verified container library, Form. Asp. Comput. 30 (5) (2018) 495–523.

[9] C. Dross, J.-C. Filliâtre, Y. Moy, Correct code containing containers, in: M. Gogolla, B. Wolff (Eds.), Tests and Proofs, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 102–118.

[10] J. Blázquez, M. Montenegro, C. Segura, Verification of mutable data structures in Dafny: methodological aspects, in: N.E. Martí Oliet (Ed.), Actas de las XX Jornadas de Programación y Lenguajes (PROLE 2021), SISTEDES, 2021, http://hdl.handle.net/11705/PROLE/2021/015.

[11] J. Blázquez, Verification of linked data structures in Dafny, BS thesis, Facultad de Informática, Universidad Complutense de Madrid, 2021, available at https://eprints.ucm.es/id/eprint/66943/.

[12] B.H. Liskov, J.M. Wing, A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. 16 (6) (1994) 1811–1841.

[13] N.R. Krishnaswami, Reasoning about iterators with separation logic, in: SAVCBS'06, ACM, New York, NY, USA, 2006, pp. 83–86.

[14] D.R. Cok, Specifying java iterators with JML and Esc/Java2, in: SAVCBS'06, ACM, New York, NY, USA, 2006, pp. 71–74.

[15] R. Gallardo, S. Hommel, S. Kannan, J. Gordon, S.B. Zakhour, The Java Tutorial: A Short Course on the Basics, 6th edition, Java Series, Addison-Wesley, 2014.

[16] ISO, ISO/IEC 14882:2020 Information Technology — Programming Languages — C++, International Organization for Standardization, Geneva, Switzerland, 2020, https://www.iso.org/standard/79358.html.

[17] Y. Cheon, G. Leavens, M. Sitaraman, S. Edwards, Model variables: cleanly supporting abstraction in design by contract, Softw. Pract. Exp. 35 (2005) 583–599, https://doi.org/10.1002/SPE.649.

[18] K.R.M. Leino, P. Müller, A verification methodology for model fields, in: Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Proceedings, in: Lecture Notes in Computer Science, vol. 3924, Springer, 2006, pp. 115–130.

[19] K.R.M. Leino, P. Müller, Object invariants in dynamic contexts, in: ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Proceedings, Oslo, Norway, June 14–18, 2004, in: Lecture Notes in Computer Science, vol. 3086, Springer, 2004, pp. 491–516.

[20] M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, W. Schulte, Verification of object-oriented programs with invariants, J. Object Technol. 3 (6) (2004) 27–56, https://doi.org/10.5381/jot.2004.3.6.a2.

[21] P. Müller, A. Poetzsch-Heffter, G.T. Leavens, Modular invariants for layered object structures, Sci. Comput. Program. 62 (3) (2006) 253–286, https://doi.org/10.1016/j.scico.2006.03.001.

[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[23] K.R.M. Leino, W. Schulte, Using history invariants to verify observers, in: R.D. Nicola (Ed.), Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Proceedings, in: Lecture Notes in Computer Science, vol. 4421, Springer, 2007, pp. 80–94.

[24] N. Polikarpova, J. Tschannen, C.A. Furia, B. Meyer, Flexible invariants through semantic collaboration, in: FM 2014: Formal Methods - 19th International Symposium, Proceedings, Singapore, May 12–16, 2014, in: Lecture Notes in Computer Science, Springer, 2014, pp. 514–530.

[25] SAVCBS '06: Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems, Association for Computing Machinery, New York, NY, USA, 2006.

[26] K. Bierhoff, Iterator specification with typestates, in: Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems, SAVCBS '06, Association for Computing Machinery, New York, NY, USA, 2006, pp. 79–82.

[27] B. Jacobs, F. Piessens, W. Schulte, VC generation for functional behavior and non-interference of iterators, in: Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems, SAVCBS '06, Association for Computing Machinery, New York, NY, USA, 2006, pp. 67–70.

[28] B.W. Weide, SAVCBS 2006 challenge: specification of iterators, in: Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems, SAVCBS '06, Association for Computing Machinery, New York, NY, USA, 2006, pp. 75–77.

[29] C. Dubois, R. Rioboo, Verified functional iterators using the FoCaLiZe environment, in: Software Engineering and Formal Methods - 12th International Conference, Proceedings, SEFM 2014, Grenoble, France, September 1–5, 2014, in: Lecture Notes in Computer Science, Springer, 2014, pp. 317–331.

[30] F. Pottier, Verifying a hash table and its iterators in higher-order separation logic, in: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 3–16.

[31] J.G. Malecha, G. Morrisett, Mechanized verification with sharing, in: Theoretical Aspects of Computing - ICTAC 2010, 7th International Colloquium, Natal, Proceedings, Rio Grande do Norte, Brazil, September 1–3, 2010, in: Lecture Notes in Computer Science, Springer, 2010, pp. 245–259.

[32] J.B. Jensen, L. Birkedal, P. Sestoft, Modular verification of linked lists with views via separation logic, in: Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs, FTFJP 2010, Maribor, Slovenia, June 22, 2010, ACM, 2010, 4.

[33] J. Blázquez, Verification of tree-like data structures with iterators in Dafny, Master's Thesis, Facultad de Informática, Universidad Complutense de Madrid, 2022, available at https://eprints.ucm.es/id/eprint/75213/.