Handout 2
A Verified Hashtable

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

25 April, 2023

Contents

This handout is due on Friday, May 12, at 23h59m. The exact details on how to turn in your solution will be made available at a later date.

This handout assumes you have some understanding of how hashtables are implemented in terms of a growable array of (singly-linked) bucket lists.

As a reference, my implementation of **Task 2** is roughly 300 lines of Dafny code, with comments and generous use of linebreaks (e.g. multiple `requires` and `ensures` clauses in pre- and post-conditions and multiple `invariant` clauses in loops, rather than using conjunctions). Don't panic. Don't leave the assignment for the last minute. Ask for help.

For your convenience, Appendix A lists the supplied code in unformatted form.

1 Implementation

Your goal is to develop a generic map implementation based on an hashtable using an array whose elements are singly-linked lists of pairs (i.e., bucket lists). For the purposes of this assignment, you will implement the map using a Dafny class with the following signature (which you will naturally have to implement and extend with invariants, ghost state, pre- and post-conditions):

```
datatype List<T> = Nil | Cons(head:T,tail:List<T>)
datatype Option<T> = None | Some(elem:T)

  class Hashtable<K(==,¬new),V(¬new)> {

    var size : int
    var data : array<List<(K,V)⟩

    function hash(key:K) : int

    function bucket(k:K,n:int) : int

    constructor(n:int)

    method clear()

    method resize()

    method find(k:K) returns (r:Option<V>)

    method remove(k:K)

    method add(k:K,v:V)

  }
```

In the code above, we define the algebraic data types `List<T>` and `Option<T>`. List is the standard functional-style definition of a singly-linked list of `Cons` cells. This will be the type you will use to implement a bucket list. The Option type allows us to represent an optional value of type `T`: if we have a value of type `T` available we use the `Some` constructor, whereas not having such a value is represented by the `None` constructor. This type is used as a result type for operations that might not produce a value. For instance, the find method returns a result of type `Option<V>` since the looked-up key may not exist in the hashtable.

Hash and Bucket. The hash function will be left **undefined** in your implementation (i.e., you **must not** give it a body at all). Instead, you will define a (simple) **specification** for hash which will allow you to use it in the bucketing function `bucket`. The `bucket(k:K,n:int)` function takes a key `k` and an array length `n` and must produce the index of the bucket-list for key `k`, according to the given array length.

Constructor. The constructor takes the initial size of the table and performs the appropriate initializations.

Clear. The `clear` method deletes all the contents of the hashtable.

Resize. The resize method allows the hashtable capacity to grow as needed during insertions (e.g. doubling the capacity of the hashtable). As you may recall, this is a somewhat expensive operation since it requires allocating a new array and *rehashing* the old array contents, meaning that all elements of *each* bucket list must have their hashes recalculated (via the `bucket` function using the new array length) in order to fill the freshly created array correctly. **Hint:** In terms of implementation, it will be helpful to split this method in two – `resize` simply allocates the new array and calls a `rehash` method on each existing bucket list. The `rehash` method *recursively* traverses the given bucket list and for each key-value pair in the list, rebuckets the key and adds the key-value pair to the appropriate bucket in the *new* array.

Find and Remove. The `find` method takes a key `k`, computes its bucket and attempts to find the key-value pair in the corresponding bucket list. If the list does not contain an appropriate key-value the method returns `None`, otherwise returns the corresponding value (wrapped in the appropriate `Some` constructor). This method should rely on the following list-based function:

```
function list_find (k:K,l:List<(K,V)>) : option<V>
   ensures match list_find(k,l) {
     case None ⇒ ∀ v • ¬mem((k,v),l)
     case Some(v) ⇒ mem((k,v),l)
   }
 {
   match l {
     case Nil ⇒ None
     case Cons((k',v),xs) ⇒ if k=k' then Some(v) else list_find(k,xs)
   }
 }
```

The `remove` method behaves similarly, trying to `list_find` the given key in the corresponding bucket list. If an appropriate key-value pair exists, remove will update the bucket list by removing the pair, relying on the following list removal function:

```
function list_remove(k:K,l:List<(K,V)>) : List<(K,V)>
    decreases l
    ensures ∀ k' , v • mem((k',v),list_remove(k,l)) ⟺
       (mem((k',v),l) ∧ k ≠ k')
 {
   match l {
     case Nil ⇒ Nil
     case Cons((k',v),xs) ⇒ if k=k' then list_remove(k,xs) else
                                 Cons((k',v),list_remove(k,xs))
   }
 }
```

Add. The `add` method will insert (or change) the association of key `k` in the hashtable to value `v`. To do so, the hashtable may need to be `resized` if full. The simplest way is to then `remove` the key (which will not produce an effect if the key does not exist in the hashtable) and then add the new hashtable binding to the (tail of the) appropriate bucket list.

2 Specification and Verification

Your hashtable must maintain the following invariants:

1. All key-value pairs are in the appropriate bucket list.
2. The hashtable and its contents implement exactly a `map`.

Point 1 above will likely require defining an auxiliary predicate of the form:

```
ghost predicate valid_hash(d:array<List<(K,V)>>,i:int) { ... }
```

which specifies that all key-value pairs of bucket list `d[i]` are in the correct bucket of `d`. Point 2 will require relating all contents of the hashtable with an adequate *abstract state* (and vice-versa!). A predicate such as:

```
ghost predicate valid_data(k:K,v:V,m:map<K,option<V>>,
                                    d:array<List<(K,V)>>){ ... }
```

relating the existence of the pair `(k,v)` somewhere in `d` with a corresponding association in `m`.

2.1 Verification Challenges

Below you will find some hints and suggestions on how to reason and prove your code correct.

2.1.1 Specifying the Rehash method

The method that will likely require a more sophisticated *specification* is the `rehash` method suggested above (as an auxiliary for the `resize` method). The method is intended to be used within a loop in *resize*, needing to make use of most of the information available at that point in the code:

- Note that `rehash` is an auxiliary method;
- It will likely be useful to pass in some "useless" parameters so they can be used in the specification (e.g. the size of the original array);
- Despite being a recursive method, it acts as a loop in the sense of *requiring* and *ensuring* related or identical properties;
- You will likely need to characterize the lengths of the given arrays (which are passed as arguments);
- You will require the correctness of the "old" array in terms of its values being correctly bucketed;
- Each recursive step rehashes one more pair in the hashtable (from the given list);
- The method additionally requires that:
  - For all keys that fall in buckets up to (but excluding) the given bucket in the "old" array, the data is consistent with the logical map;
  - For keys that fall into the bucket under consideration, any key-value pair that should logically be in the map must be in the list being rehashed or *already* in the new array;

    - For all the others, they are not *yet* in the new array;

- The method establishes that:
    - All key-value pairs in the new array are hashed correctly;
    - All key-value pairs that fall into all buckets up to and including the one being considered are consistent with the logical map (and the new array), otherwise they are not in the new array.

2.1.2 Reasoning with Quantifiers

It is very likely that your invariants and specifications require the use of quantifiers. You may have noticed that quantifiers in Dafny are often associated with a concept called a *trigger* (for instance, hover your mouse over a checked quantifier in the Dafny VSCode extension and you will see its triggers).

Because first-order logic is in general undecidable, Dafny must use approximate or incomplete reasoning when dealing with quantified logical formulas. A trigger is simply a syntactic pattern involving quantified variables that acts as an heuristic for Dafny's underlying solver to do something with a quantifier. With a universal quantifier, the trigger instructs Dafny when to *instantiate* the quantified formula with other expressions. For instance, in the formula

```
∀ x {:trigger P(x)} • P(x) ∧ Q(x)
```

the annotation disables trigger inference and specifies that the trigger for the quantifier is only `P(x)`. This trigger means that whenever Dafny finds an expression of the form `P(...)`, the quantifier will be instantiated with ... plugged in for `x`.

In the code below:

```
method test()
    requires ∀ x {:trigger P(x)} • P(x) ∧ Q(x)
    ensures Q(0)
{ }
```

Dafny cannot verify the postcondition even though it follows logically from the precondition because the trigger `P(x)` is not found and so Dafny never actually instantiates the precondition. However, if we add the seemingly useless assertion **`assert`** `P(0)` to the method body, the method can now be checked because of the way triggers operate in Dafny.

When some quantified formula follows "obviously" but Dafny is unable to prove it (e.g. you try to **`assert`** it and Dafny fails), it can often be due to quantifiers not actually being instantiated as we intuitively think of them. For instance, to prove a quantifier using another quantifier, we must be sure that all the appropriate triggers have been selected so that the reasoning goes through. It is not uncommon to have the following pattern arise (where `A` is some concrete property of interest):

```
  predicate P(x:int) { A(x) }
  predicate Pred() {
    ∀ x • P(x)
  }
  method Test()
    requires Pred()
    ensures ∀ x • A(x)
 {}
```

Dafny is unable to automatically prove the post-condition of the method, and attempts to simply `assert P(0)` or any finite number of such assertions fail to prove the post-condition. The issue is that Dafny has correctly inferred that the trigger in the quantifier in the definition of `Pred()` is `P(x)`, but simply asserting `assert P(0)` instantiates the quantifier with `0`, from which `∀ x • A(x)` does not follow.

If we instead assert `∀ x • P(x)` we are now using the trigger correctly, but the post condition still fails to check. The issue now is that the quantifier in the assertion itself also has triggers that must be used correctly. The inferred triggers in this case are `P(x)` and `A(x)`. If we use the assertion:

```
assert ∀ x  •  P(x) ∧ A(x)
```

the post-condition can now be verified because all the quantifiers are being appropriately triggered. Logically, the assert is trivial because `A(x)` is just the definition of `P(x)`, but the assert turns out to be crucial in making the proof go through due to the way quantifiers work in a verifier like Dafny.

To summarize, you will often need to use asserts of the form above to "convince" Dafny of logically obvious facts due to quantification and triggers.

3 Tasks and Grading

As mentioned above, your main goal is to implement a verified hash table. Since some functional code is required, feel free to use the code supplied in Appendix A.

3.1 Task 1

Implement the hashtable and prove it preserves the invariant described in Section 2 of this document.

3.2 Task 2

Having completed **Task 1** successfully, adjust your specification to use the dynamic frames pattern discussed in class to preserve information hiding. Note that this will require adjusting loop invariants with extra seemingly "obvious" information pertaining to pointers.
Caveat Emptor: If you are very confident, you may skip **Task 1** and develop your verified implementation only using dynamic frames (without losing any points). I strongly advise against this decision.

3.3 Grading

You are expected to use the techniques covered in lecture to ensure the ADT interface does not break abstraction. The handout is divided in two tasks which will be graded independently. As a baseline, if you are able to complete **Task 1** successfully (meaning with an adequately abstract specification that captures the properties mentioned in this document) you should expect a good grade.

If you adequately specify and verify the methods in **Task 1** according to the invariant described in Section 2 but are unable to find an adequate specification of `rehash`, you can still expect a passing grade.

A Supplied Code

```
datatype List<T> = Nil | Cons(head:T,tail:List<T>)
datatype Option<T> = None | Some(elem:T)
ghost function mem<T>(x:T,l:List<T>) : bool {
  match l {
    case Nil => false
    case Cons(y,xs) =>  x==y || mem(x,xs)
  }
}
ghost function length<T>(l:List<T>) : int {
  match l {
    case Nil => 0
    case Cons(_,xs) => 1 + length(xs)
} }
function list_find<K(==),V(!new)>(k:K,l:List<(K,V)>) : Option<V>
  ensures match list_find(k,l) {
    case None => forall v :: !mem((k,v),l)
    case Some(v) => mem((k,v),l)
  }
{
match l {
    case Nil => None
    case Cons((k',v),xs) => if k==k' then Some(v) else list_find(k,xs)
  }
}
function list_remove<K(==,!new),V(!new)>(k:K,l:List<(K,V)>) : List<(K,V)>
  decreases l
  ensures forall k' , v :: mem((k',v),list_remove(k,l)) <==>
     (mem((k',v),l) && k != k')
{
match l {
    case Nil => Nil
    case Cons((k',v),xs) => if k==k' then list_remove(k,xs) else
                   Cons((k',v),list_remove(k,xs))
} }
class Hashtable<K(==,!new),V(!new)> {
  var size : int
  var data : array<List<(K,V)>>
  function hash(key:K) : int
  function bucket(k:K,n:int) : int
  constructor(n:int)
  method clear()
  method resize()
  method find(k:K) returns (r:Option<V>)
  method remove(k:K)
  method add(k:K,v:V)
}
```