Lab 2: Implementing verified hash tables

15-414: Automated program verification

Lab goals

In this lab, we have a look at how to write verified data structures using type invariants. After studying a commented example, you will have to write a provably correct implementation of hash tables in Why3.

Lab instructions

Although we completed this lab without writing any additional helper lemma or code assertion, it should definitely put more pressure on automated provers. Therefore, we advise you to keep your code and your invariants as simple and clean as possible to reduce overhead. You are also urged to complete Part 1 as soon as possible, and visit one of the course staff's office hours if you struggled to do this.

If you come to office hours with Part 1 completed at least one week before the deadline, the course staff will be able to help you correct serious flaws in your approach much more effectively, and your final grade is likely to be significantly higher. Even if you did not completely solve the first part, you should still come to office hours where we will give you feedback on your current solution. The purpose of the voluntary "check-in" is to detect any early errors and to help you to successfully write verified hash tables.

Important note: If you are in need of this assistance with the lab, we recommend you to email the instructor/TA from the office hour that you are planning to attend. This will help us to coordinate additional office hours if needed. If none of the current office hours work for you, please let us know and we will arrange an alternative time.

1 An example of a verified data structure

1.1 Definition and type invariants

Let's illustrate how to design verified data structures in Why3 with a toy example. Let's say we want to implement potentially infinite arrays that represent functions from the non-negative integers to some set of values. Such *dynamic arrays* could be initialized without any size attribute to some constant function and the underlying representation would grow dynamically as elements are modified. You can find Figure 1 the Why3 type definition for dynamic arrays.

```
type dyn_array = {
  default: int ;
  mutable data: array int ;
  ghost mutable model: map int int }
invariant {
  forall i:int. 0 <= i < length data ->
  Map.get model i = data[i] }
invariant {
  forall i:int. i >= length data ->
  Map.get model i = default }
by { default = 0 ; data = Array.make 0 0 ; model = const 0 }
```

Figure 1: The type of dynamic arrays in Why3

As you can see, the type of dynamic arrays is a record type[1] with three fields:

- The field `default` corresponds to the value at which every cell of the dynamic array has been initialized. As opposed to the other fields, it is not declared as being `mutable`, which means that its value cannot change after being first set.[2]
- The field `data` is the concrete representation of the segment of the dynamic array

[1] Records in ML correspond to structures in C.

[2] In languages of the C family, fields and variables are mutable by default and can be declared as being immutable using keywords like `const`. In ML, we have the opposite convention.

that has been modified since initialization.

- More interestingly, the field `model` corresponds to an abstract representation of the content of the dynamic array. It is marked as a *ghost field*, which means that it only serves verification purposes and should be eliminated during compilation (more on this later). In Why3, `map` α β refers to the type of mathematical functions from values of type α to values of type β. Therefore, `model` is a function from integers to values. The fact that this abstract representation is consistent with the concrete representation of dynamic arrays (the `data` and `default` fields) is expressed by two *type invariants*.

Type invariants In Why3, a type τ can be annotated with *type invariants*. If J is declared as an invariant for τ, then:

- It has to hold for any instance of τ that is passed to (or returned by) a function.
- Any function f should preserve J in the sense that J must hold on f's arguments of type τ when f returns if it did when f was called. Type invariants can be temporarily violated in the body of f though.

By keyword The by clause ensures the non-vacuity of this type with invariants. If you omit it, a goal with an existential statement is generated which may be challenging for automated provers.

Maps in Why3 In Why3, mathematical functions can be represented using the `map` type that is exported by the `map.Map` and `map.Const` modules from the standard library. This type can be used as follows:

- The constant function that maps any value to v is written `const v`.
- If f is a function, the image $f(x)$ of x by f is written `Map.get f x`. Besides, the function $f\{x \mapsto v\}$ that only differs from f in x where it has value v is written `Map.set f x v`.

In fact, you should already be familiar with the `map` type because you used it in class to model arrays in dynamic logic.

On specifying and verifying data structures What does it mean for a data structure to be correctly implemented? Surely, it means that its implementation matches some specification but how would we specify a data structure in the first place? A standard way to proceed is as follows:

- We define an abstract representation of the data structure as a mathematical object, in our case a function from integers to values. We call this representation the *model* of the data structure.
- We make this representation a *ghost field* in the data structure's type definition. A ghost field is a field whose only purpose is to serve verification and that is eliminated when compiling WhyML code to normal ML code[3]. As a consequence, any expression accessing a ghost field becomes *ghost code* and it cannot modify non-ghost variables or fields.
- Functions manipulating the data structure are specified using its model. For example, the `dyn_set` function that modifies a cell of a dynamic array has the following specification:

```
let dyn_set (d : dyn_array) (i : int) (v : int) : unit =
  requires { i >= 0 }
  ensures { d.model = Map.set (old d.model) i v }
  ...
```

- Type invariants are used to enforce consistency between the abstract model of a data structure and its concrete representation. In our case, the `dyn_array` type declares two invariants:
  - Its model coincides with its `data` field within the bounds of the latter:

    ```
    forall i:int. 0 <= i < length data ->
    Map.get model i = data[i]
    ```

  - Outside the range of its `data` field, its model is a constant function that is equal to the value of its `default` field:

    ```
    forall i:int. i >= length data ->
    Map.get model i = default
    ```

[3]This also explains why ghost fields and variables can contain abstract mathematical objects for which there is no computer representation.

1.2 Implementation

The full implementation of dynamic arrays is shown Figure 2. Here are a few remarks about it:

- The `dyn_make` function creates a new dynamic array. Notice how the curly brackets syntax is used to create a new record of type `dyn_array` with some given initial value for every field. Also, notice how the type invariants of `dyn_array` hold for this record.
- The `dyn_resize` function is for internal use and reallocates the underlying representation of a dynamic array to make it bigger, without impacting its model. Note that the annotation "`ensures { d.model = old d.model }`" is optional as Why3 would be smart enough to generate it implicitly after noticing that the `model` field of `d` is never reassigned in the function body. Finally, remember the syntax for updating a mutable field of a record (`<-`).
- Notice how the `dyn_set` and `dyn_get` functions are specified using only the abstract model of the dynamic array they manipulate.

Exercise Download the file corresponding to this example ("dyn_array.mlw") on the class website and open it with the Why3 IDE. Split every goal and try to understand what every resulting proof obligation stands for. *(You do not have to return this file but we encourage you to do this exercise).*

```
let dyn_make (v: int) : dyn_array =
    { default = v ; data = Array.make 0 v ; model = const v }

  let dyn_resize (d : dyn_array) (n : int) : unit =
    requires { n > length d.data }
    ensures { d.model = old d.model } (* Optional *)
    ensures { length d.data = n }

    let new_data = Array.make n d.default in
    let k = length d.data in
    for i = 0 to k - 1 do
      invariant { forall j. 0 <= j < i -> new_data[j] = d.data[j] }
      invariant { forall j. k <= j < n -> new_data[j] = d.default }
      new_data[i] <- d.data[i]
    done ;
    d.data <- new_data

  let dyn_set (d : dyn_array) (i : int) (v : int) : unit =
    requires { i >= 0 }
    ensures { d.model = Map.set (old d.model) i v }
    if i >= length d.data then dyn_resize d (i + 1) ;
    d.data[i] <- v ;
    d.model <- Map.set d.model i v

  let dyn_get (d : dyn_array) (i : int) : int =
    requires { i >= 0 }
    ensures { result = Map.get d.model i }
    if i < length d.data then d.data[i] else d.default
```

Figure 2: Implementation of dynamic arrays

2 Lab instructions

2.1 Installing CVC4, learning about strategies

This lab should put more stress on automated provers. Therefore, it is useful to install an additional one that is strong in some cases where *Alt-Ergo* and *Z3* are not:

- **CVC4**: this prover is quite similar to Z3. It is not as good with arithmetic but seems to handle recursive predicates (list membership for example) better. In order to install it on Linux, just run:

  ```
  sudo apt-get install cvc4.
  ```

  On MacOs, follow the instructions on this page: `http://cvc4.cs.stanford.edu/downloads/builds/macos/ports/`.

Then, run

```
why3 config --detect-provers
```

and make sure that the new prover you installed is detected.

It may also be useful to learn a bit more about the automated proof strategies that Why3 offers, which perform smart transrofmations on proof obligations while calling multiple provers in parallel. You are strongly encouraged to make use of these in this lab. When opening Why3 IDE, you should see three new buttons that are labeled from "`Auto 0`", "`Auto 1`" and "`Auto 2`". In particular, "`Auto 2`" may be able to prove some goals that cannot be discharged by calling provers directly. You can have a look at the documentation of these strategies here:

```
http://why3.lri.fr/doc-1.2.1/technical.html#sec109
```

You may notice that `Auto 2` refers to two additional provers that we have not mentioned: `Eprover` and `SPASS`. Support for these provers across all platforms is limited, and you should be able to complete this lab without them. Feel free to download and try them out, but do not hand in a session with proofs completed by these provers, as the autograder will not have them, and we will not be able to check your work.

2.2 Getting familiar with the template

We show Figure 3 a part of the template for this homework. Our implementation of hash tables works with a type `key` that is defined to be of type `int` and a function `hash` that must have the property that its result is always positive. In this case we use the function that returns the absolute value of a number (which was used in LiveLab0).
Our hash tables have the following properties:

- The abstract model of a hash table is a partial function from keys to values. More precisely, it has type:

  ```
  type model = map key (option int).
  ```

  That is, the abstract model of a hash table t is a map f_t such that $f_t(k) =$ `Some` v if key k is associated to value v in t and $f_t(k) =$ `None` if k does not belong to t.

- Concretely, a hash table is an array of n buckets, where a bucket is an association list, that is a list of key-value pairs. The i^{th} bucket of the table only contains keys whose hash is equal to i modulo n.

- A hash table also maintains an estimate of how many elements it contains in field `size`. This is useful to detect when the table is getting saturated and thus should be reallocated with more buckets. In this lab, we do **not** ask you to prove that the content of the `size` field is accurate. This is not a huge deal as failing to update it correctly may lead to a slow implementation but never to a functionally incorrect one.

A note on `let` versus `function`: Functions introduced by the "`function`" keyword are pure functions[4] that can be used in **both** specification and code whereas functions introduced by "`let`" can only be used in code. Another difference is that functions introduced by the "`function`" keyword cannot be annotated but the provers can access their body. In contrast, functions introduced by the "`let`" keyword are black boxes that are only seen by provers through their specification.

[4] In the sense that they cannot mutate some state or perform any side effects. There are many other limitations on what functions can be defined using the "`function`" keyword. For example, they cannot feature loops.

```
type key = int

let function hash (k: key) : int =
    ensures { k >= 0 -> result = k }
    ensures { k < 0 -> result = -k }
    if k >= 0 then k
    else -k

lemma hash_nonneg: forall k: key. 0 <= hash k

function bucket (k: key) (n: int) : int = mod (hash k) n

lemma bucket_bounds:
  forall n: int. 0 < n ->
  forall k: key. 0 <= bucket k n < n

type bucket = list (key, int)
type data   = array bucket
type model  = map key (option int)

type hashtbl = {
  mutable size: int ;
  mutable data: data ;
  ghost mutable model: model }
invariant { ... }
by { ... }
```

Figure 3: A part of the template for this lab

2.3 Part I

In this first part, you have to write down invariants for the `hashtbl` datatype and implement a few basic functions on hash tables. You do **not** have to reallocate the table with more buckets when it gets saturated, as this is the object of Part II. You should try to finish Part I before asking for assistance during office hours.

1. Write down type invariants for hash tables, in such a way that any record satisfying them must correspond to a valid representation of a hash table. As mentioned earlier, we do not require field `size` to be consistent. Besides, we recommend that you allow buckets to contain duplicates. You may have to come back to this question as you progress in the lab, as missing an invariant may cause parts of your implementation to be unprovable.[5]

2. Specify and implement a function `create` that takes a positive integer n as an argument and creates a new hash table with n buckets. See the corresponding signature in the template file.

3. Specify and implement a function `bucket_find` to find the value corresponding to a key in a bucket. Use it to specify and implement a function `find` that finds the value associated to a given key in a hash table. See the corresponding signatures in the template file.

4. Specify and implement a function `bucket_remove` to remove a key along with the corresponding value from a bucket. Use it to specify and implement a function `remove` that removes a key along with the associated value in a hash table.

5. Specify and implement a function `add_new` that adds a key-value pair (k, v) to a hash table under the hypothesis that k does not already belong to it. Use it to specify and implement a function `add` that adds a key-value pair (k, v) to a hash table, overwriting an already existing pair featuring k if needed.

[5] By the way, even if your type invariants are complete, it is sometimes useful to add an invariant P that is redundant in the sense that it is implied by another invariant Q. Indeed, proving that Q is an invariant may be much easier for automated provers than proving the $P \to Q$ implication.

2.4 Part II

For the operations on a hash table to be efficient (amortized constant time), the number of elements in it should not exceed its number of buckets[6]. To preserve this property, it may have to be dynamically reallocated with more buckets. In this part, we implement a new version of `add` that performs such reallocation when necessary.

1. Specify and implement a `resize` function that takes a hash table with n buckets as an argument and reallocates it so that it features $2n + 1$ buckets, without changing its content (i.e. its abstract model). Here are some advice:

   - We recommend that `resize` t creates a new empty hash table t' with the right number of buckets and adds every key-value pair of t into it using the existing `add` function, before overwriting the `data` field of t with the one of t'. You can look at the example in section 1.2 for inspiration.
   - Once again, Why3 only enforces type invariants when a function is called and when it returns. In particular, the only way to convince provers that a type invariant holds after a loop given that it held before and is preserved by each iteration is to write an explicit loop invariant about this fact. To do this we suggest to create a predicate `valid_hashtbl` that will check the validity of the invariants of a hash table.

2. We define the predicate

   ```
   non_saturated (t :  hashtbl) = t.size <= length t.data.
   ```

   Specify and implement an `insert` function that is similar to `add`, except that it comes with the additional guarantee that it preserves the non saturated character of its argument:

   ```
   let insert (t : hashtbl) (k : key) (v : int) : unit =
       ensures { non_saturated (old t) -> non_saturated t }
       ...
   ```

   In doing so, you may have to enrich the specification of previously defined functions.

[6] Or at least not by more than a small constant factor.

What to hand back

First make sure that every goal is handled successfully by the provers in your completed version of `lab2.mlw`. Labs should be submitted on the course Canvas website, under the Lab2 assignment. Upload a zip file named AndrewID_lab2.zip, using your Andrew ID. The zip archive should contain the following content:

1. The completed `lab2.mlw` file
2. The session folder generated by Why3 IDE
3. An ASCII text file with extension `.txt` or a .pdf file containing some comments you may want to share with us (optional).

In order to make sure your archive is valid, you can uncompress it and run the following two commands (we will do the same):

```
why3 replay lab2  # should print that everything replayed OK
why3 session info --stats lab2  # prints the list of unproved goals
```

Note Having all your proof goals checked does **not** necessarily mean that you will get a perfect grade on your homework. Indeed, you also have to make sure that your specifications are correct and complete. For example, it doesn't help if everything proves but you assumed precondition *false* everywhere.

3 Some tricks on using Why3

3.1 Proving methodology

The question you will find yourself asking most often while using Why3 is the following: *why the hell didn't this goal prove?* There are three possible answers to this question:

1. The goal you are attempting to prove is *false*, which means there is an error in either your implementation or your specification.

2. The goal you are attempting to prove is *unprovable* because you missed an invariant or because some part of your implementation is underspecified. In the latter case, this means that you are missing a `requires` in the current function or that you are making a call to a function whose behavior is underconstrained (some `ensures` are missing). You have to keep in mind that, when looking at a function call, the provers have no access to this function's body and only see its specification.

3. The goal you are attempting to prove is *true* but the provers are not smart enough to figure it out. You will need to annotate your code more.

Here is a list of what you should do when one of your goals does not check:

1. Launch the "`Auto 2`" strategy, which will split your goal automatically if needed. Look at what exact subgoals fail to be proved and what part of the code they correspond to (using the *Source* tab of the Why3 IDE).

2. If a subgoal G fails to be proved automatically, think of a proof of G yourself. Then, write down each argument or intermediate step in proving G as an assertion in the code and see what assertions fail to check.

3. If the subgoal that fails to be proved is small and simple enough, you can look at the *Task* tab of Why3 IDE to see the exact proof obligation that has been sent to the provers. A red flag indicating that it may be unprovable is when the conclusion features a variable that is almost unconstrained in the hypotheses.

4. If you manage to decompose your reasoning in many small steps using assertions, you should eventually reach a point where it becomes clear that either:

   (a) the main goal is indeed wrong: you should fix your implementation or your specification.

(b) the main goal is unprovable: you should add some `invariant`, `requires` or `ensures` annotations.

(c) the provers are missing some piece of subtle reasoning and you should help them by providing external lemmas. Note that we were able to solve every Why3 lab without running into this.[7]

In our experience though, when a goal does not check and it does not feature some crazy mathematical content, you are more likely to have missed something than the provers!

3.2 Other various trick

- Although the "`Auto 2`" strategy is powerful, it is quite slow and so you should not use it as a first attempt to prove a goal. After running it successfully, it is often useful to use the "`Clean`" button of the Why3 IDE to remove unsuccessful proof attempts.
- It is sometimes useful to write a type or loop invariant P that is redundant in the sense that it is implied by another invariant Q. Indeed, proving that Q is an invariant may be much easier for automated provers than proving the $P \to Q$ implication.

[7] with one exception in the last lab that we will discuss later.