Mini-Project 1
Hybrid Data Structures

15-414: Bug Catching: Automated Program Verification

Due Friday, February 23, 2024 (checkpoint)
Friday, March 1, 2024 (final)

You should **pick one of the following three alternative mini-projects**. You may, but are not required to, do this assignment with a partner.

> WhyML implementations of the data structures below that have been verified in Why3 may exist online. While you can examine Why3 reference materials, tutorials, and examples, **you may not read or use Why3 implementations of the data structures we ask you to code**. However, you may study or use implementations in other languages (with appropriate citations), and you can freely use anything in the Why3 standard library. In addition, the Toccata gallery of verified Why3 program may provide some insight.

The mini-projects have two due dates:

- Checkpoint at 23:59pm, Fri Feb 23, 2024 (50 pts)
- Final projects at 23:59pm, Fri Mar 1 2024 (100 pts)
  Up to 20 pts you lost on the checkpoint may be recovered on your final submission if you fix the problems that were noted. You are strongly encouraged to look at our feedback even if you received a full score.

We aim to have feedback on the checkpoint as early as possible. As a result, **no late days may be used for the checkpoint** in order for us to give timely feedback and for you to have enough time to incorporate the changes for the final submission.

The mini-projects must be submitted electronically on Gradescope. Please carefully read the policies on collaboration and credit on the course web pages at `http://www.cs.cmu.edu/~15414//assignments.html`.

If you are working with a partner, only one of the two of you needs to submit to each Gradescope assignment. Once you have uploaded a submission, you should select the option to add group members on the bottom of the screen, and add your partner to your submission. Your partner should then make sure that they, too, can see the submission.

> Our main piece of advice is this: **Elegance is not optional!** For writing verified code, this applies to both: the specification and the implementation.

The Code

In each problem, we provide some suggested module outlines, but your submitted modules may be different. For example, where we say '`let`' it may actually be '`let rec`', or '`function`', or '`predicate`', etc. You may also modify the order of the functions or provide auxiliary types and functions. You may also change the type definitions or types of the function, but in this case you should justify the change in your writeup.

The Writeup

The writeup should consist of the following sections:

1. **Executive Summary.** Which problem did you solve? Did you manage to write and verify all functions? If not, where did the code or verification fall short? Which were the key decisions you had to make? What ended up being the most difficult and the easiest parts? What did you find were the best provers for your problem? What did you learn from the effort?
2. **Code Walk.** Explain the relevant or nontrivial parts of the specification or code. Point out issues or alternatives, taken or abandoned. Quoting some code is helpful, but avoid "core dumps." Basically, put yourself into the shoes of a professor or TA wanting to understand your submission (and, incidentally, grade it).
3. **Recommendations.** What would you change in the assignment if we were going to reuse it again next year?

Depending on how much code is quoted, we expect the writeup to consist of about 3-5 pages in the lecture notes style. The writeup will be worth 20 pts (out of 100 pts from the final submission).

What To Hand In

You should hand in the following files on Gradescope:

- Submit the file `mp1.zip` to MP1 Checkpoint (Code) for the checkpoint and to MP1 Final (Code) for the final handin. As we are not providing starter files for the project, please make sure you submit both the code and completed session folder in the zip. Feel free to adjust our past Makefiles for your purposes, but you are not required to create one.
- Submit a PDF containing your final writeup to MP 1 Final (Written). There is no checkpoint for the written portion of the assignment. You may use the file `mp1-sol.tex` as a template and submit `mp1-sol.pdf`. You can generate this file by running make sol (assuming you have pdflatex in your system).

  Make sure your session directories and your PDF solution files are up to date before you create the handin file.

Using LaTeX

We prefer the writeup to be typeset in LaTeX, but as long as you hand in a readable PDF with your solutions it is not a requirement. We package the assignment source `mp1.tex` and a solution template `mp1-sol.tex` in the handout to get you started on this.

General Advice, Accumulated from previous years

- Keep your code simple and your contracts at a high level.
- Introduce predicates and logical functions so contracts remain concise and clearly organized.
- If things get too complicated, reconsider.
- Do not worry about optimizations.
- Organize your approach in stages and get each stage to work end-to-end before moving on to the next stage. Save each working stage, either to submit or to backtrack to. Each MP1 final submission problem has natural stages, either explicitly stated (as in 2.2 Hash Sets) or implicit. A fully working partial solution is easier and more pleasant to grade than a messy nonworking attempt.

There are also additional hints below, with each of the options.

1 Tries

A *trie* is an efficient data structure to represent sets or maps. You can read about tries, for example, in the Wikipedia article on Tries. In brief, the *word* which we use to look up data is presented in the form of a list of *characters*. At the root of the trie the first character in the word is an index by which we select a subtrie, from which we then proceed recursively with the remainder of the word. When the word is empty, we have found the location of the associated data in the trie.

1.1 Checkpoint: Bitwise Tries

Bitwise tries are the special case where a *character* is a bit, and a *word* is a *sequence* of bits. Every bitwise trie has at most two subtries. A bitwise trie can be used, for example, to represent a set of natural numbers by associating a number with a Boolean value. Because any data can be represented as a sequence of bits this is quite flexible.

Implement and verify sets of bit sequences as bitwise tries. You do not have to be concerned with low-level efficiency issues regarding space or time, but inserting or deleting a bit sequence of length m should have asymptotic complexity of $O(m)$. You may use standard libraries as you see fit.

Your implementation should define the types and functions below, and any auxiliary functions or predicates you wish to define. Each computational function should have suitable pre- and post-conditions to express their intended meaning.

```
module BitTrie

  type bitseq
  type trie

  let mem (x : bitseq) (t : trie) : bool

  let empty () : trie
  let insert (x : bitseq) (t : trie) : trie
  let delete (x : bitseq) (s : trie) : trie

  let union (s : trie) (t : trie) : trie
  let intersection (s : trie) (t : trie) : trie
  let difference (s : trie) (t : trie) : trie

end
```

1.2 Final: Sets of Words

Now imagine we want to store all the distinct words in the collected works of Shakespeare, or the Scrabble dictionary, with efficient means to determine membership. The input to this process should be a list of strings. The output should be a trie containing exactly the given strings.

Implement and verify tries of words (in the common meaning), restricted to letters 'A' through 'Z', if you wish. As usual, your functions should have suitable pre- and post-conditions.

For this problem, likely only `mem`, `empty` and `insert` will be relevant, but they should now be on words consisting of letters rather than bit sequences. This means you should have a simple auxiliary data structure at each node (for example, a list) containing all subtries.

```
module WordTrie

  type letter
  type word
  type trie

  let mem (x : word) (t : trie) : bool

  let empty () : trie
  let insert (x : word) (t : trie) : trie

end
```

Hint: The key here is the right approach to the map from letters to tries at each non-empty node. Choose the wrong approach and it quickly becomes very tricky; choose the right approach and it is surprisingly elegant. As often in programming, and especially in verified programming, try to think of the most abstract approach you could take.

2 Hash Sets

Hash tables are a common efficient data structure to map keys to values. As your mini-project, you may choose to implement and verify hash tables, where the values are just Booleans. The data structure is inherently mutable, but must be copied when it is resized.

2.1 Checkpoint: Static Hash Set with Separate Chaining

In your implementation, the type of key and the hash function should remain *abstract* so a client can suitably clone your module with a concrete type of key and concrete hash function. We suggest testing this functionality by cloning your module with integers and a simple hash function on integers. An example of such cloning is maps of integers in MapAppInt.

The hash table is represented as an array of buckets, which are (immutable) lists of keys. In many applications, these would be key/value pairs, but for simplicity we store only keys. For the checkpoint, the operations are to create an empty hash set with a initial size n (`create n = h`), to add a key to a hash set (`add h k`), and to determine whether a key is in the set (`mem h k = b`). In the implementation you should likely have corresponding operations on buckets.

The model of your data structure should be a finite set Fset. The postconditions should express the effect of your operation on the model.

```
module HashSet

  use int.Int
  use list.List
  use list.Mem
  use set.Fset
  use array.Array
  (* any use of additional standard libraries here *)

  (* abstract, supplied by client *)

  val eq (x : key) (y : key) : bool
  ensures { result <-> x = y }
  val function hash (x : key) : int
  ensures { result >= 0 }

  type bucket = list key

  type hash_set = { mutable data : array bucket ;
                    ghost mutable model : fset key }

  let create (n : int) : hash_set
  let add (h : hash_set) (k : key) : unit
  let find (h : hash_set) (k : key) : bool

end
```

2.2 Final: Hash Sets with Deletion and Resizing

We make three extension to the data structure from the checkpoint:

1. We add a `remove` operation

   ```
   let remove (h : hash_set) (k : key) : unit
   ```

2. We add a mutable `size` field to hash sets counting the number of elements in the table, maintained when adding or removing elements.

3. We add a `resize` function which should be automatically invoked when the number of elements would exceed the length of the underlying array. The size of the new table should be $2 * n + 1$, where n is the size of the old table.

Resizing is by far the trickiest function to write and verify because most keys will end up in different buckets. It is essential to define some logical predicates and functions to express the invariants concisely and intelligibly.
Hints:

- Use the model wherever you can, even for internal functions or in loop invariants (where possible). For example, instead of counting the number of elements in the table for use in your contracts, use `Fset.cardinal` on the model.

- In loops, data structure invariants are not enforced, only at function boundaries. That means you should be prepared to define a predicate explicitly stating the data structure invariants and use it in loop invariants. In general, in imperative code (for example, the resize function) you should keep an open mind about whether to use loops or recursion. A loop is less modular in a way, but it allows more variables and information about the state to remain visible and "in scope".

- In nested loops, it is likely you will need to include invariants of the outer loop in the inner loop

- For a data structure `h` with a mutable field `data` containing an array, you can make an assignment such as `h.data <- data'` only at the end of your function. Why3 will otherwise complain because the assignment introduces a nontrivial alias between the two arrays and changing one would affect the other. Since it does not track such aliasing it just disallows it for the sake of soundness.

3 Treaps with Split and Join

Treaps are an elegant data structure for binary search trees that are balanced with high probability and have a simple implementation. Moreover, they have a high degree of parallelism. The name *treap* is a hybrid between a *tree* and a *heap*, which is exactly what this data structure represents.

First, a treap is a binary search tree and therefore satisfies the usual *order invariant*: each nonempty node stores a key k, and all keys in the left subtree are strictly smaller than k and those in the right subtree strictly larger than k. In addition, a treap maintains a randomly chosen *priority* with each key. The treap then maintains the *heap invariant* regarding the priorities: the priority of a parent is always greater than the priorities of the children. The operations maintain these invariants.

All operations on treaps derive from the basic *split* and *join*. You can find a detailed description and even a functional implementation in three brief chapters from the lectures notes from 15-210 *Algorithms: Parallel and Sequential* by Umut Acar and Guy Blelloch.

3.1 Checkpoint: Parametric Binary Search Trees

In this problem we assume that instead of key/value pairs we simply store keys. It is straightforward to extend this and does not add much insight either into programming or verification.

Assume you have a type of binary search trees and functions implementing split and join. Specify their behavior and then implement and verify the remaining functions with respect to them. Unless you give simple implementations of split and join, you won't be able to run your code at this stage.

```
module ParametricBST

  type key = int
  type tree = Leaf | Node tree key tree

  val split (t : tree) (k : key) : (tree , bool , tree)
  val join (t1 : tree) (t2 : tree) : tree

  let empty () : tree
  let singleton (k : key) : tree
  let delete (t : tree) (k : key) : tree
  let insert (t : tree) (k : key) : tree
  let intersection (t1 : tree) (t2 : tree) : tree
  let difference (t1 : tree) (t2 : tree) : tree
  let union (t1 : tree) (t2 : tree) : tree

end
```

3.2 Final: Treaps

You should now implement the operations of `split` and `join`, keeping the remaining operations the same or close to the functions at the checkpoint. Due to limitations of WhyML's `clone` operation, you should expect to copy the code from the checkpoint.

In your implementation of split and join, you should extend the data structure implementation by associating priorities with every key, say, by calling a hash function that provides a uniform distribution of priorities. It may be beneficial to keep this hash function abstract.

```
module Treap

  type key = int
  type priority = int
  type tree = Leaf | Node tree key tree

  let split (t : tree) (k : key) : (tree , bool , tree)
  let join (t1 : tree) (t2 : tree) : tree

  (* rest as before *)

end
```

Hint: In a slight deviation from the description of Data Structure 3.2 (p.21) in the 15-210 lecture notes, we recommend computing the priority of each key k with a hash function $p(k)$, as in Definition 3.1 (p.19). Depending on your choice, you may wish to add the priority p to each node and change the type `tree` or keep it as shown above.