

Mini-Project 1

Hybrid Data Structures

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Thursday, March 11, 2021 (checkpoint)
23:59pm, Thursday, March 18, 2021 (final)
150 pts

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, Thu Mar 11, 2021 (50 pts)
- Final projects at 23:59pm, Thu Mar 18 2021 (100 pts)

Midsemester grades are due on Monday, March 22 at 4pm, and we aim to have feedback for you to gauge your performance in the course by that time. As a result, no late days may be used either on the checkpoint nor on the final portion of the project. However, because we were late in getting the project descriptions out, please reach out if you need additional time to complete the checkpoint (although note that this might delay feedback on your 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 sole 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.

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`.

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.

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 subtree, 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 subtrees. 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.

```
1 module BitTrie
2
3   type bitseq
4   type trie
5
6   let mem (x : bitseq) (t : trie) : bool
7
8   let empty () : trie
9   let insert (x : bitseq) (t : trie) : trie
10  let delete (x : bitseq) (s : trie) : trie
11
12  let union (s : trie) (t : trie) : trie
13  let intersection (s : trie) (t : trie) : trie
14  let difference (s : trie) (t : trie) : trie
15
16 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 subtrees.

```

1 module WordTrie
2
3   type letter
4   type word
5   type trie
6
7   let mem (x : word) (t : trie) : bool
8
9   let empty () : true
10  let insert (x : word) (t : trie) : trie
11
12 end

```

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.

```

1 module HashSet
2
3   use int.Int
4   use list.List
5   use list.Mem
6   use set.Fset
7   use array.Array
8   (* any use of additional standard libraries here *)
9
10  (* abstract, supplied by client *)
11  type key
12  val eq (x : key) (y : key) : bool
13  ensures { result <-> x = y }
14  val function hash (x : key) : int
15  ensures { result >= 0 }
16
17  type bucket = list key
18

```

```
19 type hash_set = { mutable data : array bucket ;
20                   ghost mutable model : fset key }
21
22 let create (n : int) : hash_set
23 let add (h : hash_set) (k : key) : unit
24 let find (h : hash_set) (k : key) : bool
25
26 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.

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.

```
1 module ParametricBST
2
3   type key = int
4   type tree = Leaf | Node tree key tree
5
6   val split (t : tree) (k : key) : (tree , bool , tree)
7   val join (t1 : tree) (t2 : tree) : tree
8
9   let empty () : tree
10  let singleton (k : key) : tree
11  let delete (t : tree) (k : key) : tree
12  let insert (t : tree) (k : key) : tree
13  let intersection (t1 : tree) (t2 : tree) : tree
14  let difference (t1 : tree) (t2 : tree) : tree
15  let union (t1 : tree) (t2 : tree) : tree
16
17 end
```

3.2 Final: Treaps

Extend the data structure implementation by adding priorities to every key. These should be generated from the key when creating singletons, say, with a simple linear congruential pseudo-random number generator.

```
1 module Treap
2
3   type key = int
4   type priority = int
5   type tree = Leaf | Node tree key tree
6
7   let split (t : tree) (k : key) : (tree , bool , tree)
8   let join (t1 : tree) (t2 : tree) : tree
9
10  (* rest as before *)
11
12 end
```