Range Sum Queries

In this project, we are interested in specifying and proving correct data structures that support efficient computation of the sum of the values over an arbitrary range of an array. Concretely, given an array of integers a, and given a range delimited by indices i (inclusive) and j (exclusive), we wish to compute the value: $\sum_{k=i}^{j-1} a[k]$. In the first part of the project, we consider a simple loop for computing the sum in linear time. In the second part, we introduce a cumulative sum array that allows answering arbitrary range queries in constant time. In the third part, we explore a tree data structure that supports modification of values from the underlying array a, with logarithmic time operations.

This project is to be carried out using the Why3 tool, in combination with automated provers (Alt-Ergo, CVC4 and Z3). You may use Coq for discharging particular proof obligations, although the project can be completed without it. To get started, you need to install the **latest version** of the Why3 and automated provers, and to download the skeleton file `sumrange.mlw`, which contains the template that you need to complete. Both the details of the installation procedure and the skeleton file may be found on the web page of the course.[1]

The project must be done individually—team work is not allowed. In order to obtain a grade for the project, you must send an e-mail to `claude.marche@inria.fr` and `arthur.charguéraud@inria.fr`, no later than **Monday, February 27th, 2017** at 22:00 UTC+1. This e-mail should be entitled "Project", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- The source file `sumrange.mlw`, completed with your specifications, implementations, and proofs.
- The content of the sub-directory `sumrange` generated by Why3. In particular, this directory should contain session files `why3session.mlw` and `why3shapes.gz`, and Coq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work. The contents of this report counts for at least half of your grade for the project.

The report must be written in French or English, and should typically consist of 3 to 6 pages. The structure should follow the sections and the questions of the present document. For each question, detail your approach, focusing in particular on the design choices that you made regarding the implementations and specifications. In particular, loop invariants and assertions that you add should be explained in your report: what they mean and how they help to complete the proof.

A typical answer to a question would be: *"For this function, I propose the following implementation: [give a pseudo-code]. The contract of this function is [give a copy-paste of the contract]. It captures the fact that [rephrase the contract in natural language]. To proof this code correct, I need to add extra annotations [give the loop invariants, etc.] capturing that [rephrase the annotations in english]. This invariant is initially true because [explain]. It is preserved at each iteration because [explain]. The post-condition then follows because [explain]."*

The reader of your report should be convinced at each step that the contracts are the right ones, and should be able to understand why your program is correct, e.g. why a loop invariant is initially true, why it is preserved, and why it suffices to establish the post-condition. It is legitimate to copy-paste parts of your Why3 code in the report, yet you should only copy the most relevant parts, not all of your code. In case you are not able to fully complete a definition or a proof, you should carefully describe which parts are missing and explain the problems that you faced.

In addition, your report should contain a conclusion, providing general feedback about your work: how easy or how hard was it, what were the major difficulties, was there any unexpected result, and any other information that you think are important to consider for the evaluation of the work you did.

[1] `https://www.lri.fr/~marche/MPRI-2-36-1/`

0 Specification of sums

To describe the sum of the values in the range $[i, j)$ of an array a, namely $\sum_{k=i}^{j-1} a[k]$, we introduce a logical function called `sum`, with the prototype shown below. Note that, by convention, the sum is 0 when $i \geq j$.

```
function sum (a:array int) (i j:int) : int
```

We wish to define this function recursively, as follows:

- if $i \geq j$, then: `sum` $a\ i\ j = 0$
- if $i < j$, then: `sum` $a\ i\ j =$ `sum` $a\ (i+1)\ j$

Your goal is to formalize this definition in Why3.

1. *In the module* `ArraySum` *of the skeleton file provided, complete the formulas of the two axioms* `sum_def_empty` *and* `sum_def_non_empty`*, which capture the above two equations.*
2. *State a lemma called* `sum_right`*, to capture the following property:*

   - *if* $i < j$*, then:* `sum` $a\ i\ j =$ `sum` $a\ i\ (j-1) + a[j-1]$.

   State this lemma under the form of a lemma function, and explain in your report the corresponding informal reasoning.

1 A Simple Loop to Compute Sums

The value of `sum a i j` may be computed using a simple for-loop as follows.

```
let query a i j =
  let s = ref 0 in
  for k = i to j-1 do
    s := !s + a[k];
  done;
  !s
```

This code appears in the module `Simple` from the skeleton file.

3. *First, consider the code without any post-condition. What assumptions and proofs are needed to ensure the safety of this function? Complete the proof using Why3 and explain in the report all the annotations that you need to add.*
4. *Annotate now the function with an appropriate post-condition. Then, add a loop invariant sufficiently strong to establish this post-condition, and complete the proof.*

2 Cumulative Sum Array

To be able to answer multiple queries more efficiently, we now consider the use of an auxiliary data structure called cumulative sum array, written s. The i-th cell of s, namely $s[i]$, contains the sum $\sum_{k=0}^{i-1} a[k]$. Note that $s[0] = 0$, and that s has length $n+1$ when a has length n.

Construction

5. *The predicate* `is_cumulative_array_for` *specifies the contents of the cumulative sum array in terms of the contents of the original array. Complete the definition of this predicate.*
6. *The function* `create` *takes as input an array, and builds the corresponding cumulative sum array. Complete the code of this function, propose suitable annotations for it, and prove it correct.*

Query

7. *The function* `query` *computes the sum over a given range. It takes as argument the bounds of the range and the cumulative sum array. It also takes as ghost parameter the original array. Complete the code of this function so that it runs in constant time. Then, prove your code correct. You will need another general property of the* `sum` *function; state this property before the code of* `query`.
8. *Prove this latter property of the* `sum` *function, possibly with the help of a lemma function. Explain in your report the corresponding informal reasoning.*

Update Consider a modification of the contents of original array a. Upon such a modification, the cumulative sum array s needs to be updated. Rather than rebuilding s from scratch, we wish to modify only the cells from s that need to be updated.

9. *The function* `update` *takes as argument a cumulative sum array s, an index i, and a value v that denotes the new contents for the cell $a[i]$. It also takes as ghost parameter the array a. Complete the code of the function so that it modifies as few cells from s as possible. Propose suitable annotations, and prove the code correct. You will need yet other general properties about the* `sum` *function; state these properties before the code of* `update`.
10. *Prove these latter properties of the* `sum` *function, possibly with the help of a lemma function. Explain in your report the corresponding informal reasoning.*

3 Cumulative Sum Tree

In this section, we consider a tree data structure that supports both queries and update operations in log-time complexity. Compared with cumulative sum array, the cumulative sum tree degrades the complexity of queries, from constant time to log-time; however, it vastly improves the complexity of updates, from linear time to log-time.

A cumulative sum tree is a binary tree whose nodes contain triples of the form (l, h, s), where l and h denote the bounds of a range, and where s corresponds to the sum of elements of the original array a in the range from l (inclusive) to h (exclusive). A cumulative sum tree satisfies the following structural properties:

- If a node (l, h, s) is a leaf of the tree, then $h = l + 1$.
- If a node (l, h, s) is not a leaf, then it has exactly two subtrees, whose nodes are of the form (l_1, h_1, s_1) and (l_2, h_2, s_2), and describe the two halves of the range from l to s. Formally, $l = l_1 < h_1 = \lfloor \frac{l+h}{2} \rfloor = l_2 < h_2 = h$.

Figure 1 shows an example cumulative sum tree.

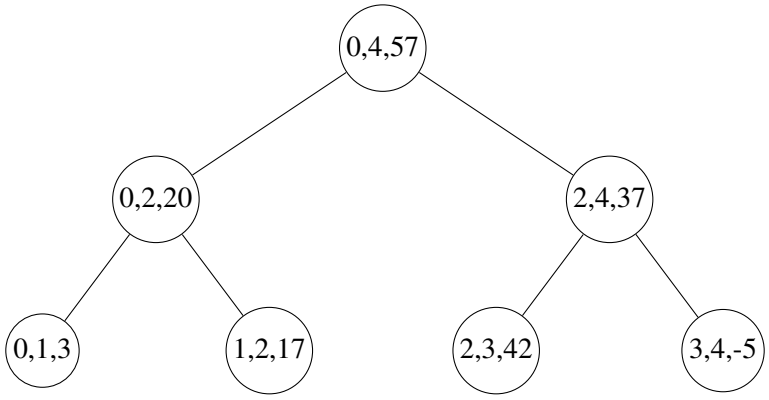

Figure 1: A cumulative sum tree for the array $[3; 17; 42; -5]$

Construction

11. *The predicate* `is_tree_for` *specifies the contents of the cumulative sum tree in terms of the contents of the original array and a given range. Complete the definition of this predicate.*
12. *The recursive function* `tree_of_array` *takes as input an array and a range, and builds the corresponding cumulative sum tree. Complete the code of this function, propose suitable annotations for it, and prove it correct.*
13. *The function* `create` *takes as input an array, and builds the corresponding cumulative sum tree. Complete the code of this function, by calling* `tree_of_array`*, then annotate your code and prove it correct.*

Query

14. *The recursive function* `query_aux` *computes the sum over a given range. It takes as argument the bounds of the range and the cumulative sum tree. It also takes as ghost parameter the original array. Complete the code of this function and prove your code correct.*
15. *Argue (in english) why your code has logarithmic time complexity.*
16. *The function* `query` *computes the sum over a given range. It takes as argument the bounds of the range and the cumulative sum tree. It also takes as ghost parameter the original array. Complete the code of the function* `query`*, by calling* `query_aux`*, then annotate your code and prove it correct.*

Update Consider a modification of the contents of original array a. Upon such a modification, only one path from the cumulative sum tree needs to be updated. In what follows, we are interested in defining a function called `update` that takes as input the cumulative sum tree associated with the original array, and produces the modified cumulative sum tree associated with the updated array, in logarithmic time. Note that we are not trying to modify the input tree in place, but rather to build an output tree that shares many branches with the input tree.

17. *The function* `update_aux` *takes as argument a cumulative sum tree t, the original array a (as a ghost parameter), an index i, and the value v that should be stored to $a[i]$. The function returns a pair (t', d) such that t' is the new cumulative sum tree, and d is equal to $v - a[i]$. Complete the code of the function so that it executes in logarithmic time. Propose suitable annotations, and prove the code correct. You will need another general property about the* `is_tree_for` *predicate; state this property before the code of* `update_aux`*.*
18. *Prove this latter property of the* `is_tree_for` *predicate, possibly with the help of a lemma function. Explain in your report the corresponding informal reasoning.*
19. *The main function* `update` *takes as argument a cumulative sum tree t, the original array a (as a ghost parameter), an index i, and a value v that should be stored in $a[i]$. It returns the cumulative sum tree associated with the updated array. The code of this function should call* `update_aux` *and perform the (ghost) write of the value v in the cell $a[i]$. Complete the code of* `update` *and prove it correct.*