Dynamic Programming

In this project, we are interested in specifying and proving correct programs that implement algorithms based on the dynamic programming paradigm. We focus on two independent case studies: Delannoy numbers and longest increasing subsequences.

This project is to be carried out using the Why3 tool (version 1.4.0), in combination with automated provers (Alt-Ergo 2.4.x, CVC4 1.7, and Z3 4.8.x). You can use other automatic provers or versions if you want, if they are freely available and recognized by Why3. You may use Coq for discharging particular proof obligations, although the project can be completed without it. The installation procedure may be found on the web page of the course at URL `https://marche.gitlabpages.inria.fr/lecture-deductive-verif/install.html`.

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 `Jean-Marie.Madiot@inria.fr`, no later than **Thursday, February 24th, 2022** at 22:00 UTC+1. This e-mail should be entitled "MPRI project 2-36-1", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- The proposed canvas file `dynamic.mlw` completed by yours.
- The content of the sub-directory `dynamic` generated by Why3. In particular, this directory should contain session files `why3session.xml` 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 added should be explained in your report: what they mean and how they help to complete the proof.

A typical answer to a question or step would be: *"For this function, I propose the following implementation: [give 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 prove 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 it was, what were the major difficulties, was there any unexpected result, and any other information that you think is important to consider for the evaluation of the work you did.

1 Delannoy numbers

The Delannoy number[1] $D(n, m)$ is the number of possible paths in the integer grid $\mathbb{N}^2$ that go from the origin $(0, 0)$ to the point (n, m) by using any combination of elementary steps that are vertical (adding $(0, 1)$), diagonal (adding $(1, 1)$) or horizontal (adding $(1, 0)$).

By a simple case analysis on the last step, we see that a path to (n, m) is either a path to $(n - 1, m)$ followed by an horizontal step, a path to $(n-1, m-1)$ followed by a diagonal step, or a path to $(n, m-1)$ followed by a vertical step. This leads to the following recursive definition:

- for any $n \geq 0$, $m \geq 0$, $D(n, 0) = D(0, m) = 1$
- for any $n \geq 1$ and $m \geq 1$, $D(n, m) = D(n - 1, m) + D(n - 1, m - 1) + D(n, m - 1)$

The proposed canvas file `dynamic.mlw` provides a module `Delannoy_spec` for the definition of these numbers.

1. *Fill in the definition of the let-function* `d` *of module* `Delannoy_spec`*, that defines Delannoy numbers. Explain what you need to specify to make your definition accepted by Why3.*
2. *The lemma-function* `d_is_positive` *is stating that all Delannoy numbers are non-negative. Complete its proof.*

1.1 Computing Delannoy numbers in a matrix

The module `Delannoy_matrix` of the canvas file proposes two versions of a program to compute Delannoy numbers using a matrix, using two nested loops. The code relies on a module `IntMatrix` that provides utility functions on matrices, in particular `(get m i j)` returns the element at row `i` and column `j` (numbered from 0) and `(set m i j v)` modifies in-place the matrix `m` by setting the value `v` at the corresponding place.

3. *Fill the holes in the logic annotations of the function* `delannoy_matrix` *so has to to prove the total correctness of this program.*
4. *Do the same with the alternative version* `delannoy_matrix_alt`*. Comment the differences with the first version, regarding the loop invariants in particular.*

1.2 Computing Delannoy numbers by dynamic programming

The module `Delannoy_dynamic` proposes an implementation using the dynamic programming approach, that avoids the use of a matrix to compute $D(n, m)$, but uses only an array of size $m + 1$ as temporary storage.

5. *Fill in the code of the program* `delannoy_dynamic`*, by providing an appropriate definition for the local variable* `y`*. Explain informally the way you chose this expression.*
6. *Complete the missing logic annotations so as to prove total correctness of this program.*

1.3 Experimental evaluation of complexities

Using the proposed `Makefile`, you can run some tests using the command '`make test_delannoy`'. The initial recursive definition of `d` is tested, together with the second matrix implementation and the dynamic programming implementation. For each test, its execution time is given inside square brackets. See the file `test_delannoy.ml` for the sequence of tests performed. Some tests may take too long time or exhaust your computer's memory, so you can comment them out.

7. *Comment on the results of the tests, regarding the time and space complexities of the three implementations: the recursive one, the matrix one and the dynamic one.*

[1] See for example `https://en.wikipedia.org/wiki/Delannoy_number`

2 Longest Increasing Subsequences

The second case study concerns Longest Increasing Subsequences (LIS for short), a classical example of application of dynamic programming.[2]

A subsequence of a sequence s is a subset of elements of s, not necessarily contiguous, ordered in the same order as they appear in s. The LIS problem is to find subsequences of a given sequence whose values are in (strictly) increasing order and of maximal length. Such a subsequence is not necessarily unique but, by construction, their length is unique. For example, the length of LIS for $[8; 4; 12; 1; 11; 6; 14; 15; 10; 5; 13; 3]$ is 4 and there are several maximal such subsequences, including $[8; 12; 14; 15]$ and $[4; 6; 10; 13]$.

A canvas for the specifications and programs related to LIS are given in several modules of file `dynamic.mlw`.

2.1 Specifications of LIS

The canvas for specifications are given in module `LISspec`. These specifications are based upon a logic type

```
type indexing = {
  size : int;
  indices : int -> int;
}
```

intended to represent subsequences as sequences of indices. For example the first subsequence of the example above is represented by

```
{ size = 4; indices = { 0 -> 0; 1 -> 2; 2 -> 6; 3 -> 7; _ -> _} }
```

Not all objects of type `indexing` represent valid subsequences, so we introduce a few predicates to specified the valid ones. `(is_subsequence s)` specifies that the size of `s` must be non-negative and that its indices should be in strictly increasing order. `(ends_at s i)` specifies that `s` must be non-empty and end at index `i`. `(ranges_in s a)` specifies that the indices of `s` must belongs to the valid range of array `a`. `(is_increasing_sub s a)` specifies that `s` is an increasing subsequence of array `a`. A final predicate introduced in this module is `(is_lis_ending_at s a i)` with an already complete definition. It specifies that `s` is an increasing subsequence of `a` that ends at index `i`, and is of maximal length among all such subsequences.

8. *Fill in the definition of the four predicates. You should test your specifications by proving the program* `test_specs` *of the same module. Do not hesitate to add more tests if you are not confident with your definitions.*

2.2 Recursive method to compute LIS

The idea is already underlying the definition of the predicate `is_lis_ending_at` above. Recursively, given an array a, we can compute the LIS ending at a given index i as follows: iterate over the indices j smaller than i and for each such j such that $a[j] < a[i]$, compute the LIS of sequences ending at j. Then the LIS for subsequences ending at i is the maximum of those, plus one. If there are no j at all such that $a[j] < a[i]$ then the answer is just 1. This method is implemented in the module `LISrecursive`.

The program `(lis_ending_at a i)` should compute the LIS of `a` ending at `i`. It returns a pair `(l,s)` of an integer and an indexing. `l` is the length of the LIS, and `s` is a ghost value intended to be a witness of such a maximal subsequence. This fact is specified with the already given post-condition.

[2] See for example `https://en.wikipedia.org/wiki/Longest_increasing_subsequence`

9. *Complete the holes in* `lis_ending_at` *to prove this program correct. The ghost code should be completed to provide an appropriate witness subsequence. Detail carefully in your report the process you followed to achieve the proof. You may or may not follow the hint below.*

Proving that the resulting subsequence is a maximal is quite tricky, and you may use the following hint: state a loop invariant of the form

```
forall t:indexing. ... -> t.size <= max
```

with the appropriate premise. Proving this invariant preserved is also tricky because of the universal quantification, and another hint is provided by the assertion given in the canvas, which proposes a form of reasoning by contradiction.

10. *Prove the program* `(all_lis_ending_at a)` *which returns an array* `b` *such that* `b[i]` *is the LIS of* `a` *ending at* `i`.
11. *Prove the program* `(lis a)` *which returns the length of LIS of* `a` *together with a ghost subsequence witnessing that LIS.*

2.3 LIS by dynamic programming

The module `LISdynamic` proposes a program `lis_dynamic` to compute the LIS by dynamic programming. As the previous program `all_lis_ending_at` before, it returns an array of LIS ending at a given index. In addition, it returns a ghost family of indexing f, so that $f(i)$ is a subsequence witnessing the LIS at index i. The postcondition for this function is already fixed.

12. *Prove the program* `lis_dynamic`.

There exists even better algorithms for computing LIS. For example at `https://en.wikipedia.org/wiki/Longest_increasing_subsequence#Efficient_algorithms` an algorithm similar to the one above is proposed, doing a binary search to find the maximal LIS ending at any index prior to index i. The following extra question is optional, not counting in the evaluation of this project.

13. *(Bonus question) Implement and prove a program based on the binary search optimization upon the dynamic programming approach.*

3 Conclusions

Don't forget to end your report with a conclusion that summarizes your achievements, explain the issues you couldn't solve if any, and comment about what you learned when doing this project.