Convex Hull

Paul Patault
Université Paris-Saclay

Abstract

The Jarvis algorithm is one way to solve the convex hull problem. I propose a formal specification of the problem and an implementation of this algorithm in the WhyML language with the aim to prove its correctness and termination.

Module `Point`

Question 1 : *Fill in the definition of the program function* `lowest_leftmost` *of module* `Point`*, that compute the index of the lowest-leftmost point of a set.*

The function `lowest_leftmost` searches for the "lowest" and "leftmost" point in a set, according to the y and x projection, respectively. It will be the initial point for the calculation of the convex hull. To identify it, I just have to iterate through every point and remember the lowest-leftmost seen. The contract of the function is

```
let lowest_leftmost (s:pt_set) : int
  requires { s.length > 0 }
  ensures  { is_index result s }
  ensures  { forall i: int. is_index i s -> is_lower_pt s[result] s[i] }
```

The post-conditions are given, I only add the pre-condition in order to allow the access to the first element of the array `s`. Then to prove this contact, I add some loop invariants : (1) the current result is indeed the lowest-leftmost point, (2) the result is a correct index of `s`, and (3) the memorized index corresponds to the lowest-leftmost point found up to the current loop tour.

```
invariant { forall j: int. 0 <= j < i -> is_lower_pt s[r] s[j] }
invariant { is_index r s }
invariant { s[r] = rp }
```

It is quite trivial that all three are initially true, and preserved by the function.

Module `CCW`

Question 2 : *Complete the definition of the predicate* `all_on_left(s,i,j)` *that states that all the points from the point set* `s`*, distinct from* `s[i]` *and* `s[j]`*, are located to the left of the segment* `s[i]`, `s[j]`.

Let the function `diff2` which I will need and use later in the project[1].

Using it to write the asked predicate gives the following

```
let function diff2 (k i j: int) : bool
= k <> i && k <> j

predicate all_on_left (s:pt_set) (i j:int)
= forall k: int.
    is_index i s ->
    is_index j s ->
    is_index k s ->
    diff2 k i j ->
    ccw s[i] s[j] s[k]
```

The predicate states as intended that all points of `s` are located on the left of the `s[i]-s[j]` segment. Indeed, I must have for all `k` in `[0..length s - 1]` different from `i` and `j`, a "ccw-triangle" `s[i]s[j]s[k]`.

Question 3 : *Complete the definition of the program function* `check_all_on_left` *that decides whether* `all_on_left(c, i, j)` *holds. Notice that it is on purpose that this function does not return a Boolean but instead raises the exception* `Exit` *when the* `all_on_left` *predicate does not hold.*

The function `check_all_on_left` can be written as

```
let check_all_on_left (s:pt_set) (i j:int) : unit
= for k = 0 to s.length - 1 do
    if diff2 k i j && not ccw s[i] s[j] s[k] then raise Exit
  done
```

with the following contract

[1] One could think that we also need to check that `s[k] <> s[i]` and `s[k] <> s[j]` but it is not required. The situation where two points of `s` are equal is never happening, thanks to the assumption `no_colinear_triple`. Indeed, given that every pairwise points are colinear, and that a point p_1 and its clone $p_{1'}$ with the same coordinates are colinear, a colinear triple can be formed by selecting any other point in the set, which is a contradiction.

```
let check_all_on_left (s:pt_set) (i j:int) : unit
  requires { is_index i s }
  requires { is_index j s }
  ensures  { all_on_left s i j }
  raises   { Exit -> not (all_on_left s i j) }
```

which expresses exactly what I want: given valid indices, we ensure that all points of `s` are on the left of the segment `s[i]s[j]`. Finally, I add the exceptional post-condition: if `Exit` is raised then it should not be true that all points of `s` are on the left of the segment `s[i]s[j]`.

To prove this specification correct and corresponding to the code, I only need next invariant to be attached to the `for` loop.

```
invariant {
    forall l: int. 0 <= l < k -> diff2 l i j -> ccw s[i] s[j] s[l] }
```

It is required and enough to prove the post-condition. Initially true because $\texttt{k} = 0$ (thus the range is null) and preserved since as long as `Exit` is not raised then the test of the `if` statement was evaluated to false.

Module `Path`

Remark: I move the signature `val nth(...)` to `val function nth(...)` in order to use it in the specifications.

Module `ConvexHull`

Question 4 : *Define a predicate* `is_convex_hull(s,p)` *that states that the path* `p` *is a convex hull of the point set* `s`. *This definition should make use of the predicate* `all_on_left`. *Justify carefully in your report why you pretend that your definition captures faithfully the notion of convex hull. For simplicity, to avoid degenerated cases, you may assume that considered sets of points have a minimal number of elements, say 2 or 3 (explain your choice in your report)*

First, I implement a program function of the given predicate distinct from the following signature

```
let distinct (p:path) : bool
  ensures { result <-> distinct p }
= ...
```

With the help of this auxiliary function I was able to define `wf_path`, another auxiliary function implementing the predicate `wf` defined as

```
predicate wf (p: path) (s: pt_set)
= distinct p /\
  forall i: int. is_valid_index i p -> is_index (nth p i) s
```

The proofs of the two last functions are quite simple. It is just necessary to add some invariants to the `for` loops which are copy-paste of the post-conditions bounded with respect to the current loop index.

In the end, the predicate `is_convex_hull` can be formulated as

```
predicate is_convex_hull (s:pt_set) (p:path)
= s.length >= __MIN_PTS_NUM__ /\
  wf p s /\
  (forall i: int. is_valid_index i p ->
    let j = if i+1 = p.size then 0 else i+1 in
    all_on_left s (nth p i) (nth p j))
```

As suggested in the subject, I declared a constant `__MIN_PTS_NUM__ = 2` to avoid degenerated cases. I choose 2 over 3 because only two points are required to be able to form one segment of the path and this was the only limitation implied by the set cardinal. I will now argue that the definition I choose for this predicate is what we want.

- First I ask for the set `s` to be at least composed of two elements.
- The second condition to be a convex hull of a set `s` is to be a subset of `s`. This is exactly what I ask with the predicate `wf p s`: all elements of the path `p` must be unique (set property) and must be a valid index in the set `s` (subset property).
- Finally, I want to specify the following idea "if X follows the path plotted by the points of `p`, then every point of `s` (except for the two points between which X moves) must be on X's left". Formally, for all pairs of consecutive points `p'` and `p"` in `p` (i.e $(\mathtt{p}_0,\mathtt{p}_1) \cdots (\mathtt{p}_{|p|-1},\mathtt{p}_0)$) the predicate `all_on_left s p' p"` must hold.

This definition seems to be complete when put in parallel to the description given in the subject and the usual definition of a convex hull.

Question 5 : *Complete the definition of the program function* `check_is_convex_hull` *that, given a set of points and a path, decides whether the path describes the convex hull of the point set. This function must return a Boolean value, and it should make use of the function* `check_all_on_left`*.*

The program function `check_is_convex_hull` can be defined as follows

```
let check_is_convex_hull (s:pt_set) (p:path) : bool
  requires { s.length >= __MIN_PTS_NUM__ }
  ensures  { result <-> is_convex_hull s p }
=
  if not wf_path p s then False else
  try
    for i = 0 to p.size - 1 do
      let j = plus1mod i p.size in
      check_all_on_left s (nth p i) (nth p j)
    done;
    True
  with Exit -> False end
```

As intended, the only pre-condition is about the minimum cardinal of the point set, while the post-condition is the equivalence between `result` and `is_convex_hull s p`. That is, there are two cases to consider, given by case analysis over the result of `wf_path p s`. In the first case, if it is false then we don't have a convex hull and we're done. In the opposite, we can assume the predicate `wf p s`. Then we only need to check that for every segment formed by a consecutive pair of points in the path, it is the right extremity of the point set — or more explicitly, we must verify that all points are on the left of this segment. If it is verified by all pairs of `p` then we have a proper convex hull, otherwise we do not.

To prove this function correct, the invariant required

```
invariant {
  forall j: int. 0 <= j < i ->
    all_on_left s (nth p j) (nth p (plus1mod j p.size)) }
```

is as usual a rewriting of the post-condition, bounded by the current index of the `for` loop.

Question 6 : *Prove the function* `jarvis_no_termination` *(a version ignoring termination), in the same time as developing the appropriate specification for largest.*

I added the following clauses to the contract of the `jarvis_no_termination` function.

```
let jarvis_no_termination (s:pt_set) : path
  requires { no_colinear_triple s }
  requires { exists i: int. unique_minimal_y s i }
  requires { length s >= __MIN_PTS_NUM__ }
  diverges
  ensures  { is_convex_hull s result }
= ...
```

The post-condition was given, also with the `diverges` option because we are not interested into proving termination for now. The idea of this algorithm is to gradually find the points of the path. Starting from the lowest-leftmost point and performing a windscreen wiper movement, we can find the next point of the path. Using the transitivity lemma, we know that we have to search for the point maximizing the given relation.

Pre-conditions clauses are induced by the subject. As above we make a statement about the size of the point set, and we add two more lemmas about this set for simplicity. The first one claim that there are no triplets of colinear points, and the second that there is exactly one lowest-leftmost point.

In order to prove this specification, I used the following invariants

```
invariant { is_index pivot s }                          (* 1 *)
invariant { is_index oldpivot s }                       (* 2 *)
invariant { p.size > 0 }                                (* 3 *)
invariant { nth p 0 <> pivot }                          (* 4 *)
invariant { nth p 0 = hd }                              (* 5 *)
invariant { nth p (p.size - 1) = pivot }                (* 6 *)
invariant { nth p (p.size - 2) = oldpivot }             (* 7 *)
invariant {
  forall i: int. 0 < i < p.size - 1 ->
    nth p i = pivot ->
    is_on_right s hd (nth p (i-1)) pivot }              (* 8 *)
invariant {
  forall i: int. 1 <= i < p.size ->
    all_on_left s (nth p (i-1)) (nth p i) }             (* 9 *)
invariant {
  forall i: int. 0 < i < p.size - 1 ->
    nth p i <> pivot }                                  (* 10 *)
invariant {
  forall i j: int.
    is_valid_index i p ->
    is_valid_index j p ->
    i < j ->
    nth p i <> nth p j }                                (* 11 *)
invariant { wf p s }                                    (* 12 *)
```

Invariants n°[1, 2] ensure the access to the corresponding elements of the point set `s`. Invariant n°3 acts as a reminder to the system that before the loop the path `p` was not empty, and will never be. The following four invariants n°[4..7] state how the path is arranged. The first element of the latter is different from `pivot` but equal to `hd`. The last two elements are, following the order in which they were added, `oldpivot` and `pivot`. Invariants n°[4,5] are preserved because the path never mutates and invariants

n°[6,7] are derived from the inner code of the `while` loop.

Invariants n°[8, 10] are used to show that we cannot find any point more than one time on the path. Let's have an in depth look at them.

A duplicated element is necessarily equal to the pivot when it has just been added. That is, I claim that if one can find two times the pivot in the path, then there is a duplicated element. The situation is the following: there is a path of length `p.size`, starting by the point `hd`, and ending with a loop linked by the pivot. By contradiction, we show that this situation cannot happen. Indeed, the point `hd` (labeled A in the following figure) would be on the right side of the segment BC where C is `s[pivot]`. In this situation the function `largest` should have returned the point `hd`. The property `distinct p` is thus true.

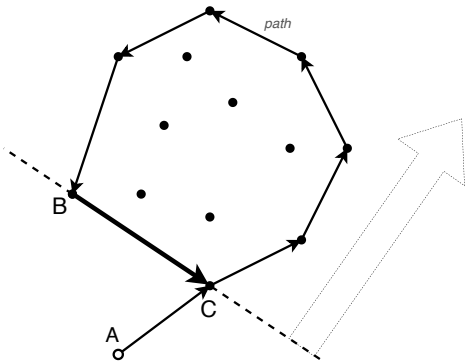

Figure 1: Intuition of the proof.
The point A is the "lowest-leftmost" point of the set, the big arrow shows the orientation of the left side of the last segment BC (in bold).

Invariant n°11 verifies the injectivity of the function `nth p`. Finally invariant n°12 is required to prove the post-condition. It checks that the current path is well-formed, in accordance with the definition of the `wf` predicate. It uses the 9th invariant and `distinct p` fact implied by invariants n°[8,10] to be proved.

Question 7 : *Prove correct the function* `largest`.

The function `largest` can be specified with

```
let function largest (s: pt_set) (pivot: int) (ghost base: pt): int
  requires { no_colinear_triple s }
  requires { length s >= __MIN_PTS_NUM__ }
  requires { is_index pivot s }
  requires {
    forall k : int. is_index k s -> k <> pivot -> s[k] <> base ->
      ccw base s[pivot] s[k] }
  ensures { is_index result s }
  ensures { result <> pivot }
  ensures { all_on_left s pivot result }
=
  let ref res = if pivot = 0 then 1 else 0 in
  for i = res+1 to s.length - 1 do
    if i <> pivot then
      if ccw s[pivot] s[i] s[res] then
        res <- i;
  done;
  res
```

The first two requirements are the same as the function `jarvis`. The next one is required to access to `s[pivot]` in the loop. The last one is an invariant of the `jarvis` loop and is useful to know that the pivot is the index in `s` of an element of the path. The first post-condition ensures that the `result` is in `s`, the second that it is different from `pivot` — otherwise we could not ensure the strict progress of the path —, the third is the main requested property of the function.

The specification is provable with the following invariants

```
invariant { is_index res s }                          (* 1 *)
invariant { is_index (i-1) s }                        (* 2 *)
invariant { res <> pivot }                            (* 3 *)
invariant {
  forall j: int. 0 <= j < i ->
    diff2 j pivot res ->
    ccw s[pivot] s[res] s[j] }                        (* 4 *)
```

Invariants n°[1,2] are needed to access to `s` and they are trivially true. Invariant n°3 proves the second to last post-conditions. Obviously it is initially true and preserved thanks to the presence of the `if` statements. Finally invariant n°4 is initially true and preserved by the same `if` conditions.

Question 8 : *Prove the termination of the jarvis function. Any lemma or ghost function used must be proved too, including the inverse function if you use it.*

The patch to apply to the function `jarvis_no_termination` to prove its termination is the following variant

```
variant { s.length - p.size }
```

I also need to use the two predicate given

```
predicate range (f: int -> int) (n: int) (m:int) =
    forall i: int. 0 <= i < n -> 0 <= f i < m

predicate injective (f: int -> int) (n: int) =
    forall i j: int. 0 <= i < j < n -> f i <> f j
```

The first is only about the bounds of a function `f`. I use it to restrict the length of the domain and co-domain of a function. The predicate `injective` is the usual definition for a function `f` over a bounded domain.

Finally the three main lemmas proving the variant of `jarvis` are

```
lemma wf_injection:
  forall p: path. forall s: pt_set.
  wf p s ->
    injective (nth p) p.size /\
    range (nth p) (p.size) (s.length)

use pigeon.Pigeonhole

lemma enough_drawer_for_my_socks:
  forall p: path. forall s: pt_set.
   range (nth p) (p.size) (s.length) /\
   injective (nth p) (p.size) ->
   s.length >= p.size
```

The first one "adds" the two facts in the invariants of `jarvis` as there is already the predicate `wf` for the path. The second lemma is taken from the standard library of Why3 and is used to prove the last but not least lemma, I named

`enough_drawer_for_my_socks`.

Given the conjunction I just proved with `wf_injection`, we deduce `s.length >= p.size`. It is the last requirement to prove the variant. Indeed, as I stated before the invariant `p.size > 0` provides a positive size of the path before the loop. In combination with the first post-condition of `Path.append` stating that the size of `p` grows, the termination problem is solved. To sum up, we have a strictly growing and bounded quantity: necessarily `p.size` will reach `s.length` at some point, if the `Exit`

exception has not been raised before. This concludes the proof of the termination of the jarvis algorithm.

Conclusion

It was a good project to deepen Why3. I really appreciated to discover in this card the proposed algorithm. I have encountered different problems, in particular when reasoning by the absurd was a way to prove properties. At the beginning it was not natural to think about "negative" invariants, but in the end I got the trick. I also wanted to emphasize the personal satisfaction of seeing IDE's lights turn green one after the other until everything is proven. Finally, I tried to write this report in English with the aim of improving myself. I hope it will be intelligible enough.