Convex Hulls

The goal of this project is to formally specify, implement, and formally prove correct an algorithm for computing the convex hull of a finite set of points in the plane. This project is to be carried out using the Why3 tool (version 0.82), in combination with automated provers (Alt-Ergo, CVC3, CVC4 and Z3). It is allowed to also use the Coq proof assistant, although the project can be completed without Coq.

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 17, 2013** at 19:00 UTC+1. This e-mail should be entitled "Convex Hulls", include your name in the body of the e-mail, and have as attachment an archive (`zip` or `tar.gz`) containing the following items:

- The source file `hull.mlw`, completed with your specifications, implementations, and proofs.
- The content of the sub-directory `hull` generated by Why3. In particular, this directory should contain the proof session file `why3session.mlw`, and your Coq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work.

In the PDF report, you need to detail your approach to each of the questions, focusing in particular on the design choices that you made regarding the specifications and the implementations. In case you are not able to fully complete a definition or a proof, you need to carefully describe which parts are missing and explain the problems that you faced. The report should be structured following the sections of the present document, and should consists of 5 to 10 pages. Note that the quality of the report counts at least for half of the final evaluation.

To get started, you need to install Why3 and the automated provers, and to download the file `hull.mlw` that contains preliminary definitions and that you need to complete. Both the installation procedure and the files may be found on the webpage of the course.[1]

1 Points in the plane

The file `hull.mlw` contains four modules: `Point`, `CCW`, `RandomAccessList`, and `ConvexHull`. Your answers are to be contributed in the module `ConvexHull`.

The module `Point` introduces a datatype `pt` to represent a point in the plane, and a type `pt_set` to represent a finite set of points. A point is represented as a pair of real coordinates. A set of point ("point set") is represented as an array of such pairs. In particular, the cardinality of a point set `s` can be written "`length s`" and the `i`-nth element is written as `s[i]`. Moreover, a predicate `is_index` is provided to state whether an integer is a valid index in a given point set, and a WhyML function `out_of_bounds` is provided to check whether an index is not valid in a piece of WhyML code[2].

As a starter, we are interested in computing extremal points, i.e., points whose x or y-coordinate is minimal or maximal in the point set. The module `Point` provides the predicate `is_lower_pt`, which induces a total order on the points of the plane: by definition, `is_lower_pt`$((x_1, y_1), (x_2, y_2))$ is true when either $y_1 < y_2$ is true or both $y_1 = y_2$ and $x_1 \geq x_2$ are true.

1. *Specify, implement, and prove a WhyML function `index_min_y` that, given a point set, computes the index of its lowest point (with respect to `is_lower_pt`).*

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

[2] Why3 forbids the use of `is_index` directly in program code.

2. *Specify, implement, and prove a WhyML function* `extremal_points` *that, given a point set, computes 4 indices using a single loop: the index of the lowest (and rightmost in case of tie), that of the highest (and leftmost in case of tie), that of the leftmost (and lowest in case of tie) and that of the rightmost (and highest in case of tie) points in the point set.*

2 Counterclockwise triangles

The module `CCW` from the file `hull.mlw` provides a predicate `ccw` which, given three points p_1, p_2 and p_3, captures the intuition that the triangle $p_1p_2p_3$ is oriented *counterclockwise* (and is not flat), as shown on Figure 1.

A thesis developed in the book *Axioms and Hulls*[3] by Donald Knuth is that all the properties needed to reason about convex hulls may be abstracted away into 5 facts about the predicate `ccw`. These facts are:

- (Cyclic symmetry) if $\texttt{ccw}(p_1, p_2, p_3)$ then $\texttt{ccw}(p_2, p_3, p_1)$.
- (Antisymmetry) if $\texttt{ccw}(p_1, p_2, p_3)$ then not $\texttt{ccw}(p_2, p_1, p_3)$.
- (Non-degeneracy) if p_1, p_2, p_3 are not colinear, then either $\texttt{ccw}(p_1, p_2, p_3)$ or $\texttt{ccw}(p_2, p_1, p_3)$.
- (Interiority) if $\texttt{ccw}(q, p_1, p_2)$, $\texttt{ccw}(q, p_2, p_3)$ and $\texttt{ccw}(q, p_3, p_1)$ then $\texttt{ccw}(p_1, p_2, p_3)$.
- (Transitivity) assuming $\texttt{ccw}(q_1, q_2, p_1)$, $\texttt{ccw}(q_1, q_2, p_2)$ and $\texttt{ccw}(q_1, q_2, p_3)$ (that is, all points p_1, p_2, p_3 are to the "left" of the segment q_1, q_2), if $\texttt{ccw}(q_2, p_1, p_2)$ and $\texttt{ccw}(q_2, p_2, p_3)$, then $\texttt{ccw}(q_2, p_1, p_3)$. In other words, given two points q_1 and q_2, the binary relation $x \; R \; y \equiv \texttt{ccw}(q_2, x, y)$ is transitive, when considering points located to the left of the segment q_1, q_2.

The definition of the predicate `ccw` is provided, as well as the statement and the proof of each of the five properties described above. These five properties allow us to abstract reasoning about angles.

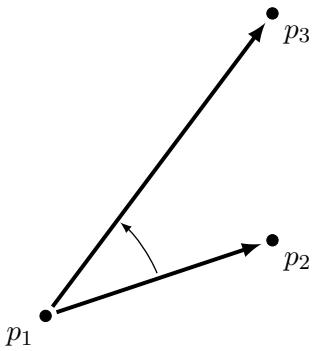

Figure 1: Counterclockwise triangle $p_1p_2p_3$

3 Points to the left of a segment

We are interested in the specification, implementation and verification of a program that computes whether all the points from a given point set are located to the left of the segment joining two particular points from this point set.

For simplicity, here and throughout the rest of the project, we assume that no three points are ever colinear. A predicate called `no_colinear_triple` is provided for this purpose. It should appear in the pre-condition of your functions.

3. *Define a predicate* $\texttt{all\_ccw}(s, i, j)$ *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* $s[i], s[j]$.

4. *Specify, implement and prove a WhyML function that decides whether* $\texttt{all\_ccw}(c, i, j)$ *holds or not. Your function must follow the pseudo-code provided below. In particular, it should raise an exception when the answer is no, and return unit when the answer is yes.*

```
check_all_ccw(s, i, j) :
  let a be s[i] and let b be s[j]
  for each k from 0 to size(s) − 1 do:
    if k ≠ i and k ≠ j and not ccw(a, b, s[k]) then
      raise exception Exit
  done
```

[3] Springer-Verlag, 1992, Lecture Notes in Computer Science, no 606. This is for reference only, there is no need to read any part of this book for the project.

4 Checking convex hulls

Intuitively, the convex hull of a point set is the smallest polygon that includes all the points from this set, as illustrated in Figure 2.

We choose to represent the convex hull of a point set as the sequence of the indices of the points that lie on the hull. To provide handy access to the i-th point on the hull, this sequence is represented as a *random-access list*. In the sequence, the points from the convex hull appear exactly once, in counterclockwise order. Remark: any of the points from the convex hull may be used as a starting point for the sequence.

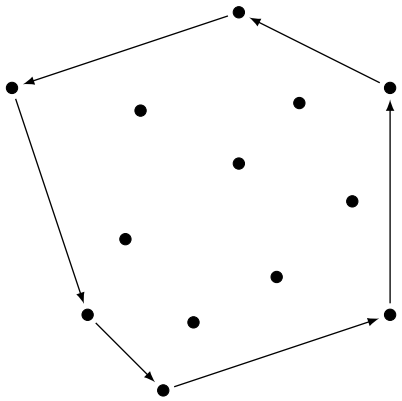

Figure 2: The convex Hull of a set of points

The module `RandomAccessList` is provided. The logic function `get` returns the element at a given index in such a list. The predicate `is_valid_index` states that an index is valid in a list. WhyML functions `create`, `size`, `nth` and `append` may be used to create an empty list, return the current size of the list, return the element at a given index in the list, and add a new element at the end of the list, respectively. To represent convex hulls, we thus use the type *path* defined as follows.

```
use import RandomAccessList
type path = random_access_list int
```

The goal of this section is to design a program that checks whether a given path indeed corresponds to the convex hull of a given point set. For simplicity, we only consider point sets containing two or more points.

5. *Define a predicate* `is_convex_hull`(s, p) *that states that path* p *is a convex hull of the point set* s*. This definition should make use of predicate* `all_ccw`*. Note that it should check that the path* p *contains 2 or more elements, and that the items in the path are all valid indices in the point set.*

6. *Specify, implement, and prove a WhyML function that, given a point set and a path, decides whether the path describes the convex hull of the point set. This function should return a Boolean value, and it should leverage on* `check_all_ccw`*, by making calls to this function.*

5 Computing convex hulls

The goal of this last section is to implement and prove an algorithm for computing a convex hull of a given point set. We focus on the *gift wrapping* algorithm, also known as *Jarvis march*.

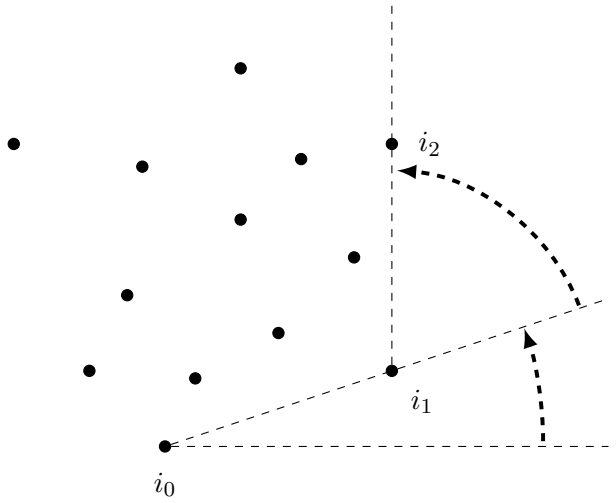

Figure 3: First steps of the gift wrapping algorithm

The idea of this algorithm is to "wrap" around the point set, starting from the point with minimal y-coordinate, as illustrated in Figure 3. At each step, we compute the next point that the wrapping will "touch". We find this point by enumerating the set of all the points from the point set in order to determine the one that is maximal for the relation $x\ R\ y \equiv$`ccw`(p, x, y), where p is the last point from the path in construction. (Recall that, by Knuth's fifth axiom, this relation is transitive.) The convex hull is completed when the next point that we find is equal to the point that we started with.

For simplicity, we rule out all degenerated cases. In addition to assuming that no 3 points are colinear, we also assume that there is a unique point with minimal y-coordinate. The predicate `unique_minimal_y`(s, i) asserts that $s[i]$ is the point with minimal y-coordinate in the point set s. Moreover, you are provided with a lemma, called `unique_minimal_pt_on_hull`, which asserts that if (x, y) is the unique point with minimal y-coordinate, then all the other points are located to the left of the segment joining $(x-1, y)$ to (x, y). Note that the point $(x-1, y)$ is given by `translate_left`(x, y). The lemma `unique_minimal_pt_on_hull` helps in the proof to justify the correctenss of the initial step.

The pseudo-code for the convex hull algorithm appears below. It consists of a main function called `jarvis` and an auxiliary function called `next_touching_point`, which computes the maximal point for the relation R defined above.

```
jarvis(s):
  let i_0 be index_min_y(s), p be the empty path, and pivot be i_0
  append index i_0 to p
  loop forever:
      let next be next_touching_point(s, pivot)
      if next is i_0 then raise exception Exit
      append index next to p
      set pivot to next
  on Exit return p

next_touching_point(s, pivot):
  let res be -1
  for each i from 0 to size(s) - 1 do
    if i ≠ pivot:
      if res < 0 or ccw(s[pivot], s[i], s[res]) then
        set res to i
  return res
```

Again, note that the program above never needs to make use of any kind of computation of angles: all computations are performed in terms of the predicate `ccw`. Finding the appropriate specification for the auxiliary function `next_touching_point` requires a good understanding of the proof of the main loop of the `jarvis` function. We therefore advise to approach the problem as follows:

7. *Implement in Why3 the pseudo-code provided above, and specify the function* `jarvis`*.*

8. *Prove the function* `jarvis` *(ignoring termination), in the same time as developing the appropriate specification for* `next_touching_point`*. Hint: introduce a ghost parameter to represent the "previous" point in the path in construction. Initially, you may assign this ghost variable to the point* `translate_left`$(s[i_0])$*.*

9. *Prove correct the function* `next_touching_point`*.*

6 Going further

The following questions may be solved independently.

10. *Prove the termination of the* `jarvis` *function. You will need to introduce additional invariants, and will very likely need to use the lemma* `map.MapInjection.injective_surjective` *from Why3's standard library.*[4]

11. *Implement and prove an optimized version of the Gift Wrapping algorithm: in a preliminary step, compute the four extremal points of the given point set (as done in question 2), then remove all the points located inside the quadrilateral joining these four extremal points, and compute the convex hull associated with the remaining points.*

[4] `http://why3.lri.fr/stdlib-0.82/map.why.html`