Formal Proof in Coq and Derivation of a Program in C++ to Compute Convex Hulls

Christophe Brun, Jean-François Dufourd, Nicolas Magaud

Formal Proof in Coq and Derivation of an Imperative Program to Compute Convex Hulls

Christophe Brun, Jean-François Dufourd, Nicolas Magaud

LSIIT, UMR 7005 CNRS - University of Strasbourg *

Abstract This article deals with a method to build programs in computational geometry from their specifications. It focuses on a case study namely computing incrementally the convex hull of a set of points in the plane using hypermaps. Our program to compute convex hulls is specified and proved correct using the Coq proof assistant. A concrete implementation in Ocaml is then automatically extracted and an efficient C++ program is derived (by hand) from the specification.

1 Introduction

This paper reports on a formal case study in computational geometry on a well-known problem: computing incrementally the convex hull of a finite set of planar points. This fits in a broader project aimed at surveying geometric modeling and computational geometry to improve the programming techniques and ensure the algorithms correctness. Our work is developed in the interactive *Coq* proof assistant [7, 1], which is based on a higher-order intuitionistic logical framework designed to formalize and prove mathematical properties. In addition, we use a generic topology-based approach to specify our algorithms using *hypermaps*.

A (two-dimensional) hypermap is a simple algebraic structure which allows us to model surface subdivisions (into vertices, edges and faces) and to distinguish between their topological and geometric aspects. For years, we have formally described hypermaps to prove topological properties of surfaces [11]. We define hypermaps inductively, which simplifies the construction of operations and proofs. The convex hull computation is a classical planar problem which is not only rich enough to highlight many interesting problems in topology and geometry, but also simple enough to reveal them easily and completely.

The geometric aspects we consider through an *embedding* of the hypermaps in the plane are particularly simple but fundamental in computational geometry. The embedding maps subdivision vertices into points, edges into line segments and faces are represented as polygonal frontiers. However, the question of the plane orientation is crucial. In our framework, it is captured using Knuth's axiom system which defines orientation according to the order in which a triple of points is enumerated in the plane (either clockwise or counter-clockwise) [17]. One of its advantages is that it allows us to isolate the required numerical tests and, in

* This work is partially supported by the french ANR project Galapagos (2008-2011).

a first step, to elude the difficult numerical accuracy problems. In fact, we do not address these issues in this work which instead focuses on the correctness of data structures and related operations. Real numbers are idealized using the axiom system provided in the Coq library [7].

A first experiment using structural induction to do the traversal of the hypermap is presented in [5]. At each step, it tests the hypermap elements in a random order but requires to use a copy of the initial hypermap to carry out the orientation tests, which can be costly in terms of memory use. In the present paper, we investigate how to have a program, and therefore a proof, which is closer to the usual implementation of the incremental algorithm to compute convex hulls, i.e. a program that proceeds by traversing the already-existing hull through its edges. Thanks to higher-level operations [12], handling hypermaps also becomes simpler and more accessible to non-specialists of both inductive specifications and combinatorial oriented maps.

2 Related Work

Convex Hull and Subdivisions The convex hull has several different definitions and several construction methods, such as the incremental algorithm, Jarvis' march, Graham's scan or the divide and conquer approach. It is a fairly simple subdivision of the plane (roughly by a polygon) which still combines topological and geometric aspects. General subdivisions of varieties − into edges, vertices, faces, etc. − have been studied extensively (see [13, 9]) and combinatorially described in [8, 18]. Among these descriptions of subdivisions, *hypermaps* are one of the most general ones, homogeneous in all the dimensions, and easy to formalize algebraically. We shall work with this notion in dimension two but constrain it, at a later time, into *combinatorial oriented maps*, in short *maps*, which is exactly what is required for our study of convex hulls. Maps led to several implementations in geometric modelling, e.g. in the libraries CGAL [14] and CGoGN [16]. We need to embed hypermaps (and maps) into the oriented Euclidian plane to be able to formalize what a convex hull is. As usual, our embedding consists in mapping vertices into points of the plane, all other objects being obtained by linearization. We thus rely on the axiom system for geometric computation and orientation proposed by Knuth [17]. This axiom system, based on properties of triples of points in the plane, allows us to isolate numerical accuracy issues in computations and let us focus on the logical tests required in the algorithms. This approach is well-suited to carry out formal proofs of correctness of the considered algorithms.

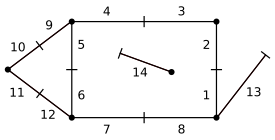
Formal Proofs in Computational Geometry Formal proofs in computational geometry, especially focusing on convex hull algorithms have been carried out in Coq [20] and in Isabelle/HOL [19]. Both use Knuth's axiom system but none of the above-mentioned works relies on any topological structure. However, hypermaps have been used highly successfully to model planar subdivisions in the formalization and proof of the four-color theorem in Coq by Gonthier et al.

[15]. Their specification approach as well as the proof techniques (using reflection in Coq) are fairly different from the methodology we follow in this paper. The formal proofs we carry out in Coq follow the tradition of algebraic specifications with constructors as studied in [3]. At Strasbourg University, the library specifying hypermaps, onto which our present work is built, was successfully used to prove some significant results in topology [11]. It was also used to carry out a formal proof of correctness of a functional algorithm of image segmentation and to develop a time-optimal C-program [10]. Recent and on-going works include specifying and proving correct simple algorithms to compute convex hulls [5] and Delaunay triangulations [12].

3 Hypermaps

We mathematically define hypermaps and cells and then specify them in Coq.

Definition 1 (Hypermap). *(1) A (two-dimensional)* hypermap *is an algebraic structure $M = (D, \alpha_0, \alpha_1)$, where D is a finite set, the elements of which are called* darts*, and where α_0, α_1 are permutations on D.*
(2) When α_0 is an involution on D (i.e. $\forall x \in D$, $\alpha_0(\alpha_0(x)) = x$), then M is called a combinatorial oriented map *− in short a* map*.*



D	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a0	2	1	4	3	6	5	8	7	10	9	12	11	13	14
a1	8	3	2	9	4	12	6	13	5	11	10	7	1	14

Figure 1. An example of hypermap (actually a combinatorial oriented map)

In this framework, 0 and 1 symbolize the two *dimensions*. The topological cells of a hypermap can be combinatorially defined through the classical notion of orbit. Let f be a permutation of darts of a hypermap. The *orbit* of x for f, noted $\langle f \rangle(x)$, is the set of darts accessible from x by iteration of f. The orbits for α_0, or 0-*orbits*, are the *edges* of the hypermap and those of α_1, or 1-*orbits*, are its *vertices*. Note that in a map, each edge is composed of at most two darts. *Faces* are the orbits for $\phi = \alpha_1^{-1} \circ \alpha_0^{-1}$. *Connected components* are also defined as usual: the hypermap is considered as a 2-graph equipped with α_0 and α_1, viewed as two binary relations.

An *embedding* of a hypermap is a drawing on a surface where darts are represented by oriented half-segments of curve with the following conventions: (1) the half-segments of all the darts of a given vertex (resp. edge) have the same origin (resp. extremity); (2) the oriented half-segments of an edge, a vertex or a face are traversed in counter-clockwise order when one follows α_0, α_1 or ϕ, respectively; (3) the (open) half-segments do not have any intersection. When

the hypermap can be embedded in a plane it is said to be *planar*. In this case, every face which encloses a bounded (resp. unbounded) region on its left is called *internal* (resp. *external*). Finally, the *degree* of an orbit is its number of darts. For more details on embedding and planarity, the reader is refered to [11]. For example, in Fig. 1, a hypermap (in fact a map) $M = (D, \alpha_0, \alpha_1)$ is embedded in the plane with straight half-edges. It has 14 darts, 8 edges (symbolized by small *strokes*), 6 vertices (symbolized by *bullets*), 4 faces and 2 components. For instance, $\langle\alpha_0\rangle(1) = \{1, 2\}$ is the edge of dart 1 and $\langle\alpha_1\rangle(1) = \{1, 8, 13\}$ its vertex. We have $\phi(1) = 3$, $\phi(3) = 5$, $\phi(5) = 7$ and $\phi(7) = 1$. The (internal) face of 1 is $\langle\phi\rangle(1) = \{1, 3, 5, 7\}$ and the (external) face of 2 is $\langle\phi\rangle(2) = \{2, 13, 8, 12, 10, 4\}$.

Specification of Free Hypermaps In our Coq specification, darts, of type `dart`, are natural numbers and dimensions, of type `dim`, are `zero` and `one`. The hypermaps are first approached by a general notion of *free hypermap*, thanks to a free algebra of terms of inductive type `fmap` with 3 constructors, `V`, `I` and `L`, respectively for the *empty* (or *void*) hypermap, the *insertion* of a dart, and the *linking* of two darts:

```
Inductive fmap : Set :=
  V : fmap
| I : fmap -> dart -> point -> fmap
| L : fmap -> dim -> dart -> dart -> fmap.
```

For example, a part of the hypermap M of Fig.1, consisting of darts $1, 2, 3$ and 8 is (functionally) represented in Coq by the following term : `(L (L (L (I (I (I (I V 1 p1) 2 p2) 3 p3) 8 p8) zero 1 2) one 2 3) one 1 8)`. When darts are inserted into a free hypermap, they come together with an embedding `point` which is a couple of real numbers. That is enough to embed darts on straight half-segments in the plane. As the reader may see from Fig. 1, some geometrical consistency properties must be enforced. For instance, the points `p2` and `p3` respectively associated with darts 2 and 3 must be equal. Coq also generates an induction principle on free hypermaps. This principle allows us to prove properties or build functions by induction on the hypermaps.

Specification of Hypermaps Preconditions written as predicates are introduced for operators `I` and `L` to avoid meaningless free maps. The precondition for `I` states that the dart to be inserted in the free hypermap `m` must be different from `nil` (= 0) and from any dart of `m`. The precondition for `L` expresses that the darts `x` and `y` we want to link together at dimension `k` in hypermap `m` are actually already inserted, that `x` has no `k`-successor and that `y` has no `k`-predecessor w.r.t α_k, denoted by `cA m k` in Coq. These preconditions allow us to define an invariant `inv_hmap` for the hypermap subclass of free maps. It is systematically used in conjunction with `fmap`. The predicate `exd m x` expresses that the dart `x` exists in the hypermap `m`. Operations `cA`, `cA_1`, `cF` and `cF_1` simulate the behavior of the functions α_k, α_k^{-1}, ϕ and ϕ^{-1}. For example, if the

hypermap in Fig. 1 is `m`, `exd m 4`, `cA m one 4 = 9`, `cA_1 m one 7 = 12`, `cA m zero 13 = 13`, `cA_1 m zero 13 = 13`. In addition, when the input dart does not belong to the hypermap, we have `cA m one 15 = nil` and `cA_1 m one 15 = nil`. Furthermore, we have `cF m 1 = 3`, `cF_1 m 1 = 7`.

Merge and Split Functions Two high-level operations on hypermaps will make easier the writing of our further algorithms. For any dimension `k` $= 0, 1$, the operation named `Merge` merges two k-orbits. To do it, we must choose a dart `x` in the first one and a dart `y` in the second one, such that the k-successor of `x` will be `y` in the newly-formed orbit (Fig. 2). The precondition for `Merge` requires that `x` and `y` do not lie in the same k-orbit.

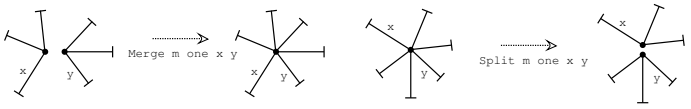


Figure 2. (a). Merging 2 k-orbits (b). Splitting an orbit

The operation named `Split` splits a k-orbit into two pieces with respect to two darts `x` and `y`. The precondition for `Split` is that `x` and `y` must be different, but belong to the same k-orbit (Fig. 2). We easily prove that, when they satisfy their preconditions, these operations preserve the hypermap invariant `inv_hmap`. Further topological properties may be considered while proving correct our convex hull algorithm. In addition, invariants dealing with geometry must be defined.

4 Convex Hull and Geometric Setting

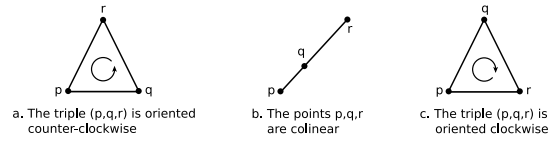


Figure 3. The orientation predicate

Computing the convex hull of a set of points requires to determine the orientation of three points of the plane (whose coordinates are real numbers). This is necessary to determine whether a point lies inside or outside a given polygon.

We use the orientation predicate $ccw(p, q, r)$ noted `ccw p q r` in Coq. It can be defined by $ccw(p, q, r) = det(p, q, r) > 0$ using the determinant:

$$det(p, q, r) = \begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{vmatrix}$$

which also allows us to define a predicate `align p q r` stating that the three points are collinear. In our development, it is established that this implementation is a *model* of Knuth's axioms about orientation [17] which are sufficient to prove all the key geometric properties we need. This approach allows us to deal cleverly with orientation from a logic point of view instead of considering numbers and computations (see [4] for details).

Convex Hull Definition In this work, we choose a definition of the convex hull well-suited for our topological hypermap model, for using Knuth's orientation predicate *ccw* and for the incremental algorithm we will study [9, 13]. As our main interest lies in using hypermaps to formalize a convex hull computation algorithm, we assume that points are in *general position*, i.e. *no two points coincide* and *no three ones are collinear*. Other authors, e.g. in [20], study how to relax this restriction using the pertubation method.

Let P be a set of points in the plane.

Definition 2 (Convex hull). *The* convex hull *of P is the polygon T whose vertices t_i, numbered in a counterclockwise order traversal for $i = 1, \ldots, n$ with $n + 1 = 1$, are points of P such that, for each edge $[t_i t_{i+1}]$ of T and for each point p of P different from t_i and t_{i+1}, ccw(t_i,t_{i+1},p) holds.*

In other words, every point p of P different from t_i and t_{i+1} lies on the left of the oriented line generated by $\overrightarrow{t_i t_{i+1}}$ (Fig. 4).

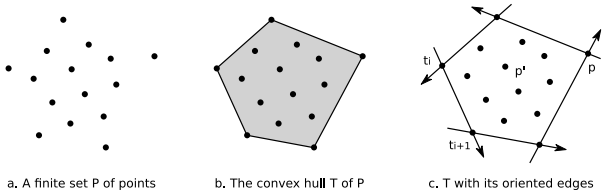


Figure 4. Characterizing a convex hull

Incremental Algorithm The convex hull of P is built step by step. Each step takes as input the current hull T (the one built with all the already-processed

points), a new point p of P and returns a new convex hull T'. Then, either p lies *inside* T and the algorithm moves on to the next step, or it lies *outside* of T and the algorithm will have to remove some edges $[t_i t_{i+1}]$ of T which are *visible* from p – $\neg\ ccw(t_i, t_{i+1}, p)$ holds –. To build T' it also creates two new edges $[t_l p]$ and $[p t_r]$ connecting p to the *leftmost vertex* t_l and to the *rightmost vertex* t_r (Fig. 5). This corresponds to the usual incremental algorithm (e.g. in [9]). Since we assume that points are in general position, p can never be *on* the already-built polygon, i.e. be equal to a previously-added point or lie on an existing edge.

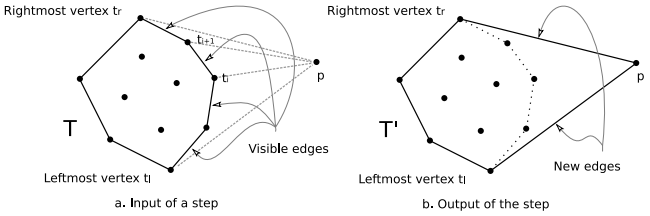


Figure 5. Computing a new convex hull T' from a convex polygon T and a point p

5 Program Specification

In our specification with hypermaps, the convex hull is an internal face of the current hypermap. To identify it, we first define a new type `mapdart` of pairs (the type cartesian product is denoted by `*`), which consists of a current hypermap and a dart of the current convex hull. This definition is mandatory to be able to perform orientation tests in the plane between the inserted point and the existing edges of the current convex hull.

```
Definition mapdart := fmap * dart.
```

The program, whose inputs must satisfy some strong preconditions, is structured with four functions `CH2`, `CHID`, `CHI` and `CH` presented hereafter using a *bottom-up* point of view.

Preconditions The initial hypermap has to satisfy several *preconditions* which are merged into the following definition:

```
Definition prec_CH (m:fmap) : Prop :=
 inv_hmap m /\ linkless m /\ is_neq_point m /\ is_noalign m.
```

Of course, the free hypermap `m` must verify the invariant `inv_hmap` (Section 3), which ensures we have coherent hypermaps. The predicate `linkless` ensures it does not contain any links between darts. The predicate `is_neq_point`

states that all darts in the initial hypermap must have different embeddings. The predicate `is_noalign` ensures three darts with different embeddings can not be embedded into three collinear points. Note that the starting point of our algorithm is actually a set of points, but we model it as a linkless hypermap.

Initialisation Function `CH2` It builds a map from two darts as described in Fig. 6. It introduces two new darts, `max+1` and `max+2`, which are linked to the two input darts, `x1` and `x2`, using the function `Merge`. Note that `max` is a counter, used to generate fresh new darts to be inserted in the map.

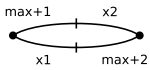


Figure 6. A convex hull of two darts built by the CH2 function (edges are slightly curved to make them visible)

Leftmost and Rightmost Darts We define two functions `search_left` and `search_right` to search resp. for the leftmost and the rightmost darts in a face (`m`,`d`) with respect to `p` performing a traversal of the whole set of darts recursively. They both proceed by general recursion, which is achieved in Coq using the keyword `Function`. A strictly *decreasing measure* has to be provided to Coq to prove the termination. The keyword `measure` actually expects two arguments, a function to compute the measure and its argument. Informally, these two functions use an integer `i` which is incremented by 1 at each recursive call (starting from 0). If the degree of the face (that is to say its number of darts), defined in Coq by the predicate `degreef`, is less or equal to `i`, then all darts of the face have been checked and there is no leftmost (resp. righmost) dart. Otherwise, it checks whether the `i`-th successor in the face containing `d` corresponds to the leftmost (resp. rightmost) dart. Functions `search_left` and `search_right` are similar and we only present the code of the first one.

```
Function search_left (m:fmap)(d:dart)(p:point)(i:nat)
  {measure (fun n:nat => (degreef m d) - n) i} :=
  if (le_lt_dec (degreef m d) i) then nil
  else let di := Iter (cF m) i d in
    if (left_dart_dec m di p) then di
    else search_left m d p (i+1).
```

Insertion Function `CHID` It computes the convex hull of the current convex polygon denoted by `md = (m,d)` and a new dart `x`. It determines the leftmost dart `l` and the rightmost one `r` in `m` with respect to `x` and its embedding `p`. If `l` is `nil`, then it is proven that `r` is also `nil` and this means `p` lies inside the convex hull. In this case, it simply inserts `x` into the free hypermap. Otherwise, it proceeds in six steps illustrated at Fig. 7. (1) It splits the leftmost dart `l` from

its 0-successor `l0 := cA m zero l`. (2) If the dart `l0` is different from `r`, it splits `r` from its 0-predecessor `r_0 := cA_1 m zero r`. (3) It inserts `x` and the *fresh* dart `max` both embedded into `p`. (4) It merges `max` and `x` at dimension `one`. (5) It merges `l` and `max` at dimension `zero` to create a new edge. (6) It merges `x` and `r` at dimension `zero` to close the convex hull.

```
Definition CHID (md:mapdart)(x:dart)(p:point)(max:dart):mapdart :=
  let m := fst md in let d := snd md in
  let l := search_left m d p 0 in
  if (eq_dart_dec l nil) then (I m x p, d)
  else let r := search_right m l p 0 in
     let l0 := cA m zero l in let r_0 := cA_1 m zero r in
     let m1 := Split m zero l l0 in                          (*1*)
     let m2 := if (eq_dart_dec l0 r) then m1                 (*2*)
               else Split m1 zero r_0 r in
     let m3 := (I (I m2 x p) max p) in                       (*3*)
     let m4 := Merge m3 one max x in                         (*4*)
     let m5 := Merge m4 zero l max in                        (*5*)
     let m6 := Merge m5 zero x r in (m6, x).                 (*6*)
```

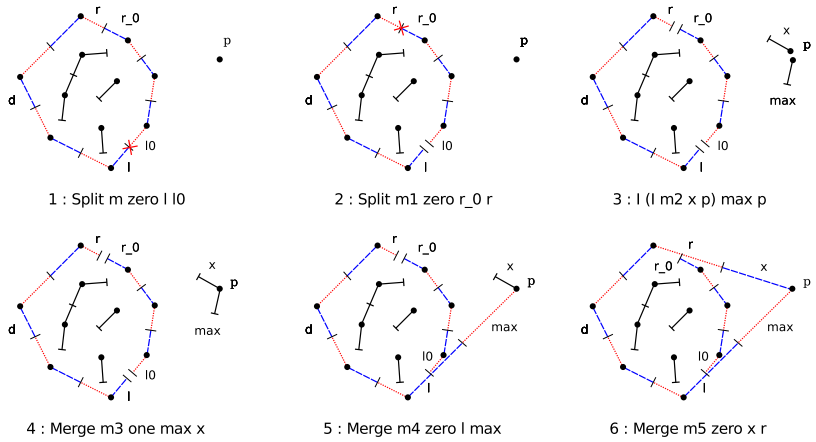


Figure 7. The six steps of the execution of the insertion function `CHID`

Recursion Function `CHI` It proceeds by recursion and handles the darts of `m` one by one. At each step, it builds a new convex hull using the insertion function `CHID`. The free hypermap `m` corresponds to the set of remaining initial points and `md` is the already constructed convex hull.

```
Fixpoint CHI (m:fmap)(md:mapdart)(max:dart) {struct m} : mapdart :=
  match m with
```

```
  | I m0 x p => (CHI m0 (CHID md x p max) (max+1))
  | _ => md
  end.
```

Main Function `CH` It computes the convex hull of a free hypermap `m` representing the finite set of darts supporting the initial points. If `m` has a single dart `x`, it returns a couple (`m`, `x`). If it has at least 2 darts, `CH` builds a first convex polygon with two of these darts using `CH2` (Fig. 6) and then calls `CHI`. To have an exhaustive pattern-matching, we have a default case which actually never happens because of the preconditions and it simply returns a couple formed by the initial set of darts and the dart `nil`.

```
Definition CH (m:fmap) : mapdart :=
  match m with
  | I V x p => (m,x)
  | I (I m0 x1 p1) x2 p2 =>
    CHI m0 (CH2 x1 p1 x2 p2 (max_dart m))
      ((max_dart m)+3)
  | _ => (m,nil)
  end.
```

This definition of `CH` is similar to the one used in the usual incremental algorithm [9, 13]. However, it is important to note that chains of darts which are inside the polygon remain in the computed hypermap. These useless chains of darts could be removed in a function built on top of `CH` or each time `CHID` is executed.

6 Topological Properties

The structure of the proofs, especially for those related to the insertion function `CHID`, closely follows the structure of the program. Therefore, we successively prove each of the following properties for the six successive hypermaps computed by the algorithm and presented in the previous section.

Hypermap Invariant and Initial Darts The first important theorem we prove is a technical one. Indeed, we have to show that the invariant `inv_hmap` holds throughout the program, from the initial hypermap `m` to the final hypermap `(CH m)`. This states that a dart can not be inserted twice in a hypermap and that darts must belong to the hypermap before being linked together (see Section 3).

```
Theorem inv_hmap_CH : forall (m:fmap),
  prec_CH m -> inv_hmap (CH m).
```

The proof proceeds in 6 steps according to the structure of the program. Each step is fairly easy to prove thanks to the invariant-preserving properties of `I`, `Merge` and `Split`. In the same way, we prove that all the initial darts are preserved, which also entails that all the initial points are preserved.

Involutions without Fixpoint One of the properties of the convex hull is to be a polygon, which means that each edge and vertex is of degree 2. This property is expressed by the predicate `inv_gmap` which states that for each dart `x` of a hypermap `m`, and for each dimension `k`, α_k is an involution.

```
Definition inv_gmap (m:fmap) : Prop :=
  forall (k:dim)(x:dart), exd m x -> cA m k (cA m k x) = x.
```

The proof that the convex hull satisfies this property is given by:

```
Theorem inv_gmap_CH : forall (m:fmap),
  prec_CH m -> inv_gmap (CH m).
```

Another property of the polygon representing the convex hull is that it does not have any fixpoint. This property is ensured by the predicate `inv_poly`. Thus, for each dart `x` belonging to the same connected component as `d` in the hypermap `m` (i.e. verifying `eqc m d x`), whatever the dimension `k`, none of the `k`-orbits admits a fixpoint.

```
Definition inv_poly (m:fmap)(d:dart) : Prop :=
  forall (k:dim)(x:dart), eqc m d x -> x <> cA m k x.
```

This definition uses the notion of connected components which is more intuitive than the notion of faces. However, note that it is established that if darts `x` and `y` belong to the same face, then they both belong to the same connected component. Then we prove the following theorem:

```
Theorem inv_poly_CH : forall (m:fmap),
  prec_CH m -> inv_poly (CH m).
```

Planarity We now verify the polygon we build is planar. The definition of the planarity property `planar` from Euler's formula can be found in the proof development and is omitted here.

```
Theorem planar_CH : forall (m:fmap),
  prec_CH m -> planar (CH m).
```

The proof of this theorem uses several planarity criteria [11] for `Merge` and `Split` and is fairly straightforward.

7 Geometric Properties

The geometric properties we prove ensure that darts are embedded in the plane in a coherent manner and that the program builds a polygon that is actually convex (see Definition 2).

Embedding, Distinct and Collinear Points We first prove that darts are embedded correctly with respect to their links: darts `x` and `y` which belong to the same vertex (`eqv m x y`) must have the same embedding whereas darts belonging to the same edge (`eqe m x y`) must have different embeddings. This is summarized in the following definition `is_well_emb`:

```
Definition is_well_emb (m:fmap) : Prop :=
  forall (x y : dart), exd m x -> exd m y -> x <> y ->
  let px := fpoint m x in let py := fpoint m y in
  (eqv m x y -> px = py) /\ (eqe m x y -> px <> py).
```

Then we prove the following theorem:

```
Theorem iswellemb : forall (md:mapdart)(x max :dart)(p:point),
  is_well_emb (fst (CHID md x p max)).
```

Finally, one of our initial preconditions, verified throughout the execution of the program states that there are no three collinear points. This property is defined by the predicate `is_noalign` (whose definition is omitted here) and we prove it holds after `CHID`:

```
Theorem noalign : forall (md:mapdart)(x max :dart)(p:point),
  is_noalign (fst (CHID md x p max)).
```

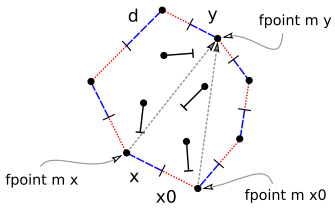


Figure 8. The convexity property

Convexity The central property states the polygon we build is actually convex. The definition `is_convex` expresses this property: a face (identified by `(m, d)`) is convex if for each dart `x` of this face (`eqf m d x`) and for each dart `y` whose embedding `py` is different from the embeddings `px` of `x` and `px0` of the 0-successor `x0 := cA m zero x` of `x`, the triple `(px, px0, py)` is oriented counter-clockwise:

```
Definition is_convex (m:fmap)(d:dart) : Prop :=
  forall (x:dart)(y:dart), eqf m d x -> exd m y ->
  let px := fpoint m x in let py := fpoint m y in
  let px0 := fpoint m (cA m zero x) in
  px <> py -> px0 <> py -> ccw px px0 py.
```

The theorem `convex` states that the result of the insertion function `CHID` verifies the convexity property. It is clear that the property also holds for `CH`.

```
Theorem convex : forall (md:mapdart)(x max :dart)(p:point),
  is_convex (fst (CHID md x p max)).
```

Convex Hull Property We put together all the properties required to have a convex hull. This is defined as follows:

```
Definition is_convex_hull (md:mapdart) : Prop :=
  let m := (fst md) in let d := (snd md) in
  inv_hmap m /\ inv_gmap m /\ inv_poly m d /\ planar m /\
  is_well_emb m /\ is_neq_point m /\ is_noalign m /\ is_convex m d.
```

8 Program Derivation in C++

Extraction into OCaml Coq features an extraction mechanism which can be used to automatically transform our Coq specification into an executable Ocaml program. This allows us to test *in real conditions* how our data structures and algorithm behave in terms of both usability and efficiency before starting any formal proof. The program extracted contains the code of functions `CH`, `CHI`, `CH2`, and `CHID`. We reuse the graphical interface we presented in [5] to select points of the plane and display the convex hull as a polygonal line (together with some remaining isolated points inside). For lack of space, we do not present the generated Ocaml code which is very close to the Coq specification and can be found in [4]. As such a basic extraction process may raise some efficiency issues especially because of the use of pattern-matching, we also study how to integrate our formally-proved program into a general platform for geometric modeling.

Implementing Hypermaps in C++ We derive a C++ implementation of an optimized program from our specification and integrate it into the library CGoGN (standing for Combinatorial and Geometric mOdeling with Generic N-dimensional Maps) [16] developped in our team to deal with hypermaps. The transformation is performed manually but most operations remain the same and are very close to those initially written in Coq. In the library CGoGN, hypermaps are represented by doubly-linked dart lists inherited from the Standard Template Library (STL) of C++. A dart in a hypermap is a pointer on a structure with a dart array to describe the topology (i.e. the links of the dart to its predecessors and successors) and, in our case, a pointer to a point structure equipped with a counter recording how many darts are embedded into this point (see Fig. 9).

Hypermaps have the type `hmap`. We build the C++ datatype `mapdart` of pairs and implement the corresponding access functions `fstmd` and `sndmd`, as well as the pair constructor `pairmd`.

```
typedef struct mapdartstruct { hmap m; dart d; } mapdart;
```

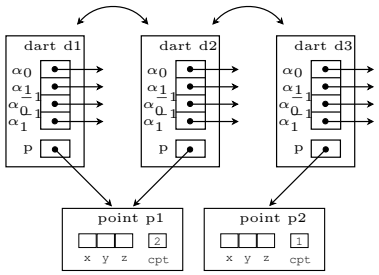


Figure 9. Representing hypermaps in C++ with CGoGN

Computing the Convex Hull We easily implement functions from our initial library on hypermaps, namely the atomic operations `V`, `I` and `L` and the higher-level ones `Merge` and `Split`. Note that the latter are programmed with side effects on the hypermap `m` and simply rewrite the already-existing links. We rewrite the Coq specification of the two functions `search_left` and `search_right` into two similar recursive C++ programs. The main functions `CH` and `CHI` do not proceed by pattern matching on the initial hypermap `m` but require to extract and then remove explicitely each dart using CCoGN functions. Finally, the insertion function `CHID` is programmed as follows. To create a dart in the hypermap, we simply use the function `gendart` which returns an identifier (a pointer), and then applies the function `I` which inserts the dart together with its embedding.

```
mapdart CHID (mapdart md, const point p) {
  hmap m = fstmd(md); dart d = sndmd(md);
  dart l = search_left(m,d,p,0);
  dart r = search_right(m,cA(m,zero,d),p,0);
  if (l==m->nil()) {
    dart x = gendart(m);
    m = I(m,x,p);
    md = pairmd(m,d);
  } else {
    dart l0 = cA(m,zero,l); dart r_0 = cA_1(m,zero,r);
    m = Split(m,zero,l,l0);
    if (l0!=r) m = Split(m,zero,r_0,r);
    dart x = gendart(m);
    m = I(m,x,p);
    dart max = gendart(m);
    m = I(m,max,p);
    m = Merge(m,one,max,x);
    m = Merge(m,zero,l,max);
    m = Merge(m,zero,x,r);
    md = pairmd(m,x); }
  return md; }
```

We also design an interface for the C++ program we derived. It allows us to test our program and also to visualize their inputs and outputs.

Complexity The complexity of the insertion C++ function `CHI` as implemented from its Coq specification is $O(n^2)$ in the worst case, where n is the number of points of the initial set. In the code of `search_left` and `search_right`, the calls to `Iter (cF m) i d` is replaced by a simple call to the function ϕ which returns the successor of a dart in the face of `d`. This avoids recomputing the first i-successors of `d` at each step. Consequently, the complexity of the incremental algorithm is $O(n^2)$ as in any implementation of this kind of algorithm.

9 Conclusion

The algorithm we specified proceeds by a traversal of the current convex hull when a new point is inserted. It is closer to the usual implementations [13, 9], than the one we presented in [5]. We proved that the algorithm is correct by establishing in Coq its topological and geometric properties. We also automatically extracted our specification into a prototype Ocaml program and derived an implementation in C++ using the library CGoGN on hypermaps.

Compared to the approach followed in [5], this work presents a higher-level specification and more compact proofs, especially thanks to `Merge` and `Split`. In addition, we follow the usual behavior of the incremental algorithm and prove more properties than in our first attempt. Proofs about numbering of faces and connected components, especially the proofs they increase by at most one with each insertion of a dart, are successfully carried out. Finally, we could consider removing the dangling chains of darts lying inside the convex hull and prove we have only one connected component at the end. Our specification fits in about 2500 lines and the proofs require around 6000. It relies on the hypermap library developped in Strasbourg, which now reaches 7000 lines of specifications and 55000 lines of proofs. Note that the size of this basic library grew extensively since our first experiment [5] and now contains several higher-level operations as well as proofs of their properties. Therefore specifying our algorithm and proving it is correct is much shorter (around 6000 lines overall).

The program derived in C++ remains very close to the specification in Coq. A further research direction would be to carry out a formal proof of the implementation using tools such as Why and Frama-C [6] as was done by one of the authors for integer arithmetics [2]. In addition, other computational geometry algorithms [12] are currently revisited using our formal approach with Coq and hypermaps. Furthermore, a great challenge would be to investigate the third dimension, by considering surfacic convex hulls as well as volumic subdivisions − into edges, vertices, faces and volumes −, for instance for tetrahedric Delaunay diagrams.

References

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[2] Y. Bertot, N. Magaud, and P. Zimmermann. A Proof of GMP Square Root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.

[3] Y. Bertrand, J.-F. Dufourd, J. Françon, and P. Lienhardt. Algebraic Specification and Development in Geometric Modeling. In *TAPSOFT'93: Theory and Practice of Software Development*. LNCS, 668:75–89, Springer, 1993.

[4] C. Brun, J.-F. Dufourd, and N. Magaud. Formal Proof of the Incremental Convex Hull Algorithm. `http://galapagos.gforge.inria.fr`.

[5] C. Brun, J.-F. Dufourd, and N. Magaud. Designing and Proving Correct a Convex Hull Algorithm with Hypermaps in Coq. *Computational Geometry, Theory and Applications*, 2010 (to appear).

[6] CEA-INRIA. The Frama-C Software Verification Toolkit. `http://frama-c.cea.fr`.

[7] Coq Development Team. The Coq Proof Assistant - Reference Manual and Library. `http://coq.inria.fr/`.

[8] R. Cori. *Un code pour les graphes planaires et ses applications*. Astérisque, 27. Société mathématique de France, 1970.

[9] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry, Algorithms and Applications*. Springer, 2008.

[10] J.-F. Dufourd. Design and Formal Proof of a New Optimal Image Segmentation Program with Hypermaps. *Pattern Recognition*, 40(11):2974–2993, 2007.

[11] J.-F. Dufourd. An Intuitionistic Proof of a Discrete Form of the Jordan Curve Theorem Formalized in Coq with Combinatorial Hypermaps. *Journal of Automated Reasoning*, 43(1):19–51, 2009.

[12] J.-F. Dufourd and Y. Bertot. Formal Study of Plane Delaunay Triangulation. In *Interactive Theorem Proving*. LNCS, 6172:211–226, Springer, 2010.

[13] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1987.

[14] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The Design and Implementation of Planar Maps in CGAL. In *WAE'99: Algorithm Engineering*. LNCS, 1668:154–168, Springer, 1999.

[15] G. Gonthier. Formal Proof - The Four-Colour Theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[16] IGG Team. CGoGN: Combinatorial and Geometric mOdeling with Generic N-dimensional Maps. `https://iggservis.u-strasbg.fr/CGoGN/`.

[17] D. Knuth. *Axioms and Hulls*. LNCS, 606. Springer, 1992.

[18] P. Lienhardt. Topological Models for Boundary Representation: a Comparison with n-Dimensional Generalized Maps. *Computer-Aided Design*, 23:59–82, 1991.

[19] L. Meikle and J. Fleuriot. Mechanical Theorem Proving in Computational Geometry. In *Automated Deduction in Geometry*. LNCS, 3763:1–18, Springer, 2006.

[20] D. Pichardie and Y. Bertot. Formalizing Convex Hull Algorithms. In *Theorem Proving in Higher Order Logics*. LNCS, 2152:346–361, Springer, 2001.