Designing and proving correct a convex hull algorithm with hypermaps in Coq [☆]

Christophe Brun, Jean-François Dufourd, Nicolas Magaud

Université de Strasbourg - Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection (LSIIT, UMR 7005 CNRS-UDS) Pôle API, Boulevard Sébastien Brant, BP 10413, 67412 Illkirch, France
{*brun,dufourd,magaud*}*@lsiit-cnrs.unistra.fr*

Abstract

This article presents the formal design of a functional algorithm which computes the convex hull of a finite set of points incrementally. This algorithm, specified in Coq, is then automatically extracted into an OCaml-program which can be plugged into an interface for data input (point selection) and graphical visualization of the output. A formal proof of total correctness, relying on structural induction, is also carried out. This requires to study many topologic and geometric properties. We use a combinatorial structure, namely hypermaps, to model planar subdivisions of the plane. Formal specifications and proofs are carried out in the Calculus of Inductive Constructions and its implementation: the Coq system.

Key words: convex hull, hypermaps, formal specifications, computer-aided proofs, Coq system.

1. Introduction

Our general aim is to lead a formal survey in geometric modeling and computational geometry in order to improve the programming techniques and ensure the algorithms correctness. In this paper, we present a formal case study in computational geometry on a classical problem which involves elementary geometric objects: computing the incremental convex hull of a finite collection of planar points.

[☆]This work is partly supported by the research project GALAPAGOS, French ANR

The originality of the means we use to achieve our purpose rely on one hand on the fact that the specifications and the formal proofs of programs are expressed in the formalism of the Calculus of Inductive Constructions implemented in the Coq system, and on the other hand on the fact that we work in a topology-based geometric modeling framework where the planar subdivisions are described by combinatorial oriented maps [34]. But, in order to be more general for the subsequent applications, we first use combinatorial hypermaps and then specialize them into combinatorial oriented maps.

A (two-dimensional) hypermap is a simple algebraic structure consisting of a finite set whose elements are called *darts* and of two permutations on this set. It allows to model surface subdivisions (into vertices, edges and faces) and to distinguish between the topologic and geometric aspects of the studied objects. For years, we have formally described hypermaps in order to handle subdivisions and their transformations as well as to prove topologic properties of surfaces [9, 11]. Our hypermap specification is done by structural induction, which makes the constructive definition of operations and the proofs of surface properties easier. However, for the moment, we almost exclusively worked on the combinatorial topology of surfaces, but we want to deal with geometric embeddings as well. That is the reason why we begin by studying a classical plane problem which is not only rich enough to highlight many interesting problems, but also simple enough to reveal them easily and completely.

The geometric aspects we consider are particularly simple but fundamental in computational geometry. The embedding is straightforward and 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 for orientation [25]. This system defines orientation according to the order in which a triple of points is enumerated in the plane (either clockwise or counter-clockwise). One of its advantages is that it allows to isolate the required numerical tests and in a first step to elude the difficult numerical accuracy problems. In fact, we do not address these issues in this first attempt 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.

In this setting, our work consists in designing a functional convex hull algorithm, automatically extracting a program in OCaml augmented with input handling and a graphical display of the result, and formally proving its total correctness. This proof consists in checking the termination of the algorithm as well as highlighting several useful topologic and geometric properties. All our specification and proof development was interactively assisted by the Coq proof assistant [1, 21, 33].

In Section 2, we list and briefly survey some related works about formalized proofs in combinatorial topology and computational geometry. In Section 3, we recall some basic mathematical definitions and properties on the combinatorial oriented hypermaps and then specify them in Coq. In Section 4, we propose a numerical model of the plane based on real numbers and slightly enlarge Knuth's axiom system for orientation. In Section 5, we describe our functional algorithm to build a convex hull incrementally. In Section 6, we formalize this algorithm in the Coq proof assistant. In Section 7, we explain how to extract from this description an operational program in OCaml. In Section 8, we present and prove the topologic properties required to establish the correctness of this algorithm. In Section 9, we do the same with the geometric properties. Finally, some conclusions and future works are given in Section 10.

In the following, the Coq notions required to understand the developments are progressively introduced, but the details of the proofs are out of the scope of this paper.

2. Related work

2.1. Convex hull computation and subdivision modelling

The computation of the convex hull of a finite set of points is one of the first and most important concepts studied in computational geometry. It has several different definitions in the literature. There are also several construction methods, such as the incremental algorithm, Jarvis' march or Graham's scan [4, 7, 8, 14, 31, 32].

In a two-dimensional setting, the convex hull is a polygon, but its construction often requires to handle broken lines or even several polygons (e.g. in the divide and conquer approach). More generally, geometric algorithms deal with irregular subdivisions of surfaces into vertices, edges and faces. Even if they may be fairly simple in most computational geometry algorithms, these subdivisions are worth being handled consistently. That is why, in our work, we have a strong focus on these subdivisions and their properties. Nowadays, a good way of studying geometric objects is to distinguish between their topological structure and their embedding. Topology, as a combinatorial tool, may be defined via a concrete datatype such as *half edges* [35], *winged edges* [27] or *quad-edges* [20].

A widespread approach in computer science is to encourage abstract representations of data. *Combinatorial oriented maps* of dimension two [34] allow to describe, in an algebraic way, general subdivisions of closed orientable surfaces, which is exactly what we require for our study of convex hulls. However, we prefer to work first with *combinatorial hypermaps* [6] which are more general, homogeneous in the two dimensions and easier to specialize depending on our needs. Thus, we shall be able to reuse our work dealing with hypermaps to study more complex subdivisions. Then, we constrain the hypermaps in order to capture combinatorial oriented maps exactly. Combinatorial oriented maps and their extensions have been studied extensively and led to several implementations in geometric modelling [15, 22, 26]. One of the implementations was carried out in the library CGAL [16].

In addition to topology, 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 D. Knuth in his book "Axioms and Hulls" [25]. This axiom system, based on orientation properties of triples of points in the plane, allows to isolate numerical accuracy issues in computations and let us focus on the logic tests required in the algorithms. This approach is particularly well-suited to carry out formal proofs of correctness of the considered algorithms.

2.2. Assisted proofs in computational geometry

Formal proofs in the field of computational geometry, especially focusing on convex hull algorithms have been carried out. Pichardie and Bertot use the Coq proof system to develop a formal proof of correctness of the incremental algorithm as well as Jarvis' march [30]. They also consider Knuth's axiom system but they simply represent convex hulls as lists of points which lead to several technicalities and is likely to prevent any further extension, especially in a three-dimensional setting. Meikle and Fleuriot [28] use the Isabelle proof system to formally prove the correctness of a program computing convex hulls using Graham's scan. Their approach relies on Hoare logic, which prevents them from having a simple functional description of the program.

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. [17, 18]. Their specification allows to prove some significant results including a proof of the Jordan curve theorem which forms the cornerstone of the proof of the four-color theorem. However, their specification approach as well as the proof techniques (using reflection in Coq) [19] are fairly different from the methodology we follow in this paper. An in-depth comparison can be found in [13].

At Strasbourg University, the library specifying hypermaps, onto which our present work on convex hulls is built, was successfully used to prove some basic results such as the genus theorem, Euler formula for polyhedra [12] and a discrete version of the Jordan curve theorem [13]. This library was also used to carry out a formal proof of correctness of a functional algorithm to perform image segmentation by merging adjacent faces and to develop a time-optimal C-program [10].

3. Hypermaps, combinatorial oriented maps and their specification in Coq

In this section, we follow the presentation carried out in [12] but restrict ourselves to the notions relevant to this work. We introduce the notions of combinatorial hypermaps and maps to represent our input data, intermediate computations, and the resulting convex hull. We start with mathematical definitions and then explain how to formalize such definitions in the framework of Coq.

3.1. Mathematical aspects

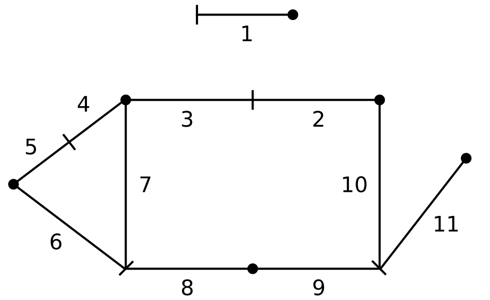
3.1.1. Definitions

Hypermaps are one of the most general structures to describe finite surface subdivisions topologically.

Definition 1 (Hypermap and combinatorial oriented map).
(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 without fixpoint on D (i.e. $\forall x \in D$, $\alpha_0(\alpha_0(x)) = x$ and $\alpha_0(x) \neq x$), then M is called a *combinatorial oriented map.*
(3) For each dimension $k \in \{0, 1\}$: if $y = \alpha_k(x)$, y is the *k-successor* of x, x is the *k-predecessor* of y, and x and y are said to be *k-linked* together.

Thus the *combinatorial oriented maps* are a subclass of the class of *hypermaps.* The notion of hypermaps is well suited to carry out formal proofs [18]. However combinatorial oriented maps are much easier to use in geometric modeling [15, 16, 22, 26, 34].

Example 1. In Fig. 1, as functions α_0 and α_1 on $D = \{1, \ldots, 11\}$ are permutations (i.e. one-to-one correspondences), $M = (D, \alpha_0, \alpha_1)$ is a hypermap. It



D	1	2	3	4	5	6	7	8	9	10	11
α_0	1	3	2	5	4	8	6	7	11	9	10
α_1	1	10	4	7	6	5	3	9	8	2	11

Figure 1: An example of hypermap

represents a subdivision of the plane with a triangle and a rectangle adjacent to one another; it also features a hanging line segment as well as an isolated segment.

In our drawings of hypermaps on surfaces, we represent each dart as a simple curved segment (a line segment in the plane) oriented from a bullet to a small stroke: 0-linked (resp. 1-linked) darts share the same small stroke (resp. bullet). By convention, we always adopt that k-successors turn counterclockwise on the plane around small strokes and bullets. Note that our hypermap definition allows the void map (i.e. $D = \emptyset$) and fixpoints with respect to k.

3.1.2. Cells of hypermaps

The topological cells of a hypermap (i.e. its vertices, edges, faces and connected components) can be combinatorially defined, mainly through the classical notion of orbit.

Definition 2 (Orbits and hypermap cells).
(1) Let f be a permutation in a finite set D. The *orbit* of $x \in D$ for f is the dart sequence $\langle f\rangle(x) = (x, f(x), f^2(x), \ldots, f^{p-1}(x))$, where p, called the *period* of the orbit, is the smallest integer such that $f^p(x) = x$.
(2) In an hypermap $M = (D, \alpha_0, \alpha_1)$, $\langle\alpha_0\rangle(x)$ is the *0-orbit* or *edge* of dart x, $\langle\alpha_1\rangle(x)$ its *1-orbit* or *vertex*, $\langle\phi\rangle(x)$ its *face*, for $\phi = \alpha_1^{-1} \circ \alpha_0^{-1}$.
(3) The *connected component* of x in M, denoted by $\langle\alpha_0, \alpha_1\rangle(x)$, is the set of darts which are accessible from x by any composition sequence of α_0 and α_1.

Faces are defined, through $\phi = \alpha_1^{-1} \circ \alpha_0^{-1}$, for a dart traversal also in counterclockwise order, when the hypermap is drawn on a surface. Then, every face which encloses a bounded (resp. unbounded) region on its left is called *internal* (resp. *external*).

Example 2. In Fig. 1, the hypermap M contains 5 edges (strokes), 6 vertices (bullets), 4 faces and 2 connected components. For instance, $\langle\alpha_0\rangle(7) = (7, 6, 8)$ is the edge of dart 7, $\langle\alpha_1\rangle(7) = (7, 3, 4)$ its vertex. Regarding ϕ which is equal to $\alpha_1^{-1} \circ \alpha_0^{-1}$, we have $\phi(2) = 7$, $\phi(7) = 9$ and $\phi(9) = 2$. Then the (internal) face of 2 is $\langle\phi\rangle(2) = (2, 7, 9)$. In addition the (external) face of 3 is $\langle\phi\rangle(3) = (3, 10, 11, 8, 5)$.

Since α_0 and α_1 are permutations, it is clear that, for $\Pi = \langle\alpha_0\rangle, \langle\alpha_1\rangle, \langle\alpha_1^{-1} \circ \alpha_0^{-1}\rangle$ or $\langle\alpha_0, \alpha_1\rangle$, $y \in \langle\Pi(x)\rangle$ is equivalent to $x \in \langle\Pi(y)\rangle$. In a combinatorial oriented map, each edge is composed of exactly 2 darts. This is standard practice in geometric modeling to represent orientable surface subdivisions [15, 16, 22, 26, 34]. We shall adopt this approach later in this work.

3.1.3. Planarity and Euler formula

Let d, e, v, f and c be the numbers of darts, edges, vertices, faces and connected components of a hypermap $M = (D, \alpha_0, \alpha_1)$.

Definition 3 (Euler characteristic, genus, planarity).
(1) The *Euler characteristic* of M is $\chi = v + e + f - d$.
(2) The *genus* of M is $g = c - \chi/2$.
(3) When $g = 0$, M is said to be *planar*.

Example 3. In Fig. 1, the Euler characteristic of M is $\chi = 6 + 5 + 4 - 11 = 4$ and its genus $g = 2 - \chi/2 = 0$. Consequently, the hypermap M is planar.

A planar hypermap satisfies the property $\chi = 2 \times c$, which is a generalization of the well-known *Euler formula*.

3.1.4. Embedding

We only consider embedding issues for combinatorial oriented maps. For this class of hypermaps, the embedding into the plane is a mapping of vertices into distinct points, edges into straight lines connecting two points (being two embedded vertices), and faces as possibly open regions of the plane. For more details on embeddings and on the planarity, the reader is refered to [12, 13].

3.2. Specifications in Coq

Coq [1, 33] is the implementation of the Calculus of Inductive Constructions, which is a type theory as well as a powerful higher-order intuitionistic logical framework designed to formalize and prove mathematical properties in an interactive way. All the definitions of the previous section are formalized in this framework.

3.2.1. Preliminary specifications

We first define a type for the dimensional indexes 0 and 1 of an hypermap. It consists in an inductive type `dim`:

```
Inductive dim : Set := zero : dim | one : dim.
```

All objects being typed in Coq, `dim` has the type `Set` of all concrete types. Its *constructors* are the constants `zero` and `one`. For each inductive type, the generic equality predicate `=` is built-in but its decidability is not, because the logic of Coq is intuitionistic. For `dim`, the latter can be established as the following lemma (note that in Coq `~` stands for logic negation, `+` or `\/` for disjunction and `/\` for conjunction):

```
Lemma eq_dim_dec : forall (i:dim)(j:dim), {i=j} + {~i=j}.
```

Once it is made, its proof is an object of the sum type {`i=j`}+{`~i=j`}, i.e. a function, named `eq_dim_dec`, which tests whenever its two arguments are equal or not. This lemma is interactively proved with some tactics, the reasoning being a simple structural induction on both `i` and `j`, which boils down to a simple case analysis here. Indeed, from each inductive type definition, Coq generates an *induction principle*, usable either to prove propositions or to build total functions on the type.

Next, we identify the type `dart` and its equality decidability `eq_dart_dec` with the built-in type of natural numbers `nat` and `eq_nat_dec`. Finally, to manage exceptions, a `nil` dart is a renaming of `0`:

```
Definition dart := nat.
Definition eq_dart_dec := eq_nat_dec.
Definition nil := 0.
```

We choose a constructive point of view for hypermaps, which is close to the usual incremental building of surface subdivisions in geometric modeling rather than considering an observational point of view with an already built set of darts equipped with all its permutations, as it is done in [18].

3.2.2. Free maps

The hypermaps are now approached by a general notion of *free map*, thanks to a free algebra of terms of inductive type `fmap` with 3 constructors, `V`, `I` and `L`, respectively for the *empty* (or *void*) map, 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.
```

Example 4. The hypermap M in Fig. 1 can be modeled by the free map represented in Fig. 2 where the 0- and 1-links by `L` are represented by arcs of circles, and where the orbits remain open. For instance, a submap of the hypermap M of Fig. 2, consisting of darts $3, 2, 9$ and 10 is represented by the following term in Coq : `(L (L (L (I (I (I (I V 3 p3) 2 p2) 10 p10) 9 p9) zero 3 2) one 10 2) zero 10 9)`.

When darts are inserted into a free map, they come together with an embedding `point` which is a couple of real numbers. As the reader may see from Fig. 2, some geometrical consistency properties must be enforced. For instance, the points `p2` and `p10` respectively associated with 2 and 10 must be equal.

Coq also generates an induction principle on free maps. In the following, the use of the constructors will be constrained by preconditions to avoid meaningless free maps. The corresponding subtype of the hypermaps will be characterized by an invariant, called `inv_hmap`, systematically used in conjunction with `fmap` (see Section 3.2.3 for details).

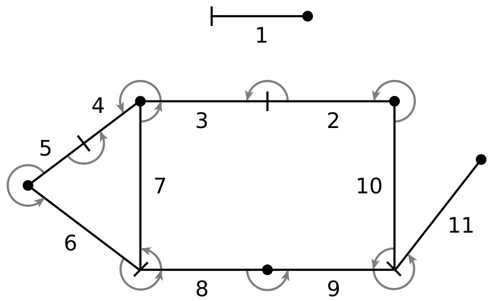


Figure 2: Hypermap example with its incompletely linked orbits

Next, *observers* of free maps can be defined. The predicate `exd` expresses that a dart exists in a hypermap. Its definition is recursive, which is indicated by the keyword `Fixpoint`. It proceeds by pattern matching on `m` written `match m with...`. The attribute {`struct m`} allows Coq to verify that the recursive calls are performed on smaller `fmap` terms, thus ensuring termination. The result is either `False` or `True`, the two basic constants of `Prop`, the built-in type of propositions. Note that terms are in prefix notation and that _ is a place holder. Proving the decidability `exd_dec` of `exd` is straightforward by induction on `m`.

```
Fixpoint exd (m:fmap)(d:dart) {struct m} : Prop :=
  match m with
    V => False
  | I m0 x _   => x = d \/ exd m0 d
  | L m0 _ _ _ => exd m0 d
  end.
```

Then, a restriction of function α_k, denoted `A`, is defined. It is designed so that its orbits can not be closed. However, because Coq only allows total functions to be defined, `A` is extended with the `nil` dart when it will otherwise close the orbit (the inverse `A_1` being similar):

```
Fixpoint A (m:fmap)(k:dim)(d:dart) {struct m} : dart :=
  match m with
    V => nil
  | I m0 _ _    => A m0 k d
  | L m0 k0 x y =>
    if eq_dim_dec k k0
    then if eq_dart_dec x d then y else A m0 k d
    else A m0 k d
  end.
```

Predicates `succ` and `pred` express that a dart has a `k`-successor and a `k`-predecessor (non-`nil`), with the decidability properties `succ_dec` and `pred_dec`:

```
Definition succ (m:fmap)(k:dim)(d:dart) : Prop := A m k d <> nil.
```

Example 5. In Fig. 2, `A m zero 6 = 8`, `A m zero 4 = nil`, `succ m zero 6`, `~succ m zero 4`, `A_1 m one 9 = 8`, `pred m one 9`.

3.2.3. Hypermaps

As said previously, preconditions written as predicates are introduced for operators `I` and `L`. The precondition `prec_I` for `I` states that the `nil` dart can not be inserted into a free map and that a dart `x` can only be inserted if it does not already belong to the free map. The precondition `prec_L` for `L` verifies that the darts `x` and `y` we want to link to one another are actually already inserted in the free map, that `x` has no successor at the involved dimension and that `y` has no predecessor at this dimension either. Finally, it also prevents a link from `x` to `y` from being added if it would close the orbit of `x`.

```
Definition prec_I (m:fmap)(x:dart) : Prop :=
  x <> nil /\ ~ exd m x.
```

```
Definition prec_L (m:fmap)(k:dim)(x:dart)(y:dart) : Prop :=
  exd m x /\ exd m y /\
  ~ succ m k x /\ ~ pred m k y /\ cA m k x <> y.
```

If `I` and `L` are only used when the appropriate precondition holds, the built free map necessarily has *open orbits*. Such a condition was required to make merging orbits by concatenation easier. It also reduces the number of links required in the computation of the convex hull. Overall the built free map satisfies the *invariant*:

```
Fixpoint inv_hmap (m:fmap) : Prop :=
  match m with
    V => True
  | I m0 x t p => inv_hmap m0 /\ prec_I m0 x
  | L m0 k x y => inv_hmap m0 /\ prec_L m0 k x y
  end.
```

Such a hypermap was already drawn in Fig. 2. In fact, thanks to other operations namely `cA` and `cA_1`, it can always be considered as a true hypermap exactly equipped with operations α_k.

Indeed, the operations `cA` and `cA_1` *close* `A` and `A_1`; thus we can do as if the `k`-orbits were closed. In addition, for any k (`A m k`) and (`cA m k`) extend the function α_k to darts which do not belong to the map m and return the dart `nil`.

Example 6. In Fig. 2, `cA m one 4 = 7, cA_1 m one 7 = 4, cA m one 11 = nil`, `cA_1 m one 11 = nil`. In addition, when the input dart does not belong to the map, we have `cA m zero 12 = nil` and `cA_1 m zero 12 = nil`.

Fundamental properties we prove are: for any `m` and `k`, `(A m k)` and `(A_1 m k)` are *injections* inverse of each other, and `(cA m k)` and `(cA_1 m k)` are *permutations* inverse of each other, and are closures. The reader interested in the technical details is referred to our formal proof development [1].

Finally, traversals of faces are based on a function `F` and its closure `cF` (see [12] for details), which correspond to ϕ as defined in Definition 2. Properties similar to the ones of `A`, `cA` are proved for `F`, `cF` and their inverses `F_1`, `cF_1`.

Example 7. In Fig. 2, `F m 4 = nil, cF m 4 = 6`.

Further topologic properties may be considered while proving the correctness of our convex hull algorithm. In addition, invariants dealing with geometry must be defined. So we now present the geometric setting in which our computations take place.

4. Geometric setting

Convex hull computations do not only rely on topology but also on geometric properties of the involved points. In this article, we choose to work with Cartesian geometry in two dimensions and we consider each point p to be a couple of reals which are its coordinates in the plane (i.e. $p = (x_p, y_p)$ with $x_p, y_p \in \mathbb{R}$). To compute convex hulls incrementally, we need a predicate to determine the orientation of three points in the plane.

As in [28] and [30], we follow Knuth's approach to handle orientation in the plane. We first specify the orientation predicate with its properties and then implement it when the plane is represented by $\mathbb{R}^2$.

4.1. Specification

The predicate `ccw`(p, q, r) expresses whether the points p, q, r are enumerated clockwise or not. Fig. 3 exemplifies this orientation predicate *ccw* for such a triple of points. The example on the left (a) denotes a case where the triple (p, q, r) is oriented counter-clockwise. The one in the middle (b) denotes a case where the three points are collinear. Finally, the one on the right (c) denotes a case where the triple (p, q, r) is oriented clockwise.

In fact, Knuth chooses not to handle degenerate cases. He assumes that three points are always in *general position*, i.e. no two of them coincide and they do not all lie on the same line. In our work, we assume the same, therefore the case (b) of Fig. 3 can not happen.

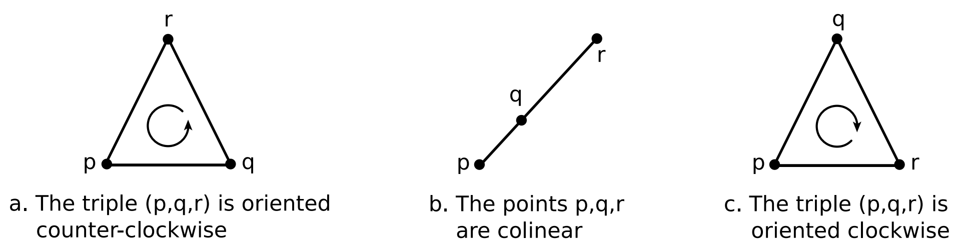

r
p
q
a. The triple (p,q,r) is oriented counter-clockwise
r
q
p
b. The points p,q,r are colinear
q
p
r
c. The triple (p,q,r) is oriented clockwise

Figure 3: The orientation predicate

The orientation predicate is specified as follows:

Property 1 (Geometric orientation predicate).
P.1 (cyclicity): $\forall p, q, r, ccw(p, q, r) \Rightarrow ccw(q, r, p)$.
P.2 (symmetry): $\forall p, q, r, ccw(p, q, r) \Rightarrow \neg ccw(p, r, q)$.
P.3 (non-degeneracy): $\forall p, q, r, \neg collinear(p, q, r) \Rightarrow ccw(p, q, r) \vee ccw(p, r, q)$.
P.4 (interiory): $\forall p, q, r, t, ccw(t, q, r) \wedge ccw(p, t, r) \wedge ccw(p, q, t) \Rightarrow ccw(p, q, r)$.

P.5 (transitivity): $\forall p, q, r, s, t, ccw(t, s, p) \wedge ccw(t, s, q) \wedge ccw(t, s, r) \wedge ccw(t, p, q) \wedge ccw(t, q, r) \Rightarrow ccw(t, p, r)$.
P.5 bis (dual transitivity): $\forall p, q, r, s, t, ccw(s, t, p) \wedge ccw(s, t, q) \wedge ccw(s, t, r) \wedge ccw(t, p, q) \wedge ccw(t, q, r) \Rightarrow ccw(t, p, r)$.

Note that even if the collinearity case does not happen, a complete axiomatization requires to have an additional predicate `collinear` which expresses that three points lie on the same line. Properties 1, 2, and 3 are immediate to understand. Properties 4, 5, and 5 bis are illustrated in Fig. 4. Dotted lines correspond to premisses and solid lines to conclusions of these properties.

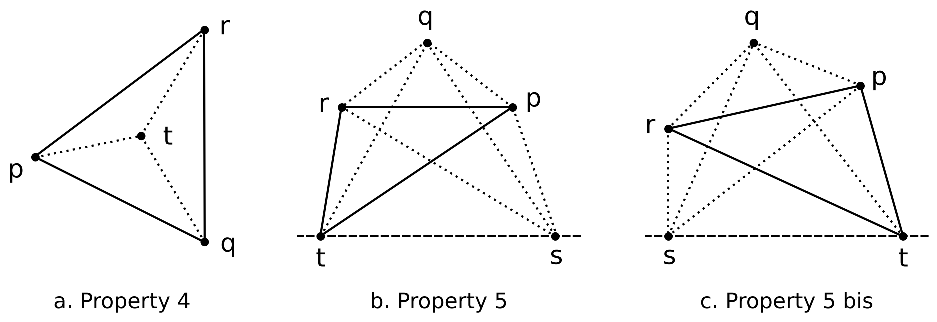


Figure 4: Properties 4, 5 and 5 bis of Knuth's orientation predicate *ccw*

All these properties are required not only to design an algorithm which works fine and without bugs for any configuration of points in general position, but also to carry out its proof of correctness. We shall use this specification of the orientation predicate as an interface in our implementation of the convex hull algorithm. However, to make sure it is consistent, we do also prove all the above mentionned properties hold in our setting.

4.2. Implementing Knuth's orientation predicate for $\mathbb{R}^2$

Our implementation uses the following concrete definition of *ccw*.

Definition 4 (Orientation of a triple of points).
Let (p, q, r) be a triple of points in the plane whose coordinates in $\mathbb{R}$ are (x_p, y_p), (x_q, y_q) et (x_r, y_r). The orientation predicate is defined according to the sign of the determinant $det(p, q, r)$.

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

This means, if $det(p, q, r) > 0$, $ccw(p, q, r)$ holds (and p, q, r are enumerated counter-clockwise), whereas, if $det(p, q, r) \leq 0$, $ccw(p, q, r)$ does not hold (and p, q, r are enumerated clockwise or are collinear).

Real numbers are described in Coq using an axiom system [33]. Basic operations (+, -, $\times$, /) are specified and their more advanced properties are derived from this abstract specification. Thus, the function `det` can be easily implemented as follows:

```
Definition det (p q r : point) : R :=
  (fst p * snd q) - (fst q * snd p) - (fst p * snd r) +
  (fst r * snd p) + (fst q * snd r) - (fst r * snd q).
```

From this definition, we derived the orientation predicate `ccw`:

```
Definition ccw (p q r : point) : Prop := (det p q r > 0).
```

From this definition and properties of real numbers, we formally prove in Coq that all the properties of the specification hold. In addition, the orientation property is decidable, meaning it can be used in conditional expressions of algorithms. The theorem `ccw_dec` expresses this decidability property and is formally proved in Coq.

```
Lemma ccw_dec : forall (p q r : point), {ccw p q r}+{~ccw p q r}.
```

We now have a framework to handle the orientation predicate in a formal way. No issue related to numerical computations shall be considered in the rest of this article. We shall only consider we have a decidable predicate `ccw` available, which satisfies the above-mentionned specification and can be used to determine the orientation of a triple of points in the plane.

5. Convex hull and incremental algorithm

In this section, we introduce the convex hull concept and we describe the incremental convex hull algorithm whose formal correctness shall be proved.

5.1. Convex hull definition

The computation of planar convex hulls in one of the first problems that was studied in computational geometry. Many definitions leading to different algorithms were proposed in the literature [4, 8, 14, 31]. In this work, we choose a definition well-suited for our topological hypermap model, for using Knuth's orientation predicate *ccw* and for the incremental algorithm we will study.

Let P be a set of points in the plane. Like most of the authors, we assume that points are in *general position*, i.e. *no two points coincide* and *no three ones are collinear.*

Definition 5 (Convex hull).
The *convex hull* of S is the convex polygon P whose vertices t_i, numbered in a counterclockwise order traversal for $i = 1, \ldots, n$ with $n + 1 = 1$, are points of S such that, for each edge $[t_i t_{i+1}]$ of P and for each point p of S different from t_i and from t_{i+1}, $ccw(t_i, t_{i+1}, p)$ holds. In other words, every point p of S different from t_i and t_{i+1} lies on the left of the oriented line generated $\overrightarrow{t_i t_{i+1}}$.

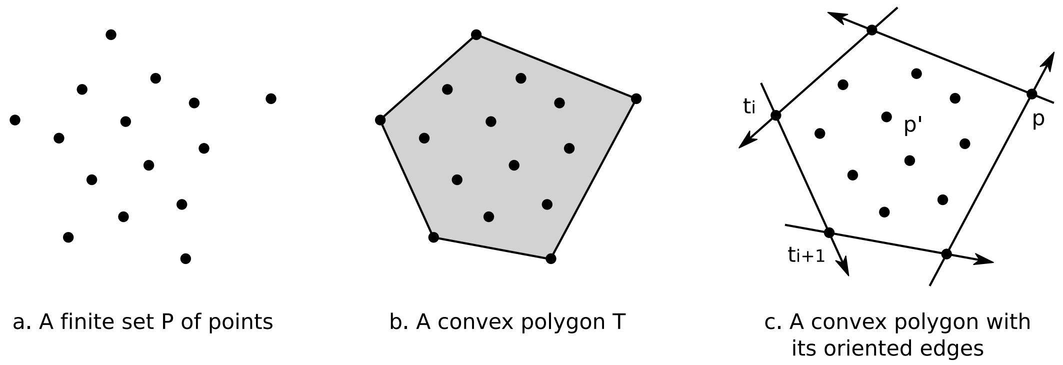

a. A finite set P of points
b. A convex polygon T
ti
p'
p
ti+1
c. A convex polygon with its oriented edges

Figure 5: Characterizing a convex hull

Fig. 5 shows a characterization of a convex hull using predicate *ccw*. Left (a), we have a finite set P of points. In the middle (b), we have a convex polygon T with its greyed interior (which shall be formally defined later in the article). Right (c), arrows denote oriented lines $\overrightarrow{t_i t_{i+1}}$ derived from edges $[t_i t_{i+1}]$ of T. All featured points $p, p', \ldots$ do lie on the left of these oriented lines.

5.2. Incremental algorithm

The *incremental algorithm* computes the convex hull of P by building it step by step. At each step, a new point of P is considered and a new convex hull is computed. It takes as input the current hull (the one built with all the already-processed points). Then, either the new point lies *inside* the already-built polygon and the algorithm moves on to the next step, or it lies *outside* of the polygon and the algorithm will have to remove some edges and add two new ones to build a new convex polygon. This corresponds to the usual naive algorithm which is found in most books of computational geometry (e.g. in [8]). Since we assume that points are in general position, the new point can never be *on* the already-built polygon, i.e. be equal to a previously-added point or lie on an existing edge.

The incremental algorithm can be decomposed into three functions named `CH`, `CHI` and `CHID` in the code.

• The first function `CH` initiates the incremental computation of the convex hull. For a single point, the convex hull is the point itself. For more than one point, the algorithm starts with an initial set containing only two points and computes a first convex hull which is simply an *edge* linking the two points. Then it calls the function `CHI` with this first convex hull and the remaining points to be treated.
• The second function, `CHI`, takes every element s of the initial set P and calls the insertion function `CHID` to build a new convex hull. It proceeds by case analysis. Then, for each new point s in P, it extends the already-built convex polygon using the insertion operation `CHID`.
• The last one, `CHID`, computes the convex hull of a convex polygon T and an extra point s, i.e. it inserts s into the already-built convex hull polygon T. It uses tests based on Knuth's orientation predicate *ccw*. According to Definition 5, we know that the *interior* of polygon T is defined by the points x of the plane such that $ccw(t_i, t_{i+1}, x)$ for any edge $[t_i t_{i+1}]$ of T.

In addition, the line generated by $\overrightarrow{t_i t_{i+1}}$ divides the plane remainder into two open half planes characterized by the value of $ccw(t_i, t_{i+1}, x)$ for every point x. Therefore, one can easily locate the point s with respect to each edge $[t_i t_{i+1}]$ of the polygon T. We simply have to evaluate $ccw(t_i, t_{i+1}, s)$. Repeating this test for all $i = 1, \ldots, n$, this tells us whether s lies inside or outside T.

If s lies *inside* T, the convex hull of $(T \cup s)$ is the same as the one of T. Otherwise, s necessarily lies *outside* T, the algorithm removes edges of T which are *visible* from s and creates two new edges $[t_l s]$ and $[s t_r]$ to connect s to the *leftmost vertex* t_l and to the *rightmost vertex* t_r. All these notions are defined precisely in the following definitions and illustrated in Fig. 6.

Definition 6 (Visible edges, leftmost vertex, rightmost vertex).
Let T be a planar convex polygon with at least two vertices and s be a point of the plane.
(1) The edge $[t_i t_{i+1}]$ of T is *visible* from s whenever $\neg\, ccw(t_i, t_{i+1}, s)$ holds.
(2) The vertex t_l of T is the *leftmost vertex* with respect to s if $ccw(t_{l-1}, t_l, s)$ and $\neg\, ccw(t_l, t_{l+1}, s)$ hold.
(3) The vertex t_r of T is the *rightmost vertex* with respect to s if $\neg\, ccw(t_{r-1}, t_r, s)$ and $ccw(t_r, t_{r+1}, s)$ hold.

Note that we shall have to prove the equivalence of the existence of t_l and t_r later in this article. Indeed, when s is inside the polygon, t_l and t_r do not exist. Otherwise, when s is outside, both of them exist. No other cases shall be consider as s can not be collinear with two of the vertices of the convex polygon. In addition, we shall prove the uniqueness of these two vertices t_l and t_r when they exist.

This algorithm shall be formalized according to our data structures, namely hypermaps.

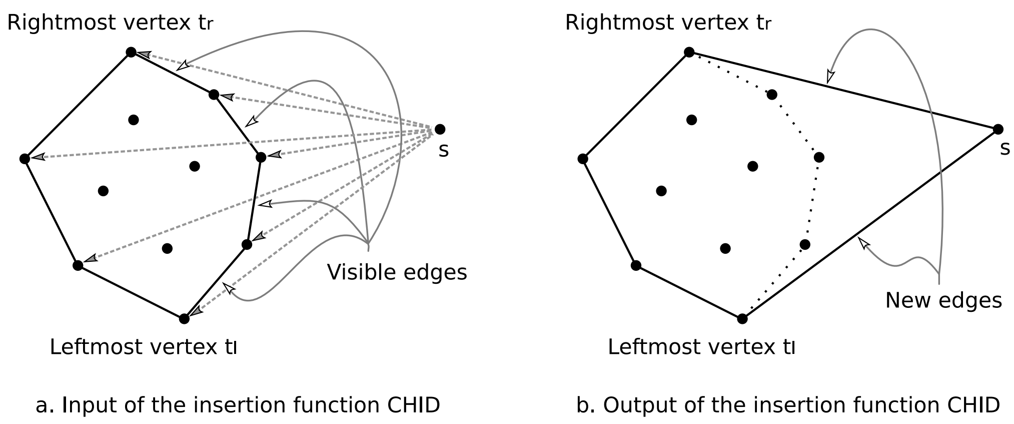


Figure 6: Computing a new convex hull from a convex polygon T and a new point s

6. Designing the incremental algorithm in Coq

6.1. Data representation

The *initial set of points* of the plane from which the convex hull is computed is represented as an object of type `fmap` which is constrained to be a combinatorial oriented map where each point is represented by an isolated linkless dart whose embedding is the point coordinates (see Fig. 7 (a)).

The final *convex hull* is a polygon represented as an object of type `fmap` which is constrained in order to be a combinatorial oriented map. Each polygon vertex is represented by a topologic vertex (two distinct darts with the same embedding linked at dimension `one`) and each edge is represented by a topologic edge (two distinct darts with different embedding linked at dimension `zero`). This is illustrated in Fig. 7 (b).

We shall see that all intermediate computations are also represented by combinatorial oriented maps possibly with isolated darts. Therefore, in the remainder of the paper, we shall only consider a subtype of objects of type `fmap` which are actually combinatorial oriented maps.

As the incremental computation of the convex hull relies on orientation tests in the plane, one must direct the polygon counter-clockwise. This is achieved by always linking darts in the same direction, links being represented by small arrows in the drawings. Furthermore, darts representing points which are inside the convex hull are kept in the final map where they are isolated non-linked darts (see Fig. 7 (b)). These darts can be erased if required.

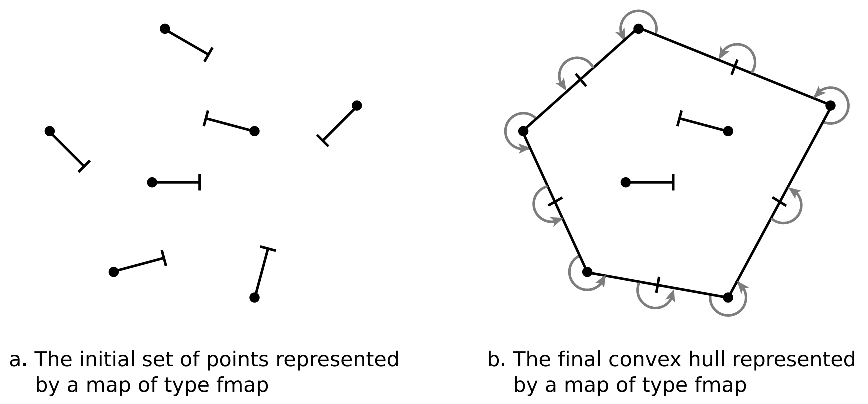

a. The initial set of points represented by a map of type fmap
b. The final convex hull represented by a map of type fmap

Figure 7: Representation of the input and output of the algorithm as maps of type `fmap`

6.2. Precondition

In the formalization, it is necessary to make the *precondition* the input map `m` must satisfy before the application of `CH` more precise. This precondition has four predicates and it is defined as follows:

```
Definition prec_CH (m:fmap) : Prop :=
  inv_hmap m /\ linkless m /\ well_emb m /\ noncollinear m.
```

The hypermap `m` of course has to verify the hypermap invariant `inv_hmap` which is explained in Section 3.2.3. It must have no link at all between darts (no `L` constructor), which is the property the predicate `linkless` expresses. The predicate `well_emb` expresses that the geometric embedding must be sound, i.e. all input darts must have different embeddings. The `well_emb` predicate captures this property although it also ensures some additional technical properties when links occur (see Section 9.2). In this first experiment, we assume no three darts having different embeddings can be embedded into three collinear points. For a map, the corresponding predicate `noncollinear` is specified as follows:

```
Definition noncollinear (m:fmap) : Prop :=
  forall (d1 d2 d3 : dart),
  let p1 := (fpoint m d1) in let p2 := (fpoint m d2) in
  let p3 := (fpoint m d3) in exd m d1 -> exd m d2 -> exd m d3 ->
  p1 <> p2 -> p1 <> p3 -> p2 <> p3 -> ~ collinear p1 p2 p3.
```

In the above definition, `fpoint m d` is the point onto which the dart `d` is embedded in the map `m`.

6.3. Classifying the darts

Following our implementation decisions to design the incremental algorithm, it appears one can classify darts into three different kinds. To make the characterization more visual, we choose to do it using three different colors (blue, red and black) depending on the links the darts are involved in. *Black darts* are isolated darts with no links at all. *Blue darts* are those with exactly one predecessor at dimension `one` and exactly one successor at dimension `zero`. *Red darts* are those with exactly one predecessor at dimension `zero` and exacly one successor at dimension `one`. Note that the meaning of these colors is completely different from those used in [30]. We remind the reader that in our definition of hypermaps in Coq, a given dart can not have more than one successor and one predecessor at each dimension. Our classification of darts is presented in Fig. 8 (which is also readable in black and white).

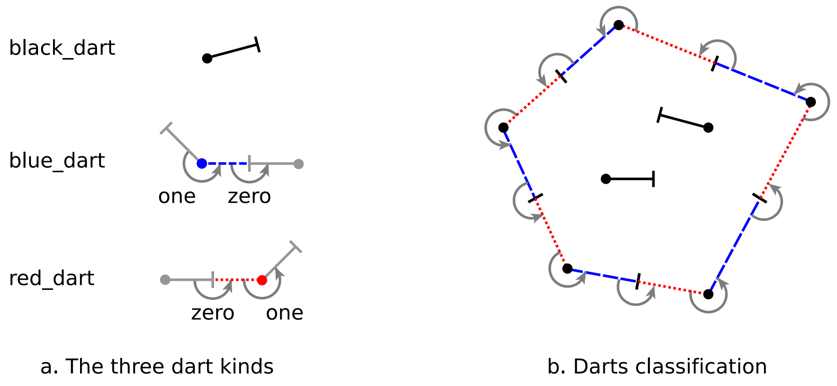


Figure 8: The three kinds of darts and their role in our description of the convex hull

In Coq, predicates `black_dart`, `blue_dart` and `red_dart` respectively express that a dart `x` is *black*, *blue* or *red* in a given map `m`:

```
Definition black_dart (m:fmap)(d:dart) : Prop :=
  ~ succ m zero d /\ ~ succ m one d /\
  ~ pred m zero d /\ ~ pred m one d.

Definition blue_dart (m:fmap)(d:dart) : Prop :=
  succ m zero d /\ ~ succ m one d /\
  ~ pred m zero d /\ pred m one d.

Definition red_dart (m:fmap)(d:dart) : Prop :=
  ~ succ m zero d /\ succ m one d /\
  pred m zero d /\ ~ pred m one d.
```

The three dart kinds appear in our description of the convex hull: *black darts* are drawn as full lines, *blue darts* as dashed lines and *red darts* as dotted lines (see Fig. 8 (b)). Their decidability is proved by the functions `black_dart_dec`, `blue_dart_dec`, `red_dart_dec` which can be used for branching in the code depending on the dart color.

6.4. Postconditions

We summarize the main properties we expect from our convex hull computation. Overall several topologic properties are required as well as the fundamental property that the built polygon is actually convex.

As far as geometric properties are concerned, we expect that the free map returned by the function `CH` actually verifies the *convex* property (see Definition 5). It relies on Knuth's orientation predicate `ccw` and can be transcripted into Coq using darts as follows:

```
Definition convex (m:fmap) : Prop := forall (x:dart)(y:dart),
  exd m x -> exd m y -> blue_dart m x ->
  let px := (fpoint m x) in let py := (fpoint m y) in
  let x0 := (A m zero x) in let px0 := (fpoint m x0) in
  px <> py -> px0 <> py -> ccw px px0 py.
```

6.5. Visible, leftmost and rightmost darts

We emphasized in Section 5.2 the role of the *visibility* of an edge from a point as well as the role of the leftmost and rightmost visible vertices. As we work with darts, the visibility of an edge is expressed on any one of its darts, and the leftmost and rightmost vertices are replaced by two darts, the actual leftmost one and the actual rightmost one (see Fig. 10 for a graphical description).

First, we define the predicates `visible` and `invisible` using the classification of darts and Knuth's orientation predicate `ccw`. The predicate `invisible` is exactly the negation of `visible`:

```
Definition visible (m:fmap)(d:dart)(p:point) : Prop :=
  if (blue_dart_dec m d)
  then (ccw (fpoint m d) p (fpoint m (A m zero d)))
  else (ccw (fpoint m (A_1 m zero d)) p (fpoint m d)).
```

As usual, decidability properties `visible_dec` and `invisible_dec` are proved. Then, following Definition 6, we specify two predicates `left_dart` and `right_dart` which state that a dart is the leftmost or the rightmost dart of `m` with respect to a point `p`:

```
Definition left_dart (m:fmap)(p:point)(d:dart) : Prop :=
  blue_dart m d /\ invisible m (A_1 m one d) p /\ visible m d p.

Definition right_dart (m:fmap)(p:point)(d:dart) : Prop :=
  red_dart m d /\ visible m d p /\ invisible m (A m one d) p.
```

These predicates decidability is proved by the two lemmas `left_dart_dec` and `right_dart_dec`. Note that, by convention, the leftmost dart always is a *blue* one and the rightmost dart is a *red* one. In addition, we shall have to prove the *equivalence of the existence* and the *uniqueness* of these two vertices. We will go back to these crucial questions in Section 9.1.

6.6. Programming the incremental algorithm with Coq

In this section, we write in Coq our incremental algorithm by structural recursion on free maps.

• We first define the main function `CH` which computes the whole convex hull of a finite set of points in the plane represented by a map `m`. If the initial map `m` is empty, it returns the empty map `V`. If `m` has only one dart, it returns a map with only one isolated dart. If it has at least two darts, it proceeds as follows: the function `CH` builds a *first convex polygon* for two of the involved darts using `CH2` (Fig. 9) and then calls the recursive function `CHI`. Since `CH` input is reduced to a dart set, no other case must be considered.

```
Definition CH (m:fmap) : fmap :=
  match m with
    V => V
  | I V x p => I V x p
  | I (I m0 x1 t1 p1) x2 t2 p2 =>
    CHI m0 (CH2 x1 p1 x2 p2 (max_dart m)) ((max_dart m)+3)
  | _ => V
  end.
```

Note that `max_dart m` returns the largest dart (in fact, darts are integers) of the map `m`. As we will see, this function helps to simulate the generation of *new* darts, i.e. darts which do not appear in `m`. Note that `CH2` uses two new darts (see above). Therefore the call to `CHI`, which also needs a new dart, is done with the parameter `(max_dart m)+3`.

• Given two distinct darts `x1` and `x2`, the function `CH2` builds the combinatorial oriented map shown in Fig. 9. To do that, it introduces two new darts, namely, `max+1` and `max+2`, and links them conveniently with `x1` and `x2`. Instead of having a simple edge as presented in Section 5.2, we actually have a flattened polygon (in Fig. 9, edges are curved for visibility reasons) consisting in four darts and their links. This allows us to handle the case of two points in the same way as the general one.

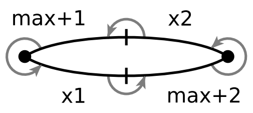


Figure 9: A convex hull of two darts built by the CH2 function

```
Definition CH2 (x1:dart)(p1:point)
  (x2:dart)(p2:point)(max:dart) : fmap :=
  let m0 := (I (I V x1 p1) x2 p2) in
  let m1 := L (I m0 (max+1) p1) one (max+1) x1 in
  let m2 := L (I m1 (max+2) p2) one (max+2) x2 in
    L (L m2 zero x1 (max+2)) zero x2 (max+1).
```

• Finally, function CHI takes the darts of m one-by-one and builds for each one a new convex hull using CHID and the parameter max. Then the recursive call of CHI is with parameter max+1.

```
Fixpoint CHI (m1:fmap)(m2:fmap)(max:dart) {struct m1} : fmap :=
  match m1 with
    V => m2
  | I m0 x p => CHI m0 (CHID m2 m2 x p max) (max+1)
  | _ => V
  end.
```

Now, we describe our function CHID.

6.7. A step of convex hull building

As already hinted in the previous section, function CHID computes the convex hull of a convex polygon represented by a map m (of type fmap) and a new point represented by a dart x. It works by structural recursion on m by studying each dart and each link separately. Darts are processed in random order (one dictated by the structure of the fmap term) while reconstructing the polygon (instead of traversing them in the sequential order dictated by the counter-clockwise traversal of the polygon). Because m is modified at each recursive call, CHID keeps a *reference* map mr, which is the same as m when CHID is first called. This reference map is useful to perform tests and will never be modified during the whole execution of the function. At each step of the computation, m is a *submap* of the reference map mr according to the following definition:

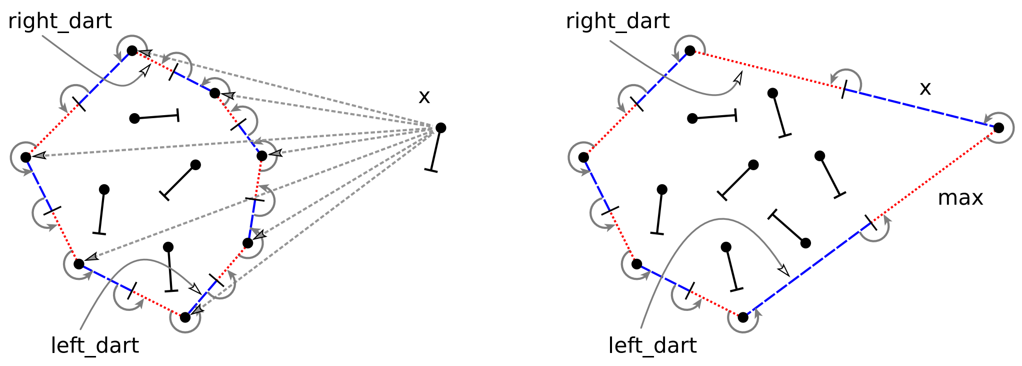


Figure 10: CHID behavior

```
Fixpoint submap (m:fmap)(mr:fmap) {struct m} : Prop :=
  match m with
    V => True
  | I m0 x p => submap m0 mr /\ exd mr x /\ (fpoint mr x) = p
  | L m0 k x y => submap m0 mr /\
    (A mr k x) = y /\ (A_1 mr k y) = x
  end.
```

Initially, m is equal to mr and at each recursive call, we formally prove in Coq the property `submap` m mr still holds.

As previously said, there are two cases in `CHID`. If the new point lies *inside* the convex polygon, the function `CHID` simply inserts the dart `x` into the map `m` without any links. If it lies *outside* the convex polygon, the function `CHID` removes the edges of the polygon which are visible from the new point and creates two new ones connecting it with the leftmost vertex and the rightmost vertex of the polygon.

In fact, `CHID` works *constructively* and not destructively: it always rebuilds from scratch the hypermap result by adding darts and their new links. If a link does not have to be reintegrated, it is quite simply forgotten. In this context, a recursive call to (`CHID m mr x p max`) unfolds as follows (see Fig. 11). Let us explain this in details:

• If `m` is the empty map (line 04), `CHID` simply returns the dart `x` with no links.

• If `m` matches (`I m0 x0 p0`) (line 05), `CHID` checks the dart kind of `x0` in `mr`.
- If `x0` is a *blue* dart in `mr` (line 06), the program tests whether `x0` belongs to an edge of `mr` which is invisible from the new dart `x` embedded into point `p` (line 07). This test is achieved using the predicate `invisible_dec`. If the edge of `x0` is invisible from `p` (line 08), the dart is simply kept in the map. Otherwise, the

```
01: Fixpoint CHID (m:fmap)(mr:fmap)(x:dart)(p:point)
02:   (max:dart) {struct m} : fmap :=
03:   match m with
04:     V => I V x p
05:   | I m0 x0 p0 =>
06:     if (blue_dart_dec mr x0) then
07:       if (invisible_dec mr x0 p) then
08:         (I (CHID m0 mr x p max) x0 p0)
09:       else if (left_dart_dec mr p x0) then
10:              (L (L (I (I (CHID m0 mr x p max) x0 p0)
11:              max p) one max x) zero x0 max)
12:            else (I (CHID m0 mr x p max) x0 p0)
13:     else if (red_dart_dec mr x0) then
14:            if (invisible_dec mr x0 p) then
15:              (I (CHID m0 mr x p max) x0 p0)
16:            else if (right_dart_dec mr p x0) then
17:            (L (I (CHID m0 mr x p max) x0 p0) zero x x0)
18:                 else (CHID m0 mr x p max)
19:          else (I (CHID m0 mr x p max) x0 p0)
20:   | L m0 zero x0 y0 =>
21:     if (invisible_dec mr x0 p) then
22:       (L (CHID m0 mr x p max) zero x0 y0)
23:     else (CHID m0 mr x p max)
24:   | L m0 one x0 y0 =>
25:     if (invisible_dec mr x0 p) then
26:       (L (CHID m0 mr x p max) one x0 y0)
27:     else if (invisible_dec mr y0 p) then
28:            (L (CHID m0 mr x p max) one x0 y0)
29:          else (CHID m0 mr x p max)
30:   end.
```

Figure 11: CHID function in Coq

program tests (line 09) whether `x0` is the new leftmost dart of `mr` with respect to `x`. If `x0` is the leftmost dart (line 10), it remains in the map. In addition, a new dart `max` (embedded into `p`) is inserted and linked to `x` at dimension `one`. Finally `x0` and `max` are linked at dimension `zero`. Otherwise, `x0` is simply kept in the map (line 12).
- If `x0` is a *red* dart in `mr`, a similar reasoning step is performed (lines 13-18).
- If `x0` is a *black* dart in `mr`, it is kept in the map (line 19).

• If `m` matches (`L m0 zero x0 y0`) (line 20), `CHID` tests whether the edge formed by `x0` and `y0` is invisible from `x` embedded into `p` (line 21). If it is invisible from `p`, the link at dimension `zero` between `x0` and `y0` is kept (line 22). Otherwise, it is not added again in the result map (line 23).

• Similar steps apply if `m` matches (`L m0 one x0 y0`) (lines 24-29).

7. Extracting our Coq program into OCaml

The Coq proof assistant features an extraction mechanism which automatically generates certified programs in OCaml or Haskell from proofs and specifications developped in Coq. It uses the Curry-Howard isomorphism between functional programming and natural deduction. This paradigm states that: "proof = program" and "proposition = type".

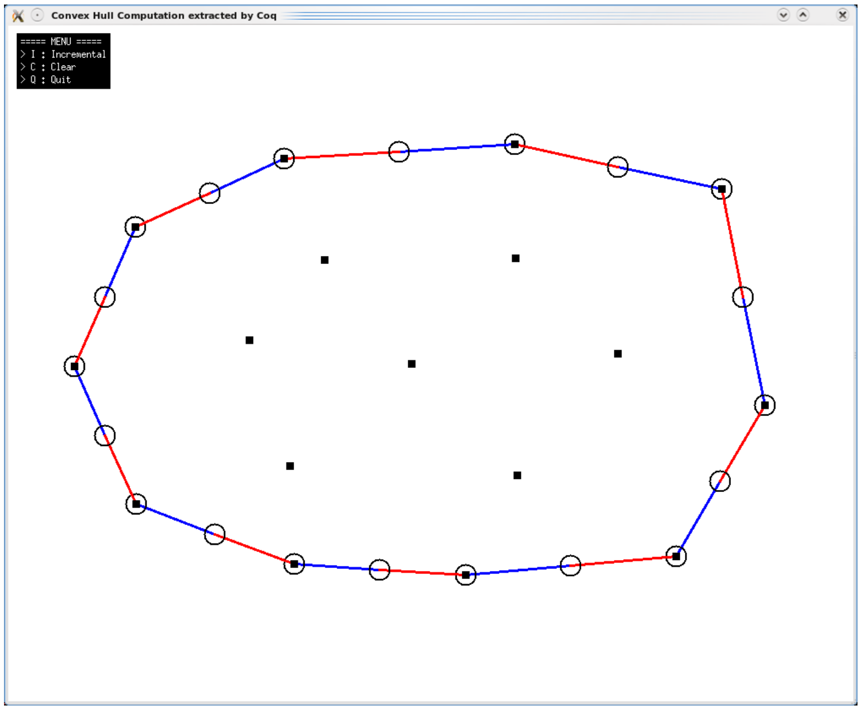


Figure 12: Graphical interface

We use this feature to extract an OCaml program which computes convex hulls from our specification of Fig. 11. Coq datatypes such as `fmap` are automatically extracted into standard OCaml datatypes. However, some basic definitions or axioms can be manually translated into OCaml terms when the extraction mechanism does not know how to translate them. We reproduce all the manual translation commands required below.

```
Extract Inductive sumbool => "bool" [ "true" "false" ].
Extract Constant R => "float".
Extract Constant R0 => "0.0".
Extract Constant R1 => "1.0".
Extract Constant Rplus => "fun x y -> x+.y".
Extract Constant Rmult => "fun x y -> x*.y".
Extract Constant Ropp => "fun x -> -.x".
Extract Constant total_order_T => "fun x y ->
  if (x<y) then (Inleft true) else
  if (x=y) then (Inleft false) else (Inright)".
```

We choose to map Coq real numbers into Ocaml floating-point numbers to be able to quickly extract our Coq implementation into a prototype program in Ocaml. However, we must underline that such a translation (which actually is an approximation) is unsound. Indeed, among other issues, adding floating-point numbers in Ocaml is not an associative operation. In addition, such a translation may lead to errors in the evaluation of the geometric predicates as underlined in [24]. A more sensible extraction we would use in the event we want to insert this proved-correct program into a modeler would be to consider rational numbers in Coq rather than real numbers. This shall be sufficient for our purposes and the extraction of rational numbers from their implementation in Coq into the one in Ocaml will be straightforward and more importantly, it would be safe.

The extracted program only contains the code of functions `CH`, `CHI`, `CH2`, and `CHID` which computes the convex hull. One then has to create a graphical interface in order to be able to select points of the plane, transform this input into a map, let the extracted function `CH` compute the convex hull and transform the resulting map into a polygonal line (together with some remaining isolated points inside) which can be displayed on the screen. For convenience, translation functions from lists of points to maps (`list_to_fmap`) and from Peano's integers to binary integers and vice-versa (`i2n` and `n2i`) are also provided:

```
let rec list_to_fmap l i : fmap =
  match l with
  | [] -> V
  | (x,y) :: l0 -> (I ((list_to_fmap l0 (i+1)),
    (i2n i), (Pair (float_of_int x, float_of_int y))));;

let rec i2n = function 0 -> O | n -> S (i2n (n-1));;
let rec n2i = function O -> 0 | S p -> 1+(n2i p);;
```

Fig. 12 presents a snapshot of the graphical interface. All links are symbolized with small circles, with a small dot inside for vertices and nothing for edges.

From our algorithm written in Coq, we manage to automatically derive a program which can actually run on a computer. The next step is to make sure the algorithm is correct, i.e. it really computes convex hulls. This consists in proving several topologic and geometric properties. Several consistency properties (e.g. free maps at stake are meaningful ones all the way) are required. We shall also prove that the output verifies the definition of convex hull given in Section 5.1.

8. Topologic properties

Initially, we decided to prove topologic properties first, and then focus on geometric ones. However, for some of the topologic properties, geometric properties inevitably interfere. We can not reason on topologic issues without taking into account some basic geometric facts. This section focuses on proofs of topologic properties, even though they do sometimes rely on geometric properties. Relevant topologic properties are the hypermap property invariant preservation, the property that the hypermap describing the convex hull (at some stage of the computation) is always a polygon, the preservation of the initial darts in the computed convex hull and the planarity property for the convex hull computed by the algorithm.

8.1. Dart kinds and their evolution throughout the algorithm

As presented in Section 6.3, it is possible to classify darts handled by our algorithm into three kinds (*blue*, *red*, or *black*). As it proceeds by structural induction on a map, the insertion function `CHID` considers darts one after another in random order. In addition, links are not studied at the same time as the darts they do link. Consequently, during recursive calls, not only some darts can shift from one kind to another but also darts may not belong to any kind anymore (e.g. when a dart loses only one of its links). They do end up in *intermediate states* during the execution of the insertion function `CHID`.

For example, *blue* darts can either lose their incoming link at dimension `zero`, or their outgoing link at dimension `one`. If both links are removed, then they actually become *black* darts.

To make proofs easier, especially in the proof of planarity (see Section 8.5), we define four intermediate kinds darts can belong to: namely `half_blue_succ` for a *half blue dart* with only a successor (at dimension `one`), `half_blue_pred` for a half blue dart with only a predecessor (at dimension `zero`):

```
Definition half_blue_succ (m:fmap)(d:dart) : Prop :=
  succ m zero d /\ ~ succ m one d /\
  ~ pred m zero d /\ ~ pred m one d.

Definition half_blue_pred (m:fmap)(d:dart) : Prop :=
  ~ succ m zero d /\ ~ succ m one d /\
  ~ pred m zero d /\ pred m one d.
```

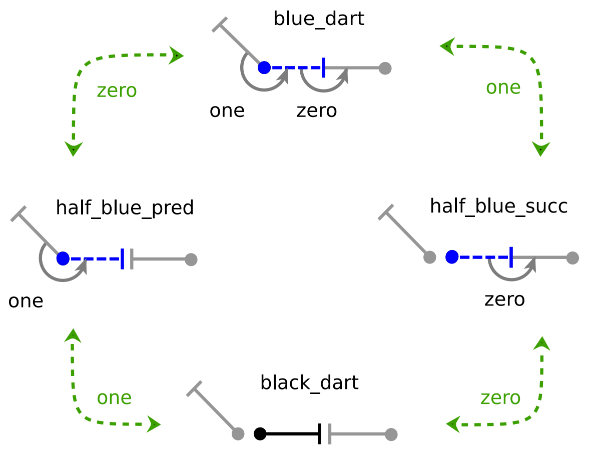


Figure 13: Possible changes of kinds for blue darts while CHID is executed

We can immediately transpose these definitions for red darts. These new kinds, representing intermediate states and possible changes of kinds for blue and half blue darts, are presented in Fig. 13. It works exactly the same for *red* darts as well. All these changes happening to the input combinatorial oriented map explain why we must have a reference map: the initial one, right before the first call to the insertion function (see Section 6.7 for details) to test the existence of darts and links between darts.

To consider all possible cases which can happen during recursive calls to this function, we establish 78 lemmas which express these changes in dart kinds. Here are two examples of such lemmas:

• The first lemma `blue_dart_CHID_1` expresses that, if a dart `d` is blue in the reference map `mr` and belongs to the current map `m`, then it remains in the result map `(CHID m mr x tx px max)`:

```
Lemma blue_dart_CHID_1 :
  forall (m mr:fmap)(x:dart)(px:point)(max:dart)(d:dart),
    blue_dart mr d -> exd m d -> exd (CHID m mr x px max) d.
```

• The second lemma `blue_dart_CHID_11` expresses that, if a dart `d`, different from the new dart `x`, is blue in the reference map `mr`, belongs to the current map `m`, and is visible from the point to be inserted `px` but that its predecessor at dimension `one` is not visible, then its successor at dimension `zero` in the map `(CHID m mr x tx px max)` is a new dart `max`:

```
  Lemma blue_dart_CHID_11 :
    forall (m mr:fmap)(x:dart)(px:point)(max:dart)(d:dart),
    submap m mr -> d <> x -> exd m d -> blue_dart mr d ->
    visible mr d px -> invisible mr (A_1 mr one d) px ->
    A (CHID m mr x px max) zero d = max.
```

8.2. Hypermap invariant preservation

The first important theorem we want to prove is that the invariant `inv_hmap` holds all the way from the initial map `m` to the final one `(CH m)`. This predicate `inv_hmap` states that darts must belong to the map before being linked together and one dart can not be inserted twice in the same map (see Section 3.2.3).

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

The proof proceeds by induction of the free map m and relies both on the lemmas of the previous section about the way darts may change kinds during the execution of the algorithm, and on the technical proofs of *uniqueness* of the leftmost and rightmost darts (see Section 9.1).

This illustrates that topologic properties do depend on geometric properties in geometric algorithms such as computing convex hulls. Indeed, the key property to establish the above-mentionned theorem is that whether a dart d is kept in the current combinatorial oriented map (topologic property) simply relies on whether it is visible or not with respect to the point being inserted into the convex hull (geometric property).

8.3. The convex hull is a topologic polygonal set

The predicate `inv_poly` on free maps expresses what a *topologic polygonal set* is. Informally, it consists of a set of polygons and isolated darts. In our algorithm, we expect to actually obtain a polygon together with some isolated darts. In a map verifying `inv_poly`, all darts are either *black*, *blue*, or *red* darts. No dart is partially linked. Therefore, for each dart d in the map, it either is *black* and isolated with no links or it is *blue* or *red*, meaning it belongs to the connected component which forms what we expect to be the convex hull.

```
Definition inv_poly (m:fmap) : Prop := forall (d:dart),
  exd m d -> black_dart m d \/ blue_dart m d \/ red_dart m d.
```

We prove that the free map returned by the function `CH` verifies the invariant `inv_poly` and therefore that it is a *topologic polygonal set.*

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

The proof proceeds the same way as in the proof of the property `inv_hmap` but also uses the *equivalence of the existence* of the leftmost and rightmost darts (see Section 9.1).

8.4. Initial darts preservation

Another fundamental property to prove is that darts which belong to the initial map do belong to the final map (denoting the convex hull) with their embeddings.

```
Theorem exd_CH : forall (m:fmap)(d:dart), prec_CH m ->
  exd m d -> exd (CH m) d /\ fpoint m d = fpoint (CH m) d.
```

In addition, only new (red) darts which are inserted during the convex hull computation can be removed from the map representing the final convex hull. According to `CHID` behavior, we note that darts extracted from the initial map and inserted into the convex hull are either *black* if they do lie inside the already-built convex hull, or *blue* if they do lie outside. Conversely, all *black* and *blue* darts of the resulting map are in the initial map. In addition, all *red* darts are new darts created using function `max_dart` (see Section 6.6). Then, to prove the initial darts are still present in the resulting map, it simply remains to be proved that *black* and *blue* darts are kept in the resulting map each time a new dart is inserted.

8.5. Planarity

So far, we proved that our algorithm eventually produces a polygon and some isolated darts. We must still formally verify that this resulting polygon is actually planar. The planarity property `planar` is defined as follows [12, 13]:

```
Definition planar (m:fmap) := genus m = 0.
```

We prove that, if m verifies the preconditions presented in Section 6.2, then the result of the incremental computation of the convex hull `(CH m)` is planar.

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

This proof uses the planarity criteria established in [12, 13] as well as the classification of the darts. One of the planarity characterizing lemmas is presented below:

```
Lemma planarity_crit_0 : forall (m:fmap)(x:dart)(y:dart),
  inv_hmap m -> prec_L m zero x y -> (planar (L m zero x y) <->
  (planar m /\ (~ eqc m x y \/ expf m (cA_1 m one x) y))).
```

The predicate `eqc` (resp. `expf`), proposed in [12], respectively express that two darts belong to the same connected component (resp. to the same face).

This lemma characterizes what is required for a free map `m` in which we link x to y at dimension `zero` to be planar. Such a free map will be planar if and only if the map `m` is planar and either x and y do not belong to the same connected components, or there exists a path in a face from the image of x by the closure function `cA_1` at dimension `one` to y. This characterization obviously requires some preconditions, namely that m verifies the `inv_hmap` property and that x and y verify the precondition for 0-linking two darts together (`prec_L`).

8.6. Connected components and face numbering

The last two topological properties we need to establish are that the number of connected components is equal to 1 and that the number of faces in the map returned by `CH` is equal to 2 (plus the number of isolated darts). Note this only holds when the initial map contains at least two darts. These two properties are stated using functions `nc` and `nf` computing the number of connected components and the number of faces of a map (see [12] for details). These two properties are shown to be equivalent to one another. However completing the proofs of these properties would be very tedious in Coq and is not done in this work.

```
Conjecture nc_1 : forall (m:fmap), prec_CH (m) ->
  nd (m) >= 2 -> nc (CH m) = 1 + nn (CH m).
```

This first property states that, as soon as the number of darts in a free map `m` is greater or equal to 2, the number of connected components in the computed convex hull `(CH m)` is equal to 1 plus the number of isolated darts (*black* darts which lie inside the convex hull). The proof would proceed by induction on the structure of the free maps and would lead to numerous and intricate cases to handle.

```
Conjecture nf_2 : forall (m:fmap), prec_CH (m) ->
  nd (m) >= 3 -> nf (CH m) = 2 + nn (CH m).
```

The second property states that, as soon as the number of darts in a free map `m` is greater or equal to 3, the number of faces in the computed convex hull `(CH m)` is equal to 2 (inside and outside) plus the number of isolated darts (black darts which lie inside the convex hull).

As said before, these two properties are equivalent thanks to Euler Formula. However neither of these two properties, nor the equivalence were formally proved in Coq. Indeed, such proofs are doable but very tedious. There are no theoritical issues involved but we would have to handle an exponential increase in the amount of cases to prove. The main difficulty is to do numbering of darts, vertices, edges, faces and connected components during the computations of the insertion function `CHID`. This would require to use the inductive definitions of the predicates `expf` and `eqc` at several levels in patterns of `CHID` like `(L (L (I (I (CHID m0 mr x t p max) x0 p0)`.

We now focus on the geometric properties required to prove that our algorithm actually computes a convex hull.

9. Geometric properties

Key geometric properties are that darts are embedded in a consistent way in the plane and that the computed free map is actually a convex hull.

9.1. Uniqueness and equivalence of existence of the leftmost and rightmost vertices

The first properties we establish are technical ones dealing with the uniqueness and the equivalence of the existence of two darts: the leftmost one and the rightmost one.

• To prove the *uniqueness*, one assumes there are two leftmost darts and then proves that they are equal. We use the convexity properties as well as visibility ones for both darts. This leads to six triples of darts whose orientation contradict either the property 5 or 5 bis of Knuth. The theorem for the leftmost dart is the following one:

```
Theorem one_left : forall (m:fmap)(p:point)(x:dart)(y:dart),
  inv_hmap m -> inv_poly m -> well_emb m ->
  inv_noncollinear_points m p -> convex m ->
  left_dart m p x -> left_dart m p y -> x = y.
```

A similar theorem is proved for the rightmost dart.

• The proof of the *equivalence of the existence* of the leftmost dart and the rightmost dart are expressed by the theorem `exd_left_right_dart` (and its reciprocal) which states that if a leftmost dart exists in m, then there also exists a rightmost dart in m. Note that both darts may not exist (when the considered dart lies inside the already constructed convex hull). Therefore, the strongest property we can show is that whenever one of the these two darts exist, then the other exists as well.

```
Theorem exd_left_dart_exd_right_dart :
  forall (m:fmap)(px:point), inv_hmap m -> inv_poly m ->
  (exists da:dart, exd m da /\ left_dart m px da) ->
  (exists db:dart, exd m db /\ right_dart m px db).
```

The proof proceeds by *iteration* on the darts belonging to the face. It relies on the property that the face is bounded, i.e. its number of darts is finite and can be computed which was proved using noetherian induction in [12]. Concretely, it means the number of iterations of `cF` on a dart `d` required to come back to dart `d` again is known in advance. One of the most significant lemmas required to prove the above-mentionned theorem is the following one:

```
Lemma blue_dart_not_right_dart_until_i_visible_i :
  forall (m:fmap)(d:dart)(p:point)(i:nat), inv_hmap m ->
  inv_poly m -> exd m d -> blue_dart m d -> visible m d p ->
  let iter0 := (A m zero (Iter (cF m) j d)) in
  (forall (j:nat), j <= i -> ~ right_dart m p iter0) ->
  visible m (Iter (cF m) i d) p.
```

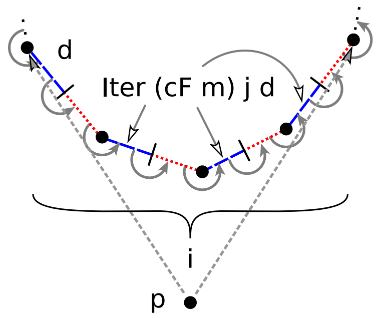


Figure 14: Face iteration from dart `d`: all traversed darts are visible from p

Note that the k-iterate of a function f from a dart d, $f^k(d)$ is written `Iter f k d` in our framework. This lemma states that if there exists a *blue* dart visible from the point `p` we want to insert, then as long as we move around the face and do not find the rightmost dart, all traversed darts are visible from p. This property is illustrated at Fig. 14.

9.2. Embedding

We prove that darts are well embedded with respect to their links. To do that, we first define the property `well_emb` which was already used in the precondition of the function `CH` in Section 6.2. For each dart in the hypermap, its embedding must be different from those of its successor and predecessor at dimension `zero` but the same that those of its successor and predecessor at dimension `one`. In addition, all other darts must have a different embedding:

```
Definition well_emb (m:fmap) : Prop :=
  forall (x:dart), exd m x -> let px := (fpoint m x) in
  let x0 := (A m zero x) in let px0 := (fpoint m x0) in
  let x1 := (A m one x) in let px1 := (fpoint m x1) in
  let x_0 := (A_1 m zero x) in let px_0 := (fpoint m x_0) in
  let x_1 := (A_1 m one x) in let px_1 := (fpoint m x_1) in
(succ m zero x -> px <> px0) /\ (succ m one x -> px = px1) /\
(pred m zero x -> px <> px_0) /\ (pred m one x -> px = px_1) /\
(forall y:dart, exd m y -> let py := (fpoint m y) in
  y <> x -> y <> x1 -> y <> x_1 -> px <> py).
```

This definition is illustrated in Fig. 15. On the left-hand side, we have a blue dart `x` with its 0-successor `x0` and its 1-predecessor `x_1`, and on the right-hand side, a red dart `x` with its 1-successor `x1` and its 0-predecessor `x_0`.

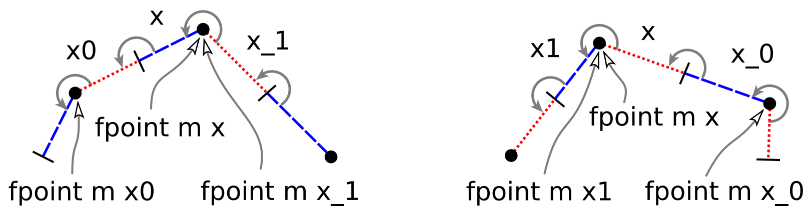


Figure 15: Correct embedding of the darts with respect to their links.

Then, we establish the theorem:

```
Theorem well_emb_CH : forall (m:fmap),
  prec_CH (m) -> well_emb (CH m).
```

To prove it, we proceed as usual using our classification of darts (see Section 8.1) and prove the darts do keep their embeddings during recursive calls to the insertion function `CHID`.

9.3. Convexity

Theorem `convex_CH` states the fundamental geometric property of the convex hull computation. It expresses that, provided the initial free map `m` verifies the preconditions, the final free map `(CH m)` is actually convex. More precisely, the final free map represents a polygon being the convex hull and some isolated darts inside this convex hull as well:

```
Theorem convex_CH : forall (m:fmap),
  prec_CH m -> convex (CH m).
```

The `convex` property was defined in Section 6.4. The proof of this theorem proceeds using the invariant property `well_emb` as well as properties of Knuth's orientation predicate.

10. Conclusion

This work is a first experiment to see how our ideas on designing and certifying geometric algorithms work. The specification of a convex hull computation algorithm constitutes some sort of *benchmark* to check whether our library on hypermaps and geometric predicates is adequate with respect to our specification and proof goals.

What we achieve is designing a functional algorithm and formally proving its total correctness with the Coq proof system. The termination of the algorithm is immediate because of its inductive construction. Properties justifying the partial correctness are all proved with the exception of the uniqueness of the polygonal convex hull, which remains complicated and shall be one of our next challenges. Fig. 16 provides some key figures about the size of the development and makes a distinction between the size of the specifications and the size of the proofs (the ratio is almost 1 to 10). The basic library corresponds to the already existing specifications and proofs presented in [12]. The amount of specifications and proofs developped for this formal proof of correctness of the incremental convex hull algorithm is summarized in the second column. The Coq files of the development are available online [5]. We manage to extract and make usable an OCaml program which, given a set of points in the plane, computes the convex hull (using the extracted code) and displays its result on the screen. This confirms the functional algorithm we designed is close to an actual implementation.

A worthwhile extension would be to derive an efficient program in a procedural language, where hypermaps would be represented with a concrete datatype such as linked lists, as in [3]. In the short term, this could be performed by hand, as we did for our image segmentation algorithm in [10]. In the longer term, one

	Basic library	Convex Hull
Number of definitions	142	52
Number of lemmas/theorems	550	409
Number of lines of specifications	3013	1620
Number of lines of formal proofs	28646	17902

Figure 16: Key figures about the size of our Coq development

expects to automate this process and formally prove the correctness of the actual program we use, as this was carried out in [2] for a square root computation algorithm for arbitrarily large numbers. In both cases, such *refinements* of programs should be studied by decomposing them into a sequence of elementary transformations.

A procedural implementation mimicking the strategy of our functional algorithm on a linked list would be really close to the classical incremental algorithm and as efficient as it, namely $O(n^2)$ in the worst case, n being the number of points in the input. One may object that it is still far from the optimal complexity which is $O(n.log(n))$, but any implementation of the incremental algorithm has this drawback.

The next step in our work is to study a variant of our incremental algorithm in Coq. In this variant, at each step when a new point is inserted, searching for the leftmost and rightmost darts would be performed by a traversal of a single face of the current polygon instead of studying darts in random order. The hypermap update can be performed by generating two new darts, unlinking of a few darts and relinking some others when the point we intend to add is *outside* the convex hull. Consequently, we would be very close to the common implementation of a convex hull incremental algorithm. In addition, all convex hull algorithms such as Graham scan or Jarvis march could be revisited using our library on hypermaps and their operations.

Switching to a three-dimensional setting with a polyhedral convex hull would then be another challenge. This means handling general polyhedral surfaces and at this stage, the use of hypermaps of dimension 2 is even more meaningful. Finally, other computational geometry algorithms must be investigated. Currently, one of the authors studies triangulations and an algorithm to build Delaunay diagrams.

Last but not least, even if our handling of geometric predicates using Knuth's orientation predicate is very convenient, we sooner or later must deal with numerical accuracy. This becomes a key issue in computational geometry as advocated in [24], which, for a large part, deals with an orientation predicate and an algorithm to compute a convex hull incrementally. More specifically, some efficient techniques to statically verify the validity of some numeric predicates in algorithms are developed using interval arithmetics and the Gappa tool [29]. This tool was successfully used in computational geometry for validating the orientation predicate. Our approach was relevant at the beginning of our investigations, but results on exact and/or lazy arithmetics and their formalizations (e.g. [23]) should be integrated in our formal development.

References

[1] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, An EATCS Series, Springer-Verlag, Berlin/Heidelberg, 2004, 469 pages.

[2] Y. Bertot, N. Magaud, P. Zimmermann, A Proof of GMP Square Root, Journal of Automated Reasoning 29 (3-4) (2002) 225–252, special Issue on Automating and Mechanising Mathematics: In honour of N.G. de Bruijn. An earlier version is available as a INRIA research report RR4475.

[3] Y. Bertrand, J.-F. Dufourd, Algebraic Specification of a 3D-modeler Based on Hypermaps, CVGIP: Graphical Models and Image Processing 56 (1) (1994) 29–60.

[4] J.-D. Boissonnat, M. Yvinec, Algorithmic Geometry, Cambridge University Press, 1998, 544 pages. Translated by Hervé Brönnimann.

[5] C. Brun, J.-F. Dufourd, N. Magaud, Designing and proving correct a convex hull algorithm with hypermaps in Coq : the formal development in Coq, available from http://galapagos.gforge.inria.fr (2010).

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

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms (Second Edition), The MIT Press, 2001, 1202 pages.

[8] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry, Algorithms and Applications (Third Edition), Springer-Verlag, Berlin/Heidelberg, 2008, 386 pages.

[9] J.-F. Dufourd, A hypermap framework for computer-aided proofs in surface subdivisions - Genus theorem and Euler formula, in: SAC'07: Proceedings of the 22nd ACM Symposium on Applied Computing, ACM Press, New York, NY, USA, 2007, pp. 757–761.

[10] J.-F. Dufourd, Design and formal proof of a new optimal image segmentation program with hypermaps, Pattern Recognition 40 (11) (2007) 2974–2993.

[11] J.-F. Dufourd, Discrete Jordan curve theorem: A proof formalized in Coq with hypermaps, in: STACS'08: Proceedings of the 25th International Symposium on Theoretical Aspects of Computer Science, 2008, pp. 253–264.

[12] J.-F. Dufourd, Polyhedra genus theorem and Euler formula: A hypermap-formalized intuitionistic proof, Theoretical Computer Science 403 (2-3) (2008) 133–159.

[13] 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) (2009) 19–51.

[14] H. Edelsbrunner, Algorithms in Combinatorial Geometry, Springer-Verlag, New York, NY, USA, 1987.

[15] J. Edmonds, A Combinatorial Representation for Polyhedral Surfaces, Notices American Mathematical Society 7 (1960) .

[16] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, E. Ezra, The Design and Implementation of Planar Maps in CGAL, in: WAE'99: Proceedings of the 3rd International Workshop on Algorithm Engineering, vol. 1668 of Lecture Notes in Computer Science, Springer-Verlag, London, UK, 1999, pp. 154–168.

[17] G. Gonthier, A Computer-Checked Proof of the Four Colour Theorem, Technical report, Microsoft Research, Cambridge (2005).

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

[19] G. Gonthier, A. Mahboubi, A Small Scale Reflection Extension for the Coq system, Tech. Rep. RR-6455, INRIA (2008).
URL `http://hal.inria.fr/inria-00258384/`

[20] L. J. Guibas, J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, in: STOC'83: Proceedings of the fifteenth annual ACM symposium on Theory of computing, ACM, New York, NY, USA, 1983, pp. 221–234.

[21] G. Huet, G. Kahn, C. Paulin-Mohring, The Coq Proof Assistant - A Tutorial, INRIA, France, 2007, version 8.1. `http://coq.inria.fr/V8.1/files/doc/Tutorial.pdf`.

[22] A. Jacques, Constellations et graphes topologiques, Combinatorial Theory and Applications 2 (1970) 657–673.

[23] N. Julien, Certified Exact Real Arithmetic Using Co-induction in Arbitrary Integer Base, in: FLOPS, 2008, pp. 48–63.

[24] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, C. Yap, Classroom Examples of Robustness Problems in Geometric Computations, Computational Geometry: Theory and Applications (CGTA) 40 (1) (2008) 61–78.

[25] D. Knuth, Axioms and Hulls, vol. 606 of Lecture Notes in Computer Science, Springer-Verlag, Berlin/Heidelberg, 1992, 109 pages.

[26] P. Lienhardt, Topological Models for Boundary Representation: a Comparison with n-Dimensional Generalized Maps, Computer-Aided Design 23 (1) (1991) 59–82.

[27] M. Mantyla, R. Sulonen, GWB: A Solid Modeler with Euler Operators, IEEE Computer Graphics and Applications 2 (7) (1982) 17–31.

[28] L. Meikle, J. Fleuriot, Mechanical Theorem Proving in Computational Geometry, in: H. Hong, D. Wang (eds.), Automated Deduction in Geometry, vol. 3763 of Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, 2006, pp. 1–18, 5th International Workshop, ADG 2004, Gainesville, FL, USA, September 16-18, 2004.

[29] G. Melquiond, Gappa : Génération Automatique de Preuves de Propriétés Arithmétiques, INRIA, France, 2010, version 0.12.3. `http://gappa.gforge.inria.fr`.

[30] D. Pichardie, Y. Bertot, Formalizing Convex Hull Algorithms, in: R. J. Boulton, P. B. Jackson (eds.), Theorem Proving in Higher Order Logics, vol. 2152 of Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, 2001, pp. 346–361, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001.

[31] F. P. Preparata, M. I. Shamos, Computational Geometry : An Introduction (5th printing), Monographs in Computer Science, Springer-Verlag, New York, 1993, 398 pages.

[32] R. Sedgewick, Algorithms, Addison-Wesley Publishing Company, 1983, 552 pages.

[33] The Coq Development Team, The Coq Proof Assistant - Reference Manual, INRIA, France, 2009, version 8.2. `http://www.lix.polytechnique.fr/coq/distrib/current/refman/`.

[34] W. Tutte, Combinatorial Oriented Maps, Canadian J. Math. 31 (5) (1979) 986–1004.

[35] K. Weiler, Edge-based Data Structures for Solid Modeling in Curved-surface Environments, IEEE Computer Graphics and Applications 5 (1) (1985) 21–40.