N° d'ordre : 836

UNIVERSITÉ DE STRASBOURG

THÈSE

présentée par

Christophe BRUN

pour l'obtention du grade de

Docteur de l'Université de Strasbourg
Discipline : Sciences et Technologies
Spécialité : Informatique

Preuves formelles pour le calcul d'enveloppes convexes dans le plan avec des hypercartes

---

Soutenue le **6 décembre 2010** devant la commission d'examen composée de :

Mme Catherine DUBOIS ..... *Rapportrice externe*
Professeure à l'ENSIIE d'Evry

M. Dominique MICHELUCCI ..... *Rapporteur externe*
Professeur à l'Université de Bourgogne

M. Yves BERTOT ..... *Examinateur externe*
Directeur de recherche à l'INRIA Sophia-Antipolis

M. Mohamed TAJINE ..... *Examinateur interne*
Professeur à l'Université de Strasbourg

M. Jean-François DUFOURD ..... *Directeur de thèse*
Professeur à l'Université de Strasbourg

M. Nicolas MAGAUD ..... *Co-encadrant*
Maître de conférences à l'Université de Strasbourg

⇝ **L**aboratoire des **S**ciences de l'**I**mage, de l'**I**nformatique et de la **T**élédétection (UMR CNRS-UdS 7005)
⇝ **E**cole **D**octorale « **M**athématiques, **S**ciences de l'**I**nformation et de l'**I**ngénieur »

Remerciements

Je tiens tout d'abord à remercier très chaleureusement mon directeur de thèse, M. Jean-François Dufourd, et mon co-encadrant, M. Nicolas Magaud, de m'avoir accordé leur confiance en me proposant ce sujet de recherche et de m'avoir ainsi permis de réaliser mon souhait de mener une thèse de doctorat en informatique à Strasbourg dans le domaine des preuves et des spécifications formelles en géométrie algorithmique.

Ils ont toujours été présents pour m'apporter leur aide précieuse et leurs conseils éclairés. D'ailleurs, ces travaux de recherche et les publications qui en résultent n'auraient probablement pu aboutir sans leur collaboration tenace, leur assistance soutenue et leur détermination à toute épreuve. Ils m'ont pour ainsi dire porté à bout de bras durant ces 3 années de doctorat et ils ont constitué pour moi un appui indispensable dans l'établissement de cette thèse. J'aimerais pour tout cela, encore une fois, leur exprimer ma plus profonde gratitude et mes plus sincères remerciements.

Je souhaite également remercier M. Yves Bertot, directeur de recherche à l'INRIA Sophia-Antipolis, Mme Catherine Dubois, professeure à l'ENSIIE d'Evry, M. Dominique Michelucci, professeur à l'Université de Bourgogne, et M. Mohamed Tajine, professeur à l'Université de Strasbourg, pour avoir bien voulu examiner mes travaux de recherche et m'avoir fait l'honneur de participer à mon jury de thèse. Merci tout particulièrement à Catherine et Dominique d'avoir accepté d'être les rapporteurs de ce mémoire.

Je voudrais aussi adresser mes remerciements à l'ensemble de mes collègues de l'équipe IGG qui, de près ou de loin, m'ont aidé par leurs compétences et leur sympathie. J'ai une pensée particulière pour mes compagnons de doctorat Thomas, Simon et Lucas ainsi que pour Fred, Pierre, Julien, Olivier et les deux Pascal que j'ai eu la chance de côtoyer. J'adresse tout spécialement un grand merci à Sylvain qui m'a été d'un grand secours tout au long de ma thèse et qui a un peu contribué à la réalisation de ce travail.

Et pour finir, je tiens à remercier de tout coeur mes amis et ma famille pour leurs encouragements et leur bienveillance. Cette thèse représente finalement l'aboutissement de nombreuses années d'études durant lesquelles j'ai pu bénéficier d'un soutien et d'un réconfort sans faille auprès de mes parents et de mon frère. Merci papa, maman et Nico.

Table des matières

Chapitre 1

Introduction

Notre objectif est de mener une étude formelle dans le domaine de la modélisation et de la géométrie algorithmique dans le plan afin d'améliorer les techniques de programmation et d'assurer la correction des algorithmes. Nous avons conduit une étude de cas en géométrie algorithmique sur un problème classique mettant en jeu des subdivisions de surfaces simples puisque réduites à des lignes polygonales : le calcul incrémental de l'enveloppe convexe d'un ensemble fini de points du plan.

Pour cela, nous nous appuyons d'une part sur les spécifications et les preuves formelles de programmes qui sont exprimées dans le formalisme du calcul des constructions inductives mis en oeuvre dans le système d'aide à la preuve Coq, et d'autre part sur un cadre de modélisation géométrique à base topologique où les subdivisions du plan sont représentées par des cartes combinatoires orientées. Cependant, afin d'être plus général pour des applications ultérieures, nous utilisons d'abord des hypercartes que nous spécialisons par la suite en cartes combinatoires orientées.

1.1 Contexte et objectif

La géométrie algorithmique [6, 10, 44] a pour but la conception et l'analyse d'algorithmes permettant de résoudre des problèmes de nature géométrique. Elle se préoccupe à la fois de leurs aspects théoriques et pratiques. Cette discipline trouve des applications dans de nombreux domaines tels que la modélisation géométrique, la robotique, l'imagerie médicale, les systèmes d'informations géographiques, etc. Parmi les différents problèmes explorés par celle-ci, le calcul de l'enveloppe convexe est certainement celui qui a le plus été étudié. Il est à la base de nombreux autres algorithmes tels que la triangulation de Delaunay [11].

Notre travail de spécifications et de démonstrations formelles n'est pas fait dans le seul but de prouver simplement un algorithme de plus en géométrie algorithmique mais

bien pour poser les fondements et les principes de base permettant d'obtenir un cadre commun de raisonnement et d'acquérir une certaine pratique de développement. Nous nous attaquons ainsi à un algorithme naïf de calcul d'enveloppe convexe en nous intéressant davantage à sa certification qu'à sa complexité. Nous nous contentons d'avoir une stratégie de calcul très simple mais pas forcément efficace.

Coq [3, 48, 29, 42] est un assistant de preuve développé par l'INRIA. Il est fondé sur une théorie des types d'ordre supérieur. Il permet de définir des fonctions ou des prédicats, d'énoncer des théorèmes ou des spécifications de programmes, d'en développer interactivement des preuves formelles au moyen de tactiques et de faire valider celles-ci par un noyau de vérification de typage. Le développement d'un algorithme dans l'environnement de preuve de Coq offre la possibilité de construire une démonstration formelle de l'adéquation de cet algorithme avec sa spécification, c'est-à-dire de certifier sa correction. De plus, à partir d'une preuve constructive, ce système permet d'extraire automatiquement un programme de construction en OCaml selon le paradigme "preuve = programme". Ce programme est nécessairement correct par rapport à sa spécification.

Les subdivisions du plan sont modélisées à l'aide d'hypercartes [9]. C'est une structure algébrique fondée sur les notions de brins et de permutations. Elle permet de bien séparer les aspects topologiques et géométriques des objets étudiés, de facilement définir les cellules d'une subdivision (i.e. ses sommets, arêtes, faces, composantes connexes) et d'en caractériser les relations d'incidence et d'adjacence. Cette structure de données peut se décrire aisément par induction structurelle. Nous nous basons sur une bibliothèque [18, 19] dans laquelle les hypercartes sont définies comme un type inductif avec des contraintes.

Les aspects géométriques que nous considérons sont particulièrement simples mais fondamentaux en géométrie algorithmique. Les plongements sont élémentaires et les subdivisions du plan (sommets en points, arêtes en segments de droites) sont représentées comme des frontières polygonales. Cependant, la question d'orientation dans le plan est cruciale. Dans notre modélisation, elle est traitée en utilisant le système d'axiomes de Knuth [33]. Ce système définit l'orientation d'un triplet de points selon l'ordre dans lequel il est énuméré dans le plan (soit dans le sens trigonométrique soit dans le sens inverse). Un de ses avantages est de permettre d'isoler les tests numériques et, dans un premier temps, d'éluder les problèmes numériques difficiles. En effet, nous ne nous intéressons pas à cette question dans ce travail qui s'occupe plutôt de la conformité des structures de données et de leurs relations apparentées. Les nombres réels sont idéalisés en utilisant le système d'axiomes fourni dans la bibliothèque de Coq.

Nous abordons deux approches de calcul différentes mais en adoptant toujours la même méthodologie : conception d'un algorithme en Coq, extraction automatique d'un programme exécutable en OCaml, preuves formelles par la mise en évidence de plusieurs propriétés topologiques et géométriques, et, éventuellement, dérivation d'un programme en langage impératif. Dans la deuxième partie, nous décrivons un algorithme incrémental de calcul d'enveloppe convexe par induction structurelle sur le type inductif `fmap` des hypercartes qui analyse séparément chaque brin et chaque liaison en les examinant dans un ordre dicté par la structure du terme de la carte. Dans la troisième partie, nous choisis-

sons une version de plus haut niveau, qui se veut aussi plus traditionnelle, en adoptant le principe de recherche par voisinage en se déplaçant le long de l'enveloppe convexe à partir d'un brin référence.

1.2 Travaux connexes

1.2.1 Enveloppe convexe et modélisation de subdivisions du plan

L'enveloppe convexe d'un ensemble fini de points est un des concepts majeurs de la géométrie algorithmique. Plusieurs définitions sont répertoriées dans la littérature, ainsi que diverses méthodes de constructions, comme l'algorithme incrémental, la marche de Jarvis ou le parcours de Graham [6, 10, 11, 21, 44, 46].

En 2D, l'enveloppe convexe est un polygone, mais sa construction nécessite de savoir manipuler des lignes brisées et parfois plusieurs polygones, comme dans l'algorithme par dichotomie. D'une manière générale, même si elles sont simples, il s'agit, comme dans la plupart des algorithmes géométriques, de subdivisions irrégulières de surfaces en sommets, arêtes et faces, qu'il faut savoir manipuler. C'est pourquoi, dans notre travail, nous attachons la plus grande importance à cette notion.

Aujourd'hui, on considère qu'une bonne manière d'appréhender les objets géométriques subdivisés est de distinguer leur topologie et leur plongement géométrique [1, 45]. La topologie, de nature combinatoire, peut être décrite par une structure de données concrète, comme les *demi-arêtes* (*half-edges*) [50], les *arêtes ailées* (*winged-edges*) [2, 36] ou celle des *quad-edges* [28].

Comme partout en informatique, on préfère aujourd'hui avoir une vision plus abstraite. La notion de *carte combinatoire orientée* de dimension 2 [49] permet de décrire algébriquement des subdivisions générales de surfaces orientables fermées, ce qui répond bien à notre besoin pour les enveloppes convexes. Cependant, nous préférons partir avec des *hypercartes combinatoires* [9] parce qu'elles sont plus générales, homogènes selon les 2 dimensions, et faciles à contraindre ensuite selon les besoins. Ainsi, nous pourrons plus tard spécialiser nos hypercartes pour appréhender des modèles de subdivisions plus complexes.

Les cartes combinatoires et leurs extensions ont fait l'objet d'études théoriques approfondies, notamment avec des spécifications formelles sous forme de spécifications algébriques [5] ou à base de modèle avec B [14]. Elles sont à la base de multiples développements théoriques et pratiques en modélisation géométrique [5, 22, 24, 30, 35, 40]. Elles font d'ailleurs l'objet de deux modules spécialisés de la bibliothèque de référence en géométrie algorithmique CGAL [23].

Pour formaliser le concept d'enveloppe convexe et sa construction, nous avons besoin de plonger géométriquement les hypercartes dans le plan euclidien et d'y exprimer l'orientation. Comme classiquement, nos plongements sont juste des plongements de sommets en

des points du plan, le reste étant obtenu par linéarisation. Alors, nous nous appuyons sur l'approche axiomatique du calcul géométrique et de l'orientation présentée par Knuth dans sa monographie intitulée "Axioms and Hulls" [33]. Basée sur les propriétés d'orientation des triplets de points dans l'espace euclidien, elle permet de bien séparer les tests logiques des problèmes numériques et elle est particulièrement bien adaptée aux preuves formelles.

1.2.2 Preuves formelles en géométrie algorithmique

Comme cela a été indiqué dans [18], plusieurs travaux en géométrie algorithmique, portant notamment sur des questions d'axiomatique, ont déjà été menés, mais nous n'en parlerons pas ici. Des travaux de preuves formelles s'intéressant aux algorithmes de calcul d'enveloppes convexes ont aussi été conduits. Ainsi, Pichardie et Bertot prouvent en Coq la correction de l'algorithme incrémental et la méthode dite du paquet-cadeau (ou marche de Jarvis) [43]. Ils ont été les premiers à utiliser l'axiomatique de Knuth mais ils représentent les enveloppes convexes comme des listes de points ce qui entraîne quelques complications et surtout risque de bloquer les extensions futures, notamment en 3D. Meikle et Fleuriot prouvent à leur tour en Isabelle la correction de l'algorithme de Graham en s'appuyant sur la logique de Hoare [38], ce qui éloigne d'une description fonctionnelle simple.

Les travaux cités ci-dessus n'utilisent pas de structure topologique. Cependant, les hypercartes ont déjà été utilisées de manière magistrale comme représentation des subdivisions planaires dans la formalisation et la preuve en Coq du théorème des 4 couleurs par Gonthier et al. [25, 26]. On y propose une spécification permettant d'obtenir des résultats remarquables comme un critère de planarité (théorème de Jordan) qui est à la base de la preuve du théorème des 4 couleurs. Cependant, la méthode de spécification proposée, ainsi que la technique de preuves par *réflexion* [27] sont assez éloignées de ce que nous proposons ici. On trouvera une comparaison approfondie dans [19].

À l'Université de Strasbourg, la bibliothèque de spécifications d'hypercartes sur laquelle s'appuie notre travail sur l'enveloppe convexe a permis de démontrer quelques grands résultats comme le théorème du genre, la formule d'Euler pour les polyèdres [18] et une version d'un théorème de Jordan discret [17, 19]. Elle est aussi à la base de la certification d'un algorithme fonctionnel de segmentation d'images par fusion de faces adjacentes et du développement d'un programme en C optimal en temps [16]. Parallèlement à ce travail, Dufourd et Bertot étudient comment prouver la correction d'un algorithme de calcul de la triangulation de Delaunay [20].

1.3 Plan du mémoire

La suite du manuscrit est organisée comme suit.

La première partie présente les modèles à base topologique de représentation des subdivisions du plan et l'environnement géométrique dans lequel nous travaillons. Le chapitre 2

donne la description mathématique et la spécification en Coq des hypercartes et des cartes combinatoires orientées. Le chapitre 3 introduit le prédicat d'orientation et l'axiomatisation de Knuth, il présente le concept d'enveloppe convexe et son calcul par l'algorithme incrémental, et il explique notre représentation des données.

La deuxième partie est consacrée à notre première expérience de preuves formelles d'un algorithme de calcul d'enveloppe convexe. Le chapitre 4 décrit la structure originale de notre programme de construction incrémentale qui travaille par induction structurelle sur le type inductif des hypercartes. Le chapitre 5 expose le mécanisme d'extraction automatique de Coq que nous avons utilisé pour produire un programme utilisable en OCaml. Les chapitres 6 et 7 s'intéressent respectivement aux propriétés topologiques et géométriques que nous avons réussi à démontrer.

La troisième partie présente notre deuxième approche de certification d'un algorithme de calcul d'enveloppe convexe. Le chapitre 8 détaille le fonctionnement de ce second algorithme qui adopte un processus de calcul plus traditionnel de recherche par voisinage. Les chapitres 9 et 10 exhibent les preuves topologiques et géométriques que nous avons pu établir sur cet algorithme. Le chapitre 11 expose notre dérivation en langage impératif de cet algorithme par l'utilisation de la bibliothèque CGoGN.

Enfin, nous résumons dans un chapitre de conclusion le bilan de nos travaux sur la preuve formelle d'algorithmes de calcul d'enveloppe convexe dans le plan et nous proposons des perspectives de travaux futurs concernant la certification d'autres algorithmes de calcul d'enveloppe convexe ou d'autres programmes en géométrie algorithmique.

Première partie

Modélisation topologique et environnement géométrique

Sommaire

Chapitre 2

Hypercarte et carte combinatoire orientée

Dans ce chapitre, nous introduisons les notions d'hypercartes et de cartes combinatoires orientées de dimension 2 utilisées pour modéliser les subdivisions du plan et représenter les objets des calculs intermédiaires et le résultat de l'enveloppe convexe. Nous commençons par donner les définitions mathématiques puis nous expliquons comment formaliser celles-ci en Coq.

2.1 Descriptions mathématiques

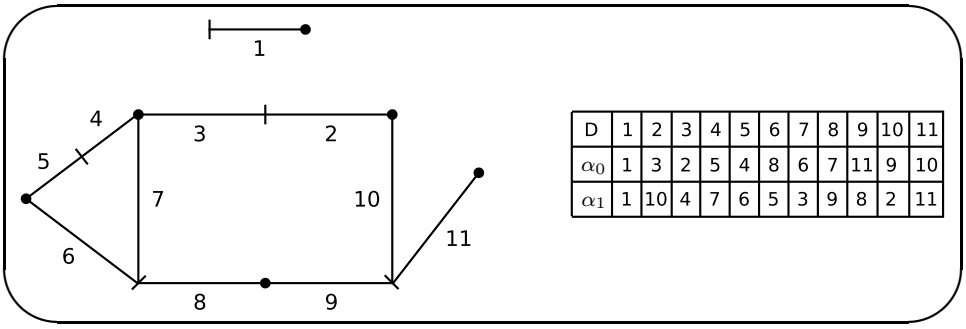
2.1.1 Définitions fondamentales

Les hypercartes sont une des structures les plus générales pour décrire topologiquement les subdivisions finies de surfaces en termes de sommets, d'arêtes, de faces et de composantes connexes. Nous les modélisons par une structure algébrique fondée sur les notions de brins et de permutations.

Définition 1 (Hypercarte et carte combinatoire orientée).
1. *Une* hypercarte *est une structure algébrique* $M = (D, \alpha_0, \alpha_1)$*, où* D *est un ensemble fini dont les éléments sont appelés des* brins*, et* α_0*,* α_1 *sont des permutations sur* D*.*
2. *Quand* α_0 *est une involution sur* D *(i.e.* $\forall x \in D$*,* $\alpha_0(\alpha_0(x)) = x$*),* M *est appelée une* carte combinatoire orientée*.*
3. *Pour chaque dimension* $k \in \{0, 1\}$ *: si* $y = \alpha_k(x)$*,* y *est le* k-successeur *de* x*,* x *est le* k-prédécesseur *de* y*, et* x *et* y *sont dits* k-liés *ensemble.*

Le brin est l'élément de base pour la définition d'une hypercarte. Il est sémantiquement associé à une demi-arête orientée. Les fonctions α_i correspondent à la liaison de brins pour

chaque dimension i. Les cartes combinatoires orientées sont une sous-classe des hypercartes. La notion d'hypercarte est bien adaptée pour accomplir des preuves formelles en géométrie algorithmique [26]. Cependant, les cartes combinatoires orientées sont beaucoup plus faciles à utiliser en modélisation géométrique [22, 23, 30, 35, 49].



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 2.1 - Un exemple d'hypercarte

Exemple 1 : *Dans la figure 2.1, comme les fonctions α_0 et α_1 sont des permutations sur $D = \{1, \ldots, 11\}$, $M = (D, \alpha_0, \alpha_1)$ est une hypercarte. Elle représente une subdivision du plan avec un triangle et un rectangle adjacent l'un à l'autre, un segment de ligne attaché et un segment isolé.*

Dans nos dessins d'hypercartes sur des surfaces, nous représentons chaque brin comme un simple segment de droite orienté d'une petite boule vers un petit trait : les brins 0-liés (resp. 1-liés) partagent le même trait (resp. boule). Nous adoptons toujours la convention que les k-successeurs tournent dans le sens trigonométrique dans le plan autour des boules et des traits. Notons que notre définition d'hypercarte nous permet de créer la carte vide (i.e. $D = \emptyset$) et d'avoir des points fixes par rapport à α_k.

2.1.2 Cellules topologiques

Les cellules topologiques d'une hypercarte (i.e. ses sommets, arêtes, faces et composantes connexes) ne sont pas représentées explicitement dans le modèle. Elles sont définies combinatoirement par des sous-ensembles de brins obtenus essentiellement à travers la classique notion d'orbite.

Définition 2 (Orbite et cellules topologiques).

1. *Soit f une permutation sur un ensemble fini D. L'*orbite *de $x \in D$ pour f est la séquence de brins $\langle f \rangle(x) = (x, f(x), f^2(x), \ldots, f^{p-1}(x))$, où p, appelée la* période *de l'orbite, est le plus petit entier tel que $f^p(x) = x$.*
2. *Dans l'hypercarte $M = (D, \alpha_0, \alpha_1)$, $\langle \alpha_0 \rangle(x)$ représente la* 0-orbite *ou l'*arête *du brin x, $\langle \alpha_1 \rangle(x)$ son* 1-orbite *ou* sommet, *et $\langle \phi \rangle(x)$ sa* face, *pour $\phi = \alpha_1^{-1} \circ \alpha_0^{-1}$. La* composante connexe *de x dans M, denotée par $\langle \alpha_0, \alpha_1 \rangle(x)$, correspond à l'ensemble des brins accessibles depuis x par n'importe quelle composition de séquence sur α_0 et α_1.*

Les faces sont définies à travers $\phi = \alpha_1^{-1} \circ \alpha_0^{-1}$ pour des brins parcourus dans le sens trigonométrique, quand l'hypercarte est dessinée sur une surface. Alors, en adoptant la convention de tracé dans le plan telle que les permutations doivent tournées dans le sens trigonométrique, chaque face qui entoure une région bornée (resp. non-bornée) à sa gauche est appelée *interne* (resp. *externe*).

Exemple 2 : *Dans la figure 2.1, l'hypercarte M contient* 5 *arêtes (traits),* 6 *sommets (boulettes),* 4 *faces et* 2 *composantes connexes. Par exemple, $\langle\alpha_0\rangle(7) = (7, 6, 8)$ est l'arête du brin* 7*, $\langle\alpha_1\rangle(7) = (7, 3, 4)$ son sommet. Concernant ϕ qui est égal à $\alpha_1^{-1} \circ \alpha_0^{-1}$, nous avons $\phi(2) = 7$, $\phi(7) = 9$ et $\phi(9) = 2$. Alors la face (interne) de* 2 *est $\langle\phi\rangle(2) = (2, 7, 9)$. De plus, la face (externe) de* 3 *est $\langle\phi\rangle(3) = (3, 10, 11, 8, 5)$.*

Comme α_0 et α_1 sont des permutations, il est clair que, pour $\Pi = \langle\alpha_0\rangle, \langle\alpha_1\rangle, \langle\alpha_1^{-1} \circ \alpha_0^{-1}\rangle$ ou $\langle\alpha_0, \alpha_1\rangle$, $y \in \langle\Pi(x)\rangle$ est équivalent à $x \in \langle\Pi(y)\rangle$. Dans une carte combinatoire orientée, chaque arête est composée d'un ou de deux brins maximum. Il est courant en modélisation géométrique de représenter des subdivisions de surfaces orientables [22, 30, 49, 35, 23]. Nous adopterons cette approche dans la suite de notre travail.

2.1.3 Planarité et formule d'Euler

Comme Jacques [30] et Tutte [49], nous avons choisi d'adopter une définition purement combinatoire de la caractéristique d'Euler χ et du genre. Ainsi, la définition de χ sur laquelle nous nous appuyions est plus générale que celle habituellement donnée du fait des k-points fixes possibles de M.

Soient d, e, v, f et c le nombre de brins, d'arêtes, de sommets, de faces et de composantes connexes d'une hypercarte $M = (D, \alpha_0, \alpha_1)$.

Définition 3 (Caractéristique d'Euler, genre et planarité).
1. *La* caractéristique d'Euler *de M est $\chi = v + e + f - d$.*
2. *Le* genre *de M est $g = c - \chi/2$.*
3. *Si $g = 0$, M est dite* planaire.

En fait, dans une carte combinatoire orientée non vide qui ne possède pas de point fixe par α_0, nous avons bien $d = 2 \times e$ et $\chi = v - e + f$. Notons que la caractéristique d'Euler peut être négative. Nous pouvons nous demander quelles sont les propriétés de χ et si le genre est toujours un entier. La réponse est donnée par le théorème du genre, dont une preuve formelle se trouve dans [15] et [18].

Théorème 1 (du genre).
1. *χ est un entier pair.*
2. *g est un entier naturel.*

Exemple 3 : *Dans la figure 2.1, la caractéristique d'Euler de M est $\chi = 6+5+4-11 = 4$*

et son genre $g = 2 - \chi/2 = 0$. *Par conséquent, l'hypercarte* M *est planaire.*

Une hypercarte planaire vérifie la propriété $\chi = 2 \times c$ qui est une généralisation de la bien connue *formule d'Euler*.

2.1.4 Plongement

Une hypercarte ne décrivant que la topologie d'une subdivision, il faut aussi définir un modèle de plongement pour décrire la géométrie des objets. Nous considérons ici seulement les plongements pour les cartes combinatoires orientées. Pour cette classe d'hypercartes, l'opération la plus simple consiste à associer un sommet topologique avec un point du plan, une arête avec un segment de droite connectant deux points (i.e. deux sommets plongés), et une face comme une possible région ouverte dans le plan. Ainsi, les plongements des arêtes et des faces d'une carte combinatoire orientée sont obtenus par interpolation linéaire du plongement de ses sommets.

2.2 Spécifications en Coq

Coq [48] est l'implémentation du calcul des constructions inductives qui est une théorie des types ainsi qu'un puissant système de logique intuitionniste conçu pour formaliser et prouver des propriétés mathématiques de manière interactive. Nous nous dispensons d'en faire une présentation exhaustive et nous renvoyons le lecteur intéressé à l'introduction pragmatique à l'écriture de preuves et de programmes certifiés fourni dans [3] ou au tutorial de référence qui se trouve dans [29]. Toutes les définitions de la section précédente sont formalisées dans ce cadre.

Nous reprenons ici les bases de la spécification des cartes présentée dans [18] et [19].

2.2.1 Définitions préliminaires

Nous définissons d'abord un type `dim` pour les dimensions 0 et 1 sur les hypercartes. C'est un type inductif avec deux constructeurs `zero` et `one` :

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

Notons qu'en Coq, tous les objets sont typés. `Prop` est le type des propositions qui correspond au « raisonnement », `Set` est le type des types concrets tels que `nat` pour les entiers naturels ou `bool` pour les booléens qui correspond au « calcul », et `Type` est le type des arités qui correspond au « type de tous les types ». De plus, le symbole `~` représente la

négation logique, le symbole `+` ou `\/` représente la disjonction, et le symbole `/\` représente la conjonction.

La logique de Coq étant intuitionniste, pour chaque type inductif, le prédicat générique d'égalité `=` est prédéfini mais sa décidabilité ne l'est pas. On déclare alors, pour chaque nouveau type `A` créé, un lemme qui s'écrit, pour tout couple (`x`,`y`) d'éléments de `A`, comme la disjonction de leur égalité et de leur différence. Par convention, ce lemme est énoncé en utilisant la somme disjointe : ∀ `(x:A)(y:A), {x=y}+{~x=y}`. Pour `dim`, la décidabilité de son égalité peut être établie par le lemme suivant :

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

La preuve de ce lemme est une fonction, nommée `eq_dim_dec`, qui teste si ses deux arguments sont égaux ou non. Cette forme permet d'obtenir un objet de type concret qui peut être utilisé dans une expression conditionnelle (if ... then ... else ...). Ce lemme est prouvé interactivement grâce à quelques tactiques. Le raisonnement est une simple induction structurelle sur `i` et `j`, qui se ramène à un simple cas d'analyse. En fait, à partir de chaque définition de type inductif, Coq génère un *principe d'induction*, utilisable pour prouver des propositions ou pour construire des fonctions totales sur les types.

Ensuite, nous définissons le type `dart` des brins avec le type prédéfini des nombres naturels `nat` et la décidabilité de son égalité `eq_dart_dec` est une réécriture de `eq_nat_dec`. Pour gérer les exceptions, un brin `nil` est un renommage de `0` :

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

Nous choisissons un point de vue constructif pour les hypercartes qui est approché par la construction incrémentale usuelle des subdivisions de surfaces en modélisation géométrique, comme dans [41], plutôt que de considérer un point de vue observationnel avec un ensemble de brins déjà construit équipé de toutes ses permutations, comme cela est fait dans [26].

2.2.2 Cartes libres

Dans un premier temps, nous définissons un type de *carte libre* grâce à une algèbre de termes de type inductif `fmap` avec trois constructeurs, `V`, `I` et `L`, respectivement pour la carte *vide* (ou *void*), l'*insertion* d'un brin, et la *liaison* de deux brins :

```
Inductive fmap : Set :=
  V : fmap
```

```
| I : fmap -> dart -> point -> fmap
| L : fmap -> dim -> dart -> dart -> fmap.
```

Les brins sont insérés dans une carte libre avec un plongement `point` qui est un couple de nombres réels. Comme nous pouvons le voir dans la figure 2.2, certaines propriétés géométriques cohérentes doivent être forcées. Par exemple, deux brins formant un même sommet (i.e. deux brins liés ensemble à la dimension `one`) doivent avoir un plongement identique. Dans la figure 2.2, les points `p2` et `p10`, respectivement associés aux brins `2` et `10`, doivent être égaux.

Par la suite, l'utilisation de ces constructeurs sera contraint par des préconditions pour éviter d'avoir des cartes libres dépourvues de sens. Le type correspondant aux hypercartes sera un sous-type du type `fmap` au sens où l'ensemble des hypercartes sera inclus dans l'ensemble des `fmap`. Il sera caractérisé par un invariant, appelé `inv_hmap`, systématiquement utilisé dans les conjonctions avec `fmap` (cf. section 2.2.4 pour les détails).

Exemple 4 : *L'hypercarte M de la figure 2.1 peut être modélisée par une carte libre représentée à la figure 2.2 où les* 0- *et* 1-*liens par* `L` *sont représentés par des arcs de cercles fléchés et où les orbites demeurent ouvertes. Par exemple, une sous-carte de l'hypercarte M de la figure 2.2, composée des brins* $2, 3, 9$ *et* 10*, est représentée par le terme en Coq suivant :* `(L (L (L (I (I (I (I V 3 p3) 2 p2) 10 p10) 9 p9) zero 2 3) one 10 2) zero 10 9)`.

Nous définissons ensuite des *observateurs* sur les cartes libres.

Le prédicat `exd` teste si un brin `d` existe dans une carte `m`. Sa définition est récursive, ce qui est indiqué par le mot-clé `Fixpoint`. Elle procède par filtrage sur `m`, ce qui s'écrit `match m with ... end`. L'attribut `{struct m}` permet à Coq de vérifier que les appels récursifs sont effectués sur des termes `fmap` plus petits et donc d'assurer la terminaison de la fonction. Le résultat est `True` ou `False`, les deux constantes basiques de `Prop`, le type prédéfini des propositions. Notons que les termes ont une notation préfixée et que le symbole `_` dénote la place d'un argument inutilisé. Prouver la décidabilité `exd_dec` de `exd` est immédiat par induction sur `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.
```

La fonction `A` renvoie le successeur à la dimension `k` d'un brin `d` dans une carte `m`. C'est une restriction de l'opération α_k car elle est construite pour fonctionner avec des cartes libres qui n'ont pas forcement leurs orbites closes. De ce fait, comme Coq ne permet que de définir des fonctions totales, elle est étendue avec le brin `nil` (l'inverse `A_1` est similaire) :

```
Fixpoint A (m:fmap)(k:dim)(d:dart) {struct m} : brin :=
  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.
```

Les prédicats `succ` et `pred` expriment qu'un brin `d` a un `k`-successeur et un `k`-prédécesseur (non-`nil`) dans une carte `m`, avec les propriétés de décidabilité `succ_dec` et `pred_dec` :

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

Definition pred (m:fmap)(k:dim)(d:dart) : Prop := A_1 m k d <> nil.
```

Exemple 5 : *Dans la figure 2.2, `A m zero 6 = 8`, `A m zero 4 = nil`, `succ m zero 6`, `~ succ m zero 4`, `A_1 m one 9 = 8`, et `pred m one 9`.*

2.2.3 Opérations de clôture

En fait, à l'aide d'opérations nommées `cA` et `cA_1`, il est toujours possible de considérer une carte libre comme une carte équipée des opérations α_k. En effet, les opérations `cA` et `cA_1` *ferment* `A` et `A_1`. Ainsi, nous pouvons procéder comme si les k-orbites étaient fermées. De plus, pour tout k, (`A m k`) et (`cA m k`) prolongent les fonctions α_k aux brins qui n'appartiennent pas à la carte `m` et retournent le brin `nil`. Les deux opérations `cA` et `cA_1` sont définies récursivement l'une de l'autre :

```
Fixpoint cA (m:fmap)(k:dim)(z:dart) {struct m} : dart :=
  match m with
     V => nil
  |  I m0 x _ =>
     if eq_dart_dec x z then z
     else cA m0 k z
  |  L m0 k0 x y  =>
     if (eq_dim_dec k0 k) then
       if (eq_dart_dec x z) then y
       else
         if (eq_dart_dec (cA_1 m0 k y) z) then cA m0 k x
         else cA m0 k z
     else cA m0 k z
```

```
    end
  with cA_1 (m:fmap)(k:dim)(z:dart) {struct m} : dart :=
    match m with [...] end.
```

Exemple 6 : *Dans la figure 2.2,* `cA m one 4 = 7, cA_1 m one 7 = 4, cA m one 11 = 11, cA_1 m one 11 = 11.` *De plus, quand les brins passés en entrée n'appartiennent pas à la carte,* `cA` *et* `cA_1` *renvoient* `nil`*. Ainsi,* `cA m zero 12 = nil` *et* `cA_1 m zero 12 = nil`*.*

Les propriétés fondamentales que nous avons à prouver sont que, pour tout `m` et `k`, `(A m k)` et `(A_1 m k)` sont des *injections* inverses l'une de l'autre, que `(cA m k)` et `(cA_1 m k)` sont des *permutations* inverses l'une de l'autre, et qu'elles sont closes. Elles se démontrent facilement par induction sur la carte `m`. Pour plus de détails techniques, le lecteur est invité à se référer à notre développement formel [8].

```
Lemma A_1_A : forall (m:fmap)(k:dim)(z:dart),
  inv_hmap m -> succ m k z -> A_1 m k (A m k z) = z.
```

```
Lemma cA_1_cA : forall (m:fmap)(k:dim)(z:dart),
  inv_hmap m -> exd m z -> cA_1 m k (cA m k z) = z.
```

Finalement, le parcours de faces est basé sur une fonction `F` et sa clôture `cF` qui correspond à ϕ telle que définie à la définition 2. Des propriétés similaires à `A` et `cA` sont prouvées pour `F`, `cF` et leurs inverses `F_1`, `cF_1`.

```
Definition F (m:fmap)(z:dart) := A_1 m one (A_1 m zero z).
```

```
Definition cF (m:fmap)(z:dart) := cA_1 m one (cA_1 m zero z).
```

Exemple 7 : *Dans la figure 2.2,* `F m 4 = nil, cF m 4 = 6.`

2.2.4 Hypercartes

Comme cela a été dit précédemment, des préconditions, écrites comme des prédicats, sont introduites pour les opérateurs `I` et `L`. La précondition `prec_I` pour `I` exprime que le brin `nil` ne peut pas être inséré dans une carte et qu'un brin `x` ne peut être inséré que s'il n'existe pas déjà dans cette carte. La précondition `prec_L` pour `L` vérifie que les brins `x` et `y` que nous voulons lier ensemble sont bien présents dans la carte, que `x` n'a pas de successeur à la dimension concernée et que `y` n'a pas de prédécesseur à cette dimension non plus. Finalement, `prec_L` interdit aussi d'insérer un lien de `x` à `y` si celui-ci clôt l'orbite de `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.
```

Si une carte est construite en respectant ces préconditions sur les opérations d'insertion `I` et de liaison `L`, elle satisfait l'*invariant* `inv_hmap`. Elle est facilement interprétable comme une hypercarte qui possède obligatoirement des *orbites ouvertes*. Une telle condition permet de fusionner plus facilement les orbites par concaténation. Cela réduit aussi le nombre de liens nécessaires dans la construction d'une enveloppe convexe. Une telle hypercarte est présentée à la figure 2.2.

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

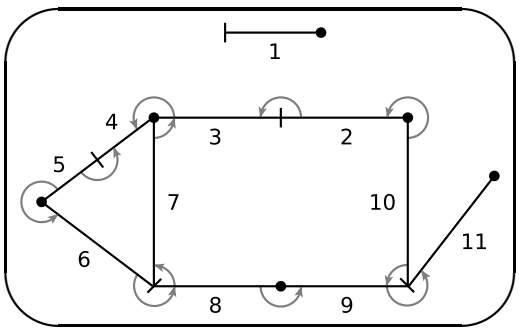


Figure 2.2 - Une hypercarte avec ses orbites incomplètement closes

2.2.5 Fonctions `Merge` et `Split`

Dans les sections précédentes, nous avons décrit les constructeurs basiques `I` et `L` des hypercartes et les diverses manières que nous avons pour observer les cartes et leurs composants. Maintenant, nous étudions des opérations de plus haut niveau pour manipuler les cartes, à savoir des opérations de fusion et de scission, appelées respectivement `Merge` et `Split`.

Décalage de l'ouverture dans une k-orbite

Etant donné que les k-orbites doivent rester ouvertes dans la structure des cartes, il est parfois nécessaire de déplacer l'ouverture dans une k-orbite par décalage de celle-ci.

Cette opération de mouvement, appelée `Shift`, n'est appelée que si un brin `x` possède un k-successeur. S'il en a un alors elle recherche les brins `t` et `b` dans la k-orbite tels que `t` n'a pas de k-successeur et `b` pas de k-prédécesseur. Ces deux brins particuliers représentent ce que l'on appelle respectivement le *top* et le *bottom* de cette k-orbite. La fonction `Shift` supprime ensuite le lien commençant en `x` à l'aide de l'opération `B` et elle ajoute celui liant `t` à `b`. Notons que cette opération ne change pas la k-orbite mais décale simplement l'ouverture dans celle-ci. L'opération `Shift` est illustrée à la figure 2.3.

```
Definition Shift (m:fmap)(k:dim)(x:dart) :=
  L (B m k x) k (top m k x) (bottom m k x).
```

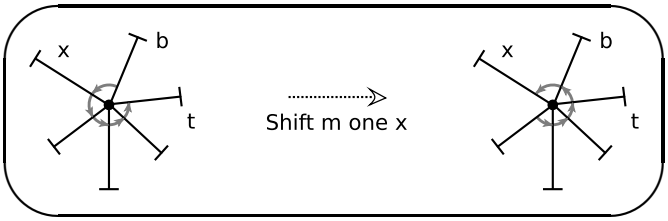


Figure 2.3 - Opération de décalage de l'ouverture dans une orbite

Opération de scission

Par cette opération, une k-orbite est séparée en deux morceaux. Avant la séparation, seul un brin de la k-orbite ne possède pas de k-successeur. Après celle-ci, ce sont deux des brins issus de la k-orbite qui n'en possèdent pas. L'opération de scission est spécifiée par la déclaration des deux brins ayant cette propriété. Ces deux brins sont appelés `x` et `y` dans la suite de ce paragraphe.

L'opération de scission, décrite pour toute dimension `k`, teste d'abord si `x` a un `k`-successeur. S'il n'en a pas, alors elle supprime simplement le lien débutant en `y`. Sinon, elle teste si `y` possède un `k`-successeur. S'il en a un, alors elle déplace l'ouverture dans l'orbite par l'opération `Shift` pour la placer devant `x` et elle supprime le lien reliant `y` à son successeur. Sinon, elle supprime le lien débutant en `x`. La précondition pour cette opération de scission est que `x` et `y` doivent être différents mais dans la même k-orbite. Dans notre développement formel, cette opération est décrite par une fonction nommée `Split`. Quelques propriétés importantes ont été prouvées sur celle-ci, comme par exemple qu'elle préserve la planarité et qu'elle est commutative pour `x` et `y` [20]. L'opération `Split` est illustrée à la figure 2.4.

```
Definition Split (m:fmap)(k:dim)(x:dart)(y:dart) : fmap :=
  if succ_dec m k x
  then if succ_dec m k y
       then B (Shift m k x) k y
       else B m k x
  else B m k y.
```

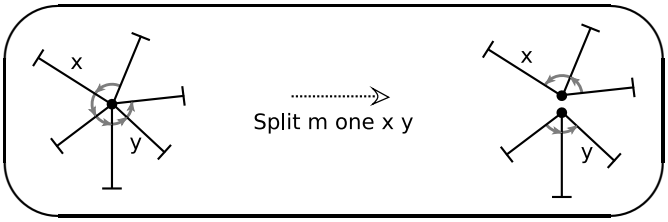


Figure 2.4 - Opération de scission `Split`

Opération de fusion

Pour fusionner deux k-orbites, nous devons choisir un brin `x` dans la première et un brin `y` dans la deuxième, tels que le k-successeur de `x` sera `y` dans la nouvelle carte. Naturellement, une première étape consiste à s'assurer que l'ouverture dans chaque k-orbite est bien placée, ce qui peut être fait à l'aide de la fonction `Shift` de manière à ce que `x` ne possède pas de successeur et `y` pas de prédécesseur avant l'ajout d'un lien de `x` à `y`. Cette opération de fusion, nommée `Merge`, a une précondition qui impose que `x` et `y` ne sont pas dans la même k-orbite. L'opération `Merge` est illustrée à la figure 2.5.

```
Definition Merge (m:fmap)(k:dim)(x:dart)(y:dart) : fmap :=
  let m1 := if succ_dec m k x then Shift m k x else m in
  let m2 := if pred_dec m1 k y then Shift m1 k (cA_1 m1 k y) else m1 in
  L m2 k x y.
```

Notons que le plongement des brins est ignoré dans les opérations `Shift`, `Split` et `Merge`. Il est en fait tout simplement conservé par ces fonctions qui ne le modifient pas.

Plus loin, des propriétés topologiques devront être considérées pour prouver la correction de notre algorithme de calcul d'enveloppe convexe. De plus, des invariants s'occupant de la géométrie seront à définir. Nous nous intéressons maintenant à l'environnement géométrique dans lequel se place cette étude.

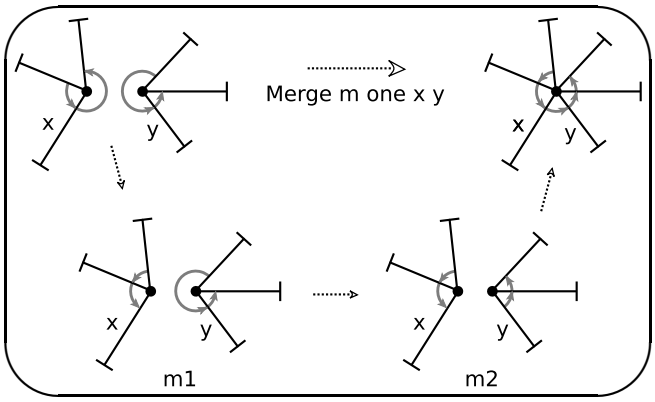


Figure 2.5 - Opération de fusion `Merge`

Chapitre 3

Enveloppe convexe et cadre géométrique

Le calcul de l'enveloppe convexe ne dépend pas que de la topologie mais il dépend aussi de propriétés géométriques sur les points impliqués. Nous avons choisi de travailler en géométrie euclidienne à deux dimensions et nous considérons chaque point p comme un couple de réels qui représente ses coordonnées dans le plan (i.e. $p = (x_p, y_p)$ avec $x_p, y_p \in \mathbb{R}$). Pour calculer incrémentalement une enveloppe convexe, nous avons besoin d'un prédicat pour déterminer l'orientation d'un triplet de points dans le plan.

Comme dans [38] et [43], nous suivons l'approche de Knuth [33] pour manipuler l'orientation dans le plan. Nous spécifions d'abord le prédicat d'orientation avec ses propriétés, puis nous l'implantons en représentant le plan par $\mathbb{R}^2$.

3.1 Axiomatisation du prédicat d'orientation

Dans son livre [33], Knuth présente une approche axiomatique du calcul géométrique qui permet la conception d'algorithmes basés sur des primitives simples vérifiant un ensemble fini d'axiomes. Cette approche offre la possibilité d'éluder les problèmes de robustesse en distinguant d'une part les prédicats (tests booléens qui conditionnent le déroulement de l'algorithme) et d'autre part les calculs arithmétiques du plongement géométrique de la structure combinatoire résultante. Cette axiomatisation isole les calculs numériques en les supposant effectués de manière sûre et exacte, et elle permet de se focaliser principalement sur les aspects algorithmiques du calcul de l'enveloppe convexe.

Le prédicat géométrique d'orientation *ccw* (counterclockwise) de Knuth prend en entrée les coordonnées d'un triplet de points du plan et indique si le triplet est orienté dans

le sens trigonométrique ou dans le sens inverse. Il est utilisé par l'algorithme incrémental pour calculer l'enveloppe convexe d'un ensemble fini de points du plan et il nous sera utile pour la spécification de l'enveloppe convexe.

La figure 3.1 illustre le prédicat géométrique d'orientation *ccw* pour un triplet de points $\widehat{pqr}$. A gauche, le triplet $\widehat{pqr}$ est orienté dans le sens trigonométrique, au centre, les trois points sont alignés, et à droite, le triplet est orienté dans le sens des aiguilles d'une montre.

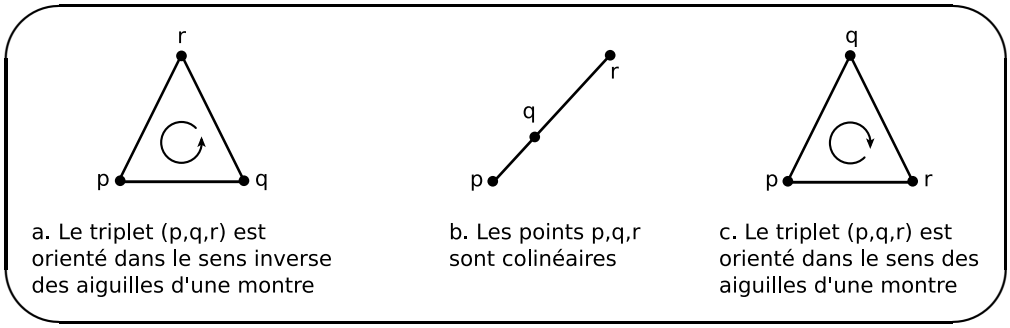


Figure 3.1 - Le prédicat d'orientation

Knuth a choisi de ne pas traiter les cas dégénérés. Il suppose que les points sont placés dans une *configuration générale*, c'est-à-dire qu'ils sont tous distincts deux à deux et qu'il n'y a jamais trois points alignés. Ainsi, dans notre implantation, pour tout triplet de points $\widehat{pqr}$ du plan, le prédicat géométrique d'orientation $ccw(p, q, r)$ est toujours défini puisque le déterminant $det(p, q, r)$ ne peut pas être nul.

Knuth propose ainsi six axiomes pour décrire les propriétés attendues du prédicat géométrique d'orientation *ccw* afin d'obtenir un algorithme qui fonctionne correctement sans erreur pour des configurations de points en position générale. La propriété 1 exprime la *cyclicité* du prédicat géométrique d'orientation *ccw*, la propriété 2 exprime son *antisymétrie*, la propriété 3 exprime sa *non-dégénérescence*, la propriété 4 exprime son *intériorité* et la propriété 5 exprime sa *transitivité*. La propriété 5 bis est une variante de la propriété 5. La figure 3.2 illustre les propriétés 4, 5 et 5 bis où les traits en pointillés représentent les hypothèses et les traits pleins représentent les conclusions.

Propriété 1 (Prédicat géométrique d'orientation)
P.1 : $\forall\ p, q, r,\ ccw(p, q, r) \Rightarrow ccw(q, r, p)$.
P.2 : $\forall\ p, q, r,\ ccw(p, q, r) \Rightarrow \neg\ ccw(p, r, q)$.
P.3 : $\forall\ p, q, r,\ p \neq q \Rightarrow q \neq r \Rightarrow p \neq r \Rightarrow ccw(p, q, r) \vee ccw(p, r, q)$.
P.4 : $\forall\ p, q, r, t,\ ccw(t, q, r) \wedge ccw(p, t, r) \wedge ccw(p, q, t) \Rightarrow ccw(p, q, r)$.
P.5 : $\forall\ p, q, r, s, t,\ ccw(t, s, p) \wedge ccw(t, s, q) \wedge ccw(t, s, r) \wedge ccw(t, p, q) \wedge ccw(t, q, r) \Rightarrow ccw(t, p, r)$.
P.5 bis : $\forall\ p, q, r, s, t,\ ccw(s, t, p) \wedge ccw(s, t, q) \wedge ccw(s, t, r) \wedge ccw(t, p, q) \wedge ccw(t, q, r) \Rightarrow$

$ccw(t, p, r)$.

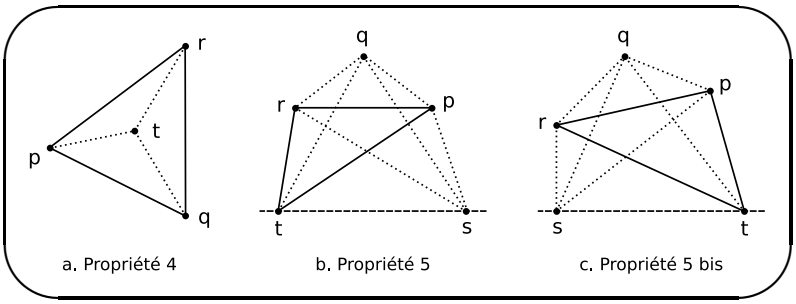


Figure 3.2 - Les propriétés 4, 5 et 5 bis du prédicat d'orientation *ccw* de Knuth

3.2 Modélisation du prédicat d'orientation en Coq

Notre implémentation utilise la définition de *ccw* suivante :

Définition 4 (Orientation d'un triplet de points du plan).
Soient p, q et r trois points du plan dont les coordonnées réelles sont respectivement (x_p, y_p), (x_q, y_q) *et* (x_r, y_r). *L'orientation du triplet de points* $\widehat{pqr}$ *est donnée par le signe du déterminant* $det(p, q, r)$. *Ainsi, si* $det(p, q, r) > 0$, *le triplet* $\widehat{pqr}$ *est orienté dans le sens trigonométrique, alors que, si* $det(p, q, r) < 0$, *le triplet* $\widehat{pqr}$ *est orienté dans le sens inverse.*

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

Les nombres réels sont décrits en Coq en utilisant un système d'axiomes [48]. Les opérations basiques (`+`, `-`, $\times$, `/`) sont spécifiées et leurs propriétés sont dérivées de cette spécification abstraite. Ainsi, la fonction `det` peut être facilement implémentée comme suit :

```
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).
```

De cette définition, nous dérivons le prédicat d'orientation `ccw` :

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

D'après cette définition et les propriétés sur les nombres réels, nous prouvons formellement en Coq que toutes les propriétés issues de l'axiomatisation de Knuth sont vérifiées. De

plus, la propriété d'orientation est décidable, c'est-à-dire qu'elle peut être utilisée dans des expressions conditionnelles d'algorithmes. Le lemme `ccw_dec` exprime cette décidabilité et il est formellement prouvé en Coq.

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

Nous avons maintenant un cadre abstrait pour manipuler le prédicat d'orientation de manière formelle. Aucune question liée aux calculs numériques ne sera considérée dans la suite. Nous considérerons seulement que nous avons un prédicat décidable `ccw` disponible qui satisfait la spécification mentionnée ci-dessus et qui peut être utilisé pour déterminer l'orientation d'un triplet de points du plan.

3.3 Enveloppe convexe

Soit E le plan euclidien. La notion de convexité obéit à la définition ci-dessous et elle est illustrée par la figure 3.3.

Définition 5 (Partie convexe).
Une partie C de E est dite convexe *si, pour toute paire de points (p, q) de C, le segment $[pq] = \{x \in E \mid \exists\ t \in [0,1],\ x = tp + (1-t)q\}$ est complétement contenu dans C.*

On peut se faire une idée de l'enveloppe convexe d'un ensemble fini de points du plan en la visualisant comme la forme que prend un élastique encerclant un ensemble de clous plantés dans une planche de bois où l'élastique représenterait l'enveloppe convexe et les clous seraient les points de cet ensemble. Cette métaphore de l'élastique est illustrée à la figure 3.4.

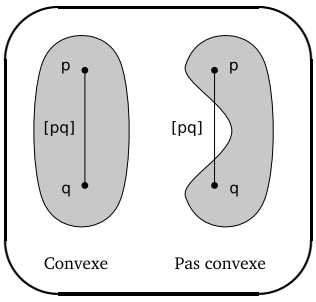


Figure 3.3 - La notion de la convexité

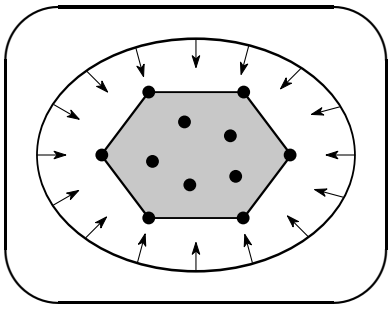


Figure 3.4 - Métaphore de l'élastique

Le calcul de l'enveloppe convexe dans le plan est un des problèmes les plus étudiés en géométrie algorithmique. Plusieurs définitions menant à différents algorithmes sont proposées dans la littérature [44, 6, 11, 21]. Pour notre part, nous choisissons une définition bien adaptée à notre modèle topologique d'hypercarte permettant d'utiliser le prédicat d'orientation de Knuth *ccw* dans un calcul incrémental.

Soit S un ensemble de points du plan. Comme la plupart des auteurs, nous supposons que les points sont placés dans une *configuration générale*, c'est-à-dire qu'ils sont tous distincts et qu'il n'y a jamais trois points alignés. Notons cependant que Pichardie et Bertot [43] se sont penchés sur ces cas dégénérés en utilisant une méthode de perturbation.

Définition 6 (Enveloppe convexe).
L'enveloppe convexe de S est le polygone convexe T dont les sommets t_i, parcourus dans le sens trigonométrique pour $i = 1, \ldots, n$ avec $n+1$ assimilé à 1, sont les points de S tels que, pour chaque arête $[t_i t_{i+1}]$ de T et pour chaque point p de S différent de t_i et t_{i+1}, on a ccw(t_i,t_{i+1},p). Autrement dit, pour toute arête $[t_i t_{i+1}]$ de T, tous les points p de S différents de t_i et t_{i+1} se situent à gauche de la droite orientée engendrée par le vecteur $\overrightarrow{t_i t_{i+1}}$.

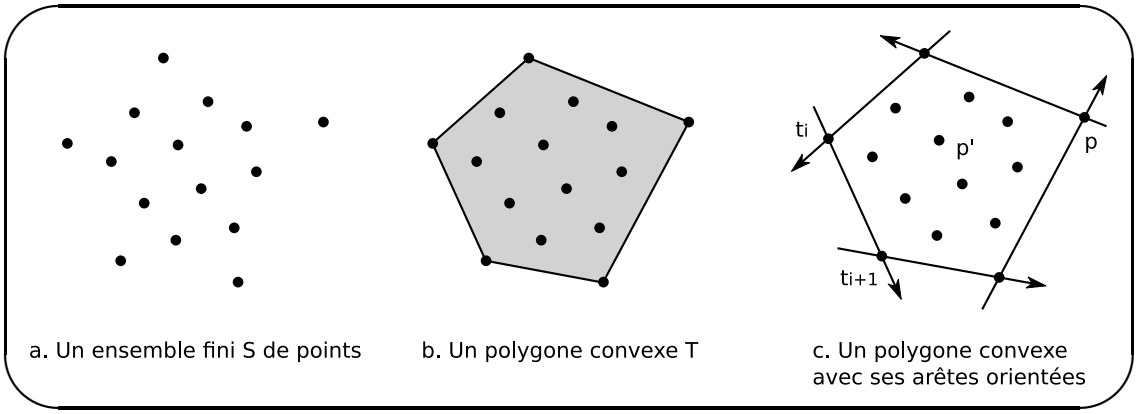


Figure 3.5 - Caractérisation de l'enveloppe convexe

La figure 3.5 illustre la caractérisation de l'enveloppe convexe qui utilise le prédicat *ccw*. A gauche (a), nous avons un ensemble fini S de points du plan. Au milieu (b), le polygone convexe T représentant l'enveloppe convexe de S avec son intérieur grisé. A droite (c), les flèches représentent les droites orientées $\overrightarrow{t_i t_{i+1}}$ issues des arêtes $[t_i t_{i+1}]$ de T. Tous les points distincts $p, p', \ldots$ se situent à gauche de ces droites orientées.

3.4 Algorithme incrémental

L'algorithme incrémental calcule l'enveloppe convexe d'un ensemble fini de points du plan en construisant progressivement pour chaque point pris l'un après l'autre une nouvelle enveloppe convexe à partir d'un polygone convexe dont les sommets sont des points de l'ensemble initial déjà traités. A chaque étape, soit le nouveau point se situe à l'intérieur du polygone et on peut passer à l'étape suivante, soit celui-ci est à l'extérieur du polygone et il faut supprimer certaines arêtes et en ajouter deux autres pour former ainsi un nouveau polygone convexe.

L'algorithme incrémental peut être vu comme une composition de deux fonctions. La première fonction CHID calcule l'enveloppe convexe d'un polygone convexe et d'un

nouveau point, c'est-à-dire qu'elle insére un nouveau point dans une enveloppe convexe déjà construite. La deuxième fonction CH se charge de prélever progressivement un à un les points de l'ensemble initial et de construire à chaque fois une nouvelle enveloppe convexe grâce à l'appel de la fonction d'insertion CHID.

La fonction CHID qui se charge de la construction de l'enveloppe convexe d'un polygone convexe T et d'un nouveau point p se base sur des tests qui utilisent le prédicat géométrique d'orientation *ccw* de Knuth. D'après la définition 6, on sait que l'intérieur du polygone T est défini par les points x du plan tels que $ccw(t_i, t_{i+1}, x)$ pour toute arête $[t_i t_{i+1}]$ de T. On remarque de plus que la droite orientée engendrée par le vecteur $\overrightarrow{t_i t_{i+1}}$ divise le plan en deux demi-plans caractérisés par la valeur de $ccw(t_i, t_{i+1}, x)$ pour tout point x du plan. On peut ainsi facilement situer le point p par rapport à chaque arête $[t_i t_{i+1}]$ du polygone T en évaluant simplement $ccw(t_i, t_{i+1}, p)$ et de ce fait savoir si p se trouve à l'intérieur de T ou pas. Si p est contenu dans T, l'enveloppe convexe de T et p est égale à T. Sinon (c'est-à-dire si p est à l'extérieur de T), il faut supprimer les arêtes visibles depuis p et créer les arêtes $[t_g p]$ et $[p t_d]$ pour relier p au sommet gauche t_g et au sommet droit t_d de T dont nous donnons les définitions ci-dessous. Retenons qu'il faudra par la suite prouver l'existence et l'unicité de ces deux sommets.

Définition 7 (Arête visible, sommet gauche, sommet droit).
Soient T un polygone convexe dans E et p un point de E.

1. *L'arête $[t_i t_{i+1}]$ de T est* visible *depuis p lorsque l'on a $\neg\ ccw(t_i, t_{i+1}, p)$.*
2. *Le sommet t_g de T est le* sommet gauche *par rapport à p si l'on a $ccw(t_{g-1}, t_g, p)$ et $\neg\ ccw(t_g, t_{g+1}, p)$.*
3. *Le sommet t_d de T est le* sommet droit *par rapport à p si l'on a $\neg\ ccw(t_{d-1}, t_d, p)$ et $ccw(t_d, t_{d+1}, p)$.*

Notons que ces définitions ne font aucune supposition sur le nombre de points. Le principe de l'algorithme incrémental est donc de construire un premier polygone convexe puis, pour chaque nouveau point p issu de S, d'étendre le polygone convexe créé à l'étape précédente selon la méthode exposée ci-dessus. Pour les cas particuliers où S n'est composé que d'un point ou deux, l'enveloppe convexe correspond respectivement à ce point isolé ou à l'arête formée par ces deux points.

Nous donnons ci-après une description informelle en pseudo-code de ces deux fonctions qui s'appuie sur les définitions classiques de l'algorithme incrémental [11, 6, 10]. Elles seront définies plus précisément dans un cadre de cartes combinatoires orientées plongées.

Notons que nous aurons à prouver plus tard l'équivalence de l'existence de t_g et t_d. En effet, quand p est à l'intérieur du polygone, t_g et t_d n'existent pas. Mais lorsque p se trouve à l'extérieur, ils existent tous les deux. Aucun autre cas n'est à considérer puisque p ne peut pas être colinéaire avec deux sommets du polygone convexe. De plus, nous prouverons l'unicité de ces deux sommets t_g and t_d lorsqu'ils existent.

Nous utilisons les hypercartes présentées au chapitre précédent pour modéliser et prouver formellement cet algorithme.

Algorithm 1 - CHID : Calcul de l'enveloppe convexe d'un polygone convexe et d'un point

Require: Un polygone convexe T et un point p
Ensure: L'enveloppe convexe de T et p
1 : **if** p est à l'extérieur de T **then**
2 : Trouver le sommet gauche t_g et le sommet droit t_d de T par rapport à p
3 : Supprimer les arêtes de T visibles depuis p
4 : Créer les deux arêtes $[t_g p]$ et $[p t_d]$
5 : **end if**
6 : **return** T

Algorithm 2 - CH : Calcul de l'enveloppe convexe d'un ensemble de points du plan

Require: Un ensemble fini S de n points du plan ($S = \{p_1, p_2, \ldots, p_n\}$)
Ensure: L'enveloppe convexe T de S
1 : Construire un premier polygone convexe T avec p_1 et p_2
2 : **for all** point p_i de S avec $i \in \{3, \ldots, n\}$ **do**
3 : $T =$ CHID T p_i
4 : **end for**
5 : **return** T

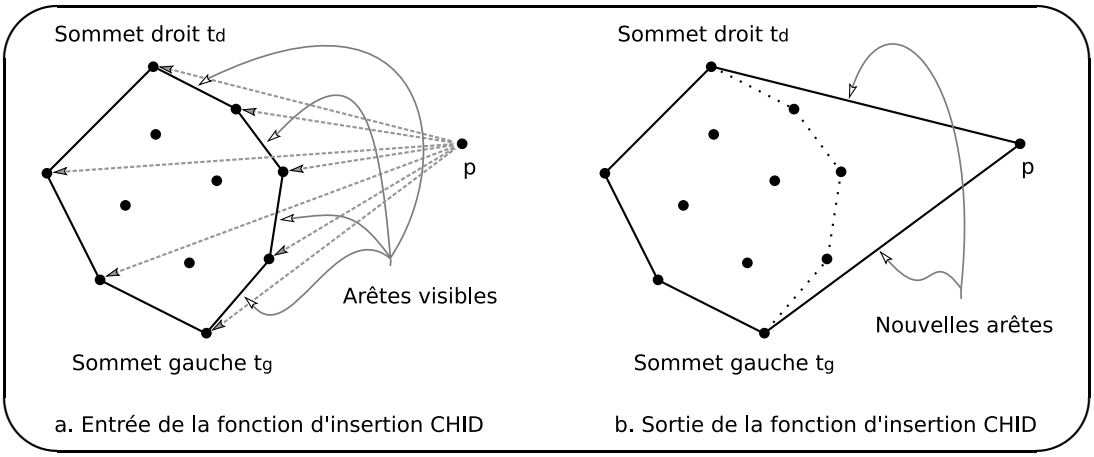


Figure 3.6 - Calcul d'une nouvelle enveloppe convexe à partir d'un polygone convexe T et d'un nouveau point p

3.5 Représentation des données

Dans notre modélisation, les subdivisions de surface que nous manipulons sont représentées par des cartes combinatoires orientées, contrairement à Pichardie et Bertot [43] ou Meikle et Fleuriot [38] qui n'utilisent pas de structure topologique mais représentent les enveloppes convexes par des listes ou des vecteurs de points.

L'ensemble initial de points du plan dont nous cherchons à calculer l'enveloppe convexe est représenté sous forme d'une carte de type `fmap` contrainte pour être une carte combinatoire orientée (cf. section 2.2.4) où chaque point est représenté par un brin isolé sans

liaison dont le plongement correspond aux coordonnées dudit point (cf. figure 3.7 (a)).

L'enveloppe convexe finale est un polygone également représenté par une carte de type `fmap` contrainte pour être une carte combinatoire orientée. Chaque sommet du polygone est représenté par un sommet topologique (deux brins distincts aux plongements identiques liés à la dimensions `one`) et chaque arête est représentée par une arête topologique (deux brins distincts aux plongements différents liés à la dimension `zero`). Ceci est illustré à la figure 3.7 (b).

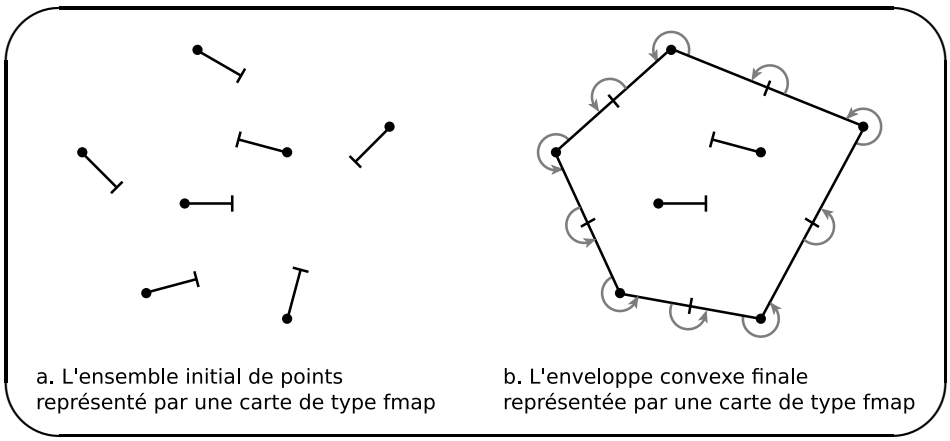


Figure 3.7 - Représentation des données par des cartes de type `fmap`

Nous verrons que tous les objets intermédiaires sont aussi représentés par des cartes combinatoires orientées contenant possiblement des brins isolés. Par conséquent, nous ne considérerons par la suite qu'un sous-type des objets de type `fmap` qui sont en fait des cartes combinatoires orientées.

Comme le calcul incrémental de l'enveloppe convexe repose sur des tests d'orientation dans le plan, il est nécessaire d'orienter le polygone dans le sens trigonométrique. C'est ce que nous réalisons facilement en liant toujours les brins dans le même sens. Les liaisons correspondantes sont représentées par des petites flèches dans le dessin. De plus, les brins représentant des points qui se situent à l'intérieur de l'enveloppe convexe sont conservés dans la carte finale où ils demeurent isolés sans liaison (cf. figure 3.7 (b)). Ces brins pourraient être supprimés en cas de besoin.

Deuxième partie

Algorithme par induction structurelle

Sommaire

Chapitre 4

Structure du programme en Coq

Dans ce chapitre, nous décrivons un algorithme incrémental de calcul d'enveloppe convexe par induction structurelle sur le type inductif `fmap` des hypercartes. En effet, comme nous le détaillerons plus loin (cf. section 4.7), dans cette première expérience, nous avons choisi d'aborder cet algorithme d'une manière originale qui se distingue de la méthode usuelle par le fait que nous ne nous déplaçons pas sur l'enveloppe convexe pour discerner les arêtes visibles des arêtes invisibles ou trouver les sommets gauche et droit mais nous analysons séparément chaque brin et chaque liaison en les examinant dans un ordre aléatoire dicté par la structure du terme de la carte de type `fmap` représentant l'enveloppe convexe.

4.1 Catégorisation des brins

D'après nos choix de représentation énoncés dans le chapitre précédent (cf. section 3.5), nous observons qu'il est possible de classer de manière très précise en trois catégories les brins utilisés dans notre description en Coq de l'algorithme incrémental. Nous identifions ces trois catégories de brins par un code de couleur et nous définissons un prédicat pour chacun d'entre eux. Les *brins noirs* sont les brins isolés sans liaison, les *brins bleus* sont ceux qui possèdent exactement un prédécesseur à la dimension `one` et un successeur à la dimension `zero`, et les *brins rouges* sont ceux qui ont exactement un prédécesseur à la dimension `zero` et un successeur à la dimension `one`. Ces trois catégories sont illustrées à la figure 4.1(a).

En Coq, les prédicats `black_dart`, `blue_dart`, `red_dart` testent respectivement si un brin `x` est un brin noir, un brin bleu ou un brin rouge dans une carte `m`. Leur décidabilité est exprimée par les fonctions `black_dart_dec`, `blue_dart_dec`, `red_dart_dec` qui peuvent être utilisées dans les définitions de fonctions suivantes pour faire des branchements suivant la couleur des brins.

```
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.
```

Par la suite, on dira par extension qu'un brin noir est un `black_dart`, qu'un brin bleu est un `blue_dart` et qu'un brin rouge est un `red_dart`.

Cette classification est illustrée à la figure 4.1(b) où nous voyons clairement les trois catégories de brins présents dans notre représentation de l'enveloppe convexe. Pour une visibilité en noir et blanc, les `black_dart` sont illustrés par un trait continu, les `blue_dart` par un trait en tirets et les `red_dart` par un trait en pointillés.

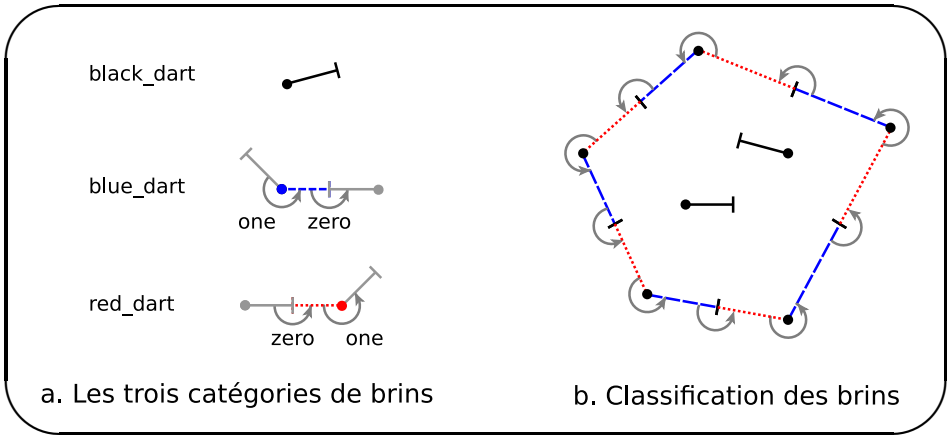


Figure 4.1 - Les trois catégories de brins et leur rôle dans notre représentation de l'enveloppe convexe

4.2 Préconditions

Dans notre formalisation, il est nécessaire de définir précisément les *préconditions* que la carte initiale `m` doit satisfaire avant le début du programme. Ces préconditions sont regroupées dans une définition `prec_CH` qui possède quatre prédicats et qui est définie de la façon suivante :

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

L'hypercarte `m` doit bien évidemment vérifier l'invariant `inv_hmap` qui est expliqué dans la section 2.2.4. Elle ne doit avoir que des brins isolés sans liaison, c'est-à-dire qu'elle ne doit pas posséder d'occurrence du constructeur `L`. Cette propriété est exprimée par le prédicat `linkless`. Le prédicat `well_emb` exprime le fait que les plongements géométriques doivent être bons, c'est-à-dire que tous les brins d'entrée doivent avoir un plongement différent. Le prédicat `well_emb` capture cette propriété bien qu'elle s'assure aussi d'autres propriétés ayant trait aux liaisons (cf. section 7.2). Dans notre travail, nous supposons que trois brins ayant des plongements différents ne peuvent pas être plongés dans trois points colinéaires. Cette propriété est exprimée par le prédicat `noalign`.

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

Fixpoint linkless (m:fmap) {struct m} : Prop :=
  match m with
    V => True
  | I m0 _ _ => linkless m0
  | L _ _ _ _ => False
  end.

Definition well_emb (m:fmap) : Prop :=
  forall (da:dart), exd m da ->
  (succ m zero da -> (fpoint m da) <> (fpoint m (A m zero da))) /\
  (pred m zero da -> (fpoint m da) <> (fpoint m (A_1 m zero da))) /\
  (succ m one da -> (fpoint m da) = (fpoint m (A m one da))) /\
  (pred m one da -> (fpoint m da) = (fpoint m (A_1 m one da))) /\
  (forall db:dart, exd m db -> db <> da -> db <> (A m one da) ->
   db <> (A_1 m one da) -> (fpoint m da) <> (fpoint m db)).

Definition noalign (m:fmap) : Prop :=
  forall (d1:dart)(d2:dart)(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 -> ~ align p1 p2 p3.
```

Dans les définitions ci-dessus, `fpoint m d` est le point sur lequel le brin `d` est plongé dans la carte `m`.

4.3 Fonction principale CH

Nous définissons d'abord la fonction principale `CH` qui calcule entièrement l'enveloppe convexe d'un ensemble fini de points dans le plan représenté par une carte `m`. Si la carte initiale `m` est vide, elle retourne la carte vide `V`. Si `m` a seulement un brin, elle retourne une carte avec juste ce brin isolé. Si elle a au moins deux brins, elle procède comme suit : la fonction `CH` construit un premier polygone convexe avec deux des brins impliqués en utilisant la fonction `CH2` (figure 4.2) puis elle appelle la fonction récursive `CHI`. La fonction `CH` devant être totale, dans les autres cas, elle retourne par défaut la carte vide `V`.

```
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.
```

Notons que `max_dart m` retourne le brin le plus grand - en fait, les brins sont des entiers - de la carte `m`. Comme nous allons voir, cette fonction nous aide à simuler la génération de nouveaux brins (des brins qui n'apparaissent pas dans `m`). Par exemple, la fonction `CH2` utilise deux nouveaux brins (cf. section 4.4). Par conséquent, l'appel à `CHI`, qui a aussi besoin d'un nouveau brin, est effectuée avec le paramètre `(max_dart m)+3`.

4.4 Fonction d'initialisation CH2

Soient `x1` et `x2` deux brins distincts, la fonction `CH2` construit la carte combinatoire orientée illustrée à la figure 4.2. Pour ce faire, elle introduit deux nouveaux brins, nommés `max+1` et `max+2`, et elle les lie convenablement avec `x1` et `x2`. Au lieu d'avoir une simple arête comme présenté dans la section 3.4, nous avons en fait un polygone aplati (cf. figure 4.2, les arêtes sont courbées pour des raisons de visibilité) composé de quatre brins avec leurs liaisons. Cela nous permet de considérer le cas de deux brins comme un cas général.

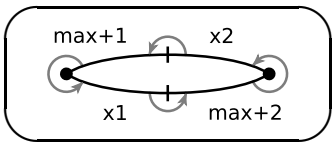


Figure 4.2 - L'enveloppe convexe de deux brins construite par la fonction CH2

```
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).
```

4.5 Fonction de récursion CHI

La fonction `CHI` extrait les brins de la carte `m1` un par un et elle fabrique pour chacun d'entre eux une nouvelle enveloppe convexe en utilisant la fonction `CHID` avec la carte `m2` représentant le polygone convexe déjà construit aux étapes précédentes et le paramètre `max`. Ainsi, l'appel récursif de `CHI` se fait avec le paramètre `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.
```

4.6 Caractérisation des brins gauche et droit

Dans la section 3.4, nous avons mis l'accent sur le rôle de la *visibilité* d'une arête depuis un point ainsi que sur le rôle des sommets gauche et droit. Comme nous travaillons avec des brins, la visibilité d'une arête est exprimée sur l'un de ces deux brins et les sommets gauche et droit sont remplacés par deux brins, le brin gauche et le brin droit (cf. figure 4.3 pour une description graphique).

Premièrement, nous définissons les prédicats `visible` et `invisible` qui utilisent la classification des brins et le prédicat d'orientation de Knuth `ccw`. Le prédicat `invisible` est exactement la négation de `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)).
```

Comme d'habitude, les propriétés de décidabilité `visible_dec` et `invisible_dec` sont prouvées. Puis, en suivant la définition 7, nous spécifions deux prédicats `left_dart` et `right_dart` qui déclarent qu'un brin `d` est le brin gauche ou le brin droit de `m` par rapport à un 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.
```

La décidabilité de ces prédicats est prouvée par deux lemmes : `left_dart_dec` et `right_dart_dec`. Notons que, par convention, le brin gauche est toujours un brin bleu et le brin droit est toujours un brin rouge. De plus, nous avons à prouver l'équivalence de l'existence et l'unicité de ces deux sommets (cf. section 7.1).

4.7 Fonction d'insertion `CHID`

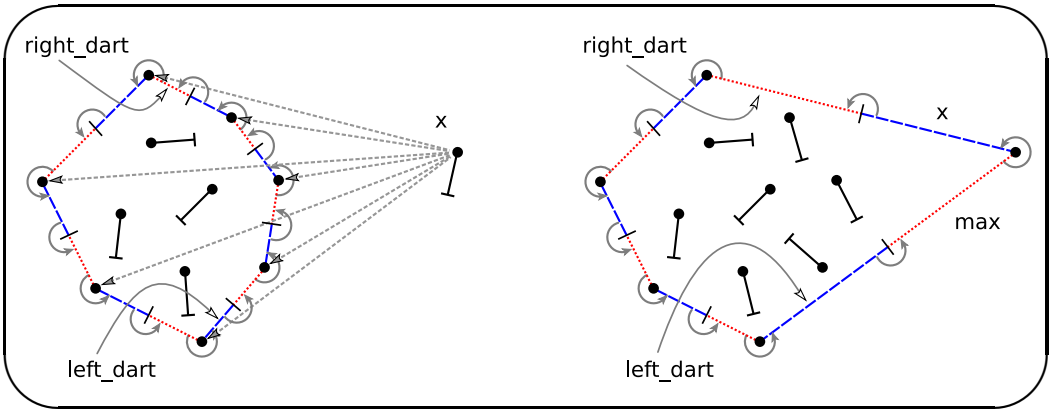


Figure 4.3 - Fonctionnement de la fonction d'insertion `CHID`

Comme déjà indiqué dans la section précédente, la fonction `CHID` calcule l'enveloppe convexe d'un polygone convexe représenté par une carte `m` (de type `fmap`) et d'un nouveau point représenté par un brin `x`. Elle agit par récursion structurelle sur `m` en étudiant chaque brin et chaque liaison séparément. Les brins sont traités dans l'ordre dans lequel ils apparaissent dans la structure du terme de la carte `fmap`, tout en reconstruisant le polygone (au lieu de le parcourir dans le sens trigonométrique). Etant donné que la carte `m` est modifiée à chaque appel récursif, la fonction `CHID` conserve une *carte de référence* `mr`, qui est identique à `m` quand `CHID` est appelée pour la première fois. Cette carte de référence est très utile pour effectuer des tests et elle n'est jamais modifiée durant l'exécution de la fonction. A chaque étape, la carte `m` est une *sous-carte* de la carte de référence `mr` conformément à la définition suivante :

```
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.
```

En pratique, une carte `m` est une sous-carte de `mr` si tous les brins qu'elle contient appartiennent aussi à `mr` en possédant le même plongement et si toutes ses liaisons de retrouvent également dans `mr`. Initialement, `m` est égale à `mr` et, à chaque appel récursif, nous prouvons formellement que la propriété `submap m mr` est encore conservée.

Comme cela a été dit précédemment, il y a deux cas possibles dans `CHID`. Soit le nouveau point se situe à l'intérieur du polygone convexe, auquel cas la fonction `CHID` insère simplement le brin `x` dans la carte `m` sans aucune liaison, soit il est à l'extérieur et elle supprime les arêtes du polygone qui sont visibles depuis le nouveau point et elle en crée deux nouvelles en les connectant avec le sommet gauche et le sommet droit.

En réalité, la fonction d'insertion `CHID` travaille de manière *constructive* et non destructive : elle démonte progressivement l'ancienne carte en extrayant individuellement chaque brin et chaque liaison puis elle reconstruit toujours, en partant de zéro, une nouvelle hypercarte. Ainsi, si un brin ou un lien de la première carte ne doit pas être réintégré dans la deuxième, il est tout simplement oublié. Dans ce contexte, un appel récursif à (`CHID m mr x p max`) se déroule comme cela a été indiqué ci-dessous (cf. figure 4.4) :

Si `m` est la carte vide (ligne 04), la fonction `CHID` retourne simplement le brin `x` sans liaison.

Si `m` correspond à (`I m0 x0 p0`) (ligne 05), `CHID` teste la catégorie du brin `x0` dans `mr`. Si `x0` est un brin *bleu* dans `mr` (ligne 06), le programme vérifie si `x0` appartient à une arête de `mr` qui est invisible depuis le nouveau brin `x` plongé sur le point `p` (ligne 07). Ce test est réalisé en utilisant le prédicat `invisible_dec`. Si l'arête de `x0` est invisible depuis `p` (ligne 08), le brin est conservé dans la carte. Autrement, le programme vérifie (ligne 09) si `x0` est le nouveau brin gauche de `mr` par rapport à `x`. Si `x0` est le brin gauche (ligne 10), il est maintenu dans la carte. De plus, un nouveau brin `max` (plongé en `p`) est inséré et lié à `x` à la dimension `one`. Finalement, `x0` et `max` sont liés à la dimension `zero`. Autrement, `x0` demeure dans la carte (ligne 12).
Si `x0` est un brin *rouge* dans `mr`, un raisonnement similaire est effectué (lignes 13-18).
Si `x0` est un brin *noir* dans `mr`, il est conservé dans la carte (ligne 19).

Si `m` correspond à (`L m0 zero x0 y0`) (ligne 20), `CHID` teste si l'arête formée par `x0` et `y0` est invisible depuis `x` plongé en `p` (ligne 21). Si elle est invisible depuis `p`, le lien à la dimension `zero` entre `x0` et `y0` est maintenu (ligne 22). Autrement, il n'est pas ajouté dans la carte résultante (ligne 23).

Des étapes similaires s'appliquent si `m` correspond à (`L m0 one x0 y0`) (lignes 24-29).

```
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 4.4 - La fonction d'insertion CHID en Coq

Chapitre 5

Extraction en Ocaml

Dans ce chapitre, nous présentons rapidement le mécanisme d'extraction automatique de Coq et son emploi dans notre travail pour produire rapidement un programme utilisable en OCaml afin de visualiser le résultat fourni par notre algorithme pour s'assurer que celui-ci correspond bien à nos attentes.

5.1 Principe d'extraction de Coq

Coq intégre un mécanisme d'extraction qui permet d'engendrer automatiquement des programmes certifiés en Objective Caml ou en Haskell à partir de preuves ou de spécifications grâce à l'isomorphisme de Curry-Howard entre la programmation fonctionnelle et la déduction naturelle qui se base sur le paradigme : « preuve = programme » et « proposition = spécification ».

Nous utilisons ce mécanisme pour extraire un programme de construction en Ocaml permettant une visualisation graphique du calcul de l'enveloppe convexe d'un ensemble fini de points du plan utilisant les cartes combinatoires orientées plongées comme modéle de représentation des subdivisions du plan.

5.2 Technique d'extraction

On choisit le langage d'extraction par la commande `Extraction Language Ocaml`. On choisit la fonction que l'on souhaite extraire avec ses dépendances dans un fichier de destination par la commande `Extraction "fmaps" CH`. On redéfinit les types et les axiomes utilisés dans notre développement en Gallina que Coq n'est pas capable de traduire en Ocaml par les commandes `Extract Inductive` et `Extract Constant`.

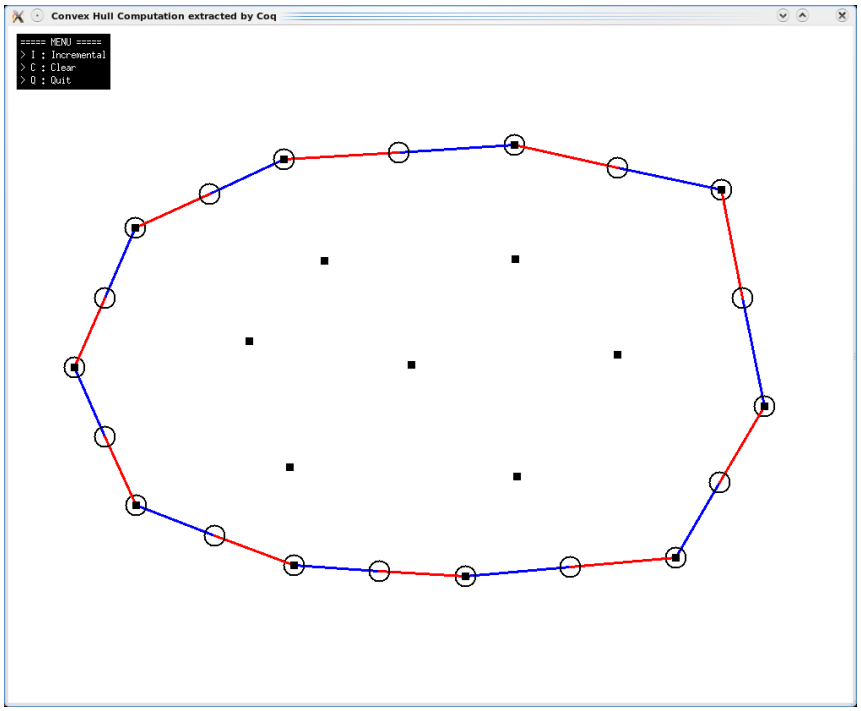


Figure 5.1 - Interface graphique

```
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)".
```

Nous choisissons de relier les nombres réels de Coq avec les nombres flottants en OCaml afin de pouvoir rapidement extraire notre spécification en Coq dans un prototype de programme en OCaml. Cependant, nous devons souligner qu'une telle traduction (qui est en fait une approximation) est quelque peu douteuse. En effet, les nombres réels utilisés en Coq sont axiomatisés, ils ne sont pas constructifs. Les propriétés des réels ne sont pas conservées par les flottants. Par exemple, l'addition de nombres flottants en OCaml n'est ni associative ni commutative. Une telle traduction mène à des erreurs dans l'évaluation des prédicats géométriques comme remarqué dans [32]. Une extraction plus judicieuse que nous pourrions faire serait d'insérer ce programme prouvé correct dans un modeleur qui considère les nombres rationnels en Coq plutôt que les nombres réels. Cela devrait être suffisant pour notre objectif et l'extraction des nombres rationnels de leur implémenta-

tion en Coq dans celle en OCaml serait simple et, plus important, elle serait plus sûre. Néanmoins, il est important de rappeler ici que les questions numériques ne sont pas notre souci principal dans cette première expérience où nous nous intéressons principalement aux aspects algorithmiques des programmes étudiés.

Comme le programme extrait dans le fichier `fmaps` ne contient que les fonctions `CH`, `CHI`, `CH2`, et `CHID` permettant de calculer l'enveloppe convexe d'un ensemble fini de points du plan reçu en paramètre, il faut créer une interface graphique afin de pouvoir saisir les points du plan et obtenir une visualisation graphique du résultat représenté par un polygone sous forme d'une carte de type `fmap`.

5.3 Interface graphique

Nous développons un petit programme dans le fichier `main` contenant la fonction `poll` pour gérer les évènements du clavier et de la souris, la fonction `show_fmap` qui s'occupe de l'affichage de l'enveloppe convexe où les brins sont colorés en fonction de leur catégorie de classification et la fonction `banner` qui affiche un menu pour indiquer à l'utilisateur les commandes disponibles.

Comme l'ensemble fini de points du plan est stocké dans le programme sous forme d'une liste de points mais qu'il doit être passé à la fonction `CH` sous forme d'une carte de type `fmap`, la fonction récursive `list_to_fmap` transforme une liste de points en une carte de type `fmap`. De plus, les fonctions `i2n` et `n2i` transforment respectivement un `int` en `nat` et un `nat` en `int`.

La figure 5.1 illustre un exemple d'utilisation de cette interface graphique.

Ce mécanisme d'extraction fonctionne correctement. Il n'est pas indispensable mais il s'avère très utile pour évaluer et expérimenter les algorithmes de géométrie algorithmique conçus en Coq avant de s'attaquer à leurs preuves formelles.

Chapitre 6

Propriétés topologiques

Initialement, nous avions décidé de prouver d'abord les propriétés topologiques puis de nous occuper des propriétés géométriques. Cependant, pour certaines propriétés topologiques, d'inévitables propriétés géométriques interfèrent. Nous ne pouvons pas raisonner sur des problèmes topologiques sans prendre en compte quelques faits géométriques de base. Ce chapitre se concentre sur des preuves de propriétés topologiques, même si elles dépendent parfois de propriétés géométriques. Les propriétés topologiques les plus pertinentes sont la préservation de l'invariant des hypercartes, la propriété que l'hypercarte décrivant l'enveloppe convexe est toujours un polygone, la conservation des brins initiaux dans la carte finale et la propriété de la planarité.

6.1 Evolutions des brins

Comme présenté dans la section 4.1, il est possible de classer les brins manipulés par notre algorithme en trois catégories (*bleu*, *rouge*, ou *noir*). De plus, comme il procède par induction structurelle sur une carte, la fonction d'insertion `CHID` examine les brins les uns après les autres dans un ordre quelconque. Aussi, les liaisons ne sont pas étudiées en même temps que les brins auxquelles ils appartiennent. Par conséquent, durant les appels récursifs, les brins ne demeurent pas continuellement dans la même catégorie et ils ne passent pas immédiatement d'une catégorie à l'autre. Ils peuvent ne perdre qu'une de leurs liaisons à la fois. Ils passent ainsi par des *états intermédiaires* pendant l'exécution de la fonction d'insertion `CHID`.

Par exemple, les brins bleus peuvent soit perdre leur successeur à la dimension `zero`, soit leur prédécesseur à la dimension `one`. Si les deux liaisons sont supprimées, ils deviennent alors des brins noirs.

Afin de faciliter les preuves, notamment dans la preuve de la planarité (cf. section 6.5), nous définissons quatre catégories de brins intermédiaires appelées `half_blue_succ` pour

un *demi-brin bleu* avec seulement un successeur (à la dimension `zero`), `half_blue_pred` pour un demi-brin bleu avec seulement un prédécesseur (à la dimension `one`) :

```
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.
```

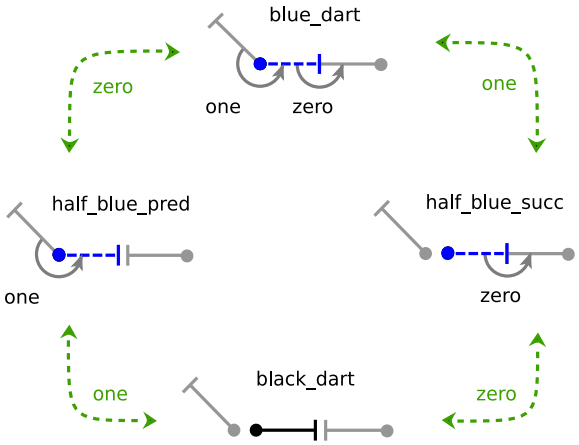


Figure 6.1 - Les évolutions possibles d'un brin bleu lors d'appels récursifs de la fonction CHID

Nous pouvons immédiatement transposer ces définitions pour les brins rouges. Ces nouvelles catégories représentant les états intermédiaires et les possibles changements de catégorie pour les brins bleus et les demi-bleus sont présentés dans la figure 6.1 où les flèches vertes indiquent l'ajout ou la suppression de liaisons aux dimensions `zero` et `one`. Cela fonctionne exactement pareil pour les brins rouges. Tous ces changements dans la carte combinatoire orientée de départ expliquent pourquoi nous avons besoin d'avoir une carte de référence. Elle est identique à la carte initiale avant le premier appel à la fonction d'insertion, comme on le voit dans le code présenté à la section 4.7. Elle sert à tester l'existence de brins ou de liaisons.

Pour considérer tous les cas possibles qui peuvent survenir pendant les appels récursifs de la fonction d'insertion `CHID`, nous établissons 78 lemmes relatifs à ces changements de catégories de brins. Nous avons 2 lemmes pour les `black_dart`, 22 lemmes pour les `blue_dart` et pour les `red_dart`, 12 lemmes pour le brin `x` et pour le brin `max`, et 8 lemmes inverses qui expriment la catégorie du brin en fonction de ses paramètres. Nous donnons ci-après deux exemples de ces lemmes :

Le premier lemme `blue_dart_CHID_1` exprime que, si un brin `d` est bleu dans la carte de référence `mr` et qu'il appartient à la carte courante `m`, alors il reste dans la carte résultante `(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.
```

Le second lemme `blue_dart_CHID_11` exprime que le successeur à la dimension `zero` du brin gauche `d` dans la carte `(CHID m mr x tx px max)` est le nouveau brin `max`. En effet, le brin gauche `d` de `m` est diffèrent du brin `x` à ajouter, il est bleu dans la carte de référence `mr`, il appartient à la carte courante `m`, et il est visible depuis le nouveau point inséré `px` mais son prédécesseur à la dimension `one` ne l'est pas.

```
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.
```

Un exemple complet de preuve d'un lemme étudiant l'évolution d'un brin lors d'un appel à la fonction d'insertion `CHID` est donné en A.

Ces lemmes sont utilisés dans la preuve de plusieurs propriétés sur la carte issue de `(CHID m mr x tx px max)`. Par exemple, il est nécessaire de démontrer que le nouveau 0-successeur du brin gauche est le brin `max` déjà présent dans la carte avant l'établissement de la liaison et n'ayant pas encore lui-même de 0-prédécesseur, afin de garantir l'invariant des hypercartes.

6.2 Préservation de l'invariant des hypercartes

Le premier théorème important que nous voulons prouver est que l'invariant `inv_hmap` est maintenu tout au long du programme, de la carte initiale `m` à la carte finale `(CH m)`. Pour rappel, ce prédicat indique que les brins doivent être dans la carte avant d'être cousus ensemble et qu'un brin ne peut pas être inséré deux fois dans la même carte (cf. section 2.2.4).

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

Cette preuve procède par induction structurelle sur une carte libre m. Elle repose à la fois sur les lemmes de la section précédente concernant les changements de catégories de brins durant l'exécution de l'algorithme, et sur les preuves techniques de l'unicité des brins gauche et droit (cf. section 7.1).

Cela illustre que les propriétés topologiques dépendent de propriétés géométriques dans des algorithmes géométriques tels que le calcul de l'enveloppe convexe. En effet, la propriété capitale pour établir le théorème mentionné ci-dessus est que la conservation d'un brin `d` dans la carte combinatoire orientée courante (propriété topologique) dépend du fait qu'il soit visible ou non par rapport au point qui est inséré dans l'enveloppe convexe (propriété géométrique).

6.3 Obtention d'un ensemble de polygones topologiques

Le prédicat `inv_poly` exprime ce qu'est un *ensemble de polygones topologiques*. Informellement, cela consiste en un ensemble de polygones et de brins isolés. Dans notre algorithme, nous nous attendons à obtenir en fait un polygone avec des brins isolés. Dans une carte vérifiant `inv_poly`, tous les brins sont soit *noirs*, *bleus*, or *rouges*. Aucun brin n'est partiellement lié. Par conséquent, chaque brin `d` dans la carte est soit *noir* (isolé sans liaison), soit *bleu* ou *rouge*, ce qui signifie qu'il appartient à la composante connexe qui forme ce que nous considérons être l'enveloppe convexe.

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

Nous prouvons que la carte libre résultat de la fonction `CH` vérifie l'invariant `inv_poly` et donc qu'elle contient un *ensemble de polygones topologiques*.

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

La preuve procède de la même manière que pour la preuve de la propriété `inv_hmap` mais elle utilise aussi l'*équivalence de l'existence* du brin gauche et du brin droit (cf. section 7.1).

6.4 Conservation des brins initiaux

Une autre propriété fondamentale à prouver est que les brins présents dans la carte initiale sont conservés dans la carte finale avec leur plongement.

```
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)).
```

En observant la structure de la fonction `CHID` de plus près, on s'aperçoit que tous les brins initiaux, issues de la carte de départ, deviennent, dans la carte finale, soit des brins noirs (s'ils sont situés à l'intérieur de l'enveloppe convexe), soit des brins bleus. Inversement, tous les brins noirs ou bleus de la carte finale sont des brins issues de la carte initiale. Ainsi, tous les brins rouges de la carte finale sont des nouveaux brins crées au fur et à mesure de la construction de l'enveloppe convexe grâce au paramètre `max` qui est initialisé dans `CH` avec la fonction `max_dart` et incrémenté à chaque étape. De plus, les seuls brins qui peuvent être supprimés de la carte au cours des appels récursifs sont les brins rouges qui sont insérés pendant le calcul de l'enveloppe convexe.

Alors, pour prouver que les brins issues de la carte initiale sont encore présents dans la carte finale, il suffit de prouver que les brins noirs et bleus sont maintenus dans la carte résultante à chaque fois qu'un nouveau brin est inséré.

6.5 Planarité de l'enveloppe convexe

La planarité est une propriété assez immédiate mais il est satisfaisant de la vérifier formellement. Jusqu'à présent, nous avons prouvé que notre algorithme produit finalement un ensemble de polygones et des brins isolés. Nous devons vérifier que cet ensemble de polygones résultant est vraiment planaire. La propriété de planarité `planar` est définie comme suit [18, 19] :

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

Nous prouvons que, si `m` vérifie les préconditions présentées dans la section 4.2, alors le résultat du calcul incrémental de l'enveloppe convexe `(CH m)` est planaire.

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

Le prédicat `expf` (resp. `eqc`), proposé aussi dans [18], exprime que deux brins `x` et `y` appartiennent à la même face (resp. à la même composante connexe) dans la carte `m`. Notons que la k-itération d'une fonction f sur un brin d, $f^k(d)$ est écrite `Iter f k d` dans notre système.

```
Definition expf (m:fmap)(x:dart)(y:dart) : Prop :=
  inv_hmap m /\ exd m x /\ exists i:nat, Iter (cF m) i x = y.
```

```
Fixpoint eqc (m:fmap)(x:dart)(y:dart) {struct m} : Prop :=
  match m with
     V => False
  |  I m0 x0 _ _ => x=x0 /\ y=x0 \/ eqc m0 x y
  |  L m0 _ x0 y0 => eqc m0 x y \/ eqc m0 x x0 /\ eqc m0 y0 y
     \/ eqc m0 x y0 /\ eqc m0 x0 y
 end.
```

La preuve de la planarité utilise les critères de planarité établies dans [18, 19] ainsi que la classification des brins. Un des lemmes caractérisant la planarité est présenté ci-dessous :

```
  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))).
```

Ce lemme caractérise ce qui est requis pour qu'une carte libre `m` dans laquelle nous lions x à y à la dimension `zero` soit planaire. Une telle carte libre est planaire si et seulement si la carte `m` est planaire et soit x et y n'appartiennent pas à la même composante connexe, soit il existe un chemin dans la face de l'image de `x` par la fonction de clôture `cA_1` à la dimension `one` à y. Cette caractérisation requiert manifestement certaines préconditions, notamment que m vérifie la propriété `inv_hmap` et que x et y vérifient la précondition `prec_L` requise pour lier deux brins ensemble à la dimension `zero`.

6.6 Dénombrement des faces et des composantes connexes

Les deux dernières propriétés topologiques que nous cherchons à établir sont que le nombre de composantes connexes est égal à 1 et que le nombre de faces dans la carte retournée par `CH` est égal à 2 (plus le nombre de brins isolés). Notons que cela n'a lieu que lorsque la carte initiale contient au moins deux brins. Ces deux propriétés sont énoncées en utilisant les fonctions `nc` et `nf` qui calculent respectivement le nombre de composantes connexes et le nombre de faces d'une carte (cf. [18] pour plus de détails). Ces deux propriétés sont équivalentes l'une de l'autre.

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

Cette première propriété indique que, dès lors que le nombre de brins dans la carte `m` est supérieur ou égal à 2, le nombre de composantes connexes dans l'enveloppe convexe calculée par `(CH m)` est égal à 1 plus le nombre de brins isolés (les brins noirs qui sont à l'intérieur de celle-ci). La preuve procède par induction sur la structure des cartes et elle mène à s'occuper de nombreux cas complexes.

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

La seconde propriété indique que, aussitôt que le nombre de brins dans la carte `m` est supérieur ou égal à 3, le nombre de faces dans l'enveloppe convexe calculée par `(CH m)` est égal à 2 (la face interne et la face externe) plus le nombre de brins isolés.

Cependant, la preuve de ces propriétés s'avère être très fastidieuse à cause de notre représentation de données actuelle. Ainsi, aucune de ces deux propriétés n'est formellement prouvée en Coq. En fait, de telles preuves sont faisables mais très compliquées car nous avons à manipuler une explosion combinatoire du nombre de cas à prouver. La plus grosse difficulté réside dans le dénombrement exact de brins, sommets, arêtes, faces et composantes connexes durant le calcul de l'enveloppe convexe lors de l'insertion d'un nouveau brin par la fonction `CHID`. Cela exigerait d'utiliser la définition inductive des prédicats `expf` et `eqc` à plusieurs niveaux dans les motifs de `CHID` tels que `(L (L (I (I (CHID m0 mr x p max) ...)`.

Néanmoins, l'équivalence de ces deux propriétés a été prouvée non seulement à l'aide de la formule d'Euler mais aussi grâce à la gestion des brins et de leurs catégories par des ensembles finis. Pour cela, il faut considérer un 7-uplet d'ensembles finis pour les 7 classes de brins, et, dans l'induction sur `m`, expliciter son évolution à chaque transition par un `I` ou un `L`.

Nous nous concentrons à présent sur les propriétés géométriques nécessaires pour prouver que notre algorithme calcule effectivement une enveloppe convexe.

Chapitre 7

Propriétés géométriques

Les principales propriétés géométriques que nous prouvons sont que les brins sont plongés de façon cohérente dans le plan et que la carte libre résultante vérifie bien la propriété de convexité.

7.1 Unicité et équivalence de l'existence du brin gauche et du brin droit

La première propriété que nous cherchons à établir s'occupe de l'unicité et de l'équivalence de l'existence de deux brins : le brin gauche et le brin droit.

Pour prouver l'*unicité*, nous supposons qu'il y a deux brins gauches et nous prouvons alors qu'ils sont égaux. Nous utilisons pour cela la propriété de convexité (cf. section 7.3) ainsi que la visibilité des deux brins. Cela amène à six triplets de brins dont l'orientation contredit soit la propriété 5 soit la propriété 5 bis de Knuth. Le théorème pour le brin gauche s'écrit comme suit :

```
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.
```

Un théorème similaire est prouvé pour le brin droit.

La preuve de l'*équivalence de l'existence* du brin gauche et du brin droit est exprimée par le théorème `exd_left_right_dart` (et sa réciproque) qui dit que s'il existe un brin gauche dans `m` alors il existe aussi un brin droit. Notons que ces deux brins peuvent aussi ne pas exister (quand le nouveau brin se trouve à l'intérieur de l'enveloppe convexe). Par

conséquent, la propriété la plus forte que nous démontrons est que si l'un de ces deux brins existe alors l'autre existe aussi.

```
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).
```

La preuve procède par *itération* sur les brins le long de la face. Elle repose sur la propriété que la face est bornée, c'est-à-dire que le nombre de ses brins est fini et qu'il peut être calculé en utilisant une induction nœtherienne comme dans [18]. Concrètement, cela signifie que le nombre d'itérations de `cF` sur un brin `d` nécessaires pour retourner de nouveau sur `d` est connu à l'avance. Notons que la k-itération d'une fonction f sur un brin d, $f^k(d)$ est écrite `Iter f k d` dans notre système. Un des lemmes les plus importants dont nous avons besoin pour prouver le théorème ci-dessus est le suivant :

```
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 -> (forall (j:nat), j <= i ->
  ~ right_dart m p (A m zero (Iter (cF m) j d))) ->
  visible m (Iter (cF m) i d) p.
```

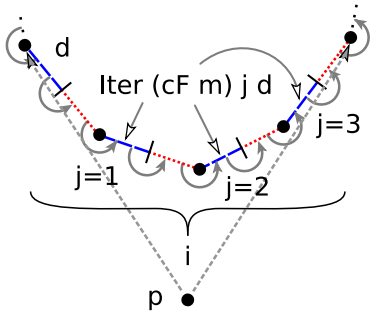


Figure 7.1 - Itération dans la face du brin `d` : tous les brins traversés sont visibles depuis `p`

Ce lemme indique que s'il existe un brin bleu visible depuis le point `p` que nous voulons insérer, alors tant que nous nous déplaçons autour de la face et que nous ne trouvons pas le brin droit, tous les brins traversés sont visibles depuis `p`. Cette propriété est illustrée à la figure 7.1.

7.2 Plongement

Nous prouvons ici que les brins sont bien plongés par rapport à leurs liaisons. Pour cela, nous définissons d'abord la propriété `well_emb` qui est aussi utilisée dans les préconditions de la fonction `CH` (cf. section 4.2). Pour chaque brin de la carte, son plongement doit être différent de celui de son successeur ou de son prédécesseur à la dimension `zero` mais il doit être identique à celui de son successeur ou de son prédécesseur à la dimension `one`. De plus, tous les autres brins doivent aussi avoir un plongement différent :

```
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).
```

Cette définition est illustrée à la figure 7.2. A gauche, nous avons un brin bleu `x` avec son 0-successeur `x0` et son 1-prédécesseur `x_1`, et à droite, un brin rouge `x` avec son 1-successeur `x1` et son 0-prédécesseur `x_0`.

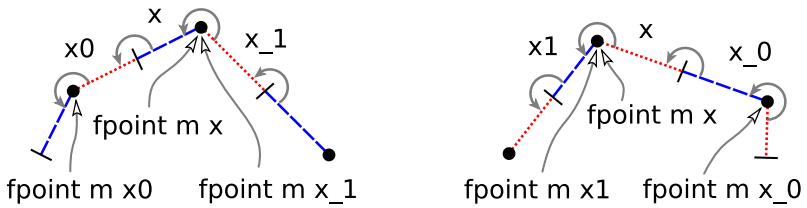


Figure 7.2 - Le bon plongement des brins par rapport à leurs liaisons

Alors, nous définissons le théorème :

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

Pour le prouver, nous utilisons de nouveau notre classification des brins (cf. section 6.1) et nous prouvons que les brins conservent le même plongement durant toute l'exécution du programme, à chaque appel récursif de `CHID`.

7.3 Démonstration de la convexité

Nous définissons d'abord la propriété géométrique fondamentale de l'enveloppe convexe : la notion de *convexité*. Elle utilise le prédicat d'orientation de Knuth `ccw`. Elle se décrit ainsi : une carte `m` est convexe si tout triplet de points de `m`, formé par le plongement `px` d'un brin bleu `x` de `m`, par celui `px0` de son successeur `x0` à la dimension `zero` et par celui de n'importe quel autre brin `y` de `m` dont le plongement `py` est différent de `px` et de `px0`, est orienté dans le sens trigonométrique (`ccw px px0 py`). Une définition similaire pourrait être exprimée sur les brins rouges vu que ceux-ci sont plongés sur les mêmes points que les brins bleus.

```
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.
```

Le théorème `convex_CH` exprime cette propriété de convexité. Elle déclare qu'à condition que la carte initiale `m` vérifie les préconditions, la carte finale `(CH m)` est effectivement convexe. Plus précisément, la carte finale représente un polygone qui est l'enveloppe convexe contenant quelques brins isolés à l'intérieur :

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

La preuve de ce théorème utilise la définition `well_emb` ainsi que les propriétés inhérentes au prédicat d'orientation de Knuth.

Notons que les preuves concernant la préservation de l'invariant des hypercartes (exprimée par le prédicat `inv_hmap`), l'obtention d'un ensemble de polygones topologiques (`inv_poly`), la planarité de l'enveloppe convexe (`planar`), le respect du plongement des brins par rapport à leurs liaisons (`well_emb`) et la convexité de l'enveloppe convexe (`convex`), c'est-à-dire aussi bien des preuves topologiques que géométriques, ont pu être démontrées individuellement sur la fonction d'insertion `CHID` et la fonction d'initialisation `CH2`. Cependant, il a été nécessaire de les prouvées simultanément sur la fonction de récursion `CHI`, comme cela est indiqué dans le lemme ci-dessous. En effet, chacun des lemmes précédents, exprimé sur `CHID` et appelé lors de `CHI`, requiert en précondition l'ensemble des autres prédicats.

```
Lemma inv_hmap_inv_poly_planar_well_emb_convex_CHI :
  forall (m1 m2 : fmap)(max:dart), prec_CH m1 ->
  inv_hmap m2 -> inv_poly m2 -> planar m2 -> well_emb m2 ->
```

```
convex m2 -> [...] -> inv_hmap (CHI m1 m2 max) /\
inv_poly (CHI m1 m2 max) /\ planar (CHI m1 m2 max) /\
well_emb (CHI m1 m2 max) /\ convex (CHI m1 m2 max).
```

Bilan de la partie II

Dans cette première approche, nous avons décrit formellement un algorithme incrémental de calcul d'enveloppe convexe d'un ensemble fini de points du plan, basé sur une modélisation à base topologique et conçu par induction structurelle sur des hypercartes en Coq. Ce système nous a permis à la fois d'en extraire un prototype de construction exécutable en OCaml et de le certifier formellement. Nous avons ainsi pu démontrer sa terminaison et mettre en évidence de nombreuses propriétés topologiques et géométriques. Cette première expérience a donné lieu à une publication internationale [7].

Le travail effectué a en effet permis de concevoir rapidement un algorithme fonctionnel et de prouver sa correction totale à l'aide de Coq. La terminaison de l'algorithme est évidente par construction inductive et les propriétés qui convainquent de sa correction partielle sont prouvées, à une exception près. Celle-ci concerne l'unicité du polygone représentant l'enveloppe convexe, qui est très technique et compliquée mais qui semble à notre portée moyennant un temps de travail conséquent. Nous avons extrait et rendu opérationnel le programme correspondant en OCaml, ce qui est rassurant quant au caractère concret de ce travail. Nous disposons également d'un outil de visualisation des enveloppes convexes dans lequel notre programme est intégré.

Nous avons ainsi pu tester nos idées sur la conception et la certification d'algorithmes géométriques. La spécification d'un algorithme de construction d'enveloppe convexe est pour nous une sorte de *benchmark* permettant de vérifier si notre bibliothèque de spécification d'hypercartes complétée par des prédicats géométriques est bien adaptée à notre objectif. Les résultats obtenus jusqu'ici indiquent que les cartes combinatoires orientées sont une structure de données particulièrement bien adaptée à la description et preuve de correction de programmes typiques de la géométrie algorithmique.

	bibliothèque basique	Enveloppe convexe
Nombre de définitions	142	52
Nombre de lemmes/théorèmes	550	409
Nombre de lignes de spécifications	3013	1620
Nombre de lignes de preuves formelles	28646	17902

Figure 7.3 - Tableau représentant la taille du développement en Coq

La figure 7.3 indique la taille du développement en faisant une distinction entre la taille des spécifications et la taille des preuves (le ratio est d'environ 1 sur 10). La bibliothèque basique correspond aux preuves et spécifications déjà existantes et présentées dans [18]. Le nombre de spécifications et de preuves développées pour la preuve formelle de l'algorithme de calcul de l'enveloppe convexe est résumé dans la seconde colonne. Les fichiers Coq du développement sont disponibles à [8].

Troisième partie

Algorithme par recherche nœthérienne

Sommaire

Chapitre 8

Structure du programme en Coq

Dans ce chapitre, nous décrivons notre deuxième approche de l'algorithme incrémental de calcul d'enveloppe convexe avec des hypercartes. Nous abordons cette deuxième version d'une manière plus traditionnelle en adoptant le principe de recherche par voisinage. Lors de l'insertion d'un nouveau point, l'algorithme recherche les arêtes visibles depuis celui-ci et les deux sommets gauche et droit par un déplacement le long de l'enveloppe convexe à partir d'un brin référence. Cette version correspond exactement à la version usuelle de l'algorithme incrémental [11, 6, 10]. De plus, nous utilisons cette fois-ci les deux opérations de plus haut niveau, `Merge` et `Split`, définies à la section 2.2.5, pour fusionner et couper des orbites.

Nous définissons tout d'abord un nouveau type `mapdart` des couples formés d'une carte et d'un brin de référence permettant de repérer une face. En effet, comme cette deuxième version de l'algorithme incrémental n'utilise pas d'induction sur la carte, il est nécessaire d'avoir un brin de départ pour commencer à chaque fois le parcours autour du polygone représentant l'enveloppe convexe. De plus, comme nous n'utilisons plus les catégories de brins, le brin de référence d'une carte sera toujours équivalent à un brin bleu de la première version. Ainsi, en utilisant, à partir du brin de référence, la fonction `expf` permettant de savoir si un brin se trouve dans la même face qu'un autre, il sera toujours possible de connaître le sens d'un brin, c'est-à-dire de savoir si l'intérieur du polygone auquel il appartient se situe à sa gauche ou à sa droite (ce qui est indispensable lors des tests d'orientation dans le plan).

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

Dans cette deuxième version, nous retrouvons les trois opérations `CH`, `CH2` et `CHI`, avec un profil légèrement différent dû au type `mapdart` et naturellement avec une écriture différente. Grâce à l'utilisation de `Merge` et de `Split`, nous obtenons un code plus intuitif et plus proche de celui habituellement utilisé dans un modeleur.

8.1 Préconditions

Comme pour la première version, il est nécessaire ici de définir précisément les *préconditions* que la carte initiale `m` doit satisfaire avant le début du programme. Ce sont exactement les même préconditions qu'auparavant. Elles sont de nouveau regroupées dans une définition `prec_CH` possèdant quatre prédicats et définie de la façon suivante :

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

L'hypercarte `m` doit toujours vérifier l'invariant `inv_hmap`, énoncé dans la section 2.2.4, permettant d'avoir des cartes cohérentes. Elle ne doit pas avoir de liaisons entre les brins (i.e. pas d'occurrence du constructeur `L`). Cette propriété est encore une fois exprimée par le prédicat `linkless`. Le prédicat `is_neq_point` est une version simplifiée du prédicat `well_emb` car il exprime simplement que tous les brins passés en entrée de l'algorithme doivent avoir un plongement différent. Il ne contient pas de condition concernant le plongement de brins appartenant au même sommet ou à la même arête. Nous supposons également que trois brins ayant des plongements différents ne peuvent pas être plongés dans trois points colinéaires. Cette propriété est exprimée par le prédicat `is_noalign` dont la définition (cf. chapitre 10) est semblable à la version précédente.

8.2 Fonction principale `CH`

Nous définissons la fonction principale `CH` qui calcule l'enveloppe convexe d'un ensemble fini de points dans le plan représenté par une carte `m`. Si la carte initiale `m` possède un seul brin `x`, elle renvoie le couple formé de `m` et de `x`. Si elle possède au moins deux brins, elle procède comme suit : la fonction `CH` construit un premier polygone convexe avec deux des brins impliqués en utilisant la fonction `CH2` (figure 4.2) puis elle appelle la fonction récursive `CHI`. Sinon elle retourne simplement le couple formé de l'ensemble de brins initial et du brin `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)
  | _ => (V,nil)
  end.
```

8.3 Fonction d'initialisation CH2

La fonction CH2 fonctionne de la même manière qu'à la version précédente (cf. section 4.4) en construisant la carte combinatoire orientée illustrée à la figure 4.2. Elle introduit deux nouveaux brins, nommés max+1 et max+2, qu'elle lie convenablement avec les deux brins passés en entrée, x1 et x2. Simplement, pour lier les brins entre eux, elle utilise la fonction Merge à la place du constructeur L afin de préserver l'homogénéité dans chaque définition des fonctions de calcul. De plus, étant donné que, dans cette deuxième approche, l'enveloppe convexe est représentée par une face, la fonction CH2 retourne le couple formé par la carte et le brin de référence x1.

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

8.4 Fonction de récursion CHI

La fonction CHI est presque identique à celle de la version précédente (cf. section 4.5), elle fonctionne toujours par récursion structurelle sur la carte initiale m. Elle traite les brins de m un par un et elle fabrique pour chacun d'entre eux une nouvelle enveloppe convexe en utilisant la fonction d'insertion CHID et le paramètre max. Néanmoins, comme pour toutes les fonctions de cette deuxième approche, elle prend en paramètre et retourne un objet de type mapdart des couples formés d'une carte et d'un brin de référence permettant de repérer une face.

```
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.
```

8.5 Recherche des brins gauche et droit

Nous définissons les fonctions search_left et search_right qui recherchent respectivement le brin gauche et le brin droit dans une face (m,d) par rapport à un point p. Pour cela, chacune parcourt l'ensemble des brins de la face de manière récursive.

Elles utilisent la construction Function de Coq qui permet de travailler en récursivité générale et d'assurer la terminaison d'une fonction par la décroissance à chaque appel récursif d'une mesure entière, spécifiée à la suite du mot-clé measure dans le code.

Ces deux fonctions utilisent un entier `i` qui augmente d'une unité à chaque appel récursif (il vaut 0 lors du premier appel). Si le degré de la face (c'est-à-dire le nombre de brins qu'elle contient), défini à l'aide du prédicat `degreef`, est inférieur ou égal à `i`, c'est que nous avons parcouru tous les brins de la face et qu'il n'existe pas de brin gauche dans celle-ci. Sinon, on teste si le `i`-ème successeur dans la face de `d` correspond au brin gauche.

```
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).
```

8.6 Fonction d'insertion `CHID`

La fonction d'insertion `CHID` calcule l'enveloppe convexe d'un polygone convexe `md` et d'un nouveau point `x`. Elle fonctionne comme suit :

Elle recherche le brin gauche `l` et le brin droit `r` dans la carte `m` par rapport au nouveau brin `x` et son plongement `p`. Si ceux-ci sont nuls, c'est que `p` est à l'intérieur de l'enveloppe convexe. Il suffit d'insérer `x` dans la carte. Sinon, la fonction procède en 6 étapes illustrées à la figure 8.1 :
1 - Elle sépare le brin gauche `l` de son successeur `l0 := cA m zero l` à la dimension `zero` (elle coupe son arête).
2 - Si le brin `l0` est différent de `r`, elle sépare `r` de son prédécesseur `r_0 := cA_1 m zero r` à la dimension `zero` (elle coupe également son arête).
3 - Elle insère les deux nouveaux brins `x` et `max` plongés tous les deux sur le point `p`.
4 - Elle lie `max` à `x` à la dimension `one` (elle les met dans le même sommet).
5 - Elle lie `l` à `max` à la dimension `zero` pour créer une nouvelle arête.
6 - Elle lie `x` à `r` à la dimension `zero` pour refermer l'enveloppe convexe.

```
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
  let r := search_right m (cA m zero d) p 0 in
  let l0 := cA m zero l in
  let r_0 := cA_1 m zero r in
  if (eq_dart_dec l nil) then (I m x p, d) else
    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*)
```

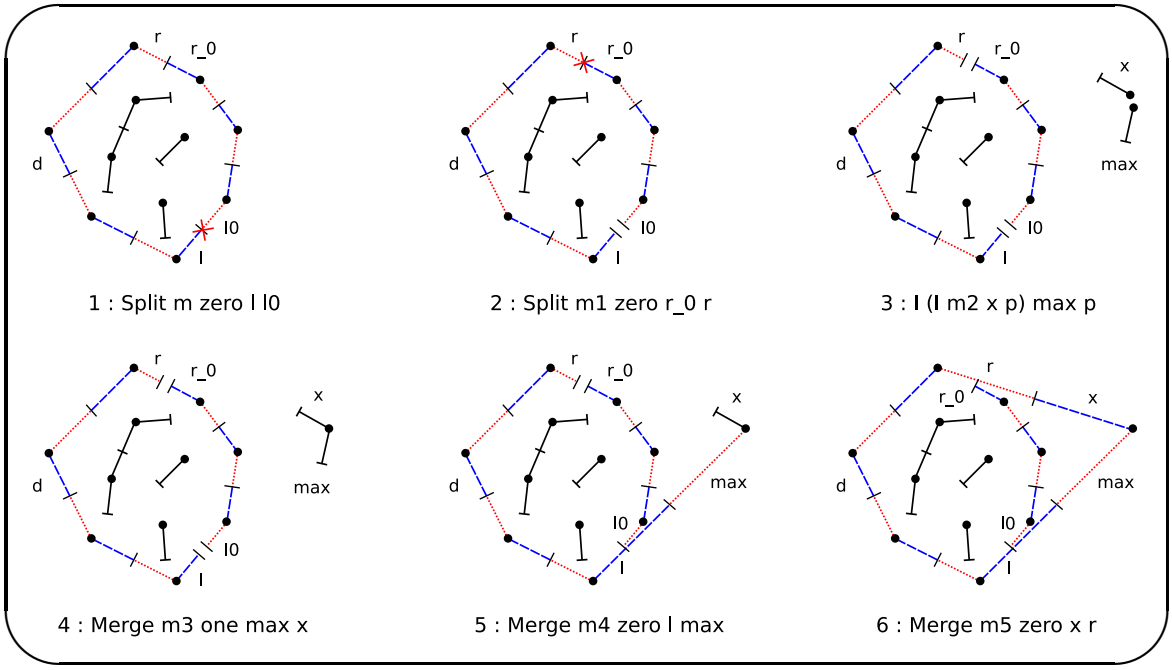


Figure 8.1 - Les six étapes de la fonction CHID

On voit que cette définition de la fonction d'insertion est conforme à celle de l'algorithme incrémental classique (cf. section 3.4). Elle recherche les brins gauche et droit, puis elle supprime ceux visibles depuis le nouveau point, et elle crée les deux nouvelles arêtes pour refermer l'enveloppe convexe.

Notons toutefois que, dans cette version, les chaînes de brins coupées restent dans la carte à l'intérieur du polygone représentant l'enveloppe convexe. Le nombre de faces et de composantes connexes peut donc augmenter d'au plus une unité à chaque insertion d'un nouveau brin. Ces chaînes de brins peuvent être supprimées dans une opération qui engloberait la fonction CHID.

Chapitre 9

Propriétés topologiques

La structure des preuves, notamment pour le passage dans la fonction d'insertion CHID, suit celle du programme. Ainsi, nous prouvons successivement chacune des propriétés suivantes sur les six cartes de la définition précédente.

9.1 Préservation de l'invariant des hypercartes

Encore une fois, le premier théorème important que nous voulons prouver est la préservation de l'invariant `inv_hmap` tout au long du programme, de la carte initiale `m` à la carte finale `(CH m)`. Pour rappel, ce prédicat indique qu'un brin ne peut pas être inséré deux fois dans la même carte et que les brins doivent être présents dans la carte avant d'être liés ensemble tout en respectant l'absence de liaison préalable et la conservation d'une ouverture dans l'orbite (cf. section 2.2.4).

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

La preuve de ce théorème est immédiate. On procède en 6 étapes en suivant le code du programme. Chacune de ces étapes est facile à prouver grâce aux propriétés de conservation de l'invariant sur le constructeur `I` et les fonctions `Merge` et `Split`.

9.2 Les k-orbites sont toutes des involutions

Une des propriétés de l'enveloppe convexe est d'être un polygone où chaque arête et chaque sommet sont de degré 2. Cette propriété est exprimée à l'aide du prédicat `inv_gmap` qui dit que pour tout brin `x` de la carte `m`, quelle que soit la dimension `k`, chaque orbite

est une involution. Cela rappel implicitement la notion de carte généralisée (ou g-carte) où, entre autre, toutes les permutations α_i sont des involutions.

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

Notons qu'une carte vérifiant le prédicat `inv_gmap` admet la propriété suivante (exprimée dans le lemme `eq_cA_cA_1`) : pour tout brin `x` de la carte `m` le successeur dans la clôture de l'orbite est égal à son prédécesseur.

```
Lemma eq_cA_cA_1 : forall (m:fmap)(k:dim)(x:dart),
  inv_hmap m -> inv_gmap m -> cA m k x = cA_1 m k x.
```

La propriété précédente est très utile et fortement exploitée dans la preuve du théorème ci-dessous.

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

Pour le prouver, il suffit d'observer l'évolution de chaque lien pour chaque brin contenu dans la carte. Ainsi, nous regardons à chaque étape, pour les deux dimensions `zero` et `one`, quel est le successeur d'un brin. Nous obtenons ainsi une vingtaine de lemmes du genre :

```
Lemma cA_m1_zero_l : cA m1 zero l = l.
Lemma cA_m5_zero_l : cA m5 zero l = max.
```

Le premier exprime que le successeur de `l` à la dimension `zero` dans la carte `m1` est le brin `l` lui-même, et le deuxième que son successeur dans `m5` correspond lui au brin `max`.

Un exemple complet de preuve d'un lemme étudiant la conservation du prédicat `inv_gmap` lors d'un appel à la fonction d'insertion `CHID` est donné dans l'annexe C.

9.3 Les k-orbites n'ont pas de point fixe

Une autre propriété du polygone représentant l'enveloppe convexe est que celui-ci ne contient pas de point fixe. Cette propriété est exprimée à l'aide du prédicat `inv_poly`. Ainsi, pour tout brin `x` appartenant à la même composante connexe que `d` dans la carte `m`, quelle que soit la dimension `k`, aucune de ses `k`-orbites n'admet de point fixe.

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

Cette définition est plus générale que nécessaire puisqu'elle prend en hypothèse deux brins appartenant à la même composante connexe alors qu'il suffirait de prendre deux brins appartenant à la même face.

Pour rappel, si un brin `x` appartient à la même face qu'un brin `y` alors ils appartiennent tous les deux à la même composante connexe :

```
Lemma expf_eqc : forall (m:fmap)(x:dart)(y:dart),
  inv_hmap m -> expf m x y -> eqc m x y.
```

Nous prouvons alors le théorème suivant :

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

9.4 Planarité de l'enveloppe convexe

Comme auparavant, nous pouvons maintenant vérifier que le polygone résultant est vraiment planaire. Nous utilisons pour cela la propriété de planarité `planar` définie précédemment à la section 6.5.

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

La preuve de ce théorème utilise les critères de planarité établies dans [18, 19] pour les opérations `Merge` et `Split`. Un des lemmes caractéristiques est présenté ci-dessous :

```
Theorem planarity_crit_Merge0 : forall (m:fmap)(x y:dart),
  inv_hmap m -> exd m x -> exd m y -> ~ expe m x y ->
  (planar (Merge m zero x y) <->
  (planar m /\ (~ eqc m x y \/ expf m (cA_1 m one x) y))).
```

Les prédicats `eqc`, `expe`, `expv` et `expf`, proposés dans [18], expriment respectivement que deux brins appartiennent à la même composante connexe, à la même arête, au même sommet et à la même face.

Ce lemme caractérise ce qui est requis pour qu'une carte libre `m` dans laquelle nous lions `x` à `y` à la dimension `zero` soit planaire. Une telle carte est planaire si et seulement si la carte `m` est planaire et soit `x` et `y` n'appartiennent pas à la même composante connexe, soit il existe un chemin dans la face de l'image de `x` par la fonction de clôture `cA_1` à la dimension `one` à `y`. Cette caractérisation requiert certaines préconditions, notamment que `m` vérifie la propriété `inv_hmap` et que `x` et `y` soient présents dans la carte `m` et qu'ils n'appartiennent pas à la même arête.

9.5 Dénombrement des faces et des composantes connexes

Les deux dernières propriétés topologiques que nous voulons établir sont que le nombre de composantes connexes augmente au plus d'une unité à chaque étape, tout comme le nombre de faces. Ces deux propriétés sont exprimées en utilisant les fonctions `nc` et `nf` qui retournent justement le nombre de composantes connexes et le nombre de faces d'une carte.

L'évolution de `nc` au travers de la fonction `CHID` est donnée par le lemme suivant :

```
Lemma nc_1 : forall (md:mapdart)(x:dart)(t:tag)
  (p:point)(max:dart),
  let m := fst md in let d := snd md in
  let l := search_left m d p 0 in
  let r := search_right m (cA m zero d) p 0 in
  let l0 := (cA m zero l) in
  nc (fst (CHID md x t p max)) =
    if (l0==r) then (nc m) else (nc m + 1).
```

L'évolution de `nf` au travers de la fonction `CHID` est donnée par le lemme suivant :

```
Lemma nf_1 : forall (md:mapdart)(x:dart)(t:tag)
  (p:point)(max:dart),
  let m := fst md in let d := snd md in
  let l := search_left m d p 0 in
  let r := search_right m (cA m zero d) p 0 in
  let l0 := (cA m zero l) in
  nf (fst (CHID md x t p max)) =
    if (l0==r) then (nf m) else (nf m + 1).
```

L'expression `fst (CHID md x t p max)` représente la carte contenant l'enveloppe convexe et les chaînes de brins contenues à l'intérieur de celle-ci.

Les preuves se déroulent de manière simple grâce aux propriétés établies dans la bibliothèque sur `Merge` et `Split` qui facilitent les dénombrements des différentes cellules topologiques de la carte.

Chapitre 10

Propriétés géométriques

Les propriétés géométriques que nous cherchons à démontrer sont que les brins sont plongés de façon cohérente dans le plan selon leurs liaisons et l'algorithme élaboré construit bien un polygone vérifiant la définition de la convexité (cf. définition 6).

10.1 Plongement

Nous prouvons d'abord que les brins sont plongés correctement au regard de leurs liaisons. C'est-à-dire, lorsque deux brins `x` et `y` sont dans le même sommet, ils ont le même plongement mais lorsqu'ils sont dans la même arête, leurs plongements sont différents. Ceci est exprimé dans la définition `is_well_emb` suivante :

```
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
  (expv m x y -> px = py) /\ (expe m x y -> px <> py).
```

Cette définition est équivalente à celle de la première version (cf. section 7.2) mais exprimée de manière légèrement différente. L'utilisation des prédicats `expv` et `expe` rend la spécification plus lisible et plus synthétique, c'est-à-dire qu'elle est d'un plus haut niveau d'abstraction.

10.2 Points distincts et non-colinéaires

Les plongements de deux brins `x` et `y` qui ne sont pas dans le même sommet sont différents.

```
Definition is_neq_point (m:fmap) : Prop :=
  forall (x:dart)(y:dart), exd m x -> exd m y ->
  let px := fpoint m x in let py := fpoint m y in
  ~ expv m x y -> px <> py.
```

Cette propriété est exprimée par le théorème suivant :

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

Et enfin, une de nos préconditions de départ, vérifiée tout au long du programme, dit que nous n'avons jamais trois points alignés. Il s'agit de la propriété `is_noalign` exprimée dans le théorème ci-dessous :

```
Definition is_noalign (m:fmap) : Prop :=
  forall (x:dart)(y:dart)(z:dart),
  exd m x -> exd m y -> exd m z ->
  let px := fpoint m x in let py := fpoint m y in
  let pz := fpoint m z in px <> py -> px <> pz ->
  py <> pz -> ~ align px py pz.

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

10.3 Démonstration de la convexité

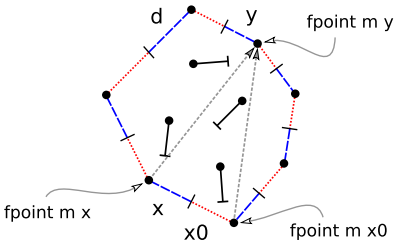


Figure 10.1 - La propriété de convexité

Finalement, nous prouvons la propriété la plus attendue, c'est-à-dire la propriété de convexité. La définition `is_convex` qui exprime cette propriété de convexité se décrit maintenant grâce à `expf` : une face repérée par la carte `m` et le brin `d` est convexe si pour tout brin `x` appartenant à celle-ci et pour tout brin `y` dont le plongement `py` est différent du plongement `px` de `x` et de celui `px0` de son 0-successeur `x0 := cA m zero x`, le triplet de points formé par `px`, `px0` et `py` est bien orienté dans le sens trigonométrique (i.e. `ccw px px0 py`) :

```
Definition is_convex (m:fmap)(d:dart) : Prop :=
  forall (x:dart)(y:dart), expf 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.
```

Le théorème `convex` exprime la propriété de convexité :

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

Chapitre 11

Implémentation en C++

Comme pour la première version, nous extrayons tout d'abord un programme exécutable en OCaml grâce au mécanisme automatique de Coq. Nous réutilisons à cet effet notre interface graphique. Cela nous conforte dans notre démarche en nous apportant un outil de vérification permettant de tester l'utilisation pratique de notre algorithme et de la structure de données issue de notre bibliothèque.

Nous dérivons finalement une implémentation d'un programme optimisé de notre spécification dans le langage impératif C++. Nous utilisons la bibliothèque CGoGN de notre équipe pour la représentation des cartes. Cela nous permet d'intégrer notre travail à cette plate-forme. Les diverses fonctions de calcul sont reproduites à l'identique et elles restent très proches du code Coq. Néanmoins, il est important de noter que cette dérivation a complètement été effectuée à la main sans utiliser un mécanisme d'automatisation. Ce programme n'est donc pas prouvé formellement. Ce travail a été réalisé à l'aide de Sylvain Théry que nous remercions chaleureusement.

11.1 Extraction en OCaml

Nous donnons ci-dessous un bref aperçu du code extrait en OCaml. Il s'avère être très proche, quasi-identique, au code initial écrit en Coq. Il reprend la même structure du programme et il garde les mêmes définitions de fonctions.

```
let cHID md x t p max =
  let m = fst md in
  let d = snd md in
  let l = search_left m d p O in
  let r1 = search_right m (cA m Zero d) p O in
  if eq_dart_dec l nil
  then Pair ((I (m, x, p)), d)
```

```
  else Pair
         ((merge
            (merge
              (merge (I ((I
                ((if eq_dart_dec (cA m Zero l) r1
                  then split m Zero l (cA m Zero l)
                  else split (split m Zero l (cA m Zero l)) Zero
                         (cA_1 m Zero r1) r1), x, p)), max, p)) One max
                x) Zero l max) Zero x r1), x)


let cH m = match m with
  | I (f0, x2, p2) ->
      (match f0 with
         | I (m0, x1, p1) ->
             cHI m0 (cH2 x1 p1 x2 p2 (max_dart m))
               (plus (max_dart m) (S (S (S O))))
         | _ -> Pair (m, nil))
  | _ -> Pair (m, nil)
```

11.2 Présentation de CGoGN

Dans le cadre de cet algorithme, nous regardons comment se rapprocher des implémentations classiques et habituelles notamment en C et C++.

CGOGN (Combinatorial and Geometric mOdeling with Generic N-dimensional Maps) [47] est une plate-forme de modélisation géométrique qui fournit un ensemble de structures de données et de fonctions permettant de développer facilement des algorithmes nécessitant des modèles topologiques.

Le but de cette plate-forme est de proposer des outils de modélisation, mais de le faire de manière innovante. Pour cela, elle propose des méthodes de programmation adaptées aux structures topologiques qui facilitent le travail du développeur en maintenant un bon niveau de performance à la fois en termes de coût mémoire et de temps de calcul. Cela a été rendu possible par l'utilisation avancée (en C++) de la généricité, autant pour les structures de données (principalement ici le brin) que pour les modèles (les cartes) et l'utilisation d'algorithmes externalisés.

11.3 Implantation des cartes

Dans CGoGN, les hypercartes sont représentées par des listes chaînées de brins. Elles sont implantées par des *itérateurs* qui sont une généralisation des pointeurs, issus de la bibliothèque standard de C++, la STL (Standard Template Library). Les itérateurs sont

utilisés pour parcourir de façon commode une série d'objets avec des opérations d'incrémentation.

Plus précisément, un brin d'une carte est une structure possédant un tableau de brins pour décrire la topologie (i.e. les liaisons des brins avec leurs successeurs et leurs précécesseurs) et un tableau de plongements qui est une classe abstraite dont on fait dériver tous les plongements que l'on veut définir. Comme nous n'avons ici qu'un plongement, nous avons simplement un pointeur vers une structure point. Dans cette classe de plongement, il y a un compteur de références qui est incrémenté lorsque l'on plonge un brin et décrémenté lorsque l'on change le plongement ou que l'on détruit le brin.

Nous illustrons à la figure 11.1 un schéma de représentation des hypercartes et des brins dans la bibliothèque CGoGN.

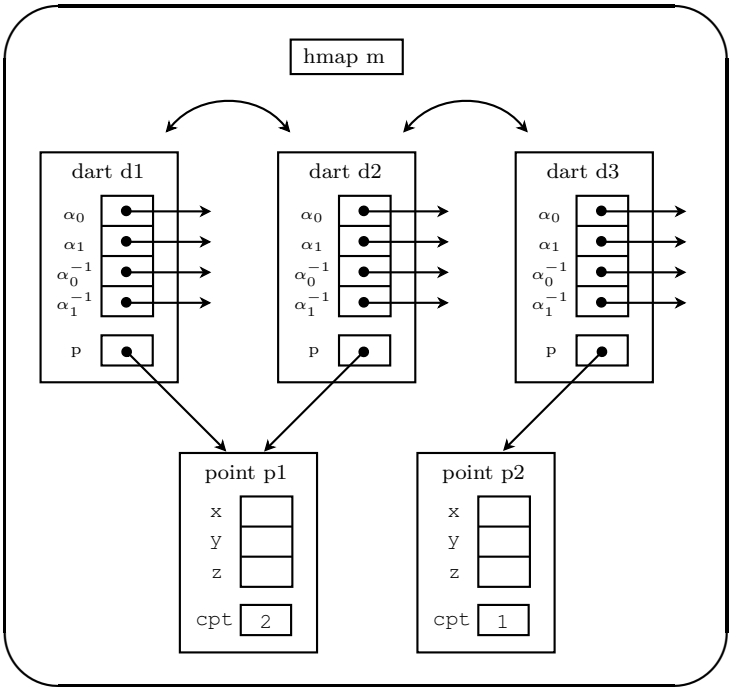


Figure 11.1 - Schéma de représentation des cartes dans CGoGN

Les hypercartes de notre spécification possèdent le type `hmap`. Nous fabriquons `mapdart` le type des structures à deux éléments et nous implémentons deux fonctions d'accès, `fstmd` et `sndmd`, et une fonction de construction `pairmd`.

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

hmap fstmd (mapdart md) { return md.m; }

hmap sndmd (mapdart md) { return md.d; }

mapdart pairmd (hmap m, dart d) {
```

```
  mapdart md;
  md.m = m;
  md.d = d;
  return md; }
```

11.4 Opérations de base

Nous implémentons d'abord les fonctions issues de notre bibliothèque de base en Coq. Cette traduction garde la forme de la spécification. Nous utilisons l'opération `alphaSew`, présente dans CGoGN, permettant de lier un brin avec un autre.

Les fonctions `Merge` et `Split` sont programmées avec effets de bord en cela qu'elles changent directement dans la carte appelée `m` les liaisons des brins concernés.

```
hmap Merge (hmap m, const dim k, const dart x, const dart y) {
m->alphaSew(k,x,cA_1(m,k,y));
return m;
}

hmap Split (hmap m, const dim k, const dart x, const dart y) {
m->alphaSew(k,x,y);
return m;
}
```

11.5 Recherche des brins gauche et droit

Puis, nous traduisons les deux fonctions `search_left` et `search_right` permettant de rechercher les brins gauche et droit. Dans cette version, elles fonctionnent par itération telles que dans la spécification en Coq. On pourrait faire mieux, dans une deuxième version, en supprimant ces itérations et en gardant une trace du brin courant dans la face (afin de ne pas tout reparcourir à chaque appel récursif), mais il faudrait s'assurer d'avoir des outils qui montreraient que ces deux versions sont les mêmes.

L'instruction `Iter (cF m) i d` permettant d'itérer `i`-fois la fonction `cF` de `d` sur `m` est remplacée ici par la fonction `cFi`. La propriété de décidabilité `left_dart_dec` est devenue la fonction `isleftdart` qui teste si un brin `di` est le brin gauche dans `m` par rapport à `p` (il en va de même pour le brin droit).

```
dart search_left (hmap m, const dart d, const point p, int i) {
if (degreef(m,d) <= i) {
return m->nil();
```

```
} else {
dart di = cFi(m,d,i);
if (isleftdart(m,di,p)) return di;
else return search_left(m,d,p,i+1);
}
}

dart search_right (hmap m, const dart d, const point p, int i) {
if (degreef(m,d) <= i) {
return m->nil();
} else {
dart di = cFi(m,d,i);
if (isrightdart(m,di,p)) return di;
else return search_right(m,d,p,i+1);
}
}
```

11.6 Calcul de l'enveloppe convexe

Finalement, nous implémentons nos fonctions de construction de l'enveloppe convexe. Elles reprennent la forme de la spécification en Coq. Notre implémentation est programmée avec des effets de bord car les fonctions modifient immédiatement les brins et les liaisons de la carte donnée en paramètre, elles ne reconstruisent pas à chaque fois une nouvelle carte.

On a vu que les brins ne sont pas représentés par des entiers mais qu'ils ont une réelle structure. Pour créer un brin, il n'est pas nécessaire de passer un identifiant (i.e. un entier). Il faut simplement appeler d'abord la fonction `gendart` qui fabrique un nouveau brin dans une carte et renvoie son identifiant (un pointeur), puis la fonction `I` qui insère ce brin dans la carte avec son plongement associé.

De plus, les fonctions `CH` et `CHI` ne procèdent pas ici par filtrage sur la carte initiale `m`. Ainsi, il faut obligatoirement extraire et supprimer explicitement chaque brin de celle-ci à l'aide des fonctions `anydart` et `deldart`. Notons que ceci pourrait également être fait en Coq.

On suppose enfin que la fonction principale `CH` est appelée avec une carte vérifiant les préconditions telles que décrites à la section 8.1.

```
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;
} // end CHID

mapdart CHI (hmap m, mapdart md) {
if (empty(m)) { return md; }
else {
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glDisable(GL_LIGHTING);
drawHM(*m,10.0f,0.7f);
drawHM(*(md.m),10.0f,0.0f);
glutSwapBuffers();
sleep(1);
// *****
dart x = anydart(m);
point p = fpoint(m,x);
m = deldart(m,x);
return CHI(m,CHID(md,p));
}
} // end CHI

mapdart CH2 (const point p1, const point p2) {
hmap m = V();
dart x1 = gendart(m);
m = I(m,x1,p1);
dart x2 = gendart(m);
m = I(m,x2,p2);
dart max1 = gendart(m);
m = I(m,max1,p1);
```

```
dart max2 = gendart(m);
m = I(m,max2,p2);
m = Merge(m,one,max1,x1);
m = Merge(m,one,max2,x2);
m = Merge(m,zero,x1,max2);
m = Merge(m,zero,x2,max1);
mapdart md = pairmd(m,x1);
return md;
} // end CH2

mapdart CH (hmap m) {
dart x1 = anydart(m);
point p1 = fpoint(m,x1);
m = deldart(m,x1);
dart x2 = anydart(m);
point p2 = fpoint(m,x2);
m = deldart(m,x2);
mapdart map = CH2(p1,p2);
return CHI(m,map);
} // end CH
```

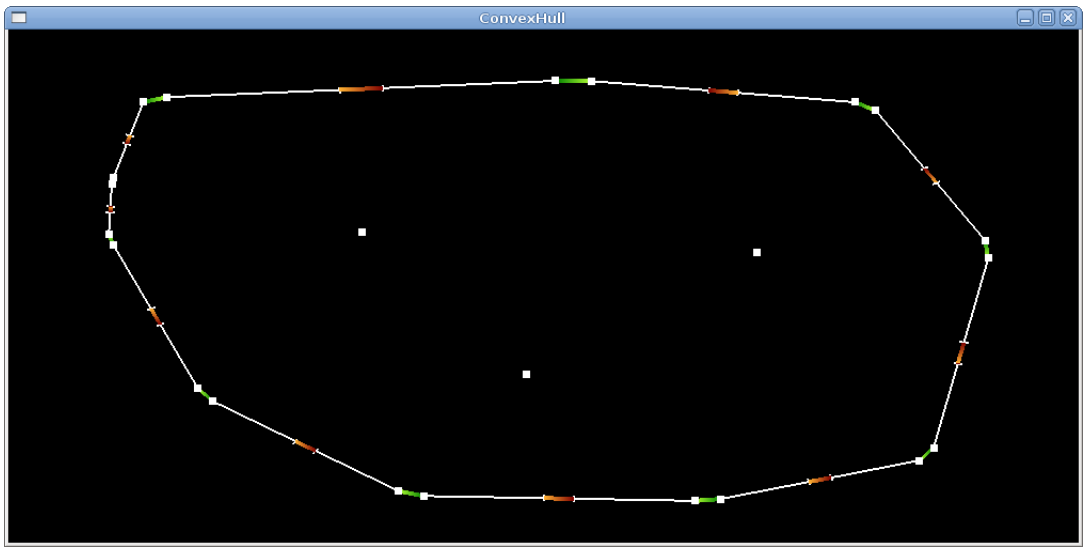


Figure 11.2 - Aperçu de l'exécution du programme en C++

L'utilisation concrète de notre algorithme implémenté en C++ ressemble à celle de l'algorithme extrait automatiquement en OCaml. Il suffit de positionner les points à l'écran et nous obtenons le résultat visible dans notre interface graphique. Nous pouvons en voir un aperçu à la figure 11.2. La complexité de la fonction `CHI` dans le pire des cas est en $O(n^2)$, où n est le nombre de points de l'ensemble initial. Elle pourrait gagner un ordre de complexité si nous remplacions les itérations dans les fonctions `search_left` et `search_right` par des fonctions de recherche par voisinage.

Bilan de la partie III

Dans cette deuxième expérience, nous avons décrit formellement un algorithme incrémental de calcul d'enveloppe convexe d'un ensemble fini de points du plan qui correspond bien plus à la version usuelle que celui de notre première approche. En effet, lors de l'insertion d'un nouveau point, il utilise le principe de recherche par voisinage pour trouver les arêtes visibles depuis celui-ci en se déplaçant le long de l'enveloppe convexe à partir d'un brin référence. Nous en avons ensuite extrait un prototype de construction exécutable en OCaml. Puis nous avons certifié formellement l'algorithme en Coq par la démonstration de plusieurs propriétés topologiques et géométriques. Finalement, nous en avons dérivé une implémentation dans le langage impératif C++.

Cette deuxième version est d'un plus haut niveau de spécification grâce notamment à l'utilisation des deux opérations `Merge` et `Split`. Elle est également beaucoup plus rapide à prouver que la première, certainement d'une part parce que les propriétés topologiques sur `Merge` et `Split` suppriment des étapes de preuve et d'autre part parce que la fonction d'insertion `CHID` suit le déroulement classique de l'algorithme incrémental. Elle ne reconstruit pas à chaque fois toute une nouvelle carte mais elle apporte simplement les modifications nécessaires, c'est-à-dire qu'à part les quelques brins ou liaisons qui sont supprimés ou ajoutés, les autres éléments de la carte ne changent pas. Nous avons non seulement prouvé les mêmes propriétés topologiques et géométriques que celles de la première version mais nous avons réussi à aller un peu plus loin du point de vue du dénombrement. En effet, nous avons pu démontrer ici que le nombre de faces et de composantes connexes augmente d'une unité au plus à chaque insertion d'un nouveau brin. Nous pourrions aussi extraire simplement le polygone représentant l'enveloppe convexe, sans les chaînes de brins internes, et obtenir ainsi une seule composante connexe.

Le passage à une implantation en langage impératif a grandement été facilité par les fonctions `Merge` et `Split`. Cette dérivation en C++ ressemble fortement à la spécification, modulo les effets de bords. Mais il faudrait pouvoir certifier également cette implémentation.

La figure 11.3 indique la taille du développement en Coq en faisant une distinction entre la taille des spécifications et la taille des preuves (le ratio est d'environ 1 sur 5). La bibliothèque basique correspond aux preuves et spécifications déjà existantes et présentées dans [19]. Le nombre de spécifications et de preuves développées pour la preuve formelle de l'algorithme de calcul de l'enveloppe convexe est résumé dans la seconde colonne.

	bibliothèque basique	Enveloppe convexe
Nombre de définitions	277	27
Nombre de lemmes/théorèmes	1005	217
Nombre de lignes de spécifications	7033	611
Nombre de lignes de preuves formelles	54952	2657

Figure 11.3 - Tableau représentant la taille du développement en Coq

Chapitre 12

Conclusion

12.1 Bilan

Nous avons mené une double expérience pour tester nos idées sur la conception et la certification d'algorithmes de construction d'enveloppe convexe dans le plan et vérifier si notre bibliothèque de spécifications d'hypercartes complétée par des prédicats géométriques est bien adaptée à cet objectif.

Le travail mené a effectivement permis de concevoir deux algorithmes fonctionnels et de prouver leur correction totale à l'aide de Coq. La terminaison des algorithmes est évidente par construction inductive et les propriétés qui convainquent de leur correction partielle sont prouvées.

Nous avons conçu deux algorithmes ayant des niveaux d'abstraction et des vitesses d'exécution différentes. Le premier, simple mais original, utilise une induction structurelle sur le type inductif `fmap` des hypercartes. Le deuxième, de plus haut niveau, utilise la recherche par voisinage et les fonctions `Merge` et `Split`, très commodes pour la manipulation de polygones et de subdivisions de surfaces en général.

Les performances de calcul de la première version, si elle était implantée en C++ par un parcours de carte avec une liste chaînée de brins, seraient tout à fait acceptables d'un point de vue de la complexité théorique. Cependant, les performances de la deuxième version sont plus proches d'une implémentation habituelle. Il faudrait dériver en langage impératif la première version afin de tenter des comparaisons de temps de calcul, et surtout, essayer de remplacer les fonctions `search_left` et `search_right` utilisant l'itération sur les brins pour gagner en rapidité.

De plus, le mariage entre propriétés topologiques et géométriques dans notre environnement est une approche singulière qu'a bien pu mettre en évidence ce travail. Dans les travaux effectués jusqu'ici sur notre bibliothèque en Coq, on se contentait simplement d'étudier l'un ou l'autre de ces deux aspects [12, 13]. Ici, ils sont explorés simultanément

et on se rend compte qu'ils ne sont pas complètement indépendants. Néanmoins, cette approche avait déjà été mené par Mota pour la spécification en B de l'alignement de droites.

Cependant, le problème de la complétude persiste. Comment être vraiment certain que les propriétés prouvées recouvrent bien l'ensemble des propriétés nécessaires et suffisantes pour certifier un algorithme de géométrie algorithmique ? Sommes-nous assez exhaustif pour envisager toutes les caractéristiques requises sur les objets considérés ? Ces questions restent ouvertes.

Finalement, nous pouvons remarquer que ce genre de travail demeure réservé aux spécialistes de preuves et du domaine abordé. En effet, nous pouvons relever d'une part les difficultés qui persistent concernant l'utilisation de Coq malgré de nombreuses améliorations, notamment l'apparition de nouvelles tactiques ou une simplification de l'écriture par l'introduction de mécanismes ingénieux tels que `Function`, et d'autre part la difficulté à bien formaliser le domaine traité, notamment concernant la définition des types et des opérations, ou l'énoncé de propriétés par de bons invariants.

12.2 Perspectives

En premier lieu, il convient de s'intéresser à la méthodologie. Celle adoptée dans ce travail consiste d'abord à spécifier le programme en Coq, ensuite à l'extraire et le tester en OCaml, puis à prouver formellement ses propriétés et finalement à le dériver dans un langage impératif. Elle constitue une bonne voie à explorer pour tenter de certifier d'autres algorithmes en géométrie algorithmique.

Nous avons vu qu'il faudrait être plus systématique dans la phase d'implantation concrète. Il faudra également formaliser cette phase. Cela passe par un modèle d'implantation qu'il faut maîtriser en Coq. Par exemple, dans notre cas, une mémoire permettant de gérer des chaînages, comme cela a pu être étudié par Mehta et Nipkow qui s'inspirent de la logique de séparation en Isabelle [37]. Autrement dit, il faudra une spécification dans laquelle les preuves pourront être conduites. Ensuite, il faudra décrire en Coq une implantation de toutes nos opérations, notamment en simulant les effets de bords et le partage de mémoire. Finalement, il faudra prouver que les propriétés des opérations abstraites sont conservées avec les opérations concrètes agissant sur le modèle de mémoire [34]. Sinon, nous pourrons aussi essayer d'automatiser le processus de dérivation en langage impératif comme cela est fait dans [4] pour un algorithme de calcul de racine carrée.

De manière concomitante, il faudra reprendre d'autres algorithmes classiques d'enveloppes convexes, comme le parcours de Graham et la marche de Jarvis, en utilisant toujours notre bibliothèque d'hypercartes, afin d'étudier l'approche de nouvelle notions géométriques comme les angles. Le plongement en 3D avec une enveloppe convexe polyédrique en est aussi la suite logique. Là, il est impératif de pouvoir gérer des subdivisions générales de surfaces et l'utilisation des hypercartes de dimension 2 prend toute sa signi-

fication. En effet, il est possible de représenter des surfaces plongées dans R^3 avec des hypercartes de dimension 2, notamment des polyèdres et donc des enveloppes convexes en 3D. Les problèmes ici viendraient surtout de prédicats topologiques et géométriques plus compliqués à définir et manipuler. Le test d'orientation se ferait sur des tétraèdres, les arêtes seraient remplacées par des faces, et les brins gauche et droit par un polygone. Enfin, d'autres algorithmes géométriques doivent être abordés. Par exemple, l'étude pour la gestion de triangulations et la construction de diagrammes de Delaunay [20] devra être poursuivie.

Il faudra aussi éviter de se limiter aux cas standard et s'attaquer aux cas généraux en acceptant des situations plus compliquées. Par exemple, pour notre problème, il faudra traiter les points alignés ou confondus lors de la construction de l'enveloppe convexe. Nous pourrions très bien nous inspirer de l'approche de Pichardie et Bertot en utilisant la méthode des perturbations [43].

Enfin, si notre traitement des prédicats géométriques grâce à une axiomatique *ad hoc*, ici celle de Knuth, est très agréable, il faudra à l'avenir aborder de front la question des approximations numériques [31]. Notre manière d'éluder la question est justifiable dans une première approche, mais il faudra intégrer dans notre cadre logique les travaux fondamentaux sur les arithmétiques exactes ou paresseuses qui sont en plein développement [32]. Plus particulièrement, quelques techniques efficaces pour vérifier statiquement la validité de certains prédicats numériques dans les algorithmes ont été développés en utilisant l'arithmétique d'intervalles et l'outil Gappa [39]. On pourra aussi s'inspirer des méthodes de perturbation déjà utilisées dans [43].

Annexe A

Preuve de l'évolution d'un brin lors de l'appel à `CHID`

Cette annexe comprend un exemple complet de preuve d'un lemme étudiant l'évolution d'un brin lors d'un appel à la fonction d'insertion `CHID`. Ce lemme indique qu'un brin `d`, visible depuis un nouveau point `px`, dont l'arête précédente est aussi visible depuis `px`, n'a pas de prédécesseur à la dimension `one`.

La preuve fonctionne par induction sur une carte `m`. On traite les trois cas issus du constructeur de base `V`, `I` et `L`. Il suffit d'éliminer à chaque fois les hypothèses de construction de `CHID`, c'est-à-dire de s'occuper de la couleur des brins et de leur visibilité par rapport au nouveau point inséré.

Dans le cas 1, la carte est vide, la simplification est immédiate. Le cas 2 concerne l'insertion d'un brin par le constructeur `I`. On procède par élimination des couleurs et de la visibilité des brins. Le cas 3 traite la liaison d'un brin à un autre par le constructeur `L`. On fait une induction sur la dimension puis on élimine les possibilités selon la visibilité des brins.

```
Lemma blue_dart_CHID_18 :
  forall (m:fmap)(mr:fmap)(x:dart)(tx:tag)(px:point)(max:dart)(d:dart),
  submap m mr -> d <> x -> blue_dart mr d ->
  visible mr d px -> visible mr (A_1 mr one d) px ->
  ~ pred (CHID m mr x tx px max) one d.
Proof.
intros m mr x tx px max da Hsub Hneq Hblue Hccw1 Hccw2.
induction m.
 (* Cas 1 : m = V *)
 unfold pred; simpl in *; tauto.
 (* Cas 2 : m = I *)
 unfold pred in *; simpl in *.
```

```
 destruct Hsub as [Hsub [Hsub1 [Hsub2 Hsub3]]].
 generalize (IHm Hsub); clear IHm; intro IHm.
 elim blue_dart_dec.
  elim invisible_dec.
   intros H_ccw H_blue.
(* == 1 == *)
simpl in *; assumption.
(* ======= *)
   elim left_dart_dec.
    intros H_left H_ccw H_blue.
(* == 2 == *)
simpl in *.
elim (eq_dart_dec x da); intro Heq.
 apply sym_eq in Heq; contradiction.
 apply IHm; assumption.
(* ======= *)
    intros H_left H_ccw H_blue.
(* == 3 == *)
simpl in *; assumption.
(* ======= *)
  elim red_dart_dec.
   elim invisible_dec.
    intros H_ccw H_red H_blue.
(* == 4 == *)
simpl in *; assumption.
(* ======= *)
    elim right_dart_dec.
    intros H_right H_ccw H_red H_blue.
(* == 5 == *)
simpl in *; assumption.
(* ======= *)
     intros H_right H_ccw H_red H_blue.
(* == 6 == *)
simpl in *; assumption.
(* ======= *)
   intros H_red H_blue.
(* == 7 == *)
simpl in *; assumption.
(* ======= *)
 (* Cas 3 : m = L *)
 unfold pred in *; simpl in *.
 destruct Hsub as [Hsub [Hsub1 Hsub2]].
 induction d; simpl in *.
  elim ccw_dec.
   intros H_ccw.
(* == 8 == *)
```

```
simpl in *; apply IHm; assumption.
(* ======= *)
   intros H_ccw.
(* == 9 == *)
apply IHm; assumption.
(* ======= *)
  elim invisible_dec.
   intros H_ccw.
(* == 10 == *)
simpl in *.
elim (eq_dart_dec d1 da); intro Heq.
 subst da; rewrite Hsub2 in Hccw2.
 apply visible_not_invisible in Hccw2; contradiction.
apply IHm; assumption.
(* ======== *)
   elim invisible_dec.
    intros H_ccw1 H_ccw2.
(* == 11 == *)
simpl in *.
elim (eq_dart_dec d1 da); intro Heq.
 apply visible_not_invisible in Hccw1;
 subst da; contradiction.
apply IHm; assumption.
(* ======== *)
    intros H_ccw1 H_ccw2.
(* == 12 == *)
simpl in *; apply IHm; assumption.
(* ======== *)
Qed.
```

Annexe B

Preuve de la convexité lors de l'appel à `CHID`

Cette annexe comprend la preuve du théorème exprimant la convexité de l'enveloppe convexe dans notre première version de l'algorithme incrémental. Elle utilise un raisonnement par cas et elle s'appuie sur les axiomes 1, 5 et 5 bis de Knuth pour prouver des contradictions selon les hypothèses. On en donne ici simplement une partie car l'ensemble de la preuve représenterait un total de 12 pages.

```
Theorem inv_convex_CHID :
  forall (mr:fmap)(x:dart)(tx:tag)(px:point)(max:dart),
  inv_hmap mr -> inv_poly mr -> well_emb mr -> inv_noalign_point mr px ->
  convex mr -> x <> nil -> max <> nil -> x <> max ->
  ~ exd mr x ->  ~ exd mr max -> convex (CHID mr mr x tx px max).
Proof.
intros mr x tx px max Hmr1 Hmr2 Hmr3 Hmr4 Hmr5.
intros Hneq1 Hneq2 Hneq0 Hexd1 Hexd2.
unfold convex.
intros da Hda1 Hda2 db Hdb1 Hp1 Hp2.
(**)
generalize Hda2; intros [Hda3 [H2 [H3 H4]]]; clear H2 H3 H4.
apply succ_zero_x_or_blue_dart in Hda3; try assumption.
Focus 2. apply submap_2_submap; try assumption. apply submap_2_refl.
elim Hda3; clear Hda3; intro Hda3; try subst da.
 (* da = x *)
 rewrite fpoint_x in *; try assumption.
 generalize Hda2; intros [Hda3 [H2 [H3 H4]]]; clear H2 H3 H4.
 apply succ_zero_x_exd_right_dart in Hda3; try assumption.
 elim Hda3; clear Hda3; intros d [Hd1 [Hd2 [Hd3 Hd4]]].
 assert (Hda3 : A (CHID mr mr x tx px max) zero x = d).
  apply x_CHID_7; try assumption.
```

```
  unfold right_dart; intuition.
 rewrite Hda3 in *.
 assert (Hd5 : fpoint mr d = fpoint (CHID mr mr x tx px max) d).
  apply inv_fpoint_CHID; try assumption.
  apply exd_not_exd_neq with mr; try assumption.
  apply exd_not_exd_neq with mr; try assumption.
  apply red_dart_CHID_11; try assumption.
 rewrite <- Hd5 in *.
(**)
generalize Hd3; clear Hd3.
unfold visible; elim blue_dart_dec; intro Hblue.
 apply red_not_blue in Hd2; try assumption; contradiction.
 intro Hd3; clear Hblue.
generalize Hd4; clear Hd4.
unfold invisible; elim blue_dart_dec; intro Hblue.
 Focus 2. apply red_A_blue in Hd2; try assumption; contradiction.
 intro Hccw; elim Hccw; clear Hccw.
  Focus 2. intro Hccw.
  assert (H1 : exd mr (A mr one d)).
   apply succ_exd_A; try assumption.
   unfold red_dart in Hd2; intuition.
  assert (H2 : exd mr (A mr zero (A mr one d))).
   apply succ_exd_A; try assumption.
   unfold blue_dart in Hblue; intuition.
  assert (H3 : fpoint mr (A mr one d) <>
   fpoint mr (A mr zero (A mr one d))).
   apply well_emb_A_zero; try assumption.
   unfold blue_dart in Hblue; intuition.
  generalize (Hmr4 (A mr one d) (A mr zero (A mr one d)) H1 H2 H3).
  intro H0; contradiction.
  intro Hd4; clear Hblue.
(* ... *)
apply ccw_axiom_1; apply ccw_axiom_1.
apply ccw_axiom_5 with (fpoint mr (A_1 mr zero d))
  (fpoint mr (A mr zero (A mr one d))).
(* ... *)
apply ccw_axiom_5_bis with (fpoint mr (A mr zero da))
  (fpoint mr (A_1 mr zero (A_1 mr one da))).
(**)
Qed.
```

Annexe C

Preuve de la conservation du prédicat `inv_gmap` lors de l'appel à `CHID`

Cette annexe comprend un exemple complet de preuve d'un lemme étudiant la conservation du prédicat `inv_gmap` lors d'un appel à la fonction d'insertion `CHID` dans notre deuxième version de l'algorithme incrémental.

Ce lemme indique que la carte `m4` de l'étape 4 vérifie bien l'invariant `inv_gmap`, définit à la section 9.2 et qui dit que pour tout brin `x` de la carte `m`, quelle que soit la dimension `k`, chaque orbite est une involution. Il fait une induction sur les deux dimensions `zero` et `one` et il applique simplement des réécritures pour montrer que les brins étudiés sont identiques.

```
Lemma inv_gmap_m4 : inv_gmap m4.
Proof.
unfold inv_gmap; intros k da Hda; induction k.
(* k = zero *)
rewrite cA_m4_zero_da; try assumption.
rewrite cA_m4_zero_da; try assumption.
apply inv_gmap_m3; apply <- exd_m4; try assumption.
apply -> exd_cA; [exact Hda | apply inv_hmap_m4].
(* k = one *)
elim (eq_dart_dec da x); intro h1.
subst da; rewrite cA_m4_one_x.
rewrite cA_m4_one_max; trivial.
elim (eq_dart_dec da max); intro h2.
subst da; rewrite cA_m4_one_max.
rewrite cA_m4_one_x; trivial.
rewrite cA_m4_one_da; try assumption.
```

```
rewrite cA_m4_one_da; try assumption.
apply inv_gmap_m3; apply <- exd_m4; try assumption.
apply -> exd_cA; [exact Hda | apply inv_hmap_m4].
rewrite cA_m4_one_da; try assumption.
rewrite cA_m3_one_da; try assumption.
intro h; apply sym_eq in h.
apply subst_cA_cA_1 in h. rewrite not_exd_cA_1 in h.
apply exd_not_nil with m4 da; try assumption.
apply inv_hmap_m4. apply inv_hmap_m2.
apply not_exd_m2_x. apply inv_hmap_m2.
apply exd_m4 in Hda; apply exd_m3_1 in Hda; try tauto.
apply <- exd_m4; assumption.
rewrite cA_m4_one_da; try assumption.
rewrite cA_m3_one_da; try assumption.
intro h; apply sym_eq in h.
apply subst_cA_cA_1 in h. rewrite not_exd_cA_1 in h.
apply exd_not_nil with m4 da; try assumption.
apply inv_hmap_m4. apply inv_hmap_m2.
apply not_exd_m2_max. apply inv_hmap_m2.
apply exd_m4 in Hda; apply exd_m3_1 in Hda; try tauto.
apply <- exd_m4; assumption.
Qed.
```

Table des figures

Bibliographie

[1] Gertrud Bauer and Tobias Nipkow. The 5 Colour Theorem in Isabelle/Isar. In *TPHOLs'02 : Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, volume 2410, pages 67–82, London, UK, 2002. Springer-Verlag. – Cité en page 3.

[2] Bruce Baumgart. A boundary representation for computer vision. In *44th AFIPS National Conference*, pages 589–596, 1975. – Cité en page 3.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions*. Theoretical Computer Science. Springer-Verlag, Berlin/Heidelberg, May 2004. 469 pages. – Cité en pages 2 et 14.

[4] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A Proof of GMP Square Root. *Journal of Automated Reasoning*, 29(3-4) :225–252, 2002. 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. – Cité en page 94.

[5] Yves Bertrand and Jean-François Dufourd. Algebraic Specification of a 3D-modeler Based on Hypermaps. *CVGIP : Graphical Models and Image Processing*, 56(1) :29–60, 1994. – Cité en page 3.

[6] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic Geometry*. Cambridge University Press, 1998. 544 pages. Translated by Hervé Brönnimann. – Cité en pages 1, 3, 26, 28 et 67.

[7] Christophe Brun, Jean-François Dufourd, and Nicolas Magaud. Designing and proving correct a convex hull algorithm with hypermaps in Coq. *CGTA : Computational Geometry - Theory and Applications*, 2010. – Cité en page 61.

[8] Christophe Brun, Jean-François Dufourd, and Nicolas Magaud. Designing and proving correct a convex hull algorithm with hypermaps in Coq : the formal development in Coq, 2010. Available from `http ://galapagos.gforge.inria.fr`. – Cité en pages 18 et 62.

[9] Robert Cori. *Un code pour les graphes planaires et ses applications*. Astérisque 27. Société mathématique de France, Paris, 1970. – Cité en pages 2 et 3.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Second Edition)*. The MIT Press, September 2001. 1202 pages. – Cité en pages 1, 3, 28 et 67.

[11] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry, Algorithms and Applications (Third Edition)*. Springer-Verlag, Berlin/Heidelberg, March 2008. 386 pages. – Cité en pages 1, 3, 26, 28 et 67.

[12] Christophe Dehlinger and Jean-François Dufourd. Formalizing the generalized maps in Coq. *Theoretical Computer Science*, 323(1-3) :351–397, September 2004. – Cité en page 93.

[13] Christophe Dehlinger and Jean-François Dufourd. Formalizing the Trading Theorem in Coq. *Theoretical Computer Science*, 323(1-3) :399–442, September 2004. – Cité en page 93.

[14] Catherine Dubois and Jean-Marc Mota. Geometric modeling with B : formal specification of generalized maps. *Journal of Scientific & Practical Computing Journal*, 1(2) :9–24, 2007. – Cité en page 3.

[15] Jean-François 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*, pages 757–761, New York, NY, USA, March 2007. ACM Press. – Cité en page 13.

[16] Jean-François Dufourd. Design and formal proof of a new optimal image segmentation program with hypermaps. *Pattern Recognition*, 40(11) :2974–2993, November 2007. – Cité en page 4.

[17] Jean-François 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*, pages 253–264, February 2008. – Cité en page 4.

[18] Jean-François Dufourd. Polyhedra genus theorem and Euler formula : A hypermap-formalized intuitionistic proof. *Theoretical Computer Science*, 403(2-3) :133–159, August 2008. – Cité en pages 2, 4, 13, 14, 51, 52, 56, 62 et 75.

[19] Jean-François 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. – Cité en pages 2, 4, 14, 51, 52, 75 et 91.

[20] Jean-François Dufourd and Yves Bertot. Formal study of plane Delaunay triangulation. Interactive Theorem Proving'2010 (In Federative Logic Conference : FLoC'2010), pages 211–226. Springer-Verlag, LNCS 6172, 2010. – Cité en pages 4, 20 et 95.

[21] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, New York, NY, USA, 1987. – Cité en pages 3 et 26.

[22] Jacques Edmonds. A Combinatorial Representation for Polyhedral Surfaces. *Notices American Mathematical Society*, 7, 1960. – Cité en pages 3, 12 et 13.

[23] Eyal Flato, Dan Halperin, Iddo Hanniel, Oren Nechushtan, and Eti Ezra. The Design and Implementation of Planar Maps in CGAL. In *WAE'99 : Proceedings of the 3rd International Workshop on Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 154–168, London, UK, 1999. Springer-Verlag. – Cité en pages 3, 12 et 13.

[24] Michel Gangnet, Jean-Claude Hervé, Thierry Pudet, and Jean-Manuel Van Thong. Incremental computation of planar maps. *SIGGRAPH Computer Graphics*, 23(3) :345–354, 1989. – Cité en page 3.

[25] Georges Gonthier. A Computer-Checked Proof of the Four Colour Theorem. Technical report, Microsoft Research, Cambridge, 2005. – Cité en page 4.

[26] Georges Gonthier. Formal Proof - The Four-Colour Theorem. *Notices of the AMS*, 55(11) :1382–1393, 2008. – Cité en pages 4, 12 et 15.

[27] Georges Gonthier and Assia Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA, 2008. – Cité en page 4.

[28] Leo Guibas and Jorge 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*, pages 221–234, New York, NY, USA, 1983. ACM. – Cité en page 3.

[29] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq Proof Assistant - A Tutorial*. INRIA, France, February 2007. Version 8.1. `http ://coq.inria.fr/V8.1/files/doc/Tutorial.pdf`. – Cité en pages 2 et 14.

[30] André Jacques. Constellations et graphes topologiques. *Combinatorial Theory and Applications*, 2 :657–673, 1970. – Cité en pages 3, 12 et 13.

[31] Nicolas Julien. Certified Exact Real Arithmetic Using Co-induction in Arbitrary Integer Base. In *FLOPS*, pages 48–63, 2008. – Cité en page 95.

[32] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom Examples of Robustness Problems in Geometric Computations. *Computational Geometry : Theory and Applications (CGTA)*, 40(1) :61–78, 2008. – Cité en pages 44 et 95.

[33] Donald Knuth. *Axioms and Hulls*, volume 606 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg, 1992. 109 pages. – Cité en pages 2, 4 et 23.

[34] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1) :1–31, 2008. – Cité en page 94.

[35] Pascal Lienhardt. Topological Models for Boundary Representation : a Comparison with n-Dimensional Generalized Maps. *Computer-Aided Design*, 23(1) :59–82, 1991. – Cité en pages 3, 12 et 13.

[36] Martti Mantyla and Reijo Sulonen. GWB : A Solid Modeler with Euler Operators. *IEEE Computer Graphics and Applications*, 2(7) :17–31, 1982. – Cité en page 3.

[37] Farhad Mehta and Tobias Nipkow. Proving Pointer Programs in Higher-Order Logic. *Information and Computation*, 199 :200–227, 2005. – Cité en page 94.

[38] Laura Meikle and Jacques Fleuriot. Mechanical Theorem Proving in Computational Geometry. In Hoon Hong and Dongming Wang, editors, *Automated Deduction in Geometry*, volume 3763 of *Lecture Notes in Computer Science*, pages 1–18, Berlin/Heidelberg, 2006. Springer. 5th International Workshop, ADG 2004, Gainesville, FL, USA, September 16-18, 2004. – Cité en pages 4, 23 et 29.

[39] Guillaume Melquiond. *Gappa : Génération Automatique de Preuves de Propriétés Arithmétiques*. INRIA, France, 2010. Version 0.12.3. `http ://gappa.gforge.inria.fr`. – Cité en page 95.

[40] Dominique Michelucci and Michel Gangnet. Saisie de plans à partir de tracés à main-levée. *Actes de MICAD 84*, 1984. – Cité en page 3.

[41] Jean-Marc Mota. *Méthodes formelles pour la modélisation géométrique à base topologique : définitions et algorithmes avec la méthode B*. PhD thesis, Université d'Evry Val d'Essonne, 2005. – Cité en page 15.

[42] Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *TLCA'93 : Proceedings of the 1st International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345, Londres, UK, 1993. Springer-Verlag. – Cité en page 2.

[43] David Pichardie and Yves Bertot. Formalizing Convex Hull Algorithms. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 346–361, Berlin/Heidelberg, 2001. Springer. 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001. – Cité en pages 4, 23, 27, 29 et 95.

[44] Franco Preparata and Michael Shamos. *Computational Geometry : An Introduction (5th printing)*. Monographs in Computer Science. Springer-Verlag, New York, 1993. 398 pages. – Cité en pages 1, 3 et 26.

[45] Aristides Requicha. Representations for rigid solids : Theory, methods, and systems. *ACM Computing Surveys*, 12(4) :437–464, 1980. – Cité en page 3.

[46] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, 1983. 552 pages. – Cité en page 3.

[47] IGG Team. CGoGN. Available from `https ://iggservis.u-strasbg.fr/CGoGN/`. – Cité en page 84.

[48] The Coq Development Team. *The Coq Proof Assistant - Reference Manual*. INRIA, France, 2009. Available from `http ://coq.inria.fr/refman/`. – Cité en pages 2, 14 et 25.

[49] William Thomas Tutte. Combinatorial Oriented Maps. *Canadian J. Math.*, 31(5) :986–1004, 1979. – Cité en pages 3, 12 et 13.

[50] Kevin Weiler. Edge-based Data Structures for Solid Modeling in Curved-surface Environments. *IEEE Computer Graphics and Applications*, 5(1) :21–40, 1985. – Cité en page 3.