Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 323 (2004) 351–397

Theoretical Computer Science

www.elsevier.com/locate/tcs

Formalizing generalized maps in Coq

Christophe Dehlinger*, Jean-François Dufourd

Laboratoire des Sciences de l'Image, de l'Informatique, et de la Télédétection (UMR CNRS 7005), Université Louis-Pasteur de Strasbourg, Pôle API, boulevard S. Brant, 67400 Illkirch, France

Received 12 November 2002; accepted 10 May 2004
Communicated by J. Gallier

Abstract

This paper is the first half of a two-part series devoted to an exemplary formal proof of a fundamental result in the field of geometry—the theorem of classification of surfaces—which has major implications in computer graphics. We study here the specification of generalized maps, a topological combinatory model for surfaces subdivisions. We show how we developed in Coq two fundamentally distinct formalizations of generalized maps, each based on one of the standard definitions, in a single common framework, then used this specification to prove for the first time their complete equivalence.
© 2004 Published by Elsevier B.V.

Keywords: Coq; Calculus of Inductive Constructions; Geometry modelling; Generalized maps; Surfaces; Specification; Proof

1. Introduction

Computer graphics is a computer-related field that has close ties with one branch of mathematics, geometry. As in many other fields, application precedes theory: most of the graphics software is being developed without any of the rigorous mathematical foundations that would ensure their reliability. Our goal is to focus on one crucial type of geometric objects, combinatory surfaces, and study both their definition and operations to handle them. Within a computer-aided-proof environment, we design and implement formal specifications and then rigorously test them by proving the well-known and important theorems of geometry. This paper is the first of a two-part series. It introduces our specification of combinatory surfaces, that was built with the goal to

* Corresponding author.
E-mail addresses: dehlinger@lsiit.u-strasbg.fr (C. Dehlinger), dufourd@lsiit.u-strasbg.fr (J.-F. Dufourd).

0304-3975/$ - see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.tcs.2004.05.003

use it to prove a difficult theorem of topological classification of surfaces with respect to numeric combinatory characteristics, which is the topic of the second paper.

We have chosen to model combinatory surfaces with *generalized maps* (or *g-maps*), introduced by Lienhardt [13]. This specification was developed within the logical framework of the *calculus of inductive constructions* (CIC), a type theory, i.e. a higher-order logic in which propositions and proofs are of the same kind as objects. We build our theorem proofs with the help of CIC-based proof assistant Coq. The considered classification theorem is actually the union of two independent subtheorems of comparable importance: a normalization theorem and a "*trading*" theorem. We only deal with the latter, the former not having been formally proved yet. Moreover, we only work with open surfaces for now, as closed surfaces would require a different set of operations. The general approach would probably work for closed surfaces as well though.

This work takes advantage of a large body of algebraic specifications that were originally developed for software design purpose (in particular for modeller Topofil [2]): Indeed, a lot of theoretical results regarding surfaces specification is already available, and we try and reuse this knowledge to build formal proofs. Type theory and CIC have already been studied a lot [4,15] and have proved themselves in the field of computer-aided proving, in particular for combinatory geometry, with for instance the proof of a discrete Jordan theorem [17].

Besides, we also take a small part in the huge task of specifying combinatory geometry by formalizing the model of generalized maps, notably by proving an equivalence theorem between two representations of g-maps. This theorem had never formally proved before; it is likely that the informal proofs are flawed as we have unearthed special cases that they overlook.

This work on surfaces is the first step in the development of a rigorous unified framework in which geometric objects and operations are defined, their properties proved, and the safety of the software that uses them is ensured (by using program extraction techniques for instance). Like all other fundamental specification and proof works, we hope that it will also provide greater insight on the considered objects and "definitive" proofs of the validity of the studied mathematical properties [12,2]. In our case, a deeper understanding of surfaces would be very welcome in the field of discrete geometry [3], where this notion still remains unclear and debated. Thus, solving the problems of surface extraction from voxel images and discrete surface subdivision, which are both crucial in 3D imagery, requires a well-adapted definition of surfaces.

From the computer-aided proving field point of view, this work can be seen as a large application developed in CIC design using a type hierarchy that exploits the casting facilities that proof assistant Coq offers in order to compensate for the lack of support of real subtyping in CIC. Our method is to define several layers of high-level constructors, which are either low-level constructors (i.e. CIC constructors) or functions; each level features an induction scheme that has been proved either automatically for the former or manually for the latter. The methodology that we have developed and used is independent from our particular problem. It may serve as a basis for many different specifications, in particular for graph-like objects specifications.

As far as we know, all existing proofs of the trading theorem are informal to some degree. They all differ on some points, the first of which being the mathematical nature of the considered combinatory surfaces. For instance, there is a proof that applies to surfaces that are defined as words on an alphabet, in which an oriented edge is represented by a signed letter [7]. We have chosen a modelling much like the ones used by Griffiths [9] and Fomenko [8]: surfaces are seen as patchworks of *panels* (i.e. surface elements that are homeomorphic to a disc). Griffith's proof proceeds by identifying two distinct subdivisions, Fomenko's by incrementally turning one into the other. We took inspiration in the latter, as it does not require a graphical interpretation of surfaces like the former does.

A little background on higher-order logics is welcome, but not needed, to fully understand this paper and the techniques we use. The reader should feel free to skip the "implementation" details of the most complex functions, as they are only mentioned for the sake of being complete. This paper is structured this way: after this introduction, we present some related work (Section 2), and then briefly describe the calculus of inductive constructions (Section 3). In Section 4, we will give a mathematical description of the generalized maps model. The next two sections present our formal specifications in Coq and some of the meaningful lemmas and theorems that we have proved. Section 5 deals with free and generalized maps, Section 6 of well-constructed maps. We conclude in Section 7.

2. Related work

The geometry modelling aspects of this work are mostly related to generalized maps. These maps are a variation of Jacques's [10] combinatory maps, which have been extended by Cori [5] into hypermaps and then by Lienhardt [13] into generalized maps, and later algebraically specified by Bertrand and Dufourd [2].

Generalized maps offer one constructive model for geometry. Another approach of constructive geometry has been explored by Von Plato [19] and formalized in Coq by Kahn [11]. A similar experiment has been undertaken by Dehlinger et al. [6]. Also using Coq, Pichardie and Bertot [16] have formalized Knuth's [12] plane orientation and location axioms, and then proved correct algorithms of computation of convex hulls.

Proof assistant Coq [1] is based on the calculus of inductive constructions [4], an intuitionistic-type theory [14]. In addition to being a powerful tool for proofs, it features the extraction of certified programs from formal proof terms [15].

3. Calculus of inductive constructions

The c*alculus of* i*nductive* c*onstructions* (*CIC*) is a type theory that is well adapted to the mechanization of mathematics [4]. It is based on two meta-theories: Girard's *polymorphic λ-calculus*, a powerful functional system that allows the representation of propositions and proofs in a single high-level formalism, and Martin–Löf's intuitionistic-type theory [14], a foundation of mathematics on constructive principles. The two theories are merged with the help of the *Curry–Howard isomorphism* that describes how function-representing λ-terms may also be used to represent proofs in natural deduction. From a more pragmatic point of view, the CIC can be seen as a mixture of PROLOG-style inductive predicates and ML-style recursive functions. The CIC provides us with a unified framework in which algebraic specifications can be expressed very naturally and preconditions and type relations can be taken into account.

Our proofs are built with Coq, a proof assistant built on the CIC. A high-level language, Gallina, allows the declaration of axioms and parameters, the definition of types and concrete or abstract objects. In proof mode, the intuitive reasoning steps are implemented by *tactics*, which are commands that modify the formal wording of the proposition that is being proved, also called the *goal*, possibly by splitting it into several subgoals.

The proof terms of Coq are very information-rich. Thus, thanks to the Curry–Howard isomorphism, a construction algorithm of an object may be extracted from the proof of its existence [15]. This is a programming paradigm by itself, that consists in extracting a program from a proof of its correctness. This technique is obviously very powerful for certified software development.

4. Generalized maps

The term "generalized maps" actually encompasses a series of combinatory models used to represent the topology of different classes of geometric objects. A *generalized map* is first characterized by its *dimension*, an integer greater or equal to -1. Depending on the dimension, the type of represented objects varies:

Dimension	Object classes
-1	Isolated vertices
0	Isolated edges
1	Simple curves
2	Surfaces
3	Volumes
4	4D volumes
$\vdots$	$\vdots$

The mathematical definition of generalized maps is:

Definition 1 (*N-G-MAP*). A g-map of dimension n (or *n-g-map*) is a $(n+2)$-uplet $(D, \alpha_0, \ldots, \alpha_n)$, where D is a finite set of abstract elements called *darts*, and where the α_i are involutions on D, i.e. permutations on D such that, for any x, $\alpha_i^2(x) = x$.

Besides, the α_i must satisfy the following constraints:

1. $\forall i < n, \alpha_i$ has no fixpoint (i.e. for any x, $\alpha_i(x) = x$);
2. $\forall i, j \mid 2 + i \leqslant j \leqslant n$, $\alpha_i \circ \alpha_j$ is an involution

Dart y is a *k-successor* of dart x if $\alpha_k(x) = y$. Dart x is then also said to be *sewn at dimension* k to y, or *k-sewn* to y. A dart may be sewn to itself. As the α_i are involutions, they obviously are symmetrical; as a consequence, x is k-sewn to y iff y is k-sewn to x.

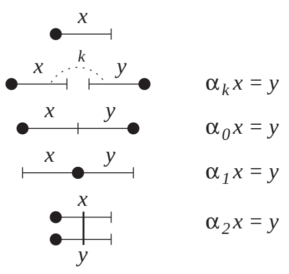


Fig. 1. Standard representation of g-maps.

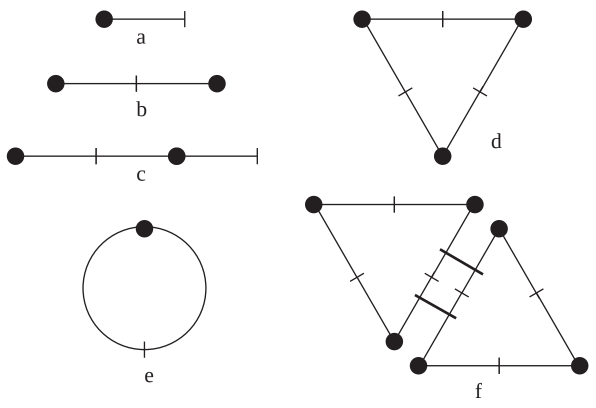


Fig. 2. Examples of generalized maps.

Intuitively, darts may be seen as half-edges, and the α_i as different ways to paste them, or *sew* them. Each α_i has its own meaning suggested by the previous table: α_0 sews darts to make up edges, α_1 simple curves, α_2 surfaces, etc. In general, for any given k, α_k is used to build cells of dimension k.

The standard graphical representation of darts and sewings is depicted in Fig. 1, in which x and y are darts. As the α_i are involutive, they are symmetrical, and hence they need not be oriented in the figure. A dart, the k-sewing of which is not explicitly shown, is implicitly k-sewn to itself.

The graphical representation of a generalized map does not explicitly feature its dimension. Though, we can try to deduce it from the type of sewings that appear in the figure: if a g-map drawing features any k-sewings, then its dimension is at least k. Moreover, if some darts are k-sewn to other darts while other darts are k-sewn to themselves, then the dimension of the map is exactly k, thanks to constraint 1. But in the case where all darts are sewn to themselves at dimension $(k+1)$ while none is at dimension k, then we cannot tell whether the represented map has dimension k or $(k+1)$. A few examples of g-maps are given in Fig. 2. Fig. 2a is a (-1)-g-map, but also a 0-g-map. We know for sure that Fig. 2b is a 1-g-map, because only some of its

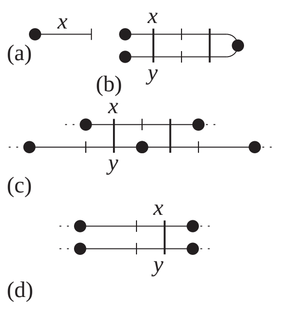


Fig. 3. (a) A 0-g-map, (b) a 2-g-map, (c) and (d) not g-maps.

darts are 1-sewn to themselves. Likewise, Fig. 2c is a 1-g-map. Figs. 2d and e may represent 1-g-maps as well as 2-g-maps, because all their darts are 1-sewn to other darts and 2-sewn to themselves. Fig. 2f represents a 2-g-map. These examples suggest that there is some degree of recursivity in the structure of g-maps with respect to the dimension. This point will be developed later. Constraints 1 and 2 of the definition of g-maps forbid some configurations, such as the ones illustrated in Fig. 3.

- For instance, constraint 1 forbids:
  - Fig. 3a as an n-g-map for $n \geqslant 1$ because x is 0-sewn to itself. This does not prevent it from being a 0-g-map however;
  - Fig. 3b as a 2-g-map because x and y are 1-sewn to themselves;
- Constraint 2 forbids shifts in sewings and forces the sew whole cells only:
  - Fig. 3c is not a generalized map: $\alpha_0 \circ \alpha_2 \circ \alpha_0 \circ \alpha_2(x) = y \neq x$ (there is a shift, i.e. the darts of a single edge are sewn to darts of two distinct edges);
  - Fig. 3d is not a generalized map: $\alpha_0 \circ \alpha_2 \circ \alpha_0 \circ \alpha_2(x) = y \neq x$ (2-sewing along a half-edge only).

Before defining the usual notions of topology in the formalism of n-g-maps, we need to introduce the mathematical notion of *orbit*.

Definition 2 (*ORBIT*). Let S be a set and $f_1, f_2, \ldots, f_k$ functions on D. For any $x \in D$, we call *orbit of* $f_1, f_2, \ldots, f_k$ at x the smallest subset of S containing x and stable by all functions f_i. In other words, it is the set of elements of S that can be obtained from x by applying any composition of the f_i. It is noted $\langle f_1, f_2, \ldots, f_k \rangle(x)$.

From now on, $G = (D, \alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_n)$ will be an n-g-map we use to give our definitions. With the notion of orbit, connected components and cells are easily defined as 2-g-maps derived from an initial 2-g-map:

Definition 3 (*CONNECTED COMPONENT*). The *connected component* of *G incident to dart* $x \in D$ is defined as the 2-g-map $G' = (D', \alpha'_0, \alpha'_1, \alpha'_2, \ldots, \alpha'_n)$ satisfying

- $D' = \langle \alpha_0, \alpha_1, \ldots, \alpha_n \rangle(x)$;
- $\forall i \mid 0 \leqslant i \leqslant n$, α'_i is the restriction of α_i to D'.

Definition 4 (*CELL*, *VERTEX*, *EDGE*, *FACE*, *OPEN*, *CLOSED*). For any $x \in D$, we call *vertex of G incident to x*, or 0-*cell of G incident to x*, the orbit $\langle \alpha_1, \alpha_2(x), \ldots, \alpha_n(x) \rangle$. Similarly, we call *edge of G incident to x*, or 1-*cell of G incident to x*, the orbit $\langle \alpha_0, \alpha_2(x), \ldots, \alpha_n(x) \rangle$; and we call *face of G incident to x*, or 2-*cell of G incident to x*, the orbit $\langle \alpha_0, \alpha_1(x), \ldots, \alpha_n(x) \rangle$. We call *map of k-cells* of G the pseudo-2-g-map obtained from G by ripping all k-sewings. It is noted G_k. In dimension 2, an edge may be either 2 darts (*open* edge) or 4 darts (*closed* edge).

Cells defined this way are *topological* cells. As such, they have no associated position or shape. The only relevant notions that define them are the way they are connected. Assigning a space position and actual metric shape to cells is done by applying an *embedding* function to the cells, i.e. a function that associates a subset of the representation space to each cell. Embedding functions must have a number of properties to be valid, notably that embeddings of two cells are (geometrically) incident iff the cells are (topologically) incident. A *standard embedding* is an embedding into Euclidean space in which vertices are embedded into points, edges into straight lines and faces into polygons, such that the vertices incident to an edge are embedded into the ends of the corresponding line, and the edges incident to a face are embedded into the boundary of the corresponding polygon. Another common possibility is to embed edges into Bézier curves.

For any $n \geqslant 1$, the G_k are *pseudo-n*-g-maps, as opposed to real n-g-maps, because while they are also built using some α_i and a set of darts, their α_k has fixpoints, which contradicts the definition of n-g-maps for $k = 0$ and $n \geqslant 1$. Obviously, two darts belong to the same k-cell of G iff these two darts belong to the same connected component of G_k. Fig. 4 shows an example of 2-g-map as well as its maps of cells; in this example, G has two connected components. Similarly, all darts from the same topological edge are embedded into the same geometric edge and all darts from the same topological face are embedded into the same geometric face. Incident topological objects are embedded into incident geometrical objects. The 1-g-map $\delta(G)$ is the 2-*g-map of boundaries* of G: its darts are the darts of G that are incident to a boundary, each of them being 1-sewn in $\delta(G)$ to its boundary-neighbour in G, the notions of boundary and boundary neighbourhood being defined as:

Definition 5 (*BOUNDARY INCIDENCE*, *OUTER*, *INNER*). A dart $x \in D$ is said to be incident to a *boundary* of G if $\alpha_n(x) = x$. A dart incident to a boundary is called *outer*, it is called *inner* else.

Definition 6 (*BOUNDARY NEIGHBOURHOOD*). For any dart x, dart $y \in D$ is called *boundary-neighbour* of x if

- y is incident to a boundary (i.e. $\alpha_n(y) = y$);
- k is the smallest natural number such that $y = (\alpha_n \circ \alpha_{n-1})^k(x)$ and $y \neq x$.

During our developments, we have proved that a dart has at most one boundary-neighbour. Though we do not use this property, it suggests that boundary neighbourhood was well formalized.

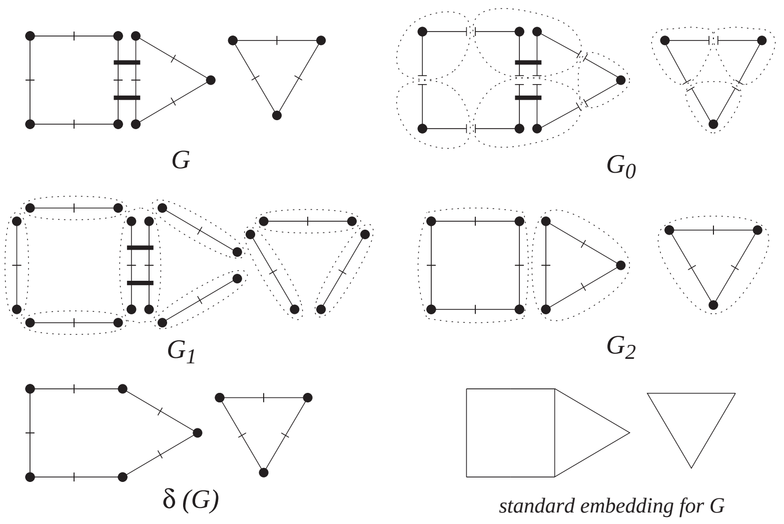


Fig. 4. An example of a 2-g-map and its maps of cells, 2-g-map of boundaries and standard embedding.

5. Formal specification of free and generalized maps

Our specification of g-maps is written in *Gallina*, the language used by proof assistant Coq for the declaration of axioms and parameters as well as the definition of concrete and abstract objects. It allows the user to handle the terms of CIC in a rather natural way. The syntax of Gallina will be introduced step by step along the specification. Gallina terms are printed in a `typewriter` font. Keywords are underlined, types are *italicized*, the names of lemmas, theorems and axiomes are UPPER-CASED, variable names are lower-cased.

Our specification is structured as a three-level map-type hierarchy, each representing a subtype of the previous. There is no real subtyping in CIC, so we have to use some techniques to simulate it. They have already been used by Dufourd and Puitg in [17] for their two-level specification of combinatory maps. To each level corresponds a concrete CIC type, respectively, *fmap*, *ngmap* and *smap*:

– *fmap*: The first level is the level of *free maps*. A free map is simply a set of darts, called *support* of the map, with a finite series of finite binary relations between darts, which may or may not belong to the support (Fig. 5). In terms of graphs, they are integer-arc-valued multigraphs, some vertices of which make up the support. A free map has no dimension. Free maps may be used to represent not only g-maps, but they also encompass very different other objects, such as combinatory maps, as in [17], or the pseudo-g-maps that we used in Definition 5.

– *ngmap*: The second level is the level of *generalized maps*. Their specification is a straightforward translation of the mathematical definition given in the previous section: the *ngmap* are *fmaps* the binary relations of which satisfy the properties of the α_i. In the CIC, a natural way to express this is to exploit the fact that proofs

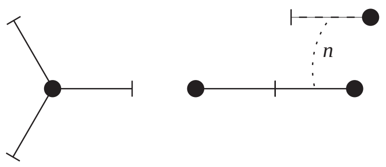


Fig. 5. Examples of free maps that do not represent g-maps (the dashed dart is a dart that does not belong to the map).

are the same kind of objects as concrete objects such as free maps. Thus, an *ngmap* is defined as a triplet made up of an *fmap* called *support of the generalized map*, a dimension and a term of proof that the support and the dimension satisfy a *well-formedness predicate*, noted `wf`. Predicate `wf` basically is a formal expression of the properties of the α_i in a g-map.

– *smap*: The third level is the level of the *sewn maps*, or *s-maps*. It is in turn a subset of the generalized maps, built only with the help of a high-level cell-sewing operation noted `sm`.

In this hierarchy, the higher the level, the more specialized the objects and the more complex the constructors. In general, using such a hierarchy provides with a clear and modular specification, while at the same time avoiding redundancies. Besides, when defining operations or proving properties on a type, especially when this type is complex, it is often necessary (or at least much easier) to work in a more general framework than the type itself. For instance, in trigonometry, although sine and cosine are real functions, it is often easier to see them as combinations of complex exponentials. In our case, the *fmap* are a such more general framework for the *smap*.

Many of the operations that we are interested in are operations on the g-maps, and hence that will be applied only to g-maps. In order to specify them formally, it is however much more simple to formulate them at the level of the *fmap*, having no concern for their behaviour when they are applied to *fmaps* that are not g-map supports, i.e. that do not satisfy predicate `wf`. We then show that whenever they are applied to g-map supports, they also yield g-map supports. Formally, this amounts to showing that these operations preserve predicate `wf`.

5.1. Binary relations

Before specifying maps themselves, we define a few secondary notions, in particular the notion of relation. It is not necessary to formalize them separately, but we choose to do it nonetheless on safety and modularity grounds, as although the considered notions, such as functionality and injectivity, are very basic and well known, it is easy to make slight mistakes and their formulation, thereby compromising the rest of the specification. Moreover, we have at our disposal Dufourd and Puitg's [17] specification of relations, which make this approach virtually costless. In this specification, relations are defined on any type, and only link objects of the same type. Thus, we cannot define this way a relation between an integer and a dart. A relation is simply represented by a 2-place predicate on a given type. Conversely, such a predicate may always be interpreted as a relation. As the two notions are equivalent simply define relations on a type as an alias for two-place predicates on objects of this type. This is performed with the *Gallina definition* mechanism.

Definition 7 (*RELATION*). Let E be a set. A *binary relation* on E is a two-place predicate on objects of this set. When two objects are put into relation, the one in the first position is called the *antecedent* and the one in the second position is called the *image*:

```
Definition relation : Set → Type
    := λE:Set. E → E → Prop
```

This command allows to declare symbol `relation` as an abbreviation for term λE:*Set*.*E* → *E* → *Prop*.[1]

In Gallina, the notation x:*T* is a type judgement meaning "term x is of type *T*". Predefined type *Prop* is the type of proposition terms. For instance, terms (0=5) or ((∀x : *nat*) x=x) are of type *Prop*; the truth or falseness of a proposition is irrelevant to the meaning of *Prop*. Predefined type *Set* is the type of concrete types, which are the types of the objects that we actually want to build.

Thus, the type of *relation* is *Set* → *Type*, thus an object of type *relation* is a function that, when applied to a concrete type *E* (of type *Set*), yields a type, precisely type *E*→ *E* → *Prop*, i.e. the type of two-place predicates on *E*.

For instance, a relation on integers is of type (*relation nat*), which is a shortcut for *nat* → *nat* → *Prop*.

The definition mechanism may also be used to express predicates on relations:

Definition 8 (*INJECTIVITY*). Let E be a set and R a relation on E. R is *injective* if equality of images by R implies equality of arguments:

```
Definition injective : (∀E : Set) (relation E) → Prop
        := λE:Set. λR:(relation E).
            (∀x,x',y : E) (R x y) → (R x' y) → x=x'.
```

Definition 9 (*FUNCTIONALITY*). Let E be a set and R a relation on E. R is *functional* if equality of arguments of R implies equality of images

```
Definition function: (∀E : Set) (relation E) → Prop
        := λE:Set. λR:(relation E).
            (∀x,y,y' : E) (R x y) → (R x y') → y=y'.
```

[1] The Gallina syntax has been slightly modified in order to make it more accessible to beginners. Symbol ¬ replaces the usual negation, symbol ∀ is added where relevant, symbol λ representing lambda-abstraction will be used instead of the notation [...], parentheses around integers are omitted, notation (∃ x|...) is used for {x | ...} and {x & ...}, the natural successor S is replaced by +1, disjunctions (+) on *Set* are replaced by their equivalent ∨ on *Prop*.

Definition 10 (*INVOLUTIVITY*). Let E be a set and R a relation on E. R is *involutive* if for any dart, an image of an image of this dart is the dart itself

```
Definition involutive : (∀E : Set) (relation E) → Prop
      := λE:Set. λR:(relation E).
         (∀x,x',y : E) (R x y) → (R y x') → x=x'.
```

Definition 11 (*IRREFLEXIVITY*). Let E be a set and R a relation on E. R is *irreflexive* if no object is in relation with itself by R:

```
Definition irreflexive : (∀E : Set) (relation E) → Prop
      := λE:Set. λR:(relation E).
         (∀x : E) ¬(R x x).
```

Definition 12 (*IDENTITY*). Let E be a set and R a relation on E. R is *identical* if a dart may only be in relation with itself by R:

```
Definition identical : (∀E : Set) (relation E) → Prop
      := λE:Set. λR:(relation E).
         (∀x,y : E) (R x y) → x=y.
```

Note that term λx:*X*. t where t has type *T* is a function term of type *(∀x : X)T*. A priori, proposition "the natural order on natural numbers (noted le) is injective" is formalized by term (injective nat le). Notice that in this term, the presence of nat is redundant: indeed, as le is of type (relation nat), nat is the only possible type T so that (injective T le) is well typed. Coq is able to infer redundant-type arguments, provided it is asked so with command Set Implicit Arguments. These types may then be omitted to simplify the notations. With this facility, the previous will simply be written (injective le). Whenever possible, we underlined in definitions the type arguments that Coq was able to infer, and hence are later omitted. Besides, we also use the pretty-printing facilities of Coq to display some expressions in a nicer way. For example, (le 3 4) is written as 3⩽4.

We also define relation composition.

Definition 13 (*COMPOSITION*). The *composition* of R et R' is a relation that links two objects x and y provided there exists a z such that (R' x z) and (R z y), x and y belonging to E:

```
Definition composition:
    (∀E : Set) (relation E) → (relation E) → (relation E)
          := λE:Set. λR,R':(relation E).
             λx,y:E. (∃z | (R' x z) ∧ (R z y))
```

Now, we wish to define surjectivity. However, we do not want to express that a relation is surjective on its whole argument type E, but only on one of its subparts. The relation will then be said to be surjective *on* that subpart. To do this, we consider an abstract set F that will be seen as the set of parts of E. We then introduce a membership predicate is_in for F, of type *E* → *F* → *Prop*. Term (is_in x m) means that x, of type E is an element of m, of type F. A relation of E on F is a function yielding a binary relation in E, hence of type *F* → (*relation E*); these objects are called *relations-on*.

Definition 14 (*SURJECTIVITY*). A relation R in E on F is *surjective* on m if for any y in E belonging to m, there exists a x in E such that (R x y):

```
Definition surjective:
    (∀E,F : Set) (E → F → Prop)
        → (F → (relation E)) → F → Prop
    := λE,F:Set. λis_in:(E → F → Prop).
            λR:(F → (relation E)). λm:F.
        (∀y : E) (is_in y m) → (∃x | (R m x y))
```

With the same conventions, we express the fact that a relation is total or localized:

Definition 15 (*TOTALITY OF A RELATION*). A relation R in E on F is *total* on m if for any x in E belonging to m, there exists a y in E such that(R x y):

```
Definition total:
    (∀E,F : Set) (E → F → Prop)
        → (F → (relation E)) → F → Prop
    :=λE,F:Set. λis_in:(E → F → Prop).
            λR:(F → (relation E)). λm:F.
        (∀x : E) (is_in y m) → (∃y | (R m x y))
```

Definition 16 (*LOCALIZATION*). A relation R in E on F is *localized* on m if for any x in E, (R x y) entails that x and y belong to m:

```
Definition localized:
    (∀E,F : Set) (E → F → Prop)
        → (F → (relation E)) → F → Prop
    := λE,F:Set. λis_in:(E → F → Prop).
            λR:(F → (relation E)). λm:F.
        (∀x,y : E) (R m x y) → (is_in x m) ∧ (is_in y m)
```

All the previous notions are combined to define the permutations:

Definition 17 (*PERMUTATION*). A relation R in E on F is a *permutation* on m if it is total, surjective, and (R m) is functional and injective

```
Definition permutation:
    (∀E,F : Set) (E → F → Prop)
        → (F → (relation E)) → F → Prop
    :=λE,F:Set. λis_in:(E → F → Prop).
            λR:(F → (relation E)). λm:F.
        (total is_in R m) ∧ (surjective is_in R m)
        ∧ (function (R m)) ∧ (injective (R m))
```

Notice that thanks to implicit arguments, variables E and F do not appear whenever they may be inferred.

5.2. *Definition of darts*

The *darts*, the basic bricks of our maps, have previously simply been defined as abstract objects, without any more information. This is exactly how we formalize them, in order to remain as general as possible. This allows us to avoid to rely on the geometric embedding of darts, which would be the case had we chosen to represent them as half-edges. In the same way, the type of darts does not have to be instantiated as a particular type, like integers. Although in most actual software darts are indeed represented by integers or pointers, this fact is irrelevant as far as theorem proving goes. Hence, we will simply consider the type of darts, noted *dart* as a parameter of our specification.

Parameter 1 (*DART*). there exists a type of darts called dart

```
Parameter dart : Set.
```

This declaration allows us to assume the existence of an object of type *Set*, i.e. a concrete type, and name it *dart*. A priori, *dart* may be any concrete type, finite or infinite. We know nothing of the elements that inhabit it. Because of our needs during the proofs, we will have to make a number of additional reasonable hypotheses. First, a recurring reasoning scheme will be to compare two darts and proceed with the proof in a different way depending whether the darts are equal or not. In classic logic, this is completely trivial: one can perform case reasoning on the truth of any proposition. On the other hand, in intuitionistic logic, which is the logic underlying the CIC, case reasoning may be performed only if it is actually known how to decide (i.e. compute) which case is the good one. But there is no general algorithm that may be applied to any type and that decides equality of two elements of this type; there are even some types for which equality is undecidable, for example real numbers. Hence, we must assume that such a decision algorithm is available for *dart*. This is another parameter of the specification:

Parameter 2 (*DECIDABILITY OF DART EQUALITY*).

```
Parameter EQ_DART_DEC : (∀x,y : dart) x=y∨¬x=y
```

This declaration binds the name EQ_DART_DEC to an object of type (∀x, y : *dart*) x=y∨¬x=y. It means that EQ_DART_DEC is an algorithm that, given two *darts* x and y, allows us to prove either $x = y$, or $x \neq y$, knowing in which case we are. In other words, EQ_DART_DEC is a computable equality criterion for objects of type *dart*.

Besides, during our proofs we will have to build a number of darts that are absent from the handled maps in order to build intermediary maps. Thus, we must make sure that we always have new darts available, which amounts to assuming that type *dart* is infinite. To do this, we assume that we have at our disposal an injection from naturals into *dart*. This injection can be seen as an infinite dart generator, noted idg:

Parameter 3 (*DART GENERATOR*). Generator idg is a function which takes a natural for argument and yields a dart

```
Parameter idg : nat → dart.
```

This command binds the symbol `idg` to an object of type *`nat`* → *`dart`*, i.e. a function that takes a *`nat`* for argument and yields a `dart`. We still have to ensure that this generator is injective.

Definition 18 (*INJECTIVITY OF THE DART GENERATOR*). Equality of images by `idg` implies equality of arguments

```
Inductive IDG_INJ : (∀n,n' : nat) (idg n)=(idg n') → n=n'.
```

The term `IDG_INJ` is declared as an axiom. This means that `IDG_INJ` is as sufficient and useable proof of the injectivity of `idg`.

5.3. Sewings

A sewing is a triplet made up of a natural representing its dimension and the two darts that are sewn. The type of sewings, noted *sw*, is defined as:

Definition 19 (*SEWINGS*).

```
Inductive sw : Set := c : nat → dart → dart → sw.
```

First notice the form of the type of `c`. If we put back the parentheses that are omitted because of the right associativity of →, the type of `c` is `(`*`nat`* → `(`*`dart`* → `(`*`dart`* → *sw*`)))`. Thus, term `c` is a function from *`nat`* to *`dart`* → `(`*`dart`* → *sw*).

As a consequence, terms `(c 0)` et `(c 1)` are of type *`dart`* → `(`*`dart`* → *sw*). In a similar way, if `x` and `y` are *`darts`*, term `((c 0) x)` is of type *`dart`* → *sw*, and term `(((c 0) x) y)` of type *sw*.

Function `(c 0)` may be interpreted as the partial function that for any `x` and `y` of type *`dart`* yields the image of `(0,x,y)` by `c`. Application being left-associative, some parentheses may be omitted, so we can write `(c 0 x y)` instead of `(((c 0) x) y)`.

The definition of `c` is said to be *inductive*. It allows to uniquely specify a concrete type or a predicate by giving an exhaustive list of the constructors of the elements of this type. The constructors are two-by-two distinct and are all injective. Here, *sw* is defined as the type of terms built from its only constructor `c`. For instance, if `x` and `y` are *`dart`* s, then `(c 0 x y)`, `(c 5 x x)` and `(c 27 y x)` all are of type *sw*. Conversely, this declaration specifies that `c` is the only constructor of *sw*. This means that if an object `s` is of type *sw*, then one can infer that there exist `n`, `x` and `y` such that `s=(c n x y)`. This property, called an *inversion* property, is wrapped in a lemma with the same name.

Lemma 1 (INVERSION OF SEWINGS). *Any sewing is part of the image of* `c`

```
Lemma SW_INV: (∀s:sw)
    (∃n:nat |(∃x,y:dart | s=(c n x y)))
```

Proving, as we did, a separate lemma for such a simple proposition may seem useless, as it is obtained by using a single tactic; hence, we could use the same tactic whenever we use this lemma in further proofs. This is justified by a general will to wrap the inductivity properties of inductive objects in lemmas; indeed, these properties, when exploited directly with inversion tactics, are analogous to "low-level" functions, and for the same reasons as in traditional programming, we prefer working at a higher level.

Among others, this allows to clearly separate the Gallina implementation of objects like sewings from the proofs in which they appear, and to put these properties into a form that is more conveniently used than the one obtained by a direct use of tactics. Besides, these lemmas can be used as any other term proofs; it is therefore possible to combine them to form term proofs of other propositions, which sometimes greatly shortens proofs. As terms, they are also sometimes even necessary when defining functions.

The third inductivity property, the injectivity of constructors, here amounts to be able to infer `n=n'`, `x=x'` and `y=y'` from `(c n x y)=(c n' x' y')`. As with `SW_INV`, this property is wrapped in the following lemma.

Lemma 2 (INVERSION OF SEWINGS EQUALITY). *Equality of sewings implies equality of its parameters*

```
Lemma EQ_SW_INV :
    (∀n,n':nat; ∀x,x',y,y':dart)
        (c n x y)=(c n' x' y') → n=n'∧x=x'∧y=y'.
```

Our first useful result on sewings is the decidability of their equality, the Coq proof of which uses the decidability of equality of naturals, a property that is proved in the standard Coq library.

Lemma 3 (DECIDABILITY OF EQUALITY OF NATURALS). *It is possible to decide whether two naturals are equal or distinct*

```
Lemma EQ_NAT_DEC : (∀n,n' : nat) n=n'∨¬n=n'
```

Lemma 4 (DECIDABILITY OF SEWINGS EQUALITY). *It is possible to decide whether two sewings are equal or not*

```
Lemma EQ_SW_DEC : (∀s,s':sw) s=s'∨¬s=s'
```

5.4. Free maps

A free map is simply made up of two independent finite sets, one of darts and the other of sewings. It is specified by a three-constructor inductive type: the first one, `v`(*oid*), is used to build an empty map, while the other two, `i`(*nsert*) and `l`(*ink*), respectively, allow the addition of a dart or a sewing to an already built map.

Definition 20 (*FREE MAP*).

```
Inductive fmap : Set :=
      v : fmap
    | i : dart  → fmap  → fmap
    | l : sw  → fmap  → fmap
```

This definition is obviously very little constrained, which allows a lot of constructions. A dart may be sewn to itself or to any other dart at any dimension, it may be sewn to several darts at the same dimension; a dart even does not have to have been inserted previously to being sewn. A free map may be seen as two interwoven lists, one of darts and the other of sewings, sharing the same empty list constructor `v` and each with its own head insertion constructor, respectively, `i` and `l`.

When *fmap* is declared, Coq automatically generates a structural induction theorem.

Lemma 5 (INDUCTION ON FREE MAPS).

```
Lemma fmap_ind : (∀P : (fmap → Prop))
    (P v)
    → ((∀d:dart; ∀f:fmap) (P f) → (P (i d f)))
    → ((∀s:sw; ∀f:fmap) (P f) → (P (l s f)))
    → (∀f : fmap)(P f)
```

This theorem states that if a predicate on *fmap* is satisfied by `v` and stable by `i` and `l`, then it is satisfied for any *fmap*. This theorem will be the backbone of most of our proofs on *fmap*.

In order to define a predicate on a free map, we always use the same technique: we state, if it is the case, that `P` is satisfied for `v`, and then we describe the behaviour of `P` whenever we add a dart or a sewing to the map. Thus, we now define three fundamental selector predicates on this type. The first expresses the existence of a dart in a map. It is noted `exd` (**ex**istence of a **d**art).

Definition 21 (*EXISTENCE OF DARTS*). Existence of darts in an *fmap* is the smallest two-place predicate `exd` satisfied for `x` on `(i x m)` for any free map `m`, that is also stable by dart insertion `i` and sewing insertion `l`

```
Inductive exd : dart → fmap → Prop :=
      EXD_I_X : (∀x:dart; ∀m:fmap)
                    (exd x (i x m))
    | EXD_I   : (∀x:dart; ∀m:fmap) (∀d : dart)
                    (exd x m) → (exd x (i d m))
    | EXD_L   : (∀x:dart; ∀m:fmap)(∀s : sw)
                    (exd x m) → (exd x (l s m)).
```

The predicate constructors are capitalized as they may be interpreted and used as the axioms that define the predicate. Just like with concrete inductive type, an inversion lemma can be defined for a predicate:

Lemma 6 (INVERSION OF EXD). *A dart belongs to a free map if it has just been inserted, or if a dart or a sewing has just been inserted into the free map and the dart already belonged to the older map*

```
Lemma EXD_INV : (∀x:dart; ∀m:fmap)
    (exd x m)
    → (∃m':fmap|
      m=(i x m')
      ∨((∃y:dart | m=(i y m'))∨(∃s:sw | m=(l s m'))
         ∧ (exd x m'))
```

For proof-development reasons, this lemma is then split into three narrower sublemmas, each corresponding to one of the possible constructors for map `m`:

Lemma 7 (EXISTENCE OF DARTS, INVERSION/V). *No dart is in fmap* `v`

```
Lemma EXD_V_INV : (∀x : dart) ¬(exd x v).
```

Lemma 8 (EXISTENCE OF DARTS, INVERSION/I). *If* `x` *is in* `(i y m)`, *then* `x=y` *or* `x` *is in* `m`

```
Lemma EXD_I_INV : (∀x,y:dart; ∀m:fmap)
    (exd x (i y m)) →y=x∨(exd x m)
```

Lemma 9 (EXISTENCE OF DARTS, INVERSION/L). *If* `x` *is in* `(l s m)`, *then* `x` *is in* `m`

```
Lemma EXD_L_INV : (∀x:dart; ∀s:sw; ∀m:fmap)
    (exd x (l s m)) →(exd x m)
```

As with the sewings, we need to prove the decidability of existence of darts.

Lemma 10 (DECIDABILITY OF EXISTENCE OF DARTS). *For any dart* `x` *and any free map* `m`, *either* `x` *exists in* `m`, *or* `x` *does not exist in* `m`

```
Lemma EXD_DEC : (∀x:dart; ∀m:fmap)
    (exd x m)∨¬(exd x m)
```

We now define three selectors that observe the sewings of a map. The first one is analogous to `exd`, it allows to test the existence of a sewing at a given dimension between two given darts.

Definition 22 (*SUCCESSORS*). Let `k` be a dimension, `x` and `y` two darts and `m` a free map. Dart `y` is a `k`-successor of `x` in map `(l (c k x y) m)`; this property is stable by dart and sewing insertion

```
Inductive succ : nat →dart →fmap →dart →Prop
    := SUCC_L_X : (∀k:nat; ∀m:fmap; ∀x,y:dart)
          (succ k x (l (c k x y) m) y)
    | SUCC_I : (∀k:nat; ∀m:fmap; ∀x,y:dart) (∀d : dart)
          (succ k x m y)
          →(succ k x (i d m) y)
    | SUCC_L : (∀k:nat; ∀m:fmap; ∀x,y:dart) (∀s : sw)
          (succ k x m y)
          →(succ k x (l s m) y)
```

We will also often need to express the fact that a dart has no successor at a given dimension. We may use `succ` to do so, but the specification we are reusing features a separate selector for this purpose:

Definition 23 (*ABSENCE OF SUCCESSOR*). Let `k` be a dimension and `x` a dart. In the empty map, `x` has no successor. The absence of successor of `x` at dimension `k` is stable by dart insertion and sewing at any dimension other than `k` or linking any dart other that `x`

```
Inductive nosucc : nat →dart →fmap →Prop :=
      NOSUCC_V : (∀k:nat; ∀x:dart) (nosucc k x v)
    | NOSUCC_I : (∀k:nat; ∀m:fmap; ∀x:dart) (∀d : dart)
          (nosucc k x m) →(nosucc k x (i d m))
    | NOSUCC_L :
```

```
(∀k,k':nat; ∀m:fmap; ∀x,x':dart) (∀y' : dart)
(nosucc k x m)
→k'=k ∨ ¬x'=x
→(nosucc k x (l (c k' x' y') m))
```

Notice that it would have been equivalent to replace the first implication → in the definition of NOSUCC_L by a conjunction ∧; however, we favor expressions built with implications as they are more easily handled in proof mode. As usual, this definition comes along a number of inversion lemmas named NOSUCC_I_INV and NOSUCC_L_INV that we do not explicitly state here. As a general writing rule, inverson lemma will not be mentioned anymore. There is no inversion lemma NOSUCC_V that would be associated to the empty map as nothing can be deduced on x or k from (nosucc k x v).

Obviously, succ and nosucc are semantically very close. Indeed, nosucc could have been defined from succ, with the Gallina definition mechanism.

Definition 24 (*ABSENCE OF SUCCESSOR*, *ALTERNATIVE VERSION*). A dart x has no k-successor in m if for any dart y this dart is not the k-successor of x in m

```
Definition nosucc_alt : nat → dart → fmap → Prop
    := λn:nat. λx:dart. λm:fmap.
        (∀y:dart) ¬(succ n x m y)
```

We easily show that the two definitions of nosucc are actually equivalent:

Lemma 11 (EQUIVALENCE OF ABSENCES OF SUCCESSOR). *The predicates* nosucc *and* nosuccalt *are equivalent*

```
Lemma NOSUCC_NOSUCC_ALT : (∀k:nat; ∀x:dart; ∀m:fmap)
    (nosucc k x m) ↔ (nosucc_alt k x m)
```

There is no particular semantic or real practical reason to choose one version over the other, it essentially comes down to the user's own specification style. To finish with that matter, we now reformulate the previous equivalence lemma in a form that will be useful during proofs:

Lemma 12 (INCOMPATIBILITY BETWEEN SUCC AND NOSUCC).

```
Lemma SUCC_NOTNOSUCC : (∀k:nat; ∀x,y:dart; ∀m:fmap)
    (succ k x m y) →¬(nosucc k x m)
```

The last selector, called alpha, is a completion of selector succ.

Definition 25 (*SELECTOR ALPHA*). If dart y is the k-successor of x in m, then y is also its k-α-successor. On the other hand, if x has no k-successor in m but belongs to m, then x is its own k-α-successor

```
Inductive alpha : nat →fmap →dart →dart →Prop :=
      SUCC_ALPHA : (∀k:nat; ∀m:fmap; ∀x,y:dart)
                  (succ k x m y) →(alpha k m x y)
    | ALPHA_REF : (∀k:nat; ∀m:fmap; ∀x:dart)
                  (nosucc k x m) →(exd x m)
                  →(alpha k m x x)
```

This selector is simply used to ensure the existence of a successor for all the darts in the map. In the version of the specification presented in this paper, we consider `alpha` to be the more semantically interesting selector, i.e. the one the properties of which are more interesting, while `succ` is only an intermediary selector used in the definition of `alpha`. Current developments tend to reverse this precedence, which is something that was inherited from previous versions of the specifications. At the semantical level, it is `alpha` rather than `succ` that should be associated to the α_i. From now on, we shall say that `x` is *α-sewn at dimension k* (or `k`-α-sewn) to `y` in `m` iff `(alpha k x m y)`. Notice that `x` is `k`-α-sewn to `y` if `x` is `k`-sewn to `y`, but that the converse may be false. Indeed, a dart that is `k`-α-sewn to itself is either `k`-sewn to itself, or has no `k`-successor. It may not hold if `x=y`. In that case, `x` will be said to be *implicitly* `k`*-sewn to itself*.

5.5. Computing of successors, decidability of their existence

It is easy to prove that the selectors are decidable. We will however take a closer look at the decidability lemma for `nosucc`, as it is more interesting than the others:

Lemma 13 (DECIDABILITY OF NOSUCC).

```
Lemma NOSUCC_DEC : (∀k:nat; ∀x:dart; ∀m:fmap)
    (nosucc k x m)∨ ¬(nosucc k x m).
```

After thinking a little about this theorem, one can see that a more powerful version of this lemma may be built: if you take advantage of the bonds between `succ` and `nosucc`, you can define a similar lemma that will also exhibit a `k`-successor of `x` in the case where `¬(nosucc k x m)`.

Lemma 14 (EXISTENCE OR ABSENCE OF SUCCESSOR). *Let* `k` *be a dimension,* `m` *a free map and* `x` *a dart. Then either* `x` *has a* `k`*-successor* `y` *in* `m`, *or it does not have any*

```
Lemma SUCC_OR_NOSUCC : (∀k:nat; ∀x:dart; ∀m:fmap)
    (∃y:dart | (succ k x m y))
    ∨(nosucc k x m)
```

It may sometimes happen that a dart has several successors at the same dimension. The above lemma always yields one of them provided it exists, but you cannot tell which. Indeed, there is no information in the formulation of `SUCC_OR_NOSUCC` that allows to separate this dart from the others. While leading proofs, this turns out to be very problematic as unless you take much care, and work in a relatively non-intuitive and cumbersome way, you cannot even compare the darts you get from two separate applications of `SUCC_OR_NOSUCC` to the same arguments. Hence, we must find another way to get more precise information about this dart. The first method consists in proving a lemma that is analogous to `SUCC_OR_NOSUCC` and that imposes stricter constraints on y. Such constraints would be directly suggested by the needs of further proofs, and gathered in a predicate called `propsucc` of type *nat* → *fmap* → *dart* → *dart* → *Prop*. Property `(propsucc k m x y)` would be satisfied only if `y` were a `k`-successor of `x` in `m` that satisfies particular properties. Such a property could be that `y` must be the `k`-successor of `x` that was last inserted in `m`. Thus, after picking a propsucc that suits our needs, we would prove the following variant:

Lemma 15 (EXISTENCE OR ABSENCE OF SUCCESSOR, VARIANT). *Let* x *be a dart*, k *a dimension and* m *a free map. Either* x *has no* k*-successor in* m, *or* x *has one called* y *such that if* m *happens to be of the form* (l (c k x y'') (l (c k x y') m')) *then* y=y'

```
  Lemma SUCC_OR_NOSUCC_2 : (∀k:nat; ∀x:dart; ∀m:fmap)
         (∃y:dart | (succ k x m y) ∧ (propsucc k m x y))
```

This lemma allows to obtain more information on the yielded dart than with SUCC_OR_NOSUCC. The problem with this method is that all the properties that you might need on this dart must be anticipated; if one is forgotten, then the lemma must be reformulated and reproved with a stronger propsucc, and all the proofs that use it must be adapted to take its new form into account. The fundamental problem with using a lemma in order to build an object is that the building algorithm is hidden within the proof of the theorem, and thus is hard to extract and use.

The second method consists in explicitly defining a function that build successors of a dart in a map and proves particular properties of this function. At the theoretical level, this method has the downside to force some choices on the successor computing process, and thus implicitly adding a number of additional constraints to the initial problem. At the practical level in our case, this method has another downside which stems from the fact that this function would be partial: indeed, a dart has no successor at most dimensions. There are two main techniques to define partial functions in Gallina. The first one consists in manipulating objects supplied with a proof that they belong to the definition domain. The second one, more simpler from both a conceptual and practical point of view, consists in using the notion of *type with error*, a basic notion of functional programming: if the argument of a function belongs to its domain, then it yields the image by the function; if not then it yields a special value noted error. Types with error are defined in the standard library of Coq as inductive objects constructed from other types.[2]

Definition 26 (*TYPE WITH ERROR*). *Let T be a type. A value of type "T with error" (noted* (Exc *T*) *is either a value of type* T, *either constant* (error *T*)

```
  Inductive Exc : Set → Set:=
          value : (∀T : Set) T → (Exc T)
        | error : (Exc T)
```

The following recursive function allows to compute the oldest k-successor (i.e. the earliest inserted) of a dart in a map, provided such a dart exists, and yields (error (Exc dart)) otherwise.

Definition 27 (*COMPUTING OF A SUCCESSOR*).

```
  Fixpoint getsucc : nat → dart → fmap → (Exc dart) :=
     λk:nat. λx:dart. λm:fmap.
     Cases m of
```

[2] We have slightly altered this definition to take advantage of the implicit arguments mechanism.

```
    v ⇒ (error dart)
  | (i _ m') ⇒ (getsucc k x m')
  | (l (c k' x' y') m') ⇒ Cases (getsucc k x m') of
      (value y) ⇒ (value y)
    | error ⇒ Cases (EQ_NAT_DEC k k') of
        (left p1) ⇒ Cases (EQ_DART_DEC x x') of
          (left p2) ⇒ (value y')
        | (right p3) ⇒ (error dart)
        end
      | (right p4) ⇒ (error dart)
      end
    end
```

First of all, let us take a look at this definition. Keyword `Fixpoint` is used to define a function that is recursive on its last argument (here `m`). The construction `Cases X of ... end` is a conditional definition, the condition of which is one the shape of term `X`, and that yields different results depending on the type of constructor used to build term `X`. This is simply standard pattern-matching. Presence or absence of a type variable applied to constructor `error` depends on Coq's ability to infer the corresponding type. The function proceeds by analysing the shape of the free map argument `m`:

– If `m=v` then the search fails, so it yields (error *dart*).
– If `m=(i d m')` then this last insertion is ignored, and the search yields the value yielded by `(getsucc k x m')`.
– If `m=(l (c k' x' y') m')` then the search is first continued in `m'`. To do so, `(getsucc k x m')` is computed and its result analysed:
  • If `(getsucc k x m')=(value y)` then the search succeeded in `m'`. The search in `m` can then yield the same value.
  • If `(getsucc k x m')=(error` *dart*`)` then the search failed in `m'`. This means that `x` has no `k`-successor in `m'`. So, the search must also fail in `m=(l (c k' x' y') m')`, unless in the case that both `k=k'` and `x=x'`, where the search succeeds and yields `y'`. To identify whether it is the case, `k` must be compared to `k'`. To do so, we use lemma `EQ_NAT_DEC`. Let us consider the term `(EQ_NAT_DEC k k')`: it is of type `k=k'`$\vee\neg$`k=k'`, i.e. a proof that either `k` and `k'` are equal, or that they are distinct. Remember that as we are working in an intuitionnistic logical framework, this means that term `(EQ_NAT_DEC k k')` is either built from a proof of `k=k'` or from a proof of $\neg$`k=k'`. We will then study two cases, depending on which of these proofs `(EQ_NAT_DEC k k')` is built from. This term is a disjunction, which in Gallina is a simple inductive type with two constructors `left` et `right`.
    * If `(EQ_NAT_DEC k k')=(left p1)` then this means that `(EQ_NAT_DEC k k')` is built from a proof `p1` of `k=k'` (the *left* half of the considered disjunction). This entails that we are in the case where `k=k'`. We will compare `x` to `x'` in a similar fashion by analysing `(EQ_DART_DEC x x')`:
    · If `(EQ_DART_DEC x x')=(left p2)` then we deduce that `x=x'`. In this case, the search succeeds and yields `(value y')`.

  · If `(EQ_DART_DEC x x')=(right p3)` then it means that `(EQ_DART_DEC x x')` is built from a proof p3 of `¬x=x'` (the *right* half of the considered disjunction). This entails that we are in the case where `¬x=x'`. So the search still fails and yields `(error dart)`.

* If `(EQ_NAT_DEC k k')=(right p4)` then we deduce that `¬k=k'`. The search fails, and yields `(errordart)`.

We are easily convinced that this function does compute a successor if there is one, and fails if there is not any. We must now ensure this, by proving how `getsucc` relates to selectors `succ` and `nosucc`.

Lemma 16 (GETSUCC TO SUCC OR NOSUCC). *If a call to* `getsucc` *for dart* `x`, *dimension* `k` *and map* `m` *fails*, *then* `x` *has no* `k`*-successor in* `m`; *if it yields a dart* `y` *then* `y` *is a* `k`*-successor of* `x` *in* `m`

```
Lemma GETSUCC_SUCC_NOSUCC : (∀k:nat; ∀x:dart; ∀m:fmap)
        Cases (getsucc k x m) of
          error ⇒(nosucc k x m)
        | (value y) ⇒(succ k x m y)
        end.
```

Lemma 17 (NOSUCC TO GETSUCC). *If* `x` *has no* `k`*-successor in* `m`, *then the call* `(getsucc k x m)` *fails*

```
Lemma NOSUCC_GETSUCC : (∀k:nat; ∀x:dart; ∀m:fmap)
        (nosucc n x m)
        →(getsucc n x m)=(error dart).
```

Lemma 18 (SUCC TO GETSUCC). *If* `y` *is a* `k`*-successor of* `x` *in* `m`, *then the call* `(getsucc k x m)` *yields a value* (*which may or may not be* `y`)

```
Lemma SUCC_GETSUCC : (∀k:nat; ∀x,y:dart; ∀m:fmap)
        (succ k x m y)
        →(∃y':dart | (getsucc k x m)=(value y')).
```

If we only prove these properties on `getsucc`, the advantage of using a function over a lemma to compute a successor is essentially to achieve greater transparency: we have simply proved differently the same proposition that was proved by `SUCC_OR_NOSUCC`, i.e. that we are able to decide whether a dart has any `k`-successors; only we do it in a more readable way with function `getsucc` instead of a process hidden inside the proof of `SUCC_OR_NOSUCC`. On the other hand, using a function allows to prove easily properties such as stability by `l`, which makes this technique much more adapted to inductive reasoning:

Lemma 19 (VARIATIONS OF GETSUCC BY L). *Term* `(getsucc k x m)` *reduces to the same value as* `(getsucc k x (l (c k' x' y') m'))` *unless* `(getsucc k x m)` *yields* `(error dart)`, `k=k'` *and* `x=x'`, *in which case the term reduces to* `(value y')`:

```
Lemma GETSUCC_L : (∀k,k':nat; ∀x,x',y':dart; ∀m:fmap)
        ((getsucc k x (l (c k' x' y') m))=(getsucc k x m)
    ∧¬((getsucc k x m)=(error dart)∨k=k'∨x=x'))
  ∨((getsucc k x (l (c k' x' y') m))=(value y')
    ∧((getsucc k x m)=(error dart)∨k=k'∨x=x'))
```

The goal that we were pursuing by using a function is not to explicitly set well-chosen constraints in order to obtain a beforehand predetermined successor computing process, but simply to have at our disposal a behaviour model of this process in situations that may not have been expected at the time of function definition. For instance, at the time we specified successor computing, it was not obvious that we would need to know how it behaves with respect to `l`. If we had used a lemma, it would have been very difficult to express and determine this. As we used a function, we are able to determine this behaviour, and we express it in lemma `GETSUCC_L`. Similarly, we define `GETSUCC_I`.

One problem with the definition of `getsucc` is that it yields values in (*Exc dart*) instead of *dart*. This complicates the definitions of operations that use the darts computed by `getsucc`, as we always have to take into account the case where the search fails, although it sometime could not come up in reality. Intuitively, we would like to wrap `getsucc` in a function with the same arguments that will either yield the dart computed by `getsucc` if such a dart exists, or a default dart else. To do so, we define a general error handling function, then use it with `getsucc`:

Definition 28 (*ERROR HANDLING*).

```
Definition err : (∀A:Set; ∀default:A; ∀e:(Exc A))
   := λA:Set. λdefault:A. λe:(Exc A).
      Cases e of
        error ⇒default
      | (value x) ⇒x
      end
```

Definition 29 (*WRAPPING GETSUCC*).

```
Definition getsucc' : dart →nat →dart →fmap →dart
   := λdef:dart. λk:nat. λx:dart. λm:fmap.
      (err def (getsucc k x m))
```

In particular, we may choose the dart the successor of which is sought as the default dart. The resulting function is noted `sos` (successor or self).

Definition 30 (*SUCCESSOR COMPUTING, IDENTITY AS DEFAULT*).

```
Definition sos : nat →dart →fmap →dart
   := λk:nat. λx:dart.
      (getsucc' x k x)
```

We show lemmas for `sos` that are analogous to `GETSUCC_L` in order to ensure the properties of `sos`. One of them shows the compatibility between `sos` and `alpha`.

Lemma 20 (COMPATIBILITY BETWEEN ALPHA AND SOS).

```
Lemma ALPHA_SOS : (∀k : nat) (∀m:fmap; ∀x:dart)
      (exd x m) →(alpha k m x (sos k x m)).
```

For any dart in any map, sos allows to compute one of its α-successors at a given dimension.

5.6. Observational equality on maps

As we mentioned before, the sets of darts and sewings that make up a free map are represented by two (interwoven) lists. This choice of representation has two well-known downsides, namely allowing duplicates and inducing an order on the elements of the dart and sewings sets. In particular, two maps m1 and m2 that feature the same darts and sewings, but in which they have been inserted in different orders, are distinct because the terms that represent them are syntactically distinct. Similarly, if a dart that belongs to m1 is inserted again into m1, the resulting map m3 is distinct from m1. But from the semantical standpoint, m1, m2 and m3 should be equal. This shows in our choice of selectors: every one of them has the same behaviour whether it is used on m1, m2 or m3. That is why we use them as a basis of our own map equality, an *observational* equality noted $\cong$.

Definition 31 (*OBSERVATIONAL EQUALITY ON FREE MAPS*). Two free maps are *observationally equal* if the behaviour of alpha and exd is the same on both maps

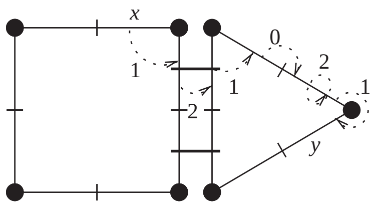
```
Inductive ≅ : fmap → fmap → Prop:=
   FMEQ : (∀m,m' : fmap)
            ((∀x : dart) (exd x m) ↔ (exd x m'))
            → ((∀n:nat; ∀x,y:dart)
                  (alpha n m x y) ↔ (alpha n m' x y))
            → (m≅m')
```

The main problem related to observational equality is that it is not possible to replace equals with equals in formulas, as is the case with Leibniz equality. An option would be to add the axiom ((∀m,m' : *fmap*) m≅m' → m=m'), which would let us replace observational equals with equals. While convenient, it turns out to be unsatisfying as it can easily be shown that this proposition is false in our specification. Adding it as an axiom would thus render the specification contradictory. However, this contradiction could possibly be tolerated, provided we are always careful never to use it to prove anything by absurd. But given the large size of our specification, it is risky to assume that we always fully control every use of the axiom. As a consequence, we cautiously rejected this axiom, at the expense of some unwieldiness in the specification and proof processes.

5.7. Paths

In order to define some operations, we need the notion of path in a map. A path is not a series of darts, but an itinerary, i.e. a succession of directions to be taken from any starting dart.

Definition 32 (*PATH*). A path is a finite series of dimensions. It is isomorphic to a list of naturals; pnil is the empty path, (pcons n p) is path p preceded with an



y=(follow m x (pcons 1 (pcons 2 (pcons 1 (pcons 0 (pcons 2 (pcons 1 pnil))))))

Fig. 6. Following a path.

extra direction n

```
  Inductive path : Set :=
              pnil : path
            | pcons : nat → path → path
```

Note that a direction might be ambiguous, as in maps where succ is not functional, a dart may have several successors for any given dimension; in that case, which sewing is being referred to is not obvious. Hence, this kind of paths is not well-suited for such maps. However, they are perfectly unambiguous in the case of g-maps.

We define on maps some standard operations on paths that are inspired from linear list operations. They are not given in detail: papp (of type *path → path → path*, that concatenates two paths), plength (of type *path → nat*, that computes the length of a path), pmirror (of type *path → path*, that computes the reverse path). We also define a number of operations that are related to the semantics of path.

Definition 33 (*FOLLOWING A PATH*). Following the empty path in any map from any dart leads to that same dart. Following path (pcons k p) in map m from dart x consists in following p in m from the image dart of x by (sos k) in m Fig. 6

```
  Fixpoint follow : fmap → dart → path → dart
      := λm:fmap. λx:dart. λp:path.
         Cases p of
           pnil ⇒ x
         | (pcons k p') ⇒ (follow m (sos k x m) p')
         end
```

Notice that as we work in free maps, in which exd and succ are completely separate, the darts reached by follow may or may not belong to the map. Whenever there is an ambiguity about which sewing to cross (when a visited dart has several successors for the dimension that corresponds to the next direction), follow systematically picks the earliest inserted one.

Definition 34 (*PATHS THE DIRECTIONS OF WHICH SATISFY A PREDICATE*). Let P be a predicate on integers. Path p satisfies predicate (pathpr P) iff all directions in p satisfy P

```
  Inductive pathpr : (nat → Prop) → path → Prop
```

```
    := PATHPR_PNIL : (∀P : (nat → Prop)) (pathpr P pnil)
    | PATHPR_PCONS : (∀P:(nat → Prop); ∀k:nat; ∀p:path)
          (pathpr P p) → (P k) → (pathpr P (pcons k p))
```

While leading in a proof, it is often needed to decide whether two darts may be joined by a path the directions of which satisfy some properties, and to compute such a path if it exists. These properties are usually maxima or minima for the value of the directions. These properties are wrapped in a predicate that specifies the directions that may be used. A priori, the only property this predicate must have is decidability, so that it can be checked during path construction. Let us formalize predicate decidability.

Definition 35 (*DECIDABILITY OF A PREDICATE*). Let T be a concrete type and P a predicate on this type. Then P is decidable iff for all x of type T it can be determined whether (P x) is satisfied or not

```
  Definition decpr : (∀T : Set) (T → Prop) → Set
    := λT:Set. λP:(T → Prop).
          (∀x : T) (P x)∨¬(P x)
```

Now we want to define getpath, a function that builds such a path and fails if none exists. Function getpath is supposed to satisfy the following fundamental lemma that states what it is supposed to do.

Lemma 21 (COMPATIBILITY BETWEEN FOLLOW AND GETPATH). *Let* m *be a free map*, x *and* y *two darts*, P *a predicate on sewings and* dec *a proof of the decidability of* P. *Then either* (getpath dec m x y) *yields a path that leads from* x *to* y *in* m *and all dimensions of which satisfy* P, *or* (getpath dec m x y) *fails*, *and there is no such path*

```
  Lemma GETPATH_FOLLOW : (∀P : (nat → Prop))
    (∀dec:decpr P; ∀m:fmap; ∀x,y:dart)
    Cases (getpath dec m x y) of
      (value p) ⇒ (pathpr P p) ∧ (follow m x p)=y
    | error ⇒ ((∀p : path) (pathpr P p)
                         →  ¬(follow m x p)=y)
```

Note that this is still formally a conjecture, as we have not declared function getpath yet. Let us now do right after introducing an intermediary lemma that is used to shorten definitions.

Lemma 22 (COMBINATION OF TWO CASE DISJUNCTIONS). *Let* p1, p2, q1 *and* q2 *be four propositions. If we know that either* p1 *or* p2 *is satisfied and that either* q1 *and* q2 *is satisfied*, *then we know that either both* p1 *and* q1 *are satisfied*, *or* p2 *or* q2 *is satisfied*

```
  Lemma AND_DEC : (∀p1,p2,q1,q2 : Prop) :
        p1∨p2 → q1∨q2 → p1∧q1∨p2∨q2
```

The main purpose of AND_DEC is to group two tests: given p and q two propositions and decp and decq proofs of their respective decidabilities, lemma AND_DEC allows to test whether they are both true or if one of them is false. This is how we use it in the definition of path computing.

Definition 36 (*PATH COMPUTING*). Let `P` be a predicate on naturals, `dec` a proof of its decidability, `m` a free map, and `x` and `y` two darts. If `x=y`, then path computing yields empty path `pnil`. Otherwise, if `m` is the empty map, then computing fails. If `m` is of the shape `(i d m')` then the path is computed in `m'`. If `m` is of the shape `(l (c n' x' y') m')`, the computing is first performed in `m'`. If it succeeds, the obtained path is returned. If it fails, if sewing `(c n' x' y')` satisfies `P`, if `x'` has no `n'`-successor in `m'`, and if path computing in `m'` from `x` to `x'` and `y` to `y'` succeeds in both cases and, respectively, yields path `xx'` and path `y'y`, then path computing yields `(papp xx' (pcons n' y'y))`. If any of these conditions is not satisfied, then path computing fails

```
  Fixpoint getpath : (∀P : (nat → Prop) → (decpr P) →)
                fmap → dart → dart → (Exc path) :=
    λP:(nat → Prop). λdec:(decpr P). λm:fmap.
    λx,y:dart. Cases (EQ_DART_DEC x y) of
      (left _) ⇒ (value pnil)
    | (right _) ⇒ Cases m of
         v ⇒ (error path)
       | (i _ m') ⇒ (getpath dec m' x y)
       | (l (c n' x' y') m') ⇒
           Cases (getpath dec m' x y) of
             (value p) ⇒ (value p)
           | error ⇒ Cases (AND_DEC (dec n')
                        (NOSUCC_DEC n' x' m')) of
                 (left _) ⇒ Cases (getpath dec m' x x') of
                    (value xx') ⇒
                     Cases (getpath dec m' y' y) of
                       (value y'y) ⇒
                              (papp xx' (pcons n' y'y))
                     | error ⇒ (error path)
                     end
                  | error ⇒ (error path)
                  end
               | (right _) ⇒ (error path)
               end
           end
       end
    end
```

This is a rather standard algorithm. It consists in rebuilding the map sewing by sewing, and at each step check if the last added sewing was the missing link to complete the sought path. Between the various ways to express this algorithm, we had to make sure that the one we picked yielded a path that was compatible with `follow`, i.e.

that following the computed path with `follow` from the first argument dart does lead to the second argument darts. Indeed, there might be some issues as to which sewing to cross when there are ambiguities. This property is ensured by the test `(NOSUCC_DEC n' x' m)` in the definition of `getpath`: for any dart `x'` and any dimension `n'` it allows to only take ignore all but the earliest inserted `n'`-sewing that links `x'` to another dart, as it is the one that is taken into account by `getsucc` and thus by `follow`. Assume that this test were absent: in that case, under the hypotheses that ¬`x=y` and ¬`y=z`, we would for example have `(getpath dec (l (c 0 x z) (l (c 0 x y) v)) x y)=(value (pcons 0 pnil))` while `(follow (l (c 0 x z) (l (c 0 x y) v)) x (pcons 0 pnil))=z`≠`y`. We now formally express the compatibility between the two notions.

5.8. Generalized maps

A generalized map `m` of dimension `n` can mathematically be described as a finite set of darts with `n+1` binary relations on the darts of this set, said relations being involutions that satisfy a number of additional properties. As we stated earlier, we formalize it with a triplet made up of a free map called *support of the map* and noted `(gsupport m)`, a natural `(gdim m)` representing its dimension and a proof of its well-formedness that expresses the fact that `(gsupport m)` is a good candidate to be a `(gdim m)`-g-map. Obviously, in order for `(gsupport m)` to be well-formed, relations `(alpha i (gsupport m))` must be involutions satisfying the properties of the α_i of the g-maps. However, this is not enough, as there are two more issues left to tackle:

- Using a free map as a support does not ensure that only the darts of the map will be sewn. Thus, we will assume that relations `(alpha i (gsupport m))` are *localized*, i.e. that they only link darts that belong to `(gsupport m)`;
- We have an infinity of relations `(alpha i)`, while the mathematical definition states that there are only $n+1$ relations α_i; thus, we must simulate the finiteness of the number of relations α_i. To do so, there are two approaches:
  - Always make sure that only sewings the dimensions of which are low enough are manipulated by adding hypotheses to theorems;
  - Assume that relations `(alpha i (gsupport m))` are equal to identity for higher dimensions.

  The first solution is cleaner, but as a downside it sigificantly increases the number of premises of the lemmas dealing with the `(alpha i)`, which are the majority of the proved lemmas, which makes their formulation and use all the more complicated. For practical reasons, we then choose the second solution.

Another small problem also stems from the dimension. As seen previously, the dimension of a g-map is an integer that is greater or equal to -1. As we have so far only worked with naturals, we would rather not introduce integers only to respect this dimension convention. This is why an `n`-g-map is formalized by an *`ngmap`* `m` such that `(gdim m)=(n+1)`. In our specification, the dimension is equal to the number of different possible sewing dimensions in this map rather than the maximal dimension of these sewings. This does not change the numbering of the α_k, and they have the same semantics in the standard definition as in ours. With this new convention, we can write the well-formedness predicate.

Definition 37 (*WELL-FORMEDNESS OF FREE MAPS*). A free map `m` is well-formed at dimension `dim` if for any `n` the relation-on `(alpha n)` is an involutive permutation localized on `m`, and if for any `n` such that `n+2`$\leqslant$`dim` relation `(alpha n m)` is irreflexive, and if for any `n` and `n'` such that `(n+3)`$\leqslant$`(n'+1)`$\leqslant$`dim` relation `(alpha n m)`$\circ$`(alpha n' m)` is involutive, and if for any `n` such that `n`$\geqslant$`dim` relation `(alpha n m)` is the identity:

```
Inductive wf : nat → fmap → Prop
    := WF : (∀dim:nat; ∀m:fmap)
       ((∀n : nat) (permutation exd (alpha n) m)
                   ∧ (localized exd (alpha n) m)
                   ∧ (involutive (alpha n m)))
       → ((∀n : nat) (n+2)<=dim
                   → (irreflexive (alpha n m)))
       → ((∀n,n' : nat) (n+2)<=n' → (n'+1)<=dim
             →  (involutive (composition
                   (alpha n m) (alpha n' m))))
       → ((∀n : nat) dim<=n → (identical (alpha n m)))
       → (wf n m)
```

This definition is very close to the mathematical definition of g-maps, thus we can be confident that it is actually g-maps that we are implementing. We can now define the type of g-maps.

Definition 38 (*G-MAP*). A g-map is a triplet made up of a free map `m`, a dimension `n` and a proof that `m` is well-formed at dimension `n`, i.e. a term of type *(wf n m)*:

```
Inductive gmap : Set
    := mkg : (∀n:nat; ∀m:fmap; ∀w:(wf n m)) gmap
```

This definition means that an object of type *gmap* may only be built with the help of constructor `mkg`. Constructor `mkg` has three arguments: a natural number, a free map and a proof term of a given property. Thus, applying three proper arguments to `mkg` yields an object of type gmap. For instance, if `WF_V_2` is a proof that the empty map is well-formed at dimension 2, then `(mkg 2 v WF_V_2)` has type *gmap*. Moreover, injectivity of constructors ensures that two objects of type *gmap* are equal iff the same arguments of `mkg` were used to build them.

As *gmap* is of type *Set*, we may use it directly as a type in definitions and lemmas, and thus use quantifiers on the *gmap* without having to use quantifiers on each of the three objects that they are made of. Thus, we may define the two selectors associated to *gmap* by combining quantifiers with terms of type *gmap* and then destructuring them.

Definition 39 (*DIMENSION OF A G-MAP*).

```
Definition gdim : gmap → nat
    := λm:gmap.
          Cases m of (mkg n m w) => n end
```

Definition 40 (*SUPPORT OF A G-MAP*).

```
Definition gsupport : gmap → nat
    := λm:gmap.
           Cases m of (mkg n m w) => m end
```

We now would like to test the existence of darts and sewings in a *gmap*. To do so, we may define new selectors on *gmap*, for instance by using the ones defined on *fmap*.

Definition 41 (*EXISTENCE OF DARTS IN A G-MAP*). A dart belongs to a g-map if it belongs to its support

```
Definition gexd : dart → gmap → Prop
       := λx:dart. λm:gmap. (exd x (gsupport m))
```

The main problem with this method is that a selector on *gmap* must be defined for each selector that should be inherited from *fmap*, and provided with associated lemmas (inversion lemmas, ...). The selectors of the *fmap* cannot be directly applied to the *gmap*, as the typing rules would not be respected. Coq features a *coercion* mechanism that allows us to solve this problem. First, `gsupport` is declared as a coercion:

```
Coercion gsupport : gmap >-> fmap
```

This command means that whenever a term of type *fmap* is expected and that in its place there is a term `m` of type *gmap*, then this term must be read as `(gsupport m)`. For instance, `(∀x:dart; ∀m:gmap) (exd x m)` is simply a more readable and all-around more convenient version of `(∀x:dart; ∀m:gmap) (exd x (gsupport m))`. This mechanism allows to transparently project *gmap* onto *fmap*, and thus from the user point of view to apply the free maps selectors to generalized maps. Similarly, *gdim* is declared as the coercion from *gmap* to *nat*. This way, we may express that `m` is a 2-g-map with proposition `m=3` (remember that there is a difference of one between the dimensions of the represented g-maps and their representations). An example lemma that is expressed with coercions states the symmetry of `succ` in generalized maps.

Lemma 23 (SYMMETRY OF SUCC IN GMAP). *Let* `k` *be a dimension*, `x` *and* `y` *two darts and* `m` *a gmap. Then if* `y` *is* `k`*-successor of* `x` *in* `m`, *then* `x` *is also* `k`*-successor of* `y` *in* `m`:

```
Lemma G_SUCC_SYM : (∀k:nat; ∀x,y:dart; ∀m:gmap)
    (succ k x m y) → (succ k y m x)
```

As for all inductive types, Coq automatically generates an induction principle on the *gmap*.

Lemma 24 (STANDARD INDUCTION ON GENERALIZED MAPS). *If for any dimension* `n`, *any free map* `m` *and any proof* `w` *of* `(wf n m)`, *the generalized map made up of* `n`, `m` *and* `w` *satisfies* `P`, *then* `P` *is satisfied by all g-maps*

```
Lemma gmap_ind : (∀P : (gmap → Prop))
       ((∀n:nat; ∀m:fmap; ∀w:(wf n m)) (P (mkg w)))
       → (∀m : gmap)(P m)
```

This induction principle, although generated with the same automated methods as in the case of the *fmap*, is semantically quite different. What we would like is a theorem analogous to the one on free maps, that would allow to incrementally prove the properties of a g-map. The first approach is to try and perform induction on the support of the g-map, which is a free map. This is no trivial, but nonetheless may be done after some manipulations. The main problem is that `gmap_ind` is applied to predicates on *gmap*. This means that the predicates describe properties not only of the support and dimension of the g-map, but also on the proof of well-formedness of this generalized map; but as the term of the support also appears in the well-formedness proof term, induction cannot be directly performed. However, the well-formedness proof itself usually is not important; all that matters is that it exists. The properties we are interested in only deal with the support and dimension of the g-maps, and not the well-formedness proofs. Thus, we may introduce a new induction principle that applies to predicates on integers and free maps:

Lemma 25 (INDUCTION ON G – MAPS WITH RESPECT TO THEIR SUPPORT). *Let* P *be a two-place predicate on a natural and a free map. Let us assume that for each* n *for which* v *is well-formed at dimension* n, P *is satisfied for* n *and* v. *Then assume that for any free map* m *and any natural* n *that satisfy* P *provided* m *is well-formed at dimension* n, P *is preserved by insertion of dart provided the resulting map is also well-formed. Finally, assume the same for insertion of sewings. Then* P *is satisfied for the dimension and support of a generalized map*

```
Lemma GMAP_FMAP_IND : (∀P : (nat → fmap → Prop))
      ((∀n : nat) (wf n v) → (P n v))
      → ((∀n:nat; ∀m:fmap; ∀x:dart)
            ((wf n m) → (P n m))
            →  (wf n (i x m)) → (P n (i x m)))
      → ((∀n:nat; ∀m:fmap; ∀s:sw)
            ((wf n m) → (P n m))
            → (wf n (l s m)) → (P n (l s m)))
      → (∀m : gmap) (P m m)
```

Notice that the use of coercions in the ending `(P m m)`: the first occurrence of `m` is coerced into its dimension, the second into its support. This lemma can be used to reason by induction this way: let `P` be the predicate that we try to prove. Then, each of the three premises of `GMAP_FMAP_IND` should successively be proved. First, we prove that `(P n v)` is true for any `n`; this is the basis of the induction. Then, we set `n`, `m` and `x`, we assume `(wf n m) → (P n m)` and `(wf n (i x m))`, then we try to prove `(P n (i x m))`; this is an induction step. The strategy with this kind of induction principle is either to prove that `(wf n (i x m))` is contradictory, or to prove that `(wf n m)` is also satisfied in order to be able to use the induction hypothesis to show `(P n m)` and then deduce `(P n (i x m))` after some calculations; this corresponds to the "recurrence step". The main problem is that, as the g-maps are synthetically defined, it is uncommon that `(wf n (i x m))` entails `(wf n m)`; even in very simple cases such as the ones where `n`$\geqslant$`2` and `m=v`. The induction hypothesis can then only seldom be used, and only in cases that are difficult to qualify. The same problem arises with constructor `l`. Using this induction principle as a consequence is practically impossible as far as useful properties go.

As we have just seen it, the induction hypothesis may not be used because well-formedness is not stable by constructors `i` and `l`. The induction principle on g-maps based on the structure of the support all suffer from the same problem, i.e. a deep structure discrepancy between the support of a g-map and the proof that it is well-formed. But the normalization theorem is proved by induction on surfaces, so we cannot skip finding a way to lead induction proofs at least on g-maps if dimension 2. To solve this incompatibility, we introduce s-maps.

6. S-MAPS

Our solution to this problem is based on an alternative definition of generalized maps. Indeed, generalized maps are often presented in a very different manner from ours [13]: using our numbering, a 0-g-map is seen as a dart set. A 1-g-map is seen as a closed 0-g-map (it is the case for all 0-g-maps). A 2-g-map is seen as a closed 1-g-map (i.e. such that no dart is 0-sewn to itself) in which only whole cells of dimension 1 (i.e. edges) have been sewn. In general, an n-g-map is seen as a closed $(n-1)$-g-map (i.e. such that α_{n-1} is irreflexive) in which whole $(n-1)$-cells have been n-sewn. For instance, for $n=2$, a 2-g-map (intuitively representing a surface) is a closed 1-g-map (a collection of edge cycles) in which cells of dimension 1 (edges) are sewn at dimension 2. An n-g-map is built by applying several times the $(n-1)$-cell sewing to an $(n-1)$-g-map. Thus, the sewing of cells may generate all of these generalized maps. This naturally allows to reason by induction on generalized maps, as they are the set of objects generated by sewing of cells. So we must prove that the two ways to define g-maps are equivalent, in order to extend all results obtained by induction on g-maps defined by cell sewing to the ones we have used so far. So far, this equivalence had never been proved. We proceed this way:

1. we start by defining on free maps a high-level operation noted `sm` (**s**ewing of **m**aps) that intuitively corresponds to cell sewing;
2. we then consider the set of free maps generated with `v`, `i` and `sm` while always respecting some preconditions. This set is noted *`smap`*;
3. we show that any *`smap`* is well-formed;
4. we show that any *`gmap`* is observationally equal to an *`smap`*.

From the last two points we deduce that *`smap`* and *`gmap`* are observationally equivalent. If we prove on *`smap`* a property that respects observational equality, may be using induction reasoning, then the property may easily be extended to *`gmap`* by using the equality. All propositions defined only with `exd` and `alpha`, including all propositions on objects represented by g-maps, belong to this category. Thus, we have a way to reason by induction on surfaces represented by the *`gmap`* in order to prove the most useful properties.

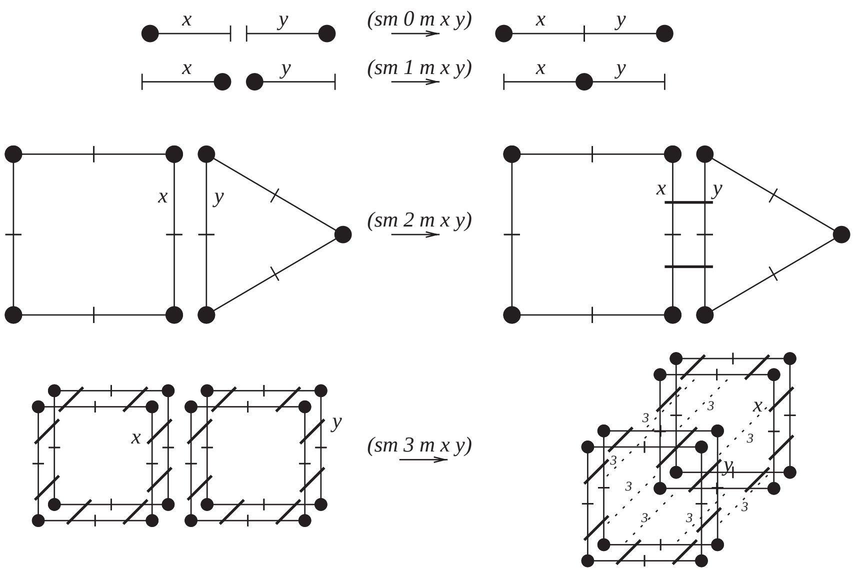


Fig. 7. Example of application of the sewing of cells.

6.1. Cell sewing

Like `getpath` (Section 5.7), cell sewing is programmed in a very straightforward and inefficient manner, as the main goal is to obtain a simple and correct algorithm. For all aforementioned reasons, it is defined on free maps, though in practice it is only applied to g-map supports. Cell sewing is depicted at several dimensions in Fig. 7. To perform the sewing of `n`-dimension cells in free map `m`, the algorithm is the following:

– Pick a dart `x` in one of the cells to be sewn.
– Pick in the other cell the one dart `y` to which `x` should end up sewn to.
– Browse `m` to determine all darts in `m` that can be reached from `x` by crossing only sewings of dimension lower or equal to `n-2`; thus obtaining the darts of `m` belonging to the same cell of dimension `n` as `x`. Each of these darts is characterized by the path $\mathtt{p}_i$ that is taken while browsing to reach it from `x`, and thus is of the form `(follow m x` $\mathtt{p}_i$`)`.
– n-sew each of these darts, of the form `(follow m x` $\mathtt{p}_i$`)`, to its match in the other cell, i.e. `(follow m y` $\mathtt{p}_i$`)`.

To determine the set of `(follow m x` $\mathtt{p}_i$`)`, every dart of the map is checked with `getpath`, by trying to build a path from `x` to it such that all its directions are lower or equal to `n-2`, thus ensuring that it belongs to the same $(n-2)$ cell as `x`. To test each of these darts, `m` is deconstructed in the course of the operations. But as computing the paths requires the whole map, it must be kept as a whole somewhere. Map `m` actually performs two distinct roles in this algorithm: this of recursion variable, for which it is deconstructed, and this of background for path computing, for which it must remain untouched. This is a common situation in functional programming, which is solved by building cell sewing with two separate functions. The more general one, `sm_int` (*int*ermediary), reifies the above algorithm, but gives the two roles of `m` to two different variables, `mvar` and `mcst`. The link between the two is made thanks to the second function, `sm`, that simply applies `sm_int` to the particular value `m` for `mrec` and `msup`. Before specifying those two functions, we need to prove that the fact that a natural is lower by at least two to another natural is decidable, as this is the predicate that we want to give as argument to `getpath`.

Lemma 26 (DECIDABILITYOF ⩽ WRAPPED IN A PREDICATE).

```
Lemma LE2_DEC : (∀m : nat) (decpr (λn:nat. (n+2)⩽ m))
```

Definition 42 (*CELL SEWING, INTERNAL FUNCTIONS WITH TWO MAP PARAMETERS*).

```
Fixpoint sm_int :
          nat → fmap → fmap → dart → dart → fmap
    := λn:nat. λmvar,mcst:fmap. λx,y:dart.
    Cases mvar of
      v ⇒ mcst
    | (i x' mvar') ⇒
        Cases (getpath (LE2_DEC n) mcst x x') of
          (value p) ⇒ (l (c n x' (follow mcst y p))
                          (l (c n (follow mcst y p) x')
                             (sm_int n mvar' mcst x y)))
        | error ⇒ (sm_int n mvar' mcst x y)
        end
  | (l _ mvar') ⇒ (sm_int n mvar' mcst x y)
  end
```

Definition 43 (*CELL SEWING*).

```
Definition sm : nat → fmap → dart → dart → fmap
    := λn:nat. λm:fmap. λx,y:dart. (sm_int n m m x y)
```

As with all functions, we prove a set of lemmas that almost completely describe behaviour of `sm`. As they are meant to be very generic, these lemmas have the same structure as the definition of `getpath`, which makes them very unwieldy and difficult to handle. Each of them takes into account a lot of situations, and whenever they are used the part of the lemma corresponding to the current situation must first be extracted. This is very inefficient, and sometimes also quite difficult. To partially solve this problem, we define a larger set of narrower lemmas that only works in the most common situations, especially when the dimensions are lower or equal to 1; a useful sample lemma is:

Lemma 27 (INVERSION OF SM, DIMENSION 0 OR 1). *Let* `X`, `Y` *be two darts and* `k` *a dimension lower or equal to* 1. *If* `X` *is* `k`*-sewn to* `Y` *in* `(sm 1 m x y)`, *then* `X` *already was* `k`*-sewn to* `Y` *in* `m`, *otherwise either* `X=x` *and* `Y=y`, *or* `X=y` *and* `Y=x`

```
  Lemma SUCC_0_1_SM_INV :
        (∀k:dart; ∀m:fmap; ∀X,Y,x,y:dart)
           k⩽1
           →(succ k X (sm k m x y) Y)
           →(succ k X m Y)∧(X=x∧Y=y∨X=y∧Y=x).
```

6.2. Useful definitions for preconditions

In order to specify *smap*, it will be useful to isolate some notions that will be used to define some preconditions. There are three of them. The first one is a simple property of paths, namely having all its directions lower than a constant k by at least 3. Thus, following such a path from any dart always leads to a dart that belongs to the same $(k-3)$-cell. Why are we interested in $(k-3)$-cells? Because we are later going to sew them and wish to designate them in order to state some preconditions on them. It is formalized by the following Gallina definition.

Definition 44 (*LIMITATION OF THE DIRECTION OF PATHS*). A path satisfies `(pathle3 k)` if all its directions, when added 3, remain lower or equal to `k`

```
  Definition pathle3 : nat → path → Prop
     := λk:nat. (pathpr (λi:nat. 3+i⩽k)).
```

Note that there is a slight difference between this predicate and a variant that would state "all dimensions of the path are lower or equal to `k-3`". Indeed, as substraction is defined on naturals, the equality `k-3 = 0` stands for any `k⩽3`. In that case, the suggested variant would be satisfied by paths that only features `0` for directions, while `(pathle3 k)` would not.

The other two properties are less trivial. The first one is a cell similarity condition, a kind of isomorphism.

Definition 45 (*CELL SIMILARITY*). Let `m` be a free map, `k` a dimension and `x` and `y` two darts. The `k`-cell of `m` incident to `x` is *similar* at `x` and `y` to the one incident to `y` if, whenever a dart of the cell incident to `x` may be reached by several paths that all satisfy `(pathle3 k)`, following any of these paths from `y` always leads to the same dart; and if the property also stands when `x` and `y` switch roles

```
  Definition similar :
                    nat → fmap → dart → dart → Prop
     := λk:nat. λm:fmap. λx,y:dart.
           (∀p,q : path)
              (pathle3 k p) → (pathle3 k q)
              →((follow m x p)=(follow m x q)
                        → (follow m y p)=(follow m y q))
              ∧ ((follow m y p)=(follow m y q)
                        → (follow m x p)=(follow m x q)).
```

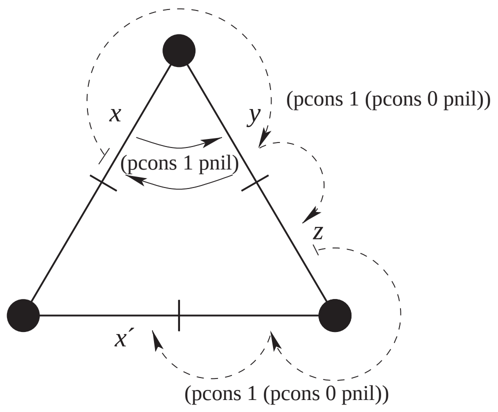


Fig. 8. Dart symmetry.

Note that for $k < 3$, $(k-3)$-cells are single darts, hence they are all similar. For $k=3$, $(k-3)$-cells are simple edges, thus they are also all similar. These are the only simple cases.

Before sewing cells, we would like to ensure that the two cells have the same structure: for example, we only want to sew octogons to octogons. However, our cell-sewing operation works for any two cells, with strange results when the cells do not share the same structure, i.e. when they are not similar. Thus, when we later sew $(k-3)$-cells, we first check that the cells are similar.

The second property deals with the set of paths that join two given darts without changing $(k-2)$-cells:

Definition 46 (*DART SYMMETRY*). Let `m` be a free map, `k` a dimension and `x` and `y` two darts. Darts `x` and `y` are symmetrical at dimension `k` if any path satisfying `(pathle3 (k+1))` that leads from `x` to `y` also leads from `y` to `x`:

```
Definition sympoints :
                nat → fmap → dart → dart → Prop
  := λk:nat. λm:fmap. λx,y:dart.
       (∀p : path) (pathle3 (k+1) p)
          → y=(follow m x p) → x=(follow m y p).
```

It is easy to see that dart symmetry is only meaningful for dimensions greater than $k \geqslant 3$. Indeed, it is a property of paths with directions no greater than $((k+1)-3) = (k-2)$. If $k < 2$, then the only such path is the empty path; symmetry is then a trivial proposition. It is also quite easy to see that all darts are symmetrical for $k=2$, because all considered paths are made up of zeroes. Thus, for $k \leqslant 2$, all darts are symmetrical.

An example of dart symmetry is given in Fig. 8. As we just said, all darts in this figure are symmetrical at dimension $k \leqslant 2$. For $k \geqslant 3$, `x` and `y` are symmetrical. The two main paths that lead from `x` to `y` are `(pcons 1 pnil)` and `(pcons 0 (pcons 1 (pcons 0 (pcons 1 (pcons 0 pnil)))))`. It is easy to see that both of these paths also leads from `y` to `x`. All other paths can be proved to also have this property. Thus, `x` and `y` are symmetrical at dimensions $k \geqslant 3$. On the other hand, it is not the case for `x` and `z`, as there exists a path, namely `(pcons 1 (pcons 0 pnil))`, that leads from `x` to `z`, but not from `z` to `x`. On this figure, `(x, y)`, `(x', y)` and `(y, z)` are pairs of symmetric darts, while `(x, z)`, `(x, x')` and `(z, x')` are pairs of dissymetric darts.

Like predicate `similar`, predicate `sympoints` will also be used when sewing $(k-3)$-cells, more precisely when sewing a $(k-3)$-cell to itself. This should only be allowed when the structure of not only the $(k-3)$-cell, but also the $(k-2)$-cell to which it is incident, is regular enough. This regularity is expressed by the notion of dart symmetry. Thus, sewing a $(k-3)$-cell to itself with `sm` should only be allowed when the darts that would be sewn are symmetric at dimension k (i.e. their common $(k-2)$-cell is regular enough). Predicate `sympoints` describes a sort of symmetry between two darts `x` and `y` that are incident to the same $(k-2)$-cell, in the intuitive sense that this cell must "look" the same seen from either dart, as an itinerary (i.e. a set of directions) that leads from `x` to `y` must as well lead from `y` to `x`. Notice that two darts that belong to different $(k-2)$-cells are obviously symmetrical at dimension `k`, as dart symmetry is a property of paths that lead from one dart to another without changing $(k-2)$-cells; no such path exists in this case so the property is trivial.

6.3. S-maps

We now focus on this special kind of maps, the sewings of which are generated by applications of `sm`, called s-maps, and of type *smap*. As they are meant to be partially normalized versions of g-maps, we want to be quite constraining in their definition, without excess lest they may be too rigid to properly handle. Type *smap* is defined like *gmap*, i.e. as a triplet made up of a dimension, a free map that is used as a support, and a proof that this map and this dimension satisfy a predicate of well-constructedness, analogous to the well-formedness predicate of *gmap*. Note that from this definition, types *gmap* and *smap* are a priori completely independent. Their only formal common point is that they are similarly typed. In particular, the dimension of well-constructedness is formally unrelated to the dimension of well-formedness; the same also holds for the supports. The actual bonds between these two types are the point of the two main theorems of this section. The predicate of well-constructedness is the smallest predicate satisfying the following properties:

1. The empty map is well-constructed at dimension `0`.
2. Well-constructedness at dimension 0 is stable by insertion of darts without duplicates.
3. A free map that is well-constructed at dimension `n` is also well-constructed at dimension `n+1` if either `n=0` or α_{n-1} is irreflexive.
4. Let `m` be a free map well constructed at dimension `n+1`, and `x` and `y` two distinct fixpoints of α_n. If the `n`-cell incident to `x` to the `n`-cell incident to `y` and if `x` and `y` are symmetrical at dimension `n`, then `(sm n m x y)` is also well-constructed at dimension `n+1`.

Well constructing a map is obviously very directed: the starting point is always the empty map, in which all darts are then inserted, then all sewings at dimension 0 are performed cell by cell, then all sewings at dimension 1, etc. Property 1 allows to start the construction, property 2 takes care of dart insertion. Property 3 is the one that allows to set the dimension; notice that a map is well-constructed at dimensions n and $n+1$ provided that all its darts are sewn at dimension $n-1$ and none at dimension n. Property 4 takes care of all sewings: it only allows whole cell sewing at a precise dimension, and only when specific preconditions are satisfied. Note that a cell may be sewn to itself under some preconditions. Well-constructedness is formalized by the following predicate.

Definition 47 (*WELL-CONSTRUCTEDNESS*).

```
Inductive wcst : nat → fmap → Prop :=
     WCST_V : (wcst 0 v)
   | WCST_I : (∀m:fmap; ∀x:dart)
          (wcst 0 m) → ¬(exd x m) → (wcst 0 (i x m))
   | WCST_INC : (∀m:fmap; ∀k:nat)
          (wcst k m)
          → ((∀pk : nat) (pk+1)=k
                    → (irreflexive (alpha pk m)))
          → (wcst (k+1) m)
   | WCST_SM : (∀m:fmap; ∀k:nat; ∀x,y:dart)
          (wcst (k+1) m) → x≠y
          → (alpha k m x x) → (alpha k m y y)
          → (similar k m x y) → (sympoints k m x y)
          → (wcst (k+1) (sm k m x y))
```

Using the same methodology as for *gmap*, we define *smap* as a triplet made up of a dimension, a free map and a proof of well-constructedness of this map at this dimension:

Definition 48 (*S-MAP*).

```
Inductive smap : Set :=
   mks : (∀n:nat; ∀m:fmap) (wcst n m) → smap
```

We then define selectors `sdim : smap → nat`, `ssupport : smap → fmap` and `swcst:((∀m : smap) (wcst (sdim m) (ssupport m)))` that allow to extract from an *smap*, respectively, its dimension, its support and a proof of its well-constructedness. However, unlike their counterparts `gdim` and `gsupport`, these selectors are not declared as coercions, for reasons explained later. Note a subtle point in the definition of dimension: an *smap* always has exactly one dimension, while its support may be well-constructed at several dimensions.

The first important property of the *smap* is their well-formedness: what we want to prove is that a free map that is well-constructed at dimension `k` is also well-formed at dimension `k`. To do so, we want to reason by induction. The question is, on what property should we reason by induction? Unlike most other proof styles, it is often necessary to try and prove a stronger proposition that the one sought when reasoning by induction. Indeed, the proposition that is studied will also appear as an induction hypothesis; there is a risk that the induction hypothesis may be too weak. In our case, we try to prove (wf k m) under the assumption (wcst k m). The first instinct is to simply try and prove proposition (wf k m). But the induction hypothesis thus obtained turns out to be far too weak, as it contains almost none of some much-needed information on selectors. Thus, we attack a stronger lemma in order to be able to use a stronger induction hypothesis.

Lemma 28 (WELL – FORMEDNESS OF WELL – CONSTRUCTED MAPS, BEHAVIOUR OF SELECTORS IN THESE MAPS). *Let* k *be a natural and* m *a well-constructed free map at dimension* k. *Then* m *is also well-formed at dimension* k. *Moreover, for any* k' *the* k'*-succession relation is irreflexive* (*i.e. no dart may be* explicitly *sewn to itself*). *Lastly, for any dart* x *and any natural* pk *such that* pk+1 *is greater or equal to* k, *if* x *is not sewn at dimension* pk, *then no dart of its* k*-cell is either*

```
Lemma WCST_WF_PROPS : (∀m:fmap; ∀k:nat)
    (wcst k m)
    → (wf k m)
      ∧ ((∀k' : nat)
         (irreflexive (λx:dart. (succ k' x m))))
      ∧ ((∀x:dart; ∀pk:nat; ∀p:path)
         k⩽(pk+1) → (exd x m)
         →(nosucc pk x m) → (pathle3 k p)
         →(nosucc pk (follow m x p) m)).
```

A problem with this method is that it produces a massive lemma with a sizeable proof and unfocused semantics, while in this domain short, narrow and simple theorems are heavily favored. However, reasoning by induction does not allow us to split this lemma into more elementary subparts; more focused subtheorems would be much longer to state than WCST_WF_PROPS itself. We can however perform a splitting of WCST_WF_PROPS after it has been proved, to obtain more focused lemmas. The first part of this theorem is thus extracted.

Theorem 29 (*well-formedness of well-constructed free maps*). *A well-constructed map at any dimension is also well-formed at the same dimension*

```
Theorem WCST_WF : (∀k:nat; ∀m:fmap) (wcst k m) → (wf k m).
```

We can deduce from this a function that projects *smap* into *gmap*.

Definition 49 (*PROJECTING SMAP INTO GMAP*). Let s be an *smap*. Then (s_ng s) is the *gmap* that shares its support and dimension with s, and the proof of well-formedness of which is obtained by application of WCST_WF to the proof of well-constructedness of s

```
Definition s_ng : smap → gmap
   λm:smap. (mkg (WCST_WF (swcst m)))
```

We may then naturally declare this function as a coercion, and thus obtain a default projection of the *smap* into *gmap*, in order to transparently apply the objects defined on *gmap* to *smap*:

```
Coercion s_ng : smap >-> gmap
```

Note that, because of coercion transitivity, *smap* are coerced to *fmap* by application of the composition of `s_ng` and `ng_support`. Selectors on *fmap* like `succ` and `exd` may then directly be applied to objects of type *smap* with the expected semantics. This is the reason why we did not previously declare `ssupport` as coercions from *smap* to *fmap*, as it was not only redundant but confusing, as there would have been two different ways to coerce *smap* to *fmap*. Coq would have picked the same way every time. If Coq picked the `ssupport` coercion, we would not be able to apply to *smap* any lemma on the selectors *fmap* that only apply to *gmap*, like for instance `NG_SUCC_SYM`.

6.4. Projecting *gmap* into *smap*

Obviously, it is not possible to easily project *gmap* into *smap* in the same way, i.e. by keeping the same support and dimension. Indeed, *smap* imposes strict constraints on its support, notably order constraints, that are absent in *gmap*. Thus, there are numerous *gmaps* the support of which are not well-constructed in the sense of *smap*, for example the ones in which some darts are inserted several times. However, as we already mentioned we are not very interested in some aspects of the *fmap*, for instance the insertion order of elements. We only are interested in studying *gmap* through selectors `exd` and `alpha`; that was the reason why we have defined an observational equality relation that expressed the fact that free maps were $\cong$-equal if `exd` and `alpha` were satisfied for the same values on these maps. As a consequence, we now decide to work at the observational level. Instead of proving that every well-formed map is also well-constructed, which is false, we instead show that every well-formed map is $\cong$-equivalent to a well-constructed one. To do so, we can either prove that such a map exists, or define a function that from a *gmap* computes a free map, that we then prove to be well-constructed and $\cong$-equal to the original map. For reasons mentioned in the `getsucc` section, we choose the second approach. The chosen algorithm works by extracting all darts of the argument map and insert them in the result map, then browse all sewings of the map and reproduce all the sewings that were present in the starting map with `sm` in the resulting map while avoiding duplicates. This algorithm is implemented with a function, noted `ng_s`, itself defined with three intermediary functions. The first one, `dartmap` removes all sewings and duplicate dart insertions from a free map to only keep the first dart insertions, thus obtaining the *duplicateless dart map* of its argument map.

Definition 50 (*COMPUTING THE DUPLICATELESS DART MAP*). Let `m` be an *fmap*. If `m=v` then its duplicateless dart map is `v`. If `m=(l s m')` then its duplicateless dart map is the one of `m'`. If `m=(i x m')` then its duplicateless dart map is the one of `m'` if `x` belongs to `m'`, or else the one of `m'` into which `x` was inserted

```
Fixpoint dartmap : fmap → fmap
    λm:fmap. Cases m of
```

```
  v ⇒ v
| (i x m') ⇒ Cases (EXD_DEC x m') of
        (left _) ⇒ (dartmap m')
      | (right _) ⇒ (i x (dartmap m'))
      end
| (l _ m') ⇒ (dartmap m')

end
```

This function obviously is used to gather the darts of the map. Notice that intermediary functions are defined on *fmap*, although they are actually only used on *gmap*. All that must be ensured is that they yield *smap* when applied to a *gmap*, no matter whether intermediary maps are ill-constructed or ill-formed.

The second intermediary function, noted `ng_s_1step`, is used to do all the sewings at a given dimension. It has four arguments: an integer—the handled dimension—, a free map noted `rest` that gives the initial value of the resulting map, and two free maps `mvar` and `mcst`. As with `sm_int`, they allow to distinguish the two roles of the support of the studied *gmap*, which are those of induction variable and background for successor computing. The used algorithm consists, for a given dimension `n`, in browsing the darts of the map and for each of them compute its `n`-successor-to-be, then sew their respective `n`-cells. At each step, two extra preconditions must be checked before actually performing the cell sewing:

– The dart must not already be sewn in the result map. Indeed, it may be part of an already treated `n`-cell, which has then already be sewn. We then do nothing for this dart as we do not want to sew darts several times.
– The dart must not be its own n-α-successor in the argument map. The definition of *smap* forbids explicit sewings of a dart to itself. In this case, the dart and all other darts in its `n`-cell remain implicitly n-α-sewn to themselves in the result map.

A third precondition is featured in the definition, being the existence of the dart in the argument map. However, it is always satisfied whenever the function is used when `mcst` is well-formed, which is the only case we later study in proofs. It just happens that this information is hard to exploit in these proofs, which is why we prefer adding a redundant test to our definition in order to easily have this information available. The three precondition tests are grouped together with `AND_DEC_3`, the three-argument version of `AND_DEC`. The inequality test is built from the equality test `EXD_DEC` using the test symmetrizing lemma `TEST_SYM`.

Definition 51 (*INTERMEDIARY FUNCTION, HANDLING OF A DIMENSION*). Let `k` be a dimension, and `rest`, `mcst` and `mvar` three free maps. If `mvar=v`, then the function yields map `rest`. If `mvar=(l _ m')` then the computing simply ignores this sewing and proceeds with `m'`. If `mvar=(i x m')` then three properties are tested: the absence of `k`-successor of `x` in the recursively obtained map, the inequality between `x` and its k-α-successor in `mcst`, and the existence of `x` in `mcst`. If all three are satisfied, then the function returns the recursively obtained map in which the `(k-2)`-cells of `x` and `(sos k x mcst)` are sewn. Otherwise, the function simply returns the recursively obtained map

```
  Fixpoint ng_s_1step :
                  nat → fmap → fmap → fmap → fmap
      := λk:nat. λrest,mcst,mvar:fmap.
         Cases mvar of
           v ⇒ rest
         | (i x mvar') ⇒
             Cases (AND_DEC_3
                 (NOSUCC_DEC k x
                          (ng_s_1step k rest mcst mvar'))
                 (TEST_SYM (EQ_DART_DEC x (sos k x mcst)))
                 (EXD_DEC x mcst)) of
               (left _) ⇒
                   (sm k (ng_s_1step k rest mcst mvar')
                         x (sos k x mcst))
             | (right _) ⇒ (ng_s_1step k rest mcst mvar')
             end
         | (l _ mvar') ⇒ (ng_s_1step k rest mcst mvar')
         end
```

The third function simply applies the previous function at every dimension lower than a value to the duplicateless dart map of the argument map. This allows to browse the map once for each dimension, as browsing only uses the darts of a map. It is defined with a recursive function on the integer representing the dimension.

Definition 52 (*INTERMEDIARY FUNCTION, TREATMENT OF ALL DIMENSIONS*). Let k be a dimension. If k=0, this function yields the duplicateless dart map of m. If k=k'+1, then it applies function ng_s_1step to the map yielded by the recursive call and to m

```
  Fixpoint ng_s_int : nat → fmap → fmap
      := λk:nat. λm:fmap. Cases k of
           0 ⇒ (dartmap m)
         | (k'+1) ⇒ (ng_s_1step k' (ng_s_int k' m) m m)
         end
```

We have just defined a function that, when applied to the dimension and support of a *gmap*, is expected to yield a free map that is equivalent to this *gmap* and also well-constructed at its dimension. We now formally prove this. To do so, we study some properties of these intermediary operations like the behaviour of selectors in their images, and their possible well-constructedness. Although it is not absolutely necessary, we choose to put all those properties in a single lemma. This is motivated by practical reasons. Our goal is to prove properties on recursively defined functions. Obviously, we are going to try and reason by induction on the recursion variable. As we underlined in the description of WCST_WF_PROPS, whenever we use this reasoning style, we must make sure that our induction hypothesis is strong enough. In particular, we should not try and attack elementary and specialized propositions, as the corresponding induction hypothesis may (and probably would) turn out to be itself too specialized and hence too weak. For instance, while analysing `ng_s_1step`, selectors `succ` and `exd` cannot be studied separately, as in this function these selectors evolve in an interdependent way. Such a strategy would inevitably fail. The other issue is that the operations and predicates we study, especially `sm` and `wcst`, are rather complex, and as a consequence it is difficult to anticipate precisely which secondary properties, if any, will be required to prove the properties we are really interested in. Practically, unless the proof is preceded by a difficult and tiresome preparation, these properties can only be identified at the moment they are actually needed. The concrete technique is to save the current proof script, then modify the lemma statement by adding the new secondary property. Then the script can be applied again to the new lemma with some minor modifications (usually at most a dozen new tactics regardless of the script size), and proceed at the point where the proof was previously stopped. The difference is that we have the new property available for the current subgoal, and that a new subgoal has been added, being the one of the new secondary property. The problem with this method is that lemma statements are much longer and unfocused. Moreover, their proof scripts are much larger and thus harder to maintain. Besides, we are not really interested in knowing the properties of these intermediary operations in their full generality, but only when they are used for the conversion of *gmap* to *smap*. This is why we add some extra hypotheses to our lemmas, which reflect some characteristics of the actual set of values that the arguments of the functions will take. The first of these lemmas states the properties of `cleandartmap`.

Lemma 30 (WELL – CONSTRUCTEDNESS OF THE IMAGES OF DARTMAP, BEHAVIOUR OF SELECTORS IN ITS IMAGES). *Any dart belongs to a map iff it belongs to the corresponding duplicateless dart map. No dart has any successors in a duplicateless dart map.*

```
Lemma DARTMAP_PROPS :
  (∀m : fmap) (wcst 0 (dartmap m))
     ∧((∀x,y : dart) (exd x m) ↔ (exd x (dartmap m)))
     ∧((∀k:nat; ∀x:dart) (nosucc k x (dartmap m))).
```

The more complex second lemma deals with `ng_s_int`. It introduces a number of hypotheses on the arguments as was mentioned before. These hypotheses basically state that map `reste` is a submap of `mcst`. The reader should feel free to skip this lemma, as understanding it is quite difficult and honestly not really worth the effort. Just notice that the extra hypotheses come up as extra premises, and also in a more subtle way by defining `rest` and `mcst`, respectively, as an *smap* and as a *gmap*, which implies the well-constructedness (resp. well-formedness) of these maps. The Gallina syntax takes advantage of the *local definition* facility with the construction `[m1step := (n_s_1step reste reste mcst mvar)]` that locally declares symbol `m1step` as an abbreviation of expression `(n_s_1step reste reste mcst mvar)`. Remember that coercions allow to use an *smap* instead of its dimension.

Lemma 31 (WELL – CONSTRUCTEDNESS OF IMAGES OF NG_S_1STEP, BEHAVIOUR OF SELECTORS IN THESE IMAGES).

```
Lemma NG_S_1STEP_PROPS :
   (∀rest:smap; ∀mcst:gmap; ∀mvar:fmap)
      ((∀k : nat) (k+1)=rest
                 → (irreflexive (alpha k rest)))
      → ((∀x : dart) (exd x rest) → (exd x mcst))
      → ((∀x,y:dart; ∀k:nat) (k+1) ⩽rest
          →  ((alpha k x rest y) ↔ (alpha k x mcst y)))
      → (rest+1) ⩽mcst
      → [m1step := (ng_s_1step rest rest mcst mvar)]
         (wcst (rest+1) m1step)
         ∧((∀x : dart)
             (exd x mcst) ↔ (exd x m1step))
         ∧((∀x,y : dart)
             (succ rest x m1step y)
             → (succ rest x mcst y)
               (∃p:path |
                   (pathle3 (rest+1) p)
                   ∧((exd (follow mcst x p) mvar)
                      ∨ (exd (sos rest (follow mcst x p)
                                  mcst) mvar))))
         ∧ ((∀x,y:dart; ∀p:path)
             (pathle3 (rest+1) p)
             → (succ rest x mcst y)
             → (exd (follow mcst x p) mvar)
               ∨ (exd (sos rest (follow mcst x p)
                                  mcst) mvar))
             → x ≠ y
             → (alpha rest m1step x y)
          ∧((∀x,y:dart; ∀k:nat)
               (k+1) ⩽rest
               → (alpha k rest m1step x y)
                 ↔ (alpha k mcst x y)).
```

The third lemma deals with ng_s_int.

Lemma 32 (WELL – CONSTRUCTEDNESS OF IMAGES OF NG_S_INT, BEHAVIOUR OF SELECTORS IN THESE IMAGES). *The image of* gmap m *and natural* k *no greater than the dimension of* m *by function* ng_s_int *is a free map that is well-constructed at dimension* k, *that has the same darts as* m *and the sewings of which are those of* m *that are of dimension lower than* k.

```
Lemma NG_S_INT_PROPS : (∀m:gmap; ∀k:nat)
   k⩽m
   → [mint := (ng_s_int m k)] (wcst k mint)
      ∧((∀x : dart) (exd x mint) ↔ (exd x m))
      ∧((∀x,y:dart; ∀k':nat)
         k'<k → (alpha k' mint x y) ↔ (alpha k' m x y)).
```

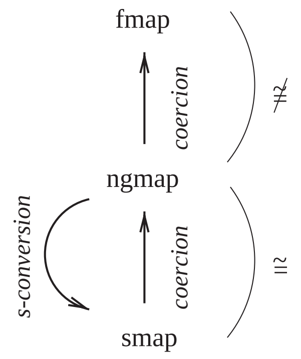


Fig. 9. Links between map types.

We immediately deduce from this the well-constructedness of the images of ng_s _int.

Lemma 33 (WELL – CONSTRUCTEDNESS OF THE IMAGES OF NG_S_INT).

```
Lemma WCST_NG_S_INT : (∀m:gmap; ∀k:nat)
            (wcst k (ng_s_int m k)).
```

This allows the definition of a function that from any *gmap* computes a *smap* that is equivalent to this gmap:

Definition 53 (*CONVERSION OF A GMAP INTO AN EQUIVALENT SMAP*). Let m be *gmap*. This function yields the g-map the support of which is (ng_s_int m m), the dimension of which is m, and the well-construction proof of which is provided by lemma WCST_NG_S_INT

```
Fixpoint ng_s : gmap → smap
   := λm:gmap. (mks (WCST_NG_S_INT m m)).
```

The equivalence of images and arguments of this function is another direct corollary of NG_S_INT_PROPS.

Theorem 34 (*Équivalence of arguments and images of ng_s*).

```
Theorem EQ_NG_S : (∀m : gmap) m ≅ (ng_s m).
```

We call an application of ng_s an *s-conversion*. We have succeeded in proving that for any well-formed map there exists a well-constructed map at its dimension that it is equivalent to, this map being its image by ng_s, all the while exhibiting an algorithm to build it. As we have previously shown that well-constructedness entails well-formedness, we deduce that *smap* is equal to *gmap* modulo ≅. Links between map types are summed up in Fig. 9. As to the properties of these sets, this translates by the fact that any property of *smap* that is stable with respect to ≅ may be extended to *gmap*. This is the corollary that will allow us to reason by induction on the *smap* for such properties, and deduce their validity on *gmap*. Moreover, this is a theoretical result that is interesting by itself, as it had never been proved for generalized maps of any dimension as far as we know. Notably, the preconditions of dart symmetry, that only applies when a cell is sewn to itself, was previously unknown. However, sewing a cell to itself is almost never actually performed, so it can be assumed that this precondition is practically irrelevant in concrete cases. Thus, the main interest of this theoretical result is in fact that there were no overlooked preconditions other than this one. As a consequence, most past uses of the duality of g-map definitions were indeed correct, as they satisfied all preconditions.

7. Conclusion

Finding inspiration in some aspects of the methodology used in [17], we have developed a hierarchical three-level formal specification of generalized maps in the calculus of inductive constructions, a type theory extended with inductive definitions. The top of the hierarchy is the type of unconstrained free maps. Then we have introduced g-maps themselves, which are free maps that are well-formed with respect to geometric modelling requirements. Sewn-cells maps are special generalized maps, exclusively built with two heavily constrained operations of dart insertion and cell sewing that give her an incremental structure, whereas g-maps have a synthetical structure. G-maps and s-maps are actually the formalizations of two different usual mathematical definitions of the generalized maps. With the help of proof CIC-based assistant Coq we have shown to some degree the soundness and completeness of our axiomatics. In particular, by explicitly building s-conversion, i.e. conversion between s-maps and g-maps modulo reordering of darts and sewings insertions, we have proved that the set of generalized maps was equivalent to the one of sewn-cells maps. Building this proof has uncovered a previously unknown and unexpected precondition to cell sewing.

The specification techniques we developed and used are quite general, and could be used in any field where a same kind of objects has two distinct representations (here $gmap$ and $smap$, modulo $\cong$) that can themselves both be expressed in a more elementary framework (here $fmap$).

In the second part of the series, we will show how we used this specification to prove the first half of a fundamental theorem of geometry, the theorem of classification of surfaces according to numerical characteristics.

References

[1] B. Barras et al., The Coq Proof Assistant Reference Manual, http://coq.inria.fr/doc/main.html.

[2] Y. Bertrand, J.-F. Dufourd, Algebraic specification of a 3D-modeler based on hypermaps, Graphical Models Image Process. 56 (1) (1994) 29–60.

[3] G. Bertrand, R. Malgouyres, Some topological properties of surfaces in Z^3, J. Math. Imaging Vision 11 (1999) 207–221.

[4] T. Coquand, G. Huet, Constructions: a higher order proof system for mechanizing mathematics, EUROCAL, in: Lecture Notes in Computer Science, Vol. 203, Springer, Berlin, 1985.

[5] R. Cori, Un Code pour les Graphes Planaires et ses Applications, Société Math. de France, Astérisque 27 (1970).

[6] C. Dehlinger, J.-F. Dufourd, P. Schreck, Higher-Order Intuitionistic Formalization and Proofs in Hilbert's Elementary Geometry, Automated Deduction in Geometry, in: Lecture Notes in Artificial Intelligence, Vol. 2061, Springer, Berlin, 2000.
[7] P.A. Firby, C.F. Gardiner, Surface Topology, Ellis Horwood Ltd., Chichester, UK, 1982.
[8] A.T. Fomenko, Differential Geometry and Topology, Consultant Associates, 1987.
[9] H. Griffiths, Surfaces, Cambridge University Press, Cambridge, 1981.
[10] A. Jacques, Constellations et graphes topologiques, Combin. Theory Appl. 1970, 657–673.
[11] G. Kahn, Elements of constructive geometry, group theory, and domain theory, Coq contribution, http://coq.inria.fr/contribs-eng.html.
[12] D.E. Knuth, Axioms and Hulls, in: Lecture Notes in Computer Science, Vol. 606, Springer, Berlin, 1992.
[13] P. Lienhardt, Subdivisions of N-dimensional spaces and N-dimensional generalized maps, Computational Geometry ACM Symp., 1989, pp. 228–236.
[14] P. Martin-Löf, Intuitionistic Type Theory, Bibliopolis, Napels, 1984.
[15] C. Parent, Synthesizing proofs from programs in the calculus of inductive constructions, Mathematics of Program Construction, in: Lecture Notes in Computer Science, Vol. 947, Springer, Berlin, 1995.
[16] D. Pichardie, Y. Bertot, Formalizing Convex Hulls Algorithms. TPHOL, Lecture Notes in Computer Science, Vol. 2152, Springer, Berlin, 2001, pp. 346–361.
[17] F. Puitg, J.-F. Dufourd, Formal Specifications and Theorem Proving Breakthroughs in Geometric Modelling, TPHOL, Lecture Notes in Computer Science, Vol. 1479, Springer, Berlin, 1998, pp. 401–427.
[19] J. Von Plato, The axioms of constructive geometry, Ann. Pure Appl. Logic 76 (1995) 169–200.