Formal Study of Functional Orbits in Finite Domains

Jean-François Dufourd

University of Strasbourg, CNRS, ICube
Pôle Technologique, Boulevard S. Brant,
BP10413, 67412 Illkirch, France

Abstract

In computer science, functional orbits − i.e tracks left by iterating a function − in a finite domain naturally appear at a high or a low level. This paper introduces a Coq logical orbit framework, the purpose of which is to help rigorously developing software systems with some complex data structures from specification to implementation.

The result is a Coq library of orbit concepts formalized as definitions, lemmas and theorems. Most of them are inspired by our previous work in geometric modelling, where combinatorial hypermaps were used to describe surface subdivisions appearing in computational geometry problems, e.g. building convex hulls or Delaunay diagrams. Now, this domain remains a reference for us, but our results are drastically extended and usable in other computer science areas. The proof of Floyd's cycle-detection algorithm, known as "the tortoise and the hare", confirms that point.

The library contains operations to observe, traverse and update orbits − addition, deletion, mutation, transposition − with proofs of their behavior. It focuses on the important case where the involved function is a partial injection. In this case, it defines a connectivity relationship and evaluates the variation of the number of connected components during updates.

Keywords: formal specification, functional orbits, program correctness, linked representation, algebraic data type, memory shape analysis, computer-aided proof, Coq system
2010 MSC: 68N30, 68P05, 68Q60, 68Q65, 03B35

1. Introduction

We present a formal study of the (functional) orbit notion, its properties and operations, using the Coq proof assistant [26, 4]. The goal of this work is to give

Email address: jfd@unistra.fr (Jean-François Dufourd)

a sound basis for a general concept which can help to correctly and completely specify and build software systems with some complex data structures.

Our background is a rich application domain consisting in the specification and implementation of computer libraries to interactively build and manipulate geometric objects [5, 9, 19, 12, 13, 18, 7]. Such libraries − e.g. CGAL [37], Topofil [5] or CGoGN [20] in the academic world, and CATIA [36] in the industrial world −, are vital in design, architecture, animation, mechanics or robotics.

Everybody knows the orbit notion used in physics to model the gravitationally curved path during the motion of a planet or the wave-like behavior of an electron in an atom. In mathematics, the orbit of an element z denotes the set of positions which may be taken by the images of z under the action of a group G, the orbits forming a partition of G's support.

In this paper, the definition we retain is quite different, even if it has some similar consequences. Let X be any space, $f : X \to X$ any total function defined on X, $D \subseteq X$ any (sub)domain of X, and $z \in X$ any element of X. Roughly speaking, the (first uninterrupted) track in D of the iterates $z_0 = z$, $z_1 = f\ z_0$, ..., $z_k = f\ z_{k-1}$, ..., is called the *f-orbit* of z in D.

It is unnecessary to recall the fundamental role of iterations in all mathematics and computer science areas. However, to our knowledge, the tracks left by them, i.e. the orbits, have never been studied as a universe of "first-class objects" which may be constructed or modified by operations like addition, deletion, mutation and transposition. For instance, in interactive geometric modelling, such orbit operations occur when manipulating high-level mathematical objects − e.g. space subdivisions, polygons and polyhedra − at the specification level, or low-level programming objects − e.g. singly- or doubly-linked lists representing geometric objects − at the implementation level.

This article only examines the case where D is *finite*, which is significant in computer science where the considered sets and iterations are often finite. Moreover, even if the starting point is clear, the correct and complete definition of orbits, operations and related notions is far from being immediate, especially because orbit operations can involve D or f, and because a tiny modification of one of them can have very large effects in the orbit universe.

The orbit notion was explicitly mentioned by several authors at conceptual level, for combinatoric aims (e.g. Berge [3]), for dynamic systems studies such as fractal investigations (e.g. Mendelbrot [32] and Barnsley-Demko [1]), and for geometric model studies (e.g. Tutte [38] and Lienhardt [38]). We have ourselves a rather good experience of formalization and use of orbits in geometric modelling [19, 12, 13, 14, 18, 7]. However, we made an effort to extract this notion from our previous work and present it as a whole for new uses.

The orbit notion has been approached in different ways at a low level to model memory sharing and aliasing, mainly for proofs of programs with pointers using Hoare logic [25] and its extensions. A pioneer work introducing linked list segments, a non-repetition predicate, separated list systems, and rules to deal with them, is due to Burstall [8]. Initiated by Reynolds [34], *separation logic* is intensively studied for upgrading the problems of memory sharing at a logical level, and avoiding to prove that a modification in a part of the memory does

not affect other parts. In fact, proofs involving various data structures − e.g. arrays, linear and cyclic data structures, possibly nested − can be achieved with our orbit results. A combinatorial map example is introduced hereafter with hierarchized data structures and a full hypermap application was developed in our recent work [16]. Classical sequence problems can also be formalized and solved using orbits. As a significant illustration, we show how the total correctness of Floyd's circuit-finding algorithm [28], also known as "the tortoise and the hare" algorithm, can be obtained from orbit features.

However, nobody seems to have considered the orbit notion as we want to do it. Our aim is to give rigorous definitions and to state and prove numerous properties, not only in (informal) mathematical style, but also in a formal language which can be handled interactively by some proof assistant. To be as generic as possible, our choice is to use an intuitionistic higher-level typed logic, or lambda-calculus, namely the Calculus of Inductive Constructions, to formalize our concepts, and the Coq system [26, 4], which is an implemention of this calculus, to guide and support our interactive proofs. Nothing is added to the calculus, except the *axiom of extensionality*. This axiom states that two functions are equal if they are equal in every point. Finally, this paper also serves as introduction to a Coq library of specifications and proofs which can be reused each time the orbit notion is involved.

The rest of the paper is organized as follows. In Section 2, we mathematically define the notion of orbit and related concepts, and give some properties. In Section 3, we develop an example in geometric modelling where orbits are central at high and low levels. In Section 4, we formalize in Coq these notions and specialize them to prove static properties. In Section 5, we formalize and prove Floyd's circuit detection algorithm. In Section 6, we particularize the results in the important case where f is a partial injection. In Section 7, we present operations to add and remove an element of D, and their effects on the orbits. In Section 8, we study a mutation operation to change the image of an element of f, and its repercussion on the orbits. In Section 9, we examine a transposition operation to split or merge orbits which are circuits. In Section 10, we define a relation of connectivity for partial injections and study the effect of the previous operations on the number of connected components. We discuss related works in Section 11 and we conclude in Section 12.

The main features of the Coq language and system used in our specifications and proofs are reminded. The whole formalization process is described and explained, but the full details of the proofs are out of the scope of this article. However, the complete Coq (Version 8.4pl2) development is available [15].

2. Mathematical definitions and properties

In this section, we mathematically define and investigate the basic orbit notions and properties. They are formalized in Coq in the following, but we prefer to introduce them more intuitively with usual mathematical notations.

Let X be any space where the equality $=$ is decidable, $f : X \to X$ be any (*total*) function defined on X, $D \subseteq X$ be any *finite (sub)domain* of X, and

$z \in X$ be *any element* of X. Let $z_k = f^k\ z$ be the k-th *iterate* of z by f, for $k \geq 0$ (with $z_0 = z$). We list sequences in brackets, [and], with [ ] for the empty sequence, and we write $In\ z\ l$, or $z \in l$ for readability, when z occurs in the sequence (or list) l of X's elements.

Since D is finite, during the iteration process from z, there is a time when the current iterate is outside D − possibly at the beginning − or encounters an iterate already met. This is reflected in the following definition:

Definition 1. (*Orbital sequence, length, orbit, limit, top*)
(i) The *f-orbital sequence* of z at the order $k \geq 0$, denoted by $orbs_f\ k\ z$, is [ ] if $k = 0$, and $[z_0, z_1, \ldots, z_{k-1}]$ otherwise.
(ii) The *length of the f-orbit* of z in D, denoted by $lorb_{f,D}\ z$, is the smallest integer p such that $z_p \notin D$ or $z_p \in orbs_f\ p\ z$.
(iii) The *f-orbit* of z in D is $orbs_f\ (lorb_{f,D}\ z)\ z$, more briefly denoted by $orb_{f,D}\ z$.
(iv) The *f-limit* of z with respect to D, denoted by $lim_{f,D}\ z$, is $z_{lorb_{f,D}}$.
(v) For $z \in D$, the *f-top* of z in D, denoted by $top_{f,D}z$, is $z_{lorb_{f,D}-1}$.

When the context is clear, the f- prefix is removed from the previous names. Fig. 1 illustrates these notions and the various shapes of orbits, for $z \in D$ or not, according to both stopping conditions, i.e. $z_p \notin D$ and $z_p \in orbs_f\ p\ z$, which are clearly exclusive. The following properties are direct consequences:

Lemma 1. *(*Orbit properties*)*
(a) If $z \notin D$, $lorb_{f,D}\ z = 0$ and $orb_{f,D}\ z = [\ ]$.
(b) All the elements of an orbit belong to D and are distinct.
(c) If $z \in D$, $0 < lorb_{f,D}\ z$, $orb_{f,D}\ z \neq [\ ]$, and $z_k \in orb_{f,D}\ z$ for all $0 \leq k < lorb_{f,D}\ z$.

It is manifest that an orbit is a bounded list *without repetition*, which can be viewed as a finite set if it is more convenient. Note that it is possible that $z_k \in D$ for $lorb_{f,D}\ z < k$, as f is total in X. The orbit *shapes* can be classified as follows regarding the conditions of the above lemma (Fig. 1):

Definition 2. (*Shape, Line, Crosier, Circuit*)
(i) When $z \notin D$, the *shape* of z's orbit is *empty*.
(ii) When $lim_{f,D}\ z \notin D$, the *shape* of z's orbit is a *line*.
(iii) When $lim_{f,D}\ z \in orb_{f,D}\ z$, the *shape* of z's orbit is a (closed) *crosier*.
(iv) The *length* of the *handle* of z's orbit, denoted by $lhand_{f,D}\ z$, is the smallest integer q such that $z_q = z_{lorb_{f,D}\ z}$. In the particular case where $lhand_{f,D}\ z = 0$, the (crosier) orbit of z is called a *circuit*.

For instance, for the line, crosier and circuit in Fig. 1, we have $lorb_{f,D}\ z = 5, 7$ and 6, respectively. For the crosier, we have $lhand_{f,D}\ z = 3$, whereas for its ending circuit, we have $lorb_{f,D}\ z_p = 4$. Note that, with our definitions, an empty orbit is a line and a circuit as well, and when $lorb_{f,D}\ z = lhand_{f,D}\ z$, z's orbit is a line, and conversely.

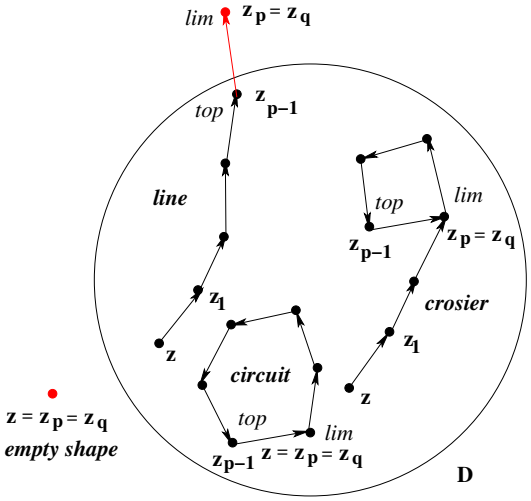


Figure 1: Shapes of orbits (for different positions of z).

Of course, if we consider the orbits of all of D's elements, we find that they are either separated or in collision. In fact, (D, f) forms a *functional graph* (Berge [3]) for which we can examine the *connected components*. Each of them is a *tree* or is composed of a *unique circuit* on which trees are grafted. Fig. 2 illustrates the various shapes of components for f. Applied to memory modelling, the connected components correspond to *list systems* in the meaning of Burstall [8].

When $orb_{f,D}\ z$ is a circuit, we have $lim_{f,D}\ z = z$, and, for $z \in D$, $top_{f,D}\ z = f^{-1}_{f,D}\ z$, where $f^{-1}_{f,D}\ z$ − or simply $f^{-1}\ z$ in a clear context − denotes the *inverse* of f at z. In fact, the "inversion" of f, along with the change of the orbital traversal direction, requires some precautions since f is any function in X.

So, for the time being, we only consider that f has an *inverse* $f^{-1}_{f,D}\ z$ at z if z's orbit is a *circuit*. Then, by definition, $f^{-1}_{f,D}\ z = f^{lorb_{f,D}-1}\ z$ when $z \in D$, and, if in addition $(f\ z) \in D$, we obtain (with our short notations) $f^{-1}\ (f\ z) = f\ (f^{-1}\ z) = z$ as expected. We will see later (Section 6) how specializing f to a *partial injection* leads to a more general inversion.

3. An example in geometric modelling

We give an example in geometric modelling involving orbits at two levels to represent surface subdivisions.

3.1. Subdivision

To manipulate on a computer parts of a − say tri-dimensional − geometric object, it is convenient to subdivide it into a *mesh of k-dimensional cells*, with

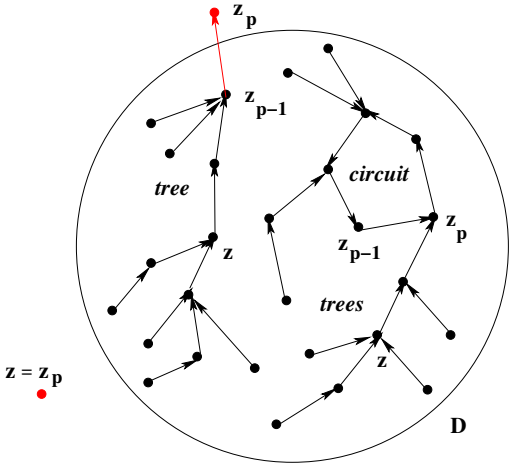


Figure 2: Shapes of components.

$k <= 3$: 0-cells, 1-cells, 2-cells and 3-cells are respectively called *vertices, edges, faces* and *volumes*. To easily manage the mesh, it is necessary to describe, not only the cells, but also the *incidence* and *adjacency* relationships between them, everything constituting the *combinatorial topology* of the subdivision.

Once the topology is known, the cells must be *embedded* on a true geometric space, say Euclidean: vertices, edges, faces and volumes are embedded on *points*, *Jordan arcs*, *surface patches* and *solids*, respectively. Then, a geometric library must offer operations allowing to build, update, visualize, traverse and query such embedded subdivisions [37, 20, 36].

In what follows, we only deal with the combinatorial topology of 2D usual objects, i.e. *surface* subdivisions [22]. Surfaces are *orientable* − e.g. the plane, the disc, the sphere, the torus −, or *non-orientable* − e.g. the Mœbius rubber, the Klein bottle. They are *open*, i.e. with *borders* − e.g. the disc, the Mœbius rubber −, or *closed*, i.e. without borders − e.g. the plane, the sphere, the torus, the Klein bottle. We restrict our example to *orientable* and *closed* surfaces which simplify the presentation without loss of generality about orbits.

3.2. Combinatorial map

The topology of an orientable closed surface subdivision can nicely be described by a *combinatorial map*. This is an algebraic structure involving only abstract objects called *darts* and operations between them (cf. Tutte [38]).

Definition 3. (*Combinatorial map*)
A *(combinatorial) map* of dimension 2 is an algebraic structure $M = (D, \alpha, \pi)$, where D is a finite set, the elements of which are called *darts*, α is an *involution* on D and π is a *permutation* on D.

D	1	2	3	4	5	6	7	8	9	10	11	12	13	14
α	2	1	4	3	6	5	8	7	10	9	11	12	14	13
π	8	3	2	9	4	10	6	1	5	7	11	13	12	14

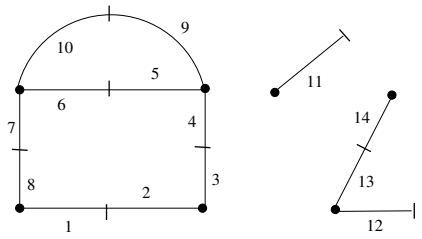


Figure 3: A combinatorial map embedded on the plane.

Fig. 3 presents a combinatorial map, with $D = \{1, ..., 14\}$ and α, π given by a table. In the drawing, the map is embedded on the plane, its darts being embedded on half-Jordan arcs (labelled by the darts) which are oriented from a bullet to a small transversal stroke. If z is any dart, $\pi\ z$ shares the bullet of z, and $\alpha\ z$ is the dart the embedding of which is the opposite to that of z with respect to its stroke. So, $\alpha\ 1 = 2$, $\alpha\ 2 = 1$, $\pi\ 4 = 9$, $\pi\ 9 = 5$, $\pi\ 5 = 4$, etc.

Now, the (topological) cells of the subdivision can be rigorously defined by the orbit notion, if we consider that *dart* is the (non-finite) space of darts with $X = dart$ and $D \subseteq dart$ in our orbit definition (Def. 1).

Definition 4. (*Orbits, connected components*)
Let $M = (D, \alpha, \pi)$ be a combinatorial map and $z \in D$, any dart in M.
(i) The *edge* of z is the α-orbit of z in D.
(ii) The *vertex* of z is the π-orbit of z in D.
(iii) The *face* of z is the ϕ-orbit of z in D, where $\phi = \pi^{-1} \circ \alpha^{-1}$.
(iv) The *connected component* of z is the combinatorial map $M' = (D', \alpha', \pi')$, where D' is the subset of the darts of D which are accessible from z by any composition of α and π, and $\alpha' = \alpha \mid D'$, $\pi' = \pi \mid D'$.

Thus, in Fig. 3, the edge of 2 is $[2, 1]$, the vertex of 4 is $[4, 9, 5]$, $\phi\ 1 = 3$, $\phi\ 3 = 5$, $\phi\ 5 = 7$, $\phi\ 7 = 1$, the face of 1 is $[1, 3, 5, 7]$, and the connected component of 2 has $\{1, ..., 10\}$ as a support.

Since α, π and ϕ are bijections on D, for any $z \in D$, the orbits of z for them in D are *circuits*. It is the same for their inverses, α^{-1}, π^{-1} and ϕ^{-1}. So, in this cyclic case, if t is in z's f-orbit then t's f-orbit and z's f-orbit may be identified into a *single circuit*, defined *modulo* a cyclic permutation. With this convention, the common orbit only counts for 1 in the number of orbits by f. In fact, these cyclic orbits modulo f correspond exactly to the *connected components* generated by f in D.

In this case, cells and components (with respect to $\{\alpha, \pi\}$) can be counted to classify maps according to their *Euler characteristic*, *genus* and *planarity*. Let

M be a combinatorial map. We name d, v, e, f and c its numbers of darts, vertices, edges, faces and connected components, respectively.

Definition 5. (*Euler characteristic, genus, planarity*)
(i) The *Euler characteristic* of M is $\chi = v + e + f - d$.
(ii) The *genus* of M is $g = c - \chi/2$.
(iii) The map is said to be *planar* if $g = 0$.

Thus, in our map example (Fig. 3), $d = 14$, $v = 7$, $e = 8$, $f = 5$, $c = 3$, $\chi = 6$ and $g = 0$. Consequently, the map is planar. An important theorem precises the possible values of χ and g:

Theorem 1. *(*of the genus*) For any combinatorial map M,*
(a) χ is an even relative number.
(b) g is a natural number.

So, χ is always even but can be negative, while g is always non-negative and possibly zero. Genus and planarity are related with the embedding on surfaces. The combinatorial topology theory shows that a map (say non-empty, connected, with no fixpoints for α and π, to simplify) with genus g can be embedded (without self-intersection) on an *orientable closed* surface with g *tunnels*. For instance, a map of genus 0 can be embedded on a sphere or on a plane (with 0 tunnel), and a map of genus 1 can be embedded on a torus (with 1 tunnel), but not on a sphere or a plane. Note that the number of tunnels of a surface is also named its genus. There is a similitude between the genus of a surface and the genus of any of its subdivisions, modelled by a combinatorial map [22, 13].

Consequently, it is of primary importance to know how to evaluate the number of connected components for any f and D, and to appreciate the influence that update operations have on it. In the following sections, we show how to formalize in Coq the counting of components and its evolution when f remains a *partial injection*, which encompasses the case of the combinatorial maps.

3.3. Combinatorial map linked representation

In geometric modelling, combinatorial maps can be represented in different ways: by separating connected components in linked blocks, by grouping α-opposite darts in a same record, by using singly- or doubly-linked lists, by introducing names for the darts, and so on. Only theoretical and experimental time and space complexity studies taking into account the frequency of some preselected map operations can justify the preferences.

Here, we retain the choices which were made in the Topofil library [5], and today in the CGoGN library, which is developed in our laboratory [20]. They favor map *topological updates* − e.g. addition/deletion of darts, merging/splitting and glueing/unglueing of cells −, which remain in constant space and time in the worst case. They do not excessively penalize *topological observations* − e.g. dart searching, orbit or component traversal − which remain in constant space and in linear time with respect to d (the dart number) in the worst case.

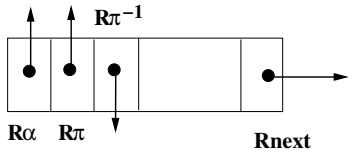


Figure 4: A memory dart record.

In these libraries, the implementation of a map in a *memory* (or storage) S mainly consists of a *singly-linked list of dart records*. A dart z is then a *pointer* on a dart record which fields are: the dart pointer to the next dart record in the list, $\alpha\ z$, $\pi\ z$, and the inverse $\pi^{-1}\ z$, in order to traverse vertices in reverse order (Fig. 4). Since $\alpha\ z = \alpha^{-1}\ z$, an inverse pointer for α is useless. Other dart record fields concern embeddings and are ignored in this presentation.

So, a map is represented by a pair (S, h), where h is the head pointer on the main singly-linked list, *null* when the list is empty. We define operations between addresses. Let z be any dart pointer, we write $Rnext\ S\ z$ for the successor of z in the general singly-linked list, and respectively $R\alpha\ S\ z$, $R\pi\ S\ z$, $R\pi^{-1}\ S\ z$ for the successors of z by α, π, π^{-1} (cf. Fig. 4). Function $R\phi$ is not stored, but just defined by composition of $R\alpha$ and $R\pi^{-1}$. The connected component of z is obtained by a graph traversal algorithm. It is nice to note that the Schorr-Waite marking algorithm [35] for bi-functional graphs exactly addresses this issue and is usable to compute our connected components. Besides, its correctness has been proved using orbits [17].

In fact, the above structure is too *loose* to always fit with maps. Some constraints must be added through an *invariant of representation*, which must ensure that α is an *involution*, π and π^{-1} are *inverse permutations* on darts *of the map*. Hence, $R\alpha\ S\ z$ must point to a *cyclic singly-linked list* with only 1 or 2 elements, and $R\pi\ S\ z$ and $R\pi^{-1}\ S\ z$ must point to elements of a *cyclic doubly-linked list* representing a cyclic dart orbit, while being sure that these lists never go out the main singly-linked list.

The *integrity* of this invariant must absolutely be maintained by update operations. Due to a lack of space, these operations are presented only in a subsequent paper [16]. Here, the point is that the invariant may be expressed by requiring some properties for the previous functions, particularly about their orbit shapes. We denote A the (finite) set of *active addresses* of S, i.e. the addresses which point to an allocated memory cell. Our orbits being possibly viewed as sets, the invariant is composed of the following constraints:

- (i) $D = orb_{Rnext\ S,A}\ h$ is a *line* with $lim_{Rnext\ S,A}\ h = null$
- (ii) For all z in D,
  - (a) $orb_{R\alpha\ S,D}\ z$ is a *circuit* and $R\alpha\ S\ (R\alpha\ S\ z) = z$
  - (b) $orb_{R\pi\ S,D}\ z$ is a *circuit* and $R\pi^{-1}\ S\ z = f^{-1}_{R\pi\ S,D}\ z$

Of course, update operations on map representations modify memory, structures, pointers and functions, and have to master the orbit characteristics which are mentioned in the invariant. Finally, it will be crucial to establish the correspondence between maps, at the conceptual level, and map representations, at the implementation level. This is the heart of [16].

In the proofs of programs community, the orbit notion was approached by several authors, under different names, like *list segment* [8, 34], *f-linked sequence* [6], *non-repetiting path* [31] or *tracked location* [23]. So, it seemed to us that a specific study of the functional orbits was essential to abstract and unify this domain.

4. Formalization of orbits

4.1. General features

In Coq [26, 4], `Prop` is the type of propositions (with two constants: `True`, the proposition which is always satisfied, and `False`, the proposition which is never satisfied) and `Type` can be viewed as the *type of types*. In the Standard Coq library, `nat` is the type of natural numbers, with the constructors `0` and `S`, the succession function, and the usual arithmetic operations `+` and `*`.

For any type `A:Type`, `list A` is the type of finite lists with elements of type `A`. It is equipped with usual operations: `[]` denotes the *empty* list, `::` the *front insertion*, `++` the *append* operation − or *concatenation*, and, for any `a:A` and `l:list A`, `In a l` denotes the membership of `a` to `l`. In the following, a finite list *without repetition* can be considered as a finite set with all its properties. So, we defined list operations which are similar to set operations: for any finite lists `l` and `l'`, `cardl l` is the *cardinal*, or number of elements, of `l`, and `minusl l l'` is the *complement* of `l'` *in* `l`.

A Coq formalization is divided in Coq *sections* corresponding to *contexts* and containing declarations of *variables*. Inside a context, all the introduced notions are implicitly parameterized by the variables. Outside the context, these notions can be reused provided that they are applied to *actual parameters*.

Our first Coq *section* declares three *variables*: (1) `X:Type` is any type equipped with a *decidable equality* predicate written `eqd X:X->X->Prop`; (2) `f:X -> X` is any total function on `X`; (3) `D:list X` is any bounded list of `X`'s elements. We always ensure that `D` is *without repetition*. Let us give an example of parameterization: for any `z:X`, the section will define `orb z` as the `f`-orbit of `z` in `D`. Outside the section, for any `X':Type,f':X'->X',D':list X,z':D`, the `f'`-orbit of `z'` in `D'` has to be denoted by `orb X' f' D' z'`.

Let `Iter` be the *generic function* such that, for any function `g:X -> X`, `Iter g k z` returns the `k`-th iterate by `g` of `z`. The notion of `f`-*orbital sequence* at the order `k`, denoted by `orbs f k z`, easily derives. For convenience, it is presented in the *reverse order* compared to the mathematical definition (Def. 1). Consequently, when `orbs f k z` is not `[]`, we have `last (orbs f k z) = z` and `hd (orbs f k z) = Iter f (k - 1) z`, with `hd` and `last` denoting in Coq, as usual, the first and last elements of a (non-empty) list.

Then we write out that `continue k z` is the *continuation* predicate of the iteration in an orbit. In Coq, the keyword `Definition` introduces the declaration of a non-recursive `function`, `~` denotes the negation, $\wedge$ the conjunction, and `let...in...` the classical functional construction:

```
Definition continue(k:nat)(z:X): Prop :=
  let zk:= Iter f k z in In zk D /\ ~ In zk (orbs f k z).
```

This predicate is easily proved *decidable*, i.e. usable as a Boolean test function which we name `continue_dec`. Then, the central definition of the orbit length, `lorb`, can be obtained from an auxiliary function called `lorb_aux`, which is *recursive* on `k`:

```
Function lorb_aux(z:X)(k:nat)
{measure (fun k:nat => cardl (minusl D (orbs f k z))) k}: nat :=
  if continue_dec k z then lorb_aux z (S k) else k.
```

In this writing, the keyword `Function` introduces the declaration of a *general recursive* function. It imposes to define "on the fly" a natural `measure` function of `k` which decreases at each recursive call, and to apply it to argument `k`. The chosen measure is the *cardinal* of $D \setminus (orbs_f\ k\ z)$, when sequences without repetition are viewed as finite sets. At each recursive call, `k` increases by one unit, and, in order to be accepted by Coq, the definition by `Function` asks to prove that the cardinal is strictly decreasing (it is by 1 unit exactly). This is deduced from a lot of small properties of the orbital sequences which are previously obtained as lemmas. This entails the *finite termination* of `lorb_aux`. At the end, the function returns the last value of `k`.

A definition by `Function` automatically generates a *reasoning principle* by *Nœtherian induction*, which can be used to prove properties of the function by case analysis. Some cases use an *induction hypothesis* on the internal function calls with parameters having a measure strictly smaller than the measure which is computed from the formal parameters. This kind of reasoning is often at the basis of the results which we present in the next paragraphs.

The definition of `lorb` takes the initial value `0` for `k`. The definitions `orb` and `lim` of the *orbit* and *limit* notions follow:

```
Definition lorb(z:nat) := lorb_aux z 0.
Definition orb(z:nat) := orbs f (lorb z) z.
Definition lim(z:nat) := Iter f (lorb z) z.
```

Now, we precise the orbit shape notions.

4.2. Lines

When its orbit is a *line*, `z` satisfies the invariant `inv_line` defined by:

```
Definition inv_line(z:X): Prop := ~ In (lim z) D.
```

It is easy to prove that all of `z`'s orbit elements also satisfy the invariant, and find their respective orbit lengths. These properties are expressed in two *theorems* (introduced by the keyword `Theorem`) called `inv_line_Iter` and `line_lorb_Iter` (Note that types of variables are synthesized):

```
Theorem inv_line_Iter: forall z k,
  inv_line z -> k < lorb z -> inv_line (Iter f k z).
Theorem line_lorb_Iter: forall z k,
  inv_line z -> k < lorb z -> lorb z = lorb (Iter f k z) + k.
```

In these statements, the functional symbol `->` also denotes the logical implication, in accordance with the intuitionistic logic *Curry-Howard correspondence*, which roughly states: "*function = proof*" [4].

In the *line* case, the definition of `lorb` can be rewritten to avoid at each step a recalculation of all the previous iterates, i.e. to lower the complexity. That is the role of the recursive function `len`:

```
Function len (z:X)(H:inv_line z)
 {measure (fun z: X => lorb z) z}: nat :=
  match In_dec z D with
     left H0 => S (len (f z) (inv_line_f z H H0))
   | right _ => 0
  end.
```

In this declaration, the measure is built with `lorb`. It is proved decreasing at each recursive call of `len`, thanks to the `line_lorb_Iter` theorem, but `len` has an additional argument, namely a proof `H` that `orb z` is a line. The role of `inv_line_f` is to rebuild a proof of `inv_line (f z)` for the recursive call.

The definition of `len` is given by a *matching* on `In_dec z D` which decides if `z` is in `D` or not. If this holds (case `left H0`), the result is recursively given by `S (len (f z) (inv_line_f z H H0))`, otherwise (case `right _`) the result is `0`. The proof of each case of the decision is written after `left` or `right` and can be reused in subsequent expressions. Only the proof of the `left` case is useful in the following. It is named `H0` while the other proof, after `right`, remains anonymous.

In fact, the proof argument and the measure are present just for proving that `len` is *finitely terminating*. They disappear in an OCaml *automatic extraction* [4] and in a concrete imperative program as well. Indeed, the extracted OCaml recursive function (introduced by the keywords `let rec`, capitals `X`, `D` being rewritten into lowercases `x`, `d`) is:

```
let rec len x f d z =
  if in_dec x z d then S (len x f d (f z)) else O
```

As usual, by managing a *counter* in the parameters, this *non-tail recursive* form can be transformed in favor of a *tail recursive* one, which is derivable into an imperative iterative program. Moreover, the test `in_dec x z d` can be avoided in most particular cases, e.g. replaced by `z = null` which leads to a best program.

Finally, it is easy to prove by Nœtherian induction on `lorb_aux` that `len z H = lorb z`, for any `H: inv_line z`. This example shows that a naive definition can serve as "bootstrapping" to build a more operative definition and prove that it is correct. Other such transformations are presented in the following.

4.3. Circuits

When its orbit is a *circuit*, `z` satisfies the invariant `inv_circuit` defined by:

```
Definition inv_circuit z := lim z = z.
```

Once again, it is easy to prove that all of `z`'s orbit elements satisfy this invariant, and that `lorb` is unchanged along the orbit:

```
Theorem inv_circuit_Iter: forall z k,
  In z D -> inv_circuit z ->
    inv_cycle (Iter f k z).
Theorem circuit_lorb_Iter: forall z k,
  In z D -> inv_circuit z ->
    lorb z = lorb (Iter f k z).
```

Note that this result is given for `z` in `D`. Once again, `lorb` could be rewritten in a more efficient function with a tail recursive form [15].

4.4. Crosiers

When its orbit is a *crosier*, `z` satisfies the invariant `inv_crosier` defined by:

```
Definition inv_crosier z := In (lim z) (orb z).
```

The proofs that a *circuit* satisfies this invariant and that all of `z`'s orbit elements also satisfy it are straightforward. Moreover, if `z`'s orbit is a crosier, `z` is in `D`, which entails `0 < lorb z`. Conversely, if all of `z`'s iterates are in `D`, `orb z` is a (non-empty) crosier.

4.5. Handles

For any `z`, the *handle length*, denoted by `lhand z`, of `z`'s orbit can be recursively defined by:

```
Function lhand (z: X)
 {measure (fun z: X => lorb z) z}: nat :=
  if In_dec z D
  then if eqd X (lim z) z then 0
       else S (lhand (f z))
  else 0.
```

The strategy is to follow the orbit until `z` leaves `D` or `lim z` is reached.

This is a very naive strategy implying an unefficient direct programming. Fortunately, Section 5 will greatly improve the situation when `z`'s orbit is a crosier. An important result gives the relationship between `lorb z` and `lhand z` for any `z`:

```
Theorem lhand_lorb: forall z,
 let zh := Iter f (lhand z) z in
   lorb z = lhand z + lorb zh.
```

In other words, `z`'s orbit length is `lhand z` plus the length of the ending circuit if any, `0` otherwise. So, it is easily proved that `z`'s orbit is a *line iff* `lhand z = lorb z`, and a *circuit iff* `lhand z = 0`.

4.6. Paths

Stating that `t` is in `z`'s orbit is the same as stating that there exists an *orbital path* (i.e. a path in an orbit) from `z` to `t`, which is expressed by the following binary *existential* predicate `expo`:

```
Definition expo (z t:X):Prop :=
  exists i:nat, i < lorb z /\ Iter f i z = t.
```

Indeed, we have the following equivalence:

```
Theorem expo_eq_In_orb:forall z t,
    expo z t <-> In t (orb z).
```

The proofs that `expo` is *decidable*, *reflexive* and *transitive* are direct. This relation is *symmetric* only if the orbit of the starting element, `z`, is a *circuit*. Moreover, a ternary predicate `betw` expresses that three elements, `z`, `t` and `u`, lie *in this order* in `z`'s orbit:

```
Definition betw (z t u:X): Prop:=
  exists i:nat, exists j: nat,
    i <= j < lorb z /\ Iter f i z = t /\ Iter f j z = u.
```

This *decidable* predicate enjoys − on lines, circuits and crosiers − expected properties [15], which can be compared with those of the usual geometric *orientation predicates* on line segments and circles (cf. Hilbert [24]), e.g. (`<->` is the equivalence symbol):

```
Theorem betw_expo_refl_1: forall z t,
  betw z z t <-> expo z t.
Theorem expo_betw_or: forall z t u,
  expo z t -> expo z u -> (betw z t u \/ betw z u t).
Theorem inv_line_expo_rev: forall z t, inv_line z ->
  expo z t -> expo t z -> z = t.
Theorem expo_betw_3:forall(x y z:X), inv_cycle x ->
  expo x y -> expo x z -> betw x z y \/ betw (f y) z (f_1 x).
```

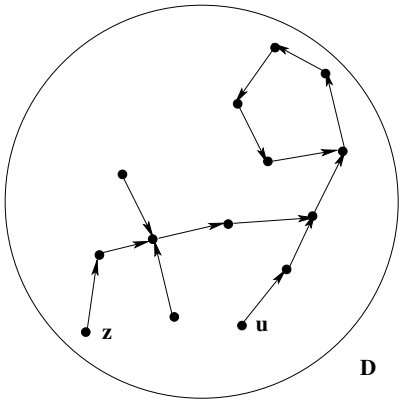


Figure 5: Distances to an orbit: `dorb z u = 4`, `dorb u z = 2`.

4.7. Distance to an orbit, disjunction, collision

A *distance* from `z` to `u`'s orbit is defined by (`inv_circuit_dec z` tests if `z`'s orbit is a circuit):

```
Function dorb (z u: X)
{measure (fun (z: X) => lorb z) z}: nat :=
  if In_dec z (orb u) then 0
  else if inv_circuit_dec z then lorb z
       else S (dorb (f z) u).
```

Fig. 5 illustrates the distance notion. Rather than explaining how the function runs, it is best to give its main properties. First, `dorb z u` is always upper-bounded by `lorb z`. Secondly, `dorb z u = 0` *iff* `z` is in `orb u`. Thirdly, it is easy to characterize the *disjunction* (expressed by `disjl`) between `orb z` and `orb u` and to predict their *collision* at some `z`'s `k`-th iterate by:

```
Theorem disjorb_dorb_lorb: forall z u,
  disjl (orb z) (orb u) <-> dorb z u = lorb z.
Theorem dorb_ge_k_In_Iter_orb: forall z u k, In u D ->
  dorb z u <= k < lorb z -> In (Iter f k z) (orb u).
```

Such results are useful at a low level, when one examines the memory behavior, particularly *aliasing* and *separation* phenomena, during program analyses and correctness proofs. For instance, in a linked − linear or cyclic − lists universe, the first theorem helps to determine if two lists are disjoint, i.e. a deallocation in one of them does not affect the other one. The second theorem gives a condition to affirm that the removal of the `k`-th record of one list will split the other one.

5. Floyd's circuit detection algorithm

In the late 1960s, Floyd worked on the *cycles* − here, we call them *circuits* − of functional finite graphs and Knuth attributes to him the authorship of an efficient circuit-detection algorithm [28](page 4) known as "the tortoise and the hare algorithm". This section shows how orbits help deriving this algorithm and proving that it is totally correct. It follows the notations and the indications of [28](Exercice 6, statement and answer) and [40] closely.

5.1. Preliminaries

The context of this algorithm is exactly the same as for our orbits. Let `x` be any element of type `X`. We name μ − `mu` in Coq −, the handle length of `x`, x_μ − `xmu` in Coq −, the `mu`-th `f`-iterate of `x`, and λ − `lam` in Coq −, the length of x_μ's orbit, i.e. the length of `x`'s orbit ending circuit:

```
Definition mu := lhand x.
Definition xmu := Iter f mu x.
Definition lam := lorb xmu.
```

We assume that `x`'s orbit is a *crosier* in the following *hypothesis* (introduced by the keyword `Hypothesis`) named `inv_cros_x`, which has the same status as a variable:

```
Hypothesis inv_cros_x : inv_crosier x.
```

Note that it can be proved that `inv_cros_x` derives from the single fact that all `f`-iterates of `x` belong to `D`. From the hypothesis, it is easy to prove the *lemma* (introduced by the keyword `Lemma`) `lam_pos`:

```
Lemma lam_pos : 0 < lam.
```

Now, the *circuit-detection problem* is the task of finding λ and μ. Our previous specifications already provide us algorithms to accomplish this task, but they are inefficient. For instance, the direct programming of `lorb` would lead to store and traverse all the current orbital sequence at each step. It would use $O((\mu+\lambda)^2)$ tests and function evaluations and $O(\mu+\lambda)$ storage space.

5.2. Finding ν, a period of the circuit

The tortoise and the hare algorithm uses only two indices which express the moving through `x`'s orbit at different speeds. The key property is that, for any integers i and k such that $\mu \leq i$, $x_i = x_{i+k*\lambda}$. The corresponding lemma is immediately proved from our orbit results:

```
Lemma Fl1 : forall i k, mu <= i ->
   Iter f i x = Iter f (i + k * lam) x.
```

When $\mu \leq i$ and i is a multiple of λ, i.e. $i = m * \lambda$, it follows that $x_i = x_{2*i}$:

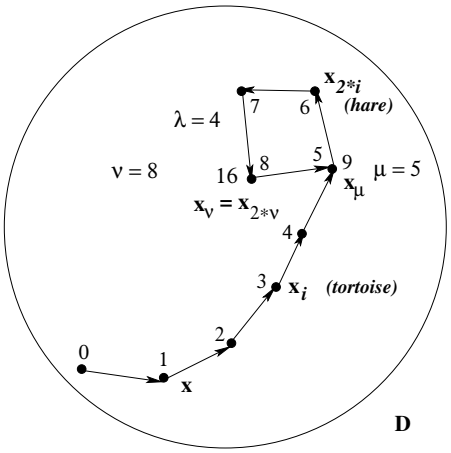


Figure 6: Floyd's circuit detection: The tortoise and the hare.

```
Lemma Fl3 : forall m, mu <= m * lam ->
   Iter f (m * lam) x = Iter f (2 * (m * lam)) x.
```

Hence, the algorithm only needs to check the equality of iterates of the forms x_i, for the tortoise, and x_{2*i}, for the hare (Fig. 6). Once the equality is reached, x_i is necessarily an element of the circuit of x's orbit and a *period* of the circuit is found. This period is a multiple of λ, but maybe not necessarily the lowest. The following lemmas summarize these facts:

```
Lemma Fl4 : forall i, 0 < i ->
 let xi := Iter f i x in
   xi = x2i -> inv_cycle X f D xi.
Lemma Fl6 : forall i j, i < j ->
 let xi := Iter f i x in let xj := Iter f j x in
   xi = xj -> exists m, j - i = m * lam.
```

A function `fnu` (with no argument and an integer result, in fact an integer) which returns such a multiple of λ, called ν (see Fig. 6), is founded on a simple recursion appearing in the following auxiliary function:

```
Function fnu_aux (i:nat)(H:0<i)
 {measure (fun i:nat => (lwm (max i mu)) * lam - i) i}: nat :=
   if (eqd X (Iter f i x) (Iter f (2*i) x)) then i
   else fnu_aux (i+1)(pos_S i).
Definition fnu := fnu_aux 1 pos_one.
```

The function `fnu_aux` has two parameters: `i`, which is incremented by 1 at each recursive call and gives the final value, and `H`, which is a proof that `0 < i`. This precondition avoids a premature termination (with `i = 0`). The term `pos_S i`,

which is a proof that `0 < i+1`, is the second parameter of the recursive call. The function stops when `Iter f i x = Iter f (2*i) x`, i.e. $x_i = x_{2*i}$. The measure is rather fine: it is built thanks to `lwm (max i mu) * lam` giving the *lowest multiple* of `lam` which is greater or equal at the same time to i and to μ. Of course, `(lwm (max i mu)) * lam - i` decreases by 1 at each recursive call. Finally, the definition of `fnu` is just a call to `fnu_aux` with the starting index `1` (and the proof `pos_one` that `0 < 1`).

With the Nœtherian induction principle of `fnu_aux`, it is simple to prove that `fnu` has a good behavior. Indeed, it returns an integer ν such that $\mu \leq \nu$, $x_\nu = x_{2*\nu}$, $\nu = \lambda$, when $\mu = 0$, or $\nu = m * \lambda$, m being *effectively* the *smallest integer* such that $\mu \leq m * \lambda$, when $0 < \mu$. Finally, we always have $\nu \leq \mu + \lambda$:

```
Theorem fnu_ub: fnu <= mu + lam.
```

So, the meeting occurs at the latest when the tortoise has traversed the circuit once, while the hare has been able to traverse it several times. The following lemma will allow us to find μ:

```
Lemma Fl27 : Iter f mu x = Iter f (fnu + mu) x.
```

5.3. Finding μ, the handle length

Once ν is found, to get μ, the algorithm replays the process with two indices having the same speed: `i` starting from `0`, and `fnu + i` starting from `fnu`. Function `fmu` does the job in a call to the recursive function `fmu_aux` (Fig. 6):

```
Function fmu_aux (i:nat)(H:i<=mu)
 {measure (fun i:nat => mu - i) i}:nat :=
   match eqd X (Iter f i x) (Iter f (fnu + i) x) with
      left _   => i
    | right H0 => fmu_aux (i+1)(Recmu i H H0)
   end.
Definition fmu := fmu_aux 0 mu_pos.
```

Function `fmu_aux` has two parameters: an index `i` and a proof `H` that `i<=mu`. The measure exploits the fact that `mu - i` decreases at each recursive call as `i` becomes `i+1`. The expression `Recmu i H H0` returns a proof that `i+1<=mu` in the case where `Iter f i x <> Iter f (fnu + i) x`, a proof of which being called `H0`. Then, by Nœtherian induction on `fmu_aux`, `fmu` is proved to return the appropriate μ value:

```
Lemma Fl32 : fmu = mu.
```

Finally, it is clear that the last value of `Iter f i x` is precisely x_μ, the starting value for the index in the next step.

5.4. Finding λ, the smallest period

Since we have $x_{\mu+\lambda} = x_\mu$, to find λ, the smallest circuit period, it is enough to count the i iteration steps from x_μ to the first $x_{\mu+i} = x_\mu$. This task is achieved by function `flam` which calls the recursive function `flam_aux`:

```
Function flam_aux (i:nat)(H:1<=i<=lam)
 {measure (fun i:nat => lam - i) i}:nat:=
   match eqd X (Iter f i xmu) xmu with
      left _   => i
    | right H0 => flam_aux (i+1)(Reclam i H H0)
   end.
Definition flam := flam_aux 1 lam_ge_one.
```

Function `flam_aux` has two parameters: an index `i` and a proof `H` that `1<=i<=lam`. The measure exploits the fact that `lam - i` decreases at each recursive call as `i` becomes `i+1`. The expression `Reclam i H H0` returns a proof that `1<=i+1<=mu` in the case where `Iter f i xmu <> xmu`, a proof of which being called `H0`. Then, by Nœtherian induction on `flam_aux`, `flam` is proved to return the appropriate λ value, which achieves the proof of *total correctness* of the tortoise and the hare algorithm:

```
Lemma Fl35 : flam = lam.
```

5.5. Towards usual programming

The definitions of the previous functions are not fully satisfying, because they mention iterates in their original forms, x_i − in Coq `Iter f i x` −, when i express the position in the orbit. A good programmer would store them in a single *variable* which value would change during the traversal.

As an example, let us go back to `fnu_aux` and `fnu`, where two variables, `tortoise` and `hare`, can be chosen to store `Iter f i x` and `Iter f 2*i x`. First, an *invariant* can be defined to capture the former constraint `0 < i` and to link variables and values:

```
Definition inv_fnu (i:nat)(tortoise hare: X) :=
  0 < i /\ tortoise = Iter f i x /\ hare = Iter f (2*i) x.
```

Then, the functions can be redefined in Coq:

```
Function fnu1_aux
 (i:nat)(tortoise hare: X)(H:inv_fnu i tortoise hare)
 {measure (fun i:nat => (lwm (max i mu)) * lam - i) i}: nat :=
   if eqd X tortoise hare then i
   else fnu1_aux (i+1) (f tortoise) (f (f hare))
        (Rec_inv_fnu i tortoise hare H).
Definition fnu1 := fnu1_aux 1 (f x) (f (f x)) inv_fnu_1
```

The parameters of `fnu1_aux` are `i`, `tortoise`, `hare`, and `H` which is a proof of `inv_fnu i tortoise hare`. The inner recursive call is changed in consequence, with a mechanism (which is constructed by the programmer) providing a proof, `Rec_inv_fnu i tortoise hare H`, of the invariant with the new parameters. The value of ν is given by the new function `fnu1` as a call to `fnu1_aux` with appropriate actual parameters. It is proved that `fnu_aux` and `fnu1_aux` give the same results, *whatever their proof-parameters* are:

```
Lemma Fl50 :
 forall i (tortoise hare: X)
   (H:0 < i)(H1:inv_fnu i tortoise hare),
  fnu_aux i H = fnu1_aux i tortoise hare H1.
```

Of course, we are still far from usual functions. However, the Coq *extraction* in OCaml leads to a function `fnu1` including a recursive auxiliary function `hrec` (introduced by `let rec`) in which all proof-features are removed (capital are rewritten into lowercases, and `x` is changed into `x0`):

```
let fnu1 x f d x0 =
  let rec hrec i tortoise hare =
    if x.eqd tortoise hare then i
    else hrec (plus i (S O)) (f tortoise) (f (f hare))
in hrec (S O) (f x0) (f (f x0))
```

That is the best form we can expect in functional programming. Since it is *tail-recursive*, its rewriting in an imperative language leads to an efficient iterative operation which uses $O(\mu + \lambda)$ tests and function evaluations and $O(1)$ storage space, if one excepts the coding of each Peano number i in space $O(i)$. The functions returning μ and λ can be transformed in the same way.

Now, the first Coq section of our orbit library is achieved and closed. Therefore in what follows, all orbit features must be parameterized by three actual parameters corresponding to `X`, `f` and `D`.

6. Partial injections

We now study the case where `f` is a *partial injection* in a new Coq section where `X` and `f` are declared as variables again. So, they do not explicitly appear in the new definitions and statements. However, `D`, which must change, is always apparent. First of all, for any `z:X`, possibly not in `D`, we define what is "having a `f`-predecessor":

```
Fixpoint pred (D:list X)(z:X): Prop:=
  match D with
     [] => False
   | x :: D0 => f x = z /\ z <> exc X \/ pred D0 z
  end.
Lemma pred_charac: forall D z,
  pred D z <-> (z <> exc X /\ exists x, In x D /\ f x = z).
```

Here, `exc X` is an *exception* in `X`. We chose this simple mechanism rather than the use of the *supertype* `option X` [26, 4] because the latter forces to construct/deconstruct its elements in every definition or proof. The predicate `pred D z` is defined by *structural induction* (which is introduced by the keyword `Fixpoint`) on `D`. It is `True` *iff* `z <> exc X` and there exists in `D` any `x`, which is the *inverse* of `z` by `f`, i.e. `f x = z`. This is confirmed by the above lemma which is proved by structural induction on `D` and gives a *characteristic property*. Note that any `z` not in `D` may have a predecessor (in `D`) if `z <> exc X`, `exc X` being excluded because it is intended to play the role of the *null* address at a low level. So, the predicate saying that `f` (which is implicit) is a *partial injection* on `D` is defined by structural induction:

```
Fixpoint inv_partinj(D:list X):Prop:=
  match D with
     [] => True
   | x :: D0 => (x <> exc X /\ ~ In x D0 /\
                 ~ pred D0 (f x)) /\ inv_partinj D0
  end.
```

This definition entails that `exc X` never lies in `D` if `inv_partinj D`. This invariant contains the usual *injection* property for `f`, stated in the following lemma, but only when `f z <> exc X` (for instance, two different lines could have `exc X` as a limit, as for singly-linked lists at a low level). The definition by structural induction of an *inverse*, denoted by `Fi`, follows:

```
Lemma f_injective: forall D z t,
  inv_partinj D -> In z D -> In t D -> f z <> exc X ->
    f z = f t -> z = t.
Fixpoint Fi(D:list X)(z:X) :=
  match D with
     [] => exc X
   | x :: D0 => if eqd X z (exc X) then exc X
                else if eqd X (f x) z then x else Fi D0 z
  end.
```

Fig. 7(b) shows the inversion of the full linear component of `z` (Fig. 7(a)). When `f` is a partial injection, `Fi` enjoys expected properties, i.e. `Fi D (f z) = f (Fi D z) = z`, under some conditions appearing in the lemmas:

```
Lemma f_Fi: forall D z, inv_partinj D ->
   pred D z -> f (Fi D z) = z.
Lemma Fi_f: forall D z, inv_partinj D ->
   In z D -> f z <> exc X -> Fi D (f z) = z.
```

Moreover, when `z`'s orbit is a *circuit*, `Fi` and `f_1` (defined in Section 4) are equal on `D`. The partial injection hypothesis drastically simplifies orbits and components. First of all, there are no more "strict" crosiers, i.e. *all the orbits are lines or circuits*. Secondly, the orbit shapes for `Fi` are the same as for `f`:

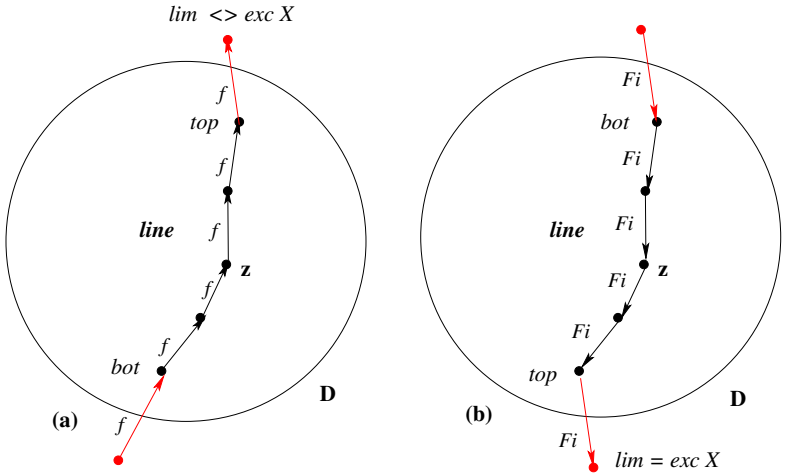


Figure 7: Inversion of a linear connected component containing `z`'s orbit.

```
Theorem line_or_circuit:  forall D z, inv_partinj D ->
   inv_line X f D z \/ inv_circuit X f D z.
Theorem inv_line_wrt_f_Fi: forall f D z, inv_partinj X f D ->
   (inv_line X f D z <-> inv_line X (Fi X f D) D z).
Theorem inv_circuit_wrt_f_Fi: forall f D z, inv_partinj X f D ->
   (inv_circuit X f D z <-> inv_circuit X (Fi X f D) D z).
```

As a consequence, for a partial injection, a connected component is just a line or a circuit. Even the general trees have disappeared, which is correct since any element has at most one predecessor by `f`.

If we want to work only with circuits, we can *close* all the "open orbits", i.e. the lines. This operation, which modifies `f`, but not `D`, goes through the definition of `top` (Section 4) and `bot` (Fig. 7(a) and 8):

```
Definition bot f D z := top (Fi X f D) D z.
Definition Cl f D z :=
  if In_dec X z D
  then if In_dec X (f z) D then f z else bot f D z
  else z.
```

After rather long proof developments, the results are the expected ones: in the closure `Cl f D` of `f`, all the orbits are circuits and the reachability `expo` on the closure is symmetrically obtained from the reachability for `f`:

```
Theorem inv_circuit_Cl: forall f D z, inv_partinj X f D ->
   inv_circuit X (Cl f D) D z.
Theorem expo_Cl: forall f D z t, inv_partinj X f D ->
  (expo X (Cl f D) D z t <-> (expo X f D z t \/ expo X f D t z)).
```

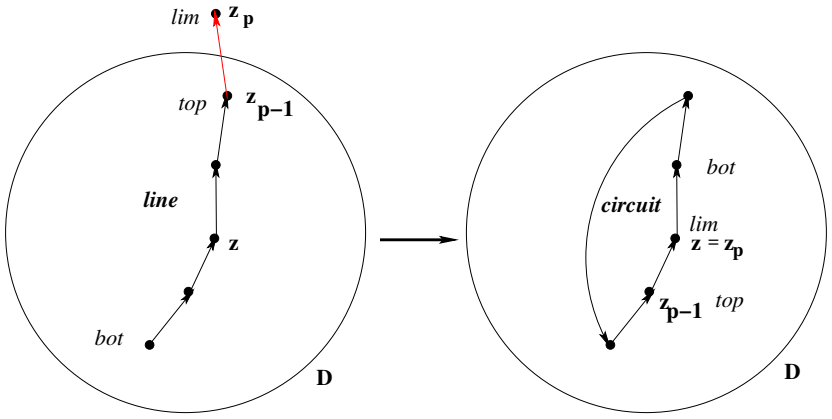


Figure 8: Effect of the closure of `f` on a line orbit.

The closure operation allows us to understand well the progressive building of circuits. It plays a great role in the algebraic specification of orbits in combinatorial (hyper)maps [13].

7. Addition and deletion

The first update operation that we consider is the *addition* of a new element `a` to `D`, while function `f:X -> X` remains the same. We require `~ In a D`, we write `Da` for `a :: D` and `a1` for `f a`. Here, `X`, `f` and `D` are in the context again thanks to the opening of a new Coq section where they are declared as variables. Regarding the orbit variation, for any `z:X`, two cases arise (Fig. 9):

- **Case A1**: `lim X D f z <> a`. Then, the orbit of `z` is fully preserved:

  ```
  Theorem orb_Ad_1: forall z,
     lim X f D z <> a ->
        orb X f Da z = orb X f D z.
  ```

  This result holds for all orbit shapes, i.e. for *lines*, *circuits* and *crosiers*, which always stay in their classes and keep their characteristics.

- **Case A2**: `lim X D f z = a`. This case is only about *lines*, because *circuits* and *crosiers* do not have a limit outside `D`. Moreover, we also suppose that `~ In a1 D`, which corresponds to the practical cases. If this was not true, the addition may be followed by a *mutation* of `f` at `a` to change the value of `a1`, using the results of Section 8. Thus, `z`'s orbit is augmented by `a`:

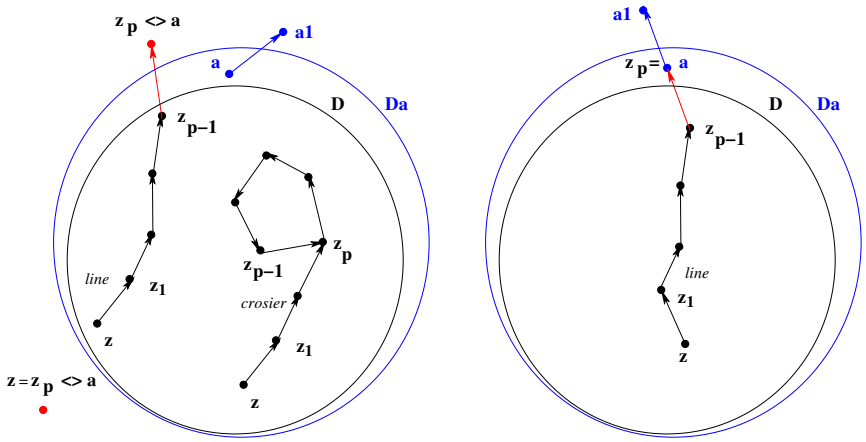


Figure 9: Addition effect on the orbits (two cases).

```
Theorem orb_Ad_2 : forall z,
  lim X f D z = a -> ~ In a1 Da ->
    orb X f Da z = a :: orb X f D z.
```

The new orbit for `z` is a *crosier* if `a1 = a`, and a *line* otherwise. Note that, with our orbit ordering conventions, the last entering element `a` comes at the orbit front. Moreover, for `a` itself, since `lim X f D a = a`, we always have `orb X f Da a = a ::[]`.

Now let us consider the "reverse" operation, i.e. the *deletion* of an element `a` from `D`. Then, we require `In a D` and we write `D_a` for `remove D a`. Regarding the orbit variation of any `z:X`, two cases arise (Fig. 10):

- **Case D1**: `~ In a (orb X f D z)`. Then `z`'s orbit is fully preserved:

```
Theorem orb_Del_1: forall z,
  ~ In a (orb X f D z) ->
     orb X f D_a z = orb X f D z.
```

This result holds for all the orbit shapes, i.e. for *lines*, *circuits* and *crosiers*, which always stay in their classes and keep their characteristics.

- **Case D2**: `In a (orb X f D z)`. Then the new `z`'s orbit is only a "beginning section" of the former one. More precisely, if `a = Iter f j z` with `j < lorb X f D z`, we have:

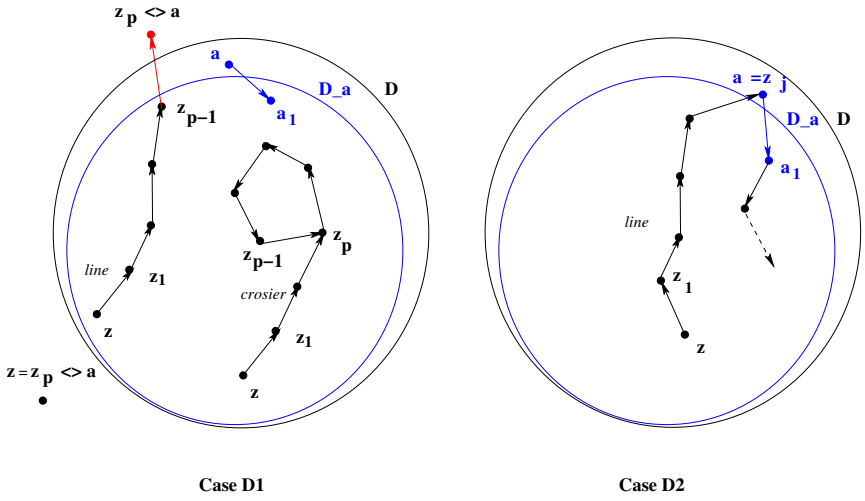


Figure 10: Deletion effect on the orbits (two cases).

```
Theorem orb_Del_2 : forall j z,
  a = Iter f j z -> j < lorb X f D z ->
    orb X f D_a z = orbs f j z.
Theorem lim_Del_2 : forall j z,
  a = Iter f j z -> j < lorb X f D z ->
    lim X f D_a z = a.
```

Consequently, the new shape of `z`'s orbit is always a *line* and its limit is `a`. Of course, for `z = a`, we have `orb X f D_a a = []`.

8. Mutation

This case is more difficult to analyze. A *mutation* modifies the *image* of an element `u` by `f`, i.e. `f u`, into an element which we name `u1`, while `D` remains the same. In a context containing `X` as a variable, the new function denoted by `Mu f u u1` is defined by:

```
Definition Mu(f:X->X)(u u1:X)(z:X):X :=
  if eqd X u z then u1 else f z.
```

Using the axiom of *extensionality* (Section 1), it is clear that `Mu f u u1 = f` if `u1 = f u`, `Mu (Mu f u u1) u u1' = Mu f u u1'`, and `Mu (Mu f u u1) u' u1' = Mu (Mu f u' u1') u u1` if `u <> u'`. Such a modification can have major consequences on orbits. Two cases can be considered:

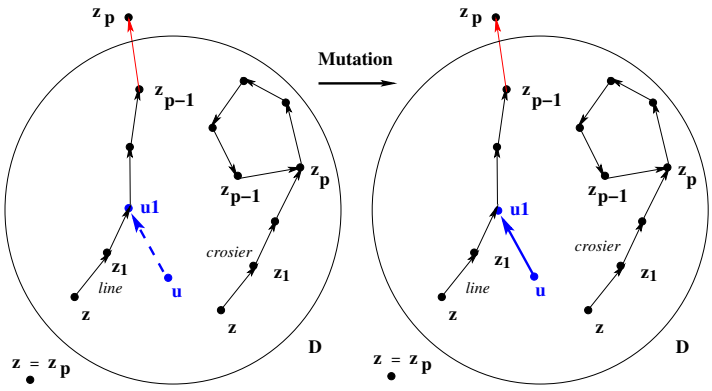


Figure 11: Mutation effect on z's orbit where u is not.

- **Case M1**: `~ In u (orb X f D z)`. Then, the orbit and limit of `z` are fully preserved. Fig. 11 shows different positions of `z`:

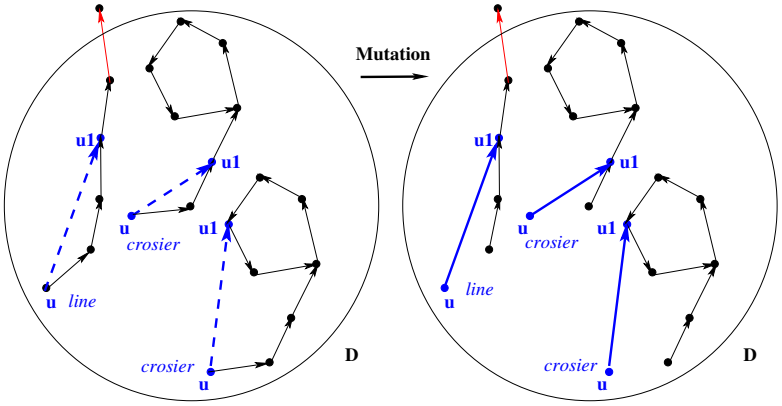
```
Theorem orb_Mu_1: forall z,
  ~ In u (orb X f D z) ->
     orb X (Mu f u u1) D z = orb X f D z.
Theorem lim_Mu_1: forall z,
  ~ In u (orb X f D z) ->
     lim X (Mu f u u1) D z = lim X f D z.
```

- **Case M2**: `In u (orb f D z)`. Then, nothing happens in the section of `orb f D z` going from `z` to `u`, both included. So, it is important to see what happens for the orbits of `u` and `u1`. Two subcases arise, depending on whether `u` is in `orb f D u1` or not:

  - **Subcase M21**: `~ In u (orb f D u1)`. In this subcase, from *Case M1*, the orbit of `u1` does not change. Then the orbit of `u`, supposed to be in `D`, is the one of `u1` plus `u` itself (by the append `++ (u :: [])`):

```
Theorem orb_Mu_21_u:
  In u D -> ~ In u (orb X f D u1) ->
    orb X (Mu f u u1) D u = (orb X f D u1) ++ (u :: []).
Theorem lim_Mu_21_u:
  In u D -> ~ In u (orb X f D u1) ->
    lim X (Mu f u u1) D u = lim X f D u1.
```

    Fig. 12 shows three configurations for `u` and `u1`. This works with any shape for `u1`'s orbit. If `orb f D u1` is a *line*, `orb (Mu f u u1)`



Case M21: Three configurations for u and u1

Figure 12: Effect of a mutation when `u` is absent from `u1`'s orbit.

`D u` is a line as well. If `orb f D u1` is a *circuit* or a *crosier*, `orb (Mu f u u1) D u` is a crosier. Then, the new orbit of any `z:X` when `In u (orb f D z)`, with `u = Iter f ku z`, can be expressed by:

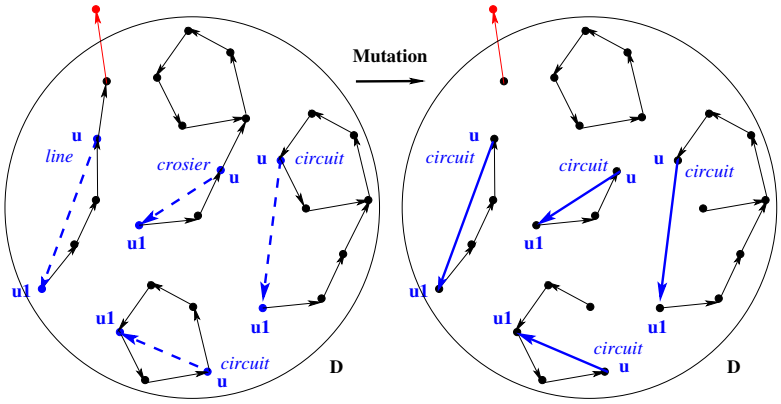
```
orb X (Mu f u u1) D z =
   orbs X f ku z ++ u :: orb X f D u1.
```

– **Subcase M22**: `In u (orb f D u1)`. Then, it is clear that the mutation closes a *new circuit* which, say, starts from `u1`, goes to `u` by the old path, then goes back to `u1` in one step. So, if `u = Iter f j u1` with `j < lorb f D u1`, we have the following results for `u1`, that Fig. 13 illustrates with different positions of `u` and `u1`:

```
Theorem lorb_Mu_22_u1: forall j,
  u = Iter f j u1 -> j < lorb X f D u1 ->
    lorb X (Mu f u u1) D u1 = 1 + j.
Theorem  inv_circuit_Mu_22_u1: forall j,
  u = Iter f j u1 ->  j < lorb X f D u1 ->
    inv_circuit X (Mu f u u1) D u1.
Theorem orb_Mu_22_u1: forall j,
  u = Iter f j u1 ->  j < lorb X f D u1 ->
    orb X (Mu f u u1) D u1 = orbs f (1 + j) u1.
```

The shape of `orb (Mu f u u1) D u1` is a *circuit* of length `1 + j`. Similar results are obtained for `u`, which lies in the same new circuit just before `u1`.

The new orbit of any `z:X` can be obtained similarly. Different cases arise depending on whether `z` is between `u1` and `u` in an orbit, or `u1` is between `z` and



Case M22: Four configurations for u and u1

Figure 13: Effect of a mutation when `u` is in `u1`'s orbit.

`u`, or `u` is in `z`'s orbit but `z` not in `u1`'s. The results are not given here, but are available in [15].

In fact, it is often sufficient to know how *reachability* is affected. The following theorem `expo_Mu_NSC` states, for all `z,t`, a *necessary and sufficient condition* for `t` being in `z`'s orbit after the mutation, in other words, for `t` being reached from `z`, or `expo X (Tu f D u u1) z t` (where `expo_dec` denotes the decision test of `expo`):

```
Theorem expo_Mu_NSC: forall z t,
  inv_partinj X f D -> prec_Mu f D u u1 ->
    (expo X fu D z t <->
      if expo_dec X f D z u
      then if expo_dec X f D u1 u
           then expo X f D t u
           else betw X f D z t u \/ expo X f D u1 t
      else expo X f D z t)
```

Fig. 14 shows the main two cases, where `u1` is located in `u`'s orbit or not, with two occurrences for `t` (the second one is denoted `t'`). The theorem covers the cases which are met in our combinatorial map example. In fact, it is available only if `f` is a *partial injection* in `D`, which is expressed by the *invariant* `inv_partinj X f D`, and if the *precondition* `prec_Mu f u u1` is satisfied. When `u` belongs to `D`, this precondition imposes that `u1` has no predecessor by `f`:

```
Definition prec_Mu f u u1 := In u D -> ~ pred X f D u1
```

This condition ensures that the new function, `Mu f u u1`, remains a partial injection. Such a theorem is rather difficult to prove and work with. However it could be generalized for even more complex cases, where `f` can be any function.

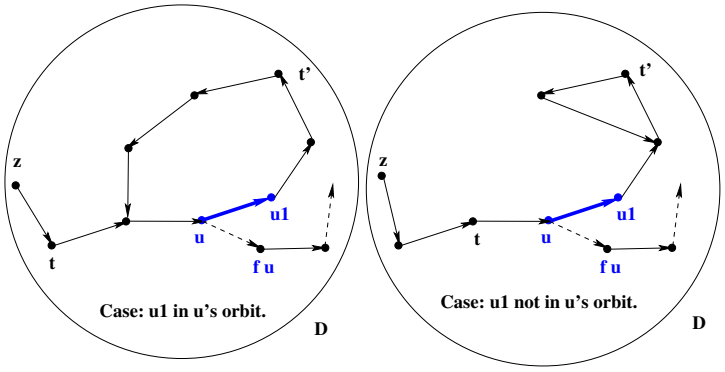


Figure 14: Existence of a path in an orbit from `z` to `t` (or `t'`) after a mutation.

9. Transposition

In a general mathematical framework where X is a space, D any finite subset of X, $f : D \to D$ a *permutation* on D, and u, u' two elements in D, the *transposition* between u and u' is a standard operation which consists in *exchanging* the images by f of the two elements.

Our Coq definition with any *total* function `f:X -> X` is quite similar when the exchange occurs between two elements the orbits of which are *circuits*. Indeed, the restriction of `f` to the elements of both circuits is a permutation. In our case, it is more convenient to provide an element `u` and its *new successor* `u1` in the transposition. Then, `X` being alone in the context, the definition of the transposition operation `Tu` is the following:

```
Definition Tu(f:X->X)(D: list X)(u u1:X)(z:X):X :=
  if eqd X u z then u1
  else if eqd X (f_1 X f D u1) z then f u
       else f z.
```

The definition also states that the former `u`'s `f`-successor, i.e. `f u`, is the new successor of `u1`'s `f`-predecessor, i.e. `f_1 X f D u1`, and that nothing is changed for the other elements (Fig. 15 and 16).

This operation is usable whether `u` and `u1` are in the same circuit or not. Intuitively, in the first case, the unique circuit is *split* into two circuits, and in the second case, the two circuits are *merged* into a single one. That is the intuition, but the formal proof of these facts is very far from being easy.

Such an operation is at the basis of a multitude of treatments. Let us simply go back to our combinatorial map example. At a high level, the transposition is the heart of the split/merge of cells of any dimension − vertices, edges or faces. For instance, merge and split of vertices are intensively used in the manipulation of plane subdivisions during a Delaunay triangulation [18]. Moreover, in order to control genus and planarity, it is necessary to count orbits and connected

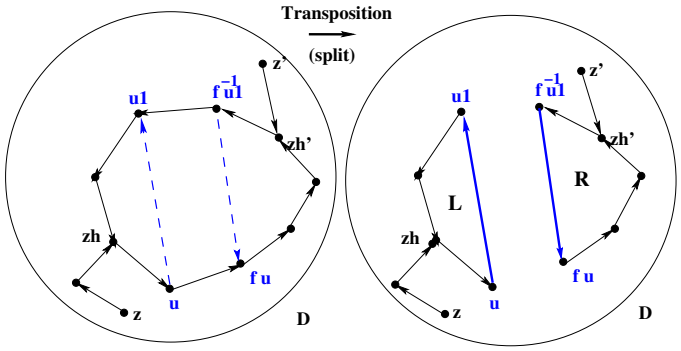


Figure 15: Case T1: splitting u's orbit into two pieces.

components, as we will do in Section 10. At a low level, a transposition allows to govern the updating of cyclic linked lists, like those describing edges and vertices in our combinatorial map representation, for instance when performing list concatenations.

We distinguish the two cases (Fig. 15 and 16), the following results being given in a context where `X`, `f` and `D` are variables:

- **Case T1 (split)**: `u1 = Iter f i1 u` with `2 <= i1 <= lorb X f D u`, i.e. `u1` is in the same circuit as `u`, or is the `i1`-th iterate of `u`, avoiding the case `u1 = f u`, where the transposition has no effect.

  Then, the circuit is *split* into two parts, named `L` and `R`, which are also circuits (cf. Fig. 15, with `i1 = 6`). The length of `L` is the length of the initial circuit minus `i1 - 1`, while the one of `R` is `i1 - 1`. Moreover, when `i1 = lorb X f D u`, i.e. `u1 = u`, `L` reduces to a *loop* — i.e. a circuit of length 1, and `R` to the initial circuit minus `u`. In Fig. 15, `lorb X f D u = 9` and the lengths of `L` and `R` are `4` and `5`, respectively.

  These results are confirmed by the following theorems, the first two about `u`, the others about `f u`, since `u` and `f u` are in `L` and `R`, respectively:

```
Theorem lorb_Tu_L_u :
  lorb X fu D u = S (lorb X f D u - i1).
Theorem inv_circuit_Tu_L_u:
  inv_circuit X fu D u.
Theorem lorb_Tu_R_f_u :
  lorb X fu D (f u) = i1 - 1.
Theorem inv_circuit_Tu_R_f_u:
  inv_circuit X fu D (f u).
```

- **Case T2 (merge)**: `~ expo X f D u u1`, i.e. `u` and `u1` are in different circuits. Then, the two cicuits are *merged* into one, the length of which

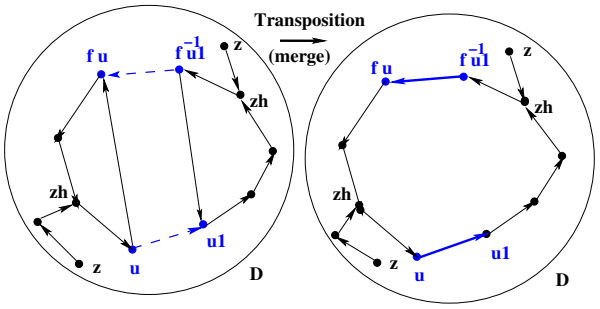


Figure 16: Case T2: merging two orbits.

is the sum of the former two (cf. Fig. 16). These results are for instance confirmed by the following theorems:

```
Theorem lorb_Tu_M_u:
  lorb X fu D u = lorb X f D u + lorb X f D u1.
Theorem lorb_Tu_M_u1:
  lorb X fu D u1 = lorb X f D u + lorb X f D u1.
Theorem inv_circuit_Tu_M_u:
  inv_circuit X fu D u.
Theorem inv_circuit_Tu_M_u1:
  inv_circuit X fu D u1.
Theorem orb_Tu_M_u1:
  orb X fu u1 = orb X f D (f u) ++ orb X f D u1.
```

Of course, the other orbits are modified by `Tu` only if they contain `u` or `f_1 X f D u1`. In this case, the same study as for `Mu` (Section 8) can be conducted. Again, for any `z`, the result depends on the position of `zh := Iter f D (lhand X f D z) z` with respect to `u` and `u1`: for split, `between u1 zh u` or `between (f u) zh (f_1 u1)`; for merge, `zh` in `u`'s circuit or in `u1`'s circuit. Fig. 15 and Fig. 16 illustrate the different cases.

In fact, as for `Mu` (Section 8), it is often enough to know how the *reachability* is modified. As an example, the following theorem states, for any `z` and `t`, a *necessary and sufficient condition* for `t` being in `z`'s orbit after the transposition, in other words, for `t` being reached from `z`, i.e. `expo X (Tu f D u u1) z t`:

```
Theorem expo_Tu_NSC: forall z t,
  inv_circuit X f D z -> inv_circuit X f D t ->
  (expo X fu D z t <->
    if expo_dec X f D u u1
    then
      betw X f D (f u) z (f_1 X f D u1)
        /\ betw X f D (f u) t (f_1 X f D u1)
```

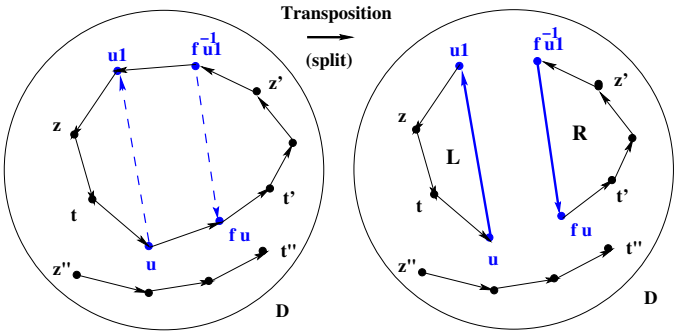


Figure 17: Case T1 (split): existence of paths.

```
   \/ betw X f D u1 z u /\ betw X f D u1 t u
   \/ ~ expo X f D u z /\ expo X f D z t
else
   expo X f D z t
   \/ (expo X f D u z \/ expo X f D u1 z)
       /\ (expo X f D u1 t \/ expo X f D u t)).
```

As before, the theorem distinguishes split and merge cases:

- **Case T1 (split)**: `expo X f D u u1`, i.e. there is an orbital path from `u` to `u1`. Then, an orbital path from `z` to `t` exists after the transposition *iff*, before the transposition, one of the three following conditions holds (Fig. 17, with three positions: `(z,t)`, `(z',t')` and `(z",t")`):
  - **(a)**: `z` and `t` lie between `f u` and `f_1 X f D u1`
  - **(b)**: `z` and `t` lie between `u1` and `u`
  - **(c)**: `z` is not in `u`'s orbit (= `u1`'s orbit) and an orbital path from `z` to `t` exists
- **Case T2 (merge)**: `~ expo X f D u u1`, i.e. no orbital path exists from `u` to `u1`. Then, an orbital path from `z` to `t` exists after the transposition *iff*, before the transposition, one of the following two conditions holds (Fig. 17, with two positions: `(z,t)` and `(z',t')`):
  - **(a)**: an orbital path from `z` to `t` exists
  - **(b)**: `z` and `t` lie in `u`'s circuit or `u1`'s circuit.

The proof of this theorem reuses the results for mutation which were presented in Section 8, since a transposition can be decomposed into three mutations satisfying the conditions of `expo_Mu_NSC` [15].

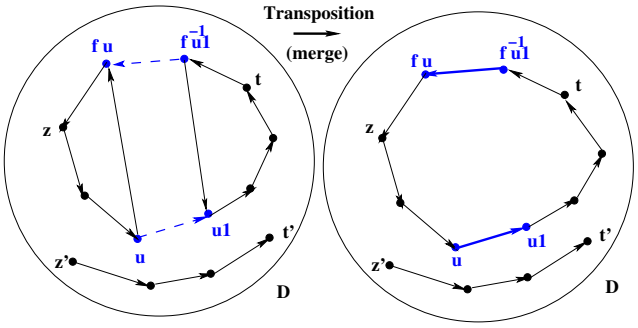


Figure 18: Case T2 (merge): existence of paths.

10. Connected components

Here, only `X` and `f` are supposed to be in the context. When `f` is a partial injection, we first define a relation of *connectivity* between elements of `X`. Intuitively, `z` and `t` are *connected iff* they are related by a *chain* of elements of `D`, i.e. a sequence `z0 = t`, ... , `zk`, ..., `zn = t`, such that `zk` is in `D` with `z(k+1) = f zk` or `zk = f z(k+1)`, for `1 <= k < n`. This relation, denoted by `eqc1 D z t`, can be defined by induction in `D`, as in Warshall's algorithm [39], without forgetting to test the existence or not of the `f`-successor, `x1 := f x`, and `f`-predecessor, `x_1 := Fi D0 x`, of the current `x` in the current set `D0`:

```
Fixpoint eqc1(D:list X)(z t:X): Prop :=
  match D with
     [] => False
   | x :: D0 => let x1 := f x in let x_1 := Fi D0 x in
     eqc1 D0 z t
       \/ (z = x \/ In x_1 D0 /\ eqc1 D0 z x_1
           \/ In x1 D0 /\ eqc1 D0 z x1)
          /\ (t = x \/ In x_1 D0 /\ eqc1 D0 t x_1
             \/ In x1 D0 /\ eqc1 D0 t x1)
  end.
```

The binary relation `eqc1 D` is proved *decidable*, *reflexive*, *symmetric* and *transitive* by structural induction on `D`. Moreover, the connectivity is equivalent to the existence of an orbital path from `z` to `t` *and* from `t` to `z`:

```
Theorem eqc1_expo: forall D z t, inv_partinj X f D ->
  (eqc1 X f D z t <-> (expo X f D z t \/ expo X f D t z)).
```

Now, the *number of connected components* for `f` in `D` is defined from `expo`:

```
Fixpoint nc1 (D:list X): Z:=
  match D with
```

```
    [] => 0
  | x :: D0 => let x1 := f x in let x_1 := Fi D0 x in
   nc1 D0 +
    if In_dec X x1 D0
    then if In_dec X x_1 D0
         then if expo_dec X f D0 x1 x_1 then 0 else -1
         else 0
    else 0
end.
```

The definition is given by structural induction on `D`. If `D` is empty then `nc1 D = 0` else `D = x ::D0` for some `x` and `D0`. So, the new connections in `D` depend whether `x1 := f x` and `x_1 := Fi D0 x` belong to `D0` and are already connected. Finally, regarding the update operations, `Del`, `Mu` and `Tu`, we have the following results:

```
Theorem nc1_Del_equation: forall X f D a,
  inv_partinj X f D -> ~ In (f a) D ->
   let a_1 := Fi X f D a in
    nc1 X f (Del X f D a) = nc1 X f D +
      if In_dec X a_1 D then 0 else -1.
Theorem nc1_Mu_equation: forall X f D u u1,
  inv_partinj X f D -> prec_Mu X f D u u1 ->
    nc1 X (Mu X f u u1) D = nc1 X f D +
     if In_dec X u D then
       (if expo_dec X f D u1 u then 0
        else if In_dec X u1 D then -1 else 0) +
       (if inv_circuit_dec X f D u then 0
        else if In_dec X (f u) D then 1 else 0)
     else 0.
Theorem nc1_Tu_equation : forall X f D u u1,
  inv_partinj X f D -> prec_Tu X f D u u1 ->
    nc1 X (Tu X f u u1) D = nc1 X f D +
      if eqd X (f u) u1 then 0
      else if expo_dec X f D u u1 then 1 else -1.
```

Of course the addition by `Ad` is encompassed by the definition of `nc1`. For `Del`, with the precondition `~ In (f a) D`, the number of components is the same as before if `a_1 := Fi X f D a` is in `D`, and it decreases by 1 otherwise (`a` is isolated in `D`).

For `Mu`, assuming the precondition `prec_Mu X f D u u1`, if `u` is not in `D`, the number of components is the same as before. If `u` is in `D`, one has to examine if `u1` and `u` are already connected, if `u`'s orbit is a circuit or not, and the belonging to `u1` and `f u` to `D` (see Fig. 12 and 13). Theorem `nc1_Mu_equation` was by far the longest and most difficult result to establish in all this work, with a proof of around 2,500 lines.

For `Tu` satisfying the precondition `prec_Tu X f D u u1` (just stating that `u`'s and `u1`'s orbits are non-empty circuits), one only has to see if the operation has no effect, i.e. `f u = u1`, else if it is a split, i.e. `expo X f D u u1`, else a merge, i.e. `~ expo X f D u u1` (Fig. 15 and 16). The variation of the number of components (always circuits) immediately follows (see Fig. 17 and 18).

These results are sufficient to deal with all the updating situations of a partial injection in a finite set. For instance, they allow us to count our cells in combinatorial (hyper)maps.

11. Related work and discussion

What we have studied, i.e. the track left in a finite subdomain D of a space X by the iterates of a total function $f : X \to X$, roughly corresponds to a (directed) *functional graph* in X [3]. But, some particularities are to be highlighted. Indeed, the graph nodes outside D can be viewed as *erased*, *isolated* or *still connected* when the interest is for connected components, orbits, or update operations, respectively. This provides us a great flexibility for delicate reasonings in some applications.

Moreover, although the graph literature often prefers the *path* notion to the *orbit* one, the latter is more accurate and synthetic, since it covers all the possible *simple* paths from an element z to *all the elements* which are accessible from z until the end. Thereby, it involves a rich classification in lines, crosiers and circuits, which can be used as *invariants* and *variants* − or *measures* − in type definitions, algorithms and proofs. The path notion does not offer this possibility directly.

11.1. Orbits at a high level

In computer science, the orbit notion is used at a high level, mostly for *permutations*, where it corresponds to the one of *circuit*, as in the pioneering work by Tutte [38].

Orbits for permutations play a major role in geometric modelling, since permutations are at the basis of the combinatorial maps [38] used to describe the topology of surface subdivisions, e.g. in the "paper" surface vision of Griffiths [22]. In fact, the topological dimension of the geometric objects can be greater than 2, e.g. 3 for volumes or 4 for animated objects (the 4-th dimension is the time), etc. When the dimension increases by 1, there must be one more permutation. Thus, n permutations are necessary to model objects in n-dimensional spaces. The cells of the objects are easily defined from orbits of the permutations or their compositions, as shown by Tutte [38] or Lienhardt [29].

However, during their construction, until their *closure*, future circuits are lines. So, if atomic operations are to be considered, orbits more general than circuits must be studied. When no constraint is imposed on the n permutations − e.g. having no fixpoint or being involutions −, the combinatorial algebraic structure corresponds to the n-dimensional *hypermap*, the 2-dimensional case being introduced by Cori [11].

Plane combinatorial hypermaps are at the foundation of a great result, the Four Color theorem, proved using Coq/SSReflect by Gonthier [21]. The notion of orbit is fully present in this work, but not considered with all the modifications we have presented.

We have intensively studied hypermaps and orbits, as well as their basic operations of construction, by algebraic specifications [5], then more thoroughly in Coq specifications [13]. Our approach is quite *constructive*, because hypermaps are defined as an inductive type with 3 constructors with preconditions [13]. These specifications are at the basis of our proofs for a discrete Jordan Curve theorem [14] and for the total correctness of algorithms in computational geometry [18, 7].

In comparison, the approach by Gonthier [21] is rather *observational*, because hypermaps are defined as in Section 3.2 by a set with 3 permutations (for α, π and ϕ) and an invariant saying that $\phi = \pi^{-1} \circ \alpha^{-1}$. In fact, past the first specification levels, the difference is less and less noticeable, apart from the fact that we use Coq without the SSReflect extension. This choice lengthens some proofs but makes the transport to other proof assistants easier.

All these works rely on an orbit notion which is attached to 2-dimensional (hyper)maps only. Thus, to work in a higher dimension or investigate frameworks other than geometric modelling, it was necessary to study the orbit notion in a more general way, which is what we achieve in the present article.

11.2. Orbits at low level

Orbits are underlying many works at a low level, particularly in the domain of the proof of pointer programs, which is still a difficult problem. This area was almost always addressed by means of Hoare logic [25] and extensions to overcome difficult *stack* and *heap* management questions, like aliasing, separation and collision in concrete data structures.

The first decisive work in this area seems due to Burstall [8]. It defines some concepts to deal with linked list processing: list segments α between two adresses u and v, denoted by $u \xrightarrow{\alpha} v$, with operations and rules, non-repetition and list disjunction predicates. It axiomatizes list systems to deal with several lists and solve classical problems, such as reversal of linear and cyclic lists. Moreover, it introduces a similar theory for trees. The formalization is firmly founded on category theory.

The study of Bornat [6] deals with address sequences like our orbits, denoted by $\Rightarrow_f$ when following a same cell pointer field called f. A lot of laws are given in order to avoid global reasoning on a memory and to favour local reasoning. The essential is the *spatial separation principle* between sequences to preclude pointer aliasing and circuits in algorithms. Bornat handles a rich variety of linked structures − linear structures, cyclic or acyclic graphs − and gives examples based on lists, including the in-place list reversal and a convincing graph marking [6].

These questions are systematized in *separation logic* by Reynolds [34], and largely developed for information hiding and partitionning [33]. This logic seems

particularly well adapted to distributed or embedded systems [30] and to composite data structures [2].

In these works, higher-order predicates are to be constructed in order to deal with all the linked memory configurations. Sophisticated methods to discover them are studied [2]. We claim that our orbit notion is sufficient to understand and deal with the problem arising in the treatment of all the linked linear or cyclic data structures possibly hierarchized, particularly to write *invariants* and *variants*, and to prove the *total correctness* of concrete operations.

So, our work could help in the field of *memory shape analysis*, like in works we discussed [6, 2]. The notion of *orbit* underlies all these works and others where *tracked locations* [23] or *multi-patterns* [10] appear.

Although the benefit in low level program verification seems evident to federate some memory accessibility features, it seemed to us that the orbit notion deserves a study for itself, independently of program analysis. Moreover, we have not proposed an axiomatic system, but we have started "from scratch" and proved all the laws which we propose to use.

12. Conclusion

In this paper, we formalized functional orbits in finite domains, proved their properties and classified them statically. We examined basic orbit update operations − addition, deletion, mutation and transposition, the last one only in the case of cyclic orbits. We particularly studied the paths in orbits in the case where functions are partial injections.

The Coq library we built contains about 20,000 lines, 230 definitions, 1900 lemmas and theorems [15]. It is reusable whenever the orbit notion is involved at a high or a low level. Here, we presented only an introduction to this library, which also contains many other results.

So far, we have experimented the library at a high level to formalize the combinatorial (hyper)maps, which underlie the geometric modellers. We also have used the library at a low level to certify linked memory structures representing hypermaps [16]. The orbit properties allowed us to easily write invariants and measures, and to prove that concrete operations are totally correct.

In the future, we will generalize the results we obtained on partial injections to the case of general functions, particularly for connectivity and numbering of connected components. We will also see how to deal with "orbits" when several functions are mixed.

At a high level, we will continue to prove classical algorithms on sequences or permutations, those which are proposed in [27, 28] being particularly rich. We will extend our applications in geometric modelling with the formalization of 3-dimensional hypermaps and the certification of their pointer representation. This will be useful to deal with the construction and correctness proof of 3D functional algorithms and imperative programs of computational geometry, like 3D Delaunay triangulation which is a very difficult problem.

At a low level, we will try to deal with more complex structures involving arrays and pointers, e.g. representing trees and graphs. Finally, we will establish

a precise link with separation logic, particularly to deal with pointer aliasing and memory collision/disjunction in general cases.

Acknowledgements

We would like to thank all the members of the French ANR project Galapagos, for the many discussions around proofs and the help for using the Coq system. Moreover, we are very grateful to the reviewers who highly helped to improve the paper. In particular, their incitation to study with orbits the proof of Floyd's tortoise and hare algorithm was especially fruitful.

References

[1] M.F. Barnsley and S. Demko. Iterated function systems and the global construction of fractals. In *The proceedings of the Royal Society of London*, 1985.

[2] J. Berdine, C. Calcagno, and B. Cook et al. Shape Analysis for Composite Data Structures. In *Computer-Aided Verification, CAV'07, LNCS 4590, Springer*, 2007.

[3] C. Berge. *Graphes et Hypergraphes.* Dunod, 1973 (2nd ed.).

[4] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Springer-Verlag, 2004.

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

[6] R. Bornat. Proving Pointer Programs in Hoare Logic. In *5th Conf. on Mathematics of Program Construction, MPC'00, LNCS 1837, Springer*, pages 102–126, 2000.

[7] C. Brun, J.-F. Dufourd, and N. Magaud. Designing and Proving Correct a Convex Hull Algorithm with Hypermaps in Coq. *Computational Geometry - Theory and Applications*, 45(8):436–457, 2012.

[8] R.M. Burstall. Some techniques for proving correctness of programs which alters data structures. *Machine Intelligence*, 7:23–50, 1972.

[9] David Cazier and Jean-François Dufourd. A formal specification of geometric refinements. *The Visual Computer*, 15(6):279–301, 1999.

[10] M. Ceska, P. Erlebach, and T. Vojnar. Generalized multi-pattern-based verification of programs with linear linked structures. *Formal Aspects of Computing*, 19(3):363–374, 2007.

[11] R. Cori. Un code pour les graphes planaires et ses applications. *Astérisque*, 27, 1970.

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

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

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

[15] J.-F. Dufourd. *Coq Orbit Library On-Line Development*. http://dpt-info.u-strasbg.fr/~jfd/ORBIT-LIB-PUBLI.tar.gz, 2013.

[16] J.-F. Dufourd. Hypermap Specification and Certified Linked Implementation using Orbits. In *Interactive Theorem Proving Conference ITP'14, LNCS, Springer*, 2014, *accepted for publication*.

[17] J.-F. Dufourd. Dérivation de l'Algorithme de Schorr-Waite par une Méthode Algébrique. In *JFLA'2012, INRIA, hal-00665909*, Carnac, Feb. 2012, 15 pages.

[18] J.-F. Dufourd and Y. Bertot. Formal Study of Plane Delaunay Triangulation. In *Interactive Theorem Proving Conference ITP'10, LNCS 6172, Springer*, pages 211–226, 2010.

[19] J.-F. Dufourd and F. Puitg. Functional specification and prototyping with oriented combinatorial maps. *Comput. Geom.*, 16(2):129–156, 2000.

[20] P. Kraemer et al. CGoGN: N-dimensional Meshes with Combinatorial Maps. In *22nd International Meshing Roundtable, Orlando, Springer Int. Publishing*, pages 485–503, 2013.

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

[22] H.B. Griffiths. *Surfaces*. Cambridge University Press, 1976.

[23] B. Hackett and R. Rugina. Region-Based Shape Analysis with Tracked Locations. In *32th ACM Symp. on Principles of Programming Languages, POPL'05*, pages 310–323, 2005.

[24] D. Hilbert. *Grundlagen der Geometrie*. Teubner, Leipzig, 1903.

[25] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

[26] Coq Development Team INRIA. *Coq V8.4 Reference Manual*. http://coq.inria.fr, 2014.

[27] D.E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[28] D.E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[29] P. Lienhardt. N-Dimensional Generalized Combinatorial Maps and Cellular Quasi-Manifolds. *Int. Journal of Computational Geometry and Appl.*, 4(3):275–324, 1994.

[30] N. Marti, R. Affeldt, and A. Yonezawa. Formal Verification of the Heap Manager of an Operating System Using Separation Logic. In *8th Int. Conf. on Formal Engineering Methods, ICFEM'06*, pages 400–419, 2006.

[31] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1-2):200–227, 2005.

[32] B. Mendelbrot. *Fractals: Form, Chance, and Dimension*. W. H. Freeman and Co, San Francisco, CA, Reading, UK, 1977.

[33] P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. *ACM Trans. on Programming Languages and Systems*, 31(3), 2009.

[34] J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Conf. on Logic in Computer Science, LICS'02*, pages 55–74, 2002.

[35] H. Schorr and W.R. Waite. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *CACM*, 10(8):501–506, 1967.

[36] CATIA Team. *CATIA-V6: 3D CAD Software*. Dassault-Systems, http://www.3ds.com, 2014.

[37] CGAL Development Team. *CGAL-4.3: CGAL Open Source Project*. https://www.cgal.org, 2014.

[38] W.T. Tutte. *Graph Theory - In Encyclopedia of Mathematics and its Applications*. Addison Wesley, Reading, MA, 1994.

[39] S. Warshall. A Theorem on Boolean Matrices. *JACM*, 9:11–12, 1962.

[40] Wikipedia. *Cycle Detection*. http://en.wikipedia.org/wiki/Cycle_detection/, 2014.