Formalization of a Surface Subdivision Allowing a Region with Holes without Coordinates

Kazuko Takahashi, Sosuke Moriguchi, and Mizuki Goto*

School of Science&Technology, Kwansei Gakuin University,
2-1, Gakuen, Sanda, 669-1337, JAPAN
ktaka@kwansei.ac.jp, chiguri@acm.org, izconnect705@gmail.com

Abstract. This paper discusses how a surface subdivision is formalized symbolically. We formalize a subdivision of a finite two-dimensional plane allowing a region with holes. We use a data structure called PLCA. A PLCA expression is defined using four constructors and represents a surface subdivision without coordinates. We show construction of a configuration for a subdivision and representation of a subdivision using PLCA. Formalization and proofs are performed within Coq proof assistant.

Keywords: surface subdivision, formalization, PLCA, planarity, Coq

1 Introduction

A number of studies have been undertaken previously on symbolic treatment of geometric or topological properties. Some of these studies used proof assistants to give a computational model representing spatial data, and to certify its formalization with a mechanical proof. They sometimes find drawbacks or oversights in pen-and-paper proofs.

Surface subdivision is a basic concept in computational geometry and topology. Intuitively, it is an embedding of a surface with a finite number of connected regions. There are a number of geometric or topological topics related to surface subdivision, for example, graph embedding, four color theorem and Jordan's simple curve theorem [1]. Surface subdivision determines locations of regions for a given set of points, and it commonly uses a region without holes such as Delaunay triangulation [1]. On the other hand, we consider a case in which a region with holes can be regarded as a subdivision. That is, we admit all configurations shown in Figure 1 as a subdivision of a plane. Configuration here means that a representation of a figure showing its geometric or topological characteristics. In these figures, a red part shows a region with a hole and a green part shows a region as a hole. When we admit such cases, the border of each region is not always a Jordan curve, that is, a simple closed curve without a self-loop [10] (e.g, Figure 1(c)(d)).

There are few symbolic expressions focusing on topological aspects without the use of coordinates and, to the best of our knowledge, none gives a mechanical proof of the algorithms for handling disconnected components.

* Currently, JFE Advantech Co., Ltd.

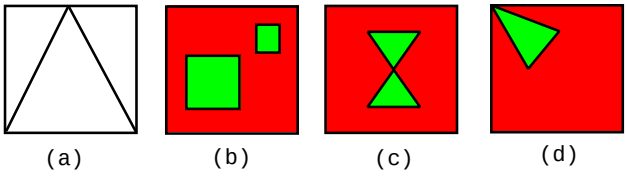


Fig. 1. Surface subdivision: (a) triangulation (no holes), (b) a region with holes, (c) a region with holes which are connected with a point, and (d) a region with a hole which is connected to itself with a point.

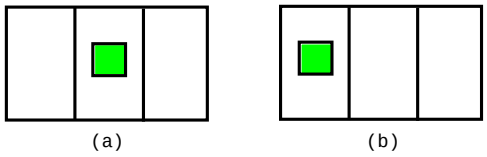


Fig. 2. Different configurations with a disconnected component.

Disconnected components can be regarded as regions that are embedded into the holes of another region. Therefore, when we admit disconnected components, we have to explicitly represent the region with holes in which the component is located, since no coordinate information is provided. For example, the two configurations shown in Figure 2 should be considered as being different, which makes formalization difficult.

In this paper, we describe formalization with PLCA to represent a surface subdivision that allows disconnected components. PLCA is a data structure in which all incidence relations between objects are stored [14]. Topological and geometric aspects can be distinguished in this structure. Previously, we have described its inductive construction and proved that the obtained class is planar [15]. We first gave a pen-and-paper proof and then a mechanical proof using a proof assistant Coq [2]. As a rigorous specification is required in a symbolic formalization, it was necessary to solve subtle problems that can be ignored in a pen-and-paper proof [11].

In these studies, "planarity" means not only embedding a PLCA expression in a two-dimensional plane, but also forming a subdivision of a plane. However, we did not refer to subdivision explicitly, nor did we explain in detail the meaning of the preconditions of each constructor; nevertheless, those are significant factors in the symbolic treatment of geometric/topological data. In this paper, we describe construction of PLCA from the viewpoint of subdivision, and how conditions for surface subdivision are represented with PLCA.

We consider a surface subdivision as a configuration in which both side of each line always belong to distinct regions. It means that there are no isolated points or isolated lines, and no bridges (Figure 3). In addition, a configuration

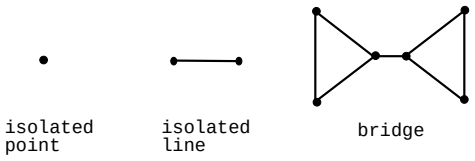


Fig. 3. Disallowed configurations of a subdivision

can include a region with holes as a piece of subdivision and regions connected with a point, which are shown in Figure 1. We represent these conditions in the form of PLCA planarity. We give a more specific expression for a subdivision and show that a planar PLCA forms a subdivision. Our final goal is to implement computational geometry algorithms related to surface subdivision in PLCA, based on the formalization described in this paper.

All the formalization is implemented in Coq. Although we will avoid describing the messy Coq code here, the entire formalization and proofs have been uploaded to [12].

The remainder of this paper is organized as follows. In Section 2, we describe a data structure of PLCA. In Section 3 we show how a surface subdivision is constructed with PLCA in Coq. In Section 4, we discuss how surface subdivision is represented in our formalization. In Section 5 we compare our work with related work, and Section 6 concludes the paper.

2 PLCA

2.1 PLCA Expression

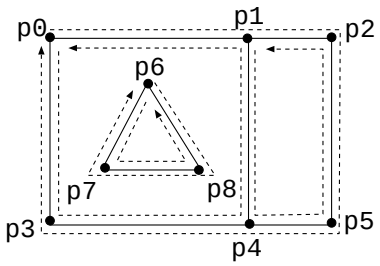
A PLCA data structure was originally designed to give a qualitative representation of a spatial object. The term "qualitative" implies no use of numerical data such as coordinates, sizes or ratios etc.; instead, we perform reasoning focusing on certain characteristics of a spatial object. So far, several qualitative spatial representations have been proposed [13, 4]. Among them, PLCA is designed to distinguish connection patterns of regions [14].

Definition 1 (PLCA expression). *A PLCA expression is defined as a five-tuple,* $\langle P, L, C, A, o\rangle$*, where* P *is a set of points,* $L \subseteq P^2$*,* $C \subseteq L^n$ *(*$n \geq 3$*),* $A \subseteq C^m$ *(*$m \geq 1$*),* $o \in C$*.*

In PLCA, there are four kinds of object: points, lines, circuits, and areas.

- Points are the most primitive objects. Points are distinguishable from each other. We use p as a variable for points.
- A line represents segments between two points. It is defined as a pair of distinct points, like (p_0, p_1)[1]. We use l as a variable for lines. Each line has

[1] The original PLCA allows curved lines. Our definition does not allow (p, p) as a line.



$$P = \{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$$
$$L = \{l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9\}$$
$$C = \{o, c_0, c_1, c_2, c_3\}$$
$$A = \{a_0, a_1, a_2\}$$

$$l_0 = (p_0, p_1) \quad c_0 = [l_2^+, l_5^+, l_3^-, l_0^-]$$
$$l_1 = (p_1, p_2) \quad c_1 = [l_3^+, l_6^+, l_4^-, l_1^-]$$
$$l_2 = (p_0, p_3) \quad c_2 = [l_9^-, l_8^-, l_7^-]$$
$$l_3 = (p_1, p_4) \quad c_3 = [l_7^+, l_8^+, l_9^+]$$
$$l_4 = (p_2, p_5) \quad o = [l_0^+, l_1^+, l_4^+, l_6^-, l_5^-, l_2^-]$$
$$l_5 = (p_3, p_4)$$
$$l_6 = (p_4, p_5) \quad a_0 = \{c_0, c_2\}$$
$$l_7 = (p_6, p_7) \quad a_1 = \{c_1\}$$
$$l_8 = (p_7, p_8) \quad a_2 = \{c_3\}$$
$$l_9 = (p_8, p_6)$$

Fig. 4. An example of a PLCA expression.

a direction from the first to the second element of the pair. The direction of a line l is denoted by l^+, and its inverse direction by l^-.
- A circuit represents a closed outline. It is defined as a list of lines, like $[l_0, l_1, \ldots, l_n]$. Each circuit is closed, i.e., the first element of the first element l_0 and the second element of the last element l_n are the same. We use c as a variable for circuits.
- An area represents a region enclosed with circuits. It is defined as a set of circuits, like $\{c_0, c_1, \ldots, c_n\}$. We use a as a variable for areas.

Additionally, we use a specific circuit in the outermost side of the figure denoted by *outermost*. We use o as a variable for *outermost*.

We show an example of a PLCA expression and an instance of its corresponding configuration in a two-dimensional plane in Figure 4. Note that this expression has three areas: area a_1 is an area without a hole, area a_0 is an area with a hole and area a_2 fits in that hole.

2.2 Equivalence on objects

We use a list as a data type to implement P, L, C, A, each line, circuit and area, as our first implementation in Coq. It causes us to define the equivalence relation on lists [2].

[2] If we use data type from a Coq library, for example, $MSet$, then we may omit these definitions.

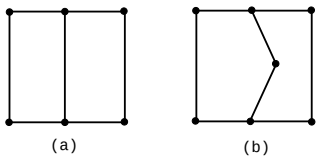


Fig. 5. Same subdivision, different data structures.

Hereafter, the symbol '+' denotes an operation of list contcatenation. For example, when $V = [v, w]$, $[u] + V$ returns $[u, v, w]$.

Two symbolic expressions that are not identical may stand for the same configuration. For example, circuits $[l_1, l_2, l_3]$ and $[l_2, l_3, l_1]$ represent the same circuit, and areas $[c_1, c_2, c_3]$ and $[c_3, c_2, c_1]$ represent the same area. This is due to the cyclic structure of a circuit, and the data structure of a list for an area. In this case, we should regard them as equivalent.

On the other hand, we have to distinguish between two expressions when they stand for the same subdivision but have different configurations. For example, the expressions for the two configurations shown in Figure 5 should be distinguished, since the number of points and lines are different.

Considering the above points, we define an equivalence relation over PLCA expressions.

As P, L, C and A are defined as lists, they are equivalent to permutations of themselves. Each element of L is equivalent to its inverse, and each element of C is equivalent to its rotation.

Definition 2 (PLCA-equivalence). *Two expressions are* PLCA-equivalent *iff the pair of them is in the transitive closure of the following binary relation* $\mathcal{R}_{eq}$*.*

- $\langle P, [l^+] + L, C, A, o\rangle \ \mathcal{R}_{eq} \ \langle P, [l^-] + L, C, A, o\rangle$.
- $\langle P, L, [c] + C, [[c] + a] + A, o\rangle \ \mathcal{R}_{eq} \ \langle P, L, [c'] + C, [[c'] + a] + A, o\rangle$*, where* c' *is a rotated circuit of* c*.*
- $\langle P, L, C, A, o\rangle \ \mathcal{R}_{eq} \ \langle P', L', C', A', o\rangle$*, where* P'*,* L'*,* C' *and* A' *are permutations of* P*,* L*,* C *and* A*, respectively.*
- $\langle P, L, C, [a] + A, o\rangle \ \mathcal{R}_{eq} \ \langle P, L, C, [a'] + A, o\rangle$*, where* a' *is a permutation of* a*.*
- $\langle P, L, [o] + C, A, o\rangle \ \mathcal{R}_{eq} \ \langle P, L, [o'] + C, A, o'\rangle$*, where* o' *is a rotated circuit of* o*.*

We call the above relation PLCA-equivalence.

Note that $\mathcal{R}_{eq}$ is reflexive because of the third condition of $\mathcal{R}_{eq}$ and the reflexivity of permutations. Therefore, PLCA-equivalence is reflexive and transitive.

2.3 PLCA consistency

A PLCA expression is too permissive to find a corresponding topological space. For example, if there exists more than one area that contain the same circuit,

such an expression does not make sense. And we should also avoid duplication in a list. Therefore, we put consistency on this data structure so that it makes sense.

Definition 3 (consistent PLCA). *Let $\langle P, L, C, A, o\rangle$ be a PLCA expression. The PLCA expression is said to be* a consistent PLCA expression *iff it satisfies all of the following conditions.*

- *For each point $p \in P$, there exists $l \in L$ such that $p \in l$.*
- *For each line $l \in L$, all points in l are in P.*
- *For each line $l \in L$, there exist circuits $c, c' \in C$ such that $l^+ \in c$ and $l^- \in c'$*[3]*.*
- *For each circuit $c \in C$, each line in c or its inverse line is in L.*
- *For each circuit $c \in C$ except for o, there exists an area $a \in A$ such that $c \in a$.*
- *For each area $a \in A$, all circuits in a are in C.*
- *For any area $a \in A$, $o \notin a$.*
- *Each point $p \in P$ appears only once in P.*
- *Each line $l^+ \in L$ appears only once in L and $l^- \notin L$.*
- *Each circuit $c \in C$ appears only once in C and any rotated circuits other than c do not appear in C.*
- *Each area $a \in A$ appears only once in A and any equivalent area other than a does not appear in A.*

3 Encoding PLCA

We show PLCA specification in Coq.

3.1 Construction of PLCA

We construct PLCA so that its configuration is a subdivision of a finite two-dimensional plane allowing a region with holes. We use four constructors, and explain each constructor according to the meaning of their preconditions.

Initially, we make a finite area, then apply constructors sequentially. We divide an existing area into two areas and as a result, the number of areas is incremented by exactly one each time a constructor is applied. At this time, we either add a connected component or a disconnected component depending on whether we cut an existing circuit or not. We introduce a new object *path* to divide an area. As PLCA has no coordinate information, we have to specify the area in which such a path is added. As a symbolic operation, we delete one existing area and add two new areas, and also reconfigure all related objects. The difficulty on re-composition of a circuit is that we have to find the beginning of a list, since a circuit has a cyclic structure.

[3] In the formalization in Coq, we define this condition as "for each line l such that $l^+ \in L$ or $l^- \in L$, there exists the circuit c such that $l \in c$." These conditions are the same.

3.2 Objects

Objects in PLCA are defined as follows, where `nat` denotes a natural number and `prod` denotes a product of the two specified arguments.

```
Definition Point := nat.
Definition Line := prod Point Point.
Definition Circuit := list Line.
Definition Area := list Circuit.
```

3.3 Basics

Here we provide some definitions of basic functions. Functions *reverse* and *reverse_linelist* are prepared to make the reverse circuit (lists of lines), i.e., reverse_linelist$([l_1^+, l_2^+, ..., l_n^+]) = [l_n^-, ..., l_2^-, l_1^-]$). Proposition *Rot* is defined on a pair of circuits to check whether one circuit is obtained by rotating the other, i.e., Rot$([l_1^+, l_2^+, ..., l_n^+], [l_2^+, l_3^+, ..., l_n^+, l_1^+])$ is *true*, which is used to define PLCA-equivalence. InL is a proposition that checks whether a line or its inverse is in L.

```
Definition reverse(l : Line) := (snd l, fst l).

Definition reverse_linelist(ll : list Line) := rev (map reverse ll).

Inductive Rot : list A -> list A -> Prop :=
| rotSame : forall (l : list A), Rot l l
| rotNext : forall (l l' : list A) (a : A), Rot l (a :: l')
    -> Rot l (l' ++ [a]).

Definition InL (l: Line)(L: set Line) : Prop :=
  set_In l L \/ set_In (reverse_linelinst l) L.
```

PLCA-equivalence is defined as a proposition on a pair of lists of points, a pair of lists of lines, a pair of lists of circuits, a pair of lists of areas, and a pair of circuits. Each condition of Definition 2 is encoded. For example, the last condition, showing that the rotated outermost is equivalent to the original one, is encoded as follows: assume that one PLCA expression has `C` as a list of circuits and `o` as *outermost*, and the other PLCA expression has `C'` as a list of circuits and `o'` as *outermost*; if `o'` is a rotated circuit of `o`, and `C'` also contains `o'` instead of `o`, then these PLCA expressions are equivalent.

```
rotateOutermost : forall(P : list Point)(L : list Line)
                        (C : list Circuit)(A : list Area)
                        (o o' : Circuit),
   Rot o o'
   -> PLCAequivalence P L C A o P L (map (replace_circuit o o') C) A o'
```

We define *a path* as a sequence of lines with non-negative length. A path does not have a point that appears more than once. We define another object *trail* that is allowed to have a point that appears more than once but is not allowed to have a line that appears more than once. A path is used as a divider of a circuit to make a new area, while a trail shows a property of a segment of a circuit. Note that a circuit is not always a Jordan curve. We can make a closed path (or trail) by adding a line connecting their start and end points, which corresponds to a circuit.

Definition 4 (path). *A list of lines* $[(p_0, p_1), (p_1, p_2), \ldots, (p_{n-1}, p_n)]$ $(n > 0)$ *is said to be* a path *if* $p_0, \ldots, p_n$ *are distinct.*

Definition 5 (trail). *A list of lines* $[(p_0, p_1), (p_1, p_2), \ldots, (p_{n-1}, p_n)]$ $(n > 0)$ *is said to be* a trail, *if there exists no* i, j $(0 < i \neq j < n)$ *such that* $(p_{i-1}, p_i) = (p_{j-1}, p_j)$ *or* $(p_{i-1}, p_i) = (p_j, p_{j-1})$.

For example, in Figure 6, a path and a trail are depicted by a red line and a green line, respectively. Circuits in Figure 6(a) are both closed paths, while circuits in Figure 6(b)(c) are two closed paths and one closed trail.

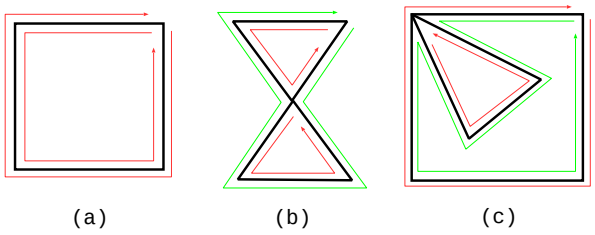


Fig. 6. Circuits constructed by paths (red) and trails (green): (a) two closed paths (b)(c) two closed paths and one closed trail.

An object trail is inductively defined as follows. The arguments show the starting point, the ending point, inner points and inner lines. As a base case, make a nil_trail that consists of a single point without lines [4]. Make a step_trail by adding a point that is not included in the existing inner points, and a line connecting to the new point and the ending point of the existing path. The function also checks that the new line is not included in the existing inner lines.

```
Inductive trail : Point -> Point -> list Point -> list Line -> Prop :=
| nil_trail  : forall(p : Point), trail p (p::nil) nil
| step_trail : forall(p1 p2 p3 : Point)(pl : list Point)(ll : list Line),
```

[4] A data structure $[(p, p)]$ is taken as a base case in this implementation, which is not allowed in the definition of a trail or path. And when trail or path is used in a constructor in the implementation, condition on the length is added.

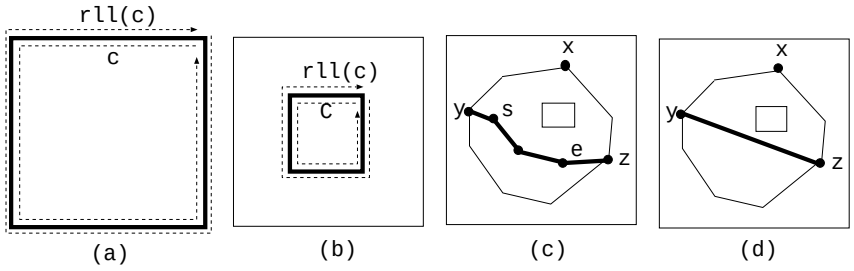


Fig. 7. Constructors: (a) single_loop, (b) add_loop, (c) add_inpath, and (d) add_inline. In these figures, 'rll' denotes *reverse_line_list*.

```
   trail p2 p1 pl ll
-> p3 <> p2
-> ~LIn (p3, p2) ll
-> trail p3 p1 (p3::pl) ((p3, p2)::ll).
```

3.4 Constructors

We show four constructors together with their preconditions: *single_loop*, *add_loop*, *add_inpath* and *add_inline*. A path is added in the inner side of the *outermost*. In the previous work, we consider another constructor *add_outpath* that adds a path on the outer side of the *outermost* [11]. In that case, we allow a trail as an *outermost*. We can construct PLCA that admits such a configuration, but when focusing on subdivision, it is not suitable to consider such a case. Therefore, the definition is little different from the one shown in the previous work.

single_loop: Constructor *single_loop* is an initialization that corresponds to add a Jordan curve to make *outermost*. It makes a simple path and adds a new line that connects its start and end points (Figure 7(a)).

The preconditions are simple. (1) The length of a path is more than two to make directions of its inner lines deterministic. (2) Points in the path should not be existing ones.

add_loop: Constructor *add_loop* adds a Jordan curve to the specified area. It makes a simple path and adds a new line that connects its start and end points (Figure 7(b)).

The preconditions are the same as those of *single_loop*. (1) The length of a path is more than two to make directions of its inner lines deterministic. (2) Points in the path should not be existing ones.

A new configuration is obtained after putting the closed path onto the existing area.

add_inpath: Constructor *add_inpath* is rather complicated. This constructor corresponds to the division of an area by cutting two points of the same existing circuit that belongs to that area (Figure 7(c)).

The preconditions are as follows. (1) The length of a path is more than zero to make directions of its inner lines deterministic. (2) Points in the path should not be existing ones. (3) Two points (that may be the same) connected to the start and end points of the path should be in the same circuit to avoid a bridge.

A new configuration is obtained after making an area by re-composing an existing circuit with a new path. In this case, at least two lines are added to L.

add_inline: Constructor *add_inline* is similar to *add_inpath* but simpler, since no new points are added. Intuitively, it makes a shortcut between the two existing, different points in the same circuit (Figure 7(d)).

The precondition is: (1) There is no line that connects two different points in the same circuit to guarantee planarity.

A new configuration is obtained after making an area by re-composing an existing circuit with a new line. In this case, only one line is added to L.

3.5 Inductive PLCA

The four described constructors are formalized below. A PLCA expression constructed in this way is called *an inductive PLCA expression.* In the following, 'rll' denotes a function *reverse_linelist*.

single_loop: Let $p_1, p_2, ..., p_n$ $(n \geq 3)$ be distinct points, $P = [p_1, p_2, ..., p_n]$, $L = [(p_n, p_1), (p_1, p_2), ..., (p_{n-1}, p_n)]$, $c = [(p_n, p_1), (p_1, p_2), ..., (p_{n-1}, p_n)]$, $C = [c, \mathrm{rll}(c)]$ and $A = [[c]]$. Then, $\langle P, L, C, A, \mathrm{rll}(c)\rangle$ is an inductive PLCA expression.

add_loop: Let $\langle P, L, C, A, o\rangle$ be an inductive PLCA expression. If $p_1, p_2, ..., p_n$ $(n \geq 3)$ are distinct, p_i for any i does not appear in P, $c = [(p_n, p_1), (p_1, p_2), ..., (p_{n-1}, p_n)]$ and $a \in A$, then $\langle P', L', C', A', o\rangle$ is an inductive PLCA expression, where P', L', C', A' are as follows.

- $P' = [p_1, ..., p_n] + P$
- $L' = [(p_n, p_1), (p_1, p_2), ..., (p_{n-1}, p_n)] + L$
- $C' = [c, \mathrm{rll}(c)] + C$
- $A' = [[c]] + [[\mathrm{rll}(c)] + a] + (A - [a])$

add_inpath: Let $\langle P, L, C, A, o\rangle$ be an inductive PLCA expression. Suppose that $s, p_1, ..., p_n, e$ are distinct, s, e and p_i for any i do not appear in P. Let $ap = [s, p_1, ..., p_n, e]$, $al = [(s, p_1), ..., (p_n, e)]$. Note that if $s = e$, then $ap = [s]$ and $al = [\,]$. Also suppose that $c \in a$, $a \in A$, and $c = lxy + lyz + lzx$ where lxy, lyz and lzx start with points x, y and z and end with points y, z and x, respectively. At least one of y and z or s and e are different. $\langle P', L', C', A', o\rangle$ is an inductive PLCA expression, where P', L', C', A' are as follows.

- $P' = ap + P$

- $L' = [(y, s), (e, z)] + al + L$
- $C' = [[(s, y)] + lyz + [(z, e)] + \mathrm{rll}(al), lxy + [(y, s)] + al + [(e, z)] + lzx] + (C - [c])$
- $A' = [[[(s, y)] + lyz + [(z, e)] + \mathrm{rll}(al)], [lxy + [(y, s)] + al + [(e, z)] + lzx] + (a - [c])] + (A - [a])$

add_inline: Let $\langle P, L, C, A, o\rangle$ be an inductive PLCA expression. Suppose that $c \in a$, $a \in A$, and $c = lxy + lyz + lzx$ where lxy, lyz and lzx start with points x, y and z and end with points y, z and x respectively. If either (y, z) or its reverse line is not in L, $\langle P, L', C', A', o\rangle$ is an inductive PLCA expression, where L', C', A' are as follows.

- $L' = [(y, z)] + L$
- $C' = [[(z, y)] + lyz, lxy + [(y, z)] + lzx] + (C - [c])$
- $A' = [[[(z, y)] + lyz], [lxy + [(y, z)] + lzx] + (a - [c])] + (A - [a])$

Figure 8 shows examples of an area division by constructor *add_inpath*. We put a path on the specified area and connect its start and end points to the circuit. Note that y and z are the cutting points of an existing circuit, while x is the starting point of a circuit. Since the circuit has a cyclic structure, and the rotation of a cycle is equivalent to the original one, reconfiguration of a cycle begins with the original starting point. Figure 8(a) is a case to put a path on the area without a hole, Figure 8(b) is a case to put a path on the area with a hole, and Figure 8(c) is a case in which y and z are the same point, which corresponds to adding a loop to the existing point.

We show the corresponding Coq specification of each constructor in the Appendix.

4 Representation of Subdivision

We show that inductive PLCA represents a surface subdivision of a finite two-dimensional plane that allows a region with holes as a subdivision.

The condition for subdivision is represented in a form of PLCA planarity which consist of three properties: PLCA-constraint, PLCA-connectedness and PLCA-euler.

Definition 6 (PLCA-constraints). *Let $\langle P, L, C, A, o\rangle$ be a PLCA expression. It is said that the expression satisfies* PLCA-constraints *iff it satisfies all of the following conditions.*

1. *For each line $(p, p') \in L$, $p \neq p'$.*
2. *For each circuit $c \in C$, $|c| \geq 3$ and c is a closed trail.*
3. *For each area $a \in A$,*
   - *if $c, c' \in a$ and $c \neq c'$, then c and c' have no shared points or lines.*
   - *if $c \in a$, then c appears in a only once and no rotated circuit of c appear in a.*
   - $|a| \geq 1$.
4. *o is a closed path.*

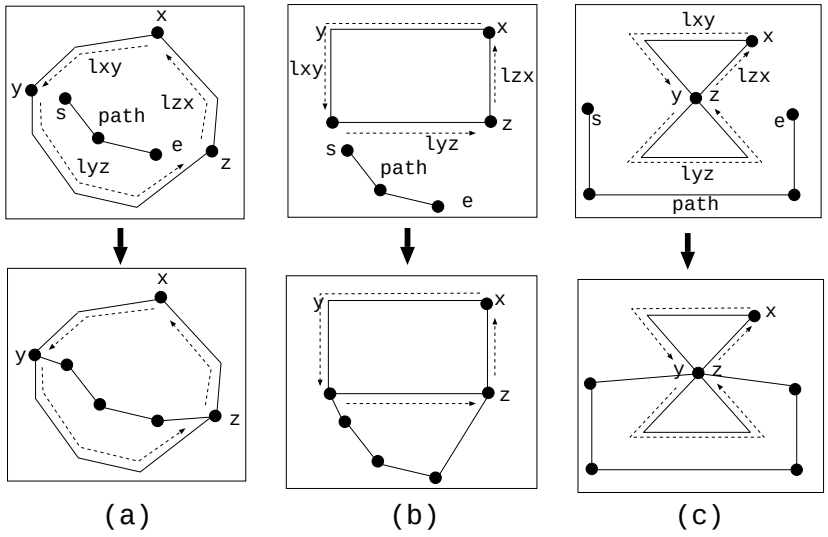


Fig. 8. Examples of dividing an area by *add_inpath*: (a) put a path on the area without a hole, (b) put a path on the area with a hole, (c) two points in the circuit are the same.

The first condition of PLCA-constraints guarantees that there is no isolated point, and the second condition guarantees that there is no bridge between points. The third condition is for handling duplicated points in a single circuit. For example, in Figure 9, a circuit $[(p_0, p_1), (p_1, p_2), (p_2, p_3), (p_3, p_1), (p_1, p_4), (p_4, p_5), (p_5, p_0)]$, that passes point p_1 more than once, is included only in the green area. Note that it is not a combination of two circuits. The fourth condition is to guarantee that our target plane to be divided is a finite plane encircled with a Jordan curve.

```
Definition PLCAconstraints(L : list Line)(C : list Circuit)
                          (A : list Area)(o : Circuit) : Prop :=
  Lconstraint L /\ Cconstraint C /\ Aconstraint A
                /\ Circuit_constraints o.
```

Each constraint is defined as a proposition. For example, the second condition of Definition 6 is encoded as follows:

```
Definition Circuit_constraints(c : Circuit) : Prop :=
 (exists x : Point, exists pl : list Point, trail x x pl c)
  /\ 3 <= length c.
Definition Cconstraint(C : list Circuit) : Prop :=
  forall(c : Circuit), In c C -> Circuit_constraints c.
```

Definition 7 (PLCA-connect). *Let $\langle P, L, C, A, o\rangle$ be a PLCA expression. A pair of objects in the expression is said to be* PLCA-connect *iff it is in the symmetric transitive closure of the following relation $\mathcal{R}_c$.*

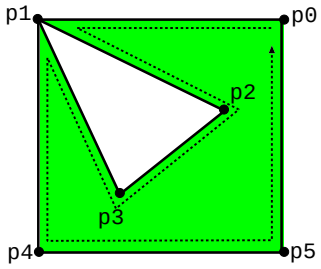


Fig. 9. Treatment of circuit including a duplicated point.

1. *If* $p \in P$, $l \in L$ *and* $p \in l$, *then* $p \mathcal{R}_c\, l$.
2. *If* $l^+ \in L$ *or* $l^- \in L$, $c \in C$ *and* $l^+ \in c$, *then* $l^+ \mathcal{R}_c\, c$.
3. *If* $c \in C$, $a \in A$ *and* $c \in a$, *then* $c \mathcal{R}_c\, a$.

Definition 8 (PLCA-connectedness). *It is said that a PLCA expression satisfies* PLCA-connectedness *iff any pair of objects in the expression are PLCA-connected.*

Intuitively, PLCA-connectedness guarantees that no objects are separated, including the *outermost*. Each object is traceable from *outermost*.

Representing PLCA-connectedness has one difficulty: this property holds over the different types of P, L, C and A. Therefore, we have to convert these data types to a common data type *object*, and prepare a new incidence function OIn.

```
Inductive object :=
| o_point   : Point   -> object
| o_line    : Line    -> object
| o_circuit : Circuit -> object
| o_area    : Area    -> object.

Definition OIn (o : object) : Prop :=
 match o with
| o_point p   => In p P
| o_line l    => In l L
| o_circuit c => In c C
| o_area a    => In a A
 end.
```

Then, PLCA-connectedness is implemented as follows.

```
Inductive PLCAconnect : object -> object -> Prop :=
| PLcon : forall(p : Point)(l : Line),
    In p P  -> In l L -> InPL p l ->
      PLCAconnect (o_point p) (o_line l)
| LCcon : forall(l : Line)(c : Circuit),
    In l L  -> In c C  -> LIn l c  ->
```

```
      PLCAconnect (o_line l) (o_circuit c)
| CAcon : forall(c : Circuit)(a : Area),
    In c C  -> In a A  -> In c a  ->
      PLCAconnect (o_circuit c) (o_area a)
| SYMME : forall(o1 o2 : object),
      PLCAconnect o1 o2 -> PLCAconnect o2 o1
| TRANS : forall(o1 o2 o3 : object),
      PLCAconnect o1 o2 -> PLCAconnect o2 o3 -> PLCAconnect o1 o3.

Definition PLCAconnected : Prop :=
 forall(o1 o2 : object), OIn o1 -> OIn o2 -> PLCAconnect o1 o2.
```

Definition 9 (PLCA-euler). *Let* $\langle P, L, C, A, o\rangle$ *be a PLCA expression. It is said that the expression satisfies* PLCA-euler *iff* $|P| - |L| - |C| + 2|A| = 0$.

PLCA-euler guarantees that a PLCA expression can be embedded in a two-dimensional plane so that the orientation of each circuit can be correctly defined.

Coq implementation is straightforward.

```
Definition PLCAeuler : Prop :=
  length P + 2 * length A = length L + length C.
```

Definition 10 (planar PLCA). *If a consistent PLCA satisfies PLCA-constraints, PLCA-connectedness, and PLCA-euler, then it is said to be* a planar PLCA*.*

The following theorem shows that inductive PLCA satisfies planarity.

Theorem 1. *Inductive PLCA is a planar PLCA.*

This is proved by proving the following three lemmas.

```
Lemma PLCA_consistency_and_constraints :
forall(P : list Point)(L : list Line)(C : list Circuit)(A : list Area)
      (o : Circuit),
 I P L C A o -> PLCAconsistency P L C A o /\ PLCAconstraints L C A o.

Lemma PLCAconnectedness :
forall(P : list Point)(L : list Line)(C : list Circuit)(A : list Area)
  (o : Circuit),
 I P L C A o ->
   (forall(p : Point)(l : Line),
     In p P -> In l L -> PLCAconnect P L C A (o_point p) (o_line l))
/\ (forall(l : Line)(c : Circuit),
     In l L -> In c C -> PLCAconnect P L C A (o_line l) (o_circuit c))
/\ (forall(c : Circuit)(a : Area),
     In c C -> In a A -> PLCAconnect P L C A (o_circuit c) (o_area a)).

Lemma PLCAeuler :
forall(P : list Point)(L : list Line)(C : list Circuit)(A : list Area)
      (o : Circuit),
 I P L C A o -> PLCAeuler P L C A.
```

The PLCA-equivalence preserves PLCA planarity.

Theorem 2. *Let e, e' be equivalent PLCA expressions. If e is a planar PLCA then e' is also a planar PLCA.*

We define that an expression of subdivision mentioned in Section 1 as follows. It is defined depending on the case whether a line is included in *outermost* or not. Intuitively, this definition means that for each line, if it is not in *outermost*, then distinct two areas are connected by this line; otherwise, there exists only one area that is connected with the outside of the figure by this line.

Definition 11 (subdivision). *Let $e = \langle P, L, C, A, o\rangle$ be a consistent PLCA expression. For each $l \in L$, if the following condition is satisfied, then e is said to be* a subdivision*: (i) there exist areas $a, a' \in A$ such that $a \neq a'$, $l^+ \in c \in a$, and $l^- \in c' \in a'$, for $l \notin$ outermost, and (ii) there exists an area a and a circuit c such that $l \in c \in a$, $c \neq$ outermost, for $l \in$ outermost.*

Finally, we get the following theorem that means a planar PLCA forms a subdivision.

Theorem 3. *A planar PLCA is a subdivision.*

It is proved by the following theorems by case split depending on whether a line is included by the outermost.

```
Theorem PLCAsubdivision_area :
forall(P : list Point)(L : list Line)(C : list Circuit)(A : list Area)
  (o : Circuit),
 PLCA_planar P L C A o ->
 forall (l : Line),
  In l L
  -> ~LIn l o
  -> exists (c c' : Circuit),
   In c C
   /\ In c' C
   /\ In l c
   /\ In (reverse l) c'
   /\ exists (a a' : Area),
       In a A
       /\ In a' A
       /\ In c a
       /\ In c' a'
       /\ a <> a'.

Theorem PLCAsubdivision_outermost :
forall(P : list Point)(L : list Line)(C : list Circuit)(A : list Area)
  (o : Circuit),
 PLCA_planar P L C A o ->
 forall (l : Line),
  LIn l o
```

```
-> exists (c : Circuit),
 In c C
 /\ LIn l c
 /\ o <> c
 /\ exists (a : Area),
     In a A
     /\ In c a.
```

5 Related Works

Hypermap is a well-known data structure for handling topological aspects of spatial data; it is an algebraic data structure that consists of a set of darts and two permutations. Hypermap uses a dart as a primitive, and an edge and a vertex are defined as compositions of darts. Circuit and connected components are calculated by permutations. Hypermap is a useful data structure but it is hard to envisage, for example, a figure embedded in a two-dimensional plane, since vertices, edges and faces are not represented directly. On the other hand, PLCA provides a more straightforward expression of a figure.

There are several works that give a formalization based on hypermap and a proof of geographic/topological properties using proof assistants. Brun et al. showed a derivation of a program to compute a convex-hull for a given set of points from their specification [3]. They specified the algorithm and proved its correctness using a structural induction. Gonthier showed the four color theorem based on surface subdivision [9]. However, they did not treat a disconnected component. Dufourd discusses that embedding polyhedra onto a plane is a sufficient condition for planarity [6]. In that work, a hypermap was applied to formalize a discrete version of a Jordan curve and to prove its soundness [7]. Dufourd's most relevant work is a formalization of an Delaunay triangulation [8]. They represented Delaunay triangulation in a form where all edges have two darts and all faces have three vertices. A coordinate was used to represent a proper triangle, and disconnected components were not handled. Our formalization on subdivision includes disconnected components.

Besides a hypermap, Yamamoto et al. studied a graph embedded on a plane using HOL [16]. That model is simpler since it does not handle regions. There are few studies on qualitative spatial representation from the viewpoint of theorem proving. Recently, an interesting work is done by Dapoigny et al. [5]. They formalized Tarski's mereogeometry using Coq. The representation is based on a part-hole relationship between regions, which is completely different from that of PLCA.

The doubly connected edge list is a popular data structure in computational geometry for representing a surface subdivision allowing disconnected components [1]. It contains records for each face, edge, vertex of the subdivision. Each edge is regarded as two half-edges bounding different faces, and each half-edge has a pointer to its connecting half-edges as well as its twin. Comparing its data structure with PLCA, an edge (or a line) is represented in relating to the two faces that share it in both data structures; however, incidence relations of

objects are represented as pointers in the record in doubly connected edge list, whereas an object is explicitly defined using other existing objects in PLCA; the location of each object is determined uniquely by the coordinates in doubly connected edge list, which is different from our work. Moreover, the main goal of computational geometry is to develop an algorithm that can reduce computational complexity, whereas here, we investigate the soundness of an algorithm. The method proposed can handle geometric data on a more abstract level, and can therefore be applied to prove the correctness of algorithms including doubly connected edge lists.

6 Conclusion

We have shown a symbolic representation with PLCA for a surface subdivision of a finite two-dimensional plane that allows holes in a region. All the formalization and proofs are performed in Coq and consist of about 12,800 lines [12].

In future work, following issues are considered:

- To specify other intuitive definition of planarity and prove the equivalence with PLCA-Euler.
- To implement high level computational geometry algorithms such as Delaunay triangulation or graph embedding algorithms in PLCA and prove them.
- To extend the proposed approach to handle more general surfaces such as polyhedra or torus with an arbitrary genus.

References

1. de Berg, M., van Kreveld, M., Overmars, M. and Schwarzkopf, O.: *Computational Geometry*, Springer-Verlag (1997).
2. Bertot, Y. and Castéran, P.: *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, Springer Verlag (1998).
3. Brun, C., Dufourd, J. -F. and Magaud, N,: Designing and Proving Correct a Convex Hull Algorithm with Hypermaps in Coq. *Computational Geometry : Theory and Applications*, 45(8):436-457 (2012).
4. Cohn, A. G. and Renz, J.: Qualitative Spatial Reasoning *Handbook of Knowledge Representation, F. Harmelen, V. Lifschitz and B. Porter(eds.)*, Chapt 13, pp.551-596, Elsevier (2007).
5. Dapoigny, R. and Barlatier, P.: A Coq-Based Axiomatization of Tarski's Mereogeometry, *12th Conference on Spatial Information Theory (COSIT2015)*, LNCS, vol. 9368, pp. 108–129, Springer-Verlag (2015).
6. Dufourd, J. -F.: Polyhedra Genus Theorem and Euler Formula: A hypermap-formalized intuitionistic proof, *Theoretical Computer Science*, 403(2-3):133-159, Elsevier (2008).
7. Dufourd, J. -F.: 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).
8. Dufourd, J. -F. and Bertot, Y.: Formal Study of Plane Delaunay Triangulation. *1st International Conference on Interactive Theorem Proving (ITP2010)*, LNCS, vol. 6172, pp. 211–226, Springer-Verlag (2010).

9. Gonthier, G.: Formal Proof - The Four Color Theorem. *Notices of the AMS*, 55(11):1382-1393 (2008).
10. Kosniowski, C.: *A First Course in Algebraic Topology*, Cambridge University Press (1980).
11. Moriguchi, S., Goto, M., Takahashi, K.: Towards Verified Construction for Planar Class of a Qualitative Spatial Representation. *7th International Symposium on Symbolic Computation in Software Science (SCSS2016)*, EPiC Series in Computing, vol.39, pp.117-129 (2016).
12. Moriguchi, S., Goto, M., Takahashi, K.: Formalization of IPLCA. `http://ist.ksc.kwansei.ac.jp/~ktaka/IPLCA2016Apr`
13. Stock, O. (Ed.): *Spatial and Temporal Reasoning*. Kluwer Academic Publishers (1997).
14. Takahashi, K., Sumitomo, T. and Takeuti, I.: On Embedding a Qualitative Representation in a Two-Dimensional Plane. *Spatial Cognition and Computation*, 8(1-2):4-26 (2008).
15. Takahashi, K., Goto, M. and Miwa, H.: Construction of a Planar PLCA Expression: A Qualitative Treatment of Spatial Data. *Agents and Artificial Intelligence*, LNCS, vol. 9494, pp. 298–315, Springer-Verlag (2015).
16. Yamamoto, M., Nishizaki, S., Hagiya, M. and Toda, Y.: Formalization of Planar Graphs. *Higher Order Logic Theorem Proving and Its Applications*, LNCS, vol.971, pp. 369–384, Springer-Verlag (2005).

Appendix

Coq specification of each constructor.

```
Inductive I : list Point -> list Line -> list Circuit -> list Area
           -> Circuit -> Prop :=
| single_loop : forall(pl : list Point)(ll : list Line)(x y : Point),
   simplepath x y pl ll
-> x <> y
-> 2 <= length ll
-> I pl ((y,x)::ll) (((y,x)::ll)::(reverse_linelist ((y,x)::ll))::nil)
     ((((y,x)::ll)::nil)::nil) ((reverse_linelist ((y,x)::ll)))

| add_inline : forall(P : list Point)(L : list Line)
             (C : list Circuit)(A : list Area)(o : Circuit)
             (a : Area)(x y z : Point)(pxy pyz pzx : list Point)
             (lxy lyz lzx : list Line),
   I P L C A o
-> In a A
-> In (lxy ++ lyz ++ lzx) a
-> trail x y pxy lxy
-> trail y z pyz lyz
-> trail z x pzx lzx
-> ~LIn (y, z) L
-> y <> z
-> 2 <= length lyz
-> 2 <= length (lxy ++ lzx)
-> I P
     ((y,z)::L)
     (((z,y)::lyz)::(lxy++(y,z)::lzx)
                  ::(set_remove eq_listline_dec (lxy ++ lyz ++ lzx) C))
     ((((z,y)::lyz)::nil)
      ::((lxy++(y,z)::lzx)::(set_remove eq_listline_dec
                                          (lxy ++ lyz ++ lzx) a))
      ::(set_remove eq_listcircuit_dec a A))
     o

| add_inpath : forall(P : list Point)(L : list Line)
             (C : list Circuit)(A : list Area)(o : Circuit)
             (a : Area)(x y z s e : Point)(ap pxy pyz pzx : list Point)
             (al lxy lyz lzx : list Line),
   I P L C A o
-> In a A
-> In (lxy ++ lyz ++ lzx) a
-> trail x y pxy lxy
-> trail y z pyz lyz
-> trail z x pzx lzx
-> simplepath s e ap al
-> 1 <= length (al ++ lyz)
-> 1 <= length (lxy ++ lzx)
```

```
-> y <> z \/ s <> e
-> (forall(p : Point), In p ap -> ~In p P)
-> I (ap++P)
     ((y,s)::(e,z)::al++L)
     ((((s,y)::lyz)++((z,e)::(reverse_linelist al)))
      ::(lxy++((y,s)::al)++((e,z)::lzx))
      ::(set_remove eq_listline_dec (lxy ++ lyz ++ lzx) C))
     (((((s,y)::lyz)++((z,e)::(reverse_linelist al)))::nil)
      ::((lxy++((y,s)::al)++((e,z)::lzx))
      ::(set_remove eq_listline_dec (lxy ++ lyz ++ lzx) a))
      ::(set_remove eq_listcircuit_dec a A))
     o

| add_loop : forall(P : list Point)(L : list Line)
              (C : list Circuit)(A : list Area)(o : Circuit)
              (a : Area)(x y : Point)(ap : list Point)(al : list Line),
   I P L C A o
-> In a A
-> simplepath x y ap al
-> x <> y
-> 2 <= length al
-> (forall(p : Point), In p ap -> ~In p P)
-> I (ap++P)
     ((y,x)::al++L)
     (((y,x)::al)::(reverse_linelist ((y,x)::al))::C)
     ((((y,x)::al)::nil)
     ::((reverse_linelist ((y,x)::al))::a)
           ::(set_remove eq_listcircuit_dec a A))
     o.
```