Tests and proofs for enumerative combinatorics

Catherine Dubois, Alain Giorgetti, Richard Genestier

Tests and Proofs for Enumerative Combinatorics

Catherine Dubois[1], Alain Giorgetti[2], and Richard Genestier[2]

[1] Samovar (UMR CNRS 5157), ENSIIE, Évry, France
catherine.dubois@ensiie.fr
[2] FEMTO-ST Institute (UMR CNRS 6174 - UBFC/UFC/ENSMM/UTBM), University of Franche-Comté, Besançon, France
{alain.giorgetti,richard.genestier}@femto-st.fr

Abstract. In this paper we show how the research domain of enumerative combinatorics can benefit from testing and formal verification. We formalize in Coq the combinatorial structures of permutations and maps, and a couple of related operations. Before formally proving soundness theorems about these operations, we first validate them, by using logic programming (Prolog) for bounded exhaustive testing and Coq/QuickChick for random testing. It is an experimental study preparing a more ambitious project about formalization of combinatorial results assisted by verification tools.

1 Introduction

Enumerative combinatorics is the branch of mathematics studying discrete structures of finite cardinality when some of their parameters are fixed. One of its objectives is counting, i.e. determining these cardinalities. This research domain also studies non-trivial structural bijections between two families of structures and algorithms for exhaustive generation up to some size. In this paper we show how the research domain of enumerative combinatorics can benefit from testing and formal verification. In enumerative combinatorics we target combinatorial maps, defined as a pair of permutations acting transitively on a set. In software engineering we focus on automated testing and interactive deductive verification with the Coq proof assistant [2].

We formalize in Coq the notions of permutation and combinatorial map, two operations on permutations, and two operations on combinatorial maps. Technically we first define these operations on functions. Then we formally prove that they can be restricted to permutations, and finally to maps for the last two. In other words we prove that they respectively preserve permutations and the map structure.

Unless the proof is trivial, it is common to test lemmas and theorems before proving. Main validation methods are random(ized) testing, bounded exhaustive testing (BET) [5] and finite model finding [3]. In the following we deal with random testing and BET. BET checks a formula for all its possible inputs up to a given small size. It is often sufficient to detect many errors, while providing counterexamples of minimal size. A challenge for BET is to design and implement

efficient algorithms to generate the data. We address it in a lightweight way by exploiting the features of logic programming implemented in a Prolog system. Prolog is well suited for algorithm prototyping due to its closeness to first-order logic specifications. Thanks to backtracking, characteristic predicates written in Prolog can often be used for free as bounded exhaustive generators.

We present a successful application of random and bounded exhaustive testing to debug Coq specifications of combinatorial structures. Our original approach of both case studies (permutations and maps) also is a contribution in formalization of mathematics. In comparison with other approaches [8,10,16], our formalization is very close to the mathematical definition of a map, as a transitive pair of permutations. Our work is freely available at http://members.femto-st.fr/alain-giorgetti/en/coq-unit-testing. It has been developed with Coq 8.4 and SWI-Prolog 5.10.4 [28].

The paper is organized as follows. Section 2 presents the testing methodology on the simple example of permutations. Section 3 introduces the notion of rooted map, its formalization in Coq, correctness theorems, and random and bounded exhaustive testing performed before trying to prove them. Section 4 describes related work and Sect. 5 concludes.

2 Testing Coq Conjectures

This section presents our methodology for testing Coq specifications. Before investing time in proving false lemmas we want to check their validity. Property-based testing (PBT) is popular for functional languages, as exemplified by QuickCheck [7] in Haskell. QuickCheck like approach has also been adopted by proof assistants, e.g. Isabelle [1], Agda [12], PVS [22], FoCaLiZe [6] and more recently Coq [23]. We consider here two kinds of PBT: random testing (in Sect. 2.2) and bounded exhaustive testing (in Sect. 2.3). They are illustrated by the running example of permutations on a finite set presented in Sect. 2.1.

2.1 Permutations in Coq

Permutations on a finite set form an elementary but central combinatorial family. In particular, permutations are the core of the definition of combinatorial maps. It is well known that any injective endofunction on a finite domain is a permutation. However, as far as we know, no popular Coq library defines permutations as injective endofunctions supporting the two operations of insertion and direct sum that we introduce here for their interest in the formal study of rooted maps in Sect. 3. In the following the reader is required to have some basic knowledge about Coq.

Listing 1.1 shows our Coq formalization of permutations. A permutation is defined as an injective function from an interval of natural numbers (whose lower bound is 0) to itself. In Coq the inductive type `nat` of Peano natural numbers is predefined, with the constructors `O` for zero and `S` for the successor function. We manipulate functions defined on `nat` (later called *natural functions*) but we

only impose constraints for the elements in the interval, whatever the definition outside the interval. The predicates is_endo and is_inj respectively define the properties of being an endofunction and injectivity. Then a permutation is a *record* structure composed of a natural function and the proofs that the latter satisfies the two previous properties. For convenience we also consider their conjunction is_permut.

```
Definition is_endo (n : nat) (f : nat → nat) := ∀ x, x < n → f x < n.
Definition is_inj (n : nat) (f : nat → nat) := ∀ x y,
 x < n → y < n → x <> y → f x <> f y.
Record permut (n : nat) : Set := MkPermut {
 fct : nat → nat;
 endo : is_endo n fct;
 inj : is_inj n fct }.
Definition is_permut n f := is_endo n f ∧ is_inj n f.
```

Listing 1.1. Permutations as injective endofunctions in Coq.

We can define a more concrete encoding of permutations: a permutation p on $\{0,\ldots,n-1\}$ may also be represented by the list $[p(0);p(1);\ldots;p(n-1)]$ of its images, called its *one-line notation* in combinatorics. For instance the list $[1;0;3;2;6;4;5]$ represents the permutation $p = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6 \\ 1\,0\,3\,2\,6\,4\,5 \end{pmatrix}$. We'll generate permutations as lists and go from this representation to the functional one with the help of the function list2fun defined by

```
Definition list2fun (l:list nat) : nat → nat := fun (n:nat) ⇒ nth n l n.
```

The function nth in Coq standard library is such that (nth n l d) returns the n-th element of l if it exists, and d otherwise.

Let f be a function defined on $\{0,\ldots,n-1\}$ and i a natural number. The *insertion before i in f* is the function f' defined on $\{0,\ldots,n\}$ as follows: (a) it is f if $i > n$; (b) it is f extended with the fixed-point $f(n) = n$ if $i = n$; (c) if $i < n$ then $f'(n) = i$, $f'(j) = n$ if $f(j) = i$, and $f'(j) = f(j)$ if $0 \leq j \leq n-1$ and $f(j) \neq i$. The operation of insertion in a natural function is defined in Coq by

```
Definition insert_fun n (f : nat → nat) (i : nat) : nat → nat :=
 fun x ⇒ if le_lt_dec i n then
           match nat_compare x n with
             Eq ⇒ i
           | Lt ⇒ if eq_nat_dec (f x) i then n else f x
           | Gt ⇒ x
           end
         else x.
```

The *direct sum* of a function f_1 defined on $\{0,\ldots,n_1-1\}$ and a function f_2 defined on $\{0,\ldots,n_2-1\}$ is the function f on $\{0,\ldots,n_1+n_2-1\}$ such that $f(x) = f_1(x)$ if $0 \leq x < n_1$ and $f(x) = f_2(x-n_1)+n_1$ if $n_1 \leq x < n_1+n_2$. It is an extension of the well-known *direct sum* on permutations [18, p. 57]. The direct sum is defined in Coq on natural functions by

```
Definition sum_fun n1 f1 n2 f2 : nat → nat := fun x ⇒
 if lt_ge_dec x n1 then f1 x else
 if lt_ge_dec x (n1+n2) then (f2 (x-n1)) + n1 else x.
```

Listing 1.2 states that both operations preserve permutations. To validate these lemmas, we define Boolean versions is_endob, is_injb and is_permutb of the

logical properties is_endo, is_inj and is_permut. Listing 1.3 shows the functions is_endob and is_permutb. An evaluation of (is_endob n f) returns true iff the function f is an endofunction on $\{0, \ldots, n-1\}$. The lemma is_endo_dec states that the Boolean function is_endob is a correct implementation of the predicate is_endo. Similar lemmas are proved for the other two Boolean functions. If the correlation between is_endo and is_endob is quite immediate, it is not the case for is_inj and is_injb. To define is_injb, we rely on another lemma we have proved: a function f is injective on $\{0, 1, \ldots, n\}$ iff the list $[f(0); f(1); \ldots; f(n)]$ of its images has no duplicate.

```
Lemma insert_permut: ∀ (n : nat) (p : permut n) (i : nat),
 is_permut (S n) (insert_fun n (fct p) i).
Lemma sum_permut: ∀ n1 (p1 : permut n1) n2 (p2 : permut n2),
 is_permut (n1 + n2) (sum_fun n1 (fct p1) n2 (fct p2)).
```

Listing 1.2. Preservation properties of the insertion and sum operations.

```
Fixpoint is_endob_aux n f m := match m with
  0 ⇒ if (lt_dec (f 0) n) then true else false
| S p ⇒ if (lt_dec (f m) n) then is_endob_aux n f p else false
end.
Definition is_endob n f := match n with
  0 ⇒ true
| S p ⇒ is_endob_aux n f p
end.
Lemma is_endo_dec : ∀ n f, (is_endob n f  = true ↔ is_endo n f).
Definition is_permutb n f := (is_endob n f) && (is_injb n f).
```

Listing 1.3. Boolean functions for permutations.

2.2 Random Testing

QuickChick [17] is a random testing plugin for Coq. It allows us to check the validity of executable conjectures with random inputs. QuickChick is mainly a generic framework providing combinators to write testing code, in particular random generators. The general workflow that we follow to validate by testing a conjecture like ∀ x: T, precondition x → conclusion (f x), where precondition and conclusion are logical predicates, starts with the definition of a random generator gen_T of values of type T that satisfy the property precondition. Then we have to turn conclusion into a Boolean function conclusionb – if it is possible, otherwise QuickChick does not apply – that we prove semantically equivalent to the logical predicate. The test is run by using the following command which generates a fixed number of inputs using the generator gen_T and for each one applies the function f and verifies the property under test (conclusion):

```
QuickCheck (forAll gen_T (fun x ⇒ conclusionb (f x))).
```

In this approach we rely on the generator which is here part of the trusted code. QuickChick proposes some theorems (or axioms) about its different combinators which could be used to prove that the generator is correct, but it may be hard work. In the following we propose to test that the generator produces correct

outputs. For that purpose, we implement the same approach: turning the logical property precondition into an executable one preconditionb.

We now illustrate QuickChick features on permutations encoded as natural functions. However, QuickChick cannot deal with functions so we generate permutations as lists and then transform them into functions, as detailed in Sect. 2.1. Let us notice that QuickChick heavily uses monads. However, in the following we explain very informally some piece of code.

We first define a generator for permutations on $\{0, \ldots, n-1\}$, as lists without any duplicate containing 0, 1, $\ldots n-1$ in any order:

```
Fixpoint gen_permutl (n : nat) : G (list nat) := match n with
  0   ⇒ returnGen nil
| S p ⇒ do! m ← choose (0, n); liftGen (insert_pos p m) (gen_permutl p)
end.
```

If n is 0, the output is the empty list. Otherwise (n is the successor of p) the recursive call (gen_permutl p) generates a list encoding a permutation on $\{0, \ldots, p-1\}$ and the function inserts p in the latter at a position m which is randomly chosen (using the combinator choose). The combinator liftGen applies a function, here insert_pos p m, to the result of a generator. To have confidence in this generator, we test that the outputs do not contain any duplicate, that their length is n and that their elements are natural numbers less than n. These three conditions are implemented by the Boolean predicate list_permutb.

```
QuickCheck (sized (fun n ⇒ forAll (gen_permutl n) (list_permutb n))).
+++ OK, passed 10000 tests
```

The maximal number of tests (10000 here) can be adjusted by the user. We iterate over different values for n thanks to the use of the combinator sized.

We can follow the same process to validate that permutations as natural functions are obtained by applying the translation function list2fun on lists generated by the previous generator gen_permutl:

```
Definition fun_permutb n l := is_permutb n (list2fun l).
QuickCheck (sized (fun n ⇒ forAll (gen_permutl n) (fun_permutb n))).
+++ OK, passed 10000 tests
```

We are now ready to test the conjectures formulated in Listing 1.2, following the same methodology: (i) when a natural function representing a permutation is to be generated, we use the list generator gen_permutl; (ii) the logical property under test is turned into its Boolean version composed with the translation function list2fun. For example testing Lemma insert_permut is obtained by

```
QuickCheck (sized (fun n ⇒ forAll (gen_permutl n)
             (fun l ⇒ (forAll arbitraryNat
               (fun i ⇒ let f := list2fun l in
                 is_permutb (S n) (insert_fun n f i)))))).
```

This QuickCheck command has the same structure as the previous ones except that we use two generators, one for permutations and another one for arbitrary natural numbers, named arbitraryNat. This command passed 10000 tests. If we inject a fault in the definition of insert_fun, e.g. replacing the result n by S n in the Lt case, we get a counterexample, e.g. $l = [0; 1]$ and $i = 0$ for $n = 2$.

2.3 Bounded Exhaustive Testing

For testing Coq specifications we also advocate in favor of bounded exhaustive testing (BET) and its lightweight support with logic programs, for many reasons. Firstly BET is especially well adapted to enumerative combinatorics, because it corresponds to the familiar research activity of generation of combinatorial objects in this domain. Secondly BET provides the author of a wrong lemma with the smallest combinatorial structure revealing her error. Thirdly the combinatorial structures formalized in Coq as inductive structures with properties are often easy to formalize in first-order logic with Prolog predicates. Fourthly the Prolog backtracking mechanism often provides bounded exhaustive generators for free. All these advantages are illustrated in this paper.

In order to make the validation tasks easier, we extend a Prolog validation library created by Valerio Senni [27] and previously applied to the validation of algorithms on words encoding rooted planar maps [14]. The library provides full automation for symmetric bounded exhaustive comparison for increasing bound values. It returns counterexamples whenever validation fails (so the debugging process is guided by those counterexamples), and it collects statistics such as generation time and memory consumption. We illustrate some of the validation library features on the example of permutations. The reader is assumed to be familiar with logic programming, or can otherwise read a short summary in [14].

We encode a function f on $\{0,\ldots,n-1\}$ by the Prolog list of its images $[f(0),\ldots,f(n-1)]$, its one-line notation. A list is *linear* if it has no duplicates. Listing 1.4 shows a Prolog predicate `line` such that the formula `line(L,N)` (resp. `line(L,K,N)`) holds iff `L` is a linear list of length `N` (resp. `K`) with elements in {`0`,...,`N-1`}. We then say for short that `L` is a *permutation list*. In other words, this characteristic predicate of permutations corresponds to (is_permut n) in Coq. The predicate is parameterized by the list length. This is not strictly required for formal specification but useful for generation purposes. The formula `in(K,I,J)` holds iff the integer `K` is in the interval [`I`..`J`].

```
line([],0,_).
line([Y|P],K,N) :- K > 0, Km1 is K-1, Nm1 is N-1, in(Y,0,Nm1),
 line(P,Km1,N), \+ member(Y,P).
line(P,N) :- line(P,N,N).
```

Listing 1.4. Permutations in Prolog.

A clear advantage of logic programming is that the predicate `line` works in two ways: as an *acceptor* of permutation lists, and as a *generator* enumerating permutation lists of a given length. For a characteristic predicate p and a given size n the query scheme

Q: `p(L,`n`), write_coq(L), fail.`

indeed allows the enumeration of all the accepted data of size n. The query forces the construction of a first datum `L` of size n accepted by p, its output on a stream, and the failure of the proof mechanism by using the built-in fail. Since the proof fails, the backtracking mechanism recovers the last choice-point (necessarily in p) and triggers the generation of a new datum, until there are no more choice-points. Here the predicate `write_coq` is defined by the user to

output (as side-effect) a test case in Coq syntax. For instance, it can easily be defined so that the query

```
line(L,3), write_coq(3), fail.
```

writes one Coq line such as

```
Eval compute in (is_permutb 3 ( list2fun  [2;0;1])) .
```

for each permutation list of length 3. These lines constitute a test suite for the Coq function is_permutb, under the assumptions that the Coq function list2fun and the Prolog program in Listing 1.4 are correct. The latter can be checked in two ways: by visual inspection of the lists it generates, or by counting. For counting, the library provides the predicate `iterate` so that the query

```
:- iterate(0,6,line).
```

outputs the numbers 1, 1, 2, 6, 24, 120 and 720 of distinct lists of length n from 0 to 6 accepted by the predicate `line`. We then easily recognize the first numbers $n!$ of permutations of length n.

We can now adapt the predicate `write_coq` of the query Q to the BET of the lemmas in Listing 1.2. For Lemma insert_permut the query evaluation can generate in a Coq file all the Coq lines of the form

```
Eval compute in (is_permutb (n + 1) ( insert_fun  n (list2fun l) i)).
```

for n up to some bound, for $0 \leq i \leq n$ and for any list l (of length n) satisfying `line(`l`,`n`)`. Then we check that the compilation of the generated Coq file always produces true. We proceed similarly with the lemma sum_permut.

As mentioned before about the property is_inj, it may be hard to write a Boolean version of a property and to prove its correctness. In that case BET sometimes remains possible, as illustrated by the following example. Suppose that we find no implementation of the property is_permut. Then we generate a non-computational proof generalizing the following example

```
Goal is_permut 3 (list2fun (2::0::1::nil)). unfold is_permut.
unfold list2fun. unfold list2funX. split.
- unfold is_endo. intros x Hx. assert (x = 0 ∨ x = 1 ∨ x = 2). omega.
  firstorder; subst; simpl; omega.
- unfold is_inj. intros x y Hx Hy Hxy.
  assert (x = 0 ∨ x = 1 ∨ x = 2). omega.
  assert (y = 0 ∨ y = 1 ∨ y = 2). omega.
  firstorder; subst; simpl; omega. Qed.
```

The proof first splits into one subproof for the property is_endo and one for is_inj. Each subproof works by enumeration of the possible values of x (and y for injectivity). This approach holds whenever the property is universally quantified with variables i of type nat upper bounded by some number b. Then the tactic enumerates all the possible values of i. The assertion is proved by the tactic omega which implements a decision procedure for linear arithmetics (the Omega test [24]). The proof is then decomposed into cases by the firstorder tactic. In the subproof for injectivity each case contains hypotheses x =... and y =... assigning values to both variables. After replacement of x and y with their values the Omega test ends the proof.

3 Case Study of Rooted Maps

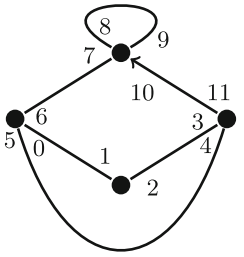
It is now time to apply our test methodology to more challenging Coq theorems. As case study we consider the combinatorial family of rooted maps, formalized in Coq as transitive permutations (Sect. 3.1). Then we introduce two operations that should construct a map from one or two smaller ones by edge addition (Sect. 3.2). Both operations are defined as combinations of the two operations of insertion and direct sum defined in Sect. 2. Finally we check by testing and then prove formally that both operations preserve permutations and transitivity (Sect. 3.3). Section 3.4 reports some testing and proving statistics.

3.1 Definitions and Formalization

A *topological map* is a cellular embedding of a connected graph (possibly with loops and multiple edges) into a compact, oriented surface without boundary [19]. A *face* of a topological map is a connected component of the complement of the graph in the surface. By definition each face is homeomorphic to an open disc. Figure 1(a) shows a topological map. It is drawn on the plane for convenience, but should be considered as drawn on the sphere, so that the *outer face* (the infinite white piece of the plane) becomes homeomorphic to an open disk. We admit the existence of a map containing a single vertex, no edges, and a single face, called the *vertex map*. Any other map contains at least one edge.

A half-edge, i.e. an edge equipped with one of its two possible orientations, is usually called a *dart*. The other dart on the same edge is called the *opposite dart*. The vertex at the source of a dart and the face to the right of a dart are said to be *incident* to that dart. A *loop* is an edge whose two associated darts are incident to the same vertex. We only consider *labeled* topological maps, whose darts are identified by a unique label. In the drawings the label of a dart is always written in its incident face and near its incident vertex. For instance, the dart 11 in Fig. 1(a) is incident to the outer face, it is incident to the same vertex as the darts 3 and 4, and its opposite dart is labeled by 10.

Edmonds [13] reduced topological maps to their combinatorial structure, defined as follows. A *(combinatorial) labeled map* with n edges is a triple (D, R, L)



(a) Topological map

$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$
$R = (5\ 0\ 6)\ (1\ 2)\ (11\ 3\ 4)\ (8\ 7\ 10\ 9)$
$L = (0\ 1)\ (2\ 3)\ (4\ 5)\ (6\ 7)\ (8\ 9)\ (10\ 11)$

(b) Combinatorial map (D, R, L)

Fig. 1. Two representations of a rooted map.

where D is a finite set of even cardinality $2n$, R is a permutation of D and L is a fixed-point free involution of D such that the group $\langle R, L\rangle$ generated by R and L acts transitively on D. This transitivity means that any element of D can be obtained from any other element of D by finitely many applications of the permutations R, L and their inverse. Figure 1(b) shows the combinatorial map (D, R, L) corresponding to the topological map in Fig. 1(a). More generally the one-to-one correspondence between topological and combinatorial labeled maps works as follows. An element of the set D is a dart of the topological map. The permutations R and L respectively encode its vertices and edges. An orbit of the permutation R lists the darts encountered when turning counterclockwise around a vertex. The involution L exchanges each dart with its opposite on the same edge. For instance, the orbit (5 0 6) of R encodes the leftmost vertex in Fig. 1(a) and the orbit (8 9) of L encodes the loop in Fig. 1(a). The transitive action of the permutations R and L corresponds to the connexity of the embedded graph.

A *rooting* of a map is essentially the choice of one of its darts, called its *root*. The edge which includes the root dart is called the *root edge*. By convention, the vertex map is also considered to be rooted. In the drawings the root dart is indicated by an arrow, as the dart 11 in Fig. 1(a). Two labeled maps (D, R, L) and (D', R', L') are *isomorphic* if there is a bijection θ from D to D' such that $R' = \theta^{-1} R\ \theta$ and $L' = \theta^{-1} L\ \theta$. This bijection is called a *labeled map isomorphism*. For a predefined root $d \in D$, two labeled maps (D, R, L) and (D, R', L') with the same set of darts D are *root-preserving isomorphic* if they are isomorphic and their isomorphism preserves the root d. A *rooted combinatorial map* (or *map* for short) is an equivalence class for the relation of root-preserving isomorphism between labeled maps. For the purpose of enumeration, the special virtue of rooted maps is that they have no symmetries, in the sense that the automorphism group of any rooted map is trivial.

In order to simplify the formalization and formal reasoning we refine the usual definition of a combinatorial labeled map. In the usual definition of a labeled map $M = (D, R, L)$ the set of darts D is any finite set and the permutation L is any fixed-point free involution on D. Here we fix D to $\{0, \ldots, 2e-1\}$ for any map with e edges. Since rooted maps are defined modulo conjugation, we also fix L to the fixed-point free involution that swaps $2i$ and $2i+1$ for all $0 \leq i < e$. This involution is formalized in Coq by

```
Definition opp (n:nat) : nat := if (even_odd_dec n) then (n+1) else (n-1).
```

Such a map is said to be *local*. For instance, the combinatorial map in Fig. 1(b) is local. A local map can be represented using only its vertex permutation R, called its *rotation*. To root a local map, we always choose the largest element $(2e-1)$ of D.

We now define the transitivity of any function f on some set D so that a triple (D, R, opp) is a local map iff its rotation R is transitive. We say that there is a *step* between two elements x and y of D by a function f iff $f(x) = y$, $f(y) = x$ or $\mathsf{opp}(x) = y$. Two numbers x and y are *connected* by f iff there is a path (i.e. a sequence of steps) from x to y. Finally, a function f is *transitive* on D if any two elements of D are connected by f.

```
Inductive connected n (f : nat → nat) : nat → nat → nat → Prop :=
 | c0    : ∀ x y, x < n → y < n → x = y → connected n f 0 x y
 | cfirst : ∀ l x y z, x < n → y < n → z < n →
    f x = y ∨ f y = x ∨ opp x = y →
    connected n f l y z → connected n f (S l) x z.
Definition is_transitive_fun (n: nat) (f : nat → nat) : Prop :=
 ∀ x, x < n → ∀ y, y < n → ∃ m, connected n f m x y.
Definition transitive_fun (n: nat) (f : nat → nat) : Prop :=
 ∀ y, y < n → ∀ x, x < y → ∃ m, connected n f m x y.
Definition is_transitive n (p : permut n) := transitive_fun n (fct p).
```

Listing 1.5. Definition of transitivity.

Listing 1.5 shows a Coq formalization of transitivity on $\{0, \ldots, n-1\}$ of a natural function. The connectivity property is specified by the inductive predicate connected so that (connected n f l x y) holds iff the natural numbers x and y are related by exactly l steps of f. The constructor cfirst states that a path between x and y can be decomposed into its first step and its end, while the constructor c0 expresses the trivial case where $x = y$. This definition is completed by three lemmas (not shown here): one lemma decomposing a path into its beginning and its last step and two lemmas respectively proving the symmetry and the transitivity of the binary relation (connected n f l). The definitions is_transitive_fun and transitive_fun implement two versions of the property of transitivity of a function. Decompositions into cases in several proofs are dramatically shortened by using the second definition transitive_fun of transitivity considering only numbers x strictly smaller than y. Using symmetry of the predicate connected we prove that both definitions of transitivity are equivalent. The predicate is_transitive defines the transitivity of a permutation as the transitivity of its associated function.

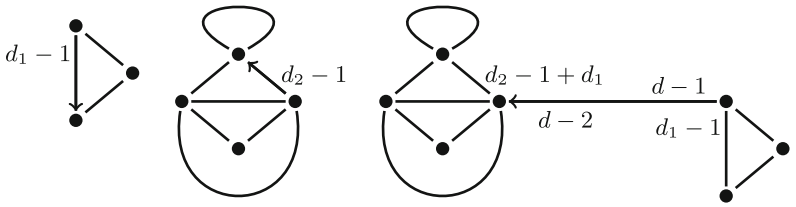
All the maps considered hereafter are local and are encoded by their transitive rotation. A local map (D, R, opp) with e edges is formalized in Coq by a record composed of its vertex permutation of length $2e$ (its rotation) and the property that this permutation is transitive, as follows:

```
Record map (e : nat) : Set := {
 rotation : permut (2*e);
 transitive : is_transitive rotation }.
```

3.2 Map Construction Operations

An edge is an *isthmus* if both of its associated darts are incident to the same face. A map is *isthmic* (resp. *non-isthmic*) if it is not the vertex map and its root edge is (resp. not) an isthmus. We define here an operation c_I constructing an isthmic map from two maps and a family of operations c_N^k (indexed by a number k) constructing a non-isthmic map from one map. Both operations proceed by addition of one edge.

Isthmic Operation. The operation c_I is illustrated by an example in Fig. 2. It adds an isthmic edge between a local map M_1 with e_1 edges and a local map M_2 with e_2 edges. The result is a map $M = c_I(M_1, M_2)$ with $e_1 + e_2 + 1$ edges.



(a) Map M_1. (b) Map M_2. (c) Map $M = c_I(M_1, M_2)$.

Fig. 2. Example of construction of an isthmic map.

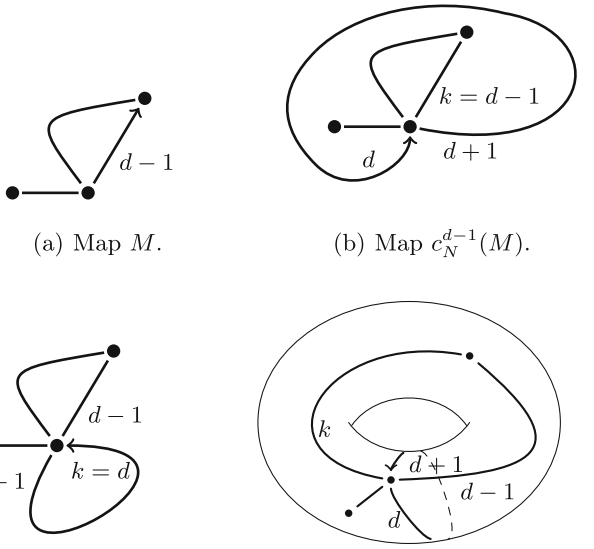
Let $d_1 = 2e_1$, $d_2 = 2e_2$ and $d = 2e_1 + 2e_2 + 2$ be the numbers of darts of M_1, M_2 and M. The additional edge is composed of the two darts $d-1$ $(= d_1 + d_2 + 1)$ and $d-2$ $(= d_1 + d_2)$. The root of M is the dart $d-1$, while its opposite dart $\mathsf{opp}(d-1)$ is $d-2$.

```
Definition isthmic_fun d1 (r1 : nat → nat) d2 (r2 : nat → nat) : nat → nat
:= match d1 with
| 0 ⇒ match d2 with
       | 0 ⇒ insert_fun 1 (insert_fun 0 r2 0) 1
       | S d2m1 ⇒ insert_fun (d2+1) (insert_fun d2 r2 d2m1) (d2+1)
       end
| S d1m1 ⇒ match d2 with
            | 0 ⇒ insert_fun (d1+1) (insert_fun d1 r1 d1) d1m1
            | S d2m1 ⇒
              insert_fun (d1+d2+1)
               (insert_fun (d1+d2) (sum_fun d1 r1 d2 r2) (d1m1+d2))
              d1m1
            end end.
```

Listing 1.6. Isthmic operation in Coq.

Listing 1.6 presents a Coq function isthmic_fun implementing this operation on two natural functions r1 and r2 representing the rotations R_1 and R_2 of M_1 and M_2. If R_1 and R_2 represent the vertex map ($d_1 = d_2 = 0$) then the resulting map M is reduced to one non-loop edge and the resulting rotation R is the permutation (0) (1). If M_1 is the vertex map ($d_1 = 0$) and M_2 is not empty ($d_2 \neq 0$) then the dart $d-2$ $(= d_2)$ is added just before the dart $d_2 - 1$ in its orbit in R_2 and then the dart $d-1$ $(= d_2 + 1)$ is added as a fixed-point of R. If M_1 is not empty ($d_1 \neq 0$) and M_2 is the vertex map ($d_2 = 0$) then the dart $d-2$ $(= d_1)$ is added as a fixed-point, and then the dart $d-1$ $(= d_1 + 1)$ is added just before the dart $d_1 - 1$ in its orbit. Otherwise, when M_1 and M_2 are not the vertex map, the direct sum $R' = sum(R_1, R_2)$ is computed thanks to a call to the function sum_fun. Then the dart $d_1 + d_2$ is inserted just before the dart $d_2 - 1 + d_1$ in its orbit in R', and finally the dart $d_1 + d_2 + 1$ is inserted just before the dart $d_1 - 1$ in its orbit in the resulting permutation.

Non-isthmic Operation. For $0 \leq k \leq 2e$ the operation c_N^k adds a non-isthmic edge in a local map M with e edges (represented by its rotation R of length $d = 2e$) to obtain a local map M' with $e+1$ edges represented by its rotation R' of length $d' = 2e + 2$. The resulting permutation R' is obtained by insertion



(a) Map M. (b) Map $c_N^{d-1}(M)$.

(c) Map $c_N^d(M)$. (d) Map $c_N^k(M)$, for $0 \leq k < d-1$.

Fig. 3. Examples of construction of non-isthmic maps.

of the new root $d+1$ and its opposite d in R. If M is the vertex map, the new edge is added – as a loop – in a unique way to obtain M'. Otherwise, there are $d+1$ ways to add the new edge, distinguished by the value of k between 0 and d. Figure 3 shows a map M with three edges (in Fig. 3(a)) and three maps obtained by application of the operation c_N^k on M, for different values of k. Two of them are planar maps whereas the last one in Fig. 3(d) is a toroidal map (a map on a torus). When $k = d-1$ and $k = d$ the added edge is a loop. These cases are respectively illustrated in Fig. 3(b) and (c). Note that the order of insertion of darts is important: in Fig. 3(b), the dart d is inserted just before the dart $d-1$ but then the dart $d+1$ is inserted just before the dart $k = d-1$, so that the dart d finally is just before $d+1$ in its orbit in the rotation R' of M'. In all the other cases, the dart d is just before the dart $d-1$ in R'. Figure 3(d) shows a case $0 \leq k < d-1$ where the dart k is not incident to the same face as $d-1$. In this case, the new edge can only be added through a hole perfored in the surface.

Listing 1.7 presents a Coq function non_isthmic_fun implementing this operation on a natural function r representing the vertex permutation R of a local map with d darts, when k is d or less. When $d = 0$ the rotation R represents the vertex map, the new edge is a loop and the resulting function is the permutation (0 1). Otherwise, the dart d is inserted just before the root $d-1$ of M in its orbit in R. Let us denote here by Q the resulting permutation. Then the dart $d+1$ is inserted just before the dart k in its orbit of Q.

```
Definition non_isthmic_fun (d:nat) (r:nat → nat) (k:nat)
 (k_le_d:k ≤ d) : nat → nat :=
match d with
| 0     ⇒ insert_fun 1 (insert_fun 0 r 0) 0
| S dm1 ⇒ insert_fun (S d) (insert_fun d r dm1) k
end.
```

Listing 1.7. Non-isthmic operation in Coq.

3.3 Validation and Proof

We have separately formalized combinatorial maps and two map constructions as operations on natural functions. It remains to prove that each operation preserves transitive permutations. We proceed in two steps. The first step consists in checking and proving that both operations preserve permutations. The second step concerns transitivity.

Preservation of Permutations. The proof that the construction operations preserve permutations is decomposed into intermediate lemmas. For example, one of them

```
Lemma isthmic_endo : ∀ d1 (r1:permut d1) d2 (r2:permut d2),
 is_endo (S (S (d1 + d2))) (isthmic_fun d1 (fct r1) d2 (fct r2)).
```

states that the isthmic operation preserves endofunctions, and

```
Lemma non_isthmic_inj: ∀ d (r:permut d) k (k_le_d:k ≤ d),
 is_inj (S (S d)) (non_isthmic_fun d (fct r) k k_le_d).
```

states that the non-isthmic operation preserves injectivity.

As in Sect. 2.3, we validate by BET that the isthmic and non-isthmic operations preserve permutations. In the non-isthmic case, we meet a specificity of testing with dependent types. The non-isthmic operation is indeed parameterized by a proof. So the BET has to generate such a proof for each test input. It can be automated only if all these proofs share a common pattern (notion of *uniform* proof). In the present case we need a uniform proof of $x \leq y$ for any pair of natural numbers x and y. Fortunately the Coq predicate $\leq$ is reflected by the Boolean function leb:nat→nat→bool through the lemma

```
Lemma leb_complete : ∀ m n, leb m n = true → m ≤ n.
```

so that the term (leb_complete x y eq_refl) is a uniform Coq proof of $x \leq y$. For the preservation of permutations by the non-isthmic operation the BET generates test cases such as

```
Eval compute in (
 let proof := leb_complete 2 3 eq_refl
 in is_permutb 5 (non_isthmic_fun 3 (list2fun[0;0;0]) 2 proof)).
```

for $x = 2$ and $y = 3$. After this validation we have proved all the permutation preservation lemmas.

Random testing also allows us to validate the preservation of permutations by the isthmic operation. The following QuickCheck command randomly generates a first natural number d1, a permutation list l1 of length d1 and then a second

number d2 together with a permutation list of length d2 and checks if the isthmic operation builts a permutation from the corresponding natural functions.

```
QuickCheck (forAll arbitraryNat (fun d1 ⇒
  forAll (gen_permutl d1) (fun l1 ⇒ let f1 := list2fun l1 in
    forAll arbitraryNat (fun d2 ⇒
      forAll (gen_permutl d2) (fun l2 ⇒ let f2 := list2fun l2 in
        is_permutb (S (S (d1 + d2))) (isthmic_fun d1 f1 d2 f2)))))).
```

Unfortunately for non_isthmic_fun, this process is not applicable. We could follow the same process: generate a natural number d, a permutation list of length d and a number k less than d and so on. Thanks to the choose combinator provided by QuickChick, it is easy to provide such a k. However, we are not able to produce a proof term for $k \leq d$ which is an argument required by non_isthmic_fun because QuickChick does not provide a correctness proof for choose. A solution could be to rewrite non_isthmic_fun without this proof argument.

Preservation of Transitivity. We first validate and then demonstrate that the isthmic and non-isthmic operations preserve transitive permutations and therefore can be considered as operations on (local) maps. These properties are formalized by the two theorems presented in Listing 1.8. Theorem isthmic_trans (resp. non_isthmic_trans) states that the isthmic (resp. non-isthmic) operation preserves the transitivity when acting on two permutations (resp. one permutation) of even length.

```
Theorem isthmic_trans: ∀ d1 (r1: permut d1) d2 (r2: permut d2),
  even d1 → even d2 → is_transitive r1 → is_transitive r2 →
    is_transitive (isthmic_permut r1 r2).
Theorem non_isthmic_trans: ∀ d (r: permut d) k (k_le_d: k ≤ d),
  even d → is_transitive r → is_transitive (non_isthmic_permut r k_le_d).
```

Listing 1.8. Preservation of transitivity by the isthmic and non-isthmic operations.

```
Fixpoint nlist n (f:nat → nat) : nat → list nat := fun x ⇒ match n with
  0   ⇒ (opp x)::nil
| S m ⇒ elimDup ((nlist m f x) ++
   (if eq_nat_dec (f m) x then m::nil else nil) ++
   (if eq_nat_dec x m then (f m)::nil else nil))
end.
Fixpoint dfs (g:nat → list nat) (n:nat) (v:list nat) (x:nat) :=
if (in_dec eq_nat_dec x v) then v else match n with
  0 ⇒ v
| S n' ⇒ fold_left (dfs g n') (g x) (x::v)
end.
Definition is_transitive_funb n f := if
 eq_nat_dec n (length (dfs (nlist n f) n nil 0)) then true else false.
```

Listing 1.9. Boolean function for transitivity.

For testing we propose in Listing 1.9 an implementation of the transitivity predicate defined in Listing 1.5. It is based on a depth-first search in the graph where a directed edge goes from x to y if $f(x) = y$, $f(y) = x$ or $\mathsf{opp}(x) = y$, for any two vertices x and y in $\{0, \ldots, n-1\}$. The call ($\mathsf{nlist}\, n\ f\ x$) returns the list of neighbors of x in this graph. The auxiliary function elimDup eliminates duplicates in a list. The depth-first search is implemented by the function dfs

inspired by the function with the same name in [21]. The function fold_left is such that (fold_left f [x1;..;xk] a) computes f (.. (f (f a x1) x2) ..) xk.

Proving the soundness of this implementation wrt. its specification in Listing 1.5 is not an easy task and is therefore left as a future work. The soundness of is_transitive_funb can however be checked, for instance by counting the first numbers of transitive permutations. The number $t(e)$ of transitive permutations of length $2e$ is indeed equal to the number of rooted maps, multiplied by the number $2^{e-1}(e-1)!$ of isomorphic local labeled maps in a rooted map if $e > 0$ (Remember that a rooted map is an equivalence class of isomorphic labeled maps, for root-preserving isomorphisms). The first numbers of rooted maps and many references about them can be found in [29].

Let d $= 2e$ be an even natural Coq number and let l be a Coq list of all the permutation lists of length d. The Coq code

```
Definition is_transitive_listb d l := is_transitive_funb d (list2fun l).
Eval compute in (length (filter (is_transitive_listb d) l)).
```

computes the length of the list obtained by filtering the transitive permutation lists. Thus it should compute $t(e)$. The list l is generated by the Prolog-based BET presented in Sect. 2. This validation is feasible only for $e = 0, 1, 2, 3$. It correctly counts $t(e) = 1, 2, 20, 592$ after examining $(2e)!$ permutations. For $e = 4$ the Coq compilation runs out of memory. After this validation by counting, we use the Boolean function is_transitive_funb to test the theorems in Listing 1.8.

The isthmic operation combines insertion and direct sum. One could think that it preserves transitivity because these two operations also do. In fact, it is not so simple. In particular, the direct sum operation does not preserve transitivity. It can be understood by coming back to its definition. But it can also be quickly invalidated by BET on an executable version of the wrong property:

```
Theorem sum_transitive: ∀ d1 r1 d2 r2, even d1 → even d2 →
 is_permut d1 r1 → is_transitive_fun d1 r1 →
 is_permut d2 r2 → is_transitive_fun d2 r2 →
 is_transitive_fun (d1 + d2) (sum_fun d1 r1 d2 r2).
```

The BET provides us with the smallest counterexample where r1 and r2 are the function (list2fun [1; 0]) encoding the transposition exchanging 0 and 1.

Random testing also allows us to invalidate this conjecture and obtain some counterexamples. For some executions, we retrieve exactly the previous smallest one. As a process for generating transitive permutations is lacking, we generate permutations as functions (as previously) and filter those which are transitive using the Boolean predicate is_transitive_funb. The following QuickCheck command does the job. If f1 (or f2) is not transitive, the test case is discarded (it is done by the combinator written as $\Longrightarrow$).

```
QuickCheck (forAll gen_even (fun d1 ⇒
    forAll (gen_permutl d1)  (fun l1 ⇒ let f1 := list2fun l1 in
     is_transitive_funb d1 f1⟹
       forAll gen_even (fun d2 ⇒
         forAll (gen_permutl d2) (fun l2 ⇒ let f2 := list2fun l2 in
           is_transitive_funb d2 f2⟹
             is_transitive_funb (d1 + d2) (sum_fun d1 f1 d2 f2))))))).
2 [0;1] 4 [2;0;1;3] *** Failed! After 3 tests and 0 shrinks
```

Each transitivity preservation proof reduces to the preservation of connectivity between any two numbers (darts) x and y with $x < y$. For instance the most complex case in the proof of Theorem isthmic_trans is $0 \leq x < d_1$ ($x \in R_1$) and $d_1 \leq y < d_1 + d_2$ ($y \in R_2$). Its proof constructs a path between x and y by concatenation of a path from the dart x to the root $d_1 - 1$ of R_1, a step between $d_1 - 1$ and the root $d - 1 = d_1 + d_2 + 1$, a step between the root $d - 1$ and its opposite $d - 2 = d_1 + d_2$ through the fixed-point free involution opp, a step from $d - 2$ to the root $d_2 - 1$ of R_2, relabelled $d_1 + d_2 - 1$, and finally a path from that dart $d_1 + d_2 - 1$ to y in the relabelling of R_2.

3.4 Some Metrics

The case study is composed of 80 definitions, 185 lemmas and 2 theorems, for a total of 5580 lines of Coq code. Among them around 280 lines are dedicated to validation. These lines contain 23 definitions and 4 lemmas. They include Boolean versions of some logical definitions used by both random testing and BET, e.g. the Boolean function is_permutb, their corresponding correctness proofs, and the generators required by QuickChick. The Prolog code for BET is composed of 44 lines added to the validation library and 860 lines whose execution generates test suites for the case study.

All the validations by counting and BET presented in the paper are executed with lists up to length 4, in less that 21 s on a PC Intel Core i5-2400 3.10 GHz $\times$ 4 under Linux Ubuntu 14.04 (the time for test generation is neglictible). The QuickChick random tests (10000 test cases for each validation step except for the wrong conjecture) are generated and executed in less than 54 s. These are reasonable times for thousands of automatically generated tests. For a comparison the Coq compilation time is around 20 s.

4 Related Work

Several techniques and tools help strengthening the trust in programs manipulating structured data. Randomized property-based testing (RPBT) consists in random generation of test data to validate given assertions about programs. RPBT has gained much popularity since the appearance of QuickCheck for Haskell [7], followed by e.g. Quickcheck for Isabelle [5]. In RPBT a random data generator can be defined by filtering the output of another one, in a similar way as an exhaustive generator can be defined by filtering another exhaustive generator in BET. A more generic approach is type-targeted testing [26], wherein types are converted into queries to SMT solvers whose answers provide counterexamples. SmallCheck and Lazy SmallCheck [25] are two Haskell libraries for property-based testing, allowing an automatic exhaustive testing for small values. In Coq, as far as we know, there is no equivalent to the Haskell library SmallCheck.

The theory of combinatorial maps was developed from the early 1970's. Tutte [30,31] proposed the most advanced work in this direction, developing an axiomatic theory of combinatorial maps without referencing topology.

More recently Lazarus [20] conducted a computational approach on graphs and surfaces based on combinatorial maps. He notably proposed a formal definition of the basic operation of edge deletion on combinatorial labeled maps. An advanced formalization related to maps is that of combinatorial hypermaps to state and prove the Four Colour Theorem in the Coq system [15,16]. Note that combinatorial hypermaps generalize combinatorial maps by allowing an arbitrary permutation L (i.e., not necessarily a fixed-point free involution). This formalization does not explicitly state that L and R are bijective, but adopt the alternative definition of a hypermap as a *triple* of endofunctions that compose to the identity [15, p. 19]. It would be interesting to investigate this idea with local maps rather than hypermaps, and to determine to what extent it could simplify our formalization. Some formal proofs about combinatorial maps or variants have already been carried out in the domain of computational geometry. Dufourd et al. have developed a large Coq library specifying hypermaps used to prove some classical results such as Euler formula for polyhedra [10], Jordan curve theorem [11], and also some algorithms such as convex hull [4] and image segmentation [9]. In these papers, a combinatorial map or hypermap is represented by an inductive type with some constraints. Its constructors are related to the insertion of a dart or the links of two darts. This representation differs from ours that relies on permutations. In [8], Dubois and Mota proposed a formalization of generalized maps using the B formalism, very close to the mathematical presentation with permutations and involutions. Here we simplify the structure by fixing the involution.

5 Conclusion

We have shown how to use random testing and bounded exhaustive testing to validate Coq definitions and theorems. The bounded exhaustive testing is based on logical specifications. It is assisted by a validation library in Prolog. We have applied these methods on two case studies. The second case study is also an original formalization of rooted maps with an interactive theorem prover. It directly encodes the combinatorial definition of a rooted map (as a transitive pair of injective endofunctions) and two basic operations for constructing them from smaller ones. The properties that these operations preserve permutations and transitivity are formalized, validated by random and bounded exhaustive testing, and then proved with some interactivity.

These two case studies about combinatorial structures show that logic programming features make Prolog an effective tool for prototyping and validating this kind of Coq code. Our focus is more on the design and validation methodology than on the resulting algorithms. The present work is intended to serve as a methodological guideline for further studies, in particular with other families of combinatorial objects.

Acknowledgments. The authors warmly thank the anonymous referees for suggestions, Noam Zeilberger for fruitful discussions and Valerio Senni for advice about his validation library.

References

1. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) Software Engineering and Formal Methods (SEFM 2004), pp. 230–239. IEEE Computer Society (2004)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, New York (2004)
3. Blanchette, J.C., Nipkow, T.: Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
4. Brun, C., Dufourd, J., Magaud, N.: Designing and proving correct a convex hull algorithm with hypermaps in Coq. Comput. Geom. **45**(8), 436–457 (2012)
5. Bulwahn, L.: The new quickcheck for Isabelle - Random, exhaustive and symbolic testing under one roof. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 92–108. Springer, Heidelberg (2012)
6. Carlier, M., Dubois, C., Gotlieb, A.: Constraint Reasoning in FOCALTEST. In: International Conference on Software and Data Technologies (ICSOFT 2010), Athens, July 2010
7. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, SIGPLAN Not., vol. 35, pp. 268–279. ACM, New York (2000)
8. Dubois, C., Mota, J.M.: Geometric modeling with B: formal specification of generalized maps. J. Sci. Pract. Comput. **1**(2), 9–24 (2007)
9. Dufourd, J.: Design and formal proof of a new optimal image segmentation program with hypermaps. Pattern Recogn. **40**(11), 2974–2993 (2007)
10. Dufourd, J.: Polyhedra genus theorem and Euler formula: a hypermap-formalized intuitionistic proof. Theor. Comput. Sci. **403**(2–3), 133–159 (2008)
11. Dufourd, J.: An intuitionistic proof of a discrete form of the Jordan curve theorem formalized in Coq with combinatorial hypermaps. J. Autom. Reasoning **43**(1), 19–51 (2009)
12. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 188–203. Springer, Heidelberg (2003)
13. Edmonds, J.R.: A combinatorial representation for oriented polyhedral surfaces. Notices Amer. Math. Soc. **7**, 646 (1960)
14. Giorgetti, A., Senni, V.: Specification and Validation of Algorithms Generating Planar Lehman Words, June 2012. https://hal.inria.fr/hal-00753008
15. Gonthier, G.: A computer checked proof of the Four Colour Theorem (2005). http://research.microsoft.com/gonthier/4colproof.pdf
16. Gonthier, G.: The four colour theorem: engineering of a formal proof. In: Kapur, D. (ed.) ASCM 2007. LNCS (LNAI), vol. 5081, pp. 333–333. Springer, Heidelberg (2008)
17. Hritcu, C., Lampropoulos, L., Dénès, M., Paraskevopoulou, Z.: Randomized property-based testing plugin for Coq. https://github.com/QuickChick
18. Kitaev, S.: Patterns in Permutations and Words. Springer, New York (2011)
19. Lando, S.K., Zvonkin, A.K.: Graphs on Surfaces and Their Applications. Springer, New York (2004)

20. Lazarus, F.: Combinatorial graphs and surfaces from the computational and topological viewpoint followed by some notes on the isometric embedding of the square flat torus (2014). http://www.gipsa-lab.grenoble-inp.fr/~francis.lazarus/Documents/hdr-Lazarus.pdf
21. Mathematical Components team: Library mathcomp.ssreflect.fingraph. http://math-comp.github.io/math-comp/htmldoc/mathcomp.ssreflect.fingraph.html
22. Owre, S.: Random testing in PVS. In: Workshop on Automated Formal Methods (AFM) (2006)
23. Paraskevopoulou, Z., Hritcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C.: Foundational property-based testing. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 325–343. Springer, Heidelberg (2015)
24. Pugh, W.: The Omega test: A fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing 1991, pp. 4–13. ACM, New York (1991)
25. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008, pp. 37–48 (2008). http://doi.acm.org/10.1145/1411286.1411292
26. Seidel, E.L., Vazou, N., Jhala, R.: Type targeted testing. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 812–836. Springer, Heidelberg (2015)
27. Senni, V.: Validation library. https://subversion.assembla.com/svn/validation/
28. SWI: Prolog. http://www.swi-prolog.org/
29. The OEIS Foundation Inc: The On-Line Encyclopedia of Integer Sequences. https://oeis.org/A000698
30. Tutte, W.T.: What is a map? In: Harary, F. (ed.) New Directions in the Theory of Graphs: Proceedings, pp. 309–325. Academic Press, New York (1973)
31. Tutte, W.T.: Combinatorial oriented maps. Canad. J. Math. **31**(5), 986–1004 (1979)