Published in Formal Aspects of Computing (2018)
`https://doi.org/10.1007/s00165-018-0459-1`

Tests and Proofs for Custom Data Generators

Catherine Dubois[1] and Alain Giorgetti[2]
[1]Samovar, ENSIIE, CNRS, Évry, France
[2]FEMTO-ST institute, Univ. of Bourgogne Franche-Comté, CNRS, Besançon, France

Abstract. We address automated testing and interactive proving of properties involving complex data structures with constraints, like the ones studied in enumerative combinatorics, e.g., permutations and maps. In this paper we show testing techniques to check properties of custom data generators for these structures. We focus on random property-based testing and bounded exhaustive testing, to find counterexamples for false conjectures in the Coq proof assistant. For random testing we rely on the existing Coq plugin QuickChick and its toolbox to write random generators. For bounded exhaustive testing, we use logic programming to generate all the data up to a given size. We also propose an extension of QuickChick with bounded exhaustive testing based on generators developed inside Coq, but also on correct-by-construction generators developed with Why3. These tools are applied to an original Coq formalization of the combinatorial structures of permutations and rooted maps, together with some operations on them and properties about them. Recursive generators are defined for each combinatorial family. They are used for debugging properties which are finally proved in Coq. This large case study is also a contribution in enumerative combinatorics.

Keywords: interactive theorem proving; random testing; bounded-exhaustive testing; logic programming; combinatorial enumeration; permutations; rooted maps

1. Introduction

We address automated testing of programs and properties manipulating data structures with constraints (such as duplicate-free lists or red-black trees) hereafter called *structured data*. A challenge is to provide effective random and exhaustive generators of structured data, controlled by some bound on the number or size of generated data. Some general testing tools can generate data or derive effective generators from data definitions, using techniques such as constraint solving or local choice with backtracking (see Section 7 for related work). When these general-purpose generation tools reach their limits (being too slow to generate data, failing in the derivation of a generator, or deriving an insufficiently efficient generator), a *custom generator* can be designed, for each family of structured data with many applications. Typical examples are lambda terms [PCRH11, Tar15] and event structures [BC17].

Our aim is to apply formal methods to design, specify, implement and check custom generators. We

---

Correspondence: Alain Giorgetti. e-mail: alain.giorgetti@femto-st.fr propose to check them by testing, with test data generated by other generators similarly designed and checked, and by proofs of some expected properties, themselves first tested to avoid trying to prove false conjectures.

When a datatype is algebraic, it is easy to derive from its recursive definition a *recursive generator*, which generates data of a given size from smaller data. Classical examples are lists, trees, or words following a context-free grammar. However, most data structures must also satisfy constraints, or *invariants*, which complicates the efficient generation of data. The approach we promote is to find a recursive description of these structures, and then derive a recursive generator for them. We illustrate this approach with examples of increasing complexity.

A large amount of research in enumerative combinatorics is devoted to the discovery of a recursive description of given combinatorial structures. A combinatorial contribution of the present work is the design of a recursive generator for rooted maps (in Section 5).

We carry out formalization and interactive proof within the framework of the theory of dependent types, with the Coq proof assistant [BC04]. Unless the proof of a conjecture is trivial, it is common to test lemmas and theorems before proving. Main validation methods are random(ized) testing, bounded exhaustive testing (BET) [Bul12] and finite model finding [BN10]. In the following we deal with random testing and BET. For random testing we use the QuickChick plugin [PHD$^+$15, HLDP18] for Coq. 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 two ways. First, in a lightweight way, we exploit the features of logic programming, implemented in a Prolog system. Thanks to backtracking, data structure invariants written in Prolog can often be used for free as bounded exhaustive generators. Then, in a more integrated way in Coq, we extend the random property testing tool QuickChick with BET.

The paper brings together two kinds of contributions: contributions to software engineering, with a presentation of random and bounded-exhaustive testing tools for the generation of counterexamples in a proof assistant (Section 2) and contributions to enumerative combinatorics, with a case study defining generators for families of combinatorial structures related to permutations and rooted maps (Sections 3 to 5). The main contributions in property-based testing are:

- a non-trivial application of random testing with QuickChick, to debug Coq specifications,
- an extension of QuickChick with BET (providing the new commands SmallCheck and SmallCheckWhy3), and
- a method of BET based on logic programming.

These tools helped us to design and check custom generators for lists of natural numbers with bounded values (illustrating example in Section 2), permutations as injective endofunctions (Section 3), permutations encoded by reversed subexcedant sequences (Section 4), and rooted maps encoded with a recursive dependent type (Section 5).

With respect to the previous work by Dubois, Giorgetti and Genestier [DGG16], the present paper additionally provides the BET extension of QuickChick and extends the case study to subexcedant sequences and rooted maps. Moreover the emphasis here is put on designing and checking generators, whereas the previous work was focussing on the benefits of tests and formal proofs for enumerative combinatorics.

The remainder of the paper is organized as follows. Section 6 reports some statistics on our tested and proved implementation of custom generators. Section 7 describes related work, and Section 8 concludes.

2. Testing Coq Conjectures

This section presents our methodology for testing Coq specifications. Before investing time in proving invalid conjectures we want to check their validity. Coq is presented in Section 2.1.

Property-based testing (PBT) is popular for functional languages, as exemplified by QuickCheck [CH00] in Haskell. PBT has also been adopted by proof assistants, e.g., Isabelle [BN04], Agda [DHT03], PVS [Owr06], FoCaLiZe [CDG10] and more recently Coq [PHD$^+$15]. We consider here two kinds of PBT: random testing (in Section 2.2) and bounded exhaustive testing (in Sections 2.3 to 2.5).

2.1. Brief Presentation of Coq

The Coq tool [BC04, Coq17] is a proof assistant, allowing the user to define objects (mathematical objects, data structures, functions and programs), write statements about these objects and prove these statements. Coq is an interactive theorem prover, meaning that making proofs with Coq is not an automatic activity. It is done with the help of tactics which transform a proof goal into a set of subgoals such that solving these subgoals is sufficient to solve the original goal. Some of them are very basic, close to elementary logical rules, some are more elaborate, close to decision procedures, like omega for solving linear arithmetic goals. Coq is based on the Calculus of Inductive Constructions (CIC), a very powerful type theory aiming at representing both ML-like functional programs and proofs in intuitionistic higher-order logic. It includes in particular polymorphic types, dependent types and inductive types defined by constructors. Coq inductive types cover both ML datatypes and Prolog-style relations. Functions can be defined using recursion and pattern-matching. Proofs can be done if necessary by induction or case analysis. Following Curry-Howard isomorphism, proofs are programs in CIC, thus making a proof via tactics builds a program whose type-checking validates the proof.

It is important to note, for our testing purpose, that Coq objects are sorted according to two different categories, the Prop sort and the Type sort. The former is dedicated to logical facts (e.g., odd 3, 2 = 4) while the latter is for mathematical objects and data structures (e.g., nat, list , bool). It means that a binary relation of type nat → nat → Prop and the corresponding Boolean function of type nat → nat → bool are not the same but might be proved “equivalent” if the relation is decidable. A proposition in Prop usually cannot be computed whereas a Boolean expression can be computed.

Last but not least, Coq features an extraction mechanism from Coq proofs and definitions to OCaml (or Haskell) programs. Roughly speaking, the computational parts are translated in OCaml while the logical parts are erased. This feature is fundamental for QuickChick and its extensions used in the rest of the paper: QuickChick extracts OCaml code from Coq code and runs tests in OCaml outside Coq for better efficiency.

In the rest of the paper, we explain Coq constructions as and when needed.

2.2. Random Testing

2.2.1. *QuickChick and the General Workflow*

QuickChick [HLDP18] 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.

We consider conjectures of the form

$$\forall \mathsf{x}\colon \mathsf{T},\ \mathsf{precondition\ x} \rightarrow \mathsf{conclusion\ (f\ x)}, \tag{1}$$

where precondition and conclusion are logical predicates of type T → Prop. The general workflow that we follow to validate these conjectures by testing starts with the definition of a random generator gen_T of values of type T that satisfy the property precondition. However the predicate conclusion is not executable. We must turn it into a Boolean function — named conclusionb — that is more adapted for testing. We must prove that the Boolean function is semantically equivalent to the logical predicate. If we cannot provide an executable version of the logical predicate conclusion, then QuickChick does not apply.

The test is run by using the command

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

Its execution generates a fixed number of inputs using the generator gen_T and applies the function f to each of them to verify the property under test (conclusion).

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 require a large proof effort. In the following we propose to test that the generator produces correct outputs. For this purpose, we also turn the logical property precondition into an executable one, named preconditionb.

2.2.2. Lists of Bounded Natural Numbers

We now illustrate QuickChick features and our approach on constrained lists of natural numbers. Let us notice that QuickChick heavily uses type classes and monads. However, in the following we explain very informally some piece of code.

In the example below, we manipulate lists whose elements are natural numbers strictly smaller than a given bound b. We call such a list a *(b-)bounded list*, or *blist* for short. Thus T in (1) is here list nat, where the inductive type nat of Peano natural numbers is defined with the constructors 0 for zero and S for the successor operation, and the inductive polymorphic type list is defined with the constructors nil for the empty list and cons (also denoted by ::) for adding an element. The predicate precondition of (1) is here the following inductively defined predicate:

```
Inductive is_blist (b : nat) : list nat → Prop :=
| Blist_nil: is_blist b nil
| Blist_cons : ∀ v l, v < b → is_blist b l → is_blist b (v::l).
```

This predicate is not executable. However we can define the Boolean function is_blistb and prove its correctness with respect to the logical predicate is_blist.

```
Fixpoint is_blistb (b : nat) (l : list nat) :=
 match l with
   nil   ⇒ true
 | v::l' ⇒ (ltb v b) && (is_blistb b l')
 end.
```

```
Lemma is_blist_dec : ∀ b l, (is_blistb b l = true ↔ is_blist b l).
```

The function is a recursive function defined by pattern-matching on the list l, where ltb is the Boolean function equivalent to the logical predicate $<$. The correctness proof, omitted here, is done by induction on the list l.

We now define a generator of blists parameterized by the length n of the list to be generated and the bound b.

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

The generator is defined by pattern-matching on n and b: if n or b is 0, the output is the empty list. Otherwise ($n = n' + 1$ and $b = b' + 1$) the recursive call (genBlistAsListnat n' b) generates a b-bounded list of length n' which is extended by a number m arbitrarily chosen in the interval $[0..b-1]$ (using the combinator choose). The combinator liftGen applies monadic lifting, and do! m ← . . . ; . . . is the usual monadic notation to bind a result and go on with the next computation.

Notice that the generator always returns a b-bounded list of length n if $b \neq 0$. However, for $b = 0$, it always returns the empty list nil of length 0, whatever the value of its input parameter n. This design choice is not important in practice because generating 0-bounded lists has no interest. The case $b = 0$ is only the basis of the recursive definition of the generator. So in the following we'll use the generator with bounds strictly greater than 0.

To have confidence in this generator of b-bounded lists, we test the well-formedness of the outputs, i.e., that they contain elements strictly smaller than the bound b. Below, the first QuickCheck command checks that the 10-bounded lists generated by the generator are well-formed. The second test not only varies the length of the generated list but also arbitrarily picks up the bound. In this latter test, we use two generators, one for the bound (arbitraryNat) and another for blists (genBlistAsListnat).

```
QuickCheck (sized (fun n ⇒
  forAll (genBlistAsListnat n 10) (fun l ⇒ (is_blistb 10) l))).
+++ Passed 10000 tests (0 discards)
```

```
QuickCheck (sized (fun n ⇒
 forAll arbitraryNat (fun b ⇒
 forAll (genBlistAsListnat n b) (fun l ⇒ is_blistb b l)))).
+++ Passed 10000 tests (0 discards)
```

The maximal number of tests (10,000 here) can be adjusted by the user. We iterate over different values for n thanks to the use of the combinator sized.

```
Fixpoint lift3 (n : nat) (p : nat) (l : list nat) {struct l} :=
 match p, l with
   0, _          ⇒ n :: l
 | _, nil        ⇒ n :: nil
 | S p', a :: l' ⇒ a :: (lift3 n p' l')
 end.

Definition lift (p : nat) (l : list nat) := lift3 (length l) p l.

Lemma lift_blist_max: ∀ l b p,
 is_blist b l → is_blist (max b (S (length l))) (lift p l).
```

Listing 1: Functions and conjectures about lists of natural numbers.

Let us test a first conjecture, about the operation lift defined in Listing 1. This operation will be useful in the next case studies. The operation lift is such that (lift p l) inserts the length of the list l at position p in l, thanks to the more general operation lift3 which is such that (lift3 n p l) inserts n at position p in the list l. We want to test the conjecture lift_blist_max about preservation of blists by the operation lift. It claims that the application of lift on a blist returns a blist, but with a change in the bound that becomes the maximum of the initial bound and the successor of the list length.

So, as previously, we randomly generate the size (n), the bound (b) of the blist, the b-bounded list l with length n and the position p where the list length n is inserted in the blist (with lift).

```
QuickCheck (sized (fun n ⇒
 forAll arbitraryNat (fun b ⇒
 forAll (genBlistAsListnat n b) (fun l ⇒
 forAll arbitraryNat (fun p ⇒ is_blistb (max b (S (length l))) (lift p l))))))
+++ Passed 10000 tests (0 discards)
```

2.2.3. *Counterexamples*

What happens if there is an error in the conjecture, e.g., a wrong bound in the conclusion? To illustrate the behavior of QuickChick in such a case, we inject such an error in the conjecture lift_blist_max that becomes the following one:

```
Lemma lift_blist_error: ∀ l b p,
 is_blist b l → is_blist (S b) (lift p l).
```

QuickChick discovers an error after generating 5 test cases. It displays the counterexample $b = 2$, $l = [1, 0, 1, 1]$ and $p = 3$ as follows:

```
QuickCheck (sized (fun n ⇒
 forAll arbitraryNat (fun b ⇒
 forAll (genBlistAsListnat n b) (fun l ⇒
 forAll arbitraryNat (fun p ⇒ is_blistb (S b) (lift p l)))))).
2
[1, 0, 1, 1]
3
*** Failed after 5 tests and 0 shrinks. (0 discards)
```

Just like Haskell's QuickCheck, QuickChick also supports shrinking and thus tries to isolate the part of the failing input that triggers the failure. We do not emphasis on this point here.

2.2.4. *Automatic Derivation of Generators*

Recently QuickChick functionalities have been extended with the possibility to automatically derive a generator from the definition of the precondition predicate, when the latter is inductively defined [LPP18]. Another new functionality is the automatic generation of the correctness proof of the generator.

However some issues were raised during our attempts with the actual prototype. The first issue comes from the restrictions on the shape of the inductive definition of the precondition. The definition of is_blist given previously cannot be used for deriving a generator because $<$ (from the Coq standard library) is not inductively defined but defined with respect to $\leq$ which is an inductively defined predicate. We circumvent this limitation by writing another inductive definition based on $\leq$ and proving the equivalence with the previous definition. Listing 2 presents the new inductive predicate is_blist_le, the equivalence lemma between the two definitions and the command Derive for deriving the random generator. In this command, the free variable b indicates that the bound will be an input of the generator while the bound variable l, the bounded list, will be the output of the generator. This command introduces a class instance named GenSizedSuchThatis_blist_le which provides a function named arbitrarySizeST which is the derived generator. The function derived_genBlistAsListnat simplifies the access to this generator. If no data satisfy the precondition, its output is the None value. Otherwise its output is (Some l), where l is a list of natural numbers satisfying the predicate is_blist_le. The QuickCheck command (omitted here) for testing conjectures is a bit more complex because the derived generator does not produce lists of natural numbers but values of type option (list nat).

```
Inductive is_blist_le : nat → list nat → Prop :=
| Blist_le_nil : ∀ b, is_blist_le b nil
| Blist_le_cons : ∀ v l b,
   v ≤ b → is_blist_le (S b) l
   → is_blist_le (S b) (v :: l).

Lemma is_blist_eq_def: ∀ l b, is_blist b l ↔ is_blist_le b l.

Derive ArbitrarySizedSuchThat for (fun l ⇒ is_blist_le b l).

Definition derived_genBlistAsListnat (n b : nat) : G (option (list nat)) :=
arbitrarySizeST _ (is_blist_le b) (GenSizedSuchThatis_blist_le b) b.
```

Listing 2: Deriving a generator with QuickChick.

A more serious issue for us is that the prototype is not able to deal with dependent types. In the following we go on with manually written generators.

2.2.5. Endofunctions in One-Line Notation

Our interest for bounded lists is motivated by the following notion: the *one-line notation* of a function f on $[0 \dots n-1]$ is the list $[f(0); f(1); \dots; f(n-1)]$ of its images. So, b-bounded lists represent functions from $[0 \dots n-1]$ to $[0 \dots b-1]$ by their one-line notation. In particular, bounded lists whose bound equal their length n — hereafter called *endolines* — encode *endo*functions on $[0 \dots n-1]$ by their one-*line* notation. This property is formalized by the logical predicate is_endoline and the equivalent Boolean function is_endolineb reproduced in Listing 3. The listing also shows invariance lemmas that we have easily proved as specializations of similar lemmas for bounded lists, first randomly tested with QuickChick and then proved by induction on lists.

```
Definition is_endoline (l : list nat) := is_blist (length l) l.
Definition is_endolineb (l : list nat) := is_blistb (length l) l.
Lemma is_endoline_dec : ∀ l, (is_endolineb l = true ↔ is_endoline l).

Lemma cons_endo: ∀ n (f : list nat),
 n ≤ length f → is_endoline f → is_endoline (n::f).

Lemma lift_endo: ∀ p (f : list nat), is_endoline f → is_endoline (lift p f).
```

Listing 3: Characterization of endolines and two invariance lemmas.

2.3. Bounded Exhaustive Testing with Prolog

For testing Coq specifications we also advocate 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 combinatorial object generation. Secondly BET provides the author of a wrong lemma with the smallest counterexample revealing her error. (This benefit is somewhat mitigated by the shrinking — counterexample size reduction — feature of QuickChick.) Thirdly many inductive structures with properties can be easily specified 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 testing tasks easier, we extend a Prolog validation library created by Valerio

Senni [Sen18] and previously applied to test algorithms on words encoding rooted planar maps [GS12]. The library returns counterexamples (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 library features on the example of endolines. The reader is assumed to be familiar with logic programming, or can otherwise read a short summary in [GS12].

We take again the example of endofunctions in one-line notation. We encode a function f on $[0..n-1]$ by the Prolog list `[`$f(0)$`,...,`$f(n-1)$`]` of its values, called its one-line notation. Listing 4 shows a Prolog predicate `line_endo` such that the formula `line_endo(L,N)` (resp. `line_endo(L,N,K)`) holds if and only if `L` is a list of length `N` with elements in `[0..N-1]` (resp. `[0..K]`). 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 if and only if the integer `K` is in the interval [`I`..`J`].

```
line_endo([],0,_).
line_endo([V|M],N,K) :- N > 0, Nm1 is N-1, in(V,0,K), line_endo(M,Nm1,K).
line_endo(L,N) :- Nm1 is N-1, line_endo(L,N,Nm1).
```

Listing 4: Endofunctions in Prolog.

A clear advantage of logic programming is that the predicate `line_endo` works in two ways: as an *acceptor* of endofunctions, and as a *generator* enumerating all of them for a given length. The query scheme

```
?- p(L,n), write_coq(L), fail.
```

indeed allows the enumeration of all the data of a given size n accepted by a characteristic predicate p. 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_endo(L,3), write_coq(3), fail.
```

writes a Coq line such as

```
Eval compute in (is_endolineb 3 [2;1;1]).
```

for each endofunction of length 3. Each line provokes the evaluation inside Coq of the expression written between parentheses. These lines constitute a test suite for the Coq function `is_endolineb`, under the assumption that the Prolog program in Listing 4 is correct. This assumption can be checked in two ways: by visual inspection of the lists it generates, or by counting. For counting, the validation library provides the predicate `iterate` so that the query

```
?- iterate(0,6,line_endo).
```

outputs the numbers 1, 1, 4, 27, 256, 3125 and 46656 of distinct lists of length n from 0 to 6 accepted by the predicate `line_endo`. We easily recognize the first numbers n^n of endofunctions of length n.

We can now adapt the predicate `write_coq` to the BET of the lemmas in Listing 3. For Lemma cons_endo the query evaluation can generate in a Coq file all the Coq lines of the form

```
Eval compute in (is_endolineb (cons i l)).
```

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

2.4. Extension of QuickChick with BET

We propose to add a Coq command named SmallCheck (after SmallCheck [RNL08]) to do BET inside Coq, reusing the QuickChick mechanisms to execute test cases that relies on OCaml extraction. We also rely on the same hypothesis that the user has to provide an equivalent Boolean version of the predicate she wants to test.

The general workflow that we follow to validate by BET a conjecture like (1) starts with the definition of a Java-style iterator iterator_T that produces iteratively all the values of type T up to size n that satisfy the property precondition. Let conclusionb be the Boolean version of conclusion. The test is run by using the command

```
SmallCheck (iterator_T) (fun x ⇒ conclusionb (f x))
```

which applies the function f on each value produced by the iterator and verifies the property under test (conclusion).

We consider here iterators modifying a structure, here a state, to build the next structure of the same size, without needing any other information. The iterator iterator_T has the type It T defined by the following polymorphic dependent record:

```
Record It {A : Type} : Type := mkIt {
 st_t : Type;
 start : st_t;
 next : st_t → option A * st_t;
 hasnext : st_t → bool
}
```

where the field st_t denotes the type of the internal state for the iterator, start is the initial state, next returns the next state and the output value that will be used as a test case if they exist, and a Boolean predicate hasnext returns true if there are more elements, false otherwise. Notice that the three last fields depend on the first one.

As a first example, let us define an iterator allowing the enumeration of all natural numbers from 0 to n, as a record parameterized by n:

```
Definition it_interval (n : nat) := {|
  st_t := nat * nat;
  start := (0, n);
  next := fun st ⇒ match st with (cur, up) ⇒ (Some (cur+1), (cur+1, up)) end;
  hasnext := fun st ⇒ match st with (cur, up) ⇒ ltb cur up end
|}.
```

We use it first to check that each enumerated number is less than or equal to 7, then to check that each enumerated number is less than or equal to 6: The former command succeeds with the result Success while the latter fails indicating the first value encountered in the interval that falsifies the property.

```
SmallCheck (it_interval 7) (fun x : nat ⇒ leb x 7).
Success

SmallCheck (it_interval 7) (fun x : nat ⇒ leb x 6).
Failure: 7
```

2.5. BET with Cursors Proved with Why3

We propose to implement iterators as cursors, as defined by Filliâtre and Pereira [FP16]. A *cursor* is composed of three functions: a constructor named create_cursor initiates the cursor to the first element of the iteration, a function has_next returns a Boolean indicating the existence of a next element in the iteration, and a function next moves to the next element and returns it. Filliâtre and Pereira defined cursors to traverse a collection such as an array, a list, a set, a binary tree or a graph. We adapt their work to exhaustive generation and define cursors to iterate over the inhabitants of a datatype or interest for testing.

In order to get trustable cursors we specify and implement them in Why3 and thus formally prove some of their expected properties. Why3 [FP13] is a platform dedicated to deductive verification. We can write WhyML programs with formal specifications and get correct-by-construction OCaml programs via an automated extraction mechanism. WhyML comes with a rich library of data structures like mutable arrays. Our choice of Why3 (instead of Coq) to define cursors is motivated by the imperative nature of the language and the existence of mutable data structures for generator efficiency.

We illustrate our approach with the example of a cursor for bounded arrays, presented in Listings 5 and 6. A *(b-)bounded array*, or *barray* for short, is an array of natural numbers strictly smaller than a given bound b. The first-order formula (is_barray a k b) formalizes this property for the subarray $a[0..k-1]$.

```
 predicate is_barray (a:array int) (k:int) (b:int) = ∀ i:int. 0 ≤ i < k→ 0 ≤ a[i] < b
 predicate cte_array (a:array int) (k:int) (b:int) = ∀ i:int. 0 ≤ i < k→ a[i] = b
```

```
type cursor = {
          arr: array int; (* current array *)
        bound: int;       (* strict upper bound for array values *)
 mutable rank: int;       (* arr is the rank-th generated array *)
 mutable last: bool;      (* true iff c.arr is the last array *)
}

predicate sound (c: cursor) = c.bound ≥ 1 && c.rank ≥ 0 && is_barray c.arr (length c.arr) c.bound

let create_cursor (n: int) : cursor
 requires { n > 0 }
 ensures  { length result.arr = n }
 ensures  { result.bound = n }
 ensures  { cte_array result.arr (length result.arr) 0 }
 ensures  { sound result }
=
 let a = make n 0 in {
  arr = a;
  bound = n;
  rank = 0;
  last = (n ≤ 1);
 }

predicate completed (c: cursor) = cte_array c.arr (length c.arr) (c.bound-1)

let has_next (c: cursor) : bool
 ensures { result ↔ not (completed c) }
=
 c.last ← true;
 let t = ref (0) in begin
  while !t < length c.arr && c.last do
   invariant { 0 ≤ !t ≤ length c.arr }
   invariant { cte_array c.arr !t (c.bound-1) }
   invariant { completed c → c.last }
   if (c.arr[!t] = c.bound-1) then
    t := !t + 1
   else
    c.last ← false
  done
 end;
 not c.last
```

Listing 5: Generator of bounded arrays in Why3, part 1.

The cursor stores the current array (field arr), the (strict) upper bound b for its elements (field bound), the rank of the current array during generation (field rank), from 1 to the number b^n of functions from $[0..n-1]$ to $[0..b-1]$, where n is the array length, and a Boolean flag (field last) indicating the end of the generation. The bound and the rank are declared as signed integers (type int) because there is no type for natural numbers in WhyML. However they are respectively assumed to be a positive and a nonnegative integer, as requested in the first part of the soundness condition for the cursor (predicate sound). (There is no array if $b \leq 0$ and the case $b = 1$ is excluded because there is only one array $\boxed{0}\boxed{\ldots}\boxed{0}$ to generate.) In addition to these two conditions, the cursor is sound if all the elements in its current array are strictly lower than its bound (last condition of the predicate sound).

The function create_cursor initializes the cursor with the smallest array $\boxed{0}\boxed{\ldots}\boxed{0}$, which is the last one if and only if the array length n is 1. For the sake of simplicity the case $n = 0$ is excluded and the presented code is limited to the case where the bound b equals the array length n. (These bounded arrays encode endofunctions in one-line notation.) The rank is set to the unsound value 0 so that the function next (Listing 6) detects this initial case and sets it to its right value 1.

The arrays are generated in increasing lexicographic order. So, the generation ends when all the array elements have reached their maximal value $b-1$. This condition is specified by the predicate completed, and the function has_next detects that it does hold yet.

The function next in Listing 6 steps from one bounded array to the next one. For all cases but the first one where the rank is 0, it works as follows. The first loop goes through the array c.arr from right to left to find its rightmost non-maximal element, that is, the maximal array index r such that c.arr$[!r] < b-1$. (The variables t and r are references and ! is the dereferencing operator.) If the search fails, the current array is the last one. Otherwise the function increments this non-maximal element c.arr$[!r]$ and the second loop fills out the upper part c.arr$[!r+1..]$ of the array with zeros.

```
 let next (c: cursor) : array int
  requires { sound c }
  requires { not (completed c) }
  ensures  { sound c }
  ensures  { old c.rank ≤ c.rank ≤ old c.rank + 1 }
 =
  if c.rank = 0 then (* First array *)
   c.rank ← 1
  else begin (* For all arrays but the first one *)
   let r = ref (-1) in
   let t = ref (length c.arr - 1) in
   (* Find the index (!r) of the rightmost digit that can be incremented: *)
   while !t ≥ 0 do
    invariant { -1 ≤ !t < length c.arr }
    invariant { -1 ≤ !r < length c.arr }
    invariant { !t ≥ 0 → ∀ i:int. !t < i < length c.arr → c.arr[i] ≥ c.bound-1 }
    invariant { !r  = -1 → ∀ i:int. !t < i < length c.arr → c.arr[i] ≥ c.bound-1 }
    invariant { 0 ≤ !r < length c.arr → c.arr[!r] < c.bound-1 }
    variant   { !t + 1 }
    if (c.arr[!t] < c.bound-1) then begin
     r := !t;
     t := -1
    end else
     t := !t - 1
   done;
   if !r < 0 then (* last array reached. *)
    c.last ← true
   else begin
    c.arr[!r] ← c.arr[!r] + 1;
    (* Fill the suffix starting at c.arr[!r+1] with zeros: *)
    for i = !r+1 to length c.arr - 1 do
     invariant { !r+1 ≤ i ≤ length c.arr }
     invariant { is_barray c.arr i c.bound }
     c.arr[i] ← 0
    done;
    c.rank ← c.rank + 1;
    c.last ← false
   end
  end;
  c.arr
end
```

Listing 6: Generator of bounded arrays in Why3, part 2.

The loop contracts specify the interval of values of the integer variables modified by the loop, and invariants for the array content that are strong enough to entail the postconditions (ensures clauses). Under the precondition requires { sound c } that the input array is bounded, the postcondition ensures { sound c } specifies that the output array is also bounded (*soundness* property).

This code is fully proven with Alt-Ergo [BCCL08]. More specification and proof efforts are required to prove the properties of *monotonicity* — the function next always generates an array higher than its input array, in lexicographic order, *completeness* — all bounded arrays are generated, and *duplicate-freedom* — no array is generated twice. This program is an adaptation to Why3 of a C program that is part of a library of 12 proved generators implemented in C, formally specified in ACSL, and automatically proved by the WP plugin within the Frama-C framework [GGP15].

Why3 allows the automated extraction of correct-by-construction OCaml programs from Why3 programs. We exploit this mechanism together with our BET command. So we define as follows a Coq iterator of type It with an abstract type for the cursor (introduced as a parameter below) and abstract functions create, next_list and hasnext (also introduced as parameters) . These type and functions will be linked with the OCaml respective type and functions genrated by Why3. Consequently it means that arrays will be used as in Why3 but any array generated by Why3 will be transformed into a list of natural numbers.

```
Parameter cursor : Type.
Parameter create : nat → cursor.
Parameter next_list : cursor → (option (list nat)) * cursor.
Parameter hasnext : cursor → bool.

Definition it_endo (n : nat) := mkIt (list nat)
  cursor (create n) next_list hasnext.
```

We introduce the variant SmallCheckWhy3 of the previous SmallCheck command, very close but linking, at extraction time, with the OCaml files generated by Why3.

For instance Lemma cons_endo (Listing 3) can be verified by BET with every value n in $[0..8]$ and every endofunction f of length 7 or less by

```
SmallCheckWhy3  (it_interval 7) (fun lf ⇒
 (smallCheckb (it_endo lf) (fun f ⇒
 (smallCheckb (it_interval lf)  (fun n  ⇒ is_endolineb (n::f)))))).
Success
```

In the previous command, smallCheckb performs as its counterpart SmallCheckWhy3 but is introduced for combining several iterations.

With SmallCheckWhy3 one can test exhaustively up to some bound the conjectures randomly tested by QuickCheck.

3. First Example: Permutations As Injective Endofunctions

Permutations on a finite set form an elementary but central combinatorial family. It is well known that any injective endofunction on a finite domain is a permutation. We could formalize permutations by adding a constraint to endofunctions in one-line notation introduced in Section 2. However, several operations on permutations, such as composition, are more easily defined with permutations seen as functions, rather than as lists of their values. Therefore we present here a formal study of permutations defined as Coq functions constrained to be injective endofunctions (Section 3.1). We discuss their random and bounded exhaustive generation in Section 3.2. For these permutations we define the operations of insertion, contraction and direct sum, by refining them from operations more generally defined on functions on natural numbers (Sections 3.3 to 3.5). This formalization and the operations of insertion and direct sum served as running example in previous work [DGG16]. The study is completed here with the operation of contraction (Section 3.4), new generators (Section 3.2), and more tested and proved properties (Sections 3.3 to 3.6).

The formalization of permutations in the library of mathematical components [Mat18] also relies on the property that permutations are injective endofunctions. However, as far as we know, no formal library defines the operations of insertion, contraction and direct sum. They are used in the case study of rooted maps in Section 5.

3.1. Characterization of Permutations

Listing 7 shows our Coq formalization of permutations. A permutation is defined as an injective function from an interval $[0..n-1]$ of natural numbers to itself. We consider functions defined on nat (hereafter called *natural functions*) but we only impose constraints for their values on the interval $[0..n-1]$, whatever their definition outside the interval. The predicates is_endo and is_inj respectively define the properties of being an endofunction and injectivity. A permutation is then a *record* structure composed of a natural function and the proofs that the latter satisfies the previous two 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 := {
 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.

Definition idNat : nat → nat := (fun (x : nat) ⇒ x).
Lemma id0_endo : is_endo 0 idNat. firstorder. Defined.
Lemma id0_inj  :  is_inj 0 idNat. firstorder. Defined.
Definition id0permut := MkPermut id0_endo id0_inj.
```

Listing 7: Permutations as injective endofunctions in Coq.

The listing also shows the definition of identity as a permutation id0permut of size 0, called the *empty permutation*. Its proof components are proved using the tactic firstorder dedicated to first-order reasoning.

3.2. Generation of Permutations

A list which is the one-line notation of some permutation is hereafter called a *permutation list*, or *permline*. We generate permutations as permutation 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.

The operation lift (Listing 1) is a classical way to construct a permutation on $[0..n]$ from a permutation p of size $[0..n-1]$, by inserting n in the one-line notation of p. The following random generator applies this operation:

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

This recursive generator is defined by pattern-matching on n: if n is 0, the output is the empty list. Otherwise ($n = n' + 1$) the recursive call (genPermlineAsListnat n') generates the one-line notation of a permutation on $[0..n'-1]$, and the operation lift inserts n in it, at some position p randomly chosen in $[0..n-1]$ (using the combinator choose).

To have confidence in this generator of permutations, we test that its outputs do not contain any duplicate, that their length is n and that their elements are natural numbers strictly smaller than n. These three conditions are implemented by the Boolean function is_permlineb (its code is omitted here).

```
QuickCheck (sized (fun n ⇒ forAll (genPermlineAsListnat n) (is_permlineb n))).
+++ Passed 10000 tests (0 discards)
```

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 genPermlineAsListnat. To this end, we implement Boolean versions is_endob, is_injb and is_permutb of the logical properties is_endo, is_inj and is_permut. Listing 8 shows the functions is_endob and is_permutb. An evaluation of (is_endob n f) returns true if and only if the function f is an endofunction on $[0..n-1]$. 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 connection 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 $[0..n]$ if and only if the list $[f(0); f(1); \ldots; f(n)]$ of its images has no duplicate.

```
Fixpoint is_endob_aux n f m :=
 match m with
   0   ⇒ if (lt_dec (f 0) n) then true else false
 | S m' ⇒ if (lt_dec (f m) n) then is_endob_aux n f m' else false end.
Definition is_endob n f :=
 match n with
   0   ⇒ true
 | S n' ⇒ is_endob_aux n f n' 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 8: Boolean functions for permutations.

3.3. Insertion

A permutation can also be considered as a finite composition of pairwise disjoint cycles, called its *cycle structure* [Sta97, Section 1.3]. For example the cycle structure of

$$p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 3 & 4 & 0 & 2 \end{pmatrix}$$

is (0 1 3) (2 4).

We now define insertion of one element in a cycle of the cycle structure of a permutation. This insertion is fundamental to define two operations for map construction (in Section 5). It is also used to define another recursive generator of permutations, in Section 4.

Let n be a natural number, p be a permutation on $[0..n-1]$ and i be a natural integer in $[0..n]$. The *insertion before i in p* is the permutation on $[0..n]$ obtained from p by adding the cycle (n) if $i = n$ or by inserting the integer n before the integer i in its cycle in p if $0 \leq i \leq n-1$. For example the insertion before 5 in the previous permutation p is the permutation (0 1 3) (2 4) (5), which adds the fixed point 5 to p. Its one-line notation is $[1; 3; 4; 0; 2; 5]$. The insertion before 4 in p is the permutation $(0\ 1\ 3)\ (2\ 5\ 4) = [1; 3; 5; 0; 2; 4]$. The insertion before 0 in p is the permutation $(0\ 1\ 3\ 5)\ (2\ 4) = [1; 3; 4; 5; 2; 0]$. This insertion in the cycle structure clearly differs from insertion in the one-line notation implemented by the Coq function `lift`, since the latter permutation $(0\ 1\ 3\ 5)\ (2\ 4) = [1; 3; 4; 5; 2; 0]$ cannot be obtained by insertion in the one-line notation $[1; 3; 4; 0; 2]$ of p.

We propose the following generalization of this operation to natural functions. For any natural number n let f be a natural function defined on $[0..n-1]$ and i a natural number. The *insertion before i in f* is the function f' defined on $[0..n]$ 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 $f(j) \neq i$.

Listing 9 presents an implementation in Coq of the operation of insertion in a natural function. Here we use two different ways to compare natural numbers: elimination of a decidability lemma (`le_lt_dec` or `eq_nat_dec`) and pattern-matching on the result of the `nat_compare` function that returns either `Eq`, `Lt` or `Gt` (for Equal, Less than or Greater than).

```
Definition insert_fun n (f : nat → nat) (i : nat) : nat → nat :=
 fun x ⇒ if le_lt_dec i n  (* Definition le_lt_dec n m : {n ≤ m} + {m < n}. *)
          then (* i ≤ n *)
           match nat_compare x n with
             Eq ⇒ (* x = n *) i
           | Lt ⇒ (* x < n *)
               if eq_nat_dec (f x) i (* Definition eq_nat_dec n m : {n = m} + ~ {m = n}. *)
                then (* f x = i *) n
                else (* f x ≠ i *) f x
           | Gt ⇒ (* x > n *) f x end
         else (* i > n *) x.
```

Listing 9: Insertion in Coq.

3.4. Contraction

We now define an inverse function for insertion. It is named *contraction* because it is a generalization to natural functions of contraction of permutations defined in [BLS18, Section 2.1].

Let n be a natural number and g be a natural function defined on $[0..n]$. The *contraction of g* is the function g' defined on $[0..n-1]$ as follows: if $x < n$ and $g(x) = n$ then $g'(x) = g(n)$; otherwise $g'(x) = g(x)$.

For example, the contraction of the permutation $(0\ 1\ 3)\ (2\ 5\ 4) = [1; 3; 5; 0; 2; 4]$ on $[0..5]$ is the permutation $p = (0\ 1\ 3)\ (2\ 4) = [1; 3; 4; 0; 2]$ on $[0..4]$.

The operation of contraction in a natural function is implemented in Coq by

```
Definition contraction_fun (n : nat) (g : nat → nat) : nat → nat :=
 fun x ⇒ if le_lt_dec n x
  then (* n ≤ x *) g x
  else (* n > x *)
   if eq_nat_dec (g x) n
   then (* g x = n *) g n
   else (* g x ≠ n *) g x.
```

3.5. Direct Sum

Insertion is the main ingredient of two operations for map construction defined in [DGG16]. A second ingredient for one of these operations is the direct sum of two permutations, generalized here to natural functions.

For any natural numbers n_1 and n_2 the *direct sum* of a natural function f_1 on $[0..n_1 - 1]$ and a natural function f_2 on $[0..n_2 - 1]$ is the function f on $[0..n_1 + n_2 - 1]$ such that

(a) $f(x) = f_1(x)$ if $0 \leq x < n_1$ and
(b) $f(x) = f_2(x - n_1) + n_1$ if $n_1 \leq x < n_1 + n_2$.

It is an extension of the well-known *direct sum* on permutations [Kit11, p. 57], denoted $\oplus$.

For example, let $p_1 = [2; 1; 0] = (0\ 2)\ (1)$ be a permutation on $[0..2]$ and $p_2 = [1; 3; 4; 0; 2] = (0\ 1\ 3)\ (2\ 4)$ a permutation on $[0..4]$. We then have $p_1 \oplus p_2 = [2; 1; 0; 4; 6; 7; 3; 5] = (0\ 2)\ (1)\ (3\ 4\ 6)\ (5\ 7)$.

The direct sum on natural functions is implemented in Coq by

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

3.6. Invariance Properties

The three operations on natural functions (insertion, contraction and direct sum) should first satisfy the *invariance property* that they preserve permutations. Their restriction to the type `permut n` for permutations is respectively named `insert`, `contraction` and `sum`.

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

Listing 10: Invariance properties for insertion, contraction, and direct sum.

The lemmas in Listing 10 state that insertion, contraction and direct sum on natural functions preserve permutations. Before proving them, they are conjectures that we can test with the usual methodology: (i) when a natural function representing a permutation is to be generated, we use the generator `genPermlineAsListnat` of permutations in one-line notation; (ii) the logical property under test is turned into its Boolean version composed with the translation function `list2fun`.

For example Lemma `insert_permut` is tested by

```
QuickCheck (sized (fun n ⇒
 forAll (genPermlineAsListnat n) (fun l ⇒
 forAll arbitraryNat (fun i ⇒ is_permutb (S n) (insert_fun n (list2fun l) i))))).
+++ Passed 10000 tests (0 discards)
```

If we inject a fault in the definition of `insert_fun` reproduced in Listing 9, e.g., replacing the result `n` by `S n` in the `Lt` case, we get a counterexample, e.g., $\mathsf{l} = [0; 1]$ and $\mathsf{i} = 0$ for $\mathsf{n} = 2$.

3.7. Detection of Missing Hypotheses

The next code snippet illustrates the way we can use this testing facility to discover missing hypotheses in a statement. For example we can try to test the following — wrong — conjecture that states that contraction preserves endofunctions:

```
Lemma contraction_fun_endo (n : nat) (f : nat → nat) :
 is_endo (S n) f → is_endo n (contraction_fun n f).
```

The corresponding random test

```
QuickCheck (sized (fun n ⇒
 forAll (genEndolineAsListnat (n+1)) (fun l ⇒ is_endob n (contraction_fun n (list2fun l ))))).
```

exhibits a counterexample which is a non-injective natural function:

```
[1, 2, 2]
*** Failed after 10 tests and 0 shrinks. (0 discards)
```

Here we use a simple generator genEndolineAsListnat of endofunctions in one-line notation: (genEndolineAsListnat n) produces a list of length n with elements in $[0..\mathsf{n}-1]$. This suggests to also require the injectivity of the function.

```
QuickCheck (sized (fun n ⇒
 forAll (genPermlineAsListnat (n+1)) (fun l ⇒ is_endob n (contraction_fun n (list2fun l))))).
+++ Passed 10000 tests (0 discards)

Lemma contraction_fun_endo (n : nat) (f : nat → nat) :
 is_endo (S n) f → is_inj (S n) f → is_endo n (contraction_fun n f).
```

The property then passes the test and the corrected lemma (contraction_fun_endo) is proved.

3.8. Relational Properties

The lemmas in Listing 11 state that contraction of permutations is a left inverse (or *retraction*) and a right inverse (or *section*) of insertion. The first lemma is a key ingredient for the proof of the cancellation lemma in Section 4. The predicate eq_natfun defines equality for functions on $[0..n-1]$ represented by natural functions. The predicate eq_permut defines equality on permutations represented by terms with type permut n.

```
Definition eq_natfun (n : nat) (f g : nat → nat) := ∀ i, i < n → f i = g i.
Definition eq_permut (n : nat) (p q : permut n) := eq_natfun n (fct p) (fct q).

Lemma contraction_insert_inv (n : nat) (p : permut n) (i : nat) :
 i ≤ n → eq_permut (contraction (insert p i)) p.

Lemma insert_contraction_inv : ∀ (n : nat) (q : permut (S n)),
 eq_permut (insert (contraction q) (apply q n)) q.
```

Listing 11: Contraction and insertion.

A natural choice for (contraction_fun $n\ g\ x$) when $x \geq n$ is $(g\ x)$, which leaves g unchanged outside $[0..n-1]$. Contraction is then a right inverse for insertion only if insertion_fun $n\ f\ i\ x = f\ x$ when $i \leq n$ and $x > n$. In that case the present definition of insertion (Listing 9) improves that one in [DGG16], which proposed insertion_fun $n\ f\ i\ x = x$ instead.

All the properties about permutations listed in Sections 3.6 to 3.8 have been tested using the random generator of permutations as lists (genPermlineAsListnat).

4. Second Example: Reversed Subexcedant Lists

A *subexcedant sequence* is a sequence $s_0, \ldots, s_{n-1}$ of natural numbers such that $0 \leq s_j \leq j$ for all $0 \leq j < n$.[1] Subexcedant sequences are studied as an alternative way to represent permutations. Any size-preserving bijection between subexcedant sequences and permutations is called a *permutation code*. Subexcedant sequences and permutation codes are widely studied in combinatorics [Leh60, DV80, MR01, Vaj11, Vaj13, BV17].

Sections 4.1 and 4.2 respectively present an encoding of subexcedant sequences by Coq lists of natural numbers and by terms of a dependent type. In Section 4.3 we manually define random generators for these datatypes. Sections 4.4 and 4.5 present a permutation code and a function whose interest will be made clear in Section 5.

---

[1] This is the 0-based definition, consistent with our other case studies. Other definitions consider a 1-based sequence $s_1, \ldots, s_n$ instead of $s_0, \ldots, s_{n-1}$ or values in $[1..n]$ instead of $[0..n-1]$ [DV80, MR01, Vaj13].

4.1. Characteristic Property

It is easier to extend a subexcedant sequence $s_0, \ldots, s_{n-1}$ by adding a last element s_n at its end, rather than shifting it to add a new first element at its start. However, Coq lists are inductively defined by adding an element at their start. Therefore, we choose to encode the sequence of natural numbers $s_0, \ldots, s_{n-1}$ by the Coq list $[s_{n-1}; \ldots; s_0]$ containing the same elements in the reversed order. When the sequence is subexcedant, the list is said to be a *Reversed Subexcedant List* (RSL, for short). Notice that the last element s_0 is always 0. We however store it in the list for simplicity.

Let (is_rsl n l) formalize the property that l is a reversed subexcedant list of length $|l| = n$. The inference rules

$$\frac{}{\textsf{is\_rsl}\ 0\ [\,]} \qquad \text{and} \qquad \frac{i \leq n, \quad \textsf{is\_rsl}\ n\ l}{\textsf{is\_rsl}\ (n+1)\ (i :: l)}$$

axiomatize this predicate. Since the empty list $[\,]$ contains no element, it is a RSL of length 0, as axiomatized by the first rule. The second rule can be justified by proving by recurrence on n that a nonempty list $(i :: l)$ is a RSL if and only if l is a RSL and $i \leq |l|$. This inference system can be directly translated in Coq as an inductive predicate is_rsl and a Boolean function is_rslb, both reproduced in Listing 12. The listing also shows a lemma stating their equivalence (proved by induction on l).

```
Inductive is_rsl : nat → list nat → Prop :=
| Rsl_nil : is_rsl 0 nil
| Rsl_cons : ∀ i n l, i ≤ n → is_subex n l → is_subex (S n) (i::l).

Fixpoint is_rslb (l : list nat) : bool :=
 match l with
 | nil   ⇒ true
 | i::l' ⇒ andb (leb i (length l')) (is_rslb l')
 end.

Lemma is_rsl_dec: ∀ l, is_rslb l = true ↔ is_rsl (length l) l.
```

Listing 12: Characterization of reversed subexcedant lists.

For validating conjectures about RSLs, we define the following generator of such lists:

```
Fixpoint genRslAsListnat (n : nat) : G (list nat) :=
 match n with
   O    ⇒ returnGen nil
 | S n' ⇒ do! i ← choose (0, n');
            liftGen (cons i) (genRslAsListnat n')
 end.
```

The QuickCheck command

```
Sample (genRslAsListnat 6 0).
```

yields examples of reversed subexcedant lists: $[0, 1, 1, 0, 0, 5]$, $[5, 0, 2, 0, 0, 0]$, $[0, 0, 1, 3, 3, 3]$, $[0, 0, 1, 2, 4, 0]$, etc.

Again to have confidence in this generator, we test it with the Boolean function is_rslb, proved correct w.r.t. the inductive predicate is_rsl. List lengths are also randomly chosen.

```
QuickCheck (sized (fun n ⇒ forAll (genRslAsListnat n) is_rslb)).
+++ Passed 10000 tests (0 discards)
```

4.2. Recursive Dependent Type

It is easy to interpret the characteristic property of reversed subexcedant lists (specified by is_rsl) by the following recursive dependent type:

```
Inductive rslType : nat → Type :=
| R0 : rslType 0
| RS : ∀ n i, i ≤ n → rslType n → rslType (S n).
```

R0 is the unique reversed subexcedant list of length 0. The term (RS i_{n-1} _ (... (RS i_0 _ R0) ...)) with this type, where the _ are proof terms, encodes the subexcedant sequence $i_0, \ldots, i_{n-1}$, as stated by the following property:

```
Lemma rslType_sound : ∀ n (r : rslType n), is_rsl n (rslType2listnat n r).
```

where rslType2listnat converts these RSL terms into lists with the same natural numbers in the same order.

In the rest of the paper, RS is sometimes replaced by @RS in order to bypass the Coq implicit parameters mechanism that sometimes does not help the reading.

4.3. Recursive Random Generator

From this (dependent) type definition, we derive a generator whose particularity is that the generated term contains a proof term.

```
Program Fixpoint genRslType (n : nat) {measure n} : G (rslType n) :=
 match n with
   O    ⇒ returnGen R0
 | S n' ⇒ do2! i, H ← choose (0, n');
          liftGen (RS n' i _) (genRslType n')
 end.
Next Obligation.
apply semChoose in H; auto.
simpl in H.
apply /leP.
apply H.
Defined.
```

Here we use the Program facility to define a recursive function where the proof term is denoted by _ in the body of the definition and then refined using proof tactics (between keywords Next Obligation and Defined). The proof term is mainly the application of a theorem semChoose provided by QuickChick establishing that any number generated by the primitive choose (x, y) is indeed a number i such that $x \leq i \leq y$. In the body of the generator, the notation do2! i, H ← choose (0, n'); ... links i to the number randomly generated by choose and H to the proof that i belongs to the semantics of choose, defined as “the set of values that have non-zero probability of being generated” [PHD+15].

The former soundness lemma (rslType_sound) is checked with this generator, by execution of the command

```
QuickCheck (sized (fun n ⇒ forAll (genRslType n) (fun r ⇒ is_rslb (rslType2listnat r)))).
+++ Passed 10000 tests (0 discards)
```

After this checking the lemma is easily proved by induction on r.

4.4. Permutation Code

In combinatorics a *permutation code* is a size-preserving bijection between subexcedant sequences and permutations. We adapt this definition to our representations of subexcedant sequences and permutations and consider here that a permutation code is a size-preserving bijection between reversed subexcedant lists and permutations encoded as injective natural endofunctions.

More rigorously, we prove formally a one-to-one correspondence between the type rslType of reversed subexcedant lists and the quotient of the type permut of injective natural endofunctions by the equivalence relation eq_permut defined in Listing 11, since this formalization accepts all natural functions with the same values on $[0..n-1]$ to represent the same function on $[0..n-1]$.

The permutation code defined here is the following function:

```
Fixpoint rslType2permut n (s : rslType n) : permut n :=
 match s in (rslType n) with
 | R0 ⇒ id0permut
 | @RS _ i _ r ⇒ insert (rslType2permut r) i
 end.
```

The empty permcode R0 is interpreted as the empty permutation. The term (RS _ i _ r) is interpreted as the permutation obtained by the insertion of i in the permutation resulting from the interpretation of r.

This permutation code corresponds to insertion in the cycle structure of permutations. It adapts to $[0..n-1]$ the function ϕ defined in [MR01, Section 3] for permutations over $[1..n]$. We suggest to call it *transposition code*, because it is related with the factorizations of permutations in transpositions. It is indeed easy to see that the insertion function *insert* defined in Section 3 corresponds to the composition $p\ \langle n, i\rangle$ from left to right of the permutation p with the transposition $\langle n, i\rangle$ exchanging n and i, if one also considers identity $\langle n, n\rangle$ as a transposition. When the subexcedant sequence $i_0, \ldots, i_{n-1}$ is interpreted by this permutation code as a permutation over $[1..n]$, the sequence is called *transposition array* [Bar07, Section 4].

We now prove that the type (`rslType` n) (for a natural number n) is the quotient of the type (`permut` n) by the equivalence relation `eq_permut` introduced in Section 3.6. To achieve this goal, we follow the methodology introduced in [Coh13]. So we define the encoding function `permut2rslType` that transforms a permutation (as a natural function) into a reversed subexcedant list (encoded by a reversed subexcedant term), by using the operation of contraction:

```
Fixpoint permut2rslType n : permut n → rslType n :=
 match n return (permut n → rslType n) with
 | 0    ⇒ fun _ ⇒ R0
 | S n' ⇒ fun p ⇒
   (@RS n' (fct p n') (@permut_bound n' p) (@permut2rslType n' (contraction p)))
end.
```

where (`permut_bound` n' p) provides the proof that the natural function underlying the permutation p applied to n' is less that n' (it comes from the fact that this function is an endofunction on the right interval). The definition of `permut2rslType` (and also the previous one) is a bit technical because it deals with dependent types and this requires some annotations to be type-checked.

The next step is to prove the following *cancellation* lemma (according to the terminology of the Mathematical Components library [Mat18]) which states that the composition of the encoding and decoding functions is the identity:

```
Lemma rslType2permutK : ∀ (n : nat) (t : rslType n), permut2rslType (rslType2permut t) = t.
```

Using the previous random generator of reversed subexcedant terms, we first test the cancellation lemma with QuickChick:

```
QuickCheck (sized (fun n ⇒
  forAll (genRslType n) (fun t ⇒
  eq_rslTypeb n (permut2rslType (rslType2permut t)) t))).
```

Here `eq_rslTypeb` is the Boolean function that checks the syntactic equality of two reversed subexcedant terms of the same type (`rslType` n).

The proof of the cancellation lemma relies on the lemma `contraction_insert_inv` presented in Section 3.6.

4.5. Sum of Two Reversed Subexcedant Lists

As an illustration of RSL as a code for permutations we design and check a binary operation `sum_RslType` of two RSLs whose interpretation is expected to be the direct sum of two permutations defined in Section 3.5. The function

```
Program Fixpoint sum_rslType n1 (l1 : rslType n1) n2 (l2 : rslType n2) : rslType (n1 + n2) :=
 match l2 with
   R0          ⇒ l1
 | @RS n i i_le_n l2' ⇒ @RS _ (n1+i) _ (sum_rslType l1 l2')
 end.
Next Obligation.
omega.
Defined.
```

computes the sum of the RSLs l_1 and l_2, by recursion on l_2. If l_2 is empty the result is clearly l_1. Otherwise, all the elements of l_2 should be augmented by the size n_1 of l_1 and appended before l_1, as specified by the recursive call and the addition of n_1 to the first number i of l_2.

5. Third Example: Rooted Maps

Rooted maps are combinatorial objects with several non-trivial applications in mathematics [MN17] and theoretical physics, in particular in 2-dimensional gravitation models [Eyn11] and in quantum physics [PGHS15]. These applications and the existence of tricky reasoning about rooted maps motivate their formal study.

In a nutshell *labeled maps* are combinatorial structures with $2e$ ($e \geq 0$) distinct labels called *darts*. Darts form e pairs called *edges*. A *rooted map* is mathematically defined as the quotient of labeled maps by relabelings which preserve a pre-distinguished dart called the *root*. The *vertex map* is the (labeled and rooted) map with 0 edges. It is assumed to exist and be unique. 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.

We adopt the following definitions where labels (darts) are the first $2e$ natural integers. A *labeled map* is a pair (R, L) where R is a permutation on $[0..2e-1]$ and L is a fixed-point free involution on $[0..2e-1]$ such that the group $\langle R, L\rangle$ generated by R and L acts transitively on D. Transitivity means that any element of $[0..2e-1]$ can be obtained from any other element of $[0..2e-1]$ by finitely many applications of the permutations R, L and their inverse. The *local involution* with e edges is the fixed-point free involution on $[0..2e-1]$ exchanging $2i$ and $2i+1$ for $0 \leq i \leq e-1$. A *local map* is a labeled map (R, L) whose involution L is a local involution [DGG16, Section 5]. The *root* of a labeled map with e edges is its largest label $2e-1$. A *rooted map* with e edges is an equivalence class of local maps, for the equivalence relation induced by the relabelings preserving the root and the local involution. The virtue of local maps is that they can be represented only by their first permutation R, called their *rotation*. No additional knowledge about maps is required to understand this section, but the interested reader can consult classical monographies on the subject, e.g., [LZ04].

In a former work [DGG16] we formalized local maps with e edges with the Coq type (`map` e) and defined two operations for map construction, whose type is given in Listing 13. (`e1`, `e2` and `e` are numbers of edges.) We admit that these operations can be quotiented as operations on rooted maps, respectively denoted by I and N_k.

```
Definition isthmic e1 (m1 : map e1) e2 (m2 : map e2) : map (e1+e2+1)
Definition non_isthmic e (m : map e) k (pr : k ≤ 2*e) : map (e+1)
```

Listing 13: Local Map Construction Operations.

This section presents a formalization of rooted maps in Coq, made self-contained by admitting that rooted maps can alternatively be defined by Proposition 1. This proposition states that the vertex map, the binary operation I and the parameterized unary operation N_k form a free and complete system of constructors for rooted maps. The origin of this construction goes back to [WL72, Section 3], where Walsh and Lehman decompose rooted maps by erasing their root edge, to justify a recursion relation for the number $F_{b,p}$ of rooted maps with $b+p$ edges and $p+1$ vertices. Although the theorem is not explicit in [WL72], we attribute its paternity to Walsh and Lehman.

Proposition 1 (Construction of rooted maps [WL72]). Let M be a rooted map with $e(M)$ edges. Then exactly one of the following cases holds:

1. M is the vertex map and $e(M) = 0$.
2. $M = I(M_1, M_2)$ for some rooted maps M_1 and M_2 such that $e(M) = 1 + e(M_1) + e(M_2)$.
3. $M = N_k(M_1)$ for some rooted map M_1 such that $e(M) = 1+e(M_1)$ and some natural number $k \leq 2e(M_1)$.

The following recursive dependent type reflects the inductive structure for rooted maps suggested by Proposition 1:

```
Inductive rom : nat → Type :=
| mty : rom 0
| bin : ∀ e1 e2, rom e1 → rom e2 → rom (e1+e2+1)
| unl : ∀ e k, k ≤ 2*e → rom e → rom (e+1).
```

The type name `rom` is an acronym for rooted ordinary maps (ROM, for short) considered in this paper, not to be confused with the larger family of rooted general maps that we plan to study as future work. Terms with type (`rom` e) are called "ROM terms of size e". They can be seen as trees with unary and binary inner nodes, whose unary nodes (`unl` e k _ t) are labeled by some natural number $k \leq n$ where n is the number of inner nodes in their direct subtree t.

By analogy with permutation codes, we say that a *map code* is an interpretation of ROM terms either as local maps, or as permutations representing them, or even as any combinatorial structure encoding these permutations.

ROM terms can be interpreted as local maps by the following representation function, where `map0` is the empty local map. The binary and unary nodes are respectively interpreted by the operations presented in Listing 13:

```
Fixpoint rom2map (e : nat) (t : rom e) : map e :=
 match t with
 | mty ⇒ map0
 | bin e1 e2 t1 t2 ⇒ isthmic (rom2map e1 t1) (rom2map e2 t2)
 | unl e _ p t ⇒ non_isthmic (rom2map e t) p
end.
```

However it is simpler to decode ROM terms as reversed subexcedant lists, and then compose this map code with the permutation code presented in Section 4.4, to obtain the rotation of the corresponding map. This map code is presented in Section 5.3. Before that, Section 5.1 shows how to quickly validate Proposition 1 by counting, and Section 5.2 presents a recursive random generator of ROM terms.

5.1. Counting ROM Terms

The perspective of a formal proof of Proposition 1 is out of the scope of the present paper. Proposition 1 can however be checked by counting, with the Prolog library, as detailed in this section.

The Prolog predicate `romterm` defined in Listing 14 characterizes a family of unary-binary trees corresponding to the type rom and so to the structure of Proposition 1: `romterm`(T,E) holds if and only if T is a ROM term of size E.

```
romterm(mty,0).
romterm(bin(T1,T2),E) :- E > 0, Em1 is E-1, in(E1,0,Em1), E2 is Em1-E1,
 romterm(T1,E1), romterm(T2,E2).
romterm(unl(K,T),E) :- E > 0, Em1 is E-1, romterm(T,Em1), Kmax is 2*Em1,
 in(K,0,Kmax).
```

Listing 14: ROM terms in Prolog.

With the Prolog library introduced in Section 2.3, the query

```
?- iterate(0,7,romterm).
```

outputs in a few seconds the numbers 1, 2, 10, 74, 706, 8162, 110410 and 1708394 of distinct ROM terms of size from 0 to 7. We recognize the first numbers of rooted ordinary maps counted by number of edges [OEIS]. This simple count gives confidence in the existence of a one-to-one correspondence between ROM terms and ordinary rooted maps with the same size.

5.2. Recursive Random Generator of ROM Terms

From the definition of the recursive dependent type rom, using the QuickChick toolbox, we manually derive the recursive random generator shown in Listing 15.

```
Program Fixpoint genRom (n : nat) {measure n} : G (rom n) :=
 match n with
   0    ⇒ returnGen mty
 | S n' ⇒ oneof
           (do2! k, H ← (choose (0, 2*n'));
            liftGen (unl n' k _) (genRom n'))
           [ (do2! e1, H (choose (0, n'));
               liftGen2 (bin2 n' e1 H) (genRom e1) (genRom (n' − e1)));
             (do2! k H ← (choose (0, 2*n'));
               liftGen (unl n' k _) (genRom n'))
           ]
 end.
```

Listing 15: Random Generator of ROM Terms.

Again we are faced with the generation of values of a dependent type requiring proofs inside. So we rely again on the Program mechanism to build the generator. The proof obligations part is omitted here but they are discharged using the semantic theorem about choose as previously. We use a combinator not yet used, oneof, which randomly chooses one of the generators in the given list or the default case (first argument) when the list is empty. So in the case of a nonzero size, the generator randomly chooses between the constructors bin and unl applied to generated data with a smaller size. We can notice that the generator is not structurally defined, so we must verify its termination. It is done here giving a measure which is exactly the size. The last technical remark about this definition concerns bin2 which is more or less a coercion lemma required for typing.

5.3. Map Code

Let us call a *map code* a size-preserving bijection from ROM terms to another representation of maps. Listing 16 presents a map code rom2rslType for local maps (whose rotation is) represented by a reversed subexcedant list. It interprets the constructors bin and unl with the operations isth and noni on RSL, that are nothing but an abstract view of the map construction operations declared in Listing 13 and defined in [DGG16, Section 3.2].

```
Program Definition isth (e1 : nat) (r1 : rslType (2*e1))
   (e2 : nat) (r2 : rslType (2*e2)) : rslType (2*(e1+e2+1)) :=
 let d1 := 2*e1 in
 let d2 := 2*e2 in
 match d1 with
 | 0      ⇒ match d2 with
             | 0     ⇒ @RS 1 1 _ (@RS 0 0 _ r2)
             | S d2' ⇒ @RS (d2+1) (d2+1) _ (@RS d2 d2' _ r2)
   end
 | S d1' ⇒ match d2 with
             | 0   ⇒ @RS (d1+1) d1' _ (@RS d1 d1 _ r1)
             | S _ ⇒ @RS (d1+d2+1) d1' _ (@RS (d1+d2) (d1'+d2) _ (@sum_rslType d1 r1 d2 r2))
             end
 end.

Program Definition noni e (r : rslType (2*e)) k (k_le_2e : k ≤ 2*e) : rslType (2*(e+1)) :=
 let d := 2*e in
 match d with
 | 0    ⇒ @RS 1 0 _ (@RS 0 0 _ r)
 | S d' ⇒ @RS (d+1) k _ (@RS d d' _ r)
 end.

Program Fixpoint rom2rslType (e : nat) (t : rom e) : rslType (2*e) :=
 match t with
 | mty ⇒ R0
 | bin e1 e2 t1 t2 ⇒ isth e1 (rom2rslType e1 t1) e2 (rom2rslType e2 t2)
 | unl e k k_le_2e t ⇒ noni e (rom2rslType e t) k k_le_2e
end.
```

Listing 16: Map Code in Coq.

Again, the Program facility turns the type-checking constraints into proof obligations, which are here only linear equalities on natural numbers. They are all discharged automatically thanks to the default obligation tactic set with the command

```
Local Obligation Tactic := intros; try omega.
```

It would be too technical to explain the definitions of the operations isth and noni. However we can gain confidence into the soundness of the map code by checking the following soundness conjecture, where is_transitive is a logical predicate characterizing transitive rotations [DGG16].

```
Lemma map_code_sound: ∀ e (r : rom e), is_transitive (rslType2permut (rom2rslType e r)).
```

In this lemma the map code is composed with the transposition code rslType2permut defined in Section 4.4 to decode ROM terms as permutations.

We check this conjecture with the following command:

```
QuickCheck (sized (fun e ⇒
 forAll (genRomRotationAsListnat e) (fun r ⇒ is_transitiveb (2*e) r))).
+++ Passed 10000 tests (0 discards)
```

Here is_transitiveb is a Boolean function on lists of natural numbers corresponding to the logical predicate is_transitive, and genRomRotationAsListnat is a random generator of rooted map rotations (in one-line notation). It is obtained by composition of a random generator of ROMs, the map code and a transposition code

Table 1. Execution time (in seconds) for cursors.

Cursor	Size n	Number of data	Why3	OCaml	Coq
n-bounded arrays	5	$5^5 = 3{,}125$	.50	0	1.16
	6	$6^6 = 46{,}656$	6.43	.02	1.20
	7	$7^7 = 823{,}543$	105.90	.31	1.96
	8	$8^8 = 16{,}777{,}216$	2136.63	6.79	17.20
Permutations	8	$8! = 40{,}320$	9.48	.03	1.25
	9	$9! = 362{,}880$	82.92	.29	1.71
	10	$10! = 3{,}628{,}800$	825.29	3.14	6.60

whose code is omitted here. This transposition code is a simple adaptation of rslType2permut to permutations in one-line notation.

Proving the lemma is left as future work.

6. Implementation

Our work is implemented in a tool named CUT, for Coq Unit Testing, freely distributed at `http://members.femto-st.fr/alain-giorgetti/en/coq-unit-testing`. It has been developed with Coq 8.7.1, SWI-Prolog 7.2.3 [SWI18] and Why3 0.88.2. It includes our extension of QuickChick for Coq 8.7 [HLDP18].

This section evaluates the efficiency of the custom generators, of their verification by tests and proofs, and gives some metrics about our development in Coq, Prolog and Why3. All the experiments are executed on a PC Intel Core i5-2400 3.10 GHz × 4 under Linux Ubuntu 16.04 with 15.5 GB of memory.

6.1. Cursors with Why3

We currently distribute two cursors whose soundness is formally proved with Why3: the cursor on bounded arrays presented in Section 2.5 and a cursor on permutations. Some durations (in seconds) of their execution are reported in Table 1, for increasing sizes n. The Why3 column concerns the interpretation by Why3 of the cursor written in WhyML. The OCaml column concerns the OCaml code extracted from this cursor in WhyML. The last column corresponds to an execution of a command SmallCheckWhy3 in Coq. For instance, the generator of permutations of size 10 in Coq is evaluated with the command

```
SmallCheckWhy3 (it_permline 10) (fun x : list nat ⇒ true).
```

The Why3 generator is slower because Why3 code is interpreted, whereas OCaml code is compiled. The overhead in Coq comes from the extraction time, but also from the translation of OCaml arrays into Coq lists.

6.2. Custom Random Generators

With the QuickChick toolbox we have implemented the following random recursive generators:

- genBlistAsListnat, for lists of natural numbers strictly smaller than a given bound b;
- genEndolineAsListnat, for lists of length n with values in $[0..n-1]$, representing endofunctions on $[0..n-1]$ in one-line notation;
- genRslAsListnat and genRslType, for reversed subexcedant lists and terms representing them;
- genRom for ROM terms representing rooted maps.

We evaluate the efficiency of the random generator genX by checking true for random data of size n, with the command

```
QuickCheck (forAll (genX n) (fun t ⇒ true)).
```

for increasing sizes n, 10,000 generated data items, and $b = 10$ for genBlistAsListnat. The results in seconds are reported in Table 2. They show that large data can be generated in a reasonable time.

Table 2. Execution time (in seconds) for random generators (for 10,000 data items).

Data	Generator	Size n			
		10	100	1,000	10,000
10-bounded lists	genBlistAsListnat	3	29	219	4,866
Endolines	genEndolineAsListnat	4	29	264	8,604
Reversed subexcedant lists	genRslAsListnat	4	28	245	7,726
RSL terms	genRslType	3	28	275	8,159
ROM terms	genRom	7	89	911	25,442

6.3. Some Metrics

The development of SmallCheck added about 346 lines of Coq code and is integrated in the QuickChick files. The CUT tool (code shared by all the generators defined in the case study) is composed of 376 lines of Coq code and 44 lines of Prolog code added to the validation library.

The case study is composed of 14 files, 70 definitions, and 161 lemmas and theorems, for a total of 9,206 lines of Coq code. The Prolog code to test the case study is composed of 1,036 lines of code. The Coq code to test the case study is composed of 1,660 lines of code.

With the Prolog library 15 BET suites are generated in less than 4 seconds, for all lists of length 4 or less. They are validated in less than 60 seconds. Among them 10 test suites (in)validate invariance properties.

Random testing is applied to generate bounded lists, reversed subexcedant lists, endofunctions and permutations in one-line notations. For each of these structured data, the tests are generated and executed in around 30 seconds. So, all QuickChick random tests (10,000 test cases for each validation step, except for the wrong conjectures) are generated and executed in less than 2 minutes.

BET with SmallCheck is applied to check 5 generator soundness properties, 7 invariance properties and 2 cancellation lemmas. They are executed in less than 13 seconds.

These are reasonable times for thousands of automatically generated tests. For comparison, the Coq compilation time is around 20 seconds.

7. Related work

Property-Based Testing Several techniques and tools help strengthening the trust in conjectures in proof assistants and assertions about programs manipulating structured data. Randomized property-based testing (RPBT) consists in random generation of test data to validate these properties. RPBT has gained much popularity since the appearance of QuickCheck for Haskell [CH00], followed by, e.g., Quickcheck for Isabelle [Bul12] and QuickChick for Coq [PHD$^+$15], already presented in Section 2. Similar tools are also available in the proof assistants Agda [DHT03], PVS [Owr06] and FoCaLiZe [CDG10]. Bounded exhaustive testing (BET) consists in generating all test data up to some given limit. SmallCheck and Lazy SmallCheck [RNL08] are two Haskell libraries for BET which enumerate data up to some depth. Feat [DJW12] is similar but enumerates test data up to some size, rather than depth.

Counterexample Generation Several tools automate counterexamples generation from a given characteristic constraint (invariant). In type-targeted testing [SVJ15] types are converted into queries to SMT solvers whose answers provide counterexamples. Constraint solving and local choice with backtracking are combined in the generator description language Luck for Haskell [LGH$^+$17]. It is integrated with Haskell, but not with Coq as far as we know. In Isabelle/HOL, Nitpick is based on model-finding [BN10]. Other tools are based on narrowing [Lin07, Cru17]. More references can be found in the introductions of the most recent papers on this subject [LGH$^+$17, Cru17].

Tools for Coq Cruanes and Blanchette have recently started developing a stand-alone counterexample generator for higher-order logic (simple type theory), named Nunchaku, and front-ends for its integration into several proof assistants [CB16]. The frontend for Isabelle/HOL is available (to replace Nitpick). A frontend for Coq is under study. When it will be available, it will be a general-purpose generation tool complementing QuickChick and our Coq unit testing tool composed of custom generators.

Map Formalization The theory of combinatorial maps was developed from the early 1970's. Tutte [Tut73, Tut79] proposed the most advanced work in this direction, developing an axiomatic theory of combinatorial maps without referencing topology. More recently Lazarus [Laz14] 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 maps. An advanced formalization related to maps is that of combinatorial hypermaps to state and prove the four color theorem in the Coq system [Gon08, Gon05]. 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 adopts the alternative definition of a hypermap as a *triple* of endofunctions that compose to the identity [Gon05, 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 [Duf08], the Jordan curve theorem [Duf09], and also some algorithms such as convex hull [BDM12] and image segmentation [Duf07]. 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 [DM07], 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.

8. Conclusion

We have shown how to use random testing and bounded exhaustive testing to validate Coq definitions and conjectures. We propose two approaches to bounded exhaustive testing. The first one, outside Coq, is based on logical specifications and uses Prolog and a validation library to generate test cases in Coq scripts. The second one uses an iterator inside Coq. The iterator can be either written in Coq or be connected to correct-by-construction OCaml generators developed outside Coq. We have applied these methods on a case study including permutations and rooted maps.

This work is also a contribution in combinatorics and in formalization of mathematics with the Coq proof assistant. We have completed a classical formalization of permutations as injective endofunctions with a formalization of subexcedant sequences, widely used in combinatorics to encode permutations. We have proved a size-preserving one-to-one correspondence (a. k. a. *permutation code*) between the two representations. We have formalized three operations on these permutations and sequences, namely insertion, contraction and direct sum. A contribution to combinatorics is a generalization of these three operations to any natural function. We have also defined a recursive dependent type for rooted maps and its correspondence with local maps — a representation of each map by one permutation, proposed in a former work [DGG16]. This formalization of maps differs from all previous ones [DM07, Gon08, Duf08].

Directions for future work are numerous, they concern testing and proving aspects but also with formalization of combinatorial objects.

First, we plan to have a better integration of SmallCheck in QuickChick. Currently the user has to choose one of the two testing methods inside Coq, but they cannot be combined in a single test. An interesting complementary work is to develop and distribute a large library of iterators proved with Why3. It is valuable both in the context of Why3 but also in the context of Coq: via the OCaml extraction mechanisms of Why3 and Coq, these generators can be used through BET inside Coq.

A next step concerns the formal proof of Proposition 1 that would finally and formally justify our type proposal for rooted maps. This could be generalized to other families of combinatorial objects. Permutations and type quotients are already available in the Mathematical Components library [Mat18] using the SSReflect proof language. The rest of our formalization is not yet in this library. One of our objectives is to integrate our project in this library.

Acknowledgements

For this work Alain Giorgetti was supported by the French "Investissements d'Avenir" program, project ISITE-BFC (contract ANR-15-IDEX-03). The authors warmly thank Nicolas Magaud for help with Coq, Valerio Senni for advice about his validation library, Noam Zeilberger and Cyril Cohen for fruitful discussions. We deeply thank Leonidas Lampropoulos for his support on QuickChick. We also thank anonymous referees for their helpful suggestions.

References

[Bar07] J.-L. Baril. Gray code for permutations with a fixed number of cycles. *Discrete Mathematics*, 307(13):1559 – 1571, 2007.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, New York, 2004.

[BC17] J. Bowles and M. B. Caminati. A verified algorithm enumerating event structures. In *Intelligent Computer Mathematics*, volume 10383 of *LNCS (LNAI)*, pages 239–254. Springer, 2017.

[BCCL08] François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In *SMT '08/BPR '08: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 1–5, New York, NY, USA, 2008. ACM.

[BDM12] C. Brun, J.-F. Dufourd, and N. Magaud. Designing and proving correct a convex hull algorithm with hypermaps in Coq. *Comput. Geom.*, 45(8):436–457, 2012.

[BLS18] S. Bereg, A. Levy, and I. H. Sudborough. Constructing permutation arrays from groups. *Designs, Codes and Cryptography*, 86(5):1095–1111, 2018.

[BN04] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.

[BN10] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *ITP 2010*, volume 6172 of *LNCS*, pages 131–146. Springer, Heidelberg, 2010.

[Bul12] L. Bulwahn. The new quickcheck for Isabelle - Random, exhaustive and symbolic testing under one roof. In *CPP 2012*, volume 7679 of *LNCS*, pages 92–108. Springer, Heidelberg, 2012.

[BV17] J.-L. Baril and V. Vajnovszki. A permutation code preserving a double Eulerian bistatistic. *Discrete Applied Mathematics*, 224:9–15, 2017.

[CB16] S. Cruanes and J. C. Blanchette. Extending Nunchaku to dependent type theory. In *Proceedings First International Workshop on Hammers for Type Theories, HaTT@IJCAR 2016, Coimbra, Portugal, July 1, 2016*, volume 210 of *EPTCS*, pages 3–12, 2016.

[CDG10] M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in FOCALTEST. In *Proceedings of the 5th International Conference on Software and Data Technologies - Volume 2: ICSOFT*, pages 82–91. SciTePress, 2010.

[CH00] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, volume 35 of *SIGPLAN Not.*, pages 268–279. ACM, New York, 2000.

[Coh13] C. Cohen. Pragmatic quotient types in coq. In *ITP 2013*, volume 7998 of *LNCS*, pages 213–228. Springer, Berlin, 2013.

[Coq17] The Coq Development Team. The Coq Proof Assistant Reference Manual. `http://coq.inria.fr/`, 2017. Version 8.7.

[Cru17] S. Cruanes. Satisfiability modulo bounded checking. In *Automated Deduction – CADE 26*, volume 10395 of *LNCS*, pages 114–129. Springer, 2017.

[DGG16] C. Dubois, A. Giorgetti, and R. Genestier. In *Tests and Proofs (TAP)*, volume 6792 of *LNCS*, pages 57–75. Springer, 2016.

[DHT03] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In *TPHOLs 2003*, volume 2758 of *LNCS*, pages 188–203. Springer, Heidelberg, 2003.

[DJW12] J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. In *Proceedings of the 2012 Haskell Symposium*, volume 47 of *SIGPLAN Notices*, pages 61–72. ACM, New York, 2012.

[DM07] C. Dubois and J.-M. Mota. Geometric modeling with B: formal specification of generalized maps. *J. Sci. Pract. Comput.*, 1(2):9–24, 2007.

[Duf07] J.-F. Dufourd. Design and formal proof of a new optimal image segmentation program with hypermaps. *Pattern Recogn.*, 40(11):2974–2993, 2007.

[Duf08] J.-F. Dufourd. Polyhedra genus theorem and Euler formula: A hypermap-formalized intuitionistic proof. *Theor. Comput. Sci.*, 403(2-3):133–159, 2008.

[Duf09] J.-F. Dufourd. 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.

[DV80] D. Dumont and G. Viennot. A combinatorial interpretation of the Seidel generation of Genocchi numbers. In J. Srivastava, editor, *Combinatorial Mathematics, Optimal Designs and Their Applications*, volume 6 of *Annals of Discrete Mathematics*, pages 77 – 87. Elsevier, 1980.

[Eyn11] B. Eynard. *Formal Matrix Integrals and Combinatorics of Maps*, pages 415–442. Springer, New York, 2011.

[FP13] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[FP16] J.-C. Filliâtre and M. Pereira. A modular way to reason about iteration. In *8th NASA Formal Methods Symposium*, volume 9690 of *LNCS*, pages 322–336. Springer, 2016.

[GGP15] R. Genestier, A. Giorgetti, and G. Petiot. Sequential generation of structured arrays and its deductive verification. In *Tests and Proofs (TAP)*, volume 9154 of *LNCS*, pages 109–128. Springer, Heidelberg, 2015.

[Gon05] G. Gonthier. A computer checked proof of the Four Colour Theorem, 2005. `http://research.microsoft.com/gonthier/4colproof.pdf`.

[Gon08] G. Gonthier. The four colour theorem: Engineering of a formal proof. In *ASCM 2007*, volume 5081 of *LNCS (LNAI)*, pages 333–333. Springer, Heidelberg, 2008.

[GS12] A. Giorgetti and V. Senni. Specification and Validation of Algorithms Generating Planar Lehman Words. GASCom'12, `https://hal.inria.fr/hal-00753008`, 2012.

[HLDP18] C. Hriţcu, L. Lampropoulos, M. Dénès, and Z. Paraskevopoulou. QuickChick: Randomized property-based testing plugin for Coq, 2018. `https://github.com/QuickChick/QuickChick`.

[Kit11] S. Kitaev. *Patterns in Permutations and Words.* Springer, New York, 2011.

[Laz14] F. Lazarus. 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`.

[Leh60] D. H. Lehmer. Teaching combinatorial tricks to a computer. In *Proc. Sympos. Appl. Math. Combinatorial Analysis*, volume 10, pages 179–193. Amer. Math. Soc., 1960.

[LGH+17] L. Lampropoulos, D. Gallois-Wong, C. Hriţcu, J. Hughes, B. C. Pierce, and L. Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 114–129. ACM, 2017.

[Lin07] F. Lindblad. Property directed generation of first-order test data. In *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4, 2007*, volume 8 of *Trends in Functional Programming*, pages 105–123. Intellect, 2007.

[LPP18] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. Generating good generators for inductive relations. *PACMPL*, 2(POPL):45:1–45:30, 2018.

[LZ04] S. K. Lando and A. K. Zvonkin. *Graphs on Surfaces and Their Applications.* Springer, 2004.

[Mat18] Mathematical Components Team. Mathematical components library, 2018. `http://math-comp.github.io/math-comp/`.

[MN17] A. Mednykh and R. Nedela. Recent progress in enumeration of hypermaps. *Journal of Mathematical Sciences*, 226(5):635–654, Nov 2017.

[MR01] R. Mantaci and F. Rakotondrajao. A permutations representation that knows what "Eulerian" means. *Discrete Mathematics and Theoretical Computer Science*, 4(2):101–108, 2001.

[OEIS] The OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences. `https://oeis.org/A000698`.

[Owr06] S. Owre. Random testing in PVS. Workshop on Automated Formal Methods (AFM), `http://fm.csl.sri.com/AFM06/papers/5-Owre.pdf`, 2006.

[PCRH11] M. H. Palka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*, pages 91–97. ACM, 2011.

[PGHS15] M. Planat, A. Giorgetti, F. Holweck, and M. Saniga. Quantum contextual finite geometries from dessins d'enfants. *Int. J. of Geometric Methods in Modern Physics*, 12:1–17, 2015. World Scientific Publishing.

[PHD+15] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In *ITP 2015*, volume 9236 of *LNCS*, pages 325–343. Springer, Heidelberg, 2015.

[RNL08] C. Runciman, M. Naylor, and F. Lindblad. 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*, pages 37–48, 2008.

[Sen18] V. Senni. Validation library. `https://subversion.assembla.com/svn/validation/`, 2018.

[Sta97] R. P. Stanley. *Enumerative Combinatorics*, volume 1. Cambridge University Press, Cambridge, England, 1997.

[SVJ15] E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In *ESOP 2015*, volume 9032 of *LNCS*, pages 812–836. Springer, Heidelberg, 2015.

[SWI18] SWI. Prolog. `http://www.swi-prolog.org/`, 2018.

[Tar15] P. Tarau. On type-directed generation of lambda terms. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015*, volume 1433 of *CEUR Workshop Proceedings.* CEUR-WS.org, 2015.

[Tut73] W. T. Tutte. What is a map? In *New Directions in the Theory of Graphs: Proceedings*, pages 309–325. Academic Press, New York, 1973.

[Tut79] W. T. Tutte. Combinatorial oriented maps. *Canad. J. Math.*, 31(5):986–1004, 1979.

[Vaj11] V. Vajnovszki. A new Euler-Mahonian constructive bijection. *Discrete Applied Mathematics*, 159(14):1453 – 1459, 2011.

[Vaj13] V. Vajnovszki. Lehmer code transforms and Mahonian statistics on permutations. *Discrete Mathematics*, 313(5):581 – 589, 2013.

[WL72] T. R. S. Walsh and A. B. Lehman. Counting rooted maps by genus I. *J. Comb. Theory, Ser. B*, 13:192–218, 1972.