Verifying Faradžev-Read Type Isomorph-Free Exhaustive Generation

Filip Marić(✉)

Faculty of Mathematics, University of Belgrade, Belgrade, Serbia
filip@matf.bg.ac.rs

Abstract. Many applications require generating catalogues of combinatorial objects, that do not contain isomorphs. Several efficient abstract schemes for this problem exist. One is described independently by I. A. Faradžev and R. C. Read and has since been applied to catalogue many different combinatorial structures. We present an Isabelle/HOL verification of this abstract scheme. To show its practicality, we instantiate it on two concrete problems: enumerating digraphs and enumerating union-closed families of sets. In the second example abstract algorithm specification is refined to an implementation that can quite efficiently enumerate all canonical union-closed families over a six element universe (there is more than 100 million such families).

Keywords: Isomorph-free exhaustive generation · Orderly · Software verification · Isabelle/HOL

1 Introduction

Cataloguing finite combinatorial structures (e.g., subsets, partitions, words, Latin squares, graphs, designs, codes) described by certain specified properties is required in many application domains. It is very desirable that such catalogues are exhaustive and isomorph-free i.e., to contain exactly one representative of each class of isomorphic structures. Often it is not enough to count objects (to enumerate them), but it is needed to generate them explicitly.

Efficient isomorph-free cataloguing algorithms are often divided into three types: (i) Faradžev–Read-type orderly algorithms based on canonical representatives [8,20], (ii) McKay-type algorithms based on canonical orderings [1], and (iii) algorithms based on the homomorphism principle for group actions [10]. These are applied to a wide variety of problems (according to Google Scholar, McKay's paper has more than 500 citations, most of which describe its concrete applications in mathematics, computer science, chemistry, biology etc.).

In this paper we present a formal verification of Faradžev-Read cataloguing scheme within Isabelle/HOL. Verified cataloguing of combinatorial structures is often used in formal proofs (e.g., enumeration of Tame Graphs given by Nipkow et al. [18] was an important part of the Flyspeck proof of Kepler conjecture). We advocate that verifying general isomorph-free catalouging schemes might facilitate verifying enumeration algorithms needed for concrete applications. Author's personal motivation for these algorithms comes from his previous and current work in formalizing combinatorics and finite geometry [15,16].

To demonstrate its usefulness, we applied our general framework on two concrete problems: cataloguing all directed graphs on n nodes (this was the first problem analyzed in the original Read's paper [20]) and cataloguing families of subsets of an n-element domain, closed under unions. A solution for the second problem was described by Brinkmann and Deklerck in 2018 [4] and it combines Faradžev-Read type orderly generation [8,20] and the homomorphism principle [3]. For $n = 6$, their C implementation found around 100 million such families in several seconds, while for $n = 7$, it found around $2 \cdot 10^{15}$ in around 10 to 12 CPU years (on a cluster computer). We refine an abstract algorithm specification to an efficient implementation (still purely functional) and show that it can solve the case $n = 6$ within Isabelle/HOL in a matter of minutes.

In the current paper we focus mainly on presenting definitions (in the most cases proofs are not discussed). It is assumed that the reader is familiar with functional programming and Isabelle/HOL [19]. Some definitions are slightly simplified, to make them more comprehensible. Proof documents are available in the Downloads section at http://argo.matf.bg.ac.rs/ and are going to be submitted to the Archive of Formal Proofs.

Contributions. Our central contribution is the verification of the abstract Faradžev-Read scheme that can be instantiated for many concrete applications. Other contributions are:

- a verified algorithm for cataloguing digraphs and other similar objects [20];
- a verified efficient algorithm for generating union-closed families [4];
- a small verified library for generating basic combinatorial objects (permutations and combinations);
- verified bitwise representation of sets, set operations and families of sets by unsigned integers and some common "bit-hacks".

Related Work. Literature on fast computer-based enumeration of various combinatorial objects is vast, but it seems that there are not many formally verified algorithms and tools. As a part of Flyspeck project, Nipkow et al. used Isabelle/HOL to verify an algorithm for enumerating tame graphs [17,18]. Bowles and Caminati used Isabelle/HOL to verify an algorithm for enumerating event structures and, as a byproduct, all preorders and partial orders [2]. Giorgetti et al. used Why3 and Coq to generate basic combinatorial objects, used in software testing [7,9]. We are not aware of any verified general methods for isomorph-free exhaustive combinatorial enumeration.

2 General Faradžev-Read Scheme

Algorithms for generating combinatorial objects are usually based on recursive schemes that build larger objects by augmenting smaller ones. We shall assume that the set of all objects S is divided into its subsets $S_0, S_1, S_2, \ldots$ grouped by object "size" (e.g., the size can be the number of edges in a graph or the number of sets in a family). Objects in S_{q+1} are produced by augmenting objects in S_q.

A classic, naive algorithm for isomorphism rejection maintains a list L_q of objects of S_q produced so far, and compares the current object with all objects in that list, adding it to the list L_q only if L_q does not contain its isomorph. That assumes that there is an efficient isomorphism test and is doomed to be inefficient when the list becomes long. Efficient schemes (including the Faradžev-Read's) avoid comparing the current object with the previous ones an can deduce whether it should be added to the list only by examining the object itself.

A central component of Faradzev-Read's scheme is a *linear order* (we shall denote it by $<$) in which objects are produced (the scheme is sometimes called *orderly generation*). L_q shall always be sorted wrt. that order. Also, it is assumed that for each isomorphism class there is a single *canonical* object, that for each object we can test if it is canonical and that lists L_q shall contain only canonical objects. We specify this in an Isabelle/HOL locale (use of locales for stepwise implementation is described by Nipkow [17]).

```
locale FaradzevRead' =
fixes S :: "nat ⇒ ('s::linorder) set"
fixes equiv :: "'s ⇒ 's ⇒ bool"
fixes is_canon :: "'s ⇒ bool"
fixes is_canon_test :: "'s ⇒ bool"
fixes augment :: "'s ⇒ 's list"
assumes "⋀ q. equivp_on (S q) equiv"
assumes "⋀ s s' q. ⟦equiv s s'; s ∈ S q⟧ ⟹ s' ∈ S q"
assumes "⋀ s q. s ∈ S q ⟹ ∃! s_c. equiv s s_c ∧ is_canon s_c"
assumes "⋀ s s' q. ⟦is_canon s; s' ∈ set (augment s)⟧ ⟹ s ∈ S q ⟷ s' ∈ S (q + 1)"
assumes "⋀ s s' q. ⟦s ∈ S q; is_canon s; s' ∈ set (augment s)⟧ ⟹
                  is_canon_test s' ⟷ is_canon s"'
```

First two assumptions ensure that *equiv* is an equivalence (isomorphism) relation on every S_q (note that Faradžev-Read scheme does not require to have an executable isomorphism test). The third one ensures that each isomorphism class contains exactly one canonical representative. The fourth one describes the augmentation procedure that builds a list containing possible extensions of a given canonical object (their dimension is always increased by one). Definition of a canonical representative should be as simple as possible, since it is used in proofs. It need not be executable and if it is executable it need not be efficient. We provide another function *is_canon_test* that is used to test for canonicity. It does not need to match the abstract *is_canon* definition in general, but they need to match on the objects obtained by augmenting canonical objects.

Faradžev-Read algorithm iterates through a sorted catalogue L_q of canonical objects of S_q and builds a sorted catalogue L_{q+1} for S_{q+1}. For each object p it iterates trough a list of objects s that augment it. If an object s is non-canonical it is eliminated. If it is canonical, then it is appended at the end of L_{q+1} only if it does not violate the list order. This procedure is specified as follows.

```
order_test s res = (if res = [] ∨ s > hd res then s # res else res)
step L = (let cs = filter is_canon_test (concat (map augment L))
          in rev (fold order_test cs []))
```

Three conditions are sufficient for the correctness of the previous procedure. First, it must be possible to obtain each canonical object s in S_{q+1} by augmenting at least one canonical object in S_q. If that holds, then all canonical objects in S_{q+1} will be enumerated at least once and we need to guarantee that they will survive the order test exactly once. Since the ordering of L_{q+1} is strict, a canonical object cannot survive the order test more than once. The first appearance of a canonical object s will be eliminated by the order test iff the list L_{q+1} constructed so far contains an object s' such that $s' > s$. Element s' could be produced either by the same p that produced s or by some element p' of L_q that precedes p. The former cannot happen if the augmentation procedure always gives elements s in sorted order. Let $f(s) = p$ be the first element of L_q which produces s, and let $f(s') = p'$ be the first element of L_q which produces s'. To forbid that s' is produced by some element p' of L_p that precedes p, we must forbid that both $s' > s$ and $f(s') < f(s)$ hold i.e., we must require that $s' > s$ implies $f(s') \geq f(s)$. To formalize this, we first define the function f (we call it the *minimal parent* function).

```
parent p s ⟷ is_canon p ∧ s ∈ set (augment p)
min_parent p s ⟶ parent p s ∧ (∀ p'. parent p' s ⟶ p ≤ p')
```

Then we extend the locale by requiring the following three conditions.

```
locale FaradzevRead = FaradzevRead' +
assumes "⋀ s q. ⟦s ∈ S (q + 1); is_canon s⟧ ⟹ (∃ p ∈ S q. parent p s)"
assumes "⋀ s q. ⟦s ∈ S q; is_canon s⟧ ⟹ sorted (filter is_canon (augment s))"
assumes "⋀ s s' p p' q.
            ⟦s ∈ S (q + 1); is_canon s; min_parent p s;
             s' ∈ S (q + 1); is_canon s'; min_parent p' s'; s < s'⟧ ⟹ p ≤ p"'
```

Now we can formulate and prove algorithm correctness. We define *catalogue* for S_q as a strictly sorted list of canonical elements of S_q that contains an element for each isomorphism class.

catalogue L q $\longleftrightarrow$ *sorted L* $\wedge$ *distinct L* $\wedge$ *set L* $\subseteq$ *S q* $\wedge$ ($\forall$ *s* $\in$ *set L. is_canon s*) $\wedge$ ($\forall$ *s* $\in$ *S q.* $\exists$ s_c $\in$ *set L. is_canon* s_c $\wedge$ *equiv s* s_c)

It is easily shown that catalogue for every q is unique and that it always exists if Sq is finite (we denote it by *the_catalogue* q).

The central theorem states that the *step* function when applied to a catalogue for S_q produces a catalogue for S_{q+1}.

theorem *"catalogue L q* $\Longrightarrow$ *catalogue (step L) (q + 1)"*

The original proof given by Read is as follows. "Condition 1 ensures that every canonical configuration X in S_{q+1} is produced at least once. Condition 2 ensures that when X is produced for the first time from $f(X)$ there cannot be an entry Y produced from $f(Y) \neq f(X)$ which follows X in L_{q+1} and whose presence will therefore block the addition of X to the list. Condition 3 ensures the same thing when $f(X) = f(Y)$." This informal proof sketch had to be expanded to around 300 lines long Isar proof which employed nested reverse list induction.

Strict Faradžev/Read Conditions. In some concrete instances stronger conditions are met that make possible to skip the order test within *step* function. Although that does not make the implementation much more efficient (the order test is usually quite fast), this can significantly simplify the depth-first variant of the procedure.

locale *FaradzevReadStrict = FaradzevRead' +*
assumes *"⋀ s q. ⟦s ∈ S (q + 1); is_canon s⟧ ⟹ (∃ p ∈ S q. parent p s)"*
assumes *"⋀ s q. ⟦s ∈ S q; is_canon s⟧ ⟹ sorted (augment s) ∧ distinct (augment s)"*
assumes *"⋀ p s p' s' q. ⟦p ∈ S q; parent p s; p' ∈ S q; parent p' s'; p < p'⟧ ⟹ s < s''*

We formally showed that the order test can be skipped.

lemma
assumes *"distinct L" "sorted L" "set L ⊆ S q" "∀ x ∈ set L. is_canon x"*
shows *"step L = filter is_canon_test (concat (map augment L))" "distinct (step L)"*

We have also shown that strict conditions imply the original conditions (by showing that *FaradzevReadStrict* is a sublocale of *FaradzevRead*).

Depth First Variant. When *step* function is iterated, objects are generated in breadth first fashion. A serious concern about such procedure is its memory usage, since at each step it needs to store both the whole list L_q and the elements of L_{q+1} that are generated. We have defined a procedure that makes the catalogue in depth first fashion, and that usually consumes significantly less memory. Note that such procedure could have been defined in the non-strict Faradžev/Read locale, but it would be more complicated, since, to be able to perform the order tests, it would have to store the largest element in L_q for all recursion levels q. In many concrete applications, including both our case studies, strict conditions hold, so we opted only for the simpler variant. We have defined a function *fold_dfs* that "folds" the elements of the Faradžev-Read tree, enumerated in the DFS order, by some given accumulating function. This tree can be formed by augmenting each node in each possible way (using the *augment* function) and retaining only the canonical descendants (filtered by the *is_canon_test* function), but it is not explicitly built in the memory. The *lvl* parameter guarantees termination by controlling the depth of the generated tree. The definition of *fold_dfs* is quite technical.

```
fold_dfs lvl f i ss =
   (if lvl = 0 then i
    else fold (λ s' x. f s' (fold_dfs (lvl - 1) f x (filter is_canon_test (augment s')))) ss i)
```

Elements of the tree are usually folded by the following functions. The function *catalogue_dfs* computes the catalogue by collecting all tree nodes in a list, while the function *count_dfs* only counts nodes, without keeping their list.

catalogue_dfs lvl ss = fold_dfs lvl (λ s x. s # x) [] ss
count_dfs lvl ss = fold_dfs lvl (λ s x. x + 1) 0 ss

If the procedure *catalogue_dfs* starts from a catalogue for S_q, then it traverses over elements of all catalogues from S_q to $S_{q+lvl-1}$ (although in different order than the traversal based on the *step* function). This is formalized by the following theorem (where *mset* denotes the multiset of list elements).

theorem
  assumes *"catalogue L q"*
  shows *"mset (catalogue_dfs lvl L) = mset (concat (map the_catalogue [q..<q+lvl]))"*

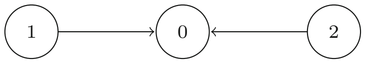
3 Cataloguing Digraphs

The first case-study used to test our general scheme was cataloguing all loopless directed graphs (digraphs) with n nodes. It was the first problem described by Read [20] and we directly follow his approach. As this was just a toy-example, we did not invest much effort into low-level implementation details (e.g., we have used lists which are the simplest data structures)—additional refinement step that would introduce more efficient data structures and some other algorithmic enhancements could make the enumeration much more efficient.

Objects. Following [20], digraphs are represented by their adjacency matrices. As only loopless digraphs with n nodes are considered, the diagonal can be excluded from the matrix and by concating matrix rows an $n \times (n-1)$ vector (a list) representation can be obtained. Graphs will be augmented by adding branches, so we define sets $S_0, S_1, \ldots$ in the following way (the number of nodes n is fixed when interpreting the locale).

*S n q = {l. length l = n * (n - 1) ∧ set l ⊆ {0, 1} ∧ sum_list l = q}*

Example graph, its matrix and list representation are shown on Fig. 1.



$$\begin{pmatrix} 0\,0\,0 \\ 1\,0\,0 \\ 1\,0\,0 \end{pmatrix} \qquad [0,0,1,0,1,0]$$

Fig. 1. A graph represented graphically, by a matrix and by a list

Equivalence. Two graphs are equivalent if there is a permutation of nodes that would map one graph onto another. Permutations are represented by lists of length n (e.g., $[2, 0, 1]$ denotes a permutation that maps 0 to 2, 1 to 0 and 2 to 1). For example, if nodes in the Fig. 1 are ordered 0, 1, 2 instead of 1, 0, 2, then the graph would be represented by the list $[1, 0, 0, 0, 0, 1]$. A direct (but not the most efficient) way to define action of node permutation to a list representing a digraph is to convert it to a matrix, permute the rows and columns of the matrix and then convert the matrix back to a digraph list.

permute_matrix p M = permute_list p (map (permute_list p) M)
permute_dig p n l $\longleftrightarrow$ *mat2dig (permute_matrix p (dig2mat n l))*

Equivalence is often defined by using permutations, so we introduce it abstractly in a separate locale and prove its properties.

locale *Permute =*
fixes *invar :: "nat ⇒ 'a ⇒ bool"*
fixes *permute :: "nat ⇒ nat list ⇒ 'a ⇒ 'a"*
assumes *"⋀ a p n. ⟦invar n a; is_perm n p⟧ ⟹ invar n (permute n p a)"*
assumes *"⋀ a n. invar n a ⟹ permute n (perm_id n) a = a"*
assumes *"⋀ a* p_1 p_2 *n. ⟦invar n a; is_perm n* p_1*; is_perm n* p_2*⟧ ⟹*
    permute n (perm_comp p_1 p_2*) a = permute n* p_1 *(permute n* p_2 *a))"*
assumes *"⋀ a p n. ⟦invar n a; is_perm n p⟧ ⟹*
    permute n (perm_inv p) (permute n p a) = a"

Predicate *is_perm* abbreviates the condition p <~~> $[0..<n]$, where <~~> is the permutation relation from the Isabelle/HOL library. Permutations are applied on objects of some abstract type *'a* (e.g., to lists that represent digraphs) that may satisfy some given invariant (e.g., that the list length is $n(n-1)$). The function *permute* is the action of permutations on the objects of type *'a*. If it respects the permutation group operations (identity *perm_id*, inverse *perm_inv*, and composition *perm_comp*), then we can use it to define equivalent objects and to prove that it is an equivalence relation.

equiv n F_1 F_2 $\longleftrightarrow$ *(∃ p. is_perm n p ∧* F_2 *= permute p* F_1*)*

Ordering. The ordering is very simple – the lexicographic order on lists used to represent graphs is used, except that the order of list elements is reversed (1 is treated as less than 0).

Canonical Objects. Permutations are also used to define canonical objects. An object is canonical if it is minimal among all its possible permutations. For example, the list $[1, 0, 0, 0, 0, 1]$ is canonical for the graph shown in Fig. 1. This definition is also generic and can be specified within the previous locale (a linear order on the type *'a* is assumed). Then it can be proved that each equivalence class contains a single canonical representative (what is needed for Faradzev-Read enumeration).

"is_canon n F ⟷ (∀ p. is_perm n p ⟶ F ≤ permute p F)"
lemma *"inv n a ⟹ ∃! c. equiv n a c ∧ is_canon n c"*

An optimization can be made when checking canonicity of a digraph. By the definition of ordering, the list that starts with as most ones as possible will be always less than lists that have zeros at that initial positions. Therefore only permutations that put a maximal degree node at the beginning and nodes that it is connected to after it need to be considered. This is the essence of our *is_canon_test* definition (that we do not show here).

Augmentation. Graphs are augmented by adding an edge i.e., by changing one 0 in the list to 1. If the list contains some elements 1, then only zeros behind the last 1 can be changed (otherwise any zero can be changed). For example, the list $[1, 0, 0, 1, 0, 0]$ can be augmented to $[1, 0, 0, 1, 1, 0]$ and $[1, 0, 0, 1, 0, 1]$. This can be formalized as a relation between two lists[1].

is_last_one i xs $\longleftrightarrow$ *xs ! i = 1* $\wedge$ ($\forall$ *i'. i < i'* $\wedge$ *i' < length xs* $\longrightarrow$ *xs ! i' = 0*)
all_zeros xs $\longleftrightarrow$ ($\forall$ *i < length xs. xs ! i = 0*)
increment_after_last_one xs ys $\longleftrightarrow$ ($\exists$ *j. j < length xs* $\wedge$ *ys = xs [j := 1]* $\wedge$
    (*all_zeros xs* $\vee$ ($\exists$ *i. is_last_one xs i* $\wedge$ *i < j*)))

All required properties of the augmentation procedure are proved using this abstract definition, and only then its concrete implementation is given (it is quite technical, so we do not show it here). It must return digraphs in sorted order, which is ensured by sequentially incrementing every 0, after the last 1, one by one.

Results. The naive implementation we defined can catalogue all 1 540 944 digraphs with 6 nodes in 276 seconds (on an 2.4GHz, Intel Core i5, 8GB RAM laptop). Interestingly, the original paper reports only 1 540 744 digraphs [20]. Cataloguing more than 800 million digraphs with 7 nodes is possible, but would require significant improvements of the implementation.

4 Cataloguing Union-Closed Families

Families of sets closed under unions have gotten a lot of research attention due to the famous conjecture by Péter Frankl, claiming that in each such family there is an element occurring in at least half of the sets. Although quite elementary, the conjecture is still open [5,16]. Recently Brinkmann and Deklerck applied Faradžev-Read type algorithm to catalogue union-closed and intersection-closed families [4]. We have formalized their procedure in Isabelle/HOL.

4.1 Abstract Procedure Specification

Objects. The most natural way to model sets of natural numbers in Isabelle/HOL is to use the built-in *nat set* type. The type *nat set set* could

[1] By following Read [20], we formalized a slightly more general case where the lists can contain larger numbers than 1 (so at some future point multigraphs can also be considered).

be used for families of sets. However, in order to apply Faradžev-Read enumeration, we need to define a very specific total order of families (based on a specific ordering of sets). We cannot change the default ordering of sets on the type *'a set* nor the ordering of families on the type *'a set set*. Additionally, only finite sets can be ordered, so we must introduce the following two types.

typedef *Set* = *"{ s :: nat set. finite s }"* **morphisms** *elems Set*
typedef *Family* = *"{ s :: Set set. finite s }"* **morphisms** *sets Family*

The union-closed property is defined as follows.

$union\ s1\ s2 = Set\ (elems\ s1 \cup elems\ s2)$
$union\_closed\ F \longleftrightarrow (\forall\ A \in sets\ F.\ \forall\ B \in sets\ F.\ union\ A\ B \in sets\ F)$

We want to enumerate all families closed for union whose largest set is $\{0, 1, \ldots, n-1\}$. Since the empty set does not affect union-closedness, when enumerating union-closed families it is usually excluded from all families. Enumeration starts from the family $\{\{0, 1, \ldots, n-1\}\}$, and extends it by adding sets with less elements. We define the dimension of a family, as the number of its sets without this largest set. Therefore, we define collections $S_0, S_1, S_2, \ldots$ by the following definition.

"$S\ n\ q = \{F.\ (\forall\ s \in sets\ F.\ elems\ s \subseteq \{0..{<}n\}) \wedge card\ (sets\ F) = q{+}1$"
$\quad Set\ \{0..{<}n\} \in sets\ s \wedge Set\ \{\} \notin sets\ F \wedge union\_closed\ F\}$

Equivalence. The function *permute_family* permutes every set in a family by applying the *permute_set* function which permutes a set by applying the given permutation to each member. The function *permute_family* interprets the locale *Permute* (with the invariant that all elements if family sets are less than n) and the definition and properties of equivalence given in that locale are used.

Ordering. The ordering of families is based on an ordering of sets. Sets are ordered first by their cardinality (sets with more elements are declared to precede sets with less elements). Sets of the same cardinality are ordered by lexicographically comparing reverse-sorted lists of their elements. For example, the following is a strictly increasing chain of sets $\{0,1,2\} < \{0,1\} < \{0,2\} < \{1,2\} < \{0\} < \{1\} < \{2\} < \{\}$.

```
less_Set (Set $s_1$) (Set $s_2$) $\longleftrightarrow$
  (let $n_1$ = card $s_1$; $n_2$ = card $s_2$
   in $n_1 > n_2 \vee (n_1 = n_2 \wedge$ rev (sorted_list_of_set $s_1$) > rev (sorted_list_of_set $s_2$)))
```

Families are ordered by lexicographically comparing sorted lists of their sets (wrt. the previous ordering of sets).

$less\_Family\ (Family\ F1)\ (Family\ F2) \longleftrightarrow sorted\_list\_of\_set\ F1 < sorted\_list\_of\_set\ F2$

Canonical Objects. Canonical families are also defined by using permutations, by reusing definitions and statements from the *Permute* locale—two families are equivalent if there is a permutation that transforms one family to the other, and a family is canonical if it is the least one (wrt. the ordering of families) among all its permutations. We have formalized an efficient method for testing if a given family is canonical. The crucial insight is that if a family is obtained by augmenting a canonical family (and that is always the case in the Faradzev-Read scheme), then it is certainly less than all families obtained by permutations that change some of its sets with cardinality greater than minimal. Therefore, it is enough to check only the permutations that fix such sets. For example, when extending the family $\{\{0,1,2\},\{0,1\},\{0,2\},\{0\}\}$, the permutation $0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0$ needs not to be considered since it maps non-minimal cardinality sets $\{0,1,2\},\{0,1\},\{0,2\}$ to $\{0,1,2\},\{0,1\},\{1,2\}$, thus always yielding a greater family. Only permutations that map 0 to 0 need to be analyzed. Note that this is one of the crucial components of the algorithm, since it tremendously reduces the number of permutations that have to be applied to check if a given family is canonical (as the number of sets in families is increased, that number very quickly drops to just a couple of permutations).

```
min_card F = Min (set_card ‘ (sets F))
above_card_sets F c = {s. s ∈ sets F ∧ set_card s > c}
perm_fixes F ⟷ (∀ s ∈ F. permute_set p s ∈ F)
filter_perms ps F = (let F’ = above_card_sets F (min_card F)
                     in filter (λ p. perm_fixes p F’) ps)
is_canon_test n F ⟷ (∀ p ∈ set (filter_perms (permute [0..<n]) F).
                       F ≤ permute_family p F)
```

Augmentation. A family is augmented by adding sets that are larger than its largest set (wrt. the ordering of sets).

```
augment_set n s = {s’. elems s’ ≠ {} ∧ elems s’ ⊆ {0..<n} ∧ s’ > s}

augment n F = (let Fs = {add_set F s | s. s ∈ augment_set n (Max (sets F))}
               in sorted_list_of_set {F’ ∈ Fs. union_closed F’})
```

Testing if a family is union-closed requires analyzing all pairs of sets. However, since families are generated by adding sets to smaller union-closed families, we only need to find unions of the new set s with the sets present in the family F that is being augmented. Since s is larger than all sets in F, the procedure can be optimized. It suffices to check only those sets of F that do not contain subsets in F (those sets form the *reduction* of F).

```
reduction F = {s ∈ sets F. ¬ (∃ s’ ∈ sets F. elems s’ ⊂ elems s)}
lemma
  assumes “union_closed F” “∀ s’ ∈ sets F. s > s”’
  shows “union_closed (add_set F s) ⟷ (∀ s’ ∈ reduction F. union s’ s ∈ sets F)”
```

Note that many previous definitions are not efficient or even not executable (e.g., in the augmentation procedure it is not specified how to construct sets larger than the given one, and the required sorted order of the resulting list of families is ensured by explicitly sorting the list, which would be inefficient in a real implementation). However, abstract specification like this one are very convenient for proving algorithm correctness, while efficient executable implementation can be defined later.

4.2 Implementation

The abstract procedure specification already contains two very important optimizations: filtering permutations when checking canonicity and filtering sets when checking union-closedness. However, there are many additional optimizations that should be done in order to get an executable, efficient implementation of the procedure:

- sets and families must be represented using efficient data structures;
- objects should be generated in-order, and a-posteriori sorting must be avoided;
- computations that are redundantly repeated many times should be avoided by applying memorization and storing results in lookup tables.

Unlike abstract specification that is stateless, an efficient implementation must be stateful. There are many methods to handle state in functional programs, and we use the simplest one: it is explicitly passed trough function calls.

Objects. Using bitwise representation is the best choice for representing sets and families. A set can be represented by an unsigned integer that has the bit i equal to 1 iff the set contains the element i. For example, if 8-bit words are used, the set $\{0, 2, 5\}$ can be represented by `00100101`, i.e., by 37. Similarly, a family can be represented by unsigned integer that has the bit i equal to 1 iff the family contains the set represented by i. Since there are 2^{2^n} families over $\{0, \ldots, n-1\}$, 64-bit words can be used to represent families over $\{0, \ldots, 5\}$.

However, since we wanted to make a very clear separation between the high-level algorithm correctness and low-level bit-twiddling hacks, we have introduced another layer of abstraction. We have introduced another locale, parametrised by the type *'s* for representing sets and *'f* for representing families, and by some primitive operations over these types.

For example, a type *'s* that represents sets must support a constant for the empty set, must support reading the list of set elements, checking if the set contains an element, adding an element to a set, finding union of two sets, determining the cardinality of a set, finding the list of all possible subsets of $\{0, \ldots, n-1\}$, etc. It must be linearly ordered and that order must be compliant with the lexicographic order of reversed lists of set elements. Since only elements up to a certain size must be represented, all assumptions in our locale are guarded by the condition $n \leq n_{max}$, where n_{max} is a locale parameter. Based on such primitives, algorithm-specific set operations are defined (e.g., ordering of sets is defined based on *card* and <, and permuting sets is defined by traversing the list of elements and inserting their permuted images into a resulting set).

A type *'f* must support a constant for the empty family, must support reading the list of family sets, adding and removing set from the given family etc. Again, value n_{max} assures that all families can be properly represented.

Caching Information About Families. Each family F must contain information about all sets that it contains (and this is represented by a value of type *'f*). However, in order to avoid repeating computations, we shall associate some additional data with each family. For augmentation of F we need to know the maximal set and the reduction of a family (so that we can efficiently check union-closure). For testing canonicity we need to know a list of permutations that fix sets in F with cardinality above minimal. We store all these in a record (permutations are represented by numbers from 0 to $n! - 1$).

```
datatype ('f, 's) FamilyRecord =
    FamilyRecord (all_sets : 'f) (max_set : 's) (reduction : 'f) (perms : "nat list")
```

Ordering. Ordering families is a bit tricky in the general case. If bitwise representation is used, the order of family codes need not necessarily comply with our abstract ordering of families (which takes into account set cardinality). However, within the enumeration we only compare families with their permuted variants for permutations that fix all sets except those with the minimal cardinality. Therefore, it suffices just to extract sets with minimal cardinality and compare two families based only on those sets. When bitwise representation is used, since all other bits will be the same, it suffices just to compare family codes.

Canonical Objects. Due to a relatively low number of subsets of $\{0, \ldots, n-1\}$ (for $n = 6$, there are only 64 such sets) and a relatively low number of permutations of $[0, \ldots, n-1]$ (for $n = 6$, there are only 720 such permutations), the action of all permutations on all sets can be precomputed and stored into a lookup table (we use a RBT Mapping available from the Isabelle/HOL library). The function that initializes the lookup table can easily be defined and it need not be very efficient (it is called only once at the very beginning of the procedure).

```
type_synonym 's SetPerms = "((nat × 's), 's) mapping"

init_set_perms n =
    (let ps = permute [0..<n]; ss = powerset n;
         keys = concat (map (λ p. map (λ s. (p, s)) ss) [0..<length ps])
     in Mapping.tabulate keys (λ (p, s). permute_set n (ps ! p) s))
```

In the previous code, the function *permute* is defined within our small library for generating basic combinatorial objects and it generates all permutations of the given list. The function that permutes a given family then just looks up permuted sets from the *set_perms* mapping.

```
permute_family F p =
   foldl (λ F' s. add F' (the (Mapping.lookup set_perms (p, s)))) empty_family (sets F)
```

Now the canonicity check can easily be implemented. The list of relevant permutations is stored within the family record. The first permutation in that list is always the identity permutation and it does not need to be checked (so in many cases no family permutations at all need to be made).

```
is_canon_test set_perms F =
  list_all (λ p. less_eq_family (all_sets F) (permute_family set_perms p (all_sets F)))
           (tl (perms F))
```

Augmentation. Implementing augmentation has several important parts. First, we need to know how to enumerate all augmenting sets for a given set, then we need to check if adding an augmenting set to a family would leave it union-closed and finally, when the set is added we need to update the list of permutations that need to be tested when checking if the family is canonical, to update the reduction of the family and to update its maximal set.

The function that finds all possible augmentations for a given set might be implemented in the following way (again, it does not need to be much efficient, since it is also called only once).

```
augment_set n s = filter (λ s'. s < s') (set_of (combine [0..<n] (card s))) @
                      concat (map (λ c'. set_of (combine [0..<n] c')) (rev [1..<card s]))
```

The function *combine* is also defined within our small library for generating basic combinatorial objects and *combine l k* computes all k-element sublists of the given list l. Augmenting sets of a set s first contain sets with the same cardinality as s that are larger than it, and then, all sets of each cardinality less than the cardinality of s, in decreasing order (this gives a sorted list of all augmenting sets wrt. our set order).

Since the same sets are augmented over and over again (as they occur in different families), results of *augment_set* for each s in *powerset* n are stored in a lookup table and that lookup table becomes a parameter of the family augmentation procedure *augment*.

Each augmenting set is analyzed and it is checked if adding it to the family leaves it union-closed. This is done by examining only the sets from the reduced family (which are stored within the family record).

```
is_union_closed F s = list_all (λ s'. contains_set F (union s' s)) (sets (reduction F))
```

If adding the set s to the family F leaves it union closed, then a new family record is created. The set is added to the collection of all sets using the primitive operation and it is set as the maximal set of the extended family (since the augmenting sets are always larger than all sets in the family). The reduction of the extended family is obtained by analyzing all sets in the reduction of the original family F, removing those that contain s (by means of the primitive operation), and by adding s to the reduction (as the maximal set it has the minimal cardinality and the family cannot contain its subset).

```
update_reduction Rs s =
    (let Rs' = foldl (λ Rs s'. if is_subset s s' then remove Rs s' else Rs) Rs (sets Rs)
      in add Rs' s)
add_set F s = FamilyRecord (add (all_sets F) s)
                                   (update_reduction (reduction F) s) s (perms F)
```

Finally, the augmented set is added to the family and if its cardinality is strictly less than cardinality of other sets in the family, the set of permutations is filtered (permutations that do not fix sets above minimal cardinality are removed).

```
perm_fixes set_perms F p ⟷
    list_all (λ s. contains_set F (the (Mapping.lookup set_perms (p, s)))) (sets F)
filter_perms n set_perms F c =
    (let perms' = filter (perm_fixes set_perms (sets_of_card F c)) (perms F)
      in FamilyRecord (all_sets F) (reduction F) (last_set F) perms')
extend_family n set_perms F s =
    (let F' = add_set F s; c = card (max_set F); c' = card s
      in if c ≠ c' then filter_perms n set_perms c F' else F')
```

With these functions available, we define the augmenting procedure.

```
augment n augmenting_sets powerset_by_card set_perms F =
    map (extend_family n powerset_by_card set_perms F)
        (filter (λ s. is_union_closed F s)
                (the (Mapping.lookup augmenting_sets (max_set F))))
```

Correctness Proof. The correctness proof reduces to showing that this stateful implementation corresponds to the abstract specification. Functions *abs_set* and *abs_family* that convert *'s* to *Set* and *'f* to *Family* are easily defined and it is easily shown (by using the locale assumptions) that primitive operations given in a locale are in accordance with operations on sets (the real burden of showing this is when interpreting the locale by bitwise representation). Then, a set of lemmas is proved that connects each implemented function with its abstract counterpart. For example, the lemma that establishes the connection between the abstract test for canonicity and its implementation is the following.

```
lemma
    assumes "n ≤ n_max" "inv_f n (all_sets F)"
            "set_perms_OK set_perms n" "perms_filtered F n" "hd_perms F"
    shows "FamilyImpl.is_canon_test n set_perms F ⟷
           FamilyAbs.is_canon_test n (abs_fam (all_sets F))"
```

The assumptions require that all sets in the family record satisfy all required representation invariants (for example, this guarantees that all sets in F are subsets of $\{0, \ldots, n-1\}$), that the lookup table *set_perms* contains permutations of all sets, that the family record contains exactly those permutations that fix all sets of F with more elements than the minimal set of F, and that the first element in the list of those permutations is the identity permutation. In many cases such lemmas are proved almost immediately (by using similar lemmas about functions called in the current function definition). However, in some cases there is more work that should be done (e.g., we need to show that our *augment_set* implementation builds a sorted and distinct list of sets that covers every set that is larger than the one being augmented).

It is also necessary to show that functions preserve invariants. All lookup tables are initialized before the enumeration starts and we prove that functions that initialize lookup tables do that correctly. For example, we show that *init_set_perms* builds a lookup table that for each set s in *powerset* n and each permutation index p from 0 to $n! - 1$ returns the set obtained by permuting s by the p-th permutation in the lexicographic ordering of permutations of $[0, \ldots, n-1]$. Other invariants characterize data in the family record. For example, one such invariant claims that *max_set F* is always a set in F that is the largest among all sets of F. Since the family record is updated only in the *augment* function, the major challenge is to show that it preservers all such invariants.

4.3 Bitwise Set Representation

Finally, we used the bitwise representation to represent sets and families, based on the Native word library [14]. Sets are represented by the type `uint8`, while families are represented by the type `uint64`. Primitive operations on sets are implemented using the bitwise operations. For example, adding element and removing element from a set, union and intersection of sets is defined by

```
add x e = x OR (1 << k)        remove x e = x AND NOT (1 << k)
union x1 x2 = x1 OR x2         inter x1 x2 = x1 AND x2
```

We have also implemented an efficient function for finding the cardinality of a set, by using the parallel bit-count algorithm.

```
card s0 = (let s1 = (s0 AND 0x55) + ((s0 >> 1) AND 0x55);
               s2 = (s1 AND 0x33) + ((s1 >> 2) AND 0x33);
               s3 = (s2 AND 0x0F) + ((s2 >> 4) AND 0x0F)
           in nat_of_uint8 s3)
```

However, since we calculate cardinality only for 8-bit numbers, it turns out that there is no much benefit to using a naive, sequential bit-testing algorithm.

Similarly, a list of sets in a family could be determined by a naive, sequential test of each of 64 bits. For families that do not contain many sets, it is more efficient to iterate only trough the bits that are set. Many hardware architectures offer *count trailing zeros* (`ctz`) instruction that is used to find the last set bit. Clearing last set bit can be achieved by calculating `x & (x-1)`. Unfortunately, it seems that `ctz` instruction is not available from functional languages. It can be implemented by a binary search approach, yielding a six-step algorithm for 64-bit words, but our experiments reveal that using such implementation is less efficient than the naive algorithm.

4.4 Results

Our verified implementation exported to Haskell catalogues all 108 281 182 union-closed families in around 11 min (on an 2.4GHz, Intel Core i5, 8GB RAM laptop). Our fastest, unverified implementation in C++ that uses the same algorithm, but is based on arrays, does it in around 28 seconds. Profiling shows that the verified implementation spends more than 60% of the time in RBT lookup. Replacing $O(\log n)$ RBT with $O(1)$ lookup array reduces the time to less than 5 min (for this we can use the Isabelle Collections Framework [11] or Imperative/HOL [6]). Unfortunately, a range-check is performed with each verified bitwise operation, and there is no direct access from Isabelle/HOL to all hardware implemented bitwise operations (e.g., `__builtin_ctzl` in GCC), so its hard to expect that C++ runtimes could be reached with standard Isabelle code generator. When families are only counted using the depth-first variant of the algorithm, memory consumption is not an issue.

5 Conclusions and Further Work

We have formalized the general Faradžev-Read scheme for making exhaustive, isomorph-free catalogues of combinatorial objects within Isabelle/HOL and have shown its applicability by instantiating in on two different problems: cataloguing directed graphs and cataloguing union-closed families. In the second case study we have created an efficient implementation capable of generating more than one hundred million union-closed families over a six-element domain.

Our experience shows that even with the general scheme verified, there is still much work to do for each concrete application, especially if efficient implementation is required (our rough estimate is that verifying the general scheme is around 30–50% of the effort needed to verify a concrete efficient algorithm). Still, a verified general scheme does save a significant amount of work in each concrete instance, and, more importantly, guides us towards elements that should be defined in order to get an efficient algorithm.

Specification and the correctness proof of the abstract Faradžev-Read scheme contains 3 locales with around 10 assumptions, 10 definitions and 40 lemmas, consuming around 1200 lines of code (LOC). The case study of digraphs contains around 25 definitions and 100 lemmas, consuming around 4000 LOC. The case study of union-closed families contains around 105 definitions and 350 lemmas, consuming around 8000 LOC (5000 LOC are devoted to efficient implementation). Some definitions and lemmas are shared between both case studies.

We use refinement based on Isabelle/HOL locales to separate reasoning about abstract procedure properties and concrete implementation details. Using a framework (e.g., Isabelle refinement framework [12]) might give us better proof automation and easier introduction of imperative features [13], so we plan to use it in our future work.

There are other general cataloguing schemes, some more efficient than Faradžev-Read's. Most notable of them is McKay's canonical path generation [1]. In our further work we plan to formalize it, too. We hope that some parts of the developed theory could be reused (e.g., the definition of isomorphism based on permutations and their action and the definition of the catalogue). On the other hand, Faradzev/Read and McKay's approach are substantially different so we are not too optimistic that any parts of Faradzev/Read algorithm specification would be useful for computing canonical labellings. A prerequisite for McKay's algorithm trusted implementation is an efficient, trusted graph isomorphism testing algorithm which we plan to construct (either by its verification within a theorem prover, or by some kind of certificate checking).

References

1. McKay, B.D.: Isomorph-free exhaustive generation. J. Algorithms **26**(2), 306–324 (1998)
2. Bowles, J., Caminati, M.B.: A verified algorithm enumerating event structures. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) CICM 2017. LNCS (LNAI), vol. 10383, pp. 239–254. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62075-6_17
3. Brinkmann, G.: Isomorphism rejection in structure generation programs. In: Discrete Mathematical Chemistry (1998)
4. Brinkmann, G., Deklerck, R.: Generation of union-closed sets and Moore families. J. Integer Sequences **21**(1), 9–18 (2018)
5. Bruhn, H., Schaudt, O.: The journey of the union-closed sets conjecture. Graphs Comb. **31**(6), 2043–2074 (2015)
6. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_14
7. Erard, C., Giorgetti, A.: Bounded exhaustive testing with certified and optimized data enumeration programs. In: Gaston, C., Kosmatov, N., Le Gall, P. (eds.) ICTSS 2019. LNCS, vol. 11812, pp. 159–175. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31280-0_10
8. Faradzev, I.A.: Constructive enumeration of combinatorial objects. Colloques Int. CNRS **260**, 131–135 (1978)
9. Giorgetti, A., Dubois, C., Lazarini, R.: Combinatoire formelle avec why3 et coq. In: Journées Francophones des Langages Applicatifs (JFLA 2019), pp. 139–154, Les Rousses, France (2019)
10. Kerber, A., Laue, R.: Group actions, double cosets, and homomorphisms: unifying concepts for the constructive theory of discrete structures. Acta Applicandae Mathematicae **52**, 63–90 (1998). https://doi.org/10.1023/A:1005998722658
11. Lammich, P.: Collections framework. Archive of Formal Proofs. Formal proof development, November 2009. http://isa-afp.org/entries/Collections.html
12. Lammich, P.: Refinement for monadic programs. Archive of Formal Proofs. Formal proof development, January2012. http://isa-afp.org/entries/Refine_Monadic.html
13. Lammich, P.: Refinement to imperative HOL. J. Autom. Reason. **62**(4), 481–503 (2019). https://doi.org/10.1007/s10817-017-9437-1
14. Lochbihler, A.: Native word. Archive of Formal Proofs. Formal proof development, September 2013. http://isa-afp.org/entries/Native_Word.html
15. Maric, F.: Fast formal proof of the Erdős-Szekeres conjecture for convex polygons with at most 6 points. J. Autom. Reasoning **62**, 301–329 (2017)

16. Marić, F., Živković, M., Vučković, B.: Formalizing Frankl's conjecture: FC-families. In: Jeuring, J., et al. (eds.) CICM 2012. LNCS (LNAI), vol. 7362, pp. 248–263. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31374-5_17
17. Nipkow, T.: Verified efficient enumeration of plane graphs modulo isomorphism. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 281–296. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22863-6_21
18. Nipkow, T., Bauer, G., Schultz, P.: Flyspeck I: tame graphs. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 21–35. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_4
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
20. Read, R.C.: Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. In: Alspach, B., Hell, P., Miller, D. (eds.) Algorithmic Aspects of Combinatorics, Annals of Discrete Mathematics, vol. 2, pp. 107–120. Elsevier (1978)