DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Verification of Combinatorial Algorithms

Paul Hofmeier

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Verification of Combinatorial Algorithms

Verifikation von kombinatorischen Algorithmen

Author:	Paul Hofmeier
Supervisor:	Prof. Tobias Nipkow
Advisor:	Emin Karayel
Submission Date:	15.09.2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2022 Paul Hofmeier

Acknowledgments

First of all, I would like to thank Prof. Tobias Nipkow for allowing me to write this bachelor thesis.

Moreover, I would like to thank my advisor Emin Karayel, who was always available for questions, and provided his time for frequent meetings, which helped me a lot to not lose track and get feedback. In addition, even though it was his first time as an advisor, he gave me valuable tips and supported me in the best way I could think of, which I am very grateful for.

Abstract

Combinatorial objects have configurations which can be enumerated by algorithms, but especially for imperative programs, it is difficult to find out if they produce the correct output and don't generate duplicates. Therefore, for some of the most common combinatorial objects, namely *n_Sequences*, *n_Permutations*, *n_Subsets*, *Powerset*, *Integer_Compositions*, *Integer_Partitions*, *Weak_Integer_Compositions* and *Trees*, this thesis formalizes efficient functional programs and verifies their correctness with help of the interactive theorem prover Isabelle. In addition, for some combinatorial structures the enumeration function is then used to show the structure's cardinality.

Contents

1 Introduction

In combinatorics, which is "[...] the field of mathematics concerned with problems of selection, arrangement, and operation within a finite or discrete system" [1], structures with certain properties occur. Those combinatorial objects can be enumerated by algorithms, but when implementing them it is sometimes difficult to tell if they produce the correct output and don't generate duplicates. Therefore, to be sure they work correctly, the programs need to be formally verified.

In this thesis, I will do this with the interactive theorem prover Isabelle. Since Isabelle provides a good infrastructure for ML programs, and imperative programs are a lot harder to argue about, the programs I will formalize and verify are functional. In addition, for the most part the functional programs will be efficient in the sense that they do not remove objects after they have already added them, but this of course means that proofs become more difficult, since for example, a remove duplicates function, which would enforce distinctness, or a filtration on all sequences to check whether it is a permutation, will not be applied in the enumeration function.

Aside from showing the correctness and distinctness of the combinatorial enumeration function's output, I will also use them to prove the cardinality for combinatorial objects of which the cardinality has not been shown yet.

Obviously, this is not the only use case for these programs, since different combinatorial structures occur very often in computer science. When for example looking at data structures, combinatorial enumeration functions can be used to cover all test cases.

While combinatorial algorithms cannot only be functions for enumeration, but also for example, functions to search for a combinatorial object which satisfies a certain condition [10], I will address in this thesis only combinatorial enumeration algorithms.

2 Selection of Combinatorial Objects

What we mean by combinatorial objects is very comprehensive. In this thesis we will define a combinatorial object as a structure which can only have a finite number of configurations, where a configuration is a state of an object which is different to all other states and which satisfies a certain condition. The set of structures with all possible configurations, is then what we call the combinatorial object's set and this is what we want to enumerate.

All kinds of combinatorial objects exit, they appear in different mathematical fields, ranging from number theory over order theory up to graph theory. For instance, in number theory the set of all natural numbers smaller or equal to 8, in order theory all equivalence relations, or in graph theory all acyclic graphs. Due to the large number of possible combinatorial structures, we need to categorize them and then select some of the most common and important ones to stay in the scope of a bachelor thesis.

First, it makes sense to distinguish between combinatorial objects with different amounts of distinct configurations, because of inequality of their set cardinality. If the cardinality would be equal, we should in general be able to find bijections between those sets, which then may be able to be applied in linear time to get an algorithm for the other object. Moreover, for some combinatorial objects it is possible to combine different enumeration algorithm to get a correct result, but it may still be necessary to use separate functions if a faster algorithm can be implemented.

2.1 The Twelvefold Way

The Twelvefold Way is another scheme we can look at, that has even been formalized in Isabelle, though the existing formalization only contains results about the cardinalities and no enumeration algorithms [5]. This scheme talks about the amount of functions $f : N \to X$ where N and X are finite sets. It is split into the cases of f being injective, surjective or any function, and in the cases of whether all elements of N or all elements of X are distinguishable [11]. Those restrictions then conclude the cardinalities in Table 2.1.

Elements of N	Elements of X	Any f	Injective f	Surjective f
distinct	distinct	x^n	$(x)_n$	$x!\left\{{n \atop x}\right\}$
indistinct	distinct	$\left(\binom{x}{n}\right)$	$\binom{x}{n}$	$\left(\binom{x}{n-x}\right)$
distinct	indistinct	$\sum_{i=0}^{x} \left\{{n \atop i}\right\}$	$[n \leq x]$	$\left\{{n \atop x}\right\}$
indistinct	indistinct	$\sum_{i=0}^{x} p_i(n)$	$[n \leq x]$	$p_x(n)$

Table 2.1: The Twelvefold Way, adapted from [11].

To clarify some notations of Table 2.1:

- n is the cardinality of N and x is the cardinality of X.
- $(x)_n$ is a falling factorial and equal to $n! \cdot \binom{x}{n}$.
- $\left\{{n \atop x}\right\}$ is the stirling number second kind.
- $\left(\binom{x}{n}\right)$ is the multiset coefficient and equal to $\binom{x+n-1}{n}$.
- $p_x(n)$ is the partition function which counts how many partition of n into x non zero parts are possible.
- $[n \leq x]$ is an Iverson bracket which is 1 if the condition holds and 0 if not.

However, these notations are only used here for compactness reasons and from now on we will derive notations from our formalization in Isabelle, which will be introduced in Chapter 1.

The Twelvefold Way, can also be interpreted in a more discrete and intuitive way, by solving the problem of putting balls into urns [8]. From Table 2.2 we can then select combinatorial object.

balls per urn	unrestricted	≤ 1	≥ 1
n labeled balls, m labeled urns	n-sequences of m things	n-permutations of m things	partitions of $\{1, ..., n\}$ into m ordered parts
n unlabeled balls, m labeled urns	n-multisubsets of m things	n-subsets of m things	compositions of n into m parts
n labeled balls, m unlabeled urns	partitions of $\{1, ..., n\}$ into $\leq m$ parts	n pigeons into m holes	partitions of $\{1, ..., n\}$ into m parts
n unlabeled balls, m unlabeled urns	partitions of n into $\leq m$ parts	n pigeons into m holes	partitions of n into m parts

Table 2.2: The Twelvefold Way with balls and urns, adapted from [8].

2.2 Other Combinatorial Objects

Some of the most common combinatorial objects are included in the Twelvefold Way, but of course it is far away from being complete. Moreover, sometimes structures may not appear to be, but are, directly related to this classification, like the stars and bars problems [11], which are similar to n-multisubsets and compositions of n.

Nevertheless, there are also more complex structures with cardinalities different from the twelvefold way. To name a few:

Dyck words and binary trees with labeled nodes have a cardinality equal to the catalan number [8].

Lattice paths with different constraints can create specific combinatorial objects [11].

Or simply anything from graph theory, including order relations , but some of course fit into the Twelvefold Way. For example equivalence relations, are similar to n_partitions.

2.3 Selection

To make a selection of combinatorial objects, we want to verify an algorithm for, it is worth taking a look at what algorithms have already been formalized in Isabelle. The standard library contains algorithms for permutations, multset permutations and lists, additionally an algorithm for equivalence relations can be found in the AFP [7]. Though I have to admit here, that there may be more which I haven't found.

Considering enumeration algorithms that have already been verified in Isabelle, and the brief categorization from before, enumeration algorithms for

n_Sequences, *n_Permutations*, *n_Subsets*, *Powerset*, *Integer_Compositions*, *Integer_Partitions*, *Weak_Integer_Compositions* and *Trees*

were verified for this thesis.

3 Infrastructure

Since we don't want to start from scratch, we need an existing foundation to rely on in the formalization. This will also reduce the amount of work that needs to be done, and provide compatibility for other Isabelle projects. Additionally, new infrastructure for generalizing our problems and thus simplifying some proof processes, is desirable.

3.1 Foundation

3.1.1 Isabelle/HOL

As already mentioned, the verification will be done in Isabelle. It is a generic proof assistant that can especially be used for mathematical proofs and formal verification [13]. In particular, we will use Isabelle/HOL, where HOL stands for Higher Order Logic, to verify algorithms. For this, Isabelle provides us with a standard ML like syntax for functional programs, proving assistance with for example computational induction, and many libraries from which we will especially use *List*, *Set*, *Multiset*, *Tree*, *Binomial* and *Permutations*. Some noteworthy functions/definitions we will use are *distinct*, $\subseteq$ and *set*. Of course there is a lot more to tell about Isabelle, but to keep it concise, refer to the documentation [13] for anything unclear or not mentioned here.

Code generation is another Isabelle feature which is useful in the context of this thesis. With this it is possible to convert executable Isabelle code directly to SML, OCaml, Haskell and Scala programs [13]. These programs are then formally verified, and the risk of making mistakes when manually adopting from Isabelle code, does no longer exist. In our case this would for instance mean, that we could with little effort get haskell programs for enumerating combinatorial objects.

3.1.2 Archive of Formal Proofs

While the Isabelle standard libraries contain common formalization, the Archive of Formal Proofs, or in short AFP, is another reliable place to get library files. To expand our foundation following AFP entries were used.

Catalan Numbers contains a formalization of catalan numbers, including a corresponding induction scheme [6].

Cardinality of Number Partitions as the name suggests, it contains proofs for the cardinality of number partitions [2].

The Falling Factorial of a Sum contains theorems about falling factorials [4].

Needless to say, the AFP has also theories which we don't directly use as infrastructure, but are still closely related to combinatorial objects. For instance, Derangements, Enumeration of Equivalence Relations and The Twelvefold Way [3, 7, 5].

3.2 Additionally Verified

We have to prove the same things for every implemented enumeration algorithm and they seem to be fairly similar, because the algorithms use list comprehension and rely on the following pattern:

$$[f\ x\ y.\ x \leftarrow xs, y \leftarrow g\ x]$$

where xs is a list, g a function that takes elements from xs, and f a function that takes elements from xs as well as results from g, f can also be seen as a constructor for combinatorial objects. However, for some of the algorithms g and f take less arguments and thus the verification is easier, but this is the general case.

Therefore, it is quite obvious that we can show some abstract properties to simplify further proofs. Moreover, the libraries shown before lack some theorems and concepts, we want in our formalization. Thus, in this chapter additional infrastructure will be verified.

3.2.1 Distinctness

Firstly, proofs of whether the result of a function is distinct can be made easier. For this purpose *inj*2 will be introduced.

definition *inj2-on* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$ **where**
$inj2\text{-}on\ f\ A\ B \longleftrightarrow (\forall x1{\in}A.\ \forall x2{\in}A.\ \forall y1{\in}B.\ \forall y2{\in}B.\ f\ x1\ y1 = f\ x2\ y2 \longrightarrow x1 = x2 \wedge y1 = y2)$

abbreviation *inj2* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow bool$ **where**
$inj2\ f \equiv inj2\text{-}on\ f\ UNIV\ UNIV$

It is a property for functions which take two arguments, and holds if the uncurried version is injective. This property implies injectivity of the partially applied functions. It is used for functions which construct a combinatorial object and thus abstracts our enumeration functions. In addition, the proofs become simpler, since we don't need to care about currying. In this regard, following lemma shows the equivalence:

lemma *inj2-curried*: *inj2-on* (*curry f*) *A B* ⟷ *inj-on f* (*A*×*B*)
unfolding *inj2-on-def inj-on-def* **by** *auto*

The first useful lemma that can be shown with *inj*2 is an extension of the existing *distinct_map* to partly applied functions.

lemma *inj2-on-distinct-map*:
assumes *inj2-on f* {*x*} (*set xs*)
shows *distinct xs* = *distinct* (*map* (*f x*) *xs*)

Considering the shape of the enumeration algorithms, we can use *inj*2 and come up with a general lemma, which can be proven by an induction on xs and some auxiliary lemmas including *inj*2-*on-distinct-map*.

lemma *inj2-distinct-concat-map-function*:
assumes *inj2 f*
shows ⟦∀ *x* ∈ *set xs*. *distinct* (*g x*); *distinct xs*⟧ ⟹ *distinct* [*f x y*. *x* ← *xs*, *y* ← *g x*]

With this theorem we consequently only need to show ∀*x* ∈ *set xs*. *distinct* (*g x*), *distinct xs* and *inj*2 *f*, to prove the distinctness for each algorithms. An even more general theorem that uses *inj*2-*on* instead of *inj*2 would also be desirable, but it wasn't needed for any of the algorithms.

To give an overview, these lemmas may be used for the following functions which satisfy the *inj*2 property.

lemma *Cons-inj2*: *inj2* (#)
lemma *Cons-Suc-inj2*: *inj2* (λ*x ys*. *Suc x* # *ys*)
lemma *Node-right-inj2*: *inj2* (λ*l r*. *Node l e r*)
lemma *Node-left-inj2*: *inj2* (λ*r l*. *Node l e r*)

While the concept of *inj*2, could also be implemented as a locale in Isabelle, to generate various lemmas automatically for every *inj*2 function, it makes more sense to implement it as a simple assumption and only derive theorems when needed, because a locale would be a bit too much for such a small assumption, and this way it is more similar to the existing theorems with *inj*.

3.2.2 Cardinality Helpers

Helping lemmas can also be implemented for the purpose of proving cardinalities.

lemma *length-concat-map-function-sum-list*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow length\ (g\ x) = h\ x$
shows $length\ [f\ x\ r\ .\ x \leftarrow xs, r \leftarrow g\ x] = sum\text{-}list\ (map\ h\ xs)$

With this we can get rid of f as well as g and introduce a counting function h , such that, for the most part, we only have to reason about a series. The next two lemmas will also be helpful in this regard.

lemma *leq-sum-to-sum-list*: $(\sum x \leq n.\ f\ x) = (\sum x \leftarrow [0..<Suc\ n].\ f\ x)$

lemma *sum-list-extract-last*: $(\sum x \leftarrow [0..<Suc\ n].\ f\ x) = (\sum x \leftarrow [0..<n].\ f\ x) + f\ n$

3.2.3 Miscellaneous

Isabelle's list library doesn't contain many lemmas about *count_list*, since it is advised to use $count\ \circ\ mset$ instead, but for our purposes *count_list* is more suited, because the combinatorial algorithms enumerate lists and no unordered structures like multisets. Moreover, the multiset library does not have to be imported if we just use *count_list*. These four lemmas for *count_list* will be used in our formalizations and they could enhance the List library.

lemma *count-list-replicate*: $count\text{-}list\ (replicate\ x\ y)\ y = x$

lemma *count-list-full-elem*: $count\text{-}list\ xs\ y = length\ xs \longleftrightarrow (\forall x \in set\ xs.\ x = y)$

lemma *count-list-zero-not-elem*: $count\text{-}list\ xs\ x = 0 \longleftrightarrow x \notin set\ xs$

lemma *count-list-length-replicate*: $count\text{-}list\ xs\ y = length\ xs \longleftrightarrow xs = replicate\ (length\ xs)\ y$

3.2.4 Map Bool List

Now we take a look at concept which we will later use for bijections, in the algorithms of *n_subsets* and *powerset*. It is the filtering of lists with other lists that contain boolean values. Another interpretation would be the application of a bitmap. In this sense the boolean lists can be seen as binary numbers. There are many ways to implement *map_bool_list*, we chose the following, since it provides a convenient computational induction scheme.

fun *map-bool-list* :: *bool list* ⇒ *'a list* ⇒ *'a list* **where**
map-bool-list [] - = []
| *map-bool-list* - [] = []
| *map-bool-list* (*x*#*xs*) (*y*#*ys*) = (*if x then y*#(*map-bool-list xs ys*) *else* (*map-bool-list xs ys*))

Another way to implement this is with the *nth* function, of which the equivalence can be shown.

lemma *map-bool-list-set-nth*:
set (*map-bool-list xs ys*) = {*ys* ! *n* |*n*. *xs* ! *n* ∧ *n* < *length xs* ∧ *n* < *length ys* }

For the verification we then need theorems about *map_bool_list*, some of which are:

lemma *map-bool-list-elem*:
x ∈ *set* (*map-bool-list ys xs*) ⟹ *x* ∈ *set xs*

lemma *map-bool-list-card*:
⟦*distinct xs*; *length xs* = *length ys*⟧ ⟹ *card* (*set* (*map-bool-list ys xs*)) = *count-list ys True*

lemma *map-bool-list-distinct*:
distinct xs ⟹ *distinct* (*map-bool-list ys xs*)

lemma *map-bool-list-inj*:
distinct xs ⟹ *inj-on* (λ*ys*. *map-bool-list ys xs*) {*ys*. *length ys* = *length xs*}

4 Combinatorial Algorithms

Finally, we will start formalizing combinatorial enumeration algorithms. The functions will operate on lists, because lists are ordered, which makes it easier to use them for programs. If we then need a set for the verification, it will be substituted with *set xs* and *distinct xs* where *xs* is a list. For every combinatorial object the following will be provided.

- The object's formalized definition.
- An example for illustration.
- The actual enumeration function(s).
- A proof the show correctness, in the manner, that the set of what the enumeration function generates is equal to the object definition.
- Distinctness proof(s) where necessary, since our function(s) only operate on lists and the *set* conversion removes duplicates.
- A verification of the combinatorial object's set cardinality, which in some cases can be directly deduced from the infrastructure and in other cases will be shown by using the enumeration function.

Even though, in Isabelle there is no difference between lemmas and theorems, we will use **lemma** for auxiliary claims and **theorem** for the main statements, which were just mentioned. This notation should make the structure of the verification more visible.

4.1 n_Sequences

The probably most simple combinatorial object we are going to formalize is *n_sequences*, also called *n_tuples*, and since it is the first one we will do it in more detail and omit non interesting parts for further objects. *n_sequences* are defined as all lists of length n, which contains only elements from a given set A. In Isabelle it looks as follows.

definition *n-sequences* :: *'a set* ⇒ *nat* ⇒ *'a list set* **where**
n-sequences A n = {*xs*. *set xs* ⊆ *A* ∧ *length xs* = *n*}

One example would be:

$$n\_sequence\ \{0,1,2\}\ 2 = \{[0,0],[0,1],[0,2],[1,0],[1,1],[1,2],[2,0],[2,1],[2,2]\}$$

The list library from Isabelle HOL already contains a function to generate those lists, which has already been verified for correctness.

primrec *n-lists* :: *nat* ⇒ *'a list* ⇒ *'a list list* **where**
n-lists 0 xs = [[]] |
n-lists (*Suc n*) *xs* = *concat* (*map* (λ*ys*. *map* (λ*y*. *y* # *ys*) *xs*) (*n-lists n xs*))

lemma *set-n-lists*: *set* (*List.n-lists n xs*) = {*ys*. *length ys* = *n* ∧ *set ys* ⊆ *set xs*}

But to exemplify what we need to do for every enumeration function, it is worth taking a look at another one which has a similar shape to enumeration algorithms from the other combinatorial objects.

fun *n-sequence-enum* :: *'a list* ⇒ *nat* ⇒ *'a list list* **where**
n-sequence-enum xs 0 = [[]]
| *n-sequence-enum xs* (*Suc n*) = [*x*#*r* . *x* ← *xs*, *r* ← *n-sequence-enum xs n*]

This function produces the same *n_sequences* as *n_lists*, which can be shown by induction on *n*.

lemma *set* (*n-sequence-enum xs n*) = *set* (*List.n-lists n xs*)

Though *n_sequences_enum* is probably slower then *n_lists*, since it doesn't reuse the same *n_sequence_enum* call for each element of *xs*. A concept to make it more efficient is structural sharing. With this, previous parts get reused and don't consume additional memory for each version [9]. It can not only be used for *n_sequences*, but also for some of the other objects, whenever the combinatorial object's configurations build up on one another, like it is the case for *trees*.

For the correctness we need to proof that it only produces lists which are *n_sequences* and that the result contains all of them. Therefore, we need a proof by exhaustion, in the sense, that we have to show the subset relation in both directions. The proofs work in this case again straightforward with an induction on *n*.

theorem *n-sequence-enum-correct*:
set (*n-sequence-enum xs n*) = *n-sequences* (*set xs*) *n*
proof *standard*
show *set* (*n-sequence-enum xs n*) ⊆ *n-sequences* (*set xs*) *n*
next
show *n-sequences* (*set xs*) *n* ⊆ *set* (*n-sequence-enum xs n*)
qed

In distinctness proof, we can now use a lemma which follows from 3.2.1, where the *inj*2 function is just the *Cons* constructor of lists.

theorem *n-sequence-enum-distinct*: *distinct xs* ⟹ *distinct* (*n-sequence-enum xs n*)
by (*induct n*) (*auto simp*: *Cons-distinct-concat-map*)

While we could directly use the *card_lists_length_eq* lemma which already verifies the cardinality, it is also possible to use our algorithm now, to get an intuition of how it could be done for every other combinatorial object.

lemma *n-sequence-enum-length*: *length* (*n-sequence-enum xs n*) = (*length xs*) ^ *n*

theorem *n-sequences-card*:
assumes *finite A*
shows *card* (*n-sequences A n*) = *card A* ^ *n*

4.2 n_Permutations

A *n_permutation* is like a *n_sequence* with the additional restriction, that the list must be distinct. It is a permutation with a constrained length.

definition *n-permutations* :: *'a set* ⇒ *nat* ⇒ *'a list set* **where**
n-permutations A n = {*xs*. *set xs* ⊆ *A* ∧ *distinct xs* ∧ *length xs* = *n*}

An example: *n_permutation* {0, 1, 2} 2 = {[0, 1], [0, 2], [1, 0], [1, 2], [2, 0], [2, 1]}

The function works similar to the one from the Multiset Permutations library, but has an additional parameter *n* to limit the length. The set is here again passed as list, like we previously discussed.

fun *n-permutation-enum* :: *'a list* ⇒ *nat* ⇒ *'a list list* **where**
n-permutation-enum xs 0 = [[]]
| *n-permutation-enum xs* (*Suc n*) = [*x*#*r* . *x* ← *xs*, *r* ← *n-permutation-enum* (*remove1 x xs*) *n*]

For the correctness, we essentially need those four auxiliary lemmas.

lemma *n-permutation-enum-subset*: *ys* ∈ *set* (*n-permutation-enum xs n*) ⟹ *set ys* ⊆ *set xs*

lemma *n-permutation-enum-length*: *ys* ∈ *set* (*n-permutation-enum xs n*) ⟹ *length ys* = *n*

lemma *n-permutation-enum-elem-distinct*:
distinct xs ⟹ *ys* ∈ *set* (*n-permutation-enum xs n*) ⟹ *distinct ys*

lemma *n-permutation-enum-correct2*:
ys ∈ *n-permutations* (*set xs*) *n* ⟹ *ys* ∈ *set* (*n-permutation-enum xs n*)

To finally show the following, where the list *xs* is assumed to be distinct, since it is meant to represents a set.

theorem *n-permutation-enum-correct*:
distinct xs $\Longrightarrow$ *set* (*n-permutation-enum xs n*) = *n-permutations* (*set xs*) *n*

To verify, that no duplicates are generated *inj*2_*distinct_concat_map_function* is used again, but here the function *g*, which has to produce distinct results for an input *x* ∈ *set xs*, is *n_permutation_enum* (*remove*1 *x xs*) *n*, though this can be shown in the induction step, to then get:

theorem *n-permutation-distinct*: *distinct xs* $\Longrightarrow$ *distinct* (*n-permutation-enum xs n*)

For this object the cardinality has already been proven, in the Falling Factorial Sum library with the *card_lists_distinct_length_eq* lemma [4]. Therefore, we only need to adopt it to our definition, which gets us the following, where *ffact* is the failing factorial function.

theorem *finite A* $\Longrightarrow$ *card* (*n-permutations A n*) = *ffact n* (*card A*)

4.2.1 n_Multiset_Permutations

It is also worth saying, that the function we just verified cannot only be used to enumerate *n_permutation*, but also to enumerate *n_multiset_permutations*, if we apply a remove duplicates function.

fun *n-multiset-permutation-enum* :: *'a list* ⇒ *nat* ⇒ *'a list list* **where**
n-multiset-permutation-enum xs n = *remdups* (*n-permutation-enum xs n*)

This is how *n_multiset_permutations* are defined:

definition *n-multiset-permutations* :: *'a multiset* ⇒ *nat* ⇒ *'a list set* **where**
n-multiset-permutations A n = {*xs*. *mset xs* ⊆# *A* ∧ *length xs* = *n*}

Example: *n_multiset_permutation* {#0, 0, 1#} 3 = {#[0, 0, 1], [0, 1, 0], [1, 0, 0]#}

The correctness proof is simple, since there are some lemmas which can be reused, and the distinctness follows directly from the application of *remdups*.

lemma *n-multiset-permutation-enum-correct*:
set (*n-multiset-permutation-enum xs n*) = *n-multiset-permutations* (*mset xs*) *n*

lemma *distinct* (*n-multiset-permutation-enum xs n*)

However, because of *remdups*, this algorithm is not very efficient. A more efficient version could use *remdups* in every iteration, or somehow don't generate duplicates, by for example remembering already generated permutations, but this we didn't formalize.

4.3 Powerset

Next we will enumerate *powersets*, which are already defined in the Isabelle library.

definition *Pow* :: *'a set ⇒ 'a set set*
where *Pow-def*: $Pow\ A = \{B.\ B \subseteq A\}$

Example: $Pow\ \{0,1\} = \{\{\}, \{0\}, \{1\}, \{0,1\}\}$

To illustrate similarities to binary numbers and to define an alternative enumeration function, the algorithm doesn't directly enumerate the *powerset*, but instead uses boolean lists.

fun *all-bool-lists* :: *nat ⇒ bool list list* **where**
all-bool-lists 0 = [[]]
| *all-bool-lists* (*Suc x*) = *concat* [[*False*#*xs*, *True*#*xs*] . *xs* ← *all-bool-lists x*]

fun *powerset-enum* **where**
powerset-enum xs = [(*map-bool-list x xs*) . *x* ← *all-bool-lists* (*length xs*)]

Here the map bool list infrastructure finds its uses. To utilize it, first those two lemmas are needed.

lemma *distinct-all-bool-lists* : *distinct* (*all-bool-lists x*)

lemma *all-bool-lists-correct*: *set* (*all-bool-lists x*) = {*xs*. *length xs* = *x*}

And now the correctness and distinctness can be shown.

theorem *powerset-enum-correct*: *set* (*map set* (*powerset-enum xs*)) = *Pow* (*set xs*)

theorem *powerset-enum-distinct*: *distinct xs* ⟹ *distinct* (*powerset-enum xs*)

However, an additional lemma has to be verified, since for correctness the elements were mapped to sets.

theorem *powerset-enum-distinct-elem*: *distinct xs* ⟹ *ys* ∈ *set* (*powerset-enum xs*) ⟹ *distinct ys*

A cardinality proof for powersets does also already exist in the library.

lemma *card-Pow*: *finite A* ⟹ *card* (*Pow A*) = 2 ^ *card A*

4.3.1 Alternative Algorithm for *all_bool_lists*

For an alternative algorithm, *n_sequence_enum* can be used to generate boolean lists.

fun *all-bool-lists2* :: *nat* ⇒ *bool list list* **where**
all-bool-lists2 n = *n-sequence-enum* [*True*, *False*] *n*

We already have verified *n_sequence_enum*, therefore the proofs for correctness and distinctness can be concluded directly, and with them we could replace *all_bool_lists* with *all_bool_lists*2.

lemma *all-bool-lists2-distinct*: *distinct* (*all-bool-lists2 n*)

lemma *all-bool-lists2-correct*: *set* (*all-bool-lists n*) = *set* (*all-bool-lists2 n*)

4.4 n_Subsets

n_subsets are subsets that contain exactly *n* elements. They are like the powerset, with an additional restriction of the cardinality being *n*.

definition *n-subsets* :: *'a set* ⇒ *nat* ⇒ *'a set set* **where**
n-subsets A n = {*B*. *B* ⊆ *A* ∧ *card B* = *n*}

Example: *n_subsets* {0,1,2} 2 = {{0,1},{0,2},{1,2}}

For the algorithm, we will do the same we did with *n_sequence_enum*. Therefore, *n_bool_lists* does enumerate boolean lists, that contain exactly *n True* elements.

fun *n-bool-lists* :: *nat* ⇒ *nat* ⇒ *bool list list* **where**
n-bool-lists n 0 = (*if n* > *0 then* [] *else* [[]])
| *n-bool-lists n* (*Suc x*) = (*if n* = *0 then* [*replicate* (*Suc x*) *False*]
else if n = *Suc x then* [*replicate* (*Suc x*) *True*]
else if n > *x then* []
else [*False*#*xs* . *xs* ← *n-bool-lists n x*] @ [*True*#*xs* . *xs* ← *n-bool-lists* (*n*−*1*) *x*])

fun *n-subset-enum* :: *'a list* ⇒ *nat* ⇒ *'a list list* **where**
n-subset-enum xs n = [(*map-bool-list x xs*) . *x* ← (*n-bool-lists n* (*length xs*))]

n_bool_lists is a little bit more complicated than it needs to be, but replicating *True* or *False* when the rest only contains *True* or *False*, may make it more efficient, however with this the proofs become more difficult. Considering this, the following two lemmas need to be shown, and here some lemmas about *count_list*, we verified for the infrastructure, are applied.

lemma *n-bool-lists-distinct*: *distinct* (*n-bool-lists n x*)

lemma *n-bool-lists-correct*: *set* (*n-bool-lists n x*) = {*xs*. *length xs* = *x* ∧ *count-list xs True* = *n*}

Then we can infer the correctness and distinctness of *n_subsets*, with the same additional distinctness, discussed preversly for *powerset_enum*.

theorem *n-subset-enum-correct*:
distinct xs $\Longrightarrow$ *set* (*map set* (*n-subset-enum xs n*)) = *n-subsets* (*set xs*) *n*

theorem *n-subset-enum-distinct-elem*: *distinct xs* $\Longrightarrow$ *ys* $\in$ *set* (*n-subset-enum xs n*) $\Longrightarrow$ *distinct ys*

theorem *n-subset-enum-distinct*: *distinct xs* $\Longrightarrow$ *distinct* (*n-subset-enum xs n*)

The cardinality has here again already been shown. It is located in the Binomial standard library, where *choose* is the the binomial coefficient.

theorem *n-subsets*:
assumes *finite A*
shows *card* {*B*. *B* $\subseteq$ *A* $\wedge$ *card B* = *k*} = *card A choose k*

4.4.1 Alternative Algorithm for *n_bool_lists*

Another algorithm for *n_bool_lists* can also be verified. It uses *permutations_of_multiset* which has an enumeration algorithm in HOL-Combinatorics.Multiset_Permutations, but due to the fact, that it uses *permutations_of_multiset*, *n_bool_lists*2 is properly less efficient.

fun *n-bool-lists2* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool list set* **where**
n-bool-lists2 n x = (*if n* > *x then* {} *else*
permutations-of-multiset (*mset* (*replicate n True* @ *replicate* (*x*−*n*) *False*)))

To use it instead of *n_bool_lists*, another lemmas was verified, but we don't need a distinctness lemma, since this algorithm, as an exception, indirectly operates on sets.

lemma *n-bool-lists2-correct*: *set* (*n-bool-lists n x*) = *n-bool-lists2 n x*

4.5 Integer_Compositions

Until now, we had only formalized combinatorial objects, which depend on a provided set, but the next objects will only take numbers. In this sense, we will define *integer_compositions*. It can be understood as the ways of how a natural number can be split into non zero summands, where the order matters. Due to this, the definition uses lists. If we would additionally restrict the list's length, *integer_compositions* should be equal to the stars and bars problem of first kind.

definition *integer-compositions* :: *nat* ⇒ *nat list set* **where**
integer-compositions i = {*xs*. *sum-list xs* = *i* ∧ 0 ∉ *set xs*}

Example: $integer\_compositions\ 3 = \{[3], [2,1], [1,2], [1,1,1]\}$"

The algorithm follows the patter we have seen previously, but this time the recursive call depends on two arguments, n and m.

fun *integer-composition-enum* :: *nat* ⇒ *nat list list* **where**
integer-composition-enum 0 = [[]]
| *integer-composition-enum* (*Suc n*) =
[*Suc m #xs*. *m* ← [*0..*< *Suc n*], *xs* ← *integer-composition-enum* (*n*−*m*)]

Showing the correctness in this case is a bit more difficult, since the recursive call takes any *nat* smaller to the previous one. However, with the right induction and some auxiliary lemmas the correctness follows.

theorem *integer-composition-enum-correct*: *set* (*integer-composition-enum n*) = *integer-compositions n*

For the distinctness $inj2\_distinct\_concat\_map\_function$ can be used. The $inj2$ function is in this case not just *Cons*, but $(\lambda x\ xs.\ Cons\ (Suc\ x)\ xs)$.

theorem *integer-composition-enum-distinct*: *distinct* (*integer-composition-enum n*)

The $length\_concat\_map\_function\_sum\_list$ lemma from our cardinality helper infrastructure will be used now, so that we essentially only have to show the equivalence of a series, to verify the cardinality.

lemma *sum-list-two-pow*: *Suc* (∑*x*←[*0..*<*n*]. *2* ^ (*n* − *Suc x*)) = *2* ^ *n*

lemma *integer-composition-enum-length*: *length* (*integer-composition-enum n*) = *2*^(*n*−*1*)

theorem *integer-compositions-card*: *card* (*integer-compositions n*) = *2*^(*n*−*1*)

4.6 Integer_Partitions

integer_partitions are like *integer_compositions*, with the only difference that their elements aren't ordered. Therefore, they are represented by multisets. They are equal to Young diagrams [12].

definition *integer-partitions* :: *nat* ⇒ *nat multiset set* **where**
integer-partitions i = {*xs*. *sum-mset xs* = *i* ∧ 0 ∉# *xs*}

Example: $integer\_partitions\ 4 = \{\{4\}, \{3,1\}, \{2,2\}\{2,1,1\}, \{1,1,1,1\}\}$

The enumeration function uses an auxiliary function, which carries a parameter m to limit how big the split integers can be. Therefore, each produced integer partition is sorted in descending order.

fun *integer-partitions-enum-aux* :: *nat* ⇒ *nat* ⇒ *nat list list* **where**
integer-partitions-enum-aux 0 m = [[]]
| *integer-partitions-enum-aux n m* =
[*h*#*r* . *h* ← [1..< *Suc* (*min n m*)], *r* ← *integer-partitions-enum-aux* (*n*−*h*) *h*]

fun *integer-partitions-enum* :: *nat* ⇒ *nat list list* **where**
integer-partitions-enum n = *integer-partitions-enum-aux n n*

By showing what we expect from *integer_partitions_enum_aux* we can then prove the correctness of *integer_partitions_enum*.

lemma *integer-partitions-enum-aux-max*:
xs ∈ *set* (*integer-partitions-enum-aux n m*) ⟹ *x* ∈ *set xs* ⟹ *x* ≤ *m*
theorem *integer-partitions-enum-correct*:
set (*map mset* (*integer-partitions-enum n*)) = *integer-partitions n*

The proof for distinctness isn't very interesting, since it doesn't involve anything new.

theorem *integer-partitions-enum-distinct*: *distinct* (*integer-partitions-enum n*)

For the cardinality we can rely on the Cardinality of Number Partitions library. However, we need one additional lemma, that surprisingly doesn't already exist. It can be shown by applying a bijection with the *count* function for multisets. If it isn't already clear *Partition* is the partition function. It can be found in the library.

lemma *card-partitions-number-partition*: *card* {*p*. *p partitions n*} = *card* {*N*. *number-partition n N*}

theorem *integer-partitions-cardinality*: *card* (*integer-partitions n*) = *Partition* (2∗*n*) *n*

4.7 Weak_Integer_Compositions

Another combinatorial object is *weak_integer_compositions*. They are *integer_compositions*, which are weak in the sense that they can contain zeros. In addition, the length is fixed, since otherwise infinitely many configurations would be possible. *weak_integer_compositions* are also equal to stars and bars of second kind.

definition *weak-integer-compositions* :: *nat* ⇒ *nat* ⇒ *nat list set* **where**
weak-integer-compositions i l = {*xs*. *length xs* = *l* ∧ *sum-list xs* = *i*}

Example: *weak_integer_compositions* 2 2 = {[2, 0], [1, 1], [0, 2]}

The algorithm is quite simple, but the base cases may not seem to be obvious.

fun *weak-integer-composition-enum* :: *nat* ⇒ *nat* ⇒ *nat list list* **where**
weak-integer-composition-enum i 0 = (*if i* = *0 then* [[]] *else* [])
| *weak-integer-composition-enum i* (*Suc 0*) = [[*i*]]
| *weak-integer-composition-enum i l* =
[*h*#*r* . *h* ← [*0*..< *Suc i*], *r* ← *weak-integer-composition-enum* (*i*−*h*) (*l*−*1*)]

The proofs for correctness and distinctness are here again not very interesting.

theorem *weak-integer-composition-enum-correct*:
set (*weak-integer-composition-enum i l*) = *weak-integer-compositions i l*

theorem *weak-integer-composition-enum-distinct*: *distinct* (*weak-integer-composition-enum i l*)

For the cardinality however, we need to show something more exciting. We are verifying it by again arguing about a series. Moreover, *multichoose* will be defined as the multiset coefficient.

definition *multichoose*:: *nat* ⇒ *nat* ⇒ *nat* (**infixl** *multichoose 65*) **where**
n multichoose k = (*n* + *k* −*1*) *choose k*

lemma *a-choose-equivalence*: *Suc* (∑*x*←[*0*..<*k*]. *n* + (*k* − *x*) *choose* (*k* − *x*)) = *Suc* (*n* + *k*) *choose k*

lemma *composition-enum-length*: *length* (*weak-integer-composition-enum i n*) = *n multichoose i*

theorem *weak-integer-compositions-cardinality*: *card* (*weak-integer-compositions n k*) = *k multichoose n*

4.8 Trees

The last structure we will formalize an algorithm for is *trees*. My supervisor, Prof. Tobias Nipkow has already provided the definition, an algorithm, the cardinality proof, and the basis of the distinctness lemma. We are only interested in the structure of the tree and don't care about what it stores. Therefore, all Nodes will be the Unity (). The *trees* we enumerate are also called binary labeled trees, since the two child nodes have an order.

definition *trees* :: *nat* ⇒ *unit tree set* **where**
trees n = {*t*. *size t* = *n*}

fun *tree-enum* :: *nat* ⇒ *unit tree list* **where**
tree-enum 0 = [*Leaf*]
| *tree-enum* (*Suc n*) = [*Node t1* () *t2*. *i* ← [*0*..<*Suc n*], *t1* ← *tree-enum i*, *t2* ← *tree-enum* (*n*−*i*)]

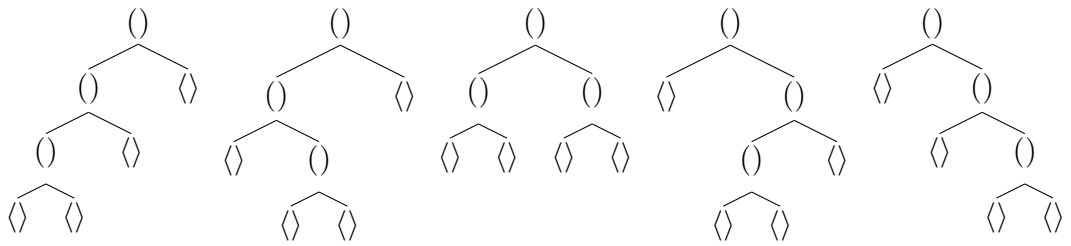

Figure 4.1: Example: *trees* 3

The cardinality was proven with an induction scheme from catalan numbers, which can be found in the Catalan Numbers library.

theorem *length* (*tree-enum n*) = *catalan n*

By using the computational induction scheme of *tree_enum*, the correctness can be verified.

theorem *tree-enum-correct*: *set*(*tree-enum n*) = *trees n*

However, showing that no duplicates will be produced is more complex, since this algorithm has two recursive calls and is different from the pattern we have seen so far. Nevertheless, we can still make use of our infrastructure for distinctness, where (*λl r.Node l* () *r*) and (*λr l. Node l* () *r*) satisfy *inj*2. With this we can show the following three auxiliary lemmas, to finally get the distinctness.

lemma *tree-enum-elem-injective2*:
x ∈ *set* (*tree-enum n*) ⟹ *y* ∈ *set* (*tree-enum m*) ⟹ *x* = *y* ⟹ *n* = *m*

lemma *tree-enum-distinct-aux-outer*:
assumes ∀ *i* ≤ *n*. *distinct* (*tree-enum i*)
and *distinct xs*
and ∀ *i* ∈ *set xs*. *i* < *n*
and *sorted-wrt* (<) *xs*
shows *distinct* (*map* (*λi.* [*Node l* () *r*. *l* ← *tree-enum i*, *r* ← *tree-enum* (*n*−*i*)]) *xs*)

lemma *tree-enum-distinct-aux-left*:
∀ *i* < *n*. *distinct* (*tree-enum i*) ⟹ *distinct* ([*Node l* () *r*. *i* ← [0..< *n*], *l* ← *tree-enum i*])

theorem *tree-enum-distinct*: *distinct*(*tree-enum n*)

5 Conclusion

The formalization did work well with Isabelle, and I successfully formalized enumeration algorithms for eight combinatorial objects, if *n_multiset_permutations* and the boolean lists aren't counted separately. It took about 2000 lines of code, and not only the resulting theorems, but also some of the infrastructure may be useful for further projects. In this regard, it is planned to make an AFP entry out of this thesis, so that other people can use it. However, there are still many things which can or should be done.

5.1 Further Work

Firstly, fitting combinatorial objects should become properly connected to the Twelvefold Way AFP entry. Also, for the objects discussed in this thesis, it is desirable to have more different enumeration algorithms, for example for *powersets* and *n_subsets*, algorithms which don't use boolean lists.

Moreover, a lot more combinatorial objects exist. Some were discussed in Chapter 2, and way more can be found. For every of those combinatorial objects enumeration algorithms could be formally formalized.

List of Figures

List of Tables

Bibliography

[1] R. C. Bose and B. Grünbaum. "combinatorics." In: *Encyclopedia Britannica* (Aug. 2022). https://www.britannica.com/science/combinatorics, [Online; accessed 22. Aug. 2022].

[2] L. Bulwahn. "Cardinality of Number Partitions." In: *Archive of Formal Proofs* (Jan. 2016). https://isa-afp.org/entries/Card_Number_Partitions.html, Formal proof development. ISSN: 2150-914x.

[3] L. Bulwahn. "Derangements Formula." In: *Archive of Formal Proofs* (June 2015). https://isa-afp.org/entries/Derangements.html, Formal proof development. ISSN: 2150-914x.

[4] L. Bulwahn. "The Falling Factorial of a Sum." In: *Archive of Formal Proofs* (Dec. 2017). https://isa-afp.org/entries/Falling_Factorial_Sum.html, Formal proof development. ISSN: 2150-914x.

[5] L. Bulwahn. "The Twelvefold Way." In: *Archive of Formal Proofs* (Dec. 2016). https://isa-afp.org/entries/Twelvefold_Way.html, Formal proof development. ISSN: 2150-914x.

[6] M. Eberl. "Catalan Numbers." In: *Archive of Formal Proofs* (June 2016). https://isa-afp.org/entries/Catalan_Numbers.html, Formal proof development. ISSN: 2150-914x.

[7] E. Karayel. "Enumeration of Equivalence Relations." In: *Archive of Formal Proofs* (Feb. 2022). https://isa-afp.org/entries/Equivalence_Relation_Enumeration.html, Formal proof development. ISSN: 2150-914x.

[8] D. E. Knuth. *The Art of Computer Programming. Volume 4A. Combinatorial Algorithms. Part 1*. Addison-Wesley Professional. ISBN: 978-0-13439457-2.

[9] D. Larchey-Wendling, D. Méry, and D. Galmiche. "STRIP: Structural Sharing for Efficient Proof-Search." In: *Automated Reasoning*. Ed. by R. Goré, A. Leitsch, and T. Nipkow. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 696–700. ISBN: 978-3-540-45744-2.

[10] A. Nijenhuis, H. Wilf, and W. Rheinboldt. *Combinatorial Algorithms: For Computers and Calculators*. Computer science and applied mathematics. Elsevier Science. ISBN: 9781483273457.

[11] R. Stanley. *Enumerative Combinatorics: Volume 1*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2011. ISBN: 9781139505369.

[12] D. Stanton and D. White. *Constructive Combinatorics*. New York, NY, USA: Springer. ISBN: 978-1-4612-4968-9.

[13] L. P. Tobias Nipkow. *Isabelle Webpage*. https://isabelle.in.tum.de/index.html, [Online; accessed 31. Aug. 2022]. Dec. 2021.