

Combinatorial Enumeration Algorithms

Paul Hofmeier and Emin Karayel

May 26, 2024

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`, `Derangements` and `Trees`, this entry formalizes efficient functional programs and verifies their correctness. In addition, it provides cardinality proofs for those combinatorial objects. Some cardinalities are verified using the enumeration functions and others are shown using existing libraries including other AFP entries.

Related books on combinatorics include [4] and [5]. Some of the cardinality theorems in this entry are also proved in another AFP entry, *The Twelffold Way* [3].

Contents

1	Injectivity for two argument functions	3
1.1	Correspondence between <i>inj2-on</i> and <i>inj-on</i>	4
1.2	Proofs with <i>inj2</i>	4
1.3	Specializations of <i>inj2</i>	5
1.3.1	<i>Cons</i>	5
1.3.2	<i>Node right</i>	5
1.3.3	<i>Node left</i>	5
1.3.4	<i>Cons Suc</i>	6
2	Lemmas for cardinality proofs	6
3	Miscellaneous	6
3.1	<i>count-list</i> and <i>replicate</i>	6

4	N-Sequences	7
4.1	Definition	7
4.2	Algorithm	7
4.3	Verification	8
4.3.1	Correctness	8
4.3.2	Distinctness	8
4.3.3	Cardinality	8
5	N-Permutations	8
5.1	Definition	8
5.2	Algorithm	9
5.3	Verification	9
5.3.1	Correctness	9
5.3.2	Distinctness	9
5.3.3	Cardinality	9
5.4	<i>n-multiset</i> extension (with <i>remdups</i>)	10
6	N-Subsets	12
6.1	Definition	12
6.2	Algorithm	12
6.3	Verification	12
6.3.1	<i>n-bool-lists</i>	12
6.3.2	Correctness	13
6.3.3	Distinctness	13
6.3.4	Cardinality	13
6.4	Alternative using Multiset permutations	13
6.5	<i>mset-count</i>	14
7	Powerset	14
7.1	Definition	14
7.2	Algorithm	14
7.3	Verification	15
7.3.1	Correctness	15
7.3.2	Distinctness	15
7.3.3	Cardinality	15
7.4	Alternative algorithm with <i>n-sequence-enum</i>	15
8	Integer Partitions	16
8.1	Definition	16
8.2	Algorithm	16
8.3	Verification	16
8.3.1	Correctness	16
8.3.2	Distinctness	17
8.3.3	Cardinality	18

9 Integer Compositions	18
9.1 Definition	18
9.2 Algorithm	18
9.3 Verification	19
9.3.1 Correctness	19
9.3.2 Distinctness	19
9.3.3 Cardinality	20
10 Weak Integer Compositions	20
10.1 Definition	20
10.2 Algorithm	20
10.3 Verification	21
10.3.1 Correctness	21
10.3.2 Distinctness	21
10.3.3 Cardinality	21
11 Derangements	22
11.1 Definition	22
11.2 Algorithm	22
11.3 Verification	23
11.3.1 Correctness	23
11.3.2 Distinctness	23
12 Trees	24
12.1 Definition	24
12.2 Algorithm	24
12.3 Verification	24
12.3.1 Cardinality	24
12.3.2 Correctness	24
12.3.3 Distinctness	25

1 Injectivity for two argument functions

```

theory Common-Lemmas
  imports
    HOL.List
    HOL-Library.Tree
begin

```

This section introduces *inj2-on* which generalizes *inj-on* on curried functions with two arguments and contains subsequent theorems about such functions.

We could use curried function directly with for example *case-prod*, but this way the proofs become simpler and easier to read.

```

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**
 $inj2 f \equiv inj2\text{-on } f UNIV UNIV$

1.1 Correspondence between $inj2\text{-on}$ and $inj\text{-on}$

lemma $inj2\text{-curried}: inj2\text{-on } (curry f) A B \longleftrightarrow inj\text{-on } f (A \times B)$
 $\langle proof \rangle$

lemma $inj2\text{-on-all}: inj2 f \Longrightarrow inj2\text{-on } f A B$
 $\langle proof \rangle$

lemma $inj2\text{-inj-first}: inj2 f \Longrightarrow inj f$
 $\langle proof \rangle$

lemma $inj2\text{-inj-second}: inj2 f \Longrightarrow inj (f x)$
 $\langle proof \rangle$

lemma $inj2\text{-inj-second-flipped}: inj2 f \Longrightarrow inj (\lambda x. f x y)$
 $\langle proof \rangle$

1.2 Proofs with $inj2$

Already existing for inj :

thm $distinct\text{-map}$

lemma $inj2\text{-on-distinct-map}$:
assumes $inj2\text{-on } f \{x\}$ ($set xs$)
shows $distinct xs = distinct (map (f x) xs)$
 $\langle proof \rangle$

lemma $inj2\text{-distinct-map}$:
assumes $inj2 f$
shows $distinct xs = distinct (map (f x) xs)$
 $\langle proof \rangle$

lemma $inj2\text{-on-distinct-concat-map}$:
assumes $inj2\text{-on } f (set xs) (set ys)$
shows $\llbracket distinct ys; distinct xs \rrbracket \Longrightarrow distinct [f x y. x \leftarrow xs, y \leftarrow ys]$
 $\langle proof \rangle$

lemma $inj2\text{-distinct-concat-map}$:
assumes $inj2 f$
shows $\llbracket distinct ys; distinct xs \rrbracket \Longrightarrow distinct [f x y. x \leftarrow xs, y \leftarrow ys]$
 $\langle proof \rangle$

lemma $inj2\text{-distinct-concat-map-function}$:

assumes *inj2* *f*
shows $\llbracket \forall x \in \text{set } xs. \text{distinct } (g \ x); \text{distinct } xs \rrbracket \implies \text{distinct } [f \ x \ y. x \leftarrow xs, y \leftarrow g \ x]$
<proof>

lemma *distinct-concat-Nil*: $\text{distinct } (\text{concat } (\text{map } (\lambda y. []) \ xs))$
<proof>

lemma *inj2-distinct-concat-map-function-filter*:

assumes *inj2* *f*
shows $\llbracket \forall x \in \text{set } xs. \text{distinct } (g \ x); \text{distinct } xs \rrbracket \implies \text{distinct } [f \ x \ y. x \leftarrow xs, y \leftarrow g \ x, h \ x]$
<proof>

1.3 Specializations of *inj2*

1.3.1 Cons

lemma *Cons-inj2*: $\text{inj2 } (\#)$
<proof>

lemma *Cons-distinct-concat-map*: $\llbracket \text{distinct } ys; \text{distinct } xs \rrbracket \implies \text{distinct } [x \# y. x \leftarrow xs, y \leftarrow ys]$
<proof>

lemma *Cons-distinct-concat-map-function*:

$\llbracket \forall x \in \text{set } xs. \text{distinct } (g \ x); \text{distinct } xs \rrbracket \implies \text{distinct } [x \# y. x \leftarrow xs, y \leftarrow g \ x]$
<proof>

lemma *Cons-distinct-concat-map-function-distinct-on-all*:

$\llbracket \forall x. \text{distinct } (g \ x); \text{distinct } xs \rrbracket \implies \text{distinct } [x \# y. x \leftarrow xs, y \leftarrow g \ x]$
<proof>

1.3.2 Node right

lemma *Node-right-inj2*: $\text{inj2 } (\lambda l \ r. \text{Node } l \ e \ r)$
<proof>

lemma *Node-right-distinct-concat-map*:

$\llbracket \text{distinct } ys; \text{distinct } xs \rrbracket \implies \text{distinct } [\text{Node } x \ e \ y. x \leftarrow xs, y \leftarrow ys]$
<proof>

1.3.3 Node left

lemma *Node-left-inj2*: $\text{inj2 } (\lambda r \ l. \text{Node } l \ e \ r)$
<proof>

lemma *Node-left-distinct-map*: $\text{distinct } xs = \text{distinct } (\text{map } (\lambda l. \langle l, () \rangle, r)) \ xs)$
<proof>

1.3.4 Cons Suc

lemma *Cons-Suc-inj2*: $\text{inj2 } (\lambda x \text{ ys. } \text{Suc } x \# \text{ys})$
<proof>

lemma *Cons-Suc-distinct-concat-map-function*:

$\llbracket \forall x \in \text{set } xs. \text{distinct } (g \ x) ; \text{distinct } xs \rrbracket \implies \text{distinct } [\text{Suc } x \# y. x \leftarrow xs, y \leftarrow g \ x]$
<proof>

2 Lemmas for cardinality proofs

lemma *length-concat-map*: $\text{length } [f \ x \ r . x \leftarrow xs, r \leftarrow ys] = \text{length } ys * \text{length } xs$
<proof>

An useful extension to *length-concat*

thm *length-concat*

lemma *length-concat-map-function-sum-list*:

assumes $\bigwedge x. x \in \text{set } xs \implies \text{length } (g \ x) = h \ x$

shows $\text{length } [f \ x \ r . x \leftarrow xs, r \leftarrow g \ x] = \text{sum-list } (\text{map } h \ xs)$

<proof>

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

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

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

3 Miscellaneous

Similar to *length-remove1*:

lemma *Suc-length-remove1*: $x \in \text{set } xs \implies \text{Suc } (\text{length } (\text{remove1 } x \ xs)) = \text{length } xs$
<proof>

3.1 count-list and replicate

HOL.List doesn't have many lemmas about *count-list* (when not using multisets)

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

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

The following lemma verifies the reverse of *count-notin*:

thm *count-notin*

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

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

lemma *count-list-True-False*: $\text{count-list } xs \ \text{True} + \text{count-list } xs \ \text{False} = \text{length } xs$
(*proof*)

end

4 N-Sequences

theory *n-Sequences*

imports

HOL.List

Common-Lemmas

begin

4.1 Definition

definition *n-sequences* :: $'a \ \text{set} \Rightarrow \text{nat} \Rightarrow 'a \ \text{list set}$ **where**
 $n\text{-sequences } A \ n = \{xs. \ \text{set } xs \subseteq A \wedge \text{length } xs = n\}$

Cardinality: $\text{card } A \ ^n$

Example: $n\text{-sequences } \{0, 1\} \ 2 = \{[0,0], [0,1], [1,0], [1,1]\}$

4.2 Algorithm

fun *n-sequence-enum* :: $'a \ \text{list} \Rightarrow \text{nat} \Rightarrow 'a \ \text{list list}$ **where**

$n\text{-sequence-enum } xs \ 0 = []$

| $n\text{-sequence-enum } xs \ (\text{Suc } n) = [x\#r . \ x \leftarrow xs, \ r \leftarrow n\text{-sequence-enum } xs \ n]$

An enumeration of n-sequences already exists: *n-lists*. This part of this AFP entry is mostly to establish the patterns used in the more complex combinatorial objects.

lemma $\text{set } (n\text{-sequence-enum } xs \ n) = \text{set } (\text{List.n-lists } n \ xs)$
(*proof*)

thm *set-n-lists*

4.3 Verification

4.3.1 Correctness

theorem *n-sequence-enum-correct:*

set (n-sequence-enum xs n) = n-sequences (set xs) n
{proof}

4.3.2 Distinctness

theorem *n-sequence-enum-distinct:*

distinct xs \implies distinct (n-sequence-enum xs n)
{proof}

4.3.3 Cardinality

lemma *n-sequence-enum-length:*

length (n-sequence-enum xs n) = (length xs) ^ n
{proof}

of course *card-lists-length-eq* can directly proof it but we want to derive it from *n-sequence-enum-length*

thm *card-lists-length-eq*

theorem *n-sequences-card:*

assumes *finite A*

shows *card (n-sequences A n) = card A ^ n*

{proof}

end

5 N-Permutations

theory *n-Permutations*

imports

HOL-Combinatorics.Multiset-Permutations

Common-Lemmas

Falling-Factorial-Sum.Falling-Factorial-Sum-Combinatorics

begin

5.1 Definition

definition *n-permutations* :: 'a set \Rightarrow nat \Rightarrow 'a list set **where**

n-permutations A n = {xs. set xs \subseteq A \wedge distinct xs \wedge length xs = n}

Permutations with a maximum length. They are different from *HOL-Combinatorics.Multiset-Permut* because the entries must all be distinct.

Cardinality: 'falling factorial' (card A) n

Example: n -permutations $\{0,1,2\}$ $2 = \{[0,1], [0,2], [1,0], [1,2], [2,0], [2,1]\}$

lemma *permutations-of-set* $A \subseteq n$ -permutations A (*card* A)
 ⟨*proof*⟩

5.2 Algorithm

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

5.3 Verification

5.3.1 Correctness

lemma *n-permutation-enum-subset*: $ys \in \text{set } (n\text{-permutation-enum } xs \ n) \Longrightarrow \text{set } ys \subseteq \text{set } xs$
 ⟨*proof*⟩

lemma *n-permutation-enum-length*: $ys \in \text{set } (n\text{-permutation-enum } xs \ n) \Longrightarrow \text{length } ys = n$
 ⟨*proof*⟩

lemma *n-permutation-enum-elem-distinct*: $\text{distinct } xs \Longrightarrow ys \in \text{set } (n\text{-permutation-enum } xs \ n) \Longrightarrow \text{distinct } ys$
 ⟨*proof*⟩

lemma *n-permutation-enum-correct1*: $\text{distinct } xs \Longrightarrow \text{set } (n\text{-permutation-enum } xs \ n) \subseteq n\text{-permutations } (\text{set } xs) \ n$
 ⟨*proof*⟩

lemma *n-permutation-enum-correct2*: $ys \in n\text{-permutations } (\text{set } xs) \ n \Longrightarrow ys \in \text{set } (n\text{-permutation-enum } xs \ n)$
 ⟨*proof*⟩

theorem *n-permutation-enum-correct*: $\text{distinct } xs \Longrightarrow \text{set } (n\text{-permutation-enum } xs \ n) = n\text{-permutations } (\text{set } xs) \ n$
 ⟨*proof*⟩

5.3.2 Distinctness

theorem *n-permutation-distinct*: $\text{distinct } xs \Longrightarrow \text{distinct } (n\text{-permutation-enum } xs \ n)$
 ⟨*proof*⟩

5.3.3 Cardinality

thm *card-lists-distinct-length-eq*

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

<proof>

5.4 *n*-multiset extension (with remdups)

definition *n*-multiset-permutations :: 'a multiset \Rightarrow nat \Rightarrow 'a list set **where**
n-multiset-permutations *A n* = {*xs*. mset *xs* $\subseteq\#$ *A* \wedge length *xs* = *n*}

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

lemma *distinct* (*n*-multiset-permutation-enum *xs n*)
<proof>

lemma *n*-multiset-permutation-enum-correct1:
mset *ys* $\subseteq\#$ mset *xs* \Longrightarrow *ys* \in set (*n*-permutation-enum *xs* (length *ys*))
<proof>

lemma *n*-multiset-permutation-enum-correct2:
ys \in set (*n*-permutation-enum *xs n*) \Longrightarrow mset *ys* $\subseteq\#$ mset *xs*
<proof>

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

end

theory *Filter-Bool-List*
imports
HOL.List
begin

A simple algorithm to filter a list by a boolean list. A different approach would be to filter by a set of indices, but this approach is faster, because lookups are slow in ML.

fun *filter-bool-list* :: bool list \Rightarrow 'a list \Rightarrow 'a list **where**
filter-bool-list [] - = []
| *filter-bool-list* - [] = []
| *filter-bool-list* (b#bs) (x#xs) =
 (if b then x#(*filter-bool-list* bs xs) else (*filter-bool-list* bs xs))

The following could be an alternative definition, but the version above provides a nice computational induction rule.

lemma *filter-bool-list* bs *xs* = map snd (*filter* fst (*zip* bs *xs*))
<proof>

lemma *filter-bool-list-in*:
n < length *xs* \Longrightarrow *n* < length *bs* \Longrightarrow bs!*n* \Longrightarrow xs!*n* \in set (*filter-bool-list* bs *xs*)
<proof>

lemma *filter-bool-list-not-elim*: $x \notin \text{set } xs \implies x \notin \text{set } (\text{filter-bool-list } bs \ xs)$
 ⟨proof⟩

lemma *filter-bool-list-elim*: $x \in \text{set } (\text{filter-bool-list } bs \ xs) \implies x \in \text{set } xs$
 ⟨proof⟩

lemma *filter-bool-list-not-in*:
 $\text{distinct } xs \implies n < \text{length } xs \implies n < \text{length } bs \implies bs!n = \text{False}$
 $\implies xs!n \notin \text{set } (\text{filter-bool-list } bs \ xs)$
 ⟨proof⟩

lemma *filter-bool-list-elim-nth*: $ys \in \text{set } (\text{filter-bool-list } bs \ xs)$
 $\implies \exists n. ys = xs ! n \wedge bs ! n \wedge n < \text{length } bs \wedge n < \text{length } xs$
 ⟨proof⟩

May be a useful conversion, since the algorithm could also be implemented with a list of indices.

lemma *filter-bool-list-set-nth*:
 $\text{set } (\text{filter-bool-list } bs \ xs) = \{xs ! n \mid n. bs ! n \wedge n < \text{length } bs \wedge n < \text{length } xs\}$
 ⟨proof⟩

lemma *filter-bool-list-exist-length*: $A \subseteq \text{set } xs$
 $\implies \exists bs. \text{length } bs = \text{length } xs \wedge A = \text{set } (\text{filter-bool-list } bs \ xs)$
 ⟨proof⟩

lemma *filter-bool-list-card*:
 $\llbracket \text{distinct } xs; \text{length } xs = \text{length } bs \rrbracket \implies \text{card } (\text{set } (\text{filter-bool-list } bs \ xs)) = \text{count-list } bs \ \text{True}$
 ⟨proof⟩

lemma *filter-bool-list-exist-length-card-True*: $\llbracket \text{distinct } xs; A \subseteq \text{set } xs; n = \text{card } A \rrbracket$
 $\implies \exists bs. \text{length } bs = \text{length } xs \wedge \text{count-list } bs \ \text{True} = \text{card } A \wedge A = \text{set } (\text{filter-bool-list } bs \ xs)$
 ⟨proof⟩

lemma *filter-bool-list-distinct*: $\text{distinct } xs \implies \text{distinct } (\text{filter-bool-list } bs \ xs)$
 ⟨proof⟩

lemma *filter-bool-list-inj-aux*:
assumes $\text{length } bs1 = \text{length } xs$
and $\text{length } xs = \text{length } bs2$
and $\text{distinct } xs$
shows $\text{filter-bool-list } bs1 \ xs = \text{filter-bool-list } bs2 \ xs \implies bs1 = bs2$
 ⟨proof⟩

lemma *filter-bool-list-inj*:
 $\text{distinct } xs \implies \text{inj-on } (\lambda bs. \text{filter-bool-list } bs \ xs) \ \{bs. \text{length } bs = \text{length } xs\}$
 ⟨proof⟩

end

6 N-Subsets

```
theory n-Subsets
  imports
    Common-Lemmas
    HOL-Combinatorics.Multiset-Permutations
    Filter-Bool-List
begin
```

6.1 Definition

definition *n-subsets* :: 'a set \Rightarrow nat \Rightarrow 'a set set **where**
n-subsets A n = {B. B \subseteq A \wedge card B = n}

Cardinality: *binomial* (card A) n

Example: *n-subsets* {0,1,2} 2 = {{0,1}, {0,2}, {1,2}}

6.2 Algorithm

```
fun n-bool-lists :: nat  $\Rightarrow$  nat  $\Rightarrow$  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  $\leftarrow$  n-bool-lists n x] @ [True#xs . xs  $\leftarrow$  n-bool-lists (n-1) x])
```

```
fun n-subset-enum :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  'a list list where
  n-subset-enum xs n = [(filter-bool-list bs xs) . bs  $\leftarrow$  (n-bool-lists n (length xs))]
```

6.3 Verification

6.3.1 n-bool-lists

lemma *n-bool-lists-True-count*: $xs \in \text{set } (n\text{-bool-lists } n \ x) \Longrightarrow \text{count-list } xs \ \text{True} = n$
<proof>

lemma *n-bool-lists-length*: $xs \in \text{set } (n\text{-bool-lists } n \ x) \Longrightarrow \text{length } xs = x$
<proof>

lemma *n-bool-lists-distinct*: *distinct* (n-bool-lists n x)
<proof>

lemma *replicate-True-not-False*: $\text{count-list } ys \ \text{True} = 0 \longleftrightarrow ys = \text{replicate } (\text{length } ys) \ \text{False}$

<proof>

lemma *n-bool-lists-correct-aux:*

$length\ xs = x \implies count\text{-}list\ xs\ True = n \implies xs \in set\ (n\text{-}bool\text{-}lists\ n\ x)$

<proof>

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

<proof>

6.3.2 Correctness

lemma *n-subset-enum-correct-aux1:*

$\llbracket distinct\ xs;\ length\ ys = length\ xs \rrbracket$

$\implies set\ (filter\text{-}bool\text{-}list\ ys\ xs) \in n\text{-}subsets\ (set\ xs)\ (count\text{-}list\ ys\ True)$

<proof>

lemma *n-subset-enum-correct-aux2:*

$distinct\ xs \implies n\text{-}subsets\ (set\ xs)\ n \subseteq set\ (map\ set\ (n\text{-}subset\text{-}enum\ xs\ n))$

<proof>

theorem *n-subset-enum-correct:*

$distinct\ xs \implies set\ (map\ set\ (n\text{-}subset\text{-}enum\ xs\ n)) = n\text{-}subsets\ (set\ xs)\ n$

<proof>

6.3.3 Distinctness

theorem *n-subset-enum-distinct-enum:*

$distinct\ xs \implies ys \in set\ (n\text{-}subset\text{-}enum\ xs\ n) \implies distinct\ ys$

<proof>

theorem *n-subset-enum-distinct:* $distinct\ xs \implies distinct\ (n\text{-}subset\text{-}enum\ xs\ n)$

<proof>

6.3.4 Cardinality

Cardinality of *n-subsets* is already shown in *Binomial.n-subsets*.

6.4 Alternative using Multiset permutations

It would be possible to define *n-bool-lists* using *permutations-of-multiset* with the following definition:

fun *n-bool-lists2* :: $nat \Rightarrow nat \Rightarrow bool\ list\ set$ **where**

$n\text{-}bool\text{-}lists2\ n\ x = (if\ n > x\ then\ \{\})$

$else\ permutations\text{-}of\text{-}multiset\ (mset\ (replicate\ n\ True\ @\ replicate\ (x-n)\ False))$

6.5 *mset-count*

Correspondence between *count-list* and *count* (*mset xs*) and transfer of a few results for multisets to lists.

lemma *count-list-count-mset*: $\text{count-list } ys \ T = n \implies \text{count } (\text{mset } ys) \ T = n$
<proof>

lemma *count-mset-count-list*: $\text{count } (\text{mset } ys) \ T = n \implies \text{count-list } ys \ T = n$
<proof>

lemma *count-mset-replicate-aux1*:
[[$\neg x < n$; $\text{mset } ys = \text{mset } (\text{replicate } n \ \text{True}) + \text{mset } (\text{replicate } (x - n) \ \text{False})$]]
 $\implies \text{count } (\text{mset } ys) \ \text{True} = n$
<proof>

lemma *count-mset-replicate-aux2*:
assumes $\neg \text{length } xs < \text{count-list } xs \ \text{True}$
shows $\text{mset } xs = \text{mset } (\text{replicate } (\text{count-list } xs \ \text{True}) \ \text{True}) + \text{mset } (\text{replicate } (\text{length } xs - \text{count-list } xs \ \text{True}) \ \text{False})$
<proof>

lemma *n-bool-lists2-correct*: $\text{set } (n\text{-bool-lists } n \ x) = n\text{-bool-lists2 } n \ x$
<proof>

end

7 Powerset

theory *Powerset*
imports
 Main
 n-Sequences
 Common-Lemmas
 Filter-Bool-List
begin

7.1 Definition

$\text{Pow } A$

Cardinality: $2^{\text{card } A}$

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

7.2 Algorithm

fun *all-bool-lists* :: $\text{nat} \Rightarrow \text{bool list list}$ **where**
 all-bool-lists 0 = [[]]

| $all\text{-}bool\text{-}lists (Suc\ x) = concat\ [[False\#\ xs,\ True\#\ xs] .\ xs \leftarrow all\text{-}bool\text{-}lists\ x]$

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

7.3 Verification

First we show the relevant theorems for *all-bool-lists*, then we'll transfer the results to the enumeration algorithm for powersets.

lemma *distinct-concat-aux*: $distinct\ xs \implies distinct\ (concat\ (map\ (\lambda xs.\ [False\ \#\ xs,\ True\ \#\ xs])\ xs))$
 ⟨*proof*⟩

lemma *distinct-all-bool-lists* : $distinct\ (all\text{-}bool\text{-}lists\ x)$
 ⟨*proof*⟩

lemma *all-bool-lists-correct*: $set\ (all\text{-}bool\text{-}lists\ x) = \{xs.\ length\ xs = x\}$
 ⟨*proof*⟩

7.3.1 Correctness

theorem *powerset-enum-correct*: $set\ (map\ set\ (powerset\text{-}enum\ xs)) = Pow\ (set\ xs)$
 ⟨*proof*⟩

7.3.2 Distinctness

theorem *powerset-enum-distinct-elem*: $distinct\ xs \implies ys \in set\ (powerset\text{-}enum\ xs) \implies distinct\ ys$
 ⟨*proof*⟩

theorem *powerset-enum-distinct*: $distinct\ xs \implies distinct\ (powerset\text{-}enum\ xs)$
 ⟨*proof*⟩

7.3.3 Cardinality

Cardinality for powersets is already shown in *card-Pow*.

7.4 Alternative algorithm with *n-sequence-enum*

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

lemma *all-bool-lists2-distinct*: $distinct\ (all\text{-}bool\text{-}lists2\ n)$
 ⟨*proof*⟩

lemma *all-bool-lists2-correct*: $set\ (all\text{-}bool\text{-}lists\ n) = set\ (all\text{-}bool\text{-}lists2\ n)$
 ⟨*proof*⟩

end

8 Integer Partitions

```
theory Integer-Partitions
imports
  HOL-Library.Multiset
  Common-Lemmas
  Card-Number-Partitions.Card-Number-Partitions
begin
```

8.1 Definition

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

Cardinality: *Partition* i (from *Card-Number-Partitions.Card-Number-Partitions* [2])

Example: *integer-partitions* 4 = $\{\{4\}, \{3,1\}, \{2,2\}, \{2,1,1\}, \{1,1,1,1\}\}$

8.2 Algorithm

```
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
```

8.3 Verification

8.3.1 Correctness

```
lemma integer-partitions-empty: [] ∈ set (integer-partitions-enum-aux n m) ⇒ n
= 0
⟨proof⟩
```

```
lemma integer-partitions-enum-aux-first:
  x # xs ∈ set (integer-partitions-enum-aux n m)
  ⇒ xs ∈ set (integer-partitions-enum-aux (n-x) x)
⟨proof⟩
```

```
lemma integer-partitions-enum-aux-max-n:
  x#xs ∈ set (integer-partitions-enum-aux n m) ⇒ x ≤ n
⟨proof⟩
```

```
lemma integer-partitions-enum-aux-max-head:
  x#xs ∈ set (integer-partitions-enum-aux n m) ⇒ x ≤ m
⟨proof⟩
```


lemma *integer-partitions-enum-aux-max:*

$xs \in \text{set } (\text{integer-partitions-enum-aux } n \ m) \implies x \in \text{set } xs \implies x \leq m$
(proof)

lemma *integer-partitions-enum-aux-sum:*

$xs \in \text{set } (\text{integer-partitions-enum-aux } n \ m) \implies \text{sum-list } xs = n$
(proof)

lemma *integer-partitions-enum-aux-not-null-aux:*

$x\#xs \in \text{set } (\text{integer-partitions-enum-aux } n \ m) \implies x \neq 0$
(proof)

lemma *integer-partitions-enum-aux-not-null:*

$x \in \text{set } (\text{integer-partitions-enum-aux } n \ m) \implies x \in \text{set } xs \implies x \neq 0$
(proof)

lemma *integer-partitions-enum-aux-head-minus:*

$h \leq m \implies h > 0 \implies n \geq h \implies$
 $ys \in \text{set } (\text{integer-partitions-enum-aux } (n-h) \ h) \implies h\#ys \in \text{set } (\text{integer-partitions-enum-aux } n \ m)$
(proof)

lemma *integer-partitions-enum-aux-head-plus:*

$h \leq m \implies h > 0 \implies ys \in \text{set } (\text{integer-partitions-enum-aux } n \ h)$
 $\implies h\#ys \in \text{set } (\text{integer-partitions-enum-aux } (h+n) \ m)$
(proof)

lemma *integer-partitions-enum-correct-aux1:*

assumes $0 \notin \# A$

and $\forall x \in \# A. x \leq m$

shows $\exists xs \in \text{set } (\text{integer-partitions-enum-aux } (\sum \# A) \ m). A = \text{mset } xs$
(proof)

theorem *integer-partitions-enum-correct:*

$\text{set } (\text{map } \text{mset } (\text{integer-partitions-enum } n)) = \text{integer-partitions } n$
(proof)

8.3.2 Distinctness

lemma *integer-partitions-enum-aux-distinct:*

$\text{distinct } (\text{integer-partitions-enum-aux } n \ m)$
(proof)

theorem *integer-partitions-enum-distinct:*

$\text{distinct } (\text{integer-partitions-enum } n)$
(proof)

8.3.3 Cardinality

lemma *partitions-bij-betw-count*:

bij-betw count {N. count N partitions n} {p. p partitions n}
<proof>

lemma *card-partitions-count-partitions*:

card {p. p partitions n} = card {N. count N partitions n}
<proof>

this sadly is not proven in *Card-Number-Partitions.Card-Number-Partitions*

lemma *card-partitions-number-partition*:

card {p. p partitions n} = card {N. number-partition n N}
<proof>

lemma *integer-partitions-number-partition-eq*:

integer-partitions n = {N. number-partition n N}
<proof>

lemma *integer-partitions-cardinality-aux*:

card (integer-partitions n) = (∑ k≤n. Partition n k)
<proof>

theorem *integer-partitions-cardinality*:

*card (integer-partitions n) = Partition (2*n) n*
<proof>

end

9 Integer Compositions

theory *Integer-Compositions*

imports

Common-Lemmas

begin

9.1 Definition

definition *integer-compositions* :: *nat* ⇒ *nat list set* **where**

integer-compositions i = {xs. sum-list xs = i ∧ 0 ∉ set xs}

Integer compositions are *integer-partitions* where the order matters.

Cardinality: *sum from n = 1 to i (binomial (i-1) (n-1)) = 2ⁱ⁻¹*

Example: *integer-compositions 3 = {[3], [2,1], [1,2], [1,1,1]}*

9.2 Algorithm

fun *integer-composition-enum* :: *nat* ⇒ *nat list list* **where**

$integer-composition-enum\ 0 = []$
 $| integer-composition-enum\ (Suc\ n) =$
 $\quad [Suc\ m\ \#\ xs.\ m \leftarrow [0..< Suc\ n],\ xs \leftarrow integer-composition-enum\ (n-m)]$

9.3 Verification

9.3.1 Correctness

lemma *integer-composition-enum-tail-elem*:

$x\#\ xs \in set\ (integer-composition-enum\ n) \implies xs \in set\ (integer-composition-enum\ (n - x))$
 $\langle proof \rangle$

lemma *integer-composition-enum-not-null-aux*:

$x\#\ xs \in set\ (integer-composition-enum\ n) \implies x \neq 0$
 $\langle proof \rangle$

lemma *integer-composition-enum-not-null*: $xs \in set\ (integer-composition-enum\ n)$

$\implies 0 \notin set\ xs$
 $\langle proof \rangle$

lemma *integer-composition-enum-empty*: $[] \in set\ (integer-composition-enum\ n)$

$\implies n = 0$
 $\langle proof \rangle$

lemma *integer-composition-enum-sum*: $xs \in set\ (integer-composition-enum\ n) \implies$

$sum-list\ xs = n$
 $\langle proof \rangle$

lemma *integer-composition-enum-head-set*:

assumes $x \neq 0$ **and** $x \leq n$

shows $xs \in set\ (integer-composition-enum\ (n-x)) \implies x\#\ xs \in set\ (integer-composition-enum\ n)$

$\langle proof \rangle$

lemma *integer-composition-enum-correct-aux*:

$0 \notin set\ xs \implies xs \in set\ (integer-composition-enum\ (sum-list\ xs))$
 $\langle proof \rangle$

theorem *integer-composition-enum-correct*:

$set\ (integer-composition-enum\ n) = integer-compositions\ n$
 $\langle proof \rangle$

9.3.2 Distinctness

theorem *integer-composition-enum-distinct*:

$distinct\ (integer-composition-enum\ n)$
 $\langle proof \rangle$

9.3.3 Cardinality

lemma *sum-list-two-pow-aux*:

$(\sum x \leftarrow [0..<n]. (2::nat) ^ (n - x)) + 2 ^ (0 - 1) + 2 ^ 0 = 2 ^ (Suc\ n)$
 $\langle proof \rangle$

lemma *sum-list-two-pow*:

$Suc\ (\sum x \leftarrow [0..<n]. 2 ^ (n - Suc\ x)) = 2 ^ n$
 $\langle proof \rangle$

lemma *integer-composition-enum-length*:

$length\ (integer-composition-enum\ n) = 2 ^ (n-1)$
 $\langle proof \rangle$

theorem *integer-compositions-card*:

$card\ (integer-compositions\ n) = 2 ^ (n-1)$
 $\langle proof \rangle$

end

10 Weak Integer Compositions

theory *Weak-Integer-Compositions*

imports

HOL-Combinatorics.Multiset-Permutations

Common-Lemmas

begin

10.1 Definition

definition *weak-integer-compositions* :: $nat \Rightarrow nat \Rightarrow nat\ list\ set$ **where**

$weak-integer-compositions\ i\ l = \{xs.\ length\ xs = l \wedge sum-list\ xs = i\}$

Weak integer compositions are similar to integer compositions, with the trade-off that 0 is allowed but the composition must have a fixed length.

Cardinality: $binomial\ (i + n - 1)\ i$

Example: $weak-integer-compositions\ 2\ 2 = \{[2,0], [1,1], [0,2]\}$

10.2 Algorithm

fun *weak-integer-composition-enum* :: $nat \Rightarrow nat \Rightarrow nat\ list\ list$ **where**

$weak-integer-composition-enum\ i\ 0 = (if\ i = 0\ then\ [[]]\ else\ [])$

$| weak-integer-composition-enum\ i\ (Suc\ 0) = [[i]]$

$| weak-integer-composition-enum\ i\ l =$

$[h\#r . h \leftarrow [0..<Suc\ i], r \leftarrow weak-integer-composition-enum\ (i-h)\ (l-1)]$

10.3 Verification

10.3.1 Correctness

lemma *weak-integer-composition-enum-length*:

$xs \in \text{set } (\text{weak-integer-composition-enum } i \ l) \implies \text{length } xs = l$
{proof}

lemma *weak-integer-composition-enum-sum-list*:

$xs \in \text{set } (\text{weak-integer-composition-enum } i \ l) \implies \text{sum-list } xs = i$
{proof}

lemma *weak-integer-composition-enum-head*:

assumes $xs \in \text{set } (\text{weak-integer-composition-enum } (\text{sum-list } xs) \ (\text{length } xs))$
shows $x \# xs \in \text{set } (\text{weak-integer-composition-enum } (x + \text{sum-list } xs) \ (\text{Suc } (\text{length } xs)))$
{proof}

lemma *weak-integer-composition-enum-correct-aux*:

$xs \in \text{set } (\text{weak-integer-composition-enum } (\text{sum-list } xs) \ (\text{length } xs))$
{proof}

theorem *weak-integer-composition-enum-correct*:

$\text{set } (\text{weak-integer-composition-enum } i \ l) = \text{weak-integer-compositions } i \ l$
{proof}

10.3.2 Distinctness

theorem *weak-integer-composition-enum-distinct*: $\text{distinct } (\text{weak-integer-composition-enum } i \ l)$

{proof}

10.3.3 Cardinality

The following is a generalization of the binomial coefficient to multisets. Sometimes it is called multiset coefficient. Here we call it "multichoose" [4].

definition *multichoose*:: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ (**infixl** *multichoose* 65) **where**

$n \ \text{multichoose } k = (n + k - 1) \ \text{choose } k$

lemma *weak-integer-composition-enum-zero*: $\text{length } (\text{weak-integer-composition-enum } 0 \ (\text{Suc } n)) = 1$

{proof}

lemma *a-choose-equivalence*: $\text{Suc } (\sum x \leftarrow [0..<k]. n + (k - x) \ \text{choose } (k - x)) = \text{Suc } (n + k) \ \text{choose } k$

{proof}

lemma *composition-enum-length*: $\text{length } (\text{weak-integer-composition-enum } i \ n) = n \ \text{multichoose } i$

{proof}

theorem *weak-integer-compositions-cardinality*: $\text{card } (\text{weak-integer-compositions } n \ k) = k \text{ multichoose } n$
 ⟨proof⟩

end

11 Derangements

theory *Derangements-Enum*

imports

HOL-Combinatorics.Multiset-Permutations

Common-Lemmas

begin

11.1 Definition

fun *no-overlap* :: 'a list ⇒ 'a list ⇒ bool **where**
no-overlap - [] = True
 | *no-overlap* [] - = True
 | *no-overlap* (x#xs) (y#ys) = (x ≠ y ∧ *no-overlap* xs ys)

lemma *no-overlap-nth*: $\text{length } xs = \text{length } ys \implies i < \text{length } xs \implies \text{no-overlap } xs \ ys \implies xs ! i \neq ys ! i$
 ⟨proof⟩

lemma *nth-no-overlap*: $\text{length } xs = \text{length } ys \implies \forall i < \text{length } xs. xs ! i \neq ys ! i \implies \text{no-overlap } xs \ ys$
 ⟨proof⟩

definition *derangements* :: 'a list ⇒ 'a list set **where**
derangements xs = {ys. *distinct* ys ∧ $\text{length } xs = \text{length } ys$ ∧ $\text{set } xs = \text{set } ys$ ∧ *no-overlap* xs ys }

A derangement of a list is a permutation where every element changes its position, assuming all elements are distinguishable.

An alternative definition exists in *Derangements.Derangements* [1].

Cardinality: *count-derangements* ($\text{length } xs$) (from *Derangements.Derangements*)

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

11.2 Algorithm

fun *derangement-enum-aux* :: 'a list ⇒ 'a list ⇒ 'a list list **where**
derangement-enum-aux [] ys = [[]]
 | *derangement-enum-aux* (x#xs) ys = [y#r . y ← ys, r ← *derangement-enum-aux* xs (remove1 y ys), y ≠ x]

fun *derangement-enum* :: 'a list \Rightarrow 'a list list **where**
derangement-enum *xs* = *derangement-enum-aux* *xs xs*

11.3 Verification

11.3.1 Correctness

lemma *derangement-enum-aux-elem-length*: *zs* \in *set* (*derangement-enum-aux* *xs* *ys*) \Longrightarrow *length* *xs* = *length* *zs*
 <proof>

lemma *derangement-enum-aux-not-in*: *y* \notin *set* *ys* \Longrightarrow *zs* \in *set* (*derangement-enum-aux* *xs* *ys*) \Longrightarrow *y* \notin *set* *zs*
 <proof>

lemma *derangement-enum-aux-in*: *y* \in *set* *zs* \Longrightarrow *zs* \in *set* (*derangement-enum-aux* *xs* *ys*) \Longrightarrow *y* \in *set* *ys*
 <proof>

lemma *derangement-enum-aux-distinct-elem*: *distinct* *ys* \Longrightarrow *zs* \in *set* (*derangement-enum-aux* *xs* *ys*) \Longrightarrow *distinct* *zs*
 <proof>

lemma *derangement-enum-aux-no-overlap*: *zs* \in *set* (*derangement-enum-aux* *xs* *ys*) \Longrightarrow *no-overlap* *xs* *zs*
 <proof>

lemma *derangement-enum-aux-set*:
length *xs* = *length* *ys* \Longrightarrow *zs* \in *set* (*derangement-enum-aux* *xs* *ys*) \Longrightarrow *set* *zs* = *set* *ys*
 <proof>

lemma *derangement-enum-correct-aux1*:
 \llbracket *distinct* *zs*; *length* *ys* = *length* *zs*; *length* *ys* = *length* *xs*; *set* *ys* = *set* *zs*; *no-overlap* *xs* *zs* \rrbracket
 \Longrightarrow *zs* \in *set* (*derangement-enum-aux* *xs* *ys*)
 <proof>

theorem *derangement-enum-correct*: *distinct* *xs* \Longrightarrow *derangements* *xs* = *set* (*derangement-enum* *xs*)
 <proof>

11.3.2 Distinctness

lemma *derangement-enum-aux-distinct*: *distinct* *ys* \Longrightarrow *distinct* (*derangement-enum-aux* *xs* *ys*)
 <proof>

theorem *derangement-enum-distinct*: $distinct\ xs \implies distinct\ (derangement-enum\ xs)$
 ⟨*proof*⟩

end

12 Trees

theory *Trees*
imports
 HOL-Library.Tree
 Common-Lemmas

begin

12.1 Definition

The set of trees can be defined with the pre-existing *tree* datatype:

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

Cardinality: *Catalan number of n*

Example: *trees* 0 = {*Leaf*}

12.2 Algorithm

fun *tree-enum* :: $nat \Rightarrow unit\ tree\ list$ **where**
tree-enum 0 = [*Leaf*] |
tree-enum (*Suc* n) = [(*t1*, (), *t2*). $i \leftarrow [0..<Suc\ n]$, $t1 \leftarrow tree-enum\ i$, $t2 \leftarrow tree-enum\ (n-i)$]

12.3 Verification

12.3.1 Cardinality

lemma *length-tree-enum*:
 $length\ (tree-enum\ (Suc\ n)) = (\sum\ i \leq n.\ length\ (tree-enum\ i) * length\ (tree-enum\ (n - i)))$
 ⟨*proof*⟩

12.3.2 Correctness

lemma *tree-enum-correct1*: $t \in set\ (tree-enum\ n) \implies size\ t = n$
 ⟨*proof*⟩

lemma *tree-enum-correct2*: $n = size\ t \implies t \in set\ (tree-enum\ n)$

<proof>

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

<proof>

12.3.3 Distinctness

lemma *tree-enum-Leaf*: $\langle \rangle \in set\ (tree-enum\ n) \iff (n = 0)$

<proof>

lemma *tree-enum-elem-injective*: $n \neq m \implies x \in set\ (tree-enum\ n) \implies y \in set\ (tree-enum\ m) \implies x \neq y$

<proof>

lemma *tree-enum-elem-injective2*: $x \in set\ (tree-enum\ n) \implies y \in set\ (tree-enum\ m) \implies x = y \implies n = m$

<proof>

lemma *concat-map-Node-not-equal*:

$xs \neq [] \implies xs2 \neq [] \implies ys \neq [] \implies ys2 \neq [] \implies$

$\forall x \in set\ xs. \forall y \in set\ ys. x \neq y \implies$

$[(l, (), r). l \leftarrow xs2, r \leftarrow xs] \neq [(l, (), r). l \leftarrow ys2, r \leftarrow ys]$

<proof>

lemma *tree-enum-not-empty*: $tree-enum\ n \neq []$

<proof>

lemma *tree-enum-distinct-aux-outer*:

assumes $\forall i \leq n. distinct\ (tree-enum\ i)$

and *distinct xs*

and $\forall i \in set\ xs. i < n$

and *sorted-wrt (<) xs*

shows $distinct\ (map\ (\lambda i. [(l, (), r). l \leftarrow tree-enum\ i, r \leftarrow tree-enum\ (n-i)])\ xs)$

<proof>

lemma *tree-enum-distinct-aux-left*:

$\forall i < n. distinct\ (tree-enum\ i) \implies distinct\ ([[(l, (), r). i \leftarrow [0..< n], l \leftarrow tree-enum\ i]])$

<proof>

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

<proof>

end

theory *Combinatorial-Enumeration-Algorithms*

imports

n-Sequences

n-Permutations

n-Subsets

Powerset

Integer-Partitions
Integer-Compositions
Weak-Integer-Compositions
Derangements-Enum
Trees
begin

end

References

- [1] L. Bulwahn. Derangements formula. *Archive of Formal Proofs*, June 2015. <https://isa-afp.org/entries/Derangements.html>, Formal proof development.
- [2] L. Bulwahn. Cardinality of number partitions. *Archive of Formal Proofs*, January 2016. https://isa-afp.org/entries/Card_Number_Partitions.html, Formal proof development.
- [3] L. Bulwahn. The twelvefold way. *Archive of Formal Proofs*, December 2016. https://isa-afp.org/entries/Twelvefold_Way.html, Formal proof development.
- [4] R. Stanley. *Enumerative Combinatorics: Volume 1*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2011.
- [5] D. Stanton and D. White. *Constructive Combinatorics*. Springer, 1986.