Combinatorial Enumeration Algorithms

Paul Hofmeier and Emin Karayel

March 29, 2023

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 Twelvefold Way [3].

Contents

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-on* and *inj-on*

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

lemma *inj2-on-all*: $inj2\ f \Longrightarrow inj2\text{-}on\ f\ A\ B$
⟨*proof*⟩

lemma *inj2-inj-first*: $inj2\ f \Longrightarrow inj\ f$
⟨*proof*⟩

lemma *inj2-inj-second*: $inj2\ f \Longrightarrow inj\ (f\ x)$
⟨*proof*⟩

lemma *inj2-inj-second-flipped*: $inj2\ f \Longrightarrow inj\ (\lambda x.\ f\ x\ y)$
⟨*proof*⟩

1.2 Proofs with inj2

Already existing for *inj*:

thm *distinct-map*

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

lemma *inj2-distinct-map*:
assumes $inj2\ f$
shows $distinct\ xs = distinct\ (map\ (f\ x)\ xs)$
⟨*proof*⟩

lemma *inj2-on-distinct-concat-map*:
assumes $inj2\text{-}on\ f\ (set\ xs)\ (set\ ys)$
shows $[\![distinct\ ys;\ distinct\ xs]\!] \Longrightarrow distinct\ [f\ x\ y.\ x \leftarrow xs,\ y \leftarrow ys]$
⟨*proof*⟩

lemma *inj2-distinct-concat-map*:
assumes $inj2\ f$
shows $[\![distinct\ ys;\ distinct\ xs]\!] \Longrightarrow distinct\ [f\ x\ y.\ x \leftarrow xs,\ y \leftarrow ys]$
⟨*proof*⟩

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

assumes *inj2 f*
shows$[\![\forall\ x \in set\ xs.\ distinct\ (g\ x);\ distinct\ xs]\!] \Longrightarrow distinct\ [f\ x\ y.\ x \leftarrow xs,\ y \leftarrow g\ x]$
⟨*proof*⟩

lemma *distinct-concat-Nil*: $distinct\ (concat\ (map\ (\lambda y.\ [])\ xs))$
⟨*proof*⟩

lemma *inj2-distinct-concat-map-function-filter*:
assumes *inj2 f*
shows$[\![\forall\ x \in set\ xs.\ distinct\ (g\ x);\ distinct\ xs]\!] \Longrightarrow distinct\ [f\ x\ y.\ x \leftarrow xs,\ y \leftarrow g\ x,\ h\ x]$
⟨*proof*⟩

1.3 Specializations of *inj2*

1.3.1 Cons

lemma *Cons-inj2*: *inj2* (#)
⟨*proof*⟩

lemma *Cons-distinct-concat-map*: $[\![distinct\ ys;\ distinct\ xs]\!] \Longrightarrow distinct\ [x\#y.\ x \leftarrow xs,\ y \leftarrow ys]$
⟨*proof*⟩

lemma *Cons-distinct-concat-map-function*:
$[\![\forall\ x \in set\ xs.\ distinct\ (g\ x)\ ;\ distinct\ xs]\!] \Longrightarrow distinct\ [x\ \#\ y.\ x \leftarrow xs,\ y \leftarrow g\ x]$
⟨*proof*⟩

lemma *Cons-distinct-concat-map-function-distinct-on-all*:
$[\![\forall\ x.\ distinct\ (g\ x)\ ;\ distinct\ xs]\!] \Longrightarrow distinct\ [x\ \#\ y.\ x \leftarrow xs,\ y \leftarrow g\ x]$
⟨*proof*⟩

1.3.2 Node right

lemma *Node-right-inj2*: $inj2\ (\lambda l\ r.\ Node\ l\ e\ r)$
⟨*proof*⟩

lemma *Node-right-distinct-concat-map*:
$[\![distinct\ ys;\ distinct\ xs]\!] \Longrightarrow distinct\ [Node\ x\ e\ y.\ x \leftarrow xs,\ y \leftarrow ys]$
⟨*proof*⟩

1.3.3 Node left

lemma *Node-left-inj2*: $inj2\ (\lambda r\ l.\ Node\ l\ e\ r)$
⟨*proof*⟩

lemma *Node-left-distinct-map*: $distinct\ xs = distinct\ (map\ (\lambda l.\ \langle l,\ (),\ r\rangle)\ xs)$
⟨*proof*⟩

1.3.4 Cons Suc

lemma *Cons-Suc-inj2*: *inj2* (λx *ys. Suc x # ys*)
⟨*proof*⟩

lemma *Cons-Suc-distinct-concat-map-function*:
$[\![\forall\ x \in set\ xs.\ distinct\ (g\ x)\ ;\ distinct\ xs]\!] \Longrightarrow distinct\ [Suc\ x\ \#\ y.\ x \leftarrow xs,\ y \leftarrow g\ x]$
⟨*proof*⟩

2 Lemmas for cardinality proofs

lemma *length-concat-map*: $length\ [f\ x\ r\ .\ x \leftarrow xs,\ r \leftarrow ys] = length\ ys * length\ xs$
⟨*proof*⟩

An useful extension to *length-concat*

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

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

lemma *leq-sum-to-sum-list*: $(\sum x \leq n.\ f\ x) = (\sum x{\leftarrow}[0..{<}Suc\ n].\ f\ x)$
⟨*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*: $x \in set\ xs \Longrightarrow Suc\ (length\ (remove1\ x\ xs)) = length\ xs$
⟨*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*: $count\text{-}list\ (replicate\ x\ y)\ y = x$
⟨*proof*⟩

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

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

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

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

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

end

4 N-Sequences

theory *n-Sequences*
imports
HOL.List
Common-Lemmas
begin

4.1 Definition

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

Cardinality: $card\ A\ \widehat{}\ n$

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

4.2 Algorithm

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

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 $set\ (n\text{-}sequence\text{-}enum\ xs\ n) = set\ (List.n\text{-}lists\ n\ xs)$
⟨*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 ⟹ *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* ⇒ *nat* ⇒ *'a list set* **where**
  n-permutations A n = {*xs*. *set xs* ⊆ *A* ∧ *distinct xs* ∧ *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* ⊆ *n-permutations A* (*card A*)
⟨*proof*⟩

5.2 Algorithm

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*]

5.3 Verification

5.3.1 Correctness

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

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

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

lemma *n-permutation-enum-correct1*: *distinct xs* ⟹ *set* (*n-permutation-enum xs n*) ⊆ *n-permutations* (*set xs*) *n*
⟨*proof*⟩

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

theorem *n-permutation-enum-correct*: *distinct xs* ⟹ *set* (*n-permutation-enum xs n*) = *n-permutations* (*set xs*) *n*
⟨*proof*⟩

5.3.2 Distinctness

theorem *n-permutation-distinct*: *distinct xs* ⟹ *distinct* (*n-permutation-enum xs n*)
⟨*proof*⟩

5.3.3 Cardinality

thm *card-lists-distinct-length-eq*
theorem *finite A* ⟹ *card* (*n-permutations A n*) = *ffact n* (*card A*)

⟨*proof*⟩

5.4 *n-multiset* extension (with remdups)

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

fun *n-multiset-permutation-enum* :: *'a list* ⇒ *nat* ⇒ *'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 ⊆# *mset xs* ⟹ *ys* ∈ *set* (*n-permutation-enum xs* (*length ys*))
⟨*proof*⟩

lemma *n-multiset-permutation-enum-correct2*:
ys ∈ *set* (*n-permutation-enum xs n*) ⟹ *mset ys* ⊆# *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* ⇒ *'a list* ⇒ *'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* ⟹ *n* < *length bs* ⟹ *bs*!*n* ⟹ *xs*!*n* ∈ *set* (*filter-bool-list bs xs*)
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *filter-bool-list-card*:
$[\![distinct\ xs;\ length\ xs = length\ bs]\!] \Longrightarrow card\ (set\ (filter\text{-}bool\text{-}list\ bs\ xs)) = count\text{-}list\ bs\ True$
⟨*proof*⟩

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

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

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

lemma *filter-bool-list-inj*:
$distinct\ xs \Longrightarrow inj\text{-}on\ (\lambda bs.\ filter\text{-}bool\text{-}list\ bs\ xs)\ \{bs.\ length\ bs = length\ xs\}$
⟨*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* ⇒ *nat* ⇒ *'a set set* **where**
  n-subsets A n = {*B*. *B* ⊆ *A* ∧ *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* ⇒ *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 = [(*filter-bool-list bs xs*) . *bs* ← (*n-bool-lists n* (*length xs*))]

6.3 Verification

6.3.1 n-bool-lists

lemma *n-bool-lists-True-count*: *xs* ∈ *set* (*n-bool-lists n x*) ⟹ *count-list xs True* = *n*
 ⟨*proof*⟩

lemma *n-bool-lists-length*: *xs* ∈ *set* (*n-bool-lists n x*) ⟹ *length xs* = *x*
  ⟨*proof*⟩

lemma *n-bool-lists-distinct*: *distinct* (*n-bool-lists n x*)
⟨*proof*⟩

lemma *replicate-True-not-False*: *count-list ys True* = *0* ⟷ *ys* = *replicate* (*length ys*) *False*

⟨*proof*⟩

lemma *n-bool-lists-correct-aux*:
length xs = *x* ⟹ *count-list xs True* = *n* ⟹ *xs* ∈ *set* (*n-bool-lists n x*)
⟨*proof*⟩

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

6.3.2 Correctness

lemma *n-subset-enum-correct-aux1*:
⟦*distinct xs*; *length ys* = *length xs*⟧
⟹ *set* (*filter-bool-list ys xs*) ∈ *n-subsets* (*set xs*) (*count-list ys True*)
⟨*proof*⟩

lemma *n-subset-enum-correct-aux2*:
distinct xs ⟹ *n-subsets* (*set xs*) *n* ⊆ *set* (*map set* (*n-subset-enum xs n*))
⟨*proof*⟩

theorem *n-subset-enum-correct*:
distinct xs ⟹ *set* (*map set* (*n-subset-enum xs n*)) = *n-subsets* (*set xs*) *n*
⟨*proof*⟩

6.3.3 Distinctness

theorem *n-subset-enum-distinct-elem*:
distinct xs ⟹ *ys* ∈ *set* (*n-subset-enum xs n*) ⟹ *distinct ys*
⟨*proof*⟩

theorem *n-subset-enum-distinct*: *distinct xs* ⟹ *distinct* (*n-subset-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* ⇒ *nat* ⇒ *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*)))

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*: *count-list ys T* = *n* ⟹ *count* (*mset ys*) *T* = *n*
⟨*proof*⟩

lemma *count-mset-count-list*: *count* (*mset ys*) *T* = *n* ⟹ *count-list ys T* = *n*
⟨*proof*⟩

lemma *count-mset-replicate-aux1*:
⟦¬ *x* < *n*; *mset ys* = *mset* (*replicate n True*) + *mset* (*replicate* (*x* − *n*) *False*)⟧
⟹ *count* (*mset ys*) *True* = *n*
⟨*proof*⟩

lemma *count-mset-replicate-aux2*:
assumes ¬ *length xs* < *count-list xs True*
shows *mset xs* = *mset* (*replicate* (*count-list xs True*) *True*) + *mset* (*replicate* (*length xs* − *count-list xs True*) *False*)
⟨*proof*⟩

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

end

7 Powerset

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

7.1 Definition

Pow A

Cardinality: *2* ^ *card A*

Example: *Pow* {*0*,*1*} = {{}, {*1*}, {*0*}, {*0*, *1*}}

7.2 Algorithm

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 = [(*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* ⟹ *distinct* (*concat* (*map* (λ*xs*. [*False* # *xs*, *True* # *xs*]) *xs*))
⟨*proof*⟩

lemma *distinct-all-bool-lists* : *distinct* (*all-bool-lists x*)
⟨*proof*⟩

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

7.3.1 Correctness

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

7.3.2 Distinctness

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

theorem *powerset-enum-distinct*: *distinct xs* ⟹ *distinct* (*powerset-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* ⇒ *bool list list* **where**
all-bool-lists2 n = *n-sequence-enum* [*True*, *False*] *n*

lemma *all-bool-lists2-distinct*: *distinct* (*all-bool-lists2 n*)
⟨*proof*⟩

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

end

8 Integer Paritions

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*:
$xs \in set\ (integer\text{-}partitions\text{-}enum\text{-}aux\ n\ m) \Longrightarrow x \in set\ xs \Longrightarrow x \leq m$
⟨*proof*⟩

lemma *integer-partitions-enum-aux-sum*:
$xs \in set\ (integer\text{-}partitions\text{-}enum\text{-}aux\ n\ m) \Longrightarrow sum\text{-}list\ xs = n$
⟨*proof*⟩

lemma *integer-partitions-enum-aux-not-null-aux*:
$x\#xs \in set\ (integer\text{-}partitions\text{-}enum\text{-}aux\ n\ m) \Longrightarrow x \neq 0$
⟨*proof*⟩

lemma *integer-partitions-enum-aux-not-null*:
$xs \in set\ (integer\text{-}partitions\text{-}enum\text{-}aux\ n\ m) \Longrightarrow x \in set\ xs \Longrightarrow x \neq 0$
⟨*proof*⟩

lemma *integer-partitions-enum-aux-head-minus*:
$h \leq m \Longrightarrow h > 0 \Longrightarrow n \geq h \Longrightarrow$
$ys \in set\ (integer\text{-}partitions\text{-}enum\text{-}aux\ (n-h)\ h) \Longrightarrow h\#ys \in set\ (integer\text{-}partitions\text{-}enum\text{-}aux\ n\ m)$
⟨*proof*⟩

lemma *integer-partitions-enum-aux-head-plus*:
$h \leq m \Longrightarrow h > 0 \Longrightarrow ys \in set\ (integer\text{-}partitions\text{-}enum\text{-}aux\ n\ h)$
$\Longrightarrow h\#ys \in set\ (integer\text{-}partitions\text{-}enum\text{-}aux\ (h + n)\ m)$
⟨*proof*⟩

lemma *integer-partitions-enum-correct-aux1*:
assumes $0 \notin\# A$
and $\forall x \in\# A.\ x \leq m$
shows $\exists xs \in set\ (integer\text{-}partitions\text{-}enum\text{-}aux\ (\sum_{\#} A)\ m).\ A = mset\ xs$
⟨*proof*⟩

theorem *integer-partitions-enum-correct*:
$set\ (map\ mset\ (integer\text{-}partitions\text{-}enum\ n)) = integer\text{-}partitions\ n$
⟨*proof*⟩

8.3.2 Distinctness

lemma *integer-partitions-enum-aux-distinct*:
$distinct\ (integer\text{-}partitions\text{-}enum\text{-}aux\ n\ m)$
⟨*proof*⟩

theorem *integer-partitions-enum-distinct*:
$distinct\ (integer\text{-}partitions\text{-}enum\ n)$
⟨*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*) = ($\sum$ *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*)) = *2*^(*i*−*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*) =
[*Suc m #xs. m* ← [*0*..< *Suc n*], *xs* ← *integer-composition-enum* (*n*−*m*)]

9.3 Verification

9.3.1 Correctness

lemma *integer-composition-enum-tail-elem*:
x#xs ∈ *set* (*integer-composition-enum n*) ⟹ *xs* ∈ *set* (*integer-composition-enum* (*n* − *x*))
⟨*proof*⟩

lemma *integer-composition-enum-not-null-aux*:
x#xs ∈ *set* (*integer-composition-enum n*) ⟹ *x* ≠ *0*
⟨*proof*⟩

lemma *integer-composition-enum-not-null*: *xs* ∈ *set* (*integer-composition-enum n*) ⟹ *0* ∉ *set xs*
⟨*proof*⟩

lemma *integer-composition-enum-empty*: [] ∈ *set* (*integer-composition-enum n*) ⟹ *n* = *0*
⟨*proof*⟩

lemma *integer-composition-enum-sum*: *xs* ∈ *set* (*integer-composition-enum n*) ⟹ *sum-list xs* = *n*
⟨*proof*⟩

lemma *integer-composition-enum-head-set*:
assumes*x* ≠ *0* **and** *x* ≤ *n*
shows *xs* ∈ *set* (*integer-composition-enum* (*n*−*x*)) ⟹ *x#xs* ∈ *set* (*integer-composition-enum n*)
⟨*proof*⟩

lemma *integer-composition-enum-correct-aux*:
0 ∉ *set xs* ⟹ *xs* ∈ *set* (*integer-composition-enum* (*sum-list xs*))
⟨*proof*⟩

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

9.3.2 Distinctness

theorem *integer-composition-enum-distinct*:
distinct (*integer-composition-enum n*)
⟨*proof*⟩

9.3.3 Cardinality

lemma *sum-list-two-pow-aux*:
$(\sum x \leftarrow [0..< n].\ (2::nat)\ \hat{}\ (n - x)) + 2\ \hat{}\ (0 - 1) + 2\ \hat{}\ 0 = 2\ \hat{}\ (Suc\ n)$
⟨*proof*⟩

lemma *sum-list-two-pow*:
$Suc\ (\sum x \leftarrow [0..<n].\ 2\ \hat{}\ (n - Suc\ x)) = 2\ \hat{}\ n$
⟨*proof*⟩

lemma *integer-composition-enum-length*:
length (*integer-composition-enum n*) = $2\hat{}(n-1)$
⟨*proof*⟩

theorem *integer-compositions-card*:
card (*integer-compositions n*) = $2\hat{}(n-1)$
⟨*proof*⟩

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* ⇒ *nat* ⇒ *nat list set* **where**
weak-integer-compositions i l = {*xs. length xs* = *l* ∧ *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* ⇒ *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*)]

10.3 Verification

10.3.1 Correctness

lemma *weak-integer-composition-enum-length*:
xs ∈ *set* (*weak-integer-composition-enum i l*) ⟹ *length xs* = *l*
⟨*proof*⟩

lemma *weak-integer-composition-enum-sum-list*:
xs ∈ *set* (*weak-integer-composition-enum i l*) ⟹ *sum-list xs* = *i*
⟨*proof*⟩

lemma *weak-integer-composition-enum-head*:
assumes *xs* ∈ *set* (*weak-integer-composition-enum* (*sum-list xs*) (*length xs*))
shows *x* # *xs* ∈ *set* (*weak-integer-composition-enum* (*x* + *sum-list xs*) (*Suc* (*length xs*)))
⟨*proof*⟩

lemma *weak-integer-composition-enum-correct-aux*:
xs ∈ *set* (*weak-integer-composition-enum* (*sum-list xs*) (*length xs*))
⟨*proof*⟩

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

10.3.2 Distinctness

theorem *weak-integer-composition-enum-distinct*: *distinct* (*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*:: *nat* ⇒ *nat* ⇒ *nat* (**infixl** *multichoose 65*) **where**
n multichoose k = (*n* + *k* −*1*) *choose k*

lemma *weak-integer-composition-enum-zero*: *length* (*weak-integer-composition-enum 0* (*Suc n*)) = *1*
⟨*proof*⟩

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

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

theorem *weak-integer-compositions-cardinality*: *card* (*weak-integer-compositions n k*) = *k 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*: *length xs* = *length ys* ⟹ *i* < *length xs* ⟹ *no-overlap xs ys* ⟹ *xs* ! *i* ≠ *ys* ! *i*
⟨*proof*⟩

lemma *nth-no-overlap*: *length xs* = *length ys* ⟹ ∀ *i* < *length xs*. *xs* ! *i* ≠ *ys* ! *i* ⟹ *no-overlap xs ys*
⟨*proof*⟩

definition *derangements* :: *'a list* ⇒ *'a list set* **where**
derangements xs = {*ys*. *distinct ys* ∧ *length xs* = *length ys* ∧ *set xs* = *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* (*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* ⇒ *'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*: $zs \in set\ (derangement\text{-}enum\text{-}aux\ xs\ ys) \Longrightarrow length\ xs = length\ zs$
⟨*proof*⟩

lemma *derangement-enum-aux-not-in*: $y \notin set\ ys \Longrightarrow zs \in set\ (derangement\text{-}enum\text{-}aux\ xs\ ys) \Longrightarrow y \notin set\ zs$
⟨*proof*⟩

lemma *derangement-enum-aux-in*: $y \in set\ zs \Longrightarrow zs \in set\ (derangement\text{-}enum\text{-}aux\ xs\ ys) \Longrightarrow y \in set\ ys$
⟨*proof*⟩

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

lemma *derangement-enum-aux-no-overlap*: $zs \in set\ (derangement\text{-}enum\text{-}aux\ xs\ ys) \Longrightarrow no\text{-}overlap\ xs\ zs$
⟨*proof*⟩

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

lemma *derangement-enum-correct-aux1*:
$[\![distinct\ zs; length\ ys = length\ zs;\ length\ ys = length\ xs;\ set\ ys = set\ zs;\ no\text{-}overlap\ xs\ zs]\!]$
$\Longrightarrow zs \in set\ (derangement\text{-}enum\text{-}aux\ xs\ ys)$
⟨*proof*⟩

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

11.3.2 Distinctness

lemma *derangement-enum-aux-distinct*: $distinct\ ys \Longrightarrow distinct\ (derangement\text{-}enum\text{-}aux\ xs\ ys)$
⟨*proof*⟩

theorem *derangement-enum-distinct*: *distinct xs* $\Longrightarrow$ *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* ⇒ *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* ⇒ *unit tree list* **where**
tree-enum 0 = [*Leaf*] |
tree-enum (*Suc n*) = [⟨*t1*, (), *t2*⟩. *i* ← [*0*..<*Suc n*], *t1* ← *tree-enum i*, *t2* ← *tree-enum* (*n*−*i*)]

12.3 Verification

12.3.1 Cardinality

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

12.3.2 Correctness

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

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

⟨*proof*⟩

theorem *tree-enum-correct*: $set(tree\text{-}enum\ n) = trees\ n$
⟨*proof*⟩

12.3.3 Distinctness

lemma *tree-enum-Leaf*: $\langle\rangle \in set\ (tree\text{-}enum\ n) \longleftrightarrow (n = 0)$
⟨*proof*⟩

lemma *tree-enum-elem-injective*: $n \neq m \Longrightarrow x \in set\ (tree\text{-}enum\ n) \Longrightarrow y \in set\ (tree\text{-}enum\ m) \Longrightarrow x \neq y$
⟨*proof*⟩

lemma *tree-enum-elem-injective2*: $x \in set\ (tree\text{-}enum\ n) \Longrightarrow y \in set\ (tree\text{-}enum\ m) \Longrightarrow x = y \Longrightarrow n = m$
⟨*proof*⟩

lemma *concat-map-Node-not-equal*:
$xs \neq [] \Longrightarrow xs2 \neq [] \Longrightarrow ys \neq [] \Longrightarrow ys2 \neq [] \Longrightarrow$
$\forall\ x\in set\ xs.\ \forall\ y \in set\ ys\ .\ x \neq y \Longrightarrow$
$[\langle l, (), r\rangle.\ l \leftarrow xs2,\ r \leftarrow xs] \neq [\langle l, (), r\rangle.\ l \leftarrow ys2,\ r \leftarrow ys]$
⟨*proof*⟩

lemma *tree-enum-not-empty*: $tree\text{-}enum\ n \neq []$
⟨*proof*⟩

lemma *tree-enum-distinct-aux-outer*:
assumes $\forall\, i \leq n.\ distinct\ (tree\text{-}enum\ i)$
and *distinct xs*
and $\forall\ i \in set\ xs.\ i < n$
and *sorted-wrt* (<) *xs*
shows $distinct\ (map\ (\lambda i.\ [\langle l, (), r\rangle.\ l \leftarrow tree\text{-}enum\ i,\ r \leftarrow tree\text{-}enum\ (n-i)])\ xs)$
⟨*proof*⟩

lemma *tree-enum-distinct-aux-left*:
$\forall\ i < n.\ distinct\ (tree\text{-}enum\ i) \Longrightarrow distinct\ ([\langle l, (), r\rangle.\ i \leftarrow [0..< n],\ l \leftarrow tree\text{-}enum\ i])$
⟨*proof*⟩

theorem *tree-enum-distinct*: $distinct(tree\text{-}enum\ n)$
⟨*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.