Combinatorial Enumeration Algorithms

Paul Hofmeier and Emin Karayel

April 18, 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 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-on f A B ⟷ (∀ x1∈A. ∀ x2∈A. ∀ y1∈B. ∀ y2∈B. f x1 y1 = f x2 y2 ⟶ x1
= x2 ∧ y1 = y2)

abbreviation inj2 :: ('a ⇒ 'b ⇒ 'c) ⇒ bool where
  inj2 f ≡ inj2-on f UNIV UNIV
```

1.1 Correspondence between *inj2-on* and *inj-on*

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

lemma inj2-on-all: inj2 f ⟹ inj2-on f A B
  unfolding inj2-on-def by simp

lemma inj2-inj-first: inj2 f ⟹ inj f
  unfolding inj2-on-def inj-on-def by simp

lemma inj2-inj-second: inj2 f ⟹ inj (f x)
  unfolding inj2-on-def inj-on-def by simp

lemma inj2-inj-second-flipped: inj2 f ⟹ inj (λx. f x y)
  unfolding inj2-on-def inj-on-def by simp
```

1.2 Proofs with inj2

Already existing for *inj*:

```
thm distinct-map

lemma inj2-on-distinct-map:
  assumes inj2-on f {x} (set xs)
  shows distinct xs = distinct (map (f x) xs)
  using assms distinct-map by (auto simp: inj2-on-def inj-onI)

lemma inj2-distinct-map:
  assumes inj2 f
  shows distinct xs = distinct (map (f x) xs)
  using assms inj2-on-distinct-map inj2-on-all by fast

lemma inj2-on-distinct-concat-map:
  assumes inj2-on f (set xs) (set ys)
  shows ⟦distinct ys; distinct xs⟧ ⟹ distinct [f x y. x ← xs, y ← ys]
using assms proof(induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then have nin: x ∉ set xs
    by simp
  then have inj2-on f {x} (set ys)
    using Cons unfolding inj2-on-def by simp
  then have 1: distinct (map (f x) ys)
    using Cons inj2-on-distinct-map by fastforce

  have 2: distinct (concat (map (λx. map (f x) ys) xs))
    using Cons unfolding inj2-on-def by simp

  have 3: ⟦xa ∈ set xs; xb ∈ set ys; f x xb = f xa xc; xc ∈ set ys⟧ ⟹ False  for
xa xb xc
    using Cons(4) unfolding inj2-on-def
    using nin by force

  from 1 2 3 show ?case
    by auto
qed

lemma inj2-distinct-concat-map:
  assumes inj2 f
  shows ⟦distinct ys; distinct xs⟧ ⟹ distinct [f x y. x ← xs, y ← ys]
  using assms inj2-on-all inj2-on-distinct-concat-map by blast

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]
proof(induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  have 1: distinct (map (f x) (g x))
    using Cons assms inj2-distinct-map by fastforce

  have 2: distinct (concat (map (λx. map (f x) (g x)) xs))
    using Cons by simp

  have 3: ⋀xa xb xc. ⟦xa ∈ set xs; xb ∈ set (g x); f x xb = f xa xc; xc ∈ set (g xa)⟧
      ⟹ False
    using Cons assms unfolding inj2-on-def by auto

   show ?case using 1 2 3
    by auto
qed

lemma distinct-concat-Nil: distinct (concat (map (λy. []) xs))
  by(induct xs) auto

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

assumes *inj2 f*
shows⟦∀ *x* ∈ *set xs. distinct* (*g x*); *distinct xs*⟧ ⟹ *distinct* [*f x y. x* ← *xs*, *y* ← *g x*, *h x*]
proof(*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
have *1*: *distinct* (*map* (*f x*) (*g x*))
using *Cons assms inj2-distinct-map* **by** *fastforce*

have *2*: *distinct* (*concat* (*map* (λ*x. concat* (*map* (λ*y. if h x then* [*f x y*] *else* []) (*g x*))) *xs*))
using *Cons* **by** *simp*

have *3*: ⋀*xa xb xc.*
⟦*h x*; *xa* ∈ *set* (*g x*); *xb* ∈ *set xs*; *f x xa* = *f xb xc*; *xc* ∈ *set* (*g xb*); *xc* ∈ (*if h xb then UNIV else* {})⟧ ⟹ *False*
by (*metis Cons.prems*(*2*) *assms distinct.simps*(*2*) *inj2-on-def iso-tuple-UNIV-I*)

then have *4*: *distinct* (*concat* (*map* (λ*y.* []) (*g x*)))
using *distinct-concat-Nil* **by** *auto*

show *?case* **using** *1 2 3 4* **by** *auto*
qed

1.3 Specializations of *inj2*

1.3.1 Cons

lemma *Cons-inj2*: *inj2* (#)
unfolding *inj2-on-def* **by** *simp*

lemma *Cons-distinct-concat-map*: ⟦*distinct ys*; *distinct xs*⟧ ⟹ *distinct* [*x*#*y. x* ← *xs*, *y* ← *ys*]
using *inj2-distinct-concat-map Cons-inj2* **by** *auto*

lemma *Cons-distinct-concat-map-function*:
⟦∀ *x* ∈ *set xs. distinct* (*g x*) ; *distinct xs*⟧ ⟹ *distinct* [*x* # *y. x* ← *xs*, *y* ← *g x*]
using *inj2-distinct-concat-map-function Cons-inj2* **by** *auto*

lemma *Cons-distinct-concat-map-function-distinct-on-all*:
⟦∀ *x. distinct* (*g x*) ; *distinct xs*⟧ ⟹ *distinct* [*x* # *y. x* ← *xs*, *y* ← *g x*]
using *Cons-distinct-concat-map-function* **by** (*metis* (*full-types*))

1.3.2 Node right

lemma *Node-right-inj2*: *inj2* (λ*l r. Node l e r*)
unfolding *inj2-on-def* **by** *simp*

lemma *Node-right-distinct-concat-map*:
  ⟦*distinct ys*; *distinct xs*⟧ ⟹ *distinct* [*Node x e y*. *x* ← *xs*, *y* ← *ys*]
  using *inj2-distinct-concat-map Node-right-inj2* **by** *fast*

1.3.3 Node left

lemma *Node-left-inj2*: *inj2* (λ*r l*. *Node l e r*)
  unfolding *inj2-on-def* **by** *simp*

lemma *Node-left-distinct-map*: *distinct xs* = *distinct* (*map* (λ*l*. ⟨*l*, (), *r*⟩) *xs*)
  using *inj2-distinct-map Node-left-inj2* **by** *fast*

1.3.4 Cons Suc

lemma *Cons-Suc-inj2*: *inj2* (λ*x ys*. *Suc x* # *ys*)
  unfolding *inj2-on-def* **by** *simp*

lemma *Cons-Suc-distinct-concat-map-function*:
  ⟦∀ *x* ∈ *set xs*. *distinct* (*g x*) ; *distinct xs*⟧ ⟹ *distinct* [*Suc x* # *y*. *x* ← *xs*, *y* ← *g x*]
  using *inj2-distinct-concat-map-function Cons-Suc-inj2* **by** *auto*

2 Lemmas for cardinality proofs

lemma *length-concat-map*: *length* [*f x r* . *x* ← *xs*, *r* ← *ys*] = *length ys* ∗ *length xs*
  by(*induct xs arbitrary*: *ys*) *auto*

An useful extension to *length-concat*

thm *length-concat*
lemma *length-concat-map-function-sum-list*:
  assumes ⋀ *x*. *x* ∈ *set xs* ⟹ *length* (*g x*) = *h x*
  shows *length* [*f x r* . *x* ← *xs*, *r* ← *g x*] = *sum-list* (*map h xs*)
  using *assms* **by**(*induct xs*) *auto*

lemma *sum-list-extract-last*: (∑*x*←[*0*..<*Suc n*]. *f x*) = (∑*x*←[*0*..<*n*]. *f x*) + *f n*
  by(*induct n*) (*auto simp*: *add.assoc*)

lemma *leq-sum-to-sum-list*: (∑*x* ≤ *n*. *f x*) = (∑*x*←[*0*..<*Suc n*]. *f x*)
  by (*metis atMost-upto sum-set-upt-conv-sum-list-nat*)

lemma *less-sum-to-sum-list*: (∑*x* < *n*. *f x*) = (∑*x*←[*0*..< *n*]. *f x*)
  by (*simp add*: *atLeast-upt sum-list-distinct-conv-sum-set*)

3 Miscellaneous

Similar to *length-remove1*:

lemma *Suc-length-remove1*: *x* ∈ *set xs* ⟹ *Suc* (*length* (*remove1 x xs*)) = *length xs*

```
  by(induct xs) auto
```

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-list (replicate x y) y = x
  by (induct x) auto

lemma count-list-full-elem: count-list xs y = length xs ⟷ (∀ x ∈ set xs. x = y)
proof(induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons z xs)
  have ⟦count-list xs y = Suc (length xs); x ∈ set xs⟧ ⟹ x = y for x
    by (metis Suc-n-not-le-n count-le-length)
  then show ?case
    using Cons by auto
qed
```

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

```
thm count-notin
lemma count-list-zero-not-elem: count-list xs x = 0 ⟷ x ∉ set xs
  by(induct xs) auto

lemma count-list-length-replicate: count-list xs y = length xs ⟷ xs = replicate
(length xs) y
  by (metis count-list-full-elem count-list-replicate replicate-length-same)

lemma count-list-True-False: count-list xs True + count-list xs False = length xs
  by(induct xs) auto


end
```

4 N-Sequences

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

4.1 Definition

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

Cardinality: *card A ^ n*

Example: *n-sequences* {*0*, *1*} *2* = {[*0*,*0*], [*0*,*1*], [*1*,*0*], [*1*,*1*]}

4.2 Algorithm

```
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]
```

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-sequence-enum xs n) = set (List.n-lists n xs)
  by(induct n) auto

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 standard
  show set (n-sequence-enum xs n) ⊆ n-sequences (set xs) n
    unfolding n-sequences-def by (induct n) auto+
next
  show n-sequences (set xs) n ⊆ set (n-sequence-enum xs n)
  proof(induct n)
    case 0
    then show ?case
      unfolding n-sequences-def by auto
  next
    case (Suc n)

     have ⟦n-sequences (set xs) n ⊆ set (n-sequence-enum xs n); set x ⊆ set xs;
length x = Suc n⟧
      ⟹ ∃ xa∈set xs. x ∈ (#) xa ` set (n-sequence-enum xs n) for x
      unfolding n-sequences-def by (cases x) auto

    from this Suc show ?case
      unfolding n-sequences-def by auto
  qed
qed
```

4.3.2 Distinctness

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

4.3.3 Cardinality

```
lemma n-sequence-enum-length:
  length (n-sequence-enum xs n) = (length xs) ^ n
  by(induct n arbitrary: xs) (auto simp: length-concat-map)
```

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 −
  obtain xs where set: set xs = A and dis: distinct xs
    using assms finite-distinct-list by auto
  have length (n-sequence-enum xs n) = (length xs) ^ n
    using n-sequence-enum-distinct n-sequence-enum-length by auto
  then have card (set (n-sequence-enum xs n)) = card (set xs) ^ n
    by (simp add: dis distinct-card n-sequence-enum-distinct)
  then have card (n-sequences (set xs) n) = card (set xs) ^ n
    by (simp add: n-sequence-enum-correct)
  then show card (n-sequences A n) = card A ^ n
    using set by simp
qed

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)
 by (simp add: length-finite-permutations-of-set n-permutations-def permutations-of-setD
subsetI)
```

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(induct n arbitrary: ys xs)
  case 0
  then show ?case by simp
next
  case (Suc n)
  obtain x where o1: x∈set xs and o2:  ys ∈ (#) x ` set (n-permutation-enum
(remove1 x xs) n)
    using Suc by auto

  have y ∈ set (n-permutation-enum (remove1 x xs) n) ⟹ set y ⊆ set xs for y
    using Suc set-remove1-subset by fast

  then show ?case using o1 o2
    by fastforce
qed

lemma n-permutation-enum-length: ys ∈ set (n-permutation-enum xs n) ⟹ length
ys = n
  by (induct n arbitrary: ys xs) auto

lemma n-permutation-enum-elem-distinct: distinct xs ⟹ ys ∈ set (n-permutation-enum
xs n) ⟹ distinct ys
proof (induct n arbitrary: ys xs)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  then obtain z zs where o: ys = z # zs
    by auto
  from this Suc have t: zs ∈ set (n-permutation-enum (remove1 z xs) n)
    by auto

  then have distinct zs
    using Suc distinct-remove1 by fast

  also have z ∉ set zs
    using o t n-permutation-enum-subset Suc by fastforce

  ultimately show ?case
    using o by simp
qed

lemma n-permutation-enum-correct1: distinct xs ⟹ set (n-permutation-enum xs
n) ⊆ n-permutations (set xs) n
  unfolding n-permutations-def
 using n-permutation-enum-subset n-permutation-enum-elem-distinct n-permutation-enum-length
  by fast

lemma n-permutation-enum-correct2: ys ∈ n-permutations (set xs) n ⟹ ys ∈ set
(n-permutation-enum xs n)
proof(induct n arbitrary: xs ys)
  case 0
  then show ?case unfolding n-permutations-def by simp
next
  case (Suc n)
  show ?case proof(cases ys)
    case Nil
    then show ?thesis using Suc
      by (simp add: n-permutations-def)
  next
    case (Cons z zs)

    have z-in: z ∈ set xs
      using Suc Cons unfolding n-permutations-def by simp

    have 1: set zs ⊆ set xs
      using Suc Cons unfolding n-permutations-def by simp

    have 2: length zs = n
      using Suc Cons unfolding n-permutations-def by simp

    have 3: distinct zs
      using Suc Cons unfolding n-permutations-def by simp

    show ?thesis proof(cases z ∈ set zs)
      case True
      then have zs ∈ set (n-permutation-enum (remove1 z xs) n)
        using Suc Cons unfolding n-permutations-def by auto
      then show ?thesis
        using True Cons z-in by auto
    next
      case False
      then have x ∈ set zs ⟹ x ∈ set (remove1 z xs) for x
        using 1 by(cases x = z) auto

      then have zs ∈ n-permutations (set (remove1 z xs)) n
        unfolding n-permutations-def using 2 3 by auto
      then have zs ∈ set (n-permutation-enum (remove1 z xs) n)
        using Suc by simp
      then have ∃x∈set xs. z # zs ∈ (#) x ‘ set (n-permutation-enum (remove1 x xs) n)
        unfolding image-def using z-in by simp
      then show ?thesis
        using False Cons by simp
    qed
  qed
qed

theorem n-permutation-enum-correct: distinct xs ⟹ set (n-permutation-enum xs n) = n-permutations (set xs) n
proof standard
  show distinct xs ⟹ set (n-permutation-enum xs n) ⊆ n-permutations (set xs) n
    by (simp add: n-permutation-enum-correct1)
next
  show distinct xs ⟹ n-permutations (set xs) n ⊆ set (n-permutation-enum xs n)
    by (simp add: n-permutation-enum-correct2 subsetI)
qed
```

5.3.2 Distinctness

```
theorem n-permutation-distinct: distinct xs ⟹ distinct (n-permutation-enum xs n)
proof(induct n arbitrary: xs)
  case 0
  then show ?case by simp
next
  case (Suc n)
  let ?f = λx. (n-permutation-enum (remove1 x xs) n)
  from Suc have distinct (?f x) for x
    by simp

  from this Suc show ?case
    by (auto simp: Cons-distinct-concat-map-function-distinct-on-all [of ?f xs])
qed
```

5.3.3 Cardinality

```
thm card-lists-distinct-length-eq
```

```
theorem finite A ⟹ card (n-permutations A n) = ffact n (card A)
  unfolding n-permutations-def using card-lists-distinct-length-eq
  by (metis (no-types, lifting) Collect-cong)
```

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)
  by auto

lemma n-multiset-permutation-enum-correct1:
  mset ys ⊆# mset xs ⟹ ys ∈ set (n-permutation-enum xs (length ys))
proof(induct ys arbitrary: xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons y ys)
  then have y ∈ set xs
    by (simp add: insert-subset-eq-iff)
  moreover have ys ∈ set (n-permutation-enum (remove1 y xs) (length ys))
    using Cons by (simp add: insert-subset-eq-iff)
  ultimately show ?case
    using Cons by auto
qed

lemma n-multiset-permutation-enum-correct2:
  ys ∈ set (n-permutation-enum xs n) ⟹ mset ys ⊆# mset xs
proof(induct n arbitrary: xs ys)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  then show ?case
    using insert-subset-eq-iff mset-remove1 by fastforce
qed

lemma n-multiset-permutation-enum-correct:
  set (n-multiset-permutation-enum xs n) = n-multiset-permutations (mset xs) n
  unfolding n-multiset-permutations-def
proof(standard)
  show set (n-multiset-permutation-enum xs n) ⊆ {xsa. mset xsa ⊆# mset xs ∧
length xsa = n}
```

```
  by (simp add: n-multiset-permutation-enum-correct2 n-permutation-enum-length
subsetI)
next
 show {xsa. mset xsa ⊆# mset xs ∧ length xsa = n} ⊆ set (n-multiset-permutation-enum
xs n)
   using n-multiset-permutation-enum-correct1 by auto
qed


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))
  by(induct bs xs rule: filter-bool-list.induct) auto

lemma filter-bool-list-in:
  n < length xs ⟹ n < length bs ⟹ bs!n ⟹ xs!n ∈ set (filter-bool-list bs xs)
proof (induct bs xs arbitrary: n rule: filter-bool-list.induct)
  case (3 b bs x xs)
  then show ?case by(cases n) auto
qed auto

lemma filter-bool-list-not-elem: x ∉ set xs ⟹ x ∉ set (filter-bool-list bs xs)
  by(induct bs xs rule: filter-bool-list.induct) auto

lemma filter-bool-list-elem: x ∈ set (filter-bool-list bs xs) ⟹ x ∈ set xs
  using filter-bool-list-not-elem by fast

lemma filter-bool-list-not-in:
  distinct xs ⟹ n < length xs⟹ n < length bs ⟹ bs!n = False
    ⟹ xs!n ∉ set (filter-bool-list bs xs)
proof (induct bs xs arbitrary: n rule: filter-bool-list.induct)
  case (3 b bs x xs)
  then show ?case proof(induct n)
    case 0
    then show ?case using filter-bool-list-not-elem
      by force
  qed auto
qed auto

lemma filter-bool-list-elem-nth: ys ∈ set (filter-bool-list bs xs)
  ⟹ ∃ n. ys = xs ! n ∧ bs ! n ∧ n < length bs ∧ n < length xs
proof(induct bs xs arbitrary: ys rule: filter-bool-list.induct)
  case (1 xs)
  then show ?case by simp
next
  case (2 b bs)
  then show ?case by simp
next
  case (3 b bs y ys)
  then show ?case
    by(cases b) (force)+
qed
```

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-bool-list bs xs) = {xs ! n |n. bs ! n ∧ n < length bs ∧ n < length xs}
  by (auto simp: filter-bool-list-in filter-bool-list-elem-nth)

lemma filter-bool-list-exist-length: A ⊆ set xs
  ⟹ ∃ bs. length bs = length xs ∧ A = set (filter-bool-list bs xs)
proof(induct xs arbitrary: A)
  case Nil
  then show ?case
    by auto
next
  case (Cons x xs)
  from Cons have A − {x} ⊆ set xs
    by auto
 from this Cons have 1: ∃ bs. length bs = length xs ∧ A − {x} = set (filter-bool-list
bs xs)
    by simp

  then have ∃ bs. length bs = length (x # xs) ∧ A = set (filter-bool-list bs (x #
xs))
    by (metis Diff-empty Diff-insert0 insert-Diff-single insert-absorb list.simps(15)
list.size(4) filter-bool-list.simps(3))

  then show ?case .
qed

lemma filter-bool-list-card:
  ⟦distinct xs; length xs = length bs⟧ ⟹ card (set (filter-bool-list bs xs)) = count-list
```

```
bs True
  by(induct bs xs rule: filter-bool-list.induct) (auto simp: filter-bool-list-not-elem)

lemma filter-bool-list-exist-length-card-True: ⟦distinct xs; A ⊆ set xs; n = card A⟧
    ⟹ ∃ bs. length bs = length xs ∧ count-list bs True = card A ∧ A = set
(filter-bool-list bs xs)
  by (metis filter-bool-list-card filter-bool-list-exist-length)

lemma filter-bool-list-distinct: distinct xs ⟹ distinct (filter-bool-list bs xs)
  by(induct bs xs rule: filter-bool-list.induct) (auto simp: filter-bool-list-not-elem)

lemma filter-bool-list-inj-aux:
  assumes length bs1 = length xs
  and length xs = length bs2
  and distinct xs
shows filter-bool-list bs1 xs = filter-bool-list bs2 xs ⟹ bs1 = bs2
using assms proof(induct rule: list-induct3)
  case Nil
  then show ?case by simp
next
  case (Cons b1 bs1 x xs b2 bs2)
  then show ?case
    by(cases b1; cases b2, auto) (metis list.set-intros(1) filter-bool-list-not-elem)+
qed

lemma filter-bool-list-inj:
  distinct xs ⟹ inj-on (λbs. filter-bool-list bs xs) {bs. length bs = length xs}
  unfolding inj-on-def using filter-bool-list-inj-aux by fastforce

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
 by (induct x arbitrary: xs n) (auto split: if-splits simp: count-list-replicate)

lemma n-bool-lists-length: xs ∈ set (n-bool-lists n x) ⟹ length xs = x
  by (induct x arbitrary: xs n) (auto split: if-splits)

lemma n-bool-lists-distinct: distinct (n-bool-lists n x)
proof(induct x arbitrary: n)
  case 0
  then show ?case by simp
next
  case (Suc x)
  then show ?case
    using distinct-map by fastforce
qed

lemma replicate-True-not-False: count-list ys True = 0 ⟷ ys = replicate (length
ys) False
   using count-list-zero-not-elem count-list-full-elem count-list-length-replicate by
fastforce

lemma n-bool-lists-correct-aux:
  length xs = x ⟹ count-list xs True = n ⟹ xs ∈ set (n-bool-lists n x)
proof(induct x arbitrary: n xs)
  case 0
  then show ?case by auto
next
  case (Suc x)
   show ?case proof(cases n = 0)
    case True
    then show ?thesis
      using Suc True replicate-True-not-False by auto
  next
```

```
    case c1: False
    then show ?thesis proof(cases n = Suc x)
      case True
      then have xs = True # replicate x True
        using Suc.prems count-list-length-replicate replicate-Suc by metis
      then show ?thesis
        using True by simp
    next
      case c2: False
      then show ?thesis proof(cases n > x)
        case True
        then have xs = []
          using Suc.prems c2 count-le-length by (metis Suc-lessI linorder-not-less)
        then show ?thesis
          using Suc by auto
      next
        case c3: False
        then show ?thesis proof (cases xs)
          case Nil
          then show ?thesis
            using Suc.prems(1) by auto
        next
          case (Cons y ys)
          then show ?thesis proof (cases y)
            case True
            then show ?thesis using Suc c1 c2 c3 Cons
              by simp
          next
            case False
            then show ?thesis using Suc c1 c2 c3 Cons
              by simp
          qed
        qed
      qed
    qed
  qed
qed

lemma n-bool-lists-correct: set (n-bool-lists n x) = {xs. length xs = x ∧ count-list
xs True = n}
proof(standard)
  show set (n-bool-lists n x) ⊆ {xs. length xs = x ∧ count-list xs True = n}
  proof(cases x)
    case 0
    then show ?thesis by simp
  next
    case (Suc x)
    then show ?thesis using n-bool-lists-True-count n-bool-lists-length
      by blast
  qed
next
  show {xs. length xs = x ∧ count-list xs True = n} ⊆ set (n-bool-lists n x)
    using n-bool-lists-correct-aux by auto
qed
```

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)
  unfolding n-subsets-def
  by (auto simp: filter-bool-list-card filter-bool-list-elem)

lemma n-subset-enum-correct-aux2:
  distinct xs ⟹ n-subsets (set xs) n ⊆ set (map set (n-subset-enum xs n))
  unfolding n-subsets-def
  by (auto simp: n-bool-lists-correct image-def filter-bool-list-exist-length-card-True)

theorem n-subset-enum-correct:
  distinct xs ⟹ set (map set (n-subset-enum xs n)) = n-subsets (set xs) n
proof(standard)
  show distinct xs ⟹ set (map set (n-subset-enum xs n)) ⊆ n-subsets (set xs) n
    using n-subset-enum-correct-aux1 n-bool-lists-correct by auto
next
  show distinct xs ⟹ n-subsets (set xs) n ⊆ set (map set (n-subset-enum xs n))
    using n-subset-enum-correct-aux2 by auto
qed
```

6.3.3 Distinctness

```
theorem n-subset-enum-distinct-elem:
  distinct xs ⟹ ys ∈ set (n-subset-enum xs n) ⟹ distinct ys
  by(cases length xs < n) (auto simp: filter-bool-list-distinct)

theorem n-subset-enum-distinct: distinct xs ⟹ distinct (n-subset-enum xs n)
  by(auto simp: distinct-map n-bool-lists-distinct inj-on-def filter-bool-list-inj-aux
n-bool-lists-length)
```

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
  by(induct ys arbitrary: n) auto

lemma count-mset-count-list: count (mset ys) T = n ⟹ count-list ys T = n
  by(induct ys arbitrary: n) auto

lemma count-mset-replicate-aux1:
  ⟦¬ x < n; mset ys = mset (replicate n True) + mset (replicate (x − n) False)⟧
    ⟹ count (mset ys) True = n
  by (auto simp: count-list-count-mset count-mset)

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 −
  have count-list xs B =
            count-list (replicate (count-list xs True) True) B + count-list (replicate
(length xs − count-list xs True) False) B
    for B
  proof(cases B)
    case True
    then show ?thesis
      by (simp add: count-list-replicate)
  next
    case False

    have count-list xs False = count-list (replicate (length xs − count-list xs True)
False) False
      by (metis count-list-True-False count-list-replicate diff-add-inverse)

    from this False show ?thesis
      using assms by auto
  qed

  then have count (mset xs) B =
          count (mset (replicate (count-list xs True) True) + mset (replicate (length
xs − count-list xs True) False)) B
    for B
    by (metis count-mset-count-list count-union)

 then show mset xs = mset (replicate (count-list xs True) True) + mset (replicate
(length xs − count-list xs True) False)
    using multiset-eqI by blast
qed

lemma n-bool-lists2-correct: set (n-bool-lists n x) = n-bool-lists2 n x
proof(standard)
  have ⟦¬ length ys < count-list ys True; x = length ys; n = count-list ys True⟧
        ⟹ ys ∈ permutations-of-multiset
                (mset (replicate (count-list ys True) True) + mset (replicate (length
ys − count-list ys True) False))
        for ys
    using count-mset-replicate-aux2 permutations-of-multisetI by blast

  then show set (n-bool-lists n x) ⊆ n-bool-lists2 n x
    unfolding n-bool-lists-correct
    by (auto simp: count-le-length leD)
next
  have ⟦¬ x < n; ys ∈ permutations-of-multiset (mset (replicate n True) + mset
(replicate (x − n) False))⟧
        ⟹ count (mset ys) True = n  for ys
    using count-mset-replicate-aux1 permutations-of-multisetD by blast
  then have ⟦¬ x < n; ys ∈ permutations-of-multiset (mset (replicate n True) +
mset (replicate (x − n) False))⟧
        ⟹  count-list ys True = n  for ys
    by (simp add: count-list-count-mset)
  then show n-bool-lists2 n x ⊆ set (n-bool-lists n x) unfolding n-bool-lists-correct

    by (auto simp: length-finite-permutations-of-multiset)
qed

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 (λxs. [False #
xs, True # xs]) xs))
  by (induct xs) auto

lemma distinct-all-bool-lists : distinct (all-bool-lists x)
  by (induct x) (auto simp add: distinct-concat-aux)

lemma all-bool-lists-correct: set (all-bool-lists x) = {xs. length xs = x}
proof(standard)
  show set (all-bool-lists x) ⊆ {xs. length xs = x}
    by (induct x) auto
next
  show {xs. length xs = x} ⊆ set (all-bool-lists x)
  proof(induct x)
    case 0
    then show ?case by simp
  next
    case (Suc x)
    have length ys = Suc x ⟹ ∃ xs. ys = False # xs ∨ ys = True # xs for ys
      by (metis (full-types) Suc-length-conv)
    then show ?case using Suc
      by fastforce
  qed
qed
```

7.3.1 Correctness

```
theorem powerset-enum-correct: set (map set (powerset-enum xs)) = Pow (set xs)
proof(standard)
  show set (map set (powerset-enum xs)) ⊆ Pow (set xs)
    using filter-bool-list-not-elem by fastforce
next
  have ⋀x. x ⊆ set xs ⟹ x ∈ (λx. set (filter-bool-list x xs)) ‘ {zs. length zs =
length xs}
    unfolding image-def using filter-bool-list-exist-length image-def by auto
  then show Pow (set xs) ⊆ set (map set (powerset-enum xs))
```

```
    using all-bool-lists-correct by auto
qed
```

7.3.2 Distinctness

```
theorem powerset-enum-distinct-elem: distinct xs ⟹ ys ∈ set (powerset-enum
xs) ⟹ distinct ys
  using filter-bool-list-distinct by auto

theorem powerset-enum-distinct: distinct xs ⟹ distinct (powerset-enum xs)
proof −
  assume dis: distinct xs
  then have distinct (map (λx. filter-bool-list x xs) (all-bool-lists (length xs)))
    using distinct-map filter-bool-list-inj distinct-all-bool-lists
    by (metis all-bool-lists-correct)
  then show ?thesis
    using dis by simp
qed
```

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)
  by (auto simp add: n-sequence-enum-distinct)

lemma all-bool-lists2-correct: set (all-bool-lists n) = set (all-bool-lists2 n)
  by (auto simp: all-bool-lists-correct n-sequence-enum-correct n-sequences-def)

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
  by(induct n) auto

lemma integer-partitions-enum-aux-first:
  x # xs ∈ set (integer-partitions-enum-aux n m)
    ⟹ xs ∈ set (integer-partitions-enum-aux (n−x) x)
  by(induct n) auto

lemma integer-partitions-enum-aux-max-n:
  x#xs ∈ set (integer-partitions-enum-aux n m) ⟹ x ≤ n
  by (induct n) auto

lemma integer-partitions-enum-aux-max-head:
  x#xs ∈ set (integer-partitions-enum-aux n m) ⟹ x ≤ m
  by (induct n) auto


lemma integer-partitions-enum-aux-max:
  xs ∈ set (integer-partitions-enum-aux n m) ⟹ x ∈ set xs ⟹ x ≤ m
proof(induct xs arbitrary: n m x)
  case Nil
  then show ?case using integer-partitions-enum-aux-max-head by simp
next
  case (Cons y xs)
  then show ?case
    using integer-partitions-enum-aux-max-head integer-partitions-enum-aux-first
    by fastforce
qed

lemma integer-partitions-enum-aux-sum:
  xs ∈ set (integer-partitions-enum-aux n m) ⟹ sum-list xs = n
proof(induct xs arbitrary: n m)
  case Nil
  then show ?case using integer-partitions-empty by simp
next
  case (Cons x xs)
  then have ⟦xs ∈ set (integer-partitions-enum-aux (n−x) x)⟧ ⟹ sum-list xs =
(n−x)
    by simp
  moreover have xs ∈ set (integer-partitions-enum-aux (n−x) x)
    using Cons integer-partitions-enum-aux-first by simp
  moreover have x ≤ n
    using Cons integer-partitions-enum-aux-max-n by simp
  ultimately show ?case
    by simp
qed

lemma integer-partitions-enum-aux-not-null-aux:
  x#xs ∈ set (integer-partitions-enum-aux n m) ⟹ x ≠ 0
  by (induct n) auto

lemma integer-partitions-enum-aux-not-null:
  xs ∈ set (integer-partitions-enum-aux n m) ⟹ x ∈ set xs ⟹ x ≠ 0
proof(induct xs arbitrary: x n m)
  case Nil
  then show ?case by simp
next
  case (Cons y xs)
  show ?case proof(cases y = x)
    case True
    then show ?thesis
      using Cons integer-partitions-enum-aux-not-null-aux by simp
  next
    case False
    then show ?thesis
     using Cons integer-partitions-enum-aux-not-null-aux integer-partitions-enum-aux-first
       by fastforce
  qed
qed

lemma integer-partitions-enum-aux-head-minus:
     h ≤ m ⟹ h > 0 ⟹ n ≥ h ⟹
  ys ∈ set (integer-partitions-enum-aux (n−h) h)⟹ h#ys ∈ set (integer-partitions-enum-aux
n m)
proof(induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then have 1: 1 ≤ m by simp

  have 2: (∃x. (x = min (Suc n) m ∨ Suc 0 ≤ x ∧ x < Suc n ∧ x < m) ∧ h # ys
      ∈ (#) x ‘ set (integer-partitions-enum-aux (Suc n − x) x))
    unfolding image-def using Suc by auto

  from 1 2 have Suc 0 ≤ m ∧(∃x. (x = min (Suc n) m ∨ Suc 0 ≤ x ∧ x < Suc
n ∧ x < m)
        ∧ h # ys ∈ (#) x ‘ set (integer-partitions-enum-aux (Suc n − x) x))
    by simp

  then show ?case by auto
qed

lemma integer-partitions-enum-aux-head-plus:
  h ≤ m ⟹ h > 0 ⟹ ys ∈ set (integer-partitions-enum-aux n h)
    ⟹ h#ys ∈ set (integer-partitions-enum-aux (h + n) m)
  using integer-partitions-enum-aux-head-minus by simp

lemma integer-partitions-enum-correct-aux1:
  assumes 0 ∉# A
  and ∀x ∈# A. x ≤ m
shows ∃xs∈set (integer-partitions-enum-aux (∑# A) m). A = mset xs
using assms proof(induct A arbitrary: m rule: multiset-induct-max)
  case empty
  then show ?case by simp
next
  case (add h A)
  have hc1: h ≤ m
    using add by simp

  have hc2: h > 0
    using add by simp

  obtain ys where o1: ys ∈ set (integer-partitions-enum-aux (∑# A) h) and o2:
A = mset ys
    using add by force

  have h#ys ∈ set (integer-partitions-enum-aux (h + ∑# A) m)
    using integer-partitions-enum-aux-head-plus hc1 o1 hc2 by blast

  then show ?case
    using o2 by force
qed

theorem integer-partitions-enum-correct:
  set (map mset (integer-partitions-enum n)) = integer-partitions n
proof(standard)
  have ⟦xs ∈ set (integer-partitions-enum-aux n n)⟧ ⟹ ∑# (mset xs) = n for xs
```

```
    by (simp add: integer-partitions-enum-aux-sum sum-mset-sum-list)
  moreover have xs ∈ set (integer-partitions-enum-aux n n) ⟹ 0 ∉# mset xs for xs
    using integer-partitions-enum-aux-not-null by auto
 ultimately show set (map mset (integer-partitions-enum n)) ⊆ integer-partitions n
    unfolding integer-partitions-def by auto
next
  have 0 ∉# A ⟹ A ∈ mset ‘ set (integer-partitions-enum-aux (∑# A) (∑# A)) for A
    unfolding image-def
    using integer-partitions-enum-correct-aux1 by (simp add: sum-mset.remove)
  then show integer-partitions n ⊆ set (map mset (integer-partitions-enum n))
    unfolding integer-partitions-def by auto
qed
```

8.3.2 Distinctness

```
lemma integer-partitions-enum-aux-distinct:
  distinct (integer-partitions-enum-aux n m)
proof(induct n m rule:integer-partitions-enum-aux.induct)
  case (1 m)
  then show ?case by simp
next
  case (2 n m)
 have distinct [h#r . h ← [1..< Suc (min (Suc n) m)], r ← integer-partitions-enum-aux ((Suc n)−h) h]
    apply(subst Cons-distinct-concat-map-function)
    using 2 by auto
  then show ?case by simp
qed
```

```
theorem integer-partitions-enum-distinct:
  distinct (integer-partitions-enum n)
  using integer-partitions-enum-aux-distinct by simp
```

8.3.3 Cardinality

```
lemma partitions-bij-betw-count:
  bij-betw count {N. count N partitions n} {p. p partitions n}
 by (rule bij-betw-byWitness[where f′=Abs-multiset]) (auto simp: partitions-imp-finite-elements)
```

```
lemma card-partitions-count-partitions:
  card {p. p partitions n} = card {N. count N partitions n}
  using bij-betw-same-card partitions-bij-betw-count by metis
```

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}
  using card-partitions-count-partitions count-partitions-iff by simp
```

lemma *integer-partitions-number-partition-eq*:
  integer-partitions n = {*N. number-partition n N*}
  using *integer-partitions-def number-partition-def* **by** *auto*

lemma *integer-partitions-cardinality-aux*:
  card (*integer-partitions n*) = ($\sum$ *k*≤*n. Partition n k*)
 using *card-partitions-number-partition integer-partitions-number-partition-eq card-partitions*
  by *simp*

theorem *integer-partitions-cardinality*:
  card (*integer-partitions n*) = *Partition* (*2*∗*n*) *n*
 using *integer-partitions-cardinality-aux Partition-sum-Partition-diff add-implies-diff le-add1 mult-2*
  by *simp*

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*))

```
  by(induct n) auto

lemma integer-composition-enum-not-null-aux:
  x#xs ∈ set (integer-composition-enum n) ⟹ x ≠ 0
  by(induct n) auto

lemma integer-composition-enum-not-null: xs ∈ set (integer-composition-enum n)
⟹ 0 ∉ set xs
proof(induct xs arbitrary: n)
  case Nil
  then show ?case
    by simp
next
  case (Cons a xs)
  then show ?case
   using integer-composition-enum-not-null-aux integer-composition-enum-tail-elem
    by fastforce
qed

lemma integer-composition-enum-empty: [] ∈ set (integer-composition-enum n)
⟹ n = 0
  by(induct n) auto

lemma integer-composition-enum-sum: xs ∈ set (integer-composition-enum n) ⟹
sum-list xs = n
proof(induct n arbitrary: xs rule: integer-composition-enum.induct)
  case 1
  then show ?case by simp
next
  case (2 x)
  show ?case proof(cases xs)
    case Nil
    then show ?thesis using 2 by auto
  next
    case (Cons y ys)
    have p1: sum-list ys = Suc x − y using 2 Cons
      by auto

    have Suc x ≥ y
      using 2 Cons by auto
    then have p2: sum-list ys = Suc x − y ⟹ y + sum-list ys = Suc x
      by simp

    show ?thesis
      using p1 p2 Cons by simp
  qed
qed

lemma integer-composition-enum-head-set:
  assumesx ≠ 0 and x ≤ n
 shows xs ∈ set (integer-composition-enum (n−x)) ⟹ x#xs ∈ set (integer-composition-enum
n)
using assms proof(induct n arbitrary: x xs)
  case 0
  then show ?case
    by simp
next
  case c1: (Suc n)
  from c1.prems have 1:
    ∀ y∈{0..<n}. x = Suc y ⟶ xs ∉ set (integer-composition-enum (n − y)) ⟹
x = Suc n
    by(induct x) simp-all

  then have 2: ∀ y∈{0..<n}. x = Suc y ⟶ xs ∉ set (integer-composition-enum
(n − y)) ⟹ xs = []
    using c1.prems(1) by simp
  show ?case using 1 2 by auto
qed

lemma integer-composition-enum-correct-aux:
  0 ∉ set xs ⟹ xs ∈ set (integer-composition-enum (sum-list xs))
  by(induct xs) (auto simp: integer-composition-enum-head-set)

theorem integer-composition-enum-correct:
  set (integer-composition-enum n) = integer-compositions n
proof standard
  show set (integer-composition-enum n) ⊆ integer-compositions n
    unfolding integer-compositions-def
      using integer-composition-enum-not-null integer-composition-enum-sum by
auto
next
  show integer-compositions n ⊆ set (integer-composition-enum n)
    unfolding integer-compositions-def
    using integer-composition-enum-correct-aux by auto
qed
```

9.3.2 Distinctness

```
theorem integer-composition-enum-distinct:
  distinct (integer-composition-enum n)
proof(induct n rule: integer-composition-enum.induct)
  case 1
  then show ?case by auto
next
  case (2 n)

  have h1: x ∈ set [0..<Suc n] ⟹ distinct (integer-composition-enum (n − x))
for x
```

```
    using 2 by simp

  have h2: distinct [0..<n]
    by simp

  have distinct [Suc m #xs. m ← [0..< n], xs ← integer-composition-enum (n−m)]
    using h1 h2 Cons-Suc-distinct-concat-map-function by simp
  then show ?case by auto
qed
```

9.3.3 Cardinality

```
lemma sum-list-two-pow-aux:
  (∑x←[0..< n]. (2::nat) ^ (n − x)) + 2 ^ (0 − 1) + 2 ^ 0 = 2 ^ (Suc n)
proof(induct n)
  case 0
  then show ?case by simp
next
  case c1: (Suc n)

  have x ≤ n ⟹ 2 ^ (Suc n − x) = 2 ∗ 2^ (n − x) for x
    by (simp add: Suc-diff-le)
  also have x ∈ set [0..<Suc n] ⟹ x ≤ n for x
    by auto
   ultimately have (∑x←[0..<Suc n]. 2 ^ (Suc n − x)) = (∑x←[0..<Suc n].
2∗ 2 ^ (n − x))
     by (metis (mono-tags, lifting) map-eq-conv)

  also have ... = (∑x←[0..< n]. 2∗ 2 ^ (n − x)) + 2∗ 2 ^ (0)
    using sum-list-extract-last by simp

  also have (∑x←[0..< n]. (2::nat)∗ (2::nat) ^ (n − x)) = 2∗(∑x←[0..< n]. 2
^ (n − x))
    using sum-list-const-mult by fast

  ultimately have (∑x←[0..<Suc n]. (2::nat) ^ (Suc n − x))
      = 2∗(∑x←[0..< n]. 2 ^ (n − x)) + 2∗ 2 ^ (0)
    by metis

  then show ?case using c1
    by simp
qed

lemma sum-list-two-pow:
  Suc (∑x←[0..<n]. 2 ^ (n − Suc x)) = 2 ^ n
  using sum-list-two-pow-aux sum-list-extract-last by(cases n) auto

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

```
proof(induct n rule: integer-composition-enum.induct)
  case 1
  then show ?case by simp
next
  case (2 n)
  then have length [Suc m #xs. m ← [0..< n], xs ← integer-composition-enum
(n−m)]
        = (∑x←[0..<n]. 2 ^(n − x − 1))
    using length-concat-map-function-sum-list [of
        [0..< n]
        λ x. integer-composition-enum (n − x)
        λ x. 2 ^(n − x − 1)
        λ m xs. Suc m #xs]
    by auto

  then show ?case
    using sum-list-two-pow
    by simp
qed

theorem integer-compositions-card:
  card (integer-compositions n) = 2^(n−1)
  using integer-composition-enum-correct integer-composition-enum-length
    integer-composition-enum-distinct distinct-card by metis

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(induct l arbitrary: xs i)
  case 0
  then show ?case by simp
next
  case (Suc l)
  then show ?case by(cases l) auto
qed

lemma weak-integer-composition-enum-sum-list:
  xs ∈ set (weak-integer-composition-enum i l) ⟹ sum-list xs = i
proof(induct l arbitrary: xs i)
  case 0
  then show ?case by simp
next
  case (Suc l)
  then show ?case by(cases l) auto
qed

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(cases length xs)
  case 0
  then show ?thesis by simp
next
  case (Suc y)


  have 1: ⟦n ∈ set xs; 0 < n⟧ ⟹ 0 < sum-list xs for n
    using sum-list-eq-0-iff by fast


  have 2: xs ∉ set (weak-integer-composition-enum 0 (Suc y)) ⟹ 0 < sum-list xs
    using Suc assms not-gr0 by fastforce

  have x # xs ∉ (#) (x + sum-list xs) ‘ set (weak-integer-composition-enum 0 (Suc y))
  ⟹ ∃ xa∈{0..<x + sum-list xs}. x # xs ∈ (#) xa ‘ set (weak-integer-composition-enum
(x + sum-list xs − xa) (Suc y))
    unfolding image-def using Suc assms 1 2 by auto

  from Suc this show ?thesis
    by auto
qed

lemma weak-integer-composition-enum-correct-aux:
  xs ∈ set (weak-integer-composition-enum (sum-list xs) (length xs))
  by (induct xs) (auto simp: weak-integer-composition-enum-head)

theorem weak-integer-composition-enum-correct:
  set (weak-integer-composition-enum i l) = weak-integer-compositions i l
proof standard
  show set (weak-integer-composition-enum i l) ⊆ weak-integer-compositions i l
    unfolding weak-integer-compositions-def
   using weak-integer-composition-enum-length weak-integer-composition-enum-sum-list
    by auto
next
  show weak-integer-compositions i l ⊆ set (weak-integer-composition-enum i l)
    unfolding weak-integer-compositions-def
    using weak-integer-composition-enum-correct-aux by auto
qed
```

10.3.2 Distinctness

```
theorem weak-integer-composition-enum-distinct: distinct (weak-integer-composition-enum
i l)
proof(induct rule: weak-integer-composition-enum.induct)
  case (1 i)
  then show ?case
    by simp
next
  case (2 i)
  then show ?case
    by simp
next
  case (3 i l)
  have distinct [h#r . h ← [0..< Suc i], r ← weak-integer-composition-enum (i−h)
(Suc l)]
    apply(subst Cons-distinct-concat-map-function)
    using 3 by auto
  then show ?case by simp
qed
```

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*
  by(*induct n*) *auto*

lemma *a-choose-equivalence*: *Suc* (∑*x*←[*0*..<*k*]. *n* + (*k* − *x*) *choose* (*k* − *x*)) = *Suc* (*n* + *k*) *choose k*
proof −
  have *m* ≥ *k* ⟹ (∑*x*←[*0*..< *Suc k*]. *m* − *x choose* (*k* − *x*)) = *Suc m choose k* **for** *m*
    using *sum-choose-diagonal leq-sum-to-sum-list* **by** *metis*
  then have *1*: *Suc* (∑*x*←[*0*..<*k*]. (*n* + *k*) − *x choose* (*k* − *x*)) = *Suc* (*n* + *k*) *choose k*
    by *simp*

  have *Suc* (∑*x*←[*0*..<*k*]. (*n* + *k*) − *x choose* (*k* − *x*)) = *Suc* (∑*x*←[*0*..<*k*]. *n* + (*k* − *x*) *choose* (*k* − *x*))
    by (*metis* (*no-types*, *opaque-lifting*) *Nat.diff-add-assoc2 add.commute binomial-n-0 diff-is-0-eq′ nle-le*)

  then show *?thesis* **using** *1* **by** *simp*
qed

lemma *composition-enum-length*: *length* (*weak-integer-composition-enum i n*) = *n multichoose i*
  unfolding *multichoose-def*
proof(*induct i n rule*: *weak-integer-composition-enum.induct*)
  case (*1 i*)
  then show *?case* **by** *simp*
next
  case (*2 i*)
  then show *?case* **by** *simp*
next
  case (*3 i n*)

  then have *x* ∈ *set* [*0*..< *i*] ⟹
    length (*weak-integer-composition-enum* (*i* − *x*) (*Suc n*)) = *n* + (*i* − *x*) *choose* (*i* − *x*) **for** *x*
    by *simp*

  then have *ev*: *length* [*h*#*r* . *h* ← [*0*..< *i*], *r* ← *weak-integer-composition-enum* (*i*−*h*) (*Suc n*)] =
      (∑*x*←[*0*..< *i*]. *n* + (*i* − *x*) *choose* (*i* − *x*))

```
    using length-concat-map-function-sum-list [of
      [0..< i]
      λx. (weak-integer-composition-enum (i−x) (Suc n))
      λx. n + (i−x) choose (i−x)
      λh r. h#r
      ] by simp

  have Suc (∑x←[0..<i]. n + (i − x) choose (i − x)) = Suc (n + i) choose i
    using a-choose-equivalence by simp

  then show ?case using weak-integer-composition-enum-zero ev by auto
qed

theorem weak-integer-compositions-cardinality: card (weak-integer-compositions n
k) = k multichoose n
 using weak-integer-composition-enum-correct weak-integer-composition-enum-distinct
composition-enum-length
  distinct-card by metis

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
  by(induct xs ys arbitrary: i rule: list-induct2) (auto simp: less-Suc-eq-0-disj)

lemma nth-no-overlap: length xs = length ys ⟹ ∀ i < length xs. xs ! i ≠ ys ! i
⟹ no-overlap xs ys
proof (induct xs ys rule: list-induct2)
  case (Cons x xs y ys)
  then show ?case using Suc-less-eq nth-Cons-Suc by fastforce
qed simp

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 ∈ set (derangement-enum-aux xs
ys) ⟹ length xs = length zs
  by(induct xs arbitrary: ys zs) auto
```

```
lemma derangement-enum-aux-not-in: y ∉ set ys ⟹ zs ∈ set (derangement-enum-aux
xs ys) ⟹ y ∉ set zs
proof(induct xs arbitrary: ys zs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then obtain z zs2 where ob: zs = z#zs2
    by auto
  have zs2 ∈ set (derangement-enum-aux xs (remove1 z ys)) ⟹ y ∉ set zs2
    using Cons notin-set-remove1 by fast
  then show ?case using Cons ob
    by auto
qed
```

```
lemma derangement-enum-aux-in: y ∈ set zs ⟹ zs ∈ set (derangement-enum-aux
xs ys) ⟹ y ∈ set ys
  using derangement-enum-aux-not-in by fast
```

```
lemma derangement-enum-aux-distinct-elem: distinct ys ⟹ zs ∈ set (derangement-enum-aux
xs ys) ⟹ distinct zs
```

```
proof(induct xs arbitrary: ys zs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  obtain z zs2 where ob: zs = z#zs2
    using Cons by auto
  then have ev: zs2 ∈ set (derangement-enum-aux xs (remove1 z ys))
    using Cons ob by auto

  have distinct zs2
    using ev Cons distinct-remove1 by fast
  moreover have z ∉ set zs2
    using ev Cons(2) derangement-enum-aux-in by fastforce
  ultimately show ?case using ob by simp
qed

lemma derangement-enum-aux-no-overlap: zs ∈ set (derangement-enum-aux xs ys)
⟹ no-overlap xs zs
  by(induct xs arbitrary: zs ys) auto

lemma derangement-enum-aux-set:
  length xs = length ys ⟹ zs ∈ set (derangement-enum-aux xs ys) ⟹ set zs =
set ys
proof(induct xs ys arbitrary: zs rule: derangement-enum-aux.induct)
  case (1 ys)
  then show ?case by simp
next
  case (2 x xs ys)
  obtain z zs2 where ob: zs = z#zs2
    using 2 by auto
  have ev1: zs2 ∈ set (derangement-enum-aux xs (remove1 z ys))
    using 2 ob  by simp
  have ev2:z ∈ set ys
    using 2 ob by simp

  have length xs = length (remove1 z ys)
    using ev2 Suc-length-remove1 2.prems(1) by force
  then have set zs2 = set (remove1 z ys)
    using 2.hyps[of z zs2] ev1 ev2  by simp

  then show ?case
    using ob notin-set-remove1 ev2 in-set-remove1 by fastforce
qed

lemma derangement-enum-correct-aux1:
 ⟦distinct zs;length ys = length zs; length ys = length xs; set ys = set zs; no-overlap
xs zs⟧
   ⟹ zs ∈ set (derangement-enum-aux xs ys)
```

```
proof(induct xs arbitrary: zs ys)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  obtain z zs2 where ob: zs = z#zs2
    using Cons length-0-conv neq-Nil-conv by metis

  have e1: z ≠ x
    using Cons.prems(5) ob  by auto

  have distinct zs2
    using Cons.prems(1) ob by auto
  moreover have length (remove1 z ys) = length zs2 using Cons.prems ob
    by (simp add: length-remove1)
  moreover have length (remove1 z ys) = length xs
    by (simp add: Cons.prems(3) Cons.prems(4) length-remove1 ob)
  moreover have set (remove1 z ys) = set zs2
     using Cons ob by (metis distinct-card distinct-remdups length-remdups-eq remove1.simps(2) set-remdups set-remove1-eq)
  moreover have no-overlap xs zs2
    using Cons.prems(5) ob by fastforce

  ultimately have zs2 ∈ set (derangement-enum-aux xs (remove1 z ys))
    using Cons.hyps[of zs2 (remove1 z ys)] by simp
  then show ?case
    using ob e1 Cons by simp
qed

theorem derangement-enum-correct: distinct xs ⟹ derangements xs = set (derangement-enum xs)
proof(standard)
  show distinct xs ⟹ derangements xs ⊆ set (derangement-enum xs)
    unfolding derangements-def using derangement-enum-correct-aux1 by auto
next
  show distinct xs ⟹ set (derangement-enum xs) ⊆ derangements xs
    unfolding derangements-def
   using derangement-enum-aux-set derangement-enum-aux-distinct-elem derangement-enum-aux-elem-length derangement-enum-aux-no-overlap
    by auto
qed
```

11.3.2 Distinctness

```
lemma derangement-enum-aux-distinct: distinct ys ⟹ distinct (derangement-enum-aux xs ys)
proof(induct xs arbitrary: ys)
  case Nil
  then show ?case by simp
```

```
next
  case (Cons x xs)
  show ?case
    using inj2-distinct-concat-map-function-filter[of
        Cons
         ys
         λy. derangement-enum-aux xs (remove1 y ys)
        λy. y ≠ x
      ]
    using Cons Cons-inj2
    by (simp)
qed

theorem derangement-enum-distinct: distinct xs ⟹ distinct (derangement-enum
xs)
  using derangement-enum-aux-distinct by auto



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)) = (∑ i≤n. length(tree-enum i) ∗ length(tree-enum (n
− i)))
  by (simp add: length-concat comp-def sum-list-triv atLeast-upt interv-sum-list-conv-sum-set-nat
        flip: lessThan-Suc-atMost)
```

12.3.2 Correctness

```
lemma tree-enum-correct1: t ∈ set (tree-enum n) ⟹ size t = n
  by (induct n arbitrary: t rule: tree-enum.induct) (simp, fastforce)

lemma tree-enum-correct2: n = size t ⟹ t ∈ set (tree-enum n)
proof (induct n arbitrary: t rule: tree-enum.induct)
  case 1
  then show ?case by simp
next
  case (2 n)
  show ?case proof(cases t)
    case Leaf
    then show ?thesis
      by (simp add: 2.prems)
  next
    case (Node l e r)

    have i1: (size l) < Suc n using 2.prems Node by auto
    have i2: (size r) < Suc n using 2.prems Node by auto

    have t1: l ∈ set (tree-enum (size l))
      apply(rule 2.hyps(1) [of (size l)])
      using i1 by auto

    have t2: r ∈ set (tree-enum (size r))
      apply(rule 2.hyps(1) [of (size r)])
      using i2 by auto

    have ⟨l, (), r⟩ ∉ (λt1. ⟨t1, (), ⟨⟩⟩) ‘ set (tree-enum (size l + size r)) ⟹
        ∃ x∈{0..<size l + size r}. ∃ xa∈set (tree-enum x). ⟨l, (), r⟩ ∈ Node xa () ‘
set (tree-enum (size l + size r − x))
      using t1 t2 by fastforce
    then have ⟨l, e, r⟩ ∈ set (tree-enum (size ⟨l, e, r⟩))
      by auto

    then show ?thesis
      using Node using 2.prems by simp
  qed
qed
```

```
theorem tree-enum-correct: set(tree-enum n) = trees n
proof(standard)
  show set (tree-enum n) ⊆ trees n
    unfolding trees-def using tree-enum-correct1 by auto
next
  show trees n ⊆ set (tree-enum n)
    unfolding trees-def using tree-enum-correct2 by auto
qed
```

12.3.3 Distinctness

```
lemma tree-enum-Leaf: ⟨⟩ ∈ set (tree-enum n) ⟷ (n = 0)
  by(cases n) auto

lemma tree-enum-elem-injective: n ≠ m ⟹ x ∈ set (tree-enum n) ⟹ y ∈ set (tree-enum m) ⟹ x ≠ y
  using tree-enum-correct1 by auto

lemma tree-enum-elem-injective2: x ∈ set (tree-enum n) ⟹ y ∈ set (tree-enum m) ⟹ x = y ⟹ n = m
  using tree-enum-elem-injective by auto


lemma concat-map-Node-not-equal:
  xs ≠ [] ⟹ xs2 ≠ [] ⟹ ys ≠ [] ⟹ ys2 ≠ [] ⟹
  ∀ x∈ set xs. ∀ y ∈ set ys . x ≠ y ⟹
  [⟨l, (), r⟩. l ← xs2, r ← xs] ≠ [⟨l, (), r⟩. l ← ys2, r ← ys]
proof(induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then show ?case proof(induct ys)
    case Nil
    then show ?case by simp
  next
    case (Cons y ys)
    obtain x2 x2s where o1: xs2 = x2 # x2s
      by (meson Cons.prems(3) neq-Nil-conv)
    obtain y2 y2s where o2: ys2 = y2 # y2s
      by (meson Cons.prems(5) neq-Nil-conv)

    have [⟨l, (), r⟩. l ← x2#x2s, r ← x # xs] ≠ [⟨l, (), r⟩. l ← y2#y2s, r ← y # ys]
      using Cons.prems(6) by auto
    then show ?case
      using o1 o2 by simp
  qed
qed
```

```
lemma tree-enum-not-empty: tree-enum n ≠ []
  by(induct n) auto

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. [⟨l, (), r⟩. l ← tree-enum i, r ← tree-enum (n−i)]) xs)
using assms proof(induct xs arbitrary: n)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  have b1: x < n using Cons by auto

  have ∀  i ∈ set xs . x < i
    using Cons.prems(4) strict-sorted-simps(2) by simp
  then have ∀  i ∈ set xs . (n − i) < (n − x)
    using b1 diff-less-mono2 by simp

  then have ∀  i ∈ set xs. ∀  t1∈ set (tree-enum (n − x)). ∀  t2 ∈ set (tree-enum
(n − i)) . t1 ≠ t2
    using tree-enum-correct1 by (metis less-irrefl-nat)
  then have 1: ∀  i ∈ set xs. [⟨l, (), r⟩. l ← tree-enum x, r ← tree-enum (n−x)] ≠
          [⟨l, (), r⟩. l ← tree-enum i, r ← tree-enum (n−i)]
    using concat-map-Node-not-equal tree-enum-not-empty by simp

  have 2: distinct (map (λi. [⟨l, (), r⟩. l ← tree-enum i, r ← tree-enum (n−i)])
xs)
    using Cons by auto

  from 1 2 show ?case by auto
qed

lemma tree-enum-distinct-aux-left:
   ∀  i < n. distinct (tree-enum i) ⟹ distinct ([⟨l, (), r⟩. i ← [0..< n], l ←
tree-enum i])
proof(induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have 1:distinct (tree-enum n)
    using Suc.prems by auto
  have 2: distinct ([⟨l, (), r⟩. i ← [0..< n], l ← tree-enum i])
    using Suc by simp
  have 3: distinct (map (λl. ⟨l, (), r⟩) (tree-enum n))
```

```
    using Node-left-distinct-map 1 by simp

  have 4: ⟦⋀t n. t ∈ set (tree-enum n) ⟹ size t = n; m < n; y ∈ set (tree-enum
n); y ∈ set (tree-enum m)⟧ ⟹ False for m y
    by blast

  from 1 2 3 4 tree-enum-correct1 show ?case
    by fastforce
qed

theorem tree-enum-distinct: distinct(tree-enum n)
proof(induct n rule: tree-enum.induct)
  case 1
  then show ?case by simp
next
  case (2 n)
  then have Ir: i < Suc n ⟹ distinct (tree-enum i) for i
    by (metis atLeastLessThan-iff set-upt zero-le)

  have c1: distinct (concat (map (λi. [⟨l, (), r⟩. l ← tree-enum i, r ← tree-enum
(n−i)]) [0..<n]))
  proof(rule distinct-concat)
     show distinct (map (λi. [⟨l, (), r⟩. l ← tree-enum i, r ← tree-enum (n−i)])
[0..<n])
      apply(rule tree-enum-distinct-aux-outer)
      using Ir by auto
  next
     have ⋀x. x < n ⟹ distinct ([⟨l, (), r⟩. l ← tree-enum x, r ← tree-enum
(n−x)])
      using Ir by (simp add: Node-right-distinct-concat-map)
    then show ⋀ys. ys ∈ set (map (λi. [⟨l, (), r⟩. l ← tree-enum i, r ← tree-enum
(n−i)]) [0..<n]) ⟹ distinct ys
      by auto
  next
    have ⟦[⟨l, (), r⟩. l ← tree-enum x, r ← tree-enum (n−x)] ≠
        [⟨l, (), r⟩. l ← tree-enum z, r ← tree-enum (n−z)];
        y ∈ set (tree-enum x); y ∈ set (tree-enum z)⟧
       ⟹ False for x z y
      using tree-enum-elem-injective2 by auto
    then show  ⋀ys zs.
         ⟦ys ∈ set (map (λi. [⟨l, (), r⟩. l ← tree-enum i, r ← tree-enum (n−i)])
[0..<n]);
          zs ∈ set (map (λi. [⟨l, (), r⟩. l ← tree-enum i, r ← tree-enum (n−i)])
[0..<n]); ys ≠ zs⟧
       ⟹ set ys ∩ set zs = {}
      by fastforce
  qed

  have distinct (tree-enum n)
```

```
    using 2 by simp
  then have c2: distinct (map (λt1. ⟨t1, (), ⟨⟩⟩) (tree-enum n))
    using Node-left-distinct-map by fastforce

  have c3: ⋀xa xb. ⟦xa < n; xb ∈ set (tree-enum xa); xb ∈ set (tree-enum n); ⟨⟩ ∈
set (tree-enum (n − xa))⟧ ⟹ False
    by (simp add: tree-enum-Leaf)

  from c1 c2 c3 show ?case
    by fastforce
qed
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.