

Combinatorial Enumeration Algorithms

Paul Hofmeier and Emin Karayel

May 26, 2024

Abstract

Combinatorial objects have configurations which can be enumerated by algorithms, but especially for imperative programs, it is difficult to find out if they produce the correct output and don't generate duplicates. Therefore, for some of the most common combinatorial objects, namely `n_Sequences`, `n_Permutations`, `n_Subsets`, `Powerset`, `Integer_Compositions`, `Integer_Partitions`, `Weak_Integer_Compositions`, `Derangements` and `Trees`, this entry formalizes efficient functional programs and verifies their correctness. In addition, it provides cardinality proofs for those combinatorial objects. Some cardinalities are verified using the enumeration functions and others are shown using existing libraries including other AFP entries.

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

Contents

1	Injectivity for two argument functions	3
1.1	Correspondence between <i>inj2-on</i> and <i>inj-on</i>	4
1.2	Proofs with <i>inj2</i>	4
1.3	Specializations of <i>inj2</i>	6
1.3.1	<i>Cons</i>	6
1.3.2	<i>Node right</i>	6
1.3.3	<i>Node left</i>	7
1.3.4	<i>Cons Suc</i>	7
2	Lemmas for cardinality proofs	7
3	Miscellaneous	7
3.1	<i>count-list</i> and <i>replicate</i>	8

4	N-Sequences	8
4.1	Definition	8
4.2	Algorithm	9
4.3	Verification	9
4.3.1	Correctness	9
4.3.2	Distinctness	9
4.3.3	Cardinality	10
5	N-Permutations	10
5.1	Definition	10
5.2	Algorithm	11
5.3	Verification	11
5.3.1	Correctness	11
5.3.2	Distinctness	13
5.3.3	Cardinality	13
5.4	<i>n-multiset</i> extension (with <i>remdups</i>)	14
6	N-Subsets	17
6.1	Definition	17
6.2	Algorithm	18
6.3	Verification	18
6.3.1	<i>n-bool-lists</i>	18
6.3.2	Correctness	20
6.3.3	Distinctness	20
6.3.4	Cardinality	20
6.4	Alternative using Multiset permutations	20
6.5	<i>mset-count</i>	21
7	Powerset	22
7.1	Definition	22
7.2	Algorithm	23
7.3	Verification	23
7.3.1	Correctness	23
7.3.2	Distinctness	24
7.3.3	Cardinality	24
7.4	Alternative algorithm with <i>n-sequence-enum</i>	24
8	Integer Partitions	24
8.1	Definition	24
8.2	Algorithm	25
8.3	Verification	25
8.3.1	Correctness	25
8.3.2	Distinctness	28
8.3.3	Cardinality	28

9 Integer Compositions	29
9.1 Definition	29
9.2 Algorithm	29
9.3 Verification	29
9.3.1 Correctness	29
9.3.2 Distinctness	31
9.3.3 Cardinality	32
10 Weak Integer Compositions	33
10.1 Definition	33
10.2 Algorithm	33
10.3 Verification	34
10.3.1 Correctness	34
10.3.2 Distinctness	35
10.3.3 Cardinality	36
11 Derangements	37
11.1 Definition	37
11.2 Algorithm	38
11.3 Verification	38
11.3.1 Correctness	38
11.3.2 Distinctness	40
12 Trees	41
12.1 Definition	41
12.2 Algorithm	41
12.3 Verification	42
12.3.1 Cardinality	42
12.3.2 Correctness	42
12.3.3 Distinctness	43

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 ⇒ 'b ⇒ 'c) ⇒ 'a set ⇒ 'b set ⇒ bool where

```

$inj2\text{-on } f A B \longleftrightarrow (\forall x1 \in A. \forall x2 \in A. \forall y1 \in B. \forall y2 \in B. f x1 y1 = f x2 y2 \longrightarrow x1 = x2 \wedge y1 = y2)$

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

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

lemma $inj2\text{-curried}: inj2\text{-on } (curry f) A B \longleftrightarrow inj\text{-on } f (A \times B)$
unfolding $inj2\text{-on-def } inj\text{-on-def}$ **by** *auto*

lemma $inj2\text{-on-all}: inj2 f \Longrightarrow inj2\text{-on } f A B$
unfolding $inj2\text{-on-def}$ **by** *simp*

lemma $inj2\text{-inj-first}: inj2 f \Longrightarrow inj f$
unfolding $inj2\text{-on-def } inj\text{-on-def}$ **by** *simp*

lemma $inj2\text{-inj-second}: inj2 f \Longrightarrow inj (f x)$
unfolding $inj2\text{-on-def } inj\text{-on-def}$ **by** *simp*

lemma $inj2\text{-inj-second-flipped}: inj2 f \Longrightarrow inj (\lambda x. f x y)$
unfolding $inj2\text{-on-def } inj\text{-on-def}$ **by** *simp*

1.2 Proofs with $inj2$

Already existing for inj :

thm *distinct-map*

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

lemma $inj2\text{-distinct-map}$:
assumes $inj2 f$
shows $distinct xs = distinct (map (f x) xs)$
using *assms inj2-on-distinct-map inj2-on-all* **by** *fast*

lemma $inj2\text{-on-distinct-concat-map}$:
assumes $inj2\text{-on } f$ (*set xs*) (*set ys*)
shows $\llbracket distinct ys; distinct xs \rrbracket \Longrightarrow distinct [f x y. x \leftarrow xs, y \leftarrow ys]$
using *assms proof(induct xs)*
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
then have $nin: x \notin set xs$
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 xs))
    using Cons assms inj2-distinct-map by fastforce

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

  have 3: ∧ xa xb xc. [xa ∈ set xs; xb ∈ set (g xs); f x xb = f xa xc; xc ∈ set (g xs)]
    ⇒ 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  $\llbracket \forall x \in \text{set } xs. \text{distinct } (g x); \text{distinct } xs \rrbracket \implies \text{distinct } [f x y. x \leftarrow xs, y \leftarrow g x, h x]$ 
proof (induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  have 1:  $\text{distinct } (\text{map } (f x) (g x))$ 
    using Cons assms inj2-distinct-map by fastforce

  have 2:  $\text{distinct } (\text{concat } (\text{map } (\lambda x. \text{concat } (\text{map } (\lambda y. \text{if } h x \text{ then } [f x y] \text{ else } [])) (g x))) xs)$ 
    using Cons by simp

  have 3:  $\bigwedge xa \ xb \ xc. \llbracket h x; xa \in \text{set } (g x); xb \in \text{set } xs; f x xa = f x xb xc; xc \in \text{set } (g xb); xc \in (\text{if } h xb \text{ then UNIV else } \{\}) \rrbracket \implies \text{False}$ 
    by (metis Cons.prem1(2) assms distinct.simps(2) inj2-on-def iso-tuple-UNIV-I)

  then have 4:  $\text{distinct } (\text{concat } (\text{map } (\lambda y. []) (g x)))$ 
    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*: $\text{inj2 } (\#)$
unfolding *inj2-on-def by simp*

lemma *Cons-distinct-concat-map*: $\llbracket \text{distinct } ys; \text{distinct } xs \rrbracket \implies \text{distinct } [x \# y. x \leftarrow xs, y \leftarrow ys]$
using *inj2-distinct-concat-map Cons-inj2 by auto*

lemma *Cons-distinct-concat-map-function*:
 $\llbracket \forall x \in \text{set } xs. \text{distinct } (g x); \text{distinct } xs \rrbracket \implies \text{distinct } [x \# y. x \leftarrow xs, y \leftarrow g x]$
using *inj2-distinct-concat-map-function Cons-inj2 by auto*

lemma *Cons-distinct-concat-map-function-distinct-on-all*:
 $\llbracket \forall x. \text{distinct } (g x); \text{distinct } xs \rrbracket \implies \text{distinct } [x \# y. x \leftarrow xs, y \leftarrow g x]$
using *Cons-distinct-concat-map-function by (metis (full-types))*

1.3.2 Node right

lemma *Node-right-inj2*: $\text{inj2 } (\lambda l r. \text{Node } l e r)$
unfolding *inj2-on-def by simp*

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

$\llbracket \text{distinct } ys; \text{ distinct } xs \rrbracket \implies \text{distinct } [\text{Node } x \ e \ y. \ x \leftarrow xs, \ y \leftarrow ys]$
using *inj2-distinct-concat-map Node-right-inj2* **by** *fast*

1.3.3 Node left

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

unfolding *inj2-on-def* **by** *simp*

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

using *inj2-distinct-map Node-left-inj2* **by** *fast*

1.3.4 Cons Suc

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

unfolding *inj2-on-def* **by** *simp*

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

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

using *inj2-distinct-concat-map-function Cons-Suc-inj2* **by** *auto*

2 Lemmas for cardinality proofs

lemma *length-concat-map*: $\text{length } [f \ x \ r \ . \ x \leftarrow xs, \ r \leftarrow ys] = \text{length } ys * \text{length } xs$
by(*induct* xs *arbitrary: ys*) *auto*

An useful extension to *length-concat*

thm *length-concat*

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

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

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

using *assms* **by**(*induct* xs) *auto*

lemma *sum-list-extract-last*: $(\sum x \leftarrow [0..<\text{Suc } n]. \ f \ x) = (\sum x \leftarrow [0..<n]. \ f \ x) + f \ n$
by(*induct* n) (*auto simp: add.assoc*)

lemma *leq-sum-to-sum-list*: $(\sum x \leq n. \ f \ x) = (\sum x \leftarrow [0..<\text{Suc } n]. \ f \ x)$
by (*metis atMost-upto sum-set-upt-conv-sum-list-nat*)

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

3 Miscellaneous

Similar to *length-remove1*:

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

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* \longleftrightarrow ($\forall x \in \text{set } xs. x = y$)

proof(*induct xs*)

case *Nil*

then show *?case* **by** *simp*

next

case (*Cons z xs*)

have $\llbracket \text{count-list } xs \text{ y} = \text{Suc } (\text{length } xs); x \in \text{set } xs \rrbracket \implies x = y$ **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* \longleftrightarrow $x \notin \text{set } xs$

by(*induct xs*) *auto*

lemma *count-list-length-replicate*: *count-list xs y = length xs* \longleftrightarrow *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* \Rightarrow *nat* \Rightarrow '*a list set* **where**

n-sequences A n = $\{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\}$

Cardinality: $\text{card } A \hat{=} n$

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-sequence-enum xs 0 = []  
| n-sequence-enum xs (Suc n) = [x#r . x  $\leftarrow$  xs, r  $\leftarrow$  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)  $\subseteq$  n-sequences (set xs) n
```

```
    unfolding n-sequences-def by (induct n) auto+
```

```
next
```

```
  show n-sequences (set xs) n  $\subseteq$  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  $\subseteq$  set (n-sequence-enum xs n); set x  $\subseteq$  set xs;  
length
```

x = Suc *n*]

```
   $\implies \exists xa \in \text{set } xs. x \in (\#) xa \wedge \text{set } (n\text{-sequence-enum } xs \ n) \text{ for } x$ 
```

```
  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 \implies *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* \Rightarrow *nat* \Rightarrow '*a list set* **where**
n-permutations A n = {xs. set xs \subseteq A \wedge distinct xs \wedge length xs = n}

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

Cardinality: '*falling factorial*' (*card A*) *n*

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

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

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

5.2 Algorithm

fun *n-permutation-enum* :: 'a list \Rightarrow nat \Rightarrow 'a list list **where**

n-permutation-enum xs $0 = []$

| *n-permutation-enum* xs (*Suc* n) = $[x\#r \mid x \leftarrow xs, r \leftarrow n\text{-permutation-enum}$
(*remove1* x xs) $n]$

5.3 Verification

5.3.1 Correctness

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

proof (*induct n arbitrary: ys xs*)

case 0

then show *?case* **by** *simp*

next

case (*Suc* n)

obtain x **where** $o1: x \in \text{set } xs$ **and** $o2: ys \in (\#) x \text{ ' set } (n\text{-permutation-enum}$
(*remove1* x xs) n)

using *Suc* **by** *auto*

have $y \in \text{set } (n\text{-permutation-enum } (remove1 \ x \ xs) \ n) \Longrightarrow \text{set } y \subseteq \text{set } xs$ **for** y

using *Suc set-remove1-subset* **by** *fast*

then show *?case* **using** $o1$ $o2$

by *fastforce*

qed

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

by (*induct n arbitrary: ys xs*) *auto*

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

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  $t: zs \in \text{set } (n\text{-permutation-enum } (\text{remove1 } z \text{ } xs) \ n)$ 
  by auto

then have distinct zs
  using Suc distinct-remove1 by fast

also have  $z \notin \text{set } zs$ 
  using  $o \ t \ n\text{-permutation-enum-subset } Suc$  by fastforce

ultimately show ?case
  using  $o$  by simp
qed

lemma n-permutation-enum-correct1:  $\text{distinct } xs \implies \text{set } (n\text{-permutation-enum } xs \ n) \subseteq n\text{-permutations } (\text{set } xs) \ n$ 
  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:  $ys \in n\text{-permutations } (\text{set } xs) \ n \implies ys \in \text{set } (n\text{-permutation-enum } xs \ n)$ 
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:  $z \in \text{set } xs$ 
          using Suc Cons unfolding n-permutations-def by simp

        have  $1$ :  $\text{set } zs \subseteq \text{set } xs$ 
          using Suc Cons unfolding n-permutations-def by simp

        have  $2$ :  $\text{length } zs = n$ 
          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 \in \text{set } zs)
          case True
            then have  $zs \in \text{set } (n\text{-permutation-enum } (\text{remove1 } z \text{ } xs) \ n)$ 
              using Suc Cons unfolding n-permutations-def by auto

```

```

    then show ?thesis
      using True Cons z-in by auto
  next
    case False
    then have  $x \in \text{set } zs \implies x \in \text{set } (\text{remove1 } z \text{ } xs)$  for  $x$ 
      using 1 by(cases  $x = z$ ) auto

    then have  $zs \in n\text{-permutations } (\text{set } (\text{remove1 } z \text{ } xs)) \ n$ 
      unfolding n-permutations-def using 2 3 by auto
    then have  $zs \in \text{set } (n\text{-permutation-enum } (\text{remove1 } z \text{ } xs) \ n)$ 
      using Suc by simp
    then have  $\exists x \in \text{set } xs. z \# zs \in (\#) \ x \ \text{set } (n\text{-permutation-enum } (\text{remove1 } x \text{ } xs) \ n)$ 
      unfolding image-def using z-in by simp
    then show ?thesis
      using False Cons by simp
  qed
qed
qed

```

```

theorem n-permutation-enum-correct:  $\text{distinct } xs \implies \text{set } (n\text{-permutation-enum } xs \ n) = n\text{-permutations } (\text{set } xs) \ n$ 
proof standard
  show  $\text{distinct } xs \implies \text{set } (n\text{-permutation-enum } xs \ n) \subseteq n\text{-permutations } (\text{set } xs) \ n$ 
    by (simp add: n-permutation-enum-correct1)
  next
  show  $\text{distinct } xs \implies n\text{-permutations } (\text{set } xs) \ n \subseteq \text{set } (n\text{-permutation-enum } xs \ n)$ 
    by (simp add: n-permutation-enum-correct2 subsetI)
qed

```

5.3.2 Distinctness

```

theorem n-permutation-distinct:  $\text{distinct } xs \implies \text{distinct } (n\text{-permutation-enum } xs \ n)$ 
proof(induct  $n$  arbitrary:  $xs$ )
  case 0
  then show ?case by simp
  next
  case (Suc  $n$ )
  let ?f =  $\lambda x. (n\text{-permutation-enum } (\text{remove1 } x \text{ } xs) \ n)$ 
  from Suc have  $\text{distinct } (?f \ x)$  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* $A \implies \text{card } (n\text{-permutations } A \ n) = \text{ffact } n \ (\text{card } A)$
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* $:: 'a \text{ multiset} \Rightarrow \text{nat} \Rightarrow 'a \text{ list set}$ **where**
n-multiset-permutations $A \ n = \{xs. \text{mset } xs \subseteq\# A \wedge \text{length } xs = n\}$

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

lemma *distinct* (*n-multiset-permutation-enum* $xs \ n$)
by *auto*

lemma *n-multiset-permutation-enum-correct1*:

$\text{mset } ys \subseteq\# \text{mset } xs \implies ys \in \text{set } (n\text{-permutation-enum } xs \ (\text{length } ys))$

proof(*induct* ys *arbitrary: xs*)

case *Nil*

then show *?case*

by *simp*

next

case (*Cons* $y \ ys$)

then have $y \in \text{set } xs$

by (*simp* *add: insert-subset-eq-iff*)

moreover have $ys \in \text{set } (n\text{-permutation-enum } (\text{remove1 } y \ xs) \ (\text{length } ys))$

using *Cons* **by** (*simp* *add: insert-subset-eq-iff*)

ultimately show *?case*

using *Cons* **by** *auto*

qed

lemma *n-multiset-permutation-enum-correct2*:

$ys \in \text{set } (n\text{-permutation-enum } xs \ n) \implies \text{mset } ys \subseteq\# \text{mset } xs$

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*:

$\text{set } (n\text{-multiset-permutation-enum } xs \ n) = n\text{-multiset-permutations } (\text{mset } xs) \ n$

unfolding *n-multiset-permutations-def*

proof(*standard*)

show $\text{set } (n\text{-multiset-permutation-enum } xs \ n) \subseteq \{xsa. \text{mset } xsa \subseteq\# \text{mset } xs \wedge \text{length } xsa = n\}$

```

  by (simp add: n-multiset-permutation-enum-correct2 n-permutation-enum-length
subsetI)
next
  show {xsa. mset xsa  $\subseteq$  # mset xs  $\wedge$  length xsa = n}  $\subseteq$  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  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  filter-bool-list [] - = []
| filter-bool-list - [] = []
| filter-bool-list (b#bs) (x#xs) =
  (if b then x#(filter-bool-list bs xs) else (filter-bool-list bs xs))

```

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

```

lemma filter-bool-list bs xs = map snd (filter fst (zip bs xs))
  by(induct bs xs rule: filter-bool-list.induct) auto

```

```

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

```

```

lemma filter-bool-list-not-elem: x  $\notin$  set xs  $\implies$  x  $\notin$  set (filter-bool-list bs xs)
  by(induct bs xs rule: filter-bool-list.induct) auto

```

```

lemma filter-bool-list-elem: x  $\in$  set (filter-bool-list bs xs)  $\implies$  x  $\in$  set xs
  using filter-bool-list-not-elem by fast

```

```

lemma filter-bool-list-not-in:
  distinct xs  $\implies$  n < length xs  $\implies$  n < length bs  $\implies$  bs!n = False
   $\implies$  xs!n  $\notin$  set (filter-bool-list bs xs)
proof (induct bs xs arbitrary: n rule: filter-bool-list.induct)
  case ( $\exists$  b bs x xs)
  then show ?case proof(induct n)
    case 0

```

```

    then show ?case using filter-bool-list-not-elim
      by force
  qed auto
qed auto

```

```

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

```

```

lemma filter-bool-list-exist-length:  $A \subseteq \text{set } xs$ 
   $\implies \exists bs. \text{length } bs = \text{length } xs \wedge A = \text{set } (\text{filter-bool-list } bs \ xs)$ 
proof(induct xs arbitrary: A)
  case Nil
  then show ?case
    by auto
next
  case (Cons x xs)
  from Cons have  $A - \{x\} \subseteq \text{set } xs$ 
    by auto
  from this Cons have 1:  $\exists bs. \text{length } bs = \text{length } xs \wedge A - \{x\} = \text{set } (\text{filter-bool-list } bs \ xs)$ 
    by simp

  then have  $\exists bs. \text{length } bs = \text{length } (x \# xs) \wedge A = \text{set } (\text{filter-bool-list } bs \ (x \# xs))$ 
    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:
   $[[\text{distinct } xs; \text{length } xs = \text{length } bs]] \implies \text{card } (\text{set } (\text{filter-bool-list } bs \ xs)) = \text{count-list}$ 

```



```

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

lemma filter-bool-list-distinct: distinct xs  $\implies$  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  $\implies$  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  $\implies$  inj-on ( $\lambda bs. \text{filter-bool-list } bs \text{ xs}$ )  $\{\text{bs}. \text{length } bs = \text{length } xs\}$ 
  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  $\Rightarrow$  nat  $\Rightarrow$  'a set set where
  n-subsets A n = {B. B  $\subseteq$  A  $\wedge$  card B = n}

```

Cardinality: *binomial (card A) n*

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

6.2 Algorithm

```

fun n-bool-lists :: nat  $\Rightarrow$  nat  $\Rightarrow$  bool list list where
  n-bool-lists n 0 = (if n > 0 then [] else [[]])
| n-bool-lists n (Suc x) = (if n = 0 then [replicate (Suc x) False]
  else if n = Suc x then [replicate (Suc x) True]
  else if n > x then []
  else [False#xs . xs  $\leftarrow$  n-bool-lists n x] @ [True#xs . xs  $\leftarrow$  n-bool-lists (n-1) x])

```

```

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

```

6.3 Verification

6.3.1 n-bool-lists

```

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

```

proof(standard)
show  $\text{set } (n\text{-bool-lists } n \ x) \subseteq \{xs. \text{length } xs = x \wedge \text{count-list } xs \ \text{True} = n\}$ 
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-elim)

```

```

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-elim:
  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

```

$n\text{-bool-lists2 } n \ x = (\text{if } n > x \text{ then } \{\})$
 $\text{else permutations-of-multiset } (\text{mset } (\text{replicate } n \ \text{True} \ @ \ \text{replicate } (x-n) \ \text{False}))$

6.5 *mset-count*

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

lemma *count-list-count-mset*: $\text{count-list } ys \ T = n \implies \text{count } (\text{mset } ys) \ T = n$
by(*induct ys arbitrary: n*) *auto*

lemma *count-mset-count-list*: $\text{count } (\text{mset } ys) \ T = n \implies \text{count-list } ys \ T = n$
by(*induct ys arbitrary: n*) *auto*

lemma *count-mset-replicate-aux1*:
 $\llbracket \neg x < n; \text{mset } ys = \text{mset } (\text{replicate } n \ \text{True}) + \text{mset } (\text{replicate } (x - n) \ \text{False}) \rrbracket$
 $\implies \text{count } (\text{mset } ys) \ \text{True} = n$
by (*auto simp: count-list-count-mset count-mset*)

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

proof –

have $\text{count-list } xs \ B =$
 $\text{count-list } (\text{replicate } (\text{count-list } xs \ \text{True}) \ \text{True}) \ B + \text{count-list } (\text{replicate } (\text{length } xs - \text{count-list } xs \ \text{True}) \ \text{False}) \ B$

for B

proof(*cases B*)

case True

then show *?thesis*

by (*simp add: count-list-replicate*)

next

case False

have $\text{count-list } xs \ \text{False} = \text{count-list } (\text{replicate } (\text{length } xs - \text{count-list } xs \ \text{True}) \ \text{False}) \ \text{False}$

by (*metis count-list-True-False count-list-replicate diff-add-inverse*)

from this False show *?thesis*

using *assms* **by** *auto*

qed

then have $\text{count } (\text{mset } xs) \ B =$
 $\text{count } (\text{mset } (\text{replicate } (\text{count-list } xs \ \text{True}) \ \text{True}) + \text{mset } (\text{replicate } (\text{length } xs - \text{count-list } xs \ \text{True}) \ \text{False})) \ B$

for B

by (*metis count-mset-count-list count-union*)

then show $\text{mset } xs = \text{mset } (\text{replicate } (\text{count-list } xs \ \text{True}) \ \text{True}) + \text{mset } (\text{replicate } (\text{length } xs - \text{count-list } xs \ \text{True}) \ \text{False})$

```

(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  $\llbracket \neg \text{length } ys < \text{count-list } ys \text{ True}; x = \text{length } ys; n = \text{count-list } ys \text{ True} \rrbracket$ 
     $\implies ys \in \text{permutations-of-multiset}$ 
      (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)  $\subseteq$  n-bool-lists2 n x
    unfolding n-bool-lists-correct
    by (auto simp: count-le-length leD)
next
  have  $\llbracket \neg x < n; ys \in \text{permutations-of-multiset} (\text{mset} (\text{replicate } n \text{ True}) + \text{mset} (\text{replicate} (x - n) \text{ False})) \rrbracket$ 
     $\implies \text{count} (\text{mset } ys) \text{ True} = n$  for ys
  using count-mset-replicate-aux1 permutations-of-multisetD by blast
  then have  $\llbracket \neg x < n; ys \in \text{permutations-of-multiset} (\text{mset} (\text{replicate } n \text{ True}) + \text{mset} (\text{replicate} (x - n) \text{ False})) \rrbracket$ 
     $\implies \text{count-list } ys \text{ True} = n$  for ys
  by (simp add: count-list-count-mset)
  then show n-bool-lists2 n x  $\subseteq$  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: $2^{\wedge} \text{card } A$

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

7.2 Algorithm

```

fun all-bool-lists :: nat  $\Rightarrow$  bool list list where
  all-bool-lists 0 = [[]]
| all-bool-lists (Suc x) = concat [[False#xs, True#xs] . xs  $\leftarrow$  all-bool-lists x]

fun powerset-enum where
  powerset-enum xs = [(filter-bool-list x xs) . x  $\leftarrow$  all-bool-lists (length xs)]

```

7.3 Verification

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

```

lemma distinct-concat-aux: distinct xs  $\implies$  distinct (concat (map ( $\lambda$ xs. [False #
xs, True # xs] xs)))
  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)  $\subseteq$  {xs. length xs = x}

```

```

    by (induct x) auto

```

```

next

```

```

  show {xs. length xs = x}  $\subseteq$  set (all-bool-lists x)

```

```

  proof(induct x)

```

```

    case 0

```

```

    then show ?case by simp

```

```

  next

```

```

    case (Suc x)

```

```

    have length ys = Suc x  $\implies$   $\exists$  xs. ys = False # xs  $\vee$  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))  $\subseteq$  Pow (set xs)

```

```

    using filter-bool-list-not-elem by fastforce

```

```

next

```

```

  have  $\bigwedge$ x. x  $\subseteq$  set xs  $\implies$  x  $\in$  ( $\lambda$ x. 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)  $\subseteq$  set (map set (powerset-enum xs))

```

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

7.3.2 Distinctness

theorem *powerset-enum-distinct-elem*: $distinct\ xs \implies ys \in set\ (powerset-enum\ xs) \implies distinct\ ys$
 using *filter-bool-list-distinct* by *auto*

theorem *powerset-enum-distinct*: $distinct\ xs \implies distinct\ (powerset-enum\ xs)$

proof –

assume *dis*: *distinct xs*

then have $distinct\ (map\ (\lambda x. filter-bool-list\ x\ xs)\ (all-bool-lists\ (length\ xs)))$

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* :: $nat \Rightarrow bool\ list\ list$ **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 Partitions

theory *Integer-Partitions*

imports

HOL-Library.Multiset

Common-Lemmas

Card-Number-Partitions.Card-Number-Partitions

begin

8.1 Definition

definition *integer-partitions* :: $nat \Rightarrow nat\ multiset\ set$ **where**
integer-partitions *i* = $\{A. sum-mset\ A = i \wedge 0 \notin\# A\}$

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:
```

$xs \in \text{set } (\text{integer-partitions-enum-aux } n \ m) \implies \text{sum-list } xs = n$
proof(*induct xs arbitrary: n m*)
 case *Nil*
 then show ?case **using** *integer-partitions-empty* **by** *simp*
next
 case (*Cons x xs*)
 then have $\llbracket xs \in \text{set } (\text{integer-partitions-enum-aux } (n-x) \ x) \rrbracket \implies \text{sum-list } xs =$
 $(n-x)$
 by *simp*
 moreover have $xs \in \text{set } (\text{integer-partitions-enum-aux } (n-x) \ x)$
 using *Cons integer-partitions-enum-aux-first* **by** *simp*
 moreover have $x \leq n$
 using *Cons integer-partitions-enum-aux-max-n* **by** *simp*
 ultimately show ?case
 by *simp*
qed

lemma *integer-partitions-enum-aux-not-null-aux*:
 $x \# xs \in \text{set } (\text{integer-partitions-enum-aux } n \ m) \implies x \neq 0$
by (*induct n*) *auto*

lemma *integer-partitions-enum-aux-not-null*:
 $x \in \text{set } (\text{integer-partitions-enum-aux } n \ m) \implies x \in \text{set } xs \implies x \neq 0$
proof(*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*:
 $h \leq m \implies h > 0 \implies n \geq h \implies$
 $ys \in \text{set } (\text{integer-partitions-enum-aux } (n-h) \ h) \implies h \# ys \in \text{set } (\text{integer-partitions-enum-aux } n \ m)$
proof(*induct n*)
 case *0*
 then show ?case **by** *simp*
next
 case (*Suc n*)

then have $1: 1 \leq m$ **by** *simp*

have $2: (\exists x. (x = \min (\text{Suc } n) m \vee \text{Suc } 0 \leq x \wedge x < \text{Suc } n \wedge x < m) \wedge h \# ys \in (\#) x \text{ ' set } (\text{integer-partitions-enum-aux } (\text{Suc } n - x) x))$
unfolding *image-def* **using** *Suc* **by** *auto*

from $1\ 2$ **have** $\text{Suc } 0 \leq m \wedge (\exists x. (x = \min (\text{Suc } n) m \vee \text{Suc } 0 \leq x \wedge x < \text{Suc } n \wedge x < m) \wedge h \# ys \in (\#) x \text{ ' set } (\text{integer-partitions-enum-aux } (\text{Suc } n - x) x))$
by *simp*

then show *?case* **by** *auto*
qed

lemma *integer-partitions-enum-aux-head-plus*:
 $h \leq m \implies h > 0 \implies ys \in \text{set } (\text{integer-partitions-enum-aux } n\ h) \implies h \# ys \in \text{set } (\text{integer-partitions-enum-aux } (h + n)\ m)$
using *integer-partitions-enum-aux-head-minus* **by** *simp*

lemma *integer-partitions-enum-correct-aux1*:
assumes $0 \notin \# A$
and $\forall x \in \# A. x \leq m$
shows $\exists xs \in \text{set } (\text{integer-partitions-enum-aux } (\sum \# A)\ m). A = \text{mset } xs$
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*: $h \leq m$
using *add* **by** *simp*

have *hc2*: $h > 0$
using *add* **by** *simp*

obtain *ys* **where** *o1*: $ys \in \text{set } (\text{integer-partitions-enum-aux } (\sum \# A)\ h)$ **and** *o2*:
 $A = \text{mset } ys$
using *add* **by** *force*

have $h \# ys \in \text{set } (\text{integer-partitions-enum-aux } (h + \sum \# A)\ m)$
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*:
 $\text{set } (\text{map } \text{mset } (\text{integer-partitions-enum } n)) = \text{integer-partitions } n$
proof(*standard*)
have $\llbracket xs \in \text{set } (\text{integer-partitions-enum-aux } n\ n) \rrbracket \implies \sum \# (\text{mset } xs) = n$ **for** *xs*

by (simp add: integer-partitions-enum-aux-sum sum-mset-sum-list)
 moreover have $xs \in \text{set } (\text{integer-partitions-enum-aux } n \ n) \implies 0 \notin \# \text{ mset } xs$
 for xs
 using integer-partitions-enum-aux-not-null by auto
 ultimately show $\text{set } (\text{map mset } (\text{integer-partitions-enum } n)) \subseteq \text{integer-partitions } n$
 unfolding integer-partitions-def by auto
 next
 have $0 \notin \# A \implies A \in \text{mset ' set } (\text{integer-partitions-enum-aux } (\sum \# A) (\sum \# A))$ for A
 unfolding image-def
 using integer-partitions-enum-correct-aux1 by (simp add: sum-mset.remove)
 then show $\text{integer-partitions } n \subseteq \text{set } (\text{map mset } (\text{integer-partitions-enum } n))$
 unfolding integer-partitions-def by auto
 qed

8.3.2 Distinctness

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

theorem integer-partitions-enum-distinct:
 $\text{distinct } (\text{integer-partitions-enum } n)$
 using integer-partitions-enum-aux-distinct by simp

8.3.3 Cardinality

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

lemma card-partitions-count-partitions:
 $\text{card } \{p. \text{p partitions } n\} = \text{card } \{N. \text{count } N \text{ partitions } n\}$
 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:
 $\text{card } \{p. \text{p partitions } n\} = \text{card } \{N. \text{number-partition } n \ N\}$
 using card-partitions-count-partitions count-partitions-iff by simp

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

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

theorem *integer-partitions-cardinality*:
 $\text{card } (\text{integer-partitions } n) = \text{Partition } (2 * n)\ n$
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* :: $\text{nat} \Rightarrow \text{nat list set}$ **where**
integer-compositions $i = \{xs. \text{sum-list } xs = i \wedge 0 \notin \text{set } xs\}$

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* :: $\text{nat} \Rightarrow \text{nat list list}$ **where**
integer-composition-enum $0 = []$
| *integer-composition-enum* $(\text{Suc } n) =$
 $[\text{Suc } m \ \# \ xs. m \leftarrow [0..< \text{Suc } n], xs \leftarrow \text{integer-composition-enum } (n-m)]$

9.3 Verification

9.3.1 Correctness

lemma *integer-composition-enum-tail-elem*:
 $x \# xs \in \text{set } (\text{integer-composition-enum } n) \implies xs \in \text{set } (\text{integer-composition-enum } (n - x))$

```

by(induct n) auto

lemma integer-composition-enum-not-null-aux:
   $x\#xs \in \text{set } (\text{integer-composition-enum } n) \implies x \neq 0$ 
  by(induct n) auto

lemma integer-composition-enum-not-null:  $xs \in \text{set } (\text{integer-composition-enum } n)$ 
 $\implies 0 \notin \text{set } xs$ 
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-enum
    by fastforce
qed

lemma integer-composition-enum-empty:  $\square \in \text{set } (\text{integer-composition-enum } n)$ 
 $\implies n = 0$ 
  by(induct n) auto

lemma integer-composition-enum-sum:  $xs \in \text{set } (\text{integer-composition-enum } n) \implies$ 
 $\text{sum-list } xs = n$ 
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:  $\text{sum-list } ys = \text{Suc } x - y$  using 2 Cons
      by auto

    have  $\text{Suc } x \geq y$ 
      using 2 Cons by auto
    then have p2:  $\text{sum-list } ys = \text{Suc } x - y \implies y + \text{sum-list } ys = \text{Suc } x$ 
      by simp

    show ?thesis
      using p1 p2 Cons by simp
  qed
qed

```

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

```

assumes  $x \neq 0$  and  $x \leq n$ 
shows  $xs \in \text{set } (\text{integer-composition-enum } (n-x)) \implies x\#xs \in \text{set } (\text{integer-composition-enum } n)$ 
using assms proof(induct n arbitrary: x xs)
  case 0
  then show ?case
    by simp
next
  case c1: (Suc n)
  from c1.prem1 have 1:
     $\forall y \in \{0..<n\}. x = \text{Suc } y \longrightarrow xs \notin \text{set } (\text{integer-composition-enum } (n - y)) \implies x = \text{Suc } n$ 
    by(induct x) simp-all

  then have 2:  $\forall y \in \{0..<n\}. x = \text{Suc } y \longrightarrow xs \notin \text{set } (\text{integer-composition-enum } (n - y)) \implies xs = []$ 
    using c1.prem1 by simp
  show ?case using 1 2 by auto
qed

```

```

lemma integer-composition-enum-correct-aux:
   $0 \notin \text{set } xs \implies xs \in \text{set } (\text{integer-composition-enum } (\text{sum-list } xs))$ 
  by(induct xs) (auto simp: integer-composition-enum-head-set)

```

```

theorem integer-composition-enum-correct:
   $\text{set } (\text{integer-composition-enum } n) = \text{integer-compositions } n$ 
proof standard
  show  $\text{set } (\text{integer-composition-enum } n) \subseteq \text{integer-compositions } n$ 
    unfolding integer-compositions-def
    using integer-composition-enum-not-null integer-composition-enum-sum by auto
next
  show  $\text{integer-compositions } n \subseteq \text{set } (\text{integer-composition-enum } n)$ 
    unfolding integer-compositions-def
    using integer-composition-enum-correct-aux by auto
qed

```

9.3.2 Distinctness

```

theorem integer-composition-enum-distinct:
   $\text{distinct } (\text{integer-composition-enum } n)$ 
proof(induct n rule: integer-composition-enum.induct)
  case 1
  then show ?case by auto
next
  case (2 n)

  have h1:  $x \in \text{set } [0..<\text{Suc } n] \implies \text{distinct } (\text{integer-composition-enum } (n - x))$ 
for x

```

using 2 **by** *simp*
have $h2$: *distinct* $[0..<n]$
by *simp*
have *distinct* $[Suc\ m\ \#\ xs.\ m\ \leftarrow\ [0..<n],\ xs\ \leftarrow\ integer-composition-enum\ (n-m)]$
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*:

$$\left(\sum_{x \leftarrow [0..<n]} (2::nat) ^ (n - x)\right) + 2 ^ (0 - 1) + 2 ^ 0 = 2 ^ (Suc\ n)$$

proof (*induct* n)

case 0

then show *?case* **by** *simp*

next

case $c1$: $(Suc\ n)$

have $x \leq n \implies 2 ^ (Suc\ n - x) = 2 * 2 ^ (n - x)$ **for** x

by (*simp* *add: Suc-diff-le*)

also have $x \in set\ [0..<Suc\ n] \implies x \leq n$ **for** x

by *auto*

ultimately have $\left(\sum_{x \leftarrow [0..<Suc\ n]} 2 ^ (Suc\ n - x)\right) = \left(\sum_{x \leftarrow [0..<Suc\ n]} 2 * 2 ^ (n - x)\right)$

by (*metis* (*mono-tags*, *lifting*) *map-eq-conv*)

also have $\dots = \left(\sum_{x \leftarrow [0..<n]} 2 * 2 ^ (n - x)\right) + 2 * 2 ^ (0)$

using *sum-list-extract-last* **by** *simp*

also have $\left(\sum_{x \leftarrow [0..<n]} (2::nat) * (2::nat) ^ (n - x)\right) = 2 * \left(\sum_{x \leftarrow [0..<n]} 2 ^ (n - x)\right)$

using *sum-list-const-mult* **by** *fast*

ultimately have $\left(\sum_{x \leftarrow [0..<Suc\ n]} (2::nat) ^ (Suc\ n - x)\right)$

$$= 2 * \left(\sum_{x \leftarrow [0..<n]} 2 ^ (n - x)\right) + 2 * 2 ^ (0)$$

by *metis*

then show *?case* **using** $c1$

by *simp*

qed

lemma *sum-list-two-pow*:

$$Suc\ \left(\sum_{x \leftarrow [0..<n]} 2 ^ (n - Suc\ x)\right) = 2 ^ n$$

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)]
```

$$= (\sum_{x \leftarrow [0..<n]}. 2^{(n-x-1)})$$

```

  using length-concat-map-function-sum-list [of
  [0..<n]
   $\lambda x. integer-composition-enum (n-x)$ 
   $\lambda x. 2^{(n-x-1)}$ 
   $\lambda m xs. Suc m \#xs]$ 
  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  $\Rightarrow$  nat  $\Rightarrow$  nat list set where
  weak-integer-compositions i l = {xs. length xs = l  $\wedge$  sum-list xs = i}

```

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

Cardinality: *binomial (i + n - 1) i*

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

10.2 Algorithm

```

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

```

```

  weak-integer-composition-enum i 0 = (if i = 0 then [[]] else [])
| weak-integer-composition-enum i (Suc 0) = [[i]]
| weak-integer-composition-enum i l =
  [h#r . h ← [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:*

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

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:*

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

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

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

proof(*cases length xs*)

case 0

then show ?thesis **by** *simp*

next

case (Suc y)

have 1: $\llbracket n \in \text{set } xs; 0 < n \rrbracket \implies 0 < \text{sum-list } xs$ **for** n

using *sum-list-eq-0-iff* **by** *fast*

have 2: $xs \notin \text{set } (\text{weak-integer-composition-enum } 0 \ (\text{Suc } y)) \implies 0 < \text{sum-list } xs$

using *Suc assms not-gr0* **by** *fastforce*

have $x \# xs \notin (\#) (x + \text{sum-list } xs) \wedge \text{set } (\text{weak-integer-composition-enum } 0 \ (\text{Suc } y))$

$\implies \exists xa \in \{0..<x + \text{sum-list } xs\}. x \# xs \in (\#) xa \text{ 'set (weak-integer-composition-enum (x + sum-list xs - xa) (Suc y))}$

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:*

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

by (*induct xs*) (*auto simp: weak-integer-composition-enum-head*)

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

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

proof *standard*

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

unfolding *weak-integer-compositions-def*

using *weak-integer-composition-enum-length weak-integer-composition-enum-sum-list*

by *auto*

next

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

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*:: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ (**infixl** *multichoose* 65) **where**
 $n \text{ multichoose } k = (n + k - 1) \text{ choose } k$

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

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

proof -

have $m \geq k \implies (\sum x \leftarrow [0..< \text{Suc } k]. m - x \text{ choose } (k - x)) = \text{Suc } m \text{ choose } k$
for m

using *sum-choose-diagonal leq-sum-to-sum-list* **by** *metis*

then have 1: $\text{Suc } (\sum x \leftarrow [0..<k]. (n + k) - x \text{ choose } (k - x)) = \text{Suc } (n + k) \text{ choose } k$

by *simp*

have $\text{Suc } (\sum x \leftarrow [0..<k]. (n + k) - x \text{ choose } (k - x)) = \text{Suc } (\sum x \leftarrow [0..<k]. n + (k - x) \text{ choose } (k - x))$

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

unfolding *multichoose-def*

proof(*induct* $i \text{ } n$ *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 $i \text{ } n$)

then have $x \in \text{set } [0..< i] \implies$

$\text{length } (\text{weak-integer-composition-enum } (i - x) \text{ (Suc } n)) = n + (i - x) \text{ choose } (i - x)$ **for** x

by *simp*

then have *ev*: $\text{length } [h\#r . h \leftarrow [0..< i], r \leftarrow \text{weak-integer-composition-enum } (i-h) \text{ (Suc } n)] =$

$(\sum x \leftarrow [0..< i]. n + (i - x) \text{ choose } (i - x))$

```

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

```

$derangements\ xs = \{ys.\ distinct\ ys \wedge length\ xs = length\ ys \wedge set\ xs = set\ ys \wedge no-overlap\ xs\ ys\}$

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  $\Rightarrow$  'a list  $\Rightarrow$  'a list list where
  derangement-enum-aux [] ys = [[]]
| derangement-enum-aux (x#xs) ys = [y#r . y  $\leftarrow$  ys, r  $\leftarrow$  derangement-enum-aux
  xs (remove1 y ys), y  $\neq$  x]
```

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

11.3 Verification

11.3.1 Correctness

```
lemma derangement-enum-aux-elem-length:  $zs \in set\ (derangement-enum-aux\ xs\ ys) \Longrightarrow length\ xs = length\ zs$ 
by(induct xs arbitrary: ys zs) auto
```

```
lemma derangement-enum-aux-not-in:  $y \notin set\ ys \Longrightarrow zs \in set\ (derangement-enum-aux\ xs\ ys) \Longrightarrow y \notin set\ zs$ 
```

```
proof(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  $\zs2 \in set\ (derangement-enum-aux\ xs\ (remove1\ z\ ys)) \Longrightarrow y \notin set\ \zs2$ 
```

```
    using Cons notin-set-remove1 by fast
```

```
  then show ?case using Cons ob
```

```
    by auto
```

```
qed
```

```
lemma derangement-enum-aux-in:  $y \in set\ zs \Longrightarrow zs \in set\ (derangement-enum-aux\ xs\ ys) \Longrightarrow y \in set\ ys$ 
```

```
  using derangement-enum-aux-not-in by fast
```

```
lemma derangement-enum-aux-distinct-elem:  $distinct\ ys \Longrightarrow zs \in set\ (derangement-enum-aux\ xs\ ys) \Longrightarrow distinct\ zs$ 
```

```

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)
   $\implies$  no-overlap xs zs
  by(induct xs arbitrary: zs ys) auto

```

```

lemma derangement-enum-aux-set:
  length xs = length ys  $\implies$  zs ∈ set (derangement-enum-aux xs ys)  $\implies$  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.prem1 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:
   $\llbracket$ distinct zs; length ys = length zs; length ys = length xs; set ys = set zs; no-overlap xs zs $\rrbracket$ 
   $\implies$  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.prem5 ob by auto

  have distinct zs2
    using Cons.prem1 ob by auto
  moreover have length (remove1 z ys) = length zs2 using Cons.prem ob
    by (simp add: length-remove1)
  moreover have length (remove1 z xs) = length xs
    by (simp add: Cons.prem3 Cons.prem4 length-remove1 ob)
  moreover have set (remove1 z ys) = set zs2
    using Cons ob by (metis distinct-card distinct-remdups length-remdups-eq remove1.simp2 set-remdups set-remove1-eq)
  moreover have no-overlap xs zs2
    using Cons.prem5 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
       $\lambda y. \text{derangement-enum-aux } xs \text{ (remove1 } y \text{ } ys)$ 
       $\lambda y. y \neq x$ 
    ]
    using Cons Cons-inj2
    by (simp)
qed

theorem derangement-enum-distinct:  $\text{distinct } xs \implies \text{distinct (derangement-enum } xs)$ 
  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 ::  $\text{nat} \Rightarrow \text{unit tree set}$  where
  trees  $n = \{t. \text{size } t = n\}$ 

```

Cardinality: *Catalan number of n*

Example: *trees 0 = {Leaf}*

12.2 Algorithm

```

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

```

12.3 Verification

12.3.1 Cardinality

lemma *length-tree-enum*:

$\text{length } (\text{tree-enum}(\text{Suc } n)) = (\sum i \leq n. \text{length}(\text{tree-enum } i) * \text{length}(\text{tree-enum } (n - i)))$

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*: $t \in \text{set } (\text{tree-enum } n) \implies \text{size } t = n$

by (*induct n arbitrary: t rule: tree-enum.induct*) (*simp, fastforce*)

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

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.prem*s)

next

case (*Node l e r*)

have *i1*: $(\text{size } l) < \text{Suc } n$ **using** *2.prem*s *Node* **by** *auto*

have *i2*: $(\text{size } r) < \text{Suc } n$ **using** *2.prem*s *Node* **by** *auto*

have *t1*: $l \in \text{set } (\text{tree-enum } (\text{size } l))$

apply(*rule 2.hyps(1) [of (size l)]*)

using *i1* **by** *auto*

have *t2*: $r \in \text{set } (\text{tree-enum } (\text{size } r))$

apply(*rule 2.hyps(1) [of (size r)]*)

using *i2* **by** *auto*

have $\langle l, (), r \rangle \notin (\lambda t1. \langle t1, (), \langle \rangle \rangle) \text{ `set } (\text{tree-enum } (\text{size } l + \text{size } r)) \implies$

$\exists x \in \{0..<\text{size } l + \text{size } r\}. \exists xa \in \text{set } (\text{tree-enum } x). \langle l, (), r \rangle \in \text{Node } xa () \text{ `}$
set } (\text{tree-enum } (\text{size } l + \text{size } r - x))

using *t1 t2* **by** *fastforce*

then have $\langle l, e, r \rangle \in \text{set } (\text{tree-enum } (\text{size } \langle l, e, r \rangle))$

by *auto*

then show *?thesis*

using *Node* **using** *2.prem*s **by** *simp*

qed

qed

```

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

```

12.3.3 Distinctness

```

lemma tree-enum-Leaf:  $\langle \rangle \in \text{set (tree-enum n)} \iff (n = 0)$ 
by(cases n) auto

```

```

lemma tree-enum-elem-injective:  $n \neq m \implies x \in \text{set (tree-enum n)} \implies y \in \text{set (tree-enum m)} \implies x \neq y$ 
using tree-enum-correct1 by auto

```

```

lemma tree-enum-elem-injective2:  $x \in \text{set (tree-enum n)} \implies y \in \text{set (tree-enum m)} \implies x = y \implies n = m$ 
using tree-enum-elem-injective by auto

```

```

lemma concat-map-Node-not-equal:

```

```

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

```

```

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

```

```

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

```

```

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.prem3) neq-Nil-conv)

```

```

          obtain y2 y2s where o2:  $ys2 = y2 \# y2s$ 

```

```

            by (meson Cons.prem5) neq-Nil-conv)

```

```

          have  $[\langle l, (), r \rangle. l \leftarrow x2 \# x2s, r \leftarrow x \# xs] \neq [\langle l, (), r \rangle. l \leftarrow y2 \# y2s, r \leftarrow y \# ys]$ 

```

```

            using Cons.prem6 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.prem1(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.prem1 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:  $\llbracket \bigwedge t n. t \in \text{set } (\text{tree-enum } n) \implies \text{size } t = n; m < n; y \in \text{set } (\text{tree-enum } n); y \in \text{set } (\text{tree-enum } m) \rrbracket \implies \text{False}$  for  $m y$ 
by blast

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

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

  have c1:  $\text{distinct } (\text{concat } (\text{map } (\lambda i. [\langle l, (), r \rangle. l \leftarrow \text{tree-enum } i, r \leftarrow \text{tree-enum } (n-i)] [0..<n])))$ 
  proof(rule distinct-concat)
    show  $\text{distinct } (\text{map } (\lambda i. [\langle l, (), r \rangle. l \leftarrow \text{tree-enum } i, r \leftarrow \text{tree-enum } (n-i)] [0..<n]))$ 
    apply(rule tree-enum-distinct-aux-outer)
    using Ir by auto
  next
  have  $\bigwedge x. x < n \implies \text{distinct } ([\langle l, (), r \rangle. l \leftarrow \text{tree-enum } x, r \leftarrow \text{tree-enum } (n-x)])$ 
  using Ir by (simp add: Node-right-distinct-concat-map)
  then show  $\bigwedge ys. ys \in \text{set } (\text{map } (\lambda i. [\langle l, (), r \rangle. l \leftarrow \text{tree-enum } i, r \leftarrow \text{tree-enum } (n-i)] [0..<n])) \implies \text{distinct } ys$ 
  by auto
  next
  have  $\llbracket [\langle l, (), r \rangle. l \leftarrow \text{tree-enum } x, r \leftarrow \text{tree-enum } (n-x)] \neq [\langle l, (), r \rangle. l \leftarrow \text{tree-enum } z, r \leftarrow \text{tree-enum } (n-z)]; y \in \text{set } (\text{tree-enum } x); y \in \text{set } (\text{tree-enum } z) \rrbracket \implies \text{False}$  for  $x z y$ 
  using tree-enum-elem-injective2 by auto
  then show  $\bigwedge ys zs. \llbracket ys \in \text{set } (\text{map } (\lambda i. [\langle l, (), r \rangle. l \leftarrow \text{tree-enum } i, r \leftarrow \text{tree-enum } (n-i)] [0..<n])); zs \in \text{set } (\text{map } (\lambda i. [\langle l, (), r \rangle. l \leftarrow \text{tree-enum } i, r \leftarrow \text{tree-enum } (n-i)] [0..<n]); ys \neq zs \rrbracket \implies \text{set } ys \cap \text{set } zs = \{\}$ 
  by fastforce
qed

have  $\text{distinct } (\text{tree-enum } n)$ 

```

```

using 2 by simp
then have c2: distinct (map ( $\lambda t1. \langle t1, (), \langle \rangle \rangle$ ) (tree-enum n))
using Node-left-distinct-map by fastforce

have c3:  $\bigwedge xa\ xb. \llbracket xa < n; xb \in \text{set } (tree\text{-enum } xa); xb \in \text{set } (tree\text{-enum } n); \langle \rangle \in \text{set } (tree\text{-enum } (n - xa)) \rrbracket \implies \text{False}$ 
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.