The Falling Factorial of a Sum

Lukas Bulwahn

March 17, 2025

Abstract

This entry shows that the falling factorial of a sum can be computed with an expression using binomial coefficients and the falling factorial of its summands. The entry provides three different proofs: a combinatorial proof, an induction proof and an algebraic proof using the Vandermonde identity.

The three formalizations try to follow their informal presentations from a Mathematics Stack Exchange page [1, 2, 3, 4] as close as possible. The induction and algebraic formalization end up to be very close to their informal presentation, whereas the combinatorial proof first requires the introduction of list interleavings, and significant more detail than its informal presentation.

Contents

1 Proving Falling Factorial of a Sum with Combinatorics

theory *Falling-Factorial-Sum-Combinatorics*
imports

Discrete-Summation.Factorials
Card-Partitions.Injectivity-Solver
begin

1.1 Preliminaries

1.1.1 Addition to Factorials Theory

lemma *card-lists-distinct-length-eq*:
assumes *finite A*
shows *card* {*xs*. *length xs* = *n* ∧ *distinct xs* ∧ *set xs* ⊆ *A*} = *ffact n* (*card A*)
proof *cases*
assume *n* ≤ *card A*
have *card* {*xs*. *length xs* = *n* ∧ *distinct xs* ∧ *set xs* ⊆ *A*} = ∏{*card A* − *n* + *1*..*card A*}
using ‹*finite A*› ‹*n* ≤ *card A*› **by** (*rule card-lists-distinct-length-eq*)
also have ... = *ffact n* (*card A*)
using ‹*n* ≤ *card A*› **by** (*simp add*: *prod-rev-ffact-nat′*[*symmetric*])
finally show *?thesis* **.**
next
assume ¬ *n* ≤ *card A*
from *this* ‹*finite A*› **have** ∀ *xs*. *length xs* = *n* ∧ *distinct xs* ∧ *set xs* ⊆ *A* ⟶ *False*
by (*metis card-mono distinct-card*)
from *this* **have** *eq-empty*: {*xs*. *length xs* = *n* ∧ *distinct xs* ∧ *set xs* ⊆ *A*} = {}
using ‹*finite A*› **by** *auto*
from ‹¬ *n* ≤ *card A*› **show** *?thesis*
by (*simp add*: *ffact-nat-triv eq-empty*)
qed

1.2 Interleavings of Two Lists

inductive *interleavings* :: ′*a list* ⇒ ′*a list* ⇒ ′*a list* ⇒ *bool*
where
interleavings [] *ys ys*
| *interleavings xs* [] *xs*
| *interleavings xs ys zs* ⟹ *interleavings* (*x*#*xs*) *ys* (*x*#*zs*)
| *interleavings xs ys zs* ⟹ *interleavings xs* (*y*#*ys*) (*y*#*zs*)

lemma *interleaving-Nil-implies-eq1*:
assumes *interleavings xs ys zs*
assumes *xs* = []
shows *ys* = *zs*
using *assms* **by** (*induct rule*: *interleavings.induct*) *auto*

lemma *interleaving-Nil-iff1*:
interleavings [] *ys zs* ⟷ (*ys* = *zs*)
using *interleaving-Nil-implies-eq1*
by (*auto simp add*: *interleavings.intros*(*1*))

lemma *interleaving-Nil-implies-eq2*:
  assumes *interleavings xs ys zs*
  assumes *ys* = []
  shows *xs* = *zs*
using *assms* **by** (*induct rule*: *interleavings.induct*) *auto*

lemma *interleaving-Nil-iff2*:
  interleavings xs [] *zs* ⟷ (*xs* = *zs*)
using *interleaving-Nil-implies-eq2*
by (*auto simp add*: *interleavings.intros*(*2*))

lemma *interleavings-Cons*:
  {*zs*. *interleavings* (*x*#*xs*) (*y*#*ys*) *zs*} =
    {*x*#*zs*|*zs*. *interleavings xs* (*y*#*ys*) *zs*} ∪ {*y*#*zs*|*zs*. *interleavings* (*x*#*xs*) *ys zs*}
  (**is** *?S* = *?expr*)
proof
  show *?S* ⊆ *?expr*
    by (*auto elim*: *interleavings.cases*)
next
  show *?expr* ⊆ *?S*
    by (*auto intro*: *interleavings.intros*)
qed

lemma *interleavings-filter*:
  assumes *X* ∩ *Y* = {} *set zs* ⊆ *X* ∪ *Y*
  shows *interleavings* [*z*←*zs* . *z* ∈ *X*] [*z*←*zs* . *z* ∈ *Y*] *zs*
using *assms* **by** (*induct zs*) (*auto intro*: *interleavings.intros*)

lemma *interleavings-filter-eq1*:
  assumes *interleavings xs ys zs*
  assumes (∀ *x*∈*set xs*. *P x*) ∧ (∀ *y*∈*set ys*. ¬ *P y*)
  shows *filter P zs* = *xs*
using *assms* **by** (*induct rule*: *interleavings.induct*) *auto*

lemma *interleavings-filter-eq2*:
  assumes *interleavings xs ys zs*
  assumes (∀ *x*∈*set xs*. ¬ *P x*) ∧ (∀ *y*∈*set ys*. *P y*)
  shows *filter P zs* = *ys*
using *assms* **by** (*induct rule*: *interleavings.induct*) *auto*

lemma *interleavings-length*:
  assumes *interleavings xs ys zs*
  shows *length xs* + *length ys* = *length zs*
using *assms* **by** (*induct xs ys zs rule*: *interleavings.induct*) *auto*

lemma *interleavings-set*:
  assumes *interleavings xs ys zs*
  shows *set xs* ∪ *set ys* = *set zs*
using *assms* **by** (*induct xs ys zs rule*: *interleavings.induct*) *auto*

lemma *interleavings-distinct*:
  assumes *interleavings xs ys zs*
  shows *distinct xs* $\wedge$ *distinct ys* $\wedge$ *set xs* $\cap$ *set ys* = {} $\longleftrightarrow$ *distinct zs*
using *assms interleavings-set* **by** (*induct xs ys zs rule*: *interleavings.induct*) *fast-force+*

lemma *two-mutual-lists-induction*:
  assumes $\bigwedge$*ys. P* [] *ys*
  assumes $\bigwedge$*xs. P xs* []
  assumes $\bigwedge$*x xs y ys. P xs* (*y*#*ys*) $\Longrightarrow$ *P* (*x*#*xs*) *ys* $\Longrightarrow$ *P* (*x*#*xs*) (*y*#*ys*)
  shows *P xs ys*
using *assms* **by** (*induction-schema*) (*pat-completeness*, *lexicographic-order*)

lemma *finite-interleavings*:
  finite {*zs. interleavings xs ys zs*}
proof (*induct xs ys rule*: *two-mutual-lists-induction*)
  case (*1 ys*)
  show *?case* **by** (*simp add*: *interleaving-Nil-iff1*)
next
  case (*2 xs*)
  then show *?case* **by** (*simp add*: *interleaving-Nil-iff2*)
next
  case (*3 x xs y ys*)
  then show *?case* **by** (*simp add*: *interleavings-Cons*)
qed

lemma *card-interleavings*:
  assumes *set xs* $\cap$ *set ys* = {}
  shows *card* {*zs. interleavings xs ys zs*} = (*length xs* + *length ys choose* (*length xs*))
using *assms*
proof (*induct xs ys rule*: *two-mutual-lists-induction*)
  case (*1 ys*)
  have *card* {*zs. interleavings* [] *ys zs*} = *card* {*ys*}
    by (*simp add*: *interleaving-Nil-iff1*)
  also have ... = (*length* [] + *length ys choose* (*length* [])) **by** *simp*
  finally show *?case* **.**
next
  case (*2 xs*)
  have *card* {*zs. interleavings xs* [] *zs*} = *card* {*xs*}
    by (*simp add*: *interleaving-Nil-iff2*)
  also have ... = (*length xs* + *length* [] *choose* (*length xs*)) **by** *simp*
  finally show *?case* **.**
next
  case (*3 x xs y ys*)
  have *card* {*zs. interleavings* (*x* # *xs*) (*y* # *ys*) *zs*} =
    card ({*x*#*zs*|*zs. interleavings xs* (*y*#*ys*) *zs*} $\cup$ {*y*#*zs*|*zs. interleavings* (*x*#*xs*) *ys zs*})

by (*simp add*: *interleavings-Cons*)
also have ... = *card* {*x#zs*|*zs*. *interleavings xs* (*y#ys*) *zs*} + *card* {*y#zs*|*zs*. *interleavings* (*x#xs*) *ys zs*}
proof −
have *finite* {*x* # *zs* |*zs*. *interleavings xs* (*y* # *ys*) *zs*}
by (*simp add*: *finite-interleavings*)
moreover have *finite* {*y* # *zs* |*zs*. *interleavings* (*x* # *xs*) *ys zs*}
by (*simp add*: *finite-interleavings*)
moreover have {*x* # *zs* |*zs*. *interleavings xs* (*y* # *ys*) *zs*} ∩ {*y* # *zs* |*zs*. *interleavings* (*x* # *xs*) *ys zs*} = {}
using ‹*set* (*x* # *xs*) ∩ *set* (*y* # *ys*) = {}› **by** *auto*
ultimately show *?thesis* **by** (*simp add*: *card-Un-disjoint*)
qed
also have ... = *card* ((λ*zs*. *x* # *zs*) ‘ {*zs*. *interleavings xs* (*y* # *ys*) *zs*}) +
card ((λ*zs*. *y* # *zs*) ‘ {*zs*. *interleavings* (*x#xs*) *ys zs*})
by (*simp add*: *setcompr-eq-image*)
also have ... = *card* {*zs*. *interleavings xs* (*y* # *ys*) *zs*} + *card* {*zs*. *interleavings* (*x#xs*) *ys zs*}
by (*simp add*: *card-image*)
also have ... = (*length xs* + *length* (*y* # *ys*) *choose length xs*) + (*length* (*x* # *xs*) + *length ys choose length* (*x* # *xs*))
using *3* **by** *simp*
also have ... = *length* (*x* # *xs*) + *length* (*y* # *ys*) *choose length* (*x* # *xs*) **by** *simp*
finally show *?case* **.**
qed

1.3 Cardinality of Distinct Fixed-Length Lists from a Union of Two Sets

lemma *lists-distinct-union-by-interleavings*:
assumes *X* ∩ *Y* = {}
shows {*zs*. *length zs* = *n* ∧ *distinct zs* ∧ *set zs* ⊆ *X* ∪ *Y*} = *do* {
k ← {*0*..*n*};
xs ← {*xs*. *length xs* = *k* ∧ *distinct xs* ∧ *set xs* ⊆ *X*};
ys ← {*ys*. *length ys* = *n* − *k* ∧ *distinct ys* ∧ *set ys* ⊆ *Y*};
{*zs*. *interleavings xs ys zs*}
} (**is** *?S* = *?expr*)
proof
show *?S* ⊆ *?expr*
proof
fix *zs*
assume *zs* ∈ *?S*
from *this* **have** *length zs* = *n* **and** *distinct zs* **and** *set zs* ⊆ *X* ∪ *Y* **by** *auto*
define *xs* **where** *xs* = *filter* (λ*z*. *z* ∈ *X*) *zs*
define *ys* **where** *ys* = *filter* (λ*z*. *z* ∈ *Y*) *zs*
have *eq*: [*z*←*zs* . *z* ∈ *Y*] = [*z*←*zs* . *z* ∉ *X*]
using ‹*set zs* ⊆ *X* ∪ *Y*› ‹*X* ∩ *Y* = {}›
by (*auto intro*: *filter-cong*)

```
    have length xs ≤ n ∧ distinct xs ∧ set xs ⊆ X
      using ‹length zs = n› ‹distinct zs› unfolding xs-def by auto
    moreover have length ys = n − length xs
      using ‹set zs ⊆ X ∪ Y› ‹length zs = n›
      unfolding xs-def ys-def eq
      by (metis diff-add-inverse sum-length-filter-compl)
    moreover have distinct ys ∧ set ys ⊆ Y
      using ‹distinct zs› unfolding ys-def by auto
    moreover have interleavings xs ys zs
      using xs-def ys-def ‹X ∩ Y = {}› ‹set zs ⊆ X ∪ Y›
      by (simp add: interleavings-filter)
    ultimately show zs ∈ ?expr by force
  qed
next
  show ?expr ⊆ ?S
  proof
    fix zs
    assume zs ∈ ?expr
    from this obtain xs ys where length xs ≤ n distinct xs set xs ⊆ X
     and length ys = n − length xs distinct ys set ys ⊆ Y interleavings xs ys zs by
auto
    have length zs = n
      using ‹length xs ≤ n› ‹length ys = n − length xs› ‹interleavings xs ys zs›
      using interleavings-length by force
    moreover have distinct zs
      using ‹distinct xs› ‹distinct ys› ‹interleavings xs ys zs› ‹set xs ⊆ X› ‹set ys
⊆ Y›
      using ‹X ∩ Y = {}› interleavings-distinct by fastforce
    moreover have set zs ⊆ X ∪ Y
      using ‹interleavings xs ys zs› ‹set xs ⊆ X› ‹set ys ⊆ Y› interleavings-set by
blast
    ultimately show zs ∈ ?S by blast
  qed
qed

lemma interleavings-inject:
  assumes (set xs ∪ set xs') ∩ (set ys ∪ set ys') = {}
  assumes interleavings xs ys zs interleavings xs' ys' zs'
  assumes zs = zs'
  shows xs = xs' and ys = ys'
proof −
  have xs = filter (λz. z ∈ set xs ∪ set xs') zs
    using ‹(set xs ∪ set xs') ∩ (set ys ∪ set ys') = {}› ‹interleavings xs ys zs›
    by (auto intro: interleavings-filter-eq1 [symmetric])
  also have ... = filter (λz. z ∈ set xs ∪ set xs') zs'
    using ‹zs = zs'› by simp
  also have ... = xs'
    using ‹(set xs ∪ set xs') ∩ (set ys ∪ set ys') = {}› ‹interleavings xs' ys' zs'›
    by (auto intro: interleavings-filter-eq1)
```

finally show *xs* = *xs′* **by** *simp*
have *ys* = *filter* (λ*z*. *z* ∈ *set ys* ∪ *set ys′*) *zs*
using ‹(*set xs* ∪ *set xs′*) ∩ (*set ys* ∪ *set ys′*) = {}› ‹*interleavings xs ys zs*›
by (*auto intro*: *interleavings-filter-eq2*[*symmetric*])
also have . . . = *filter* (λ*z*. *z* ∈ *set ys* ∪ *set ys′*) *zs′*
using ‹*zs* = *zs′*› **by** *simp*
also have . . . = *ys′*
using ‹(*set xs* ∪ *set xs′*) ∩ (*set ys* ∪ *set ys′*) = {}› ‹*interleavings xs′ ys′ zs′*›
by (*auto intro*: *interleavings-filter-eq2*)
finally show *ys* = *ys′* **.**
qed

lemma *injectivity*:
assumes *X* ∩ *Y* = {}
assumes *k* ∈ {*0*..*n*} ∧ *k′* ∈ {*0*..*n*}
assumes (*length xs* = *k* ∧ *distinct xs* ∧ *set xs* ⊆ *X*) ∧ (*length xs′* = *k′* ∧ *distinct xs′* ∧ *set xs′* ⊆ *X*)
assumes (*length ys* = *n* − *k* ∧ *distinct ys* ∧ *set ys* ⊆ *Y*) ∧ (*length ys′* = *n* − *k′* ∧ *distinct ys′* ∧ *set ys′* ⊆ *Y*)
assumes *interleavings xs ys zs* ∧ *interleavings xs′ ys′ zs′*
assumes *zs* = *zs′*
shows *k* = *k′* **and** *xs* = *xs′* **and** *ys* = *ys′*
proof −
from *assms*(*1*,*3*,*4*) **have** (*set xs* ∪ *set xs′*) ∩ (*set ys* ∪ *set ys′*) = {} **by** *blast*
from *this assms*(*5*) ‹*zs* = *zs′*› **show** *xs* = *xs′* **and** *ys* = *ys′*
using *interleavings-inject* **by** *fastforce*+
from *this assms*(*3*) **show** *k* = *k′* **by** *auto*
qed

lemma *finite-length-distinct*: *finite X* ⟹ *finite* {*xs*. *length xs* = *k* ∧ *distinct xs* ∧ *set xs* ⊆ *X*}
by(*fast elim*: *rev-finite-subset*[*OF finite-subset-distinct*])

lemma *card-lists-distinct-length-eq-union*:
assumes *finite X finite Y X* ∩ *Y* = {}
shows *card* {*zs*. *length zs* = *n* ∧ *distinct zs* ∧ *set zs* ⊆ *X* ∪ *Y*} =
(∑*k*=*0*..*n*. (*n choose k*) ∗ *ffact k* (*card X*) ∗ *ffact* (*n* − *k*) (*card Y*))
(**is** *card ?S* = -)
proof −
let *?expr* = *do* {
k ← {*0*..*n*};
xs ← {*xs*. *length xs* = *k* ∧ *distinct xs* ∧ *set xs* ⊆ *X*};
ys ← {*ys*. *length ys* = *n* − *k* ∧ *distinct ys* ∧ *set ys* ⊆ *Y*};
{*zs*. *interleavings xs ys zs*}
}
from ‹*X* ∩ *Y* = {}› **have** *card ?S* = *card ?expr*
by (*simp add*: *lists-distinct-union-by-interleavings*)
let *?S* ⤜ *?comp* = *?expr*
{

fix *k*
assume $k \in ?S$
let $?expr = ?comp\ k$
let $?S \ggg= ?comp = ?expr$
from ‹*finite X*› **have** *finite ?S* **by**(*rule finite-length-distinct*)
moreover {
fix *xs*
assume *xs*: $xs \in ?S$
let $?expr = ?comp\ xs$
let $?S \ggg= ?comp = ?expr$
from ‹*finite Y*› **have** *finite ?S* **by**(*rule finite-length-distinct*)
moreover {
fix *ys*
assume *ys*: $ys \in ?S$
let $?expr = ?comp\ ys$
have *finite ?expr*
by (*simp add*: *finite-interleavings*)
moreover have $card\ ?expr = (n\ choose\ k)$
using *xs ys* ‹$X \cap Y = \{\}$› ‹$k \in$ -›
by (*subst card-interleavings*) *auto*
ultimately have $finite\ ?expr \wedge card\ ?expr = (n\ choose\ k)$ **..**
}
moreover have *disjoint-family-on ?comp ?S*
using ‹$k \in \{0..n\}$› ‹$xs \in \{xs.\ length\ xs = k \wedge distinct\ xs \wedge set\ xs \subseteq X\}$›
by (*injectivity-solver rule*: *injectivity*(*3*)[*OF* ‹$X \cap Y = \{\}$›])
moreover have $card\ ?S = ffact\ (n - k)\ (card\ Y)$
using ‹*finite Y*› **by** (*simp add*: *card-lists-distinct-length-eq*)
ultimately have $card\ ?expr = (n\ choose\ k) * ffact\ (n - k)\ (card\ Y)$
by (*subst card-bind-constant*) *auto*
moreover have *finite ?expr*
using ‹*finite ?S*› **by** (*auto intro*!: *finite-bind finite-interleavings*)
ultimately have $finite\ ?expr \wedge card\ ?expr = (n\ choose\ k) * ffact\ (n - k)\ (card\ Y)$
by *blast*
}
moreover have *disjoint-family-on ?comp ?S*
using ‹$k \in \{0..n\}$›
by (*injectivity-solver rule*: *injectivity*(*2*)[*OF* ‹$X \cap Y = \{\}$›])
moreover have $card\ ?S = ffact\ k\ (card\ X)$
using ‹*finite X*› **by** (*simp add*: *card-lists-distinct-length-eq*)
ultimately have $card\ ?expr = (n\ choose\ k) * ffact\ k\ (card\ X) * ffact\ (n - k)\ (card\ Y)$
by (*subst card-bind-constant*) *auto*
moreover have *finite ?expr*
using ‹*finite ?S*› ‹*finite Y*› **by** (*auto intro*!: *finite-bind finite-interleavings finite-length-distinct*)
ultimately have $finite\ ?expr \wedge card\ ?expr = (n\ choose\ k) * ffact\ k\ (card\ X) * ffact\ (n - k)\ (card\ Y)$
by *blast*

```
  }
  moreover have disjoint-family-on ?comp ?S
    by (injectivity-solver rule: injectivity(1)[OF ‹X ∩ Y = {}›])
  ultimately have card ?expr = (∑ k=0..n. (n choose k) ∗ ffact k (card X) ∗
ffact (n − k) (card Y))
    by (auto simp add: card-bind)
  from ‹card - = card ?expr› this show ?thesis by simp
qed

lemma
  ffact n (x + y) = (∑ k=0..n. (n choose k) ∗ ffact k x ∗ ffact (n − k) y)
proof −
  define X where X = {..<x}
  define Y where Y = {x..<x+y}
  have finite X and card X = x unfolding X-def by auto
  have finite Y and card Y = y unfolding Y-def by auto
  have X ∩ Y = {} unfolding X-def Y-def by auto
  have ffact n (x + y) = ffact n (card X + card Y)
    using ‹card X = x› ‹card Y = y› by simp
  also have . . . = ffact n (card (X ∪ Y))
    using ‹X ∩ Y = {}› ‹finite X› ‹finite Y› by (simp add: card-Un-disjoint)
  also have . . . = card {xs. length xs = n ∧ distinct xs ∧ set xs ⊆ X ∪ Y}
   using ‹finite X› ‹finite Y› by (simp add: card-lists-distinct-length-eq)
  also have . . . = (∑ k=0..n. (n choose k) ∗ ffact k (card X) ∗ ffact (n − k) (card
Y))
   using ‹X ∩ Y = {}› ‹finite X› ‹finite Y› by (simp add: card-lists-distinct-length-eq-union)
  also have . . . = (∑ k=0..n. (n choose k) ∗ ffact k x ∗ ffact (n − k) y)
    using ‹card X = x› ‹card Y = y› by simp
  finally show ?thesis .
qed

end
```

2 Proving Falling Factorial of a Sum with Induction

```
theory Falling-Factorial-Sum-Induction
imports
  Discrete-Summation.Factorials
begin
```

Note the potentially special copyright license condition of the following proof.

```
lemma ffact-add-nat:
  ffact n (x + y) = (∑ k=0..n. (n choose k) ∗ ffact k x ∗ ffact (n − k) y)
proof (induct n)
  case 0
  show ?case by simp
```

next
  case (*Suc n*)
  let *?s* = λk. (*n choose k*) ∗ *ffact k x* ∗ *ffact* (*n* − *k*) *y*
  let *?t* = λk. *ffact k x* ∗ *ffact* (*Suc n* − *k*) *y*
  let *?u* = λk. *ffact* (*Suc k*) *x* ∗ *ffact* (*n* − *k*) *y*
  have *ffact* (*Suc n*) (*x* + *y*) = (*x* + *y* − *n*) ∗ *ffact n* (*x* + *y*)
    by (*simp add*: *ffact-Suc-rev-nat*)
  also have ... = (*x* + *y* − *n*) ∗ ($\sum$ *k* = *0*..*n*. (*n choose k*) ∗ *ffact k x* ∗ *ffact* (*n* − *k*) *y*)
    using *Suc.hyps* **by** *simp*
  also have ... = ($\sum$ *k* = *0*..*n*. *?s k* ∗ (*x* + *y* − *n*))
    by (*simp add*: *mult.commute sum-distrib-left*)
  also have ... = ($\sum$ *k* = *0*..*n*. *?s k* ∗ ((*y* + *k* − *n*) + (*x* − *k*)))
  proof −
    have *?s k* ∗ (*x* + *y* − *n*) = *?s k* ∗ ((*y* + *k* − *n*) + (*x* − *k*)) **for** *k*
      by (*cases k* ≤ *x* ∨ *n* − *k* ≤ *y*) (*auto simp add*: *ffact-nat-triv*)
    from *this* **show** *?thesis*
      by (*auto intro*: *sum.cong simp only*: *refl*)
  qed
  also have ... = ($\sum$ *k* = *0*..*n*. (*n choose k*) ∗ (*?t k* + *?u k*))
    by (*auto intro*!: *sum.cong simp add*: *Suc-diff-le ffact-Suc-rev-nat*) *algebra*
  also have ... = ($\sum$ *k* = *0*..*n*. (*n choose k*) ∗ *?t k*) + ($\sum$ *k* = *0*..*n*. (*n choose k*) ∗ *?u k*)
    by (*simp add*: *sum.distrib add-mult-distrib2 mult.commute mult.left-commute*)
  also have ... = *?t 0* + ($\sum$ *k* = *0*..*n*. ((*n choose k*) + (*n choose Suc k*)) ∗ *?u k*)
  proof −
    have ... = (*?t 0* + ($\sum$ *k* = *0*..*n*. (*n choose Suc k*) ∗ *?u k*)) + ($\sum$ *k* = *0*..*n*. (*n choose k*) ∗ *?u k*)
    proof −
      have ($\sum$ *k* = *Suc 0*..*n*. (*n choose k*) ∗ *?t k*) = ($\sum$ *k* = *0*..*n*. (*n choose Suc k*) ∗ *?u k*)
      proof −
        have ($\sum$ *k* = *Suc 0*..*n*. (*n choose k*) ∗ *?t k*) = ($\sum$ *k* = *Suc 0*..*Suc n*. (*n choose k*) ∗ *?t k*)
          by *simp*
        also have ... = (*sum* ((λk. (*n choose k*) ∗ *?t k*) *o Suc*) {*0*..*n*})
        by (*simp only*: *sum.reindex*[*symmetric*, *of Suc*] *inj-Suc image-Suc-atLeastAtMost*)
        also have ... = ($\sum$ *k* = *0*..*n*. (*n choose Suc k*) ∗ *?u k*)
          by *simp*
        finally show *?thesis* **.**
      qed
      from *this* **show** *?thesis*
        by (*simp add*: *sum.atLeast-Suc-atMost*[*of* - - λk. (*n choose k*) ∗ *?t k*])
    qed
    also have ... = *?t 0* + ($\sum$ *k* = *0*..*n*. ((*n choose k*) + (*n choose Suc k*)) ∗ *?u k*)
      by (*simp add*: *distrib-right sum.distrib*)
    finally show *?thesis* **.**
  qed

also have ... = ($\sum k = 0..Suc\ n.$ (*Suc n choose k*) ∗ *ffact k x* ∗ *ffact* (*Suc n* − *k*) *y*)
proof −
let *?v* = λ*k*. (*Suc n choose k*) ∗ *ffact k x* ∗ *ffact* (*Suc n* − *k*) *y*
have ... = *?v 0* + ($\sum k = 0..n.$ (*Suc n choose* (*Suc k*)) ∗ *?u k*)
by *simp*
also have ... = *?v 0* + ($\sum k = Suc\ 0..Suc\ n.$ *?v k*)
by (*simp only*: *sum.shift-bounds-cl-Suc-ivl diff-Suc-Suc mult.assoc*)
also have ... = ($\sum k = 0..Suc\ n.$ (*Suc n choose k*) ∗ *ffact k x* ∗ *ffact* (*Suc n* − *k*) *y*)
by (*simp add*: *sum.atLeast-Suc-atMost*)
finally show *?thesis* **.**
qed
finally show *?case* **.**
qed

lemma *ffact-add*:
fixes *x y* :: ′*a*::{*ab-group-add*, *comm-semiring-1-cancel*, *ring-1*}
shows *ffact n* (*x* + *y*) = ($\sum k{=}0..n.$ *of-nat* (*n choose k*) ∗ *ffact k x* ∗ *ffact* (*n* − *k*) *y*)
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
let *?s* = λ*k*. *of-nat* (*n choose k*) ∗ *ffact k x* ∗ *ffact* (*n* − *k*) *y*
let *?t* = λ*k*. *ffact k x* ∗ *ffact* (*Suc n* − *k*) *y*
let *?u* = λ*k*. *ffact* (*Suc k*) *x* ∗ *ffact* (*n* − *k*) *y*
have *ffact* (*Suc n*) (*x* + *y*) = (*x* + *y* − *of-nat n*) ∗ *ffact n* (*x* + *y*)
by (*simp add*: *ffact-Suc-rev*)
also have ... = (*x* + *y* − *of-nat n*) ∗ ($\sum k = 0..n.$ *of-nat* (*n choose k*) ∗ *ffact k x* ∗ *ffact* (*n* − *k*) *y*)
using *Suc.hyps* **by** *simp*
also have ... = ($\sum k = 0..n.$ *?s k* ∗ (*x* + *y* − *of-nat n*))
by (*simp add*: *mult.commute sum-distrib-left*)
also have ... = ($\sum k = 0..n.$ *?s k* ∗ ((*y* + *of-nat k* − *of-nat n*) + (*x* − *of-nat k*)))
by (*auto intro*: *sum.cong simp add*: *diff-add-eq add-diff-eq add.commute*)
also have ... = ($\sum k = 0..n.$ *of-nat* (*n choose k*) ∗ (*?t k* + *?u k*))
proof −
{
fix *k*
assume *k* ≤ *n*
have *?u k* = *ffact k x* ∗ *ffact* (*n* − *k*) *y* ∗ (*x* − *of-nat k*)
by (*simp add*: *ffact-Suc-rev Suc-diff-le of-nat-diff mult.commute mult.left-commute*)
moreover from ‹*k* ≤ *n*› **have** *?t k* = *ffact k x* ∗ *ffact* (*n* − *k*) *y* ∗ (*y* + *of-nat k* − *of-nat n*)
by (*simp add*: *ffact-Suc-rev Suc-diff-le of-nat-diff diff-diff-eq2 mult.commute*

mult.left-commute)
ultimately have
?s k ∗ ((*y* + *of-nat k* − *of-nat n*) + (*x* − *of-nat k*)) = *of-nat* (*n choose k*) ∗ (*?t k* + *?u k*)
by (*metis* (*no-types*, *lifting*) *distrib-left mult.assoc*)
}
from *this* **show** *?thesis* **by** (*auto intro*: *sum.cong*)
qed
also have ... = (∑ *k* = *0*..*n*. *of-nat* (*n choose k*) ∗ *?t k*) + (∑ *k* = *0*..*n*. *of-nat* (*n choose k*) ∗ *?u k*)
by (*simp add*: *sum.distrib distrib-left mult.commute mult.left-commute*)
also have ... = *?t 0* + (∑ *k* = *0*..*n*. *of-nat* ((*n choose k*) + (*n choose Suc k*)) ∗ *?u k*)
proof −
have ... = (*?t 0* + (∑ *k* = *0*..*n*. *of-nat* (*n choose Suc k*) ∗ *?u k*)) + (∑ *k* = *0*..*n*. *of-nat* (*n choose k*) ∗ *?u k*)
proof −
have (∑ *k* = *Suc 0*..*n*. *of-nat* (*n choose k*) ∗ *?t k*) = (∑ *k* = *0*..*n*. *of-nat* (*n choose Suc k*) ∗ *?u k*)
proof −
have (∑ *k* = *Suc 0*..*n*. *of-nat* (*n choose k*) ∗ *?t k*) = (∑ *k* = *Suc 0*..*Suc n*. *of-nat* (*n choose k*) ∗ *?t k*)
by (*simp add*: *binomial-eq-0*)
also have ... = (*sum* ((λ*k*. *of-nat* (*n choose k*) ∗ *?t k*) *o Suc*) {*0*..*n*})
by (*simp only*: *sum.reindex*[*symmetric*, *of Suc*] *inj-Suc image-Suc-atLeastAtMost*)
also have ... = (∑ *k* = *0*..*n*. *of-nat* (*n choose Suc k*) ∗ *?u k*)
by *simp*
finally show *?thesis* **.**
qed
from *this* **show** *?thesis*
by (*simp add*: *sum.atLeast-Suc-atMost*[*of* - - λ*k*. *of-nat* (*n choose k*) ∗ *?t k*])
qed
also have ... = *?t 0* + (∑ *k* = *0*..*n*. *of-nat* ((*n choose k*) + (*n choose Suc k*)) ∗ *?u k*)
by (*simp add*: *distrib-right sum.distrib*)
finally show *?thesis* **.**
qed
also have ... = (∑ *k* = *0*..*Suc n*. *of-nat* (*Suc n choose k*) ∗ *ffact k x* ∗ *ffact* (*Suc n* − *k*) *y*)
proof −
let *?v* = λ*k*. *of-nat* (*Suc n choose k*) ∗ *ffact k x* ∗ *ffact* (*Suc n* − *k*) *y*
have ... = *?v 0* + (∑ *k* = *0*..*n*. *of-nat* (*Suc n choose* (*Suc k*)) ∗ *?u k*)
by *simp*
also have ... = *?v 0* + (∑ *k* = *Suc 0*..*Suc n*. *?v k*)
by (*simp only*: *sum.shift-bounds-cl-Suc-ivl diff-Suc-Suc mult.assoc*)
also have ... = (∑ *k* = *0*..*Suc n*. *of-nat* (*Suc n choose k*) ∗ *ffact k x* ∗ *ffact* (*Suc n* − *k*) *y*)
by (*simp add*: *sum.atLeast-Suc-atMost*)
finally show *?thesis* **.**

  qed
  finally show *?case* **.**
qed

end

3 Proving Falling Factorial of a Sum with Vandermonde Identity

theory *Falling-Factorial-Sum-Vandermonde*
imports
  Discrete-Summation.Factorials
begin

Note the potentially special copyright license condition of the following proof.

lemma *ffact-add-nat*:
  shows $ffact\ k\ (n + m) = (\sum i{\le}k.\ (k\ choose\ i) * ffact\ i\ n * ffact\ (k - i)\ m)$
proof −
  have $ffact\ k\ (n + m) = fact\ k * ((n + m)\ choose\ k)$
    by (*simp only*: *ffact-eq-fact-mult-binomial*)
  also have $\ldots = fact\ k * (\sum i{\le}k.\ (n\ choose\ i) * (m\ choose\ (k - i)))$
    by (*simp only*: *vandermonde*)
  also have $\ldots = (\sum i{\le}k.\ fact\ k * (n\ choose\ i) * (m\ choose\ (k - i)))$
    by (*simp add*: *sum-distrib-left field-simps*)
  also have $\ldots = (\sum i{\le}k.\ (fact\ i * fact\ (k - i) * (k\ choose\ i)) * (n\ choose\ i) * (m\ choose\ (k - i)))$
    by (*simp add*: *binomial-fact-lemma*)
  also have $\ldots = (\sum i{\le}k.\ (k\ choose\ i) * (fact\ i * (n\ choose\ i)) * (fact\ (k - i) * (m\ choose\ (k - i))))$
    by (*auto intro*: *sum.cong*)
  also have $\ldots = (\sum i{\le}k.\ (k\ choose\ i) * ffact\ i\ n * ffact\ (k - i)\ m)$
    by (*simp only*: *ffact-eq-fact-mult-binomial*)
  finally show *?thesis* **.**
qed

end

4 Note on Copyright Licensing

The two main proofs, the induction and the algebraic proof in this AFP entry are (even textually) very close to the initial material from Mathematics Stack Exchange.

In case the two Isabelle proofs are judged to build upon the main proofs from Mathematics Stack Exchange, the CC BY-SA 3.0 license requires that these proofs must be available under the same license, and hence, these proofs are consequently licensed under CC BY-SA 3.0. In case the two Isabelle proofs are not judged to build upon the material from Mathematics Stack Exchange, I as an author provide them under the 3-Clause BSD License (https://opensource.org/licenses/BSD-3-Clause) to allow their seemless integration into the Isabelle repository at any point in time.

All other content that does not build upon the material from Mathematics Stack Exchange is licensed under the 3-clause BSD License, and can be copied, moved or integrated in other work licensed under the 3-clause BSD License without further consideration of the different obligations of the existing copyright licensing.

References

[1] ajotatxe. Combinatorial proof of falling factorial and binomial theorem. Mathematics Stack Exchange. https://math.stackexchange.com/q/1271700 (version: 2015-05-07), author profile: https://math.stackexchange.com/users/132456/ajotatxe.

[2] grapher. Combinatorial proof of falling factorial and binomial theorem. Mathematics Stack Exchange. https://math.stackexchange.com/q/1271688 (version: 2016-08-27), author profile: https://math.stackexchange.com/users/199155/grapher.

[3] F. John. Combinatorial proof of falling factorial and binomial theorem. Mathematics Stack Exchange. https://math.stackexchange.com/q/2161558 (version: 2017-11-14), author profile: https://math.stackexchange.com/users/302692/foobaz-john.

[4] B. M. Scott. Combinatorial proof of falling factorial and binomial theorem. Mathematics Stack Exchange. https://math.stackexchange.com/q/1271983 (version: 2015-05-07), author profile: https://math.stackexchange.com/users/12042/brian-m-scott.