Discrete Summation

Florian Haftmann
with contributions by Amine Chaieb

May 26, 2024

Abstract

These theories introduce basic concepts and proofs about discrete summation: shifts, formal summation, falling factorials and stirling numbers. As proof of concept, a simple summation conversion is provided.

1 Falling factorials

theory *Factorials*
imports *Complex-Main HOL−Combinatorics.Stirling*
begin

lemma *pochhammer-0* [*simp*]: — TODO move
pochhammer 0 n = (*0*::*nat*) **if** *n* > *0*
using *that* **by** (*simp add*: *pochhammer-prod*)

definition *ffact* :: *nat* ⇒ *'a*::*comm-semiring-1-cancel* ⇒ *'a*
where *ffact n a* = *pochhammer* (*a* + *1* − *of-nat n*) *n*

lemma *ffact-0* [*simp*]:
ffact 0 = (λ*x*. *1*)
by (*simp add*: *fun-eq-iff ffact-def*)

lemma *ffact-Suc*:
ffact (*Suc n*) *a* = *a* ∗ *ffact n* (*a* − *1*)
for *a* :: *'a* :: *comm-ring-1*
by (*simp add*: *ffact-def pochhammer-prod prod.atLeast0-lessThan-Suc algebra-simps*)

lemma *ffact-Suc-rev*:
ffact (*Suc n*) *m* = (*m* − *of-nat n*) ∗ *ffact n m*
for *m* :: *'a* :: {*comm-semiring-1-cancel*, *ab-group-add*}
unfolding *ffact-def pochhammer-rec* **by** (*simp add*: *diff-add-eq*)

lemma *ffact-nat-triv*:
ffact n m = *0* **if** *m* < *n*

using *that* **by** (*simp add*: *ffact-def*)

lemma *ffact-Suc-nat*:
ffact (*Suc n*) *m* = *m* ∗ *ffact n* (*m* − *1*)
for *m* :: *nat*
proof (*cases n* ≤ *m*)
case *True*
then show *?thesis*
by (*simp add*: *ffact-def pochhammer-prod algebra-simps prod.atLeast0-lessThan-Suc*)
next
case *False*
then have *m* < *n*
by *simp*
then show *?thesis*
by (*simp add*: *ffact-nat-triv*)
qed

lemma *ffact-Suc-rev-nat*:
ffact (*Suc n*) *m* = (*m* − *n*) ∗ *ffact n m*
proof (*cases n* ≤ *m*)
case *True*
then show *?thesis*
by (*simp add*: *ffact-def pochhammer-rec Suc-diff-le*)
next
case *False*
then have *m* < *n* **by** *simp*
then show *?thesis* **by** (*simp add*: *ffact-nat-triv*)
qed

lemma *fact-div-fact-ffact*:
fact n div fact m = *ffact* (*n* − *m*) *n* **if** *m* ≤ *n*
proof −
from *that* **have** *fact n* = *ffact* (*n* − *m*) *n* ∗ *fact m*
by (*simp add*: *ffact-def pochhammer-product pochhammer-fact*)
moreover have *fact m dvd* (*fact n* :: *nat*)
using *that* **by** (*rule fact-dvd*)
ultimately show *?thesis*
by *simp*
qed

lemma *fact-div-fact-ffact-nat*:
fact n div fact (*n* − *k*) = *ffact k n* **if** *k* ≤ *n*
using *that* **by** (*simp add*: *fact-div-fact-ffact*)

lemma *ffact-fact*:
ffact n (*of-nat n*) = (*of-nat* (*fact n*) :: *'a* :: *comm-ring-1*)
by (*induct n*) (*simp-all add*: *algebra-simps ffact-Suc*)

lemma *ffact-add-diff-assoc*:

```
  (a − of-nat n) * ffact n a + of-nat n * ffact n a = a * ffact n a
    for a :: 'a :: comm-ring-1
  by (simp add: algebra-simps)

lemma mult-ffact:
  a * ffact n a = ffact (Suc n) a + of-nat n * ffact n a
    for a :: 'a :: comm-ring-1
proof −
  have ffact (Suc n) a + of-nat n * (ffact n a) = (a − of-nat n) * (ffact n a) +
of-nat n * (ffact n a)
    using ffact-Suc-rev [of n] by auto
  also have ... = a * ffact n a using ffact-add-diff-assoc by (simp add: alge-
bra-simps)
  finally show ?thesis by simp
qed


lemma prod-ffact:
  fixes m :: 'a :: {ord, ring-1, comm-monoid-mult, comm-semiring-1-cancel}
  shows (∏ i = 0..<n. m − of-nat i) = ffact n m
proof −
  have inj-on (λj. j − 1) {1..n}
    by (force intro: inj-on-diff-nat)
  moreover have {0..<n} = (λj. j − 1) ' {1..n}
  proof −
    have i ∈ (λj. j − 1) ' {1..n} if i ∈ {0..<n} for i
      using that by (auto intro: image-eqI[where x=i + 1])
    from this show ?thesis by auto
  qed
  moreover have m − of-nat (i − 1) = m + 1 − of-nat n + of-nat (n − i) if i
∈ {1..n} for i
    using that by (simp add: of-nat-diff)
  ultimately have (∏ i = 0..<n. m − of-nat i) = (∏ i = 1..n. m + 1 − of-nat
n + of-nat (n − i))
    by (rule prod.reindex-cong)
  from this show ?thesis
    unfolding ffact-def by (simp only: pochhammer-prod-rev)
qed

lemma prod-ffact-nat:
  fixes m :: nat
  shows (∏ i = 0..<n. m − i) = ffact n m
proof cases
  assume n ≤ m
  have inj-on (λj. j − 1) {1..n} by (force intro: inj-on-diff-nat)
  moreover have {0..<n} = (λj. j − 1) ' {1..n}
  proof −
    have i ∈ (λj. j − 1) ' {1..n} if i ∈ {0..<n} for i
      using that by (auto intro: image-eqI[where x=i + 1])
```

```
    from this show ?thesis by auto
  qed
  ultimately have (∏ i = 0..<n. m − i) = (∏ i = 1..n. (m + 1) − i)
    by (auto intro: prod.reindex-cong[where l=λi. i − 1])
  from this ‹n ≤ m› show ?thesis
    unfolding ffact-def by (simp add: pochhammer-prod-rev)
next
  assume ¬ n ≤ m
  from this show ?thesis by (auto simp add: ffact-nat-triv)
qed


lemma prod-rev-ffact:
  fixes m :: 'a :: {ord, ring-1, comm-monoid-mult, comm-semiring-1-cancel}
  shows (∏ i = 1..n. m − of-nat n + of-nat i) = ffact n m
proof −
  have inj-on (λi. i + 1) {0..<n} by simp
  moreover have {1..n} = (λi. i + 1) ‘ {0..<n} by auto
  moreover have m − of-nat n + of-nat (i + 1) = m + 1 − of-nat n + of-nat i
for i by simp
  ultimately have (∏ i = 1..n. m − of-nat n + of-nat i) = (∏ i = 0..<n. m +
1 − of-nat n + of-nat i)
    by (rule prod.reindex-cong[where l=λi. i + 1])
  from this show ?thesis
    unfolding ffact-def by (simp only: pochhammer-prod)
qed

lemma prod-rev-ffact-nat:
  fixes m :: nat
  assumes n ≤ m
  shows (∏ i = 1..n. m − n + i) = ffact n m
proof −
  have inj-on (λi. i + 1) {0..<n} by simp
  moreover have {1..n} = (λi. i + 1) ‘ {0..<n} by auto
  moreover have m − n + (i + 1) = m + 1 − n + i for i
    using ‹n ≤ m› by auto
  ultimately have (∏ i = 1..n. m − n + i) = (∏ i = 0..<n. m + 1 − n + i)
    by (rule prod.reindex-cong)
 from this show ?thesis
   unfolding ffact-def by (simp only: pochhammer-prod of-nat-id)
qed

lemma prod-rev-ffact-nat':
  fixes m :: nat
  assumes n ≤ m
  shows ∏ {m − n + 1..m} = ffact n m
proof −
  have inj-on (λi. m − n + i) {1::nat..n} by (auto intro: inj-onI)
  moreover have {m − n + 1..m} = (λi. m − n + i) ‘ {1::nat..n}
```

```
  proof −
    have i ∈ (λi. m + i − n) ‘ {Suc 0..n} if i ∈ {m − n + 1..m} for i
      using that ‹n ≤ m› by (auto intro!: image-eqI[where x=i − (m − n)])
    with ‹n ≤ m› show ?thesis by auto
  qed
  moreover have m − n + i = m − n + i for i ..
  ultimately have ∏{m − n + (1::nat)..m} = (∏ i = 1..n. m − n + i)
    by (rule prod.reindex-cong)
  from this show ?thesis
    using ‹n ≤ m› by (simp only: prod-rev-ffact-nat)
qed

lemma ffact-eq-fact-mult-binomial:
  ffact k n = fact k * (n choose k)
proof cases
  assume k ≤ n
  have ffact k n = fact n div fact (n − k)
    using ‹k ≤ n› by (simp add: fact-div-fact-ffact-nat)
  also have ... = fact k * (n choose k)
    using ‹k ≤ n› by (simp add: binomial-fact-lemma[symmetric])
  finally show ?thesis .
next
  assume ¬ k ≤ n
  from this ffact-nat-triv show ?thesis by force
qed

lemma of-nat-ffact:
  of-nat (ffact n m) = ffact n (of-nat m :: 'a :: comm-ring-1)
proof (induct n arbitrary: m)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  show ?case
  proof (cases m)
    case 0
    then show ?thesis
      by (simp add: ffact-Suc-nat ffact-Suc)
  next
    case (Suc m)
    with Suc.hyps show ?thesis
      by (simp add: algebra-simps ffact-Suc-nat ffact-Suc)
  qed
qed

lemma of-int-ffact:
  of-int (ffact n k) = ffact n (of-int k :: 'a :: comm-ring-1)
proof (induct n arbitrary: k)
```

case *0* **then show** *?case* **by** *simp*
next
case (*Suc n k*)
then have *of-int* (*ffact n* (*k* − *1*)) = *ffact n* (*of-int* (*k* − *1*) :: *'a*) **.**
then show *?case*
by (*simp add*: *ffact-Suc-nat ffact-Suc*)
qed

lemma *ffact-minus*:
fixes *x* :: *'a* :: *comm-ring-1*
shows *ffact n* (− *x*) = (− *1*) ^ *n* ∗ *pochhammer x n*
proof −
have *ffact n* (− *x*) = *pochhammer* (− *x* + *1* − *of-nat n*) *n*
unfolding *ffact-def* **..**
also have ... = *pochhammer* (− *x* − *of-nat n* + *1*) *n*
by (*simp add*: *diff-add-eq*)
also have ... = (− *1*) ^ *n* ∗ *pochhammer* (− (− *x*)) *n*
by (*rule pochhammer-minus'*)
also have ... = (− *1*) ^ *n* ∗ *pochhammer x n* **by** *simp*
finally show *?thesis* **.**
qed

Conversion of natural potences into falling factorials and back

lemma *monomial-ffact*:
a ^ *n* = (∑ *k* = *0*..*n*. *of-nat* (*Stirling n k*) ∗ *ffact k a*)
for *a* :: *'a* :: *comm-ring-1*
proof (*induct n*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc n*)
then have *a* ^ *Suc n* = *a* ∗ (∑ *k* = *0*..*n*. *of-nat* (*Stirling n k*) ∗ *ffact k a*)
by *simp*
also have ... = (∑ *k* = *0*..*n*. *of-nat* (*Stirling n k*) ∗ (*a* ∗ *ffact k a*))
by (*simp add*: *sum-distrib-left algebra-simps*)
also have ... = (∑ *k* = *0*..*n*. *of-nat* (*Stirling n k*) ∗ *ffact* (*Suc k*) *a*) +
(∑ *k* = *0*..*n*. *of-nat* (*Stirling n k*) ∗ (*of-nat k* ∗ *ffact k a*))
by (*simp add*: *sum.distrib algebra-simps mult-ffact*)
also have ... = (∑ *k* = *0*.. *Suc n*. *of-nat* (*Stirling n k*) ∗ *ffact* (*Suc k*) *a*) +
(∑ *k* = *0*..*Suc n*. *of-nat* ((*Suc k*) ∗ (*Stirling n* (*Suc k*))) ∗ (*ffact* (*Suc k*) *a*))
proof −
have (∑ *k* = *0*..*n*. *of-nat* (*Stirling n k*) ∗ (*of-nat k* ∗ *ffact k a*)) =
(∑ *k* = *0*..*n*+*2*. *of-nat* (*Stirling n k*) ∗ (*of-nat k* ∗ *ffact k a*)) **by** *simp*
also have ... = (∑ *k* = *Suc 0* .. *Suc* (*Suc n*). *of-nat* (*Stirling n k*) ∗ (*of-nat k* ∗ *ffact k a*))
by (*simp only*: *sum.atLeast-Suc-atMost* [*of 0 n* + *2*]) *simp*
also have ... = (∑ *k* = *0* .. *Suc n*. *of-nat* (*Stirling n* (*Suc k*)) ∗ (*of-nat* (*Suc k*) ∗ *ffact* (*Suc k*) *a*))
by (*simp only*: *image-Suc-atLeastAtMost sum.shift-bounds-cl-Suc-ivl*)
also have ... = (∑ *k* = *0* .. *Suc n*. *of-nat* ((*Suc k*) ∗ *Stirling n* (*Suc k*)) ∗ *ffact*

```
(Suc k) a)
      by (simp only: of-nat-mult algebra-simps)
    finally have (∑ k = 0..n. of-nat (Stirling n k) * (of-nat k * ffact k a)) =
      (∑ k = 0..Suc n. of-nat (Suc k * Stirling n (Suc k)) * ffact (Suc k) a)
      by simp
    then show ?thesis by simp
  qed
  also have ... = (∑ k = 0..n. of-nat (Stirling (Suc n) (Suc k)) * ffact (Suc k)
a)
    by (simp add: algebra-simps sum.distrib)
  also have ... = (∑ k = Suc 0..Suc n. of-nat (Stirling (Suc n) k) * ffact k a)
    by (simp only: image-Suc-atLeastAtMost sum.shift-bounds-cl-Suc-ivl)
  also have ... = (∑ k = 0..Suc n. of-nat (Stirling (Suc n) k) * ffact k a)
    by (simp only: sum.atLeast-Suc-atMost [of 0 Suc n]) simp
  finally show ?case by simp
qed

lemma ffact-monomial:
  ffact n a = (∑ k = 0..n. (- 1) ^ (n - k) * of-nat (stirling n k) * a ^ k)
    for a :: 'a :: comm-ring-1
proof (induct n)
  case 0 show ?case by simp
next
  case (Suc n)
  then have ffact (Suc n) a = (a - of-nat n) * (∑ k = 0..n. (- 1) ^ (n - k) *
of-nat (stirling n k) * a ^ k)
    by (simp add: ffact-Suc-rev)
  also have ... = (∑ k = 0..n. (- 1) ^ (n - k) * of-nat (stirling n k) * a ^ (Suc
k)) +
    (∑ k = 0..n. (- 1) * (- 1) ^ (n - k) * of-nat (n * (stirling n k)) * a ^ k)
     by (simp only: diff-conv-add-uminus distrib-right) (simp add: sum-distrib-left
field-simps)
  also have ... = (∑ k = 0..n. (- 1) ^ (Suc n - Suc k) * of-nat (stirling n k) *
a ^ Suc k) +
  (∑ k = 0..n. (- 1) ^ (Suc n - Suc k) * of-nat (n * stirling n (Suc k)) * a ^
Suc k)
  proof -
    have (∑ k = 0..n. (- 1) * (- 1) ^ (n - k) * of-nat (n * stirling n k) * a ^
k) =
      (∑ k = 0..n. (- 1) ^ (Suc n - k) * of-nat (n * stirling n k) * a ^ k)
      by (simp add: Suc-diff-le)
     also have ... = (∑ k = Suc 0..Suc n. (- 1) ^ (Suc n - k) * of-nat (n *
stirling n k) * a ^ k)
      by (simp add: sum.atLeast-Suc-atMost) (cases n; simp)
    also have ... = (∑ k = 0..n. (- 1) ^ (Suc n - Suc k) * of-nat (n * stirling
n (Suc k)) * a ^ Suc k)
      by (simp only: sum.shift-bounds-cl-Suc-ivl)
    finally show ?thesis by simp
  qed
```

also have ... = $(\sum k = 0..n.\ (-1)\ \hat{}\ (Suc\ n - Suc\ k) * of\text{-}nat\ (n * stirling\ n\ (Suc\ k) + stirling\ n\ k) * a\ \hat{}\ Suc\ k)$
by (*simp add*: *sum.distrib algebra-simps*)
also have ... = $(\sum k = 0..n.\ (-1)\ \hat{}\ (Suc\ n - Suc\ k) * of\text{-}nat\ (stirling\ (Suc\ n)\ (Suc\ k)) * a\ \hat{}\ Suc\ k)$
by (*simp only*: *stirling.simps*)
also have ... = $(\sum k = Suc\ 0..Suc\ n.\ (-1)\ \hat{}\ (Suc\ n - k) * of\text{-}nat\ (stirling\ (Suc\ n)\ k) * a\ \hat{}\ k)$
by (*simp only*: *sum.shift-bounds-cl-Suc-ivl*)
also have ... = $(\sum k = 0..Suc\ n.\ (-1)\ \hat{}\ (Suc\ n - k) * of\text{-}nat\ (stirling\ (Suc\ n)\ k) * a\ \hat{}\ k)$
by (*simp add*: *sum.atLeast-Suc-atMost*)
finally show *?case* **.**
qed

end

2 Some basic facts about discrete summation

theory *Discrete-Summation*
imports *Main*
begin

2.1 Auxiliary

lemma *add-sum-orient*:
$sum\ f\ \{k..<j\} + sum\ f\ \{l..<k\} = sum\ f\ \{l..<k\} + sum\ f\ \{k..<j\}$
by (*fact add.commute*)

lemma *add-sum-int*:
fixes $j\ k\ l :: int$
shows $j < k \Longrightarrow k < l \Longrightarrow$
$sum\ f\ \{j..<k\} + sum\ f\ \{k..<l\} = sum\ f\ \{j..<l\}$
by (*simp-all add*: *sum.union-inter* [*symmetric*] *ivl-disj-un*)

2.2 The shift operator

definition $\Delta :: ('b::ring\text{-}1 \Rightarrow 'a::ab\text{-}group\text{-}add) \Rightarrow int \Rightarrow 'a$
where
$\Delta\ f\ k = f\ (of\text{-}int\ (k + 1)) - f\ (of\text{-}int\ k)$

lemma Δ-*shift*:
$\Delta\ (\lambda k.\ l + f\ k) = \Delta\ f$
by (*simp add*: Δ-*def fun-eq-iff*)

lemma Δ-*same-shift*:
assumes $\Delta\ f = \Delta\ g$
shows $\exists l.\ plus\ l \circ f \circ of\text{-}int = g \circ of\text{-}int$
proof −

let *?F* = $\lambda k.$ *f* (*of-int k*)
let *?G* = $\lambda k.$ *g* (*of-int k*)
from *assms* **have** $\bigwedge k.$ Δ *f* (*of-int k*) = Δ *g* (*of-int k*) **by** *simp*
then have *k-incr*: $\bigwedge k.$ *?F* (*k* + *1*) − *?G* (*k* + *1*) = *?F k* − *?G k*
by (*simp add*: Δ-*def algebra-simps*)
then have $\bigwedge k.$ *?F* ((*k* − *1*) + *1*) − *?G* ((*k* − *1*) + *1*) =
?F (*k* − *1*) − *?G* (*k* − *1*)
by *blast*
then have *k-decr*: $\bigwedge k.$ *?F* (*k* − *1*) − *?G* (*k* − *1*) = *?F k* − *?G k*
by *simp*
have $\bigwedge k.$ *?F k* − *?G k* = *?F 0* − *?G 0*
proof −
fix *k*
show *?F k* − *?G k* = *?F 0* − *?G 0*
by (*induct k rule*: *int-induct*)
(*simp-all add*: *k-incr k-decr del*: *of-int-add of-int-diff of-int-0*)
qed
then have $\bigwedge k.$ (*plus* (*?G 0* − *?F 0*) ∘ *?F*) *k* = *?G k*
by (*simp add*: *algebra-simps*)
then have *plus* (*?G 0* − *?F 0*) ∘ *?F* = *?G* **..**
then have *plus* (*?G 0* − *?F 0*) ∘ *f* ∘ *of-int* = *g* ∘ *of-int*
by (*simp only*: *comp-def*)
then show *?thesis* **..**
qed

lemma Δ-*add*:
Δ ($\lambda k.$ *f k* + *g k*) *k* = Δ *f k* + Δ *g k*
by (*simp add*: Δ-*def*)

lemma Δ-*factor*:
Δ ($\lambda k.$ *c* ∗ *k*) *k* = *c*
by (*simp add*: Δ-*def algebra-simps*)

2.3 The formal sum operator

definition Σ :: (*'b*::*ring-1* ⇒ *'a*::*ab-group-add*) ⇒ *int* ⇒ *int* ⇒ *'a*
where
Σ *f j l* = (*if j* < *l then sum* (*f* ∘ *of-int*) {*j*..<*l*}
else if j > *l then* − *sum* (*f* ∘ *of-int*) {*l*..<*j*}
else 0)

lemma Σ-*same* [*simp*]:
Σ *f j j* = *0*
by (*simp add*: Σ-*def*)

lemma Σ-*positive*:
j < *l* ⟹ Σ *f j l* = *sum* (*f* ∘ *of-int*) {*j*..<*l*}
by (*simp add*: Σ-*def*)

lemma Σ-*negative*:
$j > l \Longrightarrow \Sigma\ f\ j\ l = -\ \Sigma\ f\ l\ j$
by (*simp add*: Σ-*def*)

lemma Σ-*comp-of-int*:
$\Sigma\ (f \circ \textit{of-int}) = \Sigma\ f$
by (*simp add*: Σ-*def fun-eq-iff*)

lemma Σ-*const*:
$\Sigma\ (\lambda k.\ c)\ j\ l = \textit{of-int}\ (l - j) * c$
by (*simp add*: Σ-*def algebra-simps*)

lemma Σ-*add*:
$\Sigma\ (\lambda k.\ f\ k + g\ k)\ j\ l = \Sigma\ f\ j\ l + \Sigma\ g\ j\ l$
by (*simp add*: Σ-*def sum.distrib*)

lemma Σ-*factor*:
$\Sigma\ (\lambda k.\ c * f\ k)\ j\ l = (c::'a::\textit{ring}) * \Sigma\ (\lambda k.\ f\ k)\ j\ l$
by (*simp add*: Σ-*def sum-distrib-left*)

lemma Σ-*concat*:
$\Sigma\ f\ j\ k + \Sigma\ f\ k\ l = \Sigma\ f\ j\ l$
by (*simp add*: Σ-*def algebra-simps add-sum-int*)
(*simp-all add*: *add-sum-orient* [*of* $\lambda k.\ f\ (\textit{of-int}\ k)\ k\ j\ l$]
add-sum-orient [*of* $\lambda k.\ f\ (\textit{of-int}\ k)\ j\ l\ k$]
add-sum-orient [*of* $\lambda k.\ f\ (\textit{of-int}\ k)\ j\ k\ l$] *add-sum-int*)

lemma Σ-*incr-upper*:
$\Sigma\ f\ j\ (l + 1) = \Sigma\ f\ j\ l + f\ (\textit{of-int}\ l)$
proof −
have $\{l..<l+1\} = \{l\}$ **by** *auto*
then have $\Sigma\ f\ l\ (l + 1) = f\ (\textit{of-int}\ l)$ **by** (*simp add*: Σ-*def*)
moreover have $\Sigma\ f\ j\ (l + 1) = \Sigma\ f\ j\ l + \Sigma\ f\ l\ (l + 1)$ **by** (*simp add*: Σ-*concat*)
ultimately show *?thesis* **by** *simp*
qed

2.4 Fundamental lemmas: The relation between Δ and Σ

lemma Δ-Σ:
$\Delta\ (\Sigma\ f\ j) = f$
proof
fix k
show $\Delta\ (\Sigma\ f\ j)\ k = f\ k$
by (*simp add*: Δ-*def* Σ-*incr-upper*)
qed

lemma Σ-Δ:
$\Sigma\ (\Delta\ f)\ j\ l = f\ (\textit{of-int}\ l) - f\ (\textit{of-int}\ j)$
proof −

from Δ-Σ **have** Δ (Σ (Δ *f*) *j*) = Δ *f* **.**
then obtain *k* **where** *plus k* ∘ Σ (Δ *f*) *j* ∘ *of-int* = *f* ∘ *of-int*
by (*blast dest*: Δ-*same-shift*)
then have ⋀*q*. *f* (*of-int q*) = *k* + Σ (Δ *f*) *j q*
by (*simp add*: *fun-eq-iff*)
then show *?thesis* **by** *simp*
qed

end

3 A barebone conversion for discrete summation

theory *Summation-Conversion*
imports *Factorials Discrete-Summation*
begin

Extensible theorem collection for solving summation problems

named-theorems *summation rules for solving summation problems*

declare
Σ-*const* [*summation*] Σ-*add* [*summation*]
Σ-*factor* [*summation*] *monomial-ffact* [*summation*]

lemma *intervall-simps* [*summation*]:
(∑ *k*::*nat* = *0*..*0*. *f k*) = *f 0*
(∑ *k*::*nat* = *0*..*Suc n*. *f k*) = *f* (*Suc n*) + (∑ *k*::*nat* = *0*..*n*. *f k*)
by (*simp-all add*: *add.commute*)

lemma Δ-*ffact*:
Δ (*ffact* (*Suc n*)) *k* = *of-nat* (*Suc n*) ∗ *ffact n* (*of-int k* :: *'a* :: *comm-ring-1*)
proof (*induct n*)
case *0* **then show** *?case*
by (*simp add*: Δ-*def ffact-Suc*)
next
case (*Suc n*)
obtain *m* **where** *m* = *Suc n* **by** *blast*
have Δ (*ffact* (*Suc m*)) *k* =
ffact (*Suc m*) (*of-int* (*k* + *1*)) − *ffact* (*Suc m*) (*of-int k* :: *'a*)
by (*simp add*: Δ-*def*)
also have ... = *of-int* (*k* + *1*) ∗ *ffact m* (*of-int k*)
− (*ffact m* (*of-int k*) ∗ (*of-int k* − *of-nat m*))
using *ffact-Suc-rev* [*of m* (*of-int k* :: *'a* :: *comm-ring-1*)]
by (*simp add*: *ac-simps ffact-Suc*)
also have ... = (*of-int k* + *1* − *of-int k* + *of-nat m*) ∗ *ffact m* (*of-int k*)
by (*simp add*: *algebra-simps*)
also have ... = *of-nat* (*Suc m*) ∗ *ffact m* (*of-int k*) **by** *simp*
also have ... = *of-nat* (*Suc m*) ∗ *ffact* (*Suc m* − *1*) (*of-int k*) **by** *simp*
finally show *?case* **by** (*simp only*: ‹*m* = *Suc n*› *diff-Suc-1*)
qed

lemma Σ-*ffact-divide* [*summation*]:
  Σ (*ffact n*) *j l* =
    (*ffact* (*Suc n*) (*of-int l* :: *'a* :: {*idom-divide*, *semiring-char-0*}) − *ffact* (*Suc n*) (*of-int j*)) *div of-nat* (*Suc n*)
proof −
  have ∗: (*of-nat* (*Suc n*) ∗ Σ (*ffact n*) *j l*) *div of-nat* (*Suc n*) = (Σ (*ffact n*) *j l* :: *'a*)
    using *of-nat-neq-0* [**where** *?'a* = *'a*] **by** *simp*
  have *ffact* (*Suc n*) (*of-int l* :: *'a*) − *ffact* (*Suc n*) (*of-int j*) =
    Σ (λ*k*. Δ (*ffact* (*Suc n*)) *k*) *j l*
    by (*simp add*: Σ-Δ)
  also have ... = Σ (λ*k*. *of-nat* (*Suc n*) ∗ *ffact n* (*of-int k*)) *j l*
    by (*simp add*: Δ-*ffact*)
  also have ... = *of-nat* (*Suc n*) ∗ Σ (*ffact n* ∘ *of-int*) *j l*
    by (*simp add*: Σ-*factor comp-def*)
  finally show *?thesis* **by** (*simp only*: Σ-*comp-of-int* ∗ *of-nat-eq-0-iff*)
qed

Various other rules

lemma *of-int-coeff*:
  (*of-int l* :: *'a*::*comm-ring-1*) ∗ *numeral k* = *of-int* (*l* ∗ *numeral k*)
  by *simp*

lemmas *nat-simps* =
  add-0-left add-0-right add-Suc add-Suc-right
  mult-Suc mult-Suc-right mult-zero-left mult-zero-right
  One-nat-def of-nat-simps

lemmas *of-int-pull-out* =
  of-int-add [*symmetric*] *of-int-diff* [*symmetric*] *of-int-mult* [*symmetric*]
  of-int-coeff

lemma *of-nat-coeff*:
  (*of-nat n* :: *'a*::*comm-semiring-1*) ∗ *numeral m* = *of-nat* (*n* ∗ *numeral m*)
  by (*induct n*) *simp-all*

lemmas *of-nat-pull-out* =
  of-nat-add [*symmetric*] *of-nat-mult* [*symmetric*] *of-nat-coeff*

lemmas *nat-pull-in* =
  nat-int-add

lemmas *of-int-pull-in* =
  of-int-pull-out [*symmetric*] *add-divide-distrib diff-divide-distrib of-int-power*
  of-int-numeral of-int-neg-numeral times-divide-eq-left [*symmetric*]

Special for *nat*

definition *lift-nat* :: (*nat* ⇒ *nat*) ⇒ *int* ⇒ *int*

where
lift-nat f = int ∘ f ∘ nat

definition Σ-*nat* :: (*nat* ⇒ *nat*) ⇒ *nat* ⇒ *nat* ⇒ *nat* ($\Sigma_{\mathbb{N}}$)
where
[*summation*]: $\Sigma_{\mathbb{N}}$ *f m n* = *nat* (Σ (*lift-nat f*) (*int m*) (*int n*))

definition *pos-id* :: *int* ⇒ *int*
where
pos-id k = (*if k* < *0 then 0 else k*)

lemma Σ-*pos-id* [*summation*]:
0 ≤ *k* ⟹ *0* ≤ *l* ⟹ Σ (λ*r*. *f* (*pos-id r*)) *k l* = Σ *f k l*
by (*simp add*: Σ-*def pos-id-def*)

lemma [*summation*]:
(*0*::*int*) ≤ *0*
(*0*::*int*) ≤ *1*
(*0*::*int*) ≤ *numeral m*
(*0*::*int*) ≤ *int n*
by *simp-all*

lemma [*summation*]:
lift-nat (λ*n*. *m*) = (λ*k*. *int m*)
by (*simp add*: *lift-nat-def fun-eq-iff*)

lemma [*summation*]:
lift-nat (λ*n*. *n*) = *pos-id*
by (*simp add*: *lift-nat-def fun-eq-iff pos-id-def*)

lemma [*summation*]:
lift-nat (λ*n*. *f n* + *g n*) = (λ*k*. *lift-nat f k* + *lift-nat g k*)
by (*simp add*: *lift-nat-def fun-eq-iff*)

lemma [*summation*]:
lift-nat (λ*n*. *m* ∗ *f n*) = (λ*k*. *int m* ∗ *lift-nat f k*)
by (*simp add*: *lift-nat-def fun-eq-iff*)

lemma [*summation*]:
lift-nat (λ*n*. *f n* ∗ *m*) = (λ*k*. *lift-nat f k* ∗ *int m*)
by (*simp add*: *lift-nat-def fun-eq-iff*)

lemma [*summation*]:
lift-nat (λ*n*. *f n* ^ *m*) = (λ*k*. *lift-nat f k* ^ *m*)
by (*simp add*: *lift-nat-def fun-eq-iff*)

Generic conversion

ML ‹
signature SUMMATION =

```
sig
  val conv: Proof.context -> conv
end

structure Summation : SUMMATION =
struct

val simps2 = @{thms Stirling.simps ffact-0 ffact-Suc nat-simps};
val simpset3 =
  @{context}
  |> fold Simplifier.add-simp @{thms field-simps}
  |> Simplifier.simpset-of;
val simps4 = @{thms of-int-pull-out of-nat-pull-out nat-pull-in};
val simps6 = @{thms of-int-pull-in};

fun conv ctxt =
  let
    val ctxt1 =
      ctxt
      |> put-simpset HOL-basic-ss
        |> fold Simplifier.add-simp (Named-Theorems.get ctxt @{named-theorems
summation})
    val ctxt2 =
      ctxt
      |> put-simpset HOL-basic-ss
      |> fold Simplifier.add-simp simps2
    val ctxt3 =
      ctxt
      |> put-simpset simpset3
    val ctxt4 =
      ctxt
      |> put-simpset HOL-basic-ss
      |> fold Simplifier.add-simp simps4
    val semiring-conv-base = Semiring-Normalizer.semiring-normalize-conv ctxt
      val semiring-conv = Conv.arg-conv (Conv.arg1-conv (Conv.arg-conv semir-
ing-conv-base))
      else-conv Conv.arg1-conv (Conv.arg-conv semiring-conv-base)
      else-conv semiring-conv-base
    val ctxt6 =
      ctxt
      |> put-simpset HOL-basic-ss
      |> fold Simplifier.add-simp simps6
  in
    Simplifier.rewrite ctxt1
    then-conv Simplifier.rewrite ctxt2
    then-conv Simplifier.rewrite ctxt3
    then-conv Simplifier.rewrite ctxt4
    then-conv semiring-conv
    then-conv Simplifier.rewrite ctxt6
```

end

end
›

hide-fact (**open**) *nat-simps of-int-pull-out of-int-pull-in*

end

4 Simple examples

theory *Examples*
imports *Summation-Conversion*
begin

ML ‹
Summation.conv @{*context*}
@{*cterm* Σ (λ*q*::*rat*. *q* ^ *Suc* (*Suc* (*Suc* *0*)) + *3*) *0* *j*}
›

ML ‹
Summation.conv @{*context*}
@{*cterm* Σ (λ*x*::*real*. *x* ^ *Suc* (*Suc* (*Suc* *0*)) + *3*) *0* *j*}
›

ML ‹
Summation.conv @{*context*}
@{*cterm* Σ (λ*k*::*int*. *k* ^ *Suc* (*Suc* (*Suc* *0*)) + *3*) *0* *j*}
›

ML ‹
Summation.conv @{*context*}
@{*cterm* $\Sigma_{\mathbb{N}}$ (λ*n*::*nat*. *n* ^ *Suc* (*Suc* (*Suc* *0*)) + *3*) *0* *m*}
›

end