Randomised Skip Lists

Max W. Haslbeck, Manuel Eberl

February 6, 2026

Abstract

Skip lists are sorted linked lists enhanced with shortcuts and are an alternative to binary search trees [2]. A skip lists consists of multiple levels of sorted linked lists where a list on level n is a subsequence of the list on level $n-1$. In the ideal case, elements are *skipped* in such a way that a lookup in a skip lists takes $\mathcal{O}(\log n)$ time. In a randomised skip list the skipped elements are choosen randomly.

This entry contains formalized proofs of the textbook results about the expected height and the expected length of a search path in a randomised skip list [1].

Contents

1 Indexed products of PMFs **2**
- 1.1 Definition . . . 2
- 1.2 Dependent product sets with a default . . . 4
- 1.3 Common PMF operations on products . . . 7
- 1.4 Merging and splitting PMF products . . . 11
- 1.5 Applications . . . 15

2 Auxiliary material **17**

3 Theorems about the Geometric Distribution **21**

4 Randomized Skip Lists **26**
- 4.1 Preliminaries . . . 27
- 4.2 Definition of a Randomised Skip List . . . 27
- 4.3 Height of Skip List . . . 27
- 4.4 Expected Length of Search Path . . . 38

1 Indexed products of PMFs

theory *Pi-pmf*
  imports *HOL−Probability.Probability*
begin

Conflicting notation from *HOL−Analysis.Infinite-Sum*

no-notation *Infinite-Sum.abs-summable-on* (**infixr** ‹*abs′-summable′-on*› *46*)

1.1 Definition

In analogy to Pi_M, we define an indexed product of PMFs. In the literature, this is typically called taking a vector of independent random variables. Note that the components do not have to be identically distributed.

The operation takes an explicit index set A and a function f that maps each element from A to a PMF and defines the product measure $\bigotimes_{i \in A} f(i)$, which is represented as a $('a \Rightarrow 'b)$ *pmf*.

Note that unlike Pi_M, this only works for *finite* index sets. It could be extended to countable sets and beyond, but the construction becomes somewhat more involved.

definition *Pi-pmf* :: $'a$ *set* $\Rightarrow$ $'b$ $\Rightarrow$ $('a \Rightarrow 'b$ *pmf*$)$ $\Rightarrow$ $('a \Rightarrow 'b)$ *pmf* **where**
  Pi-pmf A dflt p =
    embed-pmf (λf. *if* ($\forall x$. $x \notin A \longrightarrow f\ x = dflt$) *then* $\prod x{\in}A$. *pmf* ($p\ x$) ($f\ x$) *else 0*)

A technical subtlety that needs to be addressed is this: Intuitively, the functions in the support of a product distribution have domain A. However, since HOL is a total logic, these functions must still return *some* value for inputs outside A. The product measure Pi_M simply lets these functions return *undefined* in these cases. We chose a different solution here, which is to supply a default value *dflt* that is returned in these cases.

As one possible application, one could model the result of n different independent coin tosses as *Pi-pmf.Pi-pmf* $\{0..<n\}$ *False* (λ-. *bernoulli-pmf* (*1* / *2*)). This returns a function of type *nat* $\Rightarrow$ *bool* that maps every natural number below n to the result of the corresponding coin toss, and every other natural number to *False*.

lemma *pmf-Pi*:
  assumes *A*: *finite A*
  shows *pmf* (*Pi-pmf A dflt p*) f =
    (*if* ($\forall x$. $x \notin A \longrightarrow f\ x = dflt$) *then* $\prod x{\in}A$. *pmf* ($p\ x$) ($f\ x$) *else 0*)
  unfolding *Pi-pmf-def*
proof (*rule pmf-embed-pmf*, *goal-cases*)
  case *2*
  define S **where** $S = \{f.\ \forall x.\ x \notin A \longrightarrow f\ x = dflt\}$
  define B **where** $B = (\lambda x.$ *set-pmf* ($p\ x$))

have *neutral-left*: $(\prod xa{\in}A.\ pmf\ (p\ xa)\ (f\ xa)) = 0$
  if $f \in PiE\ A\ B - (\lambda f.\ restrict\ f\ A)\ `\ S$ **for** f
proof −
  have $restrict\ (\lambda x.\ if\ x \in A\ then\ f\ x\ else\ dflt)\ A \in (\lambda f.\ restrict\ f\ A)\ `\ S$
    by (*intro imageI*) (*auto simp*: *S-def*)
  also have $restrict\ (\lambda x.\ if\ x \in A\ then\ f\ x\ else\ dflt)\ A = f$
    using *that* **by** (*auto simp*: *PiE-def Pi-def extensional-def fun-eq-iff*)
  finally show *?thesis* **using** *that* **by** *blast*
qed
have *neutral-right*: $(\prod xa{\in}A.\ pmf\ (p\ xa)\ (f\ xa)) = 0$
  if $f \in (\lambda f.\ restrict\ f\ A)\ `\ S - PiE\ A\ B$ **for** f
proof −
  from *that* **obtain** f' **where** f': $f = restrict\ f'\ A\ f' \in S$ **by** *auto*
  moreover from *this* **and** *that* **have** $restrict\ f'\ A \notin PiE\ A\ B$ **by** *simp*
  then obtain x **where** $x \in A\ pmf\ (p\ x)\ (f'\ x) = 0$ **by** (*auto simp*: *B-def set-pmf-eq*)
  with f' **and** A **show** *?thesis* **by** *auto*
qed

have $(\lambda f.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$ *abs-summable-on PiE A B*
  by (*intro abs-summable-on-prod-PiE A*) (*auto simp*: *B-def*)
also have *?this* $\longleftrightarrow (\lambda f.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$ *abs-summable-on* $(\lambda f.\ restrict\ f\ A)\ `\ S$
  by (*intro abs-summable-on-cong-neutral neutral-left neutral-right*) *auto*
also have $\ldots \longleftrightarrow (\lambda f.\ \prod x{\in}A.\ pmf\ (p\ x)\ (restrict\ f\ A\ x))$ *abs-summable-on S*
  by (*rule abs-summable-on-reindex-iff* [*symmetric*]) (*force simp*: *inj-on-def fun-eq-iff S-def*)
also have $\ldots \longleftrightarrow (\lambda f.\ if\ \forall x.\ x \notin A \longrightarrow f\ x = dflt\ then\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x)\ else\ 0)$
  abs-summable-on UNIV
  by (*intro abs-summable-on-cong-neutral*) (*auto simp*: *S-def*)
finally have *summable*: $\ldots$ **.**

have $1 = (\prod x{\in}A.\ 1{::}real)$ **by** *simp*
also have $(\prod x{\in}A.\ 1) = (\prod x{\in}A.\ \sum_a y{\in}B\ x.\ pmf\ (p\ x)\ y)$
  unfolding *B-def* **by** (*subst infsetsum-pmf-eq-1*) *auto*
also have $(\prod x{\in}A.\ \sum_a y{\in}B\ x.\ pmf\ (p\ x)\ y) = (\sum_a f{\in}Pi_E\ A\ B.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$
  by (*intro infsetsum-prod-PiE* [*symmetric*] *A*) (*auto simp*: *B-def*)
also have $\ldots = (\sum_a f{\in}(\lambda f.\ restrict\ f\ A)\ `\ S.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$ **using** A
  by (*intro infsetsum-cong-neutral neutral-left neutral-right refl*)
also have $\ldots = (\sum_a f{\in}S.\ \prod x{\in}A.\ pmf\ (p\ x)\ (restrict\ f\ A\ x))$
  by (*rule infsetsum-reindex*) (*force simp*: *inj-on-def fun-eq-iff S-def*)
also have $\ldots = (\sum_a f{\in}S.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$
  by (*intro infsetsum-cong*) (*auto simp*: *S-def*)
also have $\ldots = (\sum_a f.\ if\ \forall x.\ x \notin A \longrightarrow f\ x = dflt\ then\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x)\ else\ 0)$
  by (*intro infsetsum-cong-neutral*) (*auto simp*: *S-def*)

also have *ennreal* ... $= (\int^+ f.\ ennreal\ (if\ \forall x.\ x \notin A \longrightarrow f\ x = dflt$
$then\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x)\ else\ 0)\ \partial count\text{-}space\ UNIV)$
by (*intro nn-integral-conv-infsetsum* [*symmetric*] *summable*) (*auto simp*: *prod-nonneg*)
finally show *?case* **by** *simp*
qed (*auto simp*: *prod-nonneg*)

lemma *pmf-Pi'*:
assumes *finite A* $\bigwedge x.\ x \notin A \Longrightarrow f\ x = dflt$
shows $pmf\ (Pi\text{-}pmf\ A\ dflt\ p)\ f = (\prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$
using *assms* **by** (*subst pmf-Pi*) *auto*

lemma *pmf-Pi-outside*:
assumes *finite A* $\exists x.\ x \notin A \wedge f\ x \neq dflt$
shows $pmf\ (Pi\text{-}pmf\ A\ dflt\ p)\ f = 0$
using *assms* **by** (*subst pmf-Pi*) *auto*

lemma *pmf-Pi-empty* [*simp*]: $Pi\text{-}pmf\ \{\}\ dflt\ p = return\text{-}pmf\ (\lambda\text{-}.\ dflt)$
by (*intro pmf-eqI*, *subst pmf-Pi*) (*auto simp*: *indicator-def*)

lemma *set-Pi-pmf-subset*: $finite\ A \Longrightarrow set\text{-}pmf\ (Pi\text{-}pmf\ A\ dflt\ p) \subseteq \{f.\ \forall x.\ x \notin A \longrightarrow f\ x = dflt\}$
by (*auto simp*: *set-pmf-eq pmf-Pi*)

lemma *Pi-pmf-cong* [*cong*]:
assumes $A = A'\ dflt = dflt'\ \bigwedge x.\ x \in A \Longrightarrow f\ x = f'\ x$
shows $Pi\text{-}pmf\ A\ dflt\ f = Pi\text{-}pmf\ A'\ dflt'\ f'$
proof −
have $(\lambda g.\ \prod x{\in}A.\ pmf\ (f\ x)\ (g\ x)) = (\lambda g.\ \prod x{\in}A.\ pmf\ (f'\ x)\ (g\ x))$
by (*intro ext prod.cong*) (*auto simp*: *assms*)
with *assms* **show** *?thesis* **by** (*simp add*: *Pi-pmf-def cong*: *if-cong*)
qed

1.2 Dependent product sets with a default

The following describes a dependent product of sets where the functions are required to return the default value *dflt* outside their domain, in analogy to Pi_E, which uses *undefined*.

definition *PiE-dflt*
where $PiE\text{-}dflt\ A\ dflt\ B = \{f.\ \forall x.\ (x \in A \longrightarrow f\ x \in B\ x) \wedge (x \notin A \longrightarrow f\ x = dflt)\}$

lemma *restrict-PiE-dflt*: $(\lambda h.\ restrict\ h\ A)\ `\ PiE\text{-}dflt\ A\ dflt\ B = PiE\ A\ B$
proof (*intro equalityI subsetI*)
fix *h* **assume** $h \in (\lambda h.\ restrict\ h\ A)\ `\ PiE\text{-}dflt\ A\ dflt\ B$
thus $h \in PiE\ A\ B$
by (*auto simp*: *PiE-dflt-def*)
next
fix *h* **assume** *h*: $h \in PiE\ A\ B$

hence *restrict* (λx. *if* $x \in A$ *then* $h\ x$ *else dflt*) $A \in$ (λh. *restrict* $h\ A$) ‘ *PiE-dflt* A *dflt* B
by (*intro imageI*) (*auto simp*: *PiE-def extensional-def PiE-dflt-def*)
also have *restrict* (λx. *if* $x \in A$ *then* $h\ x$ *else dflt*) $A = h$
using h **by** (*auto simp*: *fun-eq-iff*)
finally show $h \in$ (λh. *restrict* $h\ A$) ‘ *PiE-dflt* A *dflt* B **.**
qed

lemma *dflt-image-PiE*: ($\lambda h\ x$. *if* $x \in A$ *then* $h\ x$ *else dflt*) ‘ *PiE* $A\ B$ = *PiE-dflt* A *dflt* B
(**is** *?f* ‘ *?X* = *?Y*)
proof (*intro equalityI subsetI*)
fix h **assume** $h \in$ *?f* ‘ *?X*
thus $h \in$ *?Y*
by (*auto simp*: *PiE-dflt-def PiE-def*)
next
fix h **assume** h: $h \in$ *?Y*
hence *?f* (*restrict* $h\ A$) $\in$ *?f* ‘ *?X*
by (*intro imageI*) (*auto simp*: *PiE-def extensional-def PiE-dflt-def*)
also have *?f* (*restrict* $h\ A$) = h
using h **by** (*auto simp*: *fun-eq-iff PiE-dflt-def*)
finally show $h \in$ *?f* ‘ *?X* **.**
qed

lemma *finite-PiE-dflt* [*intro*]:
assumes *finite* A $\bigwedge x.\ x \in A \Longrightarrow$ *finite* ($B\ x$)
shows *finite* (*PiE-dflt* $A\ d\ B$)
proof −
have *PiE-dflt* $A\ d\ B$ = ($\lambda f\ x$. *if* $x \in A$ *then* $f\ x$ *else* d) ‘ *PiE* $A\ B$
by (*rule dflt-image-PiE* [*symmetric*])
also have *finite* ...
by (*intro finite-imageI finite-PiE assms*)
finally show *?thesis* **.**
qed

lemma *card-PiE-dflt*:
assumes *finite* A $\bigwedge x.\ x \in A \Longrightarrow$ *finite* ($B\ x$)
shows *card* (*PiE-dflt* $A\ d\ B$) = ($\prod x{\in}A$. *card* ($B\ x$))
proof −
from *assms* **have** ($\prod x{\in}A$. *card* ($B\ x$)) = *card* (*PiE* $A\ B$)
by (*intro card-PiE* [*symmetric*]) *auto*
also have *PiE* $A\ B$ = (λf. *restrict* $f\ A$) ‘ *PiE-dflt* $A\ d\ B$
by (*rule restrict-PiE-dflt* [*symmetric*])
also have *card* ... = *card* (*PiE-dflt* $A\ d\ B$)
by (*intro card-image*) (*force simp*: *inj-on-def restrict-def fun-eq-iff PiE-dflt-def*)
finally show *?thesis* **..**
qed

lemma *PiE-dflt-empty-iff* [*simp*]: *PiE-dflt* A *dflt* $B = \{\} \longleftrightarrow (\exists x{\in}A.\ B\ x = \{\})$

by (*simp add*: *dflt-image-PiE* [*symmetric*] *PiE-eq-empty-iff*)

The probability of an independent combination of events is precisely the product of the probabilities of each individual event.

lemma *measure-Pi-pmf-PiE-dflt*:
assumes [*simp*]: *finite A*
shows *measure-pmf.prob* (*Pi-pmf A dflt p*) (*PiE-dflt A dflt B*) =
($\prod x{\in}A$. *measure-pmf.prob* (*p x*) (*B x*))
proof −
define B' **where** $B' = (\lambda x.\ B\ x \cap$ *set-pmf* (*p x*))
have *measure-pmf.prob* (*Pi-pmf A dflt p*) (*PiE-dflt A dflt B*) =
($\sum_a h{\in}$*PiE-dflt A dflt B*. *pmf* (*Pi-pmf A dflt p*) *h*)
by (*rule measure-pmf-conv-infsetsum*)
also have ... = ($\sum_a h{\in}$*PiE-dflt A dflt B*. $\prod x{\in}A$. *pmf* (*p x*) (*h x*))
by (*intro infsetsum-cong*, *subst pmf-Pi'*) (*auto simp*: *PiE-dflt-def*)
also have ... = ($\sum_a h{\in}(\lambda h.$ *restrict h A*) ‘ *PiE-dflt A dflt B*. $\prod x{\in}A$. *pmf* (*p x*) (*h x*))
by (*subst infsetsum-reindex*) (*force simp*: *inj-on-def PiE-dflt-def fun-eq-iff*)+
also have ($\lambda h.$ *restrict h A*) ‘ *PiE-dflt A dflt B* = *PiE A B*
by (*rule restrict-PiE-dflt*)
also have ($\sum_a h{\in}$*PiE A B*. $\prod x{\in}A$. *pmf* (*p x*) (*h x*)) = ($\sum_a h{\in}$*PiE A B'*. $\prod x{\in}A$. *pmf* (*p x*) (*h x*))
by (*intro infsetsum-cong-neutral*) (*auto simp*: *B'-def set-pmf-eq*)
also have ($\sum_a h{\in}$*PiE A B'*. $\prod x{\in}A$. *pmf* (*p x*) (*h x*)) = ($\prod x{\in}A$. *infsetsum* (*pmf* (*p x*)) (*B' x*))
by (*intro infsetsum-prod-PiE*) (*auto simp*: *B'-def*)
also have ... = ($\prod x{\in}A$. *infsetsum* (*pmf* (*p x*)) (*B x*))
by (*intro prod.cong infsetsum-cong-neutral*) (*auto simp*: *B'-def set-pmf-eq*)
also have ... = ($\prod x{\in}A$. *measure-pmf.prob* (*p x*) (*B x*))
by (*subst measure-pmf-conv-infsetsum*) (*rule refl*)
finally show *?thesis* **.**
qed

lemma *set-Pi-pmf-subset'*:
assumes *finite A*
shows *set-pmf* (*Pi-pmf A dflt p*) $\subseteq$ *PiE-dflt A dflt* (*set-pmf* $\circ$ *p*)
using *assms* **by** (*auto simp*: *set-pmf-eq pmf-Pi PiE-dflt-def*)

lemma *Pi-pmf-return-pmf* [*simp*]:
assumes *finite A*
shows *Pi-pmf A dflt* ($\lambda x.$ *return-pmf* (*f x*)) = *return-pmf* ($\lambda x.$ *if* $x \in A$ *then f x else dflt*)
proof −
have *set-pmf* (*Pi-pmf A dflt* ($\lambda x.$ *return-pmf* (*f x*))) $\subseteq$
PiE-dflt A dflt (*set-pmf* $\circ$ ($\lambda x.$ *return-pmf* (*f x*)))
by (*intro set-Pi-pmf-subset' assms*)
also have ... $\subseteq$ {$\lambda x.$ *if* $x \in A$ *then f x else dflt*}
by (*auto simp*: *PiE-dflt-def*)
finally show *?thesis*

```
  by (simp add: set-pmf-subset-singleton)
qed

lemma Pi-pmf-return-pmf' [simp]:
  assumes finite A
  shows   Pi-pmf A dflt (λ-. return-pmf dflt) = return-pmf (λ-. dflt)
  using assms by simp

lemma measure-Pi-pmf-Pi:
  fixes t::nat
  assumes [simp]: finite A
  shows   measure-pmf.prob (Pi-pmf A dflt p) (Pi A B) =
          (∏x∈A. measure-pmf.prob (p x) (B x)) (is ?lhs = ?rhs)
proof −
  have ?lhs = measure-pmf.prob (Pi-pmf A dflt p) (PiE-dflt A dflt B)
    by (intro measure-prob-cong-0)
       (auto simp: PiE-dflt-def PiE-def intro!: pmf-Pi-outside)+
  also have ... = ?rhs
    using assms by (simp add: measure-Pi-pmf-PiE-dflt)
  finally show ?thesis
    by simp
qed
```

1.3 Common PMF operations on products

Pi-pmf.Pi-pmf distributes over the 'bind' operation in the Giry monad:

```
lemma Pi-pmf-bind:
  assumes finite A
  shows   Pi-pmf A d (λx. bind-pmf (p x) (q x)) =
          do {f ← Pi-pmf A d' p; Pi-pmf A d (λx. q x (f x))} (is ?lhs = ?rhs)
proof (rule pmf-eqI, goal-cases)
  case (1 f)
  show ?case
  proof (cases ∃x∈−A. f x ≠ d)
    case False
    define B where B = (λx. set-pmf (p x))
    have [simp]: countable (B x) for x by (auto simp: B-def)

    {
      fix x :: 'a
      have (λa. pmf (p x) a * 1) abs-summable-on B x
        by (simp add: pmf-abs-summable)
      moreover have norm (pmf (p x) a * 1) ≥ norm (pmf (p x) a * pmf (q x
a) (f x)) for a
        unfolding norm-mult by (intro mult-left-mono) (auto simp: pmf-le-1)
      ultimately have (λa. pmf (p x) a * pmf (q x a) (f x)) abs-summable-on B x
        by (rule abs-summable-on-comparison-test)
    } note summable = this
```

have *pmf ?rhs f* = ($\sum_a g$. *pmf* (*Pi-pmf A d′ p*) *g* ∗ ($\prod x{\in}A$. *pmf* (*q x* (*g x*)) (*f x*)))
by (*subst pmf-bind*, *subst pmf-Pi′*)
(*insert assms False*, *simp-all add*: *pmf-expectation-eq-infsetsum*)
also have ... = ($\sum_a g{\in}$*PiE-dflt A d′ B*.
pmf (*Pi-pmf A d′ p*) *g* ∗ ($\prod x{\in}A$. *pmf* (*q x* (*g x*)) (*f x*)))
unfolding *B-def*
using *assms* **by** (*intro infsetsum-cong-neutral*) (*auto simp*: *pmf-Pi PiE-dflt-def set-pmf-eq*)
also have ... = ($\sum_a g{\in}$*PiE-dflt A d′ B*.
($\prod x{\in}A$. *pmf* (*p x*) (*g x*) ∗ *pmf* (*q x* (*g x*)) (*f x*)))
using *assms* **by** (*intro infsetsum-cong*) (*auto simp*: *pmf-Pi PiE-dflt-def prod.distrib*)
also have ... = ($\sum_a g{\in}$(λ*g*. *restrict g A*) ‘ *PiE-dflt A d′ B*.
($\prod x{\in}A$. *pmf* (*p x*) (*g x*) ∗ *pmf* (*q x* (*g x*)) (*f x*)))
by (*subst infsetsum-reindex*) (*force simp*: *PiE-dflt-def inj-on-def fun-eq-iff*)+
also have (λ*g*. *restrict g A*) ‘ *PiE-dflt A d′ B* = *PiE A B*
by (*rule restrict-PiE-dflt*)
also have ($\sum_a g{\in}$.... ($\prod x{\in}A$. *pmf* (*p x*) (*g x*) ∗ *pmf* (*q x* (*g x*)) (*f x*))) =
($\prod x{\in}A$. $\sum_a a{\in}B$ *x*. *pmf* (*p x*) *a* ∗ *pmf* (*q x a*) (*f x*))
using *assms summable* **by** (*subst infsetsum-prod-PiE*) *simp-all*
also have ... = ($\prod x{\in}A$. $\sum_a a$. *pmf* (*p x*) *a* ∗ *pmf* (*q x a*) (*f x*))
by (*intro prod.cong infsetsum-cong-neutral*) (*auto simp*: *B-def set-pmf-eq*)
also have ... = *pmf ?lhs f*
using *False assms* **by** (*subst pmf-Pi′*) (*simp-all add*: *pmf-bind pmf-expectation-eq-infsetsum*)
finally show *?thesis* **..**
next
case *True*
have *pmf ?rhs f* =
measure-pmf.expectation (*Pi-pmf A d′ p*) (λ*x*. *pmf* (*Pi-pmf A d* (λ*xa*. *q xa* (*x xa*))) *f*)
using *assms* **by** (*simp add*: *pmf-bind*)
also have ... = *measure-pmf.expectation* (*Pi-pmf A d′ p*) (λ*x*. *0*)
using *assms True* **by** (*intro Bochner-Integration.integral-cong pmf-Pi-outside*) *auto*
also have ... = *pmf ?lhs f*
using *assms True* **by** (*subst pmf-Pi-outside*) *auto*
finally show *?thesis* **..**
qed
qed

Analogously any componentwise mapping can be pulled outside the product:

lemma *Pi-pmf-map*:
assumes [*simp*]: *finite A* **and** *f dflt* = *dflt′*
shows *Pi-pmf A dflt′* (λ*x*. *map-pmf f* (*g x*)) = *map-pmf* (λ*h*. *f* ∘ *h*) (*Pi-pmf A dflt g*)
proof −
have *Pi-pmf A dflt′* (λ*x*. *map-pmf f* (*g x*)) =
Pi-pmf A dflt′ (λ*x*. *g x* ≫= (λ*x*. *return-pmf* (*f x*)))

```
    using assms by (simp add: map-pmf-def Pi-pmf-bind)
  also have ... = Pi-pmf A dflt g ≫= (λh. return-pmf (λx. if x ∈ A then f (h x)
else dflt'))
   by (subst Pi-pmf-bind[where d' = dflt]) auto
  also have ... = map-pmf (λh. f ∘ h) (Pi-pmf A dflt g)
    unfolding map-pmf-def using set-Pi-pmf-subset'[of A dflt g]
    by (intro bind-pmf-cong refl arg-cong[of - - return-pmf])
       (auto dest: simp: fun-eq-iff PiE-dflt-def assms(2))
  finally show ?thesis .
qed
```

We can exchange the default value in a product of PMFs like this:

```
lemma Pi-pmf-default-swap:
  assumes finite A
  shows   map-pmf (λf x. if x ∈ A then f x else dflt') (Pi-pmf A dflt p) =
          Pi-pmf A dflt' p (is ?lhs = ?rhs)
proof (rule pmf-eqI, goal-cases)
  case (1 f)
  let ?B = (λf x. if x ∈ A then f x else dflt') −‘ {f} ∩ PiE-dflt A dflt (λ-. UNIV)
  show ?case
  proof (cases ∃x∈−A. f x ≠ dflt')
    case False
    let ?f' = λx. if x ∈ A then f x else dflt
    from False have pmf ?lhs f = measure-pmf.prob (Pi-pmf A dflt p) ?B
      using assms unfolding pmf-map
      by (intro measure-prob-cong-0) (auto simp: PiE-dflt-def pmf-Pi-outside)
    also from False have ?B = {?f'}
      by (auto simp: fun-eq-iff PiE-dflt-def)
    also have measure-pmf.prob (Pi-pmf A dflt p) {?f'} = pmf (Pi-pmf A dflt p)
?f'
      by (simp add: measure-pmf-single)
    also have ... = pmf ?rhs f
      using False assms by (subst (1 2) pmf-Pi) auto
    finally show ?thesis .
  next
    case True
    have pmf ?lhs f = measure-pmf.prob (Pi-pmf A dflt p) ?B
      using assms unfolding pmf-map
      by (intro measure-prob-cong-0) (auto simp: PiE-dflt-def pmf-Pi-outside)
    also from True have ?B = {} by auto
    also have measure-pmf.prob (Pi-pmf A dflt p) ... = 0
      by simp
    also have 0 = pmf ?rhs f
      using True assms by (intro pmf-Pi-outside [symmetric]) auto
    finally show ?thesis .
  qed
qed
```

The following rule allows reindexing the product:

```
lemma Pi-pmf-bij-betw:
  assumes finite A bij-betw h A B ⋀x. x ∉ A ⟹ h x ∉ B
  shows Pi-pmf A dflt (λ-. f) = map-pmf (λg. g ∘ h) (Pi-pmf B dflt (λ-. f))
    (is ?lhs = ?rhs)
proof −
  have B: finite B
    using assms bij-betw-finite by auto
  have pmf ?lhs g = pmf ?rhs g for g
  proof (cases ∀ a. a ∉ A ⟶ g a = dflt)
    case True
    define h′ where h′ = the-inv-into A h
    have h′: h′ (h x) = x if x ∈ A for x
     unfolding h′-def using that assms by (auto simp add: bij-betw-def the-inv-into-f-f)
    have h: h (h′ x) = x if x ∈ B for x
      unfolding h′-def using that assms f-the-inv-into-f-bij-betw by fastforce
     have pmf ?rhs g = measure-pmf.prob (Pi-pmf B dflt (λ-. f)) ((λg. g ∘ h) −‘
{g})
      unfolding pmf-map by simp
    also have ... = measure-pmf.prob (Pi-pmf B dflt (λ-. f))
                          (((λg. g ∘ h) −‘ {g}) ∩ PiE-dflt B dflt (λ-. UNIV))
     using B by (intro measure-prob-cong-0) (auto simp: PiE-dflt-def pmf-Pi-outside)
     also have ... = pmf (Pi-pmf B dflt (λ-. f)) (λx. if x ∈ B then g (h′ x) else
dflt)
    proof −
      have (if h x ∈ B then g (h′ (h x)) else dflt) = g x for x
        using h′ assms True by (cases x ∈ A) (auto simp add: bij-betwE)
      then have (λg. g ∘ h) −‘ {g} ∩ PiE-dflt B dflt (λ-. UNIV) =
            {(λx. if x ∈ B then g (h′ x) else dflt)}
        using assms h′ h True unfolding PiE-dflt-def by auto
      then show ?thesis
        by (simp add: measure-pmf-single)
    qed
    also have ... = pmf (Pi-pmf A dflt (λ-. f)) g
      using B assms True  h′-def
      by (auto simp add: pmf-Pi intro!: prod.reindex-bij-betw bij-betw-the-inv-into)
    finally show ?thesis
      by simp
  next
    case False
   have pmf ?rhs g = infsetsum (pmf (Pi-pmf B dflt (λ-. f))) ((λg. g ∘ h) −‘ {g})
      using assms by (auto simp add: measure-pmf-conv-infsetsum pmf-map)
    also have ... = infsetsum (λ-. 0) ((λg x. g (h x)) −‘ {g})
      using B False assms by (intro infsetsum-cong pmf-Pi-outside) fastforce+
    also have ... = 0
      by simp
    finally show ?thesis
      using assms False by (auto simp add: pmf-Pi pmf-map)
  qed
  then show ?thesis
```

```
    by (rule pmf-eqI)
qed
```

A product of uniform random choices is again a uniform distribution.

```
lemma Pi-pmf-of-set:
  assumes finite A ⋀x. x ∈ A ⟹ finite (B x) ⋀x. x ∈ A ⟹ B x ≠ {}
  shows   Pi-pmf A d (λx. pmf-of-set (B x)) = pmf-of-set (PiE-dflt A d B) (is
?lhs = ?rhs)
proof (rule pmf-eqI, goal-cases)
  case (1 f)
  show ?case
  proof (cases ∃x. x ∉ A ∧ f x ≠ d)
    case True
    hence pmf ?lhs f = 0
      using assms by (intro pmf-Pi-outside) (auto simp: PiE-dflt-def)
    also from True have f ∉ PiE-dflt A d B
      by (auto simp: PiE-dflt-def)
    hence 0 = pmf ?rhs f
      using assms by (subst pmf-of-set) auto
    finally show ?thesis .
  next
    case False
    hence pmf ?lhs f = (∏x∈A. pmf (pmf-of-set (B x)) (f x))
      using assms by (subst pmf-Pi') auto
    also have ... = (∏x∈A. indicator (B x) (f x) / real (card (B x)))
      by (intro prod.cong refl, subst pmf-of-set) (use assms False in auto)
    also have ... = (∏x∈A. indicator (B x) (f x)) / real (∏x∈A. card (B x))
      by (subst prod-divide f) simp-all
    also have (∏x∈A. indicator (B x) (f x) :: real) = indicator (PiE-dflt A d B) f
      using assms False by (auto simp: indicator-def PiE-dflt-def)
    also have (∏x∈A. card (B x)) = card (PiE-dflt A d B)
      using assms by (intro card-PiE-dflt [symmetric]) auto
    also have indicator (PiE-dflt A d B) f / ... = pmf ?rhs f
      using assms by (intro pmf-of-set [symmetric]) auto
    finally show ?thesis .
  qed
qed
```

1.4 Merging and splitting PMF products

The following lemma shows that we can add a single PMF to a product:

```
lemma Pi-pmf-insert:
  assumes finite A x ∉ A
  shows   Pi-pmf (insert x A) dflt p = map-pmf (λ(y,f). f(x:=y)) (pair-pmf (p
x) (Pi-pmf A dflt p))
proof (intro pmf-eqI)
  fix f
  let ?M = pair-pmf (p x) (Pi-pmf A dflt p)
  have pmf (map-pmf (λ(y, f). f(x := y)) ?M) f =
```

measure-pmf.prob ?M $((\lambda(y, f).\ f(x := y)) -`\ \{f\})$
by (*subst pmf-map*) *auto*
also have $((\lambda(y, f).\ f(x := y)) -`\ \{f\}) = (\bigcup y'.\ \{(f\ x, f(x := y'))\})$
by (*auto simp*: *fun-upd-def fun-eq-iff*)
also have *measure-pmf.prob ?M* ... = *measure-pmf.prob ?M* $\{(f\ x, f(x := dflt))\}$
using *assms* **by** (*intro measure-prob-cong-0*) (*auto simp*: *pmf-pair pmf-Pi split*: *if-splits*)
also have ... = *pmf* (*p x*) (*f x*) ∗ *pmf* (*Pi-pmf A dflt p*) ($f(x := dflt)$)
by (*simp add*: *measure-pmf-single pmf-pair pmf-Pi*)
also have ... = *pmf* (*Pi-pmf* (*insert x A*) *dflt p*) *f*
proof (*cases* $\forall y.\ y \notin$ *insert x A* $\longrightarrow f\ y = dflt$)
case *True*
with *assms* **have** *pmf* (*p x*) (*f x*) ∗ *pmf* (*Pi-pmf A dflt p*) ($f(x := dflt)$) =
pmf (*p x*) (*f x*) ∗ ($\prod xa{\in}A$. *pmf* (*p xa*) (($f(x := dflt)$) *xa*))
by (*subst pmf-Pi′*) *auto*
also have ($\prod xa{\in}A$. *pmf* (*p xa*) (($f(x := dflt)$) *xa*)) = ($\prod xa{\in}A$. *pmf* (*p xa*) (*f xa*))
using *assms* **by** (*intro prod.cong*) *auto*
also have *pmf* (*p x*) (*f x*) ∗ ... = *pmf* (*Pi-pmf* (*insert x A*) *dflt p*) *f*
using *assms True* **by** (*subst pmf-Pi′*) *auto*
finally show *?thesis* **.**
qed (*insert assms*, *auto simp*: *pmf-Pi*)
finally show ... = *pmf* (*map-pmf* ($\lambda(y, f).\ f(x := y)$) *?M*) *f* **..**
qed

lemma *Pi-pmf-insert′*:
assumes *finite A* $x \notin A$
shows *Pi-pmf* (*insert x A*) *dflt p* =
do {*y* ← *p x*; *f* ← *Pi-pmf A dflt p*; *return-pmf* ($f(x := y)$)}
using *assms*
by (*subst Pi-pmf-insert*)
(*auto simp add*: *map-pmf-def pair-pmf-def case-prod-beta′ bind-return-pmf bind-assoc-pmf*)

lemma *Pi-pmf-singleton*:
Pi-pmf {*x*} *dflt p* = *map-pmf* (λ*a b*. *if b* = *x then a else dflt*) (*p x*)
proof −
have *Pi-pmf* {*x*} *dflt p* = *map-pmf* (*fun-upd* (λ-. *dflt*) *x*) (*p x*)
by (*subst Pi-pmf-insert*) (*simp-all add*: *pair-return-pmf2 pmf.map-comp o-def*)
also have *fun-upd* (λ-. *dflt*) *x* = (λ*z y*. *if y* = *x then z else dflt*)
by (*simp add*: *fun-upd-def fun-eq-iff*)
finally show *?thesis* **.**
qed

Projecting a product of PMFs onto a component yields the expected result:

lemma *Pi-pmf-component*:
assumes *finite A*
shows *map-pmf* (λ*f*. *f x*) (*Pi-pmf A dflt p*) = (*if x* ∈ *A then p x else return-pmf dflt*)

proof (*cases* $x \in A$)
  case *True*
  define A' **where** $A' = A - \{x\}$
  from *assms* **and** *True* **have** A': $A = insert\ x\ A'$
    by (*auto simp*: A'*-def*)
  from *assms* **have** *map-pmf* ($\lambda f.\ f\ x$) (*Pi-pmf A dflt p*) = *p x* **unfolding** A'
    by (*subst Pi-pmf-insert*)
      (*auto simp*: A'*-def pmf.map-comp o-def case-prod-unfold map-fst-pair-pmf*)
  with *True* **show** *?thesis* **by** *simp*
next
  case *False*
  have *map-pmf* ($\lambda f.\ f\ x$) (*Pi-pmf A dflt p*) = *map-pmf* (λ-. *dflt*) (*Pi-pmf A dflt p*)
    using *assms False set-Pi-pmf-subset*[*of A dflt p*]
    by (*intro pmf.map-cong refl*) (*auto simp*: *set-pmf-eq pmf-Pi-outside*)
  with *False* **show** *?thesis* **by** *simp*
qed

We can take merge two PMF products on disjoint sets like this:

lemma *Pi-pmf-union*:
  assumes *finite A finite B* $A \cap B = \{\}$
  shows *Pi-pmf* $(A \cup B)$ *dflt p* =
    map-pmf ($\lambda(f,g)\ x.$ *if* $x \in A$ *then f x else g x*)
    (*pair-pmf* (*Pi-pmf A dflt p*) (*Pi-pmf B dflt p*)) (**is** - = *map-pmf* (*?h A*) (*?q A*))
  using *assms*(*1*,*3*)
proof (*induction rule*: *finite-induct*)
  case (*insert x A*)
  have *map-pmf* (*?h* (*insert x A*)) (*?q* (*insert x A*)) =
      do {$v \leftarrow p\ x$; $(f,\ g) \leftarrow$ *pair-pmf* (*Pi-pmf A dflt p*) (*Pi-pmf B dflt p*);
        return-pmf ($\lambda y.$ *if* $y \in$ *insert x A then* $(f(x := v))\ y$ *else g y*)}
    by (*subst Pi-pmf-insert*)
      (*insert insert.hyps insert.prems*,
        simp-all add: *pair-pmf-def map-bind-pmf bind-map-pmf bind-assoc-pmf bind-return-pmf*)
  also have ... = *do* {$v \leftarrow p\ x$; $(f,\ g) \leftarrow$ *?q A*; *return-pmf* ((*?h A* (f,g))$(x := v)$)}
    by (*intro bind-pmf-cong refl*) (*auto simp*: *fun-eq-iff*)
  also have ... = *do* {$v \leftarrow p\ x$; $f \leftarrow$ *map-pmf* (*?h A*) (*?q A*); *return-pmf* ($f(x := v)$)}
    by (*simp add*: *bind-map-pmf map-bind-pmf case-prod-unfold cong*: *if-cong*)
  also have ... = *do* {$v \leftarrow p\ x$; $f \leftarrow$ *Pi-pmf* $(A \cup B)$ *dflt p*; *return-pmf* ($f(x := v)$)}
    using *insert.hyps* **and** *insert.prems* **by** (*intro bind-pmf-cong insert.IH* [*symmetric*] *refl*) *auto*
  also have ... = *Pi-pmf* (*insert x* $(A \cup B)$) *dflt p*
    by (*subst Pi-pmf-insert*)
      (*insert assms insert.hyps insert.prems*, *auto simp*: *pair-pmf-def map-bind-pmf*)
  also have *insert x* $(A \cup B)$ = *insert x A* $\cup$ *B*
    by *simp*

finally show *?case* **..**
qed (*simp-all add*: *case-prod-unfold map-snd-pair-pmf*)

We can also project a product to a subset of the indices by mapping all the other indices to the default value:

lemma *Pi-pmf-subset*:
assumes *finite A* $A' \subseteq A$
shows *Pi-pmf* A' *dflt p* = *map-pmf* ($\lambda f\ x$. *if* $x \in A'$ *then f x else dflt*) (*Pi-pmf A dflt p*)
proof −
let *?P* = *pair-pmf* (*Pi-pmf* A' *dflt p*) (*Pi-pmf* ($A - A'$) *dflt p*)
from *assms* **have** [*simp*]: *finite* A'
by (*blast dest*: *finite-subset*)
from *assms* **have** $A = A' \cup (A - A')$
by *blast*
also have *Pi-pmf* ... *dflt p* = *map-pmf* ($\lambda(f,g)\ x$. *if* $x \in A'$ *then f x else g x*) *?P*
using *assms* **by** (*intro Pi-pmf-union*) *auto*
also have *map-pmf* ($\lambda f\ x$. *if* $x \in A'$ *then f x else dflt*) ... = *map-pmf fst ?P*
unfolding *map-pmf-comp o-def case-prod-unfold*
using *set-Pi-pmf-subset*[*of* A' *dflt p*] **by** (*intro map-pmf-cong refl*) (*auto simp*: *fun-eq-iff*)
also have ... = *Pi-pmf* A' *dflt p*
by (*simp add*: *map-fst-pair-pmf*)
finally show *?thesis* **..**
qed

lemma *Pi-pmf-subset'*:
fixes *f* :: *'a* ⇒ *'b pmf*
assumes *finite A* $B \subseteq A$ $\bigwedge x.\ x \in A - B \Longrightarrow f\ x$ = *return-pmf dflt*
shows *Pi-pmf A dflt f* = *Pi-pmf B dflt f*
proof −
have *Pi-pmf* ($B \cup (A - B)$) *dflt f* =
map-pmf ($\lambda(f,\ g)\ x$. *if* $x \in B$ *then f x else g x*)
(*pair-pmf* (*Pi-pmf B dflt f*) (*Pi-pmf* ($A - B$) *dflt f*))
using *assms* **by** (*intro Pi-pmf-union*) (*auto dest*: *finite-subset*)
also have *Pi-pmf* ($A - B$) *dflt f* = *Pi-pmf* ($A - B$) *dflt* (λ-. *return-pmf dflt*)
using *assms* **by** (*intro Pi-pmf-cong*) *auto*
also have ... = *return-pmf* (λ-. *dflt*)
using *assms* **by** *simp*
also have *map-pmf* ($\lambda(f,\ g)\ x$. *if* $x \in B$ *then f x else g x*)
(*pair-pmf* (*Pi-pmf B dflt f*) (*return-pmf* (λ-. *dflt*))) =
map-pmf ($\lambda f\ x$. *if* $x \in B$ *then f x else dflt*) (*Pi-pmf B dflt f*)
by (*simp add*: *map-pmf-def pair-pmf-def bind-assoc-pmf bind-return-pmf bind-return-pmf'*)
also have ... = *Pi-pmf B dflt f*
using *assms* **by** (*intro Pi-pmf-default-swap*) (*auto dest*: *finite-subset*)
also have $B \cup (A - B) = A$
using *assms* **by** *auto*
finally show *?thesis* **.**
qed

lemma *Pi-pmf-if-set*:
  assumes *finite A*
  shows *Pi-pmf A dflt* (λx. *if b x then f x else return-pmf dflt*) =
    Pi-pmf $\{x \in A.\ b\ x\}$ *dflt f*
proof −
  have *Pi-pmf A dflt* (λx. *if b x then f x else return-pmf dflt*) =
    Pi-pmf $\{x \in A.\ b\ x\}$ *dflt* (λx. *if b x then f x else return-pmf dflt*)
    using *assms* **by** (*intro Pi-pmf-subset′*) *auto*
  also have ... = *Pi-pmf* $\{x \in A.\ b\ x\}$ *dflt f*
    by (*intro Pi-pmf-cong*) *auto*
  finally show *?thesis* **.**
qed

lemma *Pi-pmf-if-set′*:
  assumes *finite A*
  shows *Pi-pmf A dflt* (λx. *if b x then return-pmf dflt else f x*) =
    Pi-pmf $\{x \in A.\ \neg b\ x\}$ *dflt f*
proof −
  have *Pi-pmf A dflt* (λx. *if b x then return-pmf dflt else f x*) =
    Pi-pmf $\{x \in A.\ \neg b\ x\}$ *dflt* (λx. *if b x then return-pmf dflt else f x*)
    using *assms* **by** (*intro Pi-pmf-subset′*) *auto*
  also have ... = *Pi-pmf* $\{x \in A.\ \neg b\ x\}$ *dflt f*
    by (*intro Pi-pmf-cong*) *auto*
  finally show *?thesis* **.**
qed

Lastly, we can delete a single component from a product:

lemma *Pi-pmf-remove*:
  assumes *finite A*
  shows *Pi-pmf* $(A - \{x\})$ *dflt p* = *map-pmf* (λf. $f(x :=$ *dflt*)) (*Pi-pmf A dflt p*)
proof −
  have *Pi-pmf* $(A - \{x\})$ *dflt p* =
    map-pmf ($\lambda f\ xa$. *if* $xa \in A - \{x\}$ *then f xa else dflt*) (*Pi-pmf A dflt p*)
    using *assms* **by** (*intro Pi-pmf-subset*) *auto*
  also have ... = *map-pmf* (λf. $f(x :=$ *dflt*)) (*Pi-pmf A dflt p*)
    using *set-Pi-pmf-subset*[*of A dflt p*] *assms*
    by (*intro map-pmf-cong refl*) (*auto simp*: *fun-eq-iff*)
  finally show *?thesis* **.**
qed

1.5 Applications

Choosing a subset of a set uniformly at random is equivalent to tossing a fair coin independently for each element and collecting all the elements that came up heads.

lemma *pmf-of-set-Pow-conv-bernoulli*:

```
  assumes finite (A :: 'a set)
  shows map-pmf (λb. {x∈A. b x}) (Pi-pmf A P (λ-. bernoulli-pmf (1/2))) =
pmf-of-set (Pow A)
proof −
  have Pi-pmf A P (λ-. bernoulli-pmf (1/2)) = pmf-of-set (PiE-dflt A P (λx.
UNIV))
    using assms by (simp add: bernoulli-pmf-half-conv-pmf-of-set Pi-pmf-of-set)
  also have map-pmf (λb. {x∈A. b x}) ... = pmf-of-set (Pow A)
  proof −
    have bij-betw (λb. {x ∈ A. b x}) (PiE-dflt A P (λ-. UNIV)) (Pow A)
      by (rule bij-betwI[of - - - λB b. if b ∈ A then b ∈ B else P]) (auto simp add:
PiE-dflt-def)
    then show ?thesis
      using assms by (intro map-pmf-of-set-bij-betw) auto
  qed
  finally show ?thesis
    by simp
qed
```

A binomial distribution can be seen as the number of successes in n independent coin tosses.

```
lemma binomial-pmf-altdef':
  fixes A :: 'a set
  assumes finite A and card A = n and p: p ∈ {0..1}
  shows   binomial-pmf n p =
            map-pmf (λf. card {x∈A. f x}) (Pi-pmf A dflt (λ-. bernoulli-pmf p)) (is
?lhs = ?rhs)
proof −
  from assms have ?lhs = binomial-pmf (card A) p
    by simp
  also have ... = ?rhs
  using assms(1)
  proof (induction rule: finite-induct)
    case empty
    with p show ?case by (simp add: binomial-pmf-0)
  next
    case (insert x A)
    from insert.hyps have card (insert x A) = Suc (card A)
      by simp
    also have binomial-pmf ... p = do {
                                  b ← bernoulli-pmf p;
                                  f ← Pi-pmf A dflt (λ-. bernoulli-pmf p);
                                 return-pmf ((if b then 1 else 0) + card {y ∈ A. f y})
                                }
      using p by (simp add: binomial-pmf-Suc insert.IH bind-map-pmf)
    also have ... = do {
                     b ← bernoulli-pmf p;
                     f ← Pi-pmf A dflt (λ-. bernoulli-pmf p);
                     return-pmf (card {y ∈ insert x A. (f(x := b)) y})
```

```
                }
    proof (intro bind-pmf-cong refl, goal-cases)
      case (1 b f)
      have (if b then 1 else 0) + card {y∈A. f y} = card ((if b then {x} else {}) ∪
{y∈A. f y})
        using insert.hyps by auto
      also have (if b then {x} else {}) ∪ {y∈A. f y} = {y∈insert x A. (f(x := b))
y}
        using insert.hyps by auto
      finally show ?case by simp
    qed
    also have ... = map-pmf (λf. card {y∈insert x A. f y})
                    (Pi-pmf (insert x A) dflt (λ-. bernoulli-pmf p))
    using insert.hyps by (subst Pi-pmf-insert) (simp-all add: pair-pmf-def map-bind-pmf)
    finally show ?case .
  qed
  finally show ?thesis .
qed

end
```

2 Auxiliary material

```
theory Misc
  imports HOL−Analysis.Analysis
begin
```

Based on *sorted-list-of-set* and *the-inv-into* we construct a bijection between a finite set A of type 'a::linorder and a set of natural numbers {..<*card A*}

```
lemma bij-betw-mono-on-the-inv-into:
  fixes A::'a::linorder set and B::'b::linorder set
  assumes b: bij-betw f A B and m: mono-on A f
  shows mono-on B (the-inv-into A f)
proof (rule ccontr)
  assume ¬ mono-on B (the-inv-into A f)
  then have ∃r s. r ∈ B ∧ s ∈ B ∧ r ≤ s ∧ ¬ the-inv-into A f s ≥ the-inv-into
A f r
    unfolding mono-on-def by blast
  then obtain r s where rs: r ∈ B s ∈ B r ≤ s the-inv-into A f s < the-inv-into
A f r
    by fastforce
  have f: f (the-inv-into A f b) = b if b ∈ B for b
    using that assms f-the-inv-into-f-bij-betw by metis
  have the-inv-into A f s ∈ A the-inv-into A f r ∈ A
    using rs assms by (auto simp add: bij-betw-def the-inv-into-into)
  then have f (the-inv-into A f s) ≤ f (the-inv-into A f r)
   using rs by (intro mono-onD[OF m]) (auto)
  then have r = s
    using rs f by simp
```

```
  then show False
    using rs by auto
qed

lemma rev-removeAll-removeAll-rev: rev (removeAll x xs) = removeAll x (rev xs)
  by (simp add: removeAll-filter-not-eq rev-filter)

lemma sorted-list-of-set-Min-Cons:
  assumes finite A A ≠ {}
  shows sorted-list-of-set A = Min A # sorted-list-of-set (A − {Min A})
proof −
  have ∗: A = insert (Min A) A
    using assms Min-in by (auto)
  then have sorted-list-of-set A = insort (Min A) (sorted-list-of-set (A − {Min
A}))
    using assms by (subst ∗, intro sorted-list-of-set-insert-remove) auto
  also have ... = Min A # sorted-list-of-set (A − {Min A})
    using assms by (intro insort-is-Cons) (auto)
  finally show ?thesis
    by simp
qed

lemma sorted-list-of-set-filter:
  assumes finite A
  shows sorted-list-of-set ({x∈A. P x}) = filter P (sorted-list-of-set A)
  using assms proof (induction sorted-list-of-set A arbitrary: A)
  case (Cons x xs)
  have x: x ∈ A
    using Cons sorted-list-of-set list.set-intros(1) by metis
  have sorted-list-of-set A = Min A # sorted-list-of-set (A − {Min A})
    using Cons by (intro sorted-list-of-set-Min-Cons) auto
  then have 1: x = Min A xs = sorted-list-of-set (A − {x})
    using Cons by auto
  { assume Px: P x
    have 2: sorted-list-of-set {x ∈ A. P x} = Min {x ∈ A. P x} # sorted-list-of-set
({x ∈ A. P x} − {Min {x ∈ A. P x}})
      using Px Cons 1 sorted-list-of-set-eq-Nil-iff
      by (intro sorted-list-of-set-Min-Cons) fastforce+
    also have 3: Min {x ∈ A. P x} = x
      using Cons 1 Px x by (auto intro!: Min-eqI)
    also have 4: {x ∈ A. P x} − {x} = {y ∈ A − {x}. P y}
      by blast
    also have 5: sorted-list-of-set {y ∈ A − {x}. P y} = filter P (sorted-list-of-set
(A − {x}))
      using 1 Cons by (intro Cons) (auto)
    also have ... = filter P xs
      using 1 by simp
    also have filter P (sorted-list-of-set A) = x # filter P xs
      using Px by (simp flip: ‹x # xs = sorted-list-of-set A›)
```

```
    finally have ?case
      by auto }
  moreover
  { assume Px: ¬ P x
    then have {x ∈ A. P x} = {y ∈ A − {x}. P y}
      by blast
    also have sorted-list-of-set ... = filter P (sorted-list-of-set (A − {x}))
      using 1 Cons by (intro Cons) auto
    also have filter P (sorted-list-of-set (A − {x})) = filter P (sorted-list-of-set
A)
      using 1 Px by (simp flip: ‹x # xs = sorted-list-of-set A›)
    finally have ?case
      by simp }
  ultimately show ?case
    by blast
qed (use sorted-list-of-set-eq-Nil-iff in fastforce)

lemma sorted-list-of-set-Max-snoc:
  assumes finite A A ≠ {}
  shows sorted-list-of-set A = sorted-list-of-set (A − {Max A}) @ [Max A]
proof −
  have ∗: A = insert (Max A) A
    using assms Max-in by (auto)
  then have sorted-list-of-set A = insort (Max A) (sorted-list-of-set (A − {Max
A}))
    using assms by (subst ∗, intro sorted-list-of-set-insert-remove) auto
  also have ... = sorted-list-of-set (A − {Max A}) @ [Max A]
    using assms by (intro sorted-insort-is-snoc) (auto)
  finally show ?thesis
    by simp
qed

lemma sorted-list-of-set-image:
  assumes mono-on A g inj-on g A
  shows (sorted-list-of-set (g ‘ A)) = map g (sorted-list-of-set A)
proof (cases finite A)
  case True
  then show ?thesis
    using assms proof (induction sorted-list-of-set A arbitrary: A)
    case Nil
    then show ?case
      using sorted-list-of-set-eq-Nil-iff by fastforce
  next
    case (Cons x xs A)
    have not-empty-A: A ≠ {}
      using Cons sorted-list-of-set-eq-Nil-iff by auto
    have ∗: Min (g ‘ A) = g (Min A)
    proof −
      have g (Min A) ≤ g a if a ∈ A for a
```

```
      using that Cons Min-in Min-le not-empty-A by (auto intro!: mono-onD[of
- g])
    then show ?thesis
      using Cons not-empty-A by (intro Min-eqI) auto
  qed
  have g ‘ A ≠ {} finite (g ‘ A)
    using Cons by auto
  then have (sorted-list-of-set (g ‘ A)) =
          Min (g ‘ A) # sorted-list-of-set ((g ‘ A) − {Min (g ‘ A)})
    by (auto simp add: sorted-list-of-set-Min-Cons)
  also have (g ‘ A) − {Min (g ‘ A)} = g ‘ (A − {Min A})
      using Cons Min-in not-empty-A ∗ by (subst inj-on-image-set-diff[of - A])
auto
  also have sorted-list-of-set (g ‘ (A − {Min A})) = map g (sorted-list-of-set (A
− {Min A}))
    using not-empty-A Cons mono-on-subset[of A - A − {Min A}] inj-on-subset[of
- A A − {Min A}]
    by (intro Cons) (auto simp add: sorted-list-of-set-Min-Cons)
  finally show ?case
    using Cons not-empty-A ∗ by (auto simp add: sorted-list-of-set-Min-Cons)
 qed
next
 case False
 then show ?thesis
   using assms by (simp add: finite-image-iff)
qed

lemma sorted-list-of-set-length: length (sorted-list-of-set A) = card A
 using distinct-card sorted-list-of-set[of A] by (cases finite A) fastforce+

lemma sorted-list-of-set-bij-betw:
 assumes finite A
 shows bij-betw (λn. sorted-list-of-set A ! n) {..<card A} A
 by (rule bij-betw-nth) (fastforce simp add: assms sorted-list-of-set-length)+

lemma nth-mono-on:
 assumes sorted xs distinct xs set xs = A
 shows mono-on {..<card A} (λn. xs ! n)
 using assms by (intro mono-onI sorted-nth-mono) (auto simp add: distinct-card)

lemma sorted-list-of-set-mono-on:
 finite A ⟹ mono-on {..<card A} (λn. sorted-list-of-set A ! n)
 by (rule nth-mono-on) (auto)

definition bij-mono-map-set-to-nat :: ′a::linorder set ⇒ ′a ⇒ nat where
 bij-mono-map-set-to-nat A =
   (λx. if x ∈ A then the-inv-into {..<card A} ((!) (sorted-list-of-set A)) x
            else card A)
```

lemma *bij-mono-map-set-to-nat*:
  assumes *finite A*
  shows *bij-betw* (*bij-mono-map-set-to-nat A*) *A* {..<*card A*}
      mono-on A (*bij-mono-map-set-to-nat A*)
      (*bij-mono-map-set-to-nat A*) ‘ *A* = {..<*card A*}
proof −
  let *?f* = *bij-mono-map-set-to-nat A*
  have *bij-betw* (*the-inv-into* {..<*card A*} ((!) (*sorted-list-of-set A*))) *A* {..<*card A*}
    using *assms sorted-list-of-set-bij-betw bij-betw-the-inv-into* **by** *blast*
  moreover have *bij-betw* (*the-inv-into* {..<*card A*} ((!) (*sorted-list-of-set A*))) *A* {..<*card A*}
                = *bij-betw ?f A* {..<*card A*}
    unfolding *bij-mono-map-set-to-nat-def* **by** (*rule bij-betw-cong*) *simp*
  ultimately show ∗: *bij-betw* (*bij-mono-map-set-to-nat A*) *A* {..<*card A*}
    by *blast*
  have *mono-on A* (*the-inv-into* {..<*card A*} ((!) (*sorted-list-of-set A*)))
    using *assms sorted-list-of-set-bij-betw*
      sorted-list-of-set-mono-on **by** (*intro bij-betw-mono-on-the-inv-into*) *auto*
  then show *mono-on A* (*bij-mono-map-set-to-nat A*)
    unfolding *bij-mono-map-set-to-nat-def* **using** *mono-onD* **by** (*intro mono-onI*) (*auto*)
  show *?f* ‘ *A* = {..<*card A*}
      using *assms bij-betw-imp-surj-on* ∗ **by** *blast*
qed

end

3 Theorems about the Geometric Distribution

theory *Geometric-PMF*
  imports
    HOL−Probability.Probability
    Pi-pmf
    Monad-Normalisation.Monad-Normalisation
begin

lemma *nn-integral-geometric-pmf*:
  assumes *p* ∈ {*0*<..*1*}
  shows *nn-integral* (*geometric-pmf p*) *real* = (*1* − *p*) / *p*
 using *assms expectation-geometric-pmf integrable-real-geometric-pmf*
  by (*subst nn-integral-eq-integral*) *auto*

lemma *geometric-pmf-prob-atMost*:
  assumes *p* ∈ {*0*<..*1*}
  shows *measure-pmf.prob* (*geometric-pmf p*) {..*n*} = (*1* − (*1* − *p*)^(*n* + *1*))
proof −
  have (∑*x*≤*n*. (*1* − *p*) ^ *x* ∗ *p*) = *1* − (*1* − *p*) ∗ (*1* − *p*) ^ *n*
    by (*induction n*) (*auto simp add*: *algebra-simps*)
  then show *?thesis*

```
  using assms by (auto simp add: measure-pmf-conv-infsetsum)
qed

lemma geometric-pmf-prob-lessThan:
  assumes p ∈ {0<..1}
  shows  measure-pmf.prob (geometric-pmf p) {..<n} = 1 − (1 − p) ^ n
proof −
  have (∑x<n. (1 − p) ^ x ∗ p) = 1 − (1 − p) ^ n
    by (induction n) (auto simp add: algebra-simps)
  then show ?thesis
  using assms by (auto simp add: measure-pmf-conv-infsetsum)
qed

lemma geometric-pmf-prob-greaterThan:
  assumes p ∈ {0<..1}
  shows  measure-pmf.prob (geometric-pmf p) {n<..} = (1 − p)^(n + 1)
proof −
  have (UNIV − {..n}) = {n<..}
    by auto
  moreover have measure-pmf.prob (geometric-pmf p) (UNIV − {..n}) = (1 −
p) ^ (n + 1)
    using assms by (subst measure-pmf.finite-measure-Diff)
                   (auto simp add: geometric-pmf-prob-atMost)
  ultimately show ?thesis
    by auto
qed

lemma geometric-pmf-prob-atLeast:
  assumes p ∈ {0<..1}
  shows  measure-pmf.prob (geometric-pmf p) {n..} = (1 − p)^n
proof −
  have (UNIV − {..<n}) = {n..}
    by auto
  moreover have measure-pmf.prob (geometric-pmf p) (UNIV − {..<n}) = (1 −
p) ^ n
    using assms by (subst measure-pmf.finite-measure-Diff)
                   (auto simp add: geometric-pmf-prob-lessThan)
  ultimately show ?thesis
    by auto
qed

lemma bernoulli-pmf-of-set':
  assumes finite A
  shows map-pmf (λb. {x ∈ A. ¬ b x}) (Pi-pmf A P (λ-. bernoulli-pmf (1/2)))
= pmf-of-set (Pow A)
proof −
  have ∗: Pi-pmf A P (λ-. pmf-of-set (UNIV :: bool set)) = pmf-of-set (PiE-dflt
A P (λ-. UNIV :: bool set))
    using assms by (intro Pi-pmf-of-set) auto
```

```
  have map-pmf (λb. {x ∈ A. ¬ b x}) (Pi-pmf A P (λ-. bernoulli-pmf (1 / 2))) =
        map-pmf (λb. {x ∈ A. ¬ b x}) (Pi-pmf A P (λ-. pmf-of-set UNIV))
    using bernoulli-pmf-half-conv-pmf-of-set by auto
  also have ... = map-pmf (λb. {x ∈ A. ¬ b x}) (pmf-of-set (PiE-dflt A P (λ-.
UNIV)))
    using assms by (subst Pi-pmf-of-set) (auto)
  also have ... = pmf-of-set (Pow A)
  proof −
    have bij-betw (λb. {x ∈ A. ¬ b x}) (PiE-dflt A P (λ-. UNIV)) (Pow A)
      by (rule bij-betwI[of - - - λB b. if b ∈ A then ¬ (b ∈ B) else P]) (auto simp
add: PiE-dflt-def)
    then show ?thesis
      using assms by (intro map-pmf-of-set-bij-betw) auto
  qed
  finally show ?thesis
    by simp
qed

lemma Pi-pmf-pmf-of-set-Suc:
  assumes finite A
  shows Pi-pmf A 0 (λ-. geometric-pmf (1/2)) =
        do {
          B ← pmf-of-set (Pow A);
          Pi-pmf B 0 (λ-. map-pmf Suc (geometric-pmf (1/2))) }
proof −
  have Pi-pmf A 0 (λ-. geometric-pmf (1/2)) =
        Pi-pmf A 0 (λ-. bernoulli-pmf (1/2) ≫=
        (λb. if b then return-pmf 0 else map-pmf Suc (geometric-pmf (1/2))))
    using assms by (subst geometric-bind-pmf-unfold) auto
  also have ... =
            Pi-pmf A False (λ-. bernoulli-pmf (1/2)) ≫=
                (λb. Pi-pmf A 0 (λx. if b x then return-pmf 0 else map-pmf Suc
(geometric-pmf (1/2))))
    using assms by (subst Pi-pmf-bind[of - - - - False]) auto
  also have ... =
            do {b ← Pi-pmf A False (λ-. bernoulli-pmf (1/2));
               Pi-pmf {x ∈ A. ~b x} 0 (λx. map-pmf Suc (geometric-pmf (1/2)))}
    using assms by (subst Pi-pmf-if-set') auto
  also have ... =
          do {B ← map-pmf (λb. {x ∈ A. ¬ b x}) (Pi-pmf A False (λ-. bernoulli-pmf
(1/2)));
                Pi-pmf B 0 (λx. map-pmf Suc (geometric-pmf (1/2)))}
   unfolding map-pmf-def apply(subst bind-assoc-pmf) apply(subst bind-return-pmf)
by auto
   also have ... = pmf-of-set (Pow A) ≫= (λB. Pi-pmf B 0 (λx. map-pmf Suc
(geometric-pmf (1 / 2))))
    unfolding assms using assms by (subst bernoulli-pmf-of-set') auto
  finally show ?thesis
    by simp
```

```
qed

lemma Pi-pmf-pmf-of-set-Suc':
  assumes finite A
  shows Pi-pmf A 0 (λ-. geometric-pmf (1/2)) =
      do {
        B ← pmf-of-set (Pow A);
        Pi-pmf B 0 (λ-. map-pmf Suc (geometric-pmf (1/2))) }
proof −
  have Pi-pmf A 0 (λ-. geometric-pmf (1/2)) =
       Pi-pmf A 0 (λ-. bernoulli-pmf (1/2) ≫=
       (λb. if b then return-pmf 0 else map-pmf Suc (geometric-pmf (1/2))))
    using assms by (subst geometric-bind-pmf-unfold) auto
  also have ... =
            Pi-pmf A False (λ-. bernoulli-pmf (1/2)) ≫=
                (λb. Pi-pmf A 0 (λx. if b x then return-pmf 0 else map-pmf Suc
(geometric-pmf (1/2))))
    using assms by (subst Pi-pmf-bind[of - - - - False]) auto
  also have ... =
            do {b ← Pi-pmf A False (λ-. bernoulli-pmf (1/2));
               Pi-pmf {x ∈ A. ~b x} 0 (λx. map-pmf Suc (geometric-pmf (1/2)))}
    using assms by (subst Pi-pmf-if-set') auto
  also have ... =
          do {B ← map-pmf (λb. {x ∈ A. ¬ b x}) (Pi-pmf A False (λ-. bernoulli-pmf
(1/2)));
                Pi-pmf B 0 (λx. map-pmf Suc (geometric-pmf (1/2)))}
    unfolding map-pmf-def by (auto simp add: bind-assoc-pmf bind-return-pmf)
  also have ... = pmf-of-set (Pow A) ≫= (λB. Pi-pmf B 0 (λx. map-pmf Suc
(geometric-pmf (1 / 2))))
    unfolding assms using assms by (subst bernoulli-pmf-of-set') auto
  finally show ?thesis
    by simp
qed

lemma binomial-pmf-altdef':
  fixes A :: 'a set
  assumes finite A and card A = n and p: p ∈ {0..1}
  shows   binomial-pmf n p =
            map-pmf (λf. card {x∈A. f x}) (Pi-pmf A dflt (λ-. bernoulli-pmf p)) (is
?lhs = ?rhs)
proof −
  from assms have ?lhs = binomial-pmf (card A) p
    by simp
  also have ... = ?rhs
  using assms(1)
  proof (induction rule: finite-induct)
    case empty
    with p show ?case by (simp add: binomial-pmf-0)
  next
```

```
    case (insert x A)
    from insert.hyps have card (insert x A) = Suc (card A)
      by simp
    also have binomial-pmf ... p = do {
                                    b ← bernoulli-pmf p;
                                    f ← Pi-pmf A dflt (λ-. bernoulli-pmf p);
                                   return-pmf ((if b then 1 else 0) + card {y ∈ A. f y})
                                  }
      using p by (simp add: binomial-pmf-Suc insert.IH bind-map-pmf)
    also have ... = do {
                      b ← bernoulli-pmf p;
                      f ← Pi-pmf A dflt (λ-. bernoulli-pmf p);
                      return-pmf (card {y ∈ insert x A. (f(x := b)) y})
                    }
    proof (intro bind-pmf-cong refl, goal-cases)
      case (1 b f)
      have (if b then 1 else 0) + card {y∈A. f y} = card ((if b then {x} else {}) ∪
{y∈A. f y})
        using insert.hyps by auto
      also have (if b then {x} else {}) ∪ {y∈A. f y} = {y∈insert x A. (f(x := b))
y}
        using insert.hyps by auto
      finally show ?case by simp
    qed
    also have ... = map-pmf (λf. card {y∈insert x A. f y})
                    (Pi-pmf (insert x A) dflt (λ-. bernoulli-pmf p))
    using insert.hyps by (subst Pi-pmf-insert) (simp-all add: pair-pmf-def map-bind-pmf)
    finally show ?case .
  qed
  finally show ?thesis .
qed

lemma bernoulli-pmf-Not:
  assumes p ∈ {0..1}
  shows bernoulli-pmf p = map-pmf Not (bernoulli-pmf (1 − p))
proof −
  have ∗: (Not −‘ {True}) = {False} (Not −‘ {False}) = {True}
    by blast+
  have pmf (bernoulli-pmf p) b = pmf (map-pmf Not (bernoulli-pmf (1 − p))) b
for b
    using assms by (cases b) (auto simp add: pmf-map ∗ measure-pmf-single)
  then show ?thesis
    by (rule pmf-eqI)
qed

lemma binomial-pmf-altdef'':
  assumes p: p ∈ {0..1}
  shows   binomial-pmf n p =
          map-pmf (λf. card {x. x < n ∧ f x}) (Pi-pmf {..<n} dflt (λ-. bernoulli-pmf
```

```
p))
  using assms by (subst binomial-pmf-altdef'[of {..<n}]) (auto)

context includes monad-normalisation
begin

lemma Pi-pmf-geometric-filter:
  assumes finite A p ∈ {0<..1}
  shows Pi-pmf A 0 (λ-. geometric-pmf p) =
      do {
        fb ← Pi-pmf A dflt (λ-. bernoulli-pmf p);
        Pi-pmf {x ∈ A. ¬ fb x} 0 (λ-. map-pmf Suc (geometric-pmf p)) }
proof −
  have Pi-pmf A 0 (λ-. geometric-pmf p) =
        Pi-pmf A 0 (λ-. bernoulli-pmf p ≫=
                (λb. if b then return-pmf 0 else map-pmf Suc (geometric-pmf p)))
    using assms by (subst geometric-bind-pmf-unfold) auto
  also have ... =
            Pi-pmf A dflt (λ-. bernoulli-pmf p) ≫=
                (λb. Pi-pmf A 0 (λx. if b x then return-pmf 0 else map-pmf Suc
(geometric-pmf p)))
    using assms by (subst Pi-pmf-bind[of - - - - dflt]) auto
  also have ... =
            do {b ← Pi-pmf A dflt (λ-. bernoulli-pmf p);
                Pi-pmf {x ∈ A. ¬ b x} 0 (λx. map-pmf Suc (geometric-pmf p))}
    using assms by (subst Pi-pmf-if-set') (auto)
  finally show ?thesis
    by simp
qed

lemma Pi-pmf-geometric-filter':
  assumes finite A p ∈ {0<..1}
  shows Pi-pmf A 0 (λ-. geometric-pmf p) =
      do {
        fb ← Pi-pmf A dflt (λ-. bernoulli-pmf (1 − p));
        Pi-pmf {x ∈ A. fb x} 0 (λ-. map-pmf Suc (geometric-pmf p)) }
 using assms by (auto simp add: Pi-pmf-geometric-filter[of - - ¬ dflt] bernoulli-pmf-Not[of
p]
     Pi-pmf-map[of - - dflt] map-pmf-def[of ((∘) Not)])

end

end
```

4 Randomized Skip Lists

```
theory Skip-List
  imports Geometric-PMF
          Misc
```

Monad-Normalisation.Monad-Normalisation
begin

Conflicting notation from *HOL−Analysis.Infinite-Sum*

no-notation *Infinite-Sum.abs-summable-on* (**infixr** ‹*abs′-summable′-on*› *46*)

4.1 Preliminaries

lemma *bind-pmf-if′*: (*do* {*c* ← *C*;
ab ← (*if c then A else B*);
D ab}::′*a pmf*) =
do {*c* ← *C*;
(*if c then* (*A* ≫= *D*) *else* (*B* ≫= *D*))}
by (*metis* (*mono-tags*, *lifting*))

abbreviation (*input*) Max_0 **where** $Max_0 \equiv (\lambda A.\ Max\ (A \cup \{0\}))$

4.2 Definition of a Randomised Skip List

Given a set A we assign a geometric random variable (counting the number of failed Bernoulli trials before the first success) to every element in A. That means an arbitrary element of A is on level n with probability $(1-p)^n p$. We define he height of the skip list as the maximum assigned level. So a skip list with only one level has height 0 but the calculation of the expected height is cleaner this way.

locale *random-skip-list* =
fixes *p*::*real*
begin

definition *q* **where** *q* = *1* − *p*

definition *SL* :: (′*a*::*linorder*) *set* ⇒ (′*a* ⇒ *nat*) *pmf* **where** *SL A* = *Pi-pmf A 0* (λ-. *geometric-pmf p*)
definition SL_N :: *nat* ⇒ (*nat* ⇒ *nat*) *pmf* **where** SL_N *n* = *SL* {..<*n*}

4.3 Height of Skip List

definition *H* **where** *H A* = *map-pmf* (λ*f*. Max_0 (*f* ‘ *A*)) (*SL A*)
definition H_N :: *nat* ⇒ *nat pmf* **where** H_N *n* = *H* {..<*n*}

context includes *monad-normalisation*
begin

The height of a skip list is independent of the values in a set A. For simplicity we can therefore work on the skip list over the set {..<*card A*}

lemma
assumes *finite A*
shows *H A* = H_N (*card A*)

proof −
  define f' **where** $f' = (\lambda x.\ \textit{if}\ x \in A$
    $\textit{then the-inv-into}\ \{..<\textit{card}\ A\}\ ((!)\ (\textit{sorted-list-of-set}\ A))\ x$
    $\textit{else card}\ A)$
  have *bij-f'*: *bij-betw* f' A $\{..<\textit{card}\ A\}$
  proof −

    have *bij-betw* (*the-inv-into* $\{..<\textit{card}\ A\}$ ((!) (*sorted-list-of-set* A))) A $\{..<\textit{card}\ A\}$
      unfolding f'*-def* **using** *sorted-list-of-set-bij-betw assms bij-betw-the-inv-into* **by** *blast*
    moreover have *bij-betw* (*the-inv-into* $\{..<\textit{card}\ A\}$ ((!) (*sorted-list-of-set* A))) A $\{..<\textit{card}\ A\}$
      $=$ *bij-betw* f' A $\{..<\textit{card}\ A\}$
      unfolding f'*-def* **by** (*rule bij-betw-cong*) *simp*
    ultimately show *?thesis*
      by *blast*
  qed
  have $*$: $\textit{Max}_0\ ((f \circ f')\ `\ A) = \textit{Max}_0\ (f\ `\ \{..<\textit{card}\ A\})$ **for** f :: $\textit{nat} \Rightarrow \textit{nat}$
    using *bij-betw-imp-surj-on bij-f' image-comp* **by** *metis*
  have $H\ A = \textit{map-pmf}\ (\lambda f.\ \textit{Max}_0\ (f\ `\ A))\ (\textit{map-pmf}\ (\lambda g.\ g \circ f')\ (\textit{SL}_N\ (\textit{card}\ A)))$
    using *assms bij-f'* **unfolding** *H-def SL-def* $\textit{SL}_N$*-def*
    by (*subst Pi-pmf-bij-betw*[*of - f'* $\{..<\textit{card}\ A\}$]) (*auto simp add*: f'*-def*)
  also have ... $=$ H_N (*card* A)
    unfolding H_N*-def H-def* $\textit{SL}_N$*-def* **using** $*$ **by** (*auto intro*!: *bind-pmf-cong simp add*: *map-pmf-def*)
  finally show *?thesis*
    by *simp*
qed

The cumulative distribution function (CDF) of the height is the CDF of the geometric PMF to the power of n

lemma *prob-Max-IID-geometric-atMost*:
  assumes $p \in \{0..1\}$
  shows *measure-pmf.prob* (H_N n) $\{..i\}$
    $=$ (*measure-pmf.prob* (*geometric-pmf* p) $\{..i\}$) ^ n (**is** *?lhs* $=$ *?rhs*)
proof −
  note *SL-def*[*simp*] $\textit{SL}_N$*-def*[*simp*] *H-def*[*simp*] H_N*-def*[*simp*]
  have $\{f.\ \textit{Max}_0\ (f\ `\ \{..<n\}) \le i\} = \{..<n\} \to \{..i\}$
    by *auto*
  then have *?lhs* $=$ *measure-pmf.prob* ($\textit{SL}_N$ n) ($\{..<n\} \to \{..i\}$)
    by (*simp add*: *vimage-def*)
  also have ... $=$ *measure-pmf.prob* ($\textit{SL}_N$ n) (*PiE-dflt* $\{..<n\}$ *0* (λ-. $\{..i\}$))
    by (*intro measure-prob-cong-0*) (*auto simp add*: *PiE-dflt-def pmf-Pi split*: *if-splits*)
  also have ... $=$ *measure-pmf.prob* (*geometric-pmf* p) $\{..i\}$ ^ n
    using *assms* **by** (*auto simp add*: *measure-Pi-pmf-PiE-dflt*)
  finally show *?thesis*
    by *simp*

qed

lemma *prob-Max-IID-geometric-greaterThan*:
assumes $p \in \{0<..1\}$
shows *measure-pmf.prob* $(H_N\ n)$ $\{i<..\}$ =
$1 - (1 - q\ \hat{}\ (i + 1))\ \hat{}\ n$
proof −
have $UNIV - \{..i\} = \{i<..\}$
by *auto*
then have *measure-pmf.prob* $(H_N\ n)$ $\{i<..\}$ = *measure-pmf.prob* $(H_N\ n)$ (*space* (*measure-pmf* $(H_N\ n)$) − $\{..i\}$)
by (*auto*)
also have ... = $1 - ($*measure-pmf.prob* (*geometric-pmf p*) $\{..i\})\ \hat{}\ n$
using *assms* **by** (*subst measure-pmf.prob-compl*) (*auto simp add*: *prob-Max-IID-geometric-atMost*)
also have ... = $1 - (1 - q\ \hat{}\ (i + 1))\ \hat{}\ n$
using *assms* **unfolding** *q-def* **by** (*subst geometric-pmf-prob-atMost*) *auto*
finally show *?thesis*
by *simp*
qed

end
end

An alternative definition of the expected value of a non-negative random variable [1]

lemma *expectation-prob-atLeast*:
assumes (λi. *measure-pmf.prob N* $\{i..\}$) *abs-summable-on* $\{1..\}$
shows *measure-pmf.expectation N real* = *infsetsum* (λi. *measure-pmf.prob N* $\{i..\}$) $\{1..\}$
integrable N real
proof −
have ($\lambda(x, y)$. *pmf N y*) *abs-summable-on Sigma* {*Suc 0*..} *atLeast*
using *assms* **by** (*auto simp add*: *measure-pmf-conv-infsetsum abs-summable-on-Sigma-iff*)
then have *summable*: ($\lambda(x, y)$. *pmf N x*) *abs-summable-on Sigma* {*Suc 0*..} (*atLeastAtMost* (*Suc 0*))
by (*subst abs-summable-on-reindex-bij-betw*[*of* $\lambda(x,y)$. (y,x), *symmetric*])
(*auto intro*!: *bij-betw-imageI simp add*: *inj-on-def case-prod-beta*)
have *measure-pmf.expectation N real* = $(\sum_a x.$ *pmf N x* $*_R$ *real x*)
by (*auto simp add*: *infsetsum-def integral-density measure-pmf-eq-density*)
also have ... = $(\sum_a x \in (\{0\} \cup \{Suc\ 0..\}).$ *pmf N x* $*_R$ *real x*)
by (*auto intro*!: *infsetsum-cong*)
also have ... = $(\sum_a x \in \{Suc\ 0..\}.$ *pmf N x* $*$ *real x*)
proof −
have (λx. *pmf N x* $*_R$ *real x*) *abs-summable-on* $\{0\} \cup \{Suc\ 0..\}$
using *summable* **by** (*subst* (*asm*) *abs-summable-on-Sigma-iff*) (*auto simp add*: *mult.commute*)
then show *?thesis*

[1] https://en.wikipedia.org/w/index.php?title=Expected_value&oldid=881384346#Formula_for_non-negative_random_variables

by (*subst infsetsum-Un-Int*) *auto*
qed
also have ... = ($\sum_a$(*x*, *y*)∈*Sigma* {*Suc 0*..} (*atLeastAtMost* (*Suc 0*)). *pmf N x*)
using *summable* **by** (*subst infsetsum-Sigma*) (*auto simp add*: *mult.commute*)
also have ... = ($\sum_a$*x*∈*Sigma* {*Suc 0*..} *atLeast*. *pmf N* (*snd x*))
by (*subst infsetsum-reindex-bij-betw*[*of* λ(*x*,*y*). (*y*,*x*), *symmetric*])
(*auto intro*!: *bij-betw-imageI simp add*: *inj-on-def case-prod-beta*)
also have ... = *infsetsum* (λ*i*. *measure-pmf.prob N* {*i*..}) {*1*..}
using *assms*
by (*subst infsetsum-Sigma*)
(*auto simp add*: *measure-pmf-conv-infsetsum abs-summable-on-Sigma-iff infsetsum-Sigma'*)
finally show *measure-pmf.expectation N real* = *infsetsum* (λ*i*. *measure-pmf.prob N* {*i*..}) {*1*..}
by *simp*
have (λ*x*. *pmf N x* $*_R$ *real x*) *abs-summable-on* {*0*} ∪ {*Suc 0*..}
using *summable* **by** (*subst* (*asm*) *abs-summable-on-Sigma-iff*) (*auto simp add*: *mult.commute*)
then have (λ*x*. *pmf N x* $*_R$ *real x*) *abs-summable-on UNIV*
by (*simp add*: *atLeast-Suc*)
then have *integrable* (*count-space UNIV*) (λ*x*. *pmf N x* $*_R$ *real x*)
by (*subst abs-summable-on-def*[*symmetric*]) *blast*
then show *integrable N real*
by (*subst measure-pmf-eq-density*, *subst integrable-density*) *auto*
qed

The expected height of a skip list has no closed-form expression but we can approximate it. We start by showing how we can calculate an infinite sum over the natural numbers with an integral over the positive reals and the floor function.

lemma *infsetsum-set-nn-integral-reals*:
assumes *f abs-summable-on UNIV* ⋀*n*. *f n* ≥ *0*
shows *infsetsum f UNIV* = *set-nn-integral lborel* {*0*::*real*..} (λ*x*. *f* (*nat* (*floor x*)))
proof −
have *x* < *1* + (*floor x*)**for** *x*::*real*
by *linarith*
then have ∃ *n*. *real n* ≤ *x* ∧ *x* < *1* + *real n* **if** *x* ≥ *0* **for** *x*
using *that of-nat-floor* **by** (*intro exI*[*of - nat* (*floor x*)]) *auto*
then have {*0*..} = (⋃ *n*. {*real n*..<*real* (*Suc n*)})
by *auto*
then have ($\int^+$*x*∈{*0*::*real*..}. *ennreal* (*f* (*nat* ⌊*x*⌋))∂*lborel*) =
($\sum$ *n*. $\int^+$*x*∈{*real n*..<*1* + *real n*}. *ennreal* (*f* (*nat* ⌊*x*⌋))∂*lborel*)
by (*auto simp add*: *disjoint-family-on-def nn-integral-disjoint-family*)
also have ... = ($\sum$ *n*. $\int^+$*x*∈{*real n*..<*1* + *real n*}. *ennreal* (*f n*)∂*lborel*)
by(*subst suminf-cong*, *rule nn-integral-cong-AE*)
(*auto intro*!: *eventuallyI simp add*: *indicator-def floor-eq4*)
also have ... = ($\sum$ *n*. *ennreal* (*f n*))

by (*auto intro*!: *suminf-cong simp add*: *nn-integral-cmult*)
also have ... = *infsetsum f* {*0*..}
using *assms suminf-ennreal2 abs-summable-on-nat-iff′ summable-norm-cancel*
by (*auto simp add*: *infsetsum-nat*)
finally show *?thesis*
by *simp*
qed

lemma *nn-integral-nats-reals*:
shows ($\int^+$ *i*. *ennreal* (*f i*) ∂*count-space UNIV*) = ($\int^+ x \in$ {*0*::*real*..}. *ennreal* (*f* (*nat* ⌊*x*⌋))∂*lborel*)
proof −
have *x* < *1* + (*floor x*)**for** *x*::*real*
by *linarith*
then have ∃ *n*. *real n* ≤ *x* ∧ *x* < *1* + *real n* **if** *x* ≥ *0* **for** *x*
using *that of-nat-floor* **by** (*intro exI*[*of - nat* (*floor x*)]) *auto*
then have {*0*..} = (⋃ *n*. {*real n*..<*real* (*Suc n*)})
by *auto*
then have ($\int^+ x \in$ {*0*::*real*..}. *f* (*nat* ⌊*x*⌋)∂*lborel*) =
(∑ *n*. $\int^+ x \in$ {*real n*..<*1* + *real n*}. *ennreal* (*f* (*nat* ⌊*x*⌋))∂*lborel*)
by (*auto simp add*: *disjoint-family-on-def nn-integral-disjoint-family*)
also have ... = (∑ *n*. $\int^+ x \in$ {*real n*..<*1* + *real n*}. *ennreal* (*f n*)∂*lborel*)
by(*subst suminf-cong*,*rule nn-integral-cong-AE*)
(*auto intro*!: *eventuallyI simp add*: *indicator-def floor-eq4*)
also have ... = (∑ *n*. *ennreal* (*f n*))
by (*auto intro*!: *suminf-cong simp add*: *nn-integral-cmult*)
also have ... = ($\int^+$ *i*. *ennreal* (*f i*) ∂*count-space UNIV*)
by (*simp add*: *nn-integral-count-space-nat*)
finally show *?thesis*
by *simp*
qed

lemma *nn-integral-floor-less-eq*:
assumes ⋀*x y*. *x* ≤ *y* ⟹ *f y* ≤ *f x*
shows ($\int^+ x \in$ {*0*::*real*..}. *ennreal* (*f x*)∂*lborel*) ≤ ($\int^+ x \in$ {*0*::*real*..}. *ennreal* (*f* (*nat* ⌊*x*⌋))∂*lborel*)
using *assms* **by** (*auto simp add*: *indicator-def intro*!: *nn-integral-mono ennreal-leI*)

lemma *nn-integral-finite-imp-abs-sumable-on*:
fixes *f* :: *′a* ⇒ *′b*::{*banach*, *second-countable-topology*}
assumes *nn-integral* (*count-space A*) (λ*x*. *norm* (*f x*)) < ∞
shows *f abs-summable-on A*
using *assms* **unfolding** *abs-summable-on-def integrable-iff-bounded* **by** *auto*

lemma *nn-integral-finite-imp-abs-sumable-on′*:
assumes *nn-integral* (*count-space A*) (λ*x*. *ennreal* (*f x*)) < ∞ ⋀*x*. *f x* ≥ *0*
shows *f abs-summable-on A*
using *assms* **unfolding** *abs-summable-on-def integrable-iff-bounded* **by** *auto*

We now show that $\int_0^\infty 1-(1-q^x)^n\,dx = \frac{-H_n}{\ln q}$ if $0 < q < 1$.

```
lemma harm-integral-x-raised-n:
  set-integrable lborel {0::real..1} (λx. (∑ i∈{..<n}. x ^ i)) (is ?thesis1)
  LBINT x = 0..1. (∑ i∈{..<n}. x ^ i) = harm n (is ?thesis2)
proof −
  have h: set-integrable lborel {0::real..1} (λx. (∑ i∈{..<n}. x ^ i)) for n
    by (intro borel-integrable-atLeastAtMost′) (auto intro!: continuous-intros)
  then show ?thesis1
    by (intro borel-integrable-atLeastAtMost′) (auto intro!: continuous-intros)
  show ?thesis2
  proof (induction n)
    case (Suc n)
    have (LBINT x=0..1.(∑ i∈{..<n}. x ^ i) + x ^ n) =
          (LBINT x=0..1. (∑ i∈{..<n}. x ^ i)) + (LBINT x=0..1. x ^ n)
    proof −
      have set-integrable lborel (einterval 0 1) (λx. (∑ i∈{..<n}. x ^ i))
        by (rule set-integrable-subset) (use h in ‹auto simp add: einterval-def›)
      moreover have set-integrable lborel (einterval 0 1) (λx. (x ^ n))
      proof −
        have set-integrable lborel {0::real..1} (λx. (x ^ n))
          by (rule borel-integrable-atLeastAtMost′)
            (auto intro!: borel-integrable-atLeastAtMost′ continuous-intros)
        then show ?thesis
          by (rule set-integrable-subset) (auto simp add: einterval-def)
      qed
      ultimately show ?thesis
      by (auto intro!: borel-integrable-atLeastAtMost′ simp add: interval-lebesgue-integrable-def)
    qed
    also have (LBINT x=0..1. x ^ n) = 1 / (1 + real n)
    proof −
      have (LBINT x=0..1. x ^ n) = (LBINT x. x ^ n ∗ indicator {0..1} x)
      proof −
        have AE x in lborel. x ^ n ∗ indicator {0..1} x = indicator (einterval 0 1)
x ∗ x ^ n
          by(rule eventually-mono[OF eventually-conj[OF  AE-lborel-singleton[of 1]
                AE-lborel-singleton[of 0]]])
            (auto simp add: indicator-def einterval-def)
        then show ?thesis
       using integral-cong-AE unfolding interval-lebesgue-integral-def set-lebesgue-integral-def
          by (auto intro!: integral-cong-AE)
      qed
      then show ?thesis
        by (auto simp add: integral-power)
    qed
    finally show ?case
      using Suc by (auto simp add: harm-def inverse-eq-divide)
  qed (auto simp add: harm-def)
qed

lemma harm-integral-0-1-fraction:
```

set-integrable lborel {*0*::*real*..*1*} (λ*x*. (*1* − *x* ^ *n*) / (*1* − *x*))
(*LBINT x* = *0*..*1*. ((*1* − *x* ^ *n*) / (*1* − *x*))) = *harm n*
proof −
show *set-integrable lborel* {*0*::*real*..*1*} (λ*x*. (*1* − *x* ^ *n*) / (*1* − *x*))
proof −
have *AE x*∈{*0*::*real*..*1*} *in lborel*. (*1* − *x* ^ *n*) / (*1* − *x*) = *sum* ((^) *x*) {..<*n*}
by (*smt* (*verit*, *best*) *AE-lborel-singleton eventually-mono sum-gp-strict*)
with *harm-integral-x-raised-n* **show** *?thesis*
by (*subst set-integrable-cong-AE*) *auto*
qed
moreover have *AE x*∈{*0*::*real*<..<*1*} *in lborel*. (*1* − *x* ^ *n*) / (*1* − *x*) = *sum* ((^) *x*) {..<*n*}
by (*auto simp add*: *sum-gp-strict*)
moreover have *einterval* (*min 0 1*) (*max 0 1*) = {*0*::*real*<..<*1*}
by (*auto simp add*: *min-def max-def einterval-iff*)
ultimately show (*LBINT x* = *0*..*1*. ((*1* − *x* ^ *n*) / (*1* − *x*))) = *harm n*
using *harm-integral-x-raised-n* **by** (*subst interval-integral-cong-AE*) *auto*
qed

lemma *one-minus-one-minus-q-x-n-integral*:
assumes *q* ∈ {*0*<..<*1*}
shows *set-integrable lborel* (*einterval 0* ∞) (λ*x*. (*1* − (*1* − *q powr x*) ^ *n*))
(*LBINT x*=*0*..∞. *1* − (*1* − *q powr x*) ^ *n*) = − *harm n* / *ln q*
proof −
have [*simp*]: *q powr* (*log q* (*1*−*x*)) = *1* − *x* **if** *x* ∈ {*0*<..<*1*} **for** *x*
using *that assms* **by** (*subst powr-log-cancel*) *auto*
have *1*: ((*ereal* ∘ (λ*x*. *log q* (*1* − *x*)) ∘ *real-of-ereal*) ⟶ *0*) (*at-right 0*)
using *assms* **unfolding** *zero-ereal-def ereal-tendsto-simps* **by** (*auto intro*!: *tendsto-eq-intros*)
have *2*: ((*ereal* ∘ (λ*x*. *log q* (*1*−*x*)) ∘ *real-of-ereal*) ⟶ ∞) (*at-left 1*)
proof −
have *filterlim* ((−) *1*) (*at-right 0*) (*at-left* (*1*::*real*))
by (*intro filterlim-at-withinI eventually-at-leftI*[*of 0*]) (*auto intro*!: *tendsto-eq-intros*)
then have *LIM x at-left 1*. − *inverse* (*ln q*) ∗ − *ln* (*1* − *x*) :> *at-top*
using *assms*
by (*intro filterlim-tendsto-pos-mult-at-top* [*OF tendsto-const*])
(*auto simp*: *filterlim-uminus-at-top intro*!: *filterlim-compose*[*OF ln-at-0*])
then show *?thesis*
unfolding *one-ereal-def ereal-tendsto-simps log-def* **by** (*simp add*: *field-simps*)
qed
have *3*: *set-integrable lborel* (*einterval 0 1*)
(λ*x*. (*1* − (*1* − *q powr* (*log q* (*1* − *x*))) ^ *n*) ∗ (− *1* / (*ln q* ∗ (*1* − *x*))))
proof −
have *set-integrable lborel* (*einterval 0 1*) (λ*x*. − (*1* / *ln q*) ∗ ((*1* − *x* ^ *n*) / (*1* − *x*)))
by(*intro set-integrable-mult-right*)
(*auto intro*!: *harm-integral-0-1-fraction intro*: *set-integrable-subset simp add*: *einterval-def*)

```
    then show ?thesis
      by(subst set-integrable-cong-AE[where g=λx. − (1 / ln q) ∗ ((1 − x ^ n) /
(1 − x))])
        (auto intro!: eventuallyI simp add: einterval-def)
  qed
  have 4: LBINT x=0..1. − ((1 − (1 − q powr log q (1 − x)) ^ n) / (ln q ∗ (1
− x))) = − (harm n / ln q)
    (is ?lhs = ?rhs)
  proof −
    have ?lhs = LBINT x=0..1. ((1 − x ^ n) / (1 − x)) ∗ (− 1 / ln q)
      using assms
      by (intro interval-integral-cong-AE)
      (auto intro!: eventuallyI simp add: max-def einterval-def field-simps)
    also have . . . = harm n ∗ (−1 / ln q)
     using harm-integral-0-1-fraction by (subst interval-lebesgue-integral-mult-left)
auto
    finally show ?thesis
      by auto
  qed
  note sub = interval-integral-substitution-nonneg
            [where f = (λx. (1 − (1 − q powr x) ^ n)) and g=(λx. log q (1−x))
                  and g′=(λx. − 1 / (ln q ∗ (1 − x))) and a = 0 and b = 1]
  show set-integrable lborel (einterval 0 ∞) (λx. (1 − (1 − q powr x) ^ n))
    using assms 1 2 3 4
   by (intro sub) (auto intro!: sub derivative-eq-intros continuous-intros mult-nonneg-nonpos2
power-le-one)
  show (LBINT x=0..∞. 1 − (1 − q powr x) ^ n) = − harm n / ln q
    using assms 1 2 3 4
   by (subst sub) (auto intro!: derivative-eq-intros continuous-intros mult-nonneg-nonpos2
power-le-one)
qed

lemma one-minus-one-minus-q-x-n-nn-integral:
  fixes q::real
  assumes q ∈ {0<..<1}
  shows set-nn-integral lborel {0..} (λx. (1 − (1 − q powr x) ^ n)) =
        LBINT x=0..∞. 1 − (1 − q powr x) ^ n
proof −
  have set-nn-integral  lborel {0..} (λx. (1 − (1 − q powr x) ^ n)) =
        nn-integral lborel (λx. indicator (einterval 0 ∞) x ∗  (1 − (1 − q powr x)
^ n))
   using assms by (intro nn-integral-cong-AE eventually-mono[OF AE-lborel-singleton[of
0]])
      (auto simp add: indicator-def einterval-def)
  also have . . . = ennreal (LBINT x. indicator (einterval 0 ∞) x ∗ (1 − (1 − q
powr x) ^ n))
  using one-minus-one-minus-q-x-n-integral assms
  by(intro nn-integral-eq-integral)
    (auto simp add: indicator-def einterval-def set-integrable-def
```

intro!: *eventuallyI power-le-one powr-le1*)
finally show *?thesis*
by (*simp add*: *interval-lebesgue-integral-def set-lebesgue-integral-def*)
qed

We can now derive bounds for the expected height.

context *random-skip-list*
begin

definition EH_N **where** $EH_N\ n = measure\text{-}pmf.expectation\ (H_N\ n)\ real$

lemma EH_N*-bounds'*:
fixes *n*::*nat*
assumes $p \in \{0<..<1\}\ 0 < n$
shows $-\ harm\ n\ /\ ln\ q - 1 \leq EH_N\ n$
$EH_N\ n \leq -\ harm\ n\ /\ ln\ q$
$integrable\ (H_N\ n)\ real$
proof −
define *f* **where** $f = (\lambda x.\ 1 - (1 - q\ \hat{}\ x)\ \hat{}\ n)$
define *f'* **where** $f' = (\lambda x.\ 1 - (1 - q\ powr\ x)\ \hat{}\ n)$
have *q*: $q \in \{0<..<1\}$
unfolding *q-def* **using** *assms* **by** *auto*
have *f-descending*: $f\ y \leq f\ x$ **if** $x \leq y$ **for** *x y*
unfolding *f-def* **using** *that q*
by (*auto intro*!: *power-mono simp add*: *power-decreasing power-le-one-iff*)
have *f'-descending*: $f'\ y \leq f'\ x$ **if** $x \leq y\ 0 \leq x$ **for** *x y*
unfolding *f'-def* **using** *that q*
by (*auto intro*!: *power-mono simp add*: *ln-powr powr-def mult-nonneg-nonpos*)
have [*simp*]: $harm\ n\ /\ ln\ q <= 0$
using *harm-nonneg ln-ge-zero-imp-ge-one q* **by** (*intro divide-nonneg-neg*) *auto*
have *f-nn-integral-harm*:
$-\ harm\ n\ /\ ln\ q \leq \int^+ x.\ (f\ x)\ \partial count\text{-}space\ UNIV$
$(\int^+ i.\ f\ (i + 1)\ \partial count\text{-}space\ UNIV) \leq -\ harm\ n\ /\ ln\ q$
proof −
have $(\int^+ i.\ f\ (i + 1)\ \partial count\text{-}space\ UNIV) = (\int^+ x{\in}\{0::real..\}.\ (f\ (nat\ \lfloor x\rfloor + 1))\partial lborel)$
using *nn-integral-nats-reals* **by** *auto*
also have $\ldots = (\int^+ x{\in}\{0::real..\}.\ ennreal\ (f'\ (nat\ \lfloor x\rfloor + 1))\partial lborel)$
proof −
have $0 \leq x \Longrightarrow (1 - q * q\ \hat{}\ nat\ \lfloor x\rfloor)\ \hat{}\ n = (1 - q\ powr\ (1 + real\text{-}of\text{-}int\ \lfloor x\rfloor))\ \hat{}\ n$ **for** *x*::*real*
using *q* **by** (*subst powr-realpow* [*symmetric*]) (*auto simp*: *powr-add*)
then show *?thesis*
unfolding *f-def f'-def* **using** *q*
by (*auto intro*!: *nn-integral-cong ennreal-cong simp add*: *powr-real-of-int indicator-def*)
qed
also have $\ldots \leq set\text{-}nn\text{-}integral\ lborel\ \{0..\}\ f'$
proof −

```
      have x ≤ 1 + real-of-int ⌊x⌋ for x
        by linarith
      then show ?thesis
          by (auto simp add: indicator-def intro!: f'-descending nn-integral-mono
ennreal-leI)
    qed
    also have harm-integral-f': ... = − harm n / ln q
      unfolding f'-def using q
      by (auto intro!: ennreal-cong
       simp add: one-minus-one-minus-q-x-n-nn-integral one-minus-one-minus-q-x-n-integral)
    finally show (∫⁺ i. f (i + 1) ∂count-space UNIV) ≤ − harm n / ln q
      by simp
    note harm-integral-f'[symmetric]
   also have set-nn-integral lborel {0..} f' ≤ (∫⁺x∈{0::real..}. f' (nat ⌊x⌋)∂lborel)
      using assms f'-descending
      by (auto simp add: indicator-def intro!: nn-integral-mono ennreal-leI)
    also have ... = (∫⁺x∈{0::real..}. f (nat ⌊x⌋)∂lborel)
      unfolding f-def f'-def
     using q by (auto intro!: nn-integral-cong ennreal-cong simp add: powr-real-of-int
indicator-def)
    also have ... = (∫⁺ x. f x ∂count-space UNIV)
      using nn-integral-nats-reals by auto
    finally show − harm n / ln q ≤ ∫⁺ x. f x ∂count-space UNIV
      by simp
  qed
  then have f1-abs-summable-on: (λi. f (i + 1)) abs-summable-on UNIV
    unfolding f-def using q
    by (intro nn-integral-finite-imp-abs-sumable-on')
      (auto simp add: f-def le-less-trans intro!: power-le-one mult-le-one)
  then have f-abs-summable-on: f abs-summable-on {1..}
    using Suc-le-lessD greaterThan-0
     by (subst abs-summable-on-reindex-bij-betw[symmetric, where g=λx. x + 1
and A=UNIV]) auto
  also have (f abs-summable-on {1..}) = ((λx. measure-pmf.prob (H_N n) {x..})
abs-summable-on {1..})
  proof −
    have ((λx. measure-pmf.prob (H_N n) {x..}) abs-summable-on {1..}) =
          ((λx. measure-pmf.prob (H_N n) {x − 1<..}) abs-summable-on {1..})
      by (auto intro!: measure-prob-cong-0 abs-summable-on-cong)
    also have ... = (f abs-summable-on {1..})
      using assms
     by (intro abs-summable-on-cong) (auto simp add: f-def prob-Max-IID-geometric-greaterThan)
    finally show ?thesis
      by simp
  qed
  finally have EH_N-sum:
    EH_N n = (∑_a i∈{1..}. measure-pmf.prob (H_N n) {i..})
    integrable (measure-pmf (H_N n)) real
    unfolding EH_N-def using expectation-prob-atLeast by auto
```

```
then show integrable (measure-pmf (H_N n)) real
  by simp
have EH_N-sum': EH_N n = infsetsum f {1..}
proof -
  have EH_N n = (∑_a k∈{1..}. measure-pmf.prob (H_N n) {k - 1<..})
    unfolding EH_N-sum by (auto intro!: measure-prob-cong-0 infsetsum-cong)
  also have ... = infsetsum f {1..}
    using assms
   by (intro infsetsum-cong) (auto simp add: f-def prob-Max-IID-geometric-greaterThan)
  finally show ?thesis
    by simp
qed
also have ... = (∑_a k. f (k + 1))
  using Suc-le-lessD greaterThan-0
    by (subst infsetsum-reindex-bij-betw[symmetric, where g=λx. x + 1 and
A=UNIV]) auto
also have ennreal ... = (∫⁺x∈{0::real..}. f (nat ⌊x⌋ + 1)∂lborel)
  using f1-abs-summable-on q
    by (intro infsetsum-set-nn-integral-reals) (auto simp add: f-def mult-le-one
power-le-one)
also have ... = (∫⁺ i. f (i + 1) ∂count-space UNIV)
  using nn-integral-nats-reals by auto
also have ... ≤ - harm n / ln q
  using f-nn-integral-harm by auto
finally show EH_N n ≤ - harm n / ln q
  by (subst (asm) ennreal-le-iff) (auto)
have EH_N n + 1 = (∑_a x∈{Suc 0..}. f x) + (∑_a x∈{0}. f x)
  using assms by (subst EH_N-sum') (auto simp add: f-def)
also have ... = infsetsum f UNIV
   using f-abs-summable-on by (subst infsetsum-Un-disjoint[symmetric]) (auto
intro!: infsetsum-cong)
also have ... = (∫⁺x∈{0::real..}. f (nat ⌊x⌋)∂lborel)
proof -
  have f abs-summable-on ({0} ∪ {1..})
    using f-abs-summable-on by (intro abs-summable-on-union) (auto)
  also have {0::nat} ∪ {1..} = UNIV
    by auto
  finally show ?thesis
    using q
     by (intro infsetsum-set-nn-integral-reals) (auto simp add: f-def mult-le-one
power-le-one)
qed
also have ... = (∫⁺ x. f x ∂count-space UNIV)
  using nn-integral-nats-reals by auto
also have ... ≥ - harm n / ln q
  using f-nn-integral-harm by auto
finally have - harm n / ln q ≤ EH_N n + 1
  by (subst (asm) ennreal-le-iff) (auto simp add: EH_N-def)
then show - harm n / ln q - 1 ≤ EH_N n
```

```
    by simp
qed

theorem EH_N-bounds:
  fixes n::nat
  assumes p ∈ {0<..<1}
  shows
    − harm n / ln q − 1 ≤ EH_N n
    EH_N n ≤ − harm n / ln q
    integrable (H_N n) real
proof −
  show − harm n / ln q − 1 ≤ EH_N n
    using assms EH_N-bounds'
    by (cases n = 0) (auto simp add: EH_N-def H_N-def H-def SL-def harm-expand)
  show EH_N n ≤ − harm n / ln q
    using assms EH_N-bounds'
    by (cases n = 0) (auto simp add: EH_N-def H_N-def H-def SL-def harm-expand)
  show integrable (H_N n) real
    using assms EH_N-bounds'
    by (cases n = 0) (auto simp add: H_N-def H-def SL-def intro!: integrable-measure-pmf-finite)
qed

end
```

4.4 Expected Length of Search Path

Let A and f where f is an abstract description of a skip list (assign each value its maximum level). steps A f s u l starts on the rightmost element on level s in the skip lists. If possible it moves up, if not it moves to the left. For every step up it adds cost u and for every step to the left it adds cost l. steps A f 0 1 1 therefore walks from the bottom right corner of a skip list to the top left corner of a skip list and counts all steps.

```
function steps :: 'a :: linorder set ⇒ ('a ⇒ nat) ⇒ nat ⇒ nat ⇒ nat ⇒ nat
where
  steps A f l up left = (if A = {} ∨ infinite A
            then 0
            else (let m = Max A in (if f m < l then      steps (A − {m}) f l up left
                                  else (if f m > l then up + steps A f (l + 1) up left
                                  else                 left + steps (A − {m}) f l up left))))
  by pat-completeness auto
termination
proof (relation (λ(A,f,l,a,b). card A) <*mlex*> (λ(A,f,l,a,b). Max (f ` A) − l)
<*mlex*> {}, goal-cases)
  case 1
  then show ?case
    by(intro wf-mlex wf-on-bot)
next
  case 2
```

```
  then show ?case
    by (intro mlex-less) (auto simp: card-gt-0-iff)
next
  case (3 A f l a b x)
  then have Max (f ‘ A) − Suc l < Max (f ‘ A) − l
   by (meson Max-gr-iff Max-in diff-less-mono2 finite-imageI imageI image-is-empty
lessI)
   with 3 have ((A, f, l + 1, a, b), A, f, l, a, b) ∈ (λ(A, f, l, a, b). Max (f ‘ A)
− l) <∗mlex∗> {}
     by (intro mlex-less) (auto)
   with 3 show ?case apply − apply(rule mlex-leq) by auto
next
  case 4
  then show ?case by (intro mlex-less) (auto simp: card-gt-0-iff)
qed

declare steps.simps[simp del]
```

lsteps is similar to steps but is using lists instead of sets. This makes the proofs where we use induction easier.

```
function lsteps :: ′a list ⇒ (′a ⇒ nat) ⇒ nat ⇒ nat ⇒ nat ⇒ nat where
  lsteps [] f l up left = 0 |
  lsteps (x#xs) f l up left = (if       f x < l then lsteps xs f l up left
                                else (if f x > l then up + lsteps (x#xs) f (l + 1) up left
                                 else                      left + lsteps xs f l up left))
  by pat-completeness auto
termination
proof (relation (λ(xs,f,l,a,b). length xs) <∗mlex∗> (λ(xs,f,l,a,b).
                Max (f ‘ set xs) − l) <∗mlex∗> {},
       goal-cases)
  case 1
  then show ?case
    by(intro wf-mlex wf-on-bot)
next
  case 2
  then show ?case
    by (auto intro: mlex-less simp: card-gt-0-iff)
next
  case (3 n f l a b)
  show ?case
   by (rule mlex-leq) (use 3 in ‹auto intro: mlex-less mlex-leq intro!: diff-less-mono2
simp add: Max-gr-iff›)
next
  case 4
  then show ?case by (intro mlex-less) (auto simp: card-gt-0-iff)
qed

declare lsteps.simps(2)[simp del]
```

```
lemma steps-empty [simp]: steps {} f l up left = 0
  by (simp add: steps.simps)

lemma steps-lsteps: steps A f l u v = lsteps (rev (sorted-list-of-set A)) f l u v
proof (cases finite A ∧ A ≠ {})
  case True
  then show ?thesis
  proof(induction (rev (sorted-list-of-set A)) f l u v arbitrary: A rule: lsteps.induct)
    case (2 y ys f l u v A)
    then have y-ys: y = Max A ys  = rev (sorted-list-of-set (A − {y}))
      by (auto simp add: sorted-list-of-set-Max-snoc)
    consider (a) l < f y | (b) f y < l | (c) f y = l
      by fastforce
    then have steps A f l u v = lsteps (y#ys) f l u v
    proof cases
      case a
      then show ?thesis
        by (subst steps.simps, subst lsteps.simps) (use y-ys 2 in auto)
    next
      case b
      then show ?thesis
       using y-ys 2(1) by (cases ys = []) (auto simp add: steps.simps lsteps.simps)
    next
      case c
      then have steps (A − {Max A}) f l u v =
                lsteps (rev (sorted-list-of-set (A − {Max A}))) f l u v
          by (cases A = {Max A}) (use y-ys 2 in ‹auto intro!: 2(3) simp add:
steps.simps›)
      then show ?thesis
        by (subst steps.simps, subst lsteps.simps) (use y-ys 2 in auto)
    qed
    then show ?case
      using 2 by simp
  qed (auto simp add: steps.simps)
qed (auto simp add: steps.simps)

lemma lsteps-comp-map: lsteps zs (f ∘ g) l u v = lsteps (map g zs) f l u v
  by (induction zs f ∘ g l u v rule: lsteps.induct) (auto simp add: lsteps.simps)

lemma steps-image:
  assumes finite A mono-on A g inj-on g A
  shows steps A (f ∘ g) l u v = steps (g ‘ A) f l u v
proof −
  have (sorted-list-of-set (g ‘ A)) = map g (sorted-list-of-set A)
    using sorted-list-of-set-image assms by auto
  also have rev ... = map g (rev (sorted-list-of-set A))
    using rev-map by auto
  finally show ?thesis
    by (simp add: steps-lsteps lsteps-comp-map)
```

qed

lemma *lsteps-cong*:
assumes $ys = xs$ $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x\ l = l'$
shows $lsteps\ xs\ f\ l\ u\ v = lsteps\ ys\ g\ l'\ u\ v$
using *assms* **proof** (*induction xs f l u v arbitrary*: *ys l′ rule*: *lsteps.induct*)
case (*2 x xs f l up left*)
then show *?case*
by (*subst* ‹*ys* = *x* # *xs*›, *subst lsteps.simps*, *subst* (*2*) *lsteps.simps*) *auto*
qed (*auto*)

lemma *steps-cong*:
assumes $A = B$ $\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x\ l = l'$
shows $steps\ A\ f\ l\ u\ v = steps\ B\ g\ l'\ u\ v$
using *assms*
by (*cases* $A = \{\} \vee$ *infinite A*) (*auto simp add*: *steps-lsteps steps.simps intro*!: *lsteps-cong*)

lemma *lsteps-f-add′*:
shows $lsteps\ xs\ f\ l\ u\ v = lsteps\ xs\ (\lambda x.\ f\ x + m)\ (l + m)\ u\ v$
by (*induction xs f l u v rule*: *lsteps.induct*) (*auto simp add*: *lsteps.simps*)

lemma *steps-f-add′*:
shows $steps\ A\ f\ l\ u\ v = steps\ A\ (\lambda x.\ f\ x + m)\ (l + m)\ u\ v$
by (*cases* $A = \{\} \vee$ *infinite A*) (*auto simp add*: *steps-lsteps steps.simps intro*!: *lsteps-f-add′*)

lemma *lsteps-smaller-set*:
assumes $m \leq l$
shows $lsteps\ xs\ f\ l\ u\ v = lsteps\ [x \leftarrow xs.\ m \leq f\ x]\ f\ l\ u\ v$
using *assms* **by** (*induction xs f l u v rule*: *lsteps.induct*) (*auto simp add*: *lsteps.simps*)

lemma *steps-smaller-set*:
assumes *finite A* $m \leq l$
shows $steps\ A\ f\ l\ u\ v = steps\ \{x \in A.\ f\ x \geq m\}\ f\ l\ u\ v$
using *assms*
by(*cases* $A = \{\} \vee$ *infinite A*)
(*auto simp add*: *steps-lsteps steps.simps rev-filter sorted-list-of-set-filter intro*!: *lsteps-smaller-set*)

lemma *lsteps-level-greater-fun-image*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x < l$
shows $lsteps\ xs\ f\ l\ u\ v = 0$
using *assms* **by** (*induction xs f l u v rule*: *lsteps.induct*) (*auto simp add*: *lsteps.simps*)

lemma *lsteps-smaller-card-Max-fun′*:
assumes $\exists x \in set\ xs.\ l \leq f\ x$
shows $lsteps\ xs\ f\ l\ u\ v + l * u \leq v * length\ xs + u * Max\ ((f\ `\ (set\ xs)) \cup \{0\})$
using *assms* **proof** (*induction xs f l u v rule*: *lsteps.induct*)

```
  case (1 f l up left)
  then show ?case by (simp)
next
  case (2 x xs f l up left)
  consider l = f x ∃ y∈set xs. l ≤ f y | f x = l ¬ (∃ y∈set xs. l ≤ f y) |
   f x < l | l < f x
    by fastforce
  then show ?case
  proof cases
    assume a: l = f x ∃ y∈set xs. l ≤ f y
    have lsteps (x # xs) f l up left + l ∗ up = lsteps xs f l up left + f x ∗ up + left
      using a by (auto simp add: lsteps.simps)
    also have lsteps xs f l up left + f x ∗ up ≤ left ∗ length xs + up ∗ Max (f ‘ set
xs ∪ {0})
      using a 2 by blast
    also have up ∗ Max (f ‘ set xs ∪ {0}) ≤ up ∗ Max (insert (f x) (f ‘ set xs))
      by simp
    finally  show ?case
      by auto
  next
    assume a: f x = l ¬ (∃ y∈set xs. l ≤ f y)
    have lsteps (x # xs) f l up left + l ∗ up = lsteps xs f l up left + f x ∗ up + left
      using a by (auto simp add: lsteps.simps)
    also have lsteps xs f l up left = 0
      using a by (subst lsteps-level-greater-fun-image) auto
    also have f x ∗ up ≤ up ∗ Max (insert (f x) (f ‘ set xs))
      by simp
    finally show ?case
      by simp
  next
    assume a: f x < l
    then have lsteps (x # xs) f l up left = lsteps xs f l up left
      by (auto simp add: lsteps.simps)
    also have ... + l ∗ up ≤ left ∗ length (x # xs) + up ∗ Max (insert 0 (f ‘ set
xs))
      using a 2 by auto
    also have Max (insert 0 (f ‘ set xs)) ≤ Max (f ‘ set (x # xs) ∪ {0})
      by simp
    finally show ?case
      by simp
  next
  assume f x > l
    then show ?case
      using 2 by (subst lsteps.simps) auto
  qed
qed

lemma steps-smaller-card-Max-fun′:
  assumes finite A ∃ x∈A. l ≤ f x
```

```
  shows   steps A f l up left + l ∗ up ≤ left ∗ card A + up ∗ Max0 (f ‘ A)
proof −
  let ?xs = rev (sorted-list-of-set A)
  have steps A f l up left  = lsteps (rev (sorted-list-of-set A)) f l up left
    using steps-lsteps by blast
  also have . . . + l ∗ up ≤ left ∗ length ?xs + up ∗ Max (f ‘ set ?xs ∪ {0})
    using assms by (intro lsteps-smaller-card-Max-fun′) auto
  also have left ∗ length ?xs = left ∗ card A
    using assms sorted-list-of-set-length by (auto)
  also have set ?xs = A
    using assms by (auto)
  finally show ?thesis
    by simp
qed

lemma lsteps-height:
  assumes  ∃ x ∈ set xs. l ≤ f x
  shows lsteps xs f l up 0 + up ∗ l = up ∗ Max0 (f ‘ (set xs))
  using assms proof (induction xs f l up 0::nat rule: lsteps.induct)
  case (2 x xs f l up)
  consider l = f x ∃ y∈set xs. l ≤ f y | f x = l ¬ (∃ y∈set xs. l ≤ f y) |
   f x < l | l < f x
    by fastforce
  then show ?case
 proof cases
   assume 0: l = f x ∃ y∈set xs. l ≤ f y
    then have 1: set xs ≠ {}
      using 2 by auto
    then have ∃ xa∈set xs. f x ≤ f xa
      using 0 2 by force
    then have f x ≤ Max (f ‘ set xs)
      using 0 2 by (subst Max-ge-iff) auto
    then have max (f x) (Max (f ‘ set xs)) = (Max (f ‘ set xs))
      using 0 2 by (auto intro!: simp add: max-def)
    then show ?case
      using 0 1 2 by (subst lsteps.simps) (auto)
  next
    assume 0: f x = l ¬ (∃ y∈set xs. l ≤ f y)
    then have Max (insert l (f ‘ set xs)) = l
      by (intro Max-eqI) (auto)
    moreover have lsteps xs f l up 0 = 0
      using 0 by (subst lsteps-level-greater-fun-image) auto
    ultimately show ?case
       using 0 by (subst lsteps.simps) auto
  next
    assume 0: f x < l
    then have 1: set xs ≠ {}
      using 2 by auto
    then have ∃ xa∈set xs. f x ≤ f xa
```

```
    using 0 2 by force
  then have  f x ≤ Max (f ‘ set xs)
    using 0 2 by (subst Max-ge-iff) auto
  then have max (f x) (Max (f ‘ set xs)) = Max (f ‘ set xs)
    using 0 2 by (auto intro!: simp add: max-def)
  then show ?case
    using 0 1 2 by (subst lsteps.simps) (auto)
next
assume f x > l
  then show ?case
    using 2 by (subst lsteps.simps) auto
qed
qed (simp)

lemma steps-height:
  assumes finite A
  shows   steps A f 0 up 0 = up ∗ Max₀ (f ‘ A)
proof −
  have steps A f 0 up 0 = lsteps (rev (sorted-list-of-set A)) f 0 up 0 + up ∗ 0
    by (subst steps-lsteps) simp
  also have ... = up ∗ Max (f ‘ A ∪ {0}) if A ≠ {}
    using assms that by (subst lsteps-height) auto
  finally show ?thesis
    using assms by (cases A = {}) (auto)
qed

context random-skip-list
begin
```

We can now define the pmf describing the length of the search path in a skip list. Like the height it only depends on the number of elements in the skip list's underlying set.

```
definition R where R A u l = map-pmf (λf. steps A f 0 u l) (SL A)
definition R_N :: nat ⇒ nat ⇒ nat ⇒ nat pmf where R_N n u l = R {..<n} u l


lemma R_N-alt-def: R_N n u l = map-pmf (λf. steps {..<n} f 0 u l) (SL_N n)
  unfolding SL_N-def R_N-def R-def by simp

context includes monad-normalisation
begin

lemma R-R_N:
  assumes finite A p ∈ {0..1}
  shows R A u l = R_N (card A) u l
proof −
  let ?steps = λA f. steps A f 0 u l
  let ?f′ = bij-mono-map-set-to-nat A
  have R A u l = SL A ≫= (λf. return-pmf (?steps A f))
```

```
    unfolding R-def map-pmf-def by simp
  also have ... = SL_N (card A) ≫= (λf. return-pmf (?steps A (f ∘ ?f')))
  proof −
    have ?f' x ∉ {..<card A} if x ∉ A for x
      using that unfolding bij-mono-map-set-to-nat-def by (auto)
    then show ?thesis
      using assms bij-mono-map-set-to-nat unfolding SL-def SL_N-def
      by (subst Pi-pmf-bij-betw[of - ?f' {..<card A}])
        (auto simp add: map-pmf-def)
  qed
  also have ... = SL_N (card A) ≫= (λf. return-pmf (?steps {..<card A} f))
   using assms bij-mono-map-set-to-nat bij-betw-def by (subst steps-image) (fastforce)+
  finally show ?thesis
    unfolding R_N-def R-def SL_N-def SL-def by (simp add: map-pmf-def)
qed
```

R_N fulfills a recurrence relation. If we move up or to the left the "remaining" length of the search path is again a slightly different probability distribution over the length.

```
lemma R_N-recurrence:
  assumes 0 < n p ∈ {0<..1}
  shows   R_N n u l =
          do {
            b ← bernoulli-pmf p;
            if b then                — leftwards
              map-pmf (λn. n + l) (R_N (n − 1) u l)
            else do {                — upwards
              m ← binomial-pmf (n − 1) (1 − p);
              map-pmf (λn. n + u) (R_N (m + 1) u l)
            }
          }
proof −
  define B where B = (λb. insert (n−1) {x ∈ {..<n − 1}. ¬ b x})
  have R_N n u l = map-pmf (λf. steps {..<n} f 0 u l) (SL_N n)
    by (auto simp add: R_N-def R-def SL_N-def)
  also have ... = map-pmf (λf. steps {..<n} f 0 u l)
                    (map-pmf (λ(y, f). f(n−1 := y)) (pair-pmf (geometric-pmf
p) (SL_N (n − 1))))
  proof −
    have {..<n} = insert (n − Suc 0) {..<n − 1}
      using assms by force
    then have (Pi-pmf {..<n} 0 (λ-. geometric-pmf p)) =
              map-pmf (λ(y, f). f(n − 1 := y)) (pair-pmf (geometric-pmf p)
                (Pi-pmf {..<n−1} 0 (λ-. geometric-pmf p)))
      using assms
     by (subst Pi-pmf-insert[of {..<n−1} n−1 0 λ-. geometric-pmf p, symmetric])
(auto)
    then show ?thesis
      by (simp add: SL_N-def SL-def)
```

```
  qed
  also have ... =
        do { g ← geometric-pmf p;
             f ← SL_N (n − 1);
             return-pmf (steps {..<n} (f(n − 1 := g)) 0 u l)}
    by (simp add: case-prod-beta map-pmf-def pair-pmf-def)
  also have ... =
        do { b ←  bernoulli-pmf p;
             g ← if b then return-pmf 0 else map-pmf Suc (geometric-pmf p);
             f ← SL_N (n − 1);
             return-pmf (steps {..<n} (f(n − 1 := g)) 0 u l)}
    using assms by (subst geometric-bind-pmf-unfold) (auto)
  also have ... =
        do { b ← bernoulli-pmf p;
             if b
               then do { g ← return-pmf 0;
                         f ← SL_N (n − 1);
                         return-pmf (steps {..<n} (f(n − 1 := g)) 0 u l) }
               else do { g ← map-pmf Suc (geometric-pmf p);
                         f ← SL_N (n − 1);
                         return-pmf (steps {..<n} (f(n − 1 := g)) 0 u l) }}
    by (subst bind-pmf-if') (auto)
  also have do { g ← return-pmf 0;
                 f ← SL_N (n − 1);
                 return-pmf (steps {..<n} (f(n − 1 := g)) 0 u l) } =
            do { f ← SL_N (n − 1);
                 return-pmf (steps {..<n} (f(n − 1 := 0)) 0 u l) }
    by (subst bind-return-pmf) auto
  also have ... = map-pmf (λn. n + l) (map-pmf (λf. steps {..<n − 1} f 0 u l)
(SL_N (n − 1)))
  proof −
    have I: {..<n} − {n − Suc 0} = {..<n − Suc 0}
      by fastforce
    have Max {..<n} = n − Suc 0
      using assms by (intro Max-eqI) (auto)
    then have steps {..<n} (f(n − 1 := 0)) 0 u l = l + steps {..<n − 1} f 0 u l
for f
      using assms by (subst steps.simps) (auto intro!: steps-cong simp add: I simp
add: Let-def)
    then show ?thesis
      by (auto simp add: add-ac map-pmf-def)
  qed
  also have ... = map-pmf (λn. n + l) (R_N (n − 1) u l)
    unfolding R_N-def R-def SL_N-def by simp
  also have map-pmf Suc (geometric-pmf p) ≫=
             (λg. SL_N (n − 1) ≫=
             (λf. return-pmf (steps {..<n} (f(n − 1 := g)) 0 u l)))
             =
             Pi-pmf {..<n − 1} True (λ-. bernoulli-pmf p) ≫=
```

```
        (λb. map-pmf Suc (geometric-pmf p) ≫=
      (λg. Pi-pmf {x ∈ {..<n − 1}. ¬ b x} 0 (λ-. map-pmf Suc (geometric-pmf
p)) ≫=
        (λf. return-pmf (steps {..<n} (f(n − 1 := g)) 0 u l))))
    using assms unfolding SL_N-def SL-def by (subst Pi-pmf-geometric-filter)
(auto)
  also have ... =
        do {
        b ← Pi-pmf {..<n − 1} True (λ-. bernoulli-pmf p);
        f ← Pi-pmf (insert (n−1) {x ∈ {..<n − 1}. ¬ b x}) 0 (λ-. map-pmf
Suc (geometric-pmf p));
        return-pmf (steps {..<n} f 0 u l)} (is - = ?rhs)
    using assms by (subst Pi-pmf-insert') (auto)
  also have ... =
        do {
          b ← Pi-pmf {..<n − 1} True (λ-. bernoulli-pmf p);
          f ← Pi-pmf (B b) 1 (λ-. map-pmf Suc (geometric-pmf p));
          return-pmf (steps {..<n} (λx. if x ∈ (B b) then f x else 0) 0 u l)}
   by (subst Pi-pmf-default-swap[symmetric, of - - - 1]) (auto simp add: map-pmf-def
B-def)
  also have ... =
        do {
          b ← Pi-pmf {..<n − 1} True (λ-. bernoulli-pmf p);
          f ← SL (B b);
          return-pmf (steps {..<n} (λx. if x ∈ (B b) then Suc (f x) else 0) 0 u
l)}
  proof −
    have ∗: (Suc ∘ f) x = Suc (f x) for x and f::nat ⇒ nat
      by simp
    have (λf. return-pmf (steps {..<n} (λx. if x ∈ B b then (Suc ∘ f) x else 0) 0
u l)) =
        (λf. return-pmf (steps {..<n} (λx. if x ∈ B b then Suc (f x) else 0) 0 u
l)) for b
      by (subst ∗) (simp)
    then show ?thesis
      by (subst Pi-pmf-map[of - - 0]) (auto simp add: map-pmf-def B-def SL-def)
  qed
  also have ... =
          do {
          b ← Pi-pmf {..<n − 1} True (λ-. bernoulli-pmf p);
          r ← R (B b) u l;
          return-pmf (u + r)}
  proof −
    have steps {..<n} (λx. if x ∈ B b then Suc (f x) else 0) 0 u l = u + steps (B
b) f 0 u l
      for f b
    proof −
      have Max {..<n} = n − 1
        using assms by (intro Max-eqI) auto
```

```
      then have steps {..<n} (λx. if x ∈ B b then Suc (f x) else 0) 0 u l =
            u + (steps {..<n} (λx. if x ∈ (B b) then Suc (f x) else 0) 1 u l)
          unfolding B-def using assms by (subst steps.simps) (auto simp add:
Let-def)
      also have steps {..<n} (λx. if x ∈ (B b) then Suc (f x) else 0) 1 u l =
                steps (B b) (λx. if x ∈ (B b) then Suc (f x) else 0) 1 u l
      proof −
        have {x ∈ {..<n}. 1 ≤ (if x ∈ B b then Suc (f x) else 0)} = B b
          using assms unfolding B-def by force
        then show ?thesis
          by (subst steps-smaller-set[of - 1]) auto
      qed
      also have ... = steps (B b) (λx. f x + 1) 1 u l
        by (rule steps-cong) (auto)
      also have ... = steps (B b) f 0 u l
        by (subst (2) steps-f-add'[of - - - - - 1]) simp
      finally show ?thesis
        by auto
    qed
    then show ?thesis
      by (simp add: R-def map-pmf-def)
  qed
  also have ... = do {
                b ← Pi-pmf {..<n − 1} False (λ-. bernoulli-pmf (1 − p));
                let m = 1 + card {x. x < n − 1 ∧ b x};
                r ← R {..<m} u l;
                return-pmf (u + r)}
  proof −
    have ∗: card (insert (n − Suc 0) {x. x < n − 1 ∧ b x}) =
            (Suc (card {x. x < n − 1 ∧ b x})) for b
      using assms by (auto simp add: card-insert-if)
    have Pi-pmf {..<n − 1} True (λ-. bernoulli-pmf p) =
            Pi-pmf {..<n − 1} True (λ-. map-pmf Not (bernoulli-pmf (1 − p)))
      using assms by (subst bernoulli-pmf-Not) auto
   also have ... = map-pmf ((∘) Not) (Pi-pmf {..<n − 1} False (λ-. bernoulli-pmf
(1 − p)))
      using assms by (subst Pi-pmf-map[of - - False]) auto
    finally show ?thesis
      unfolding B-def using assms ∗
      by (subst R-R_N) (auto simp add: R-R_N map-pmf-def)
  qed
  also have ... = binomial-pmf (n − 1) (1 − p) ≫= (λm. map-pmf (λn. n + u)
(R_N (m + 1) u l))
    using assms
    by (subst binomial-pmf-altdef'[where A = {..<n − 1} and dflt = False])
      (auto simp add: R_N-def R-def SL-def map-pmf-def ac-simps)
  finally show ?thesis
    by simp
qed
```

end

The expected height and length of search path defined as non-negative integral. It's easier to prove the recurrence relation of the expected length of the search path using non-negative integrals.

definition NH_N **where** $NH_N\ n = nn\text{-}integral\ (H_N\ n)\ real$
definition NR_N **where** $NR_N\ n\ u\ l = nn\text{-}integral\ (R_N\ n\ u\ l)\ real$

lemma $NH_N\text{-}EH_N$:
  assumes $p \in \{0<..<1\}$
  shows $NH_N\ n = EH_N\ n$
  using *assms* EH_N*-bounds* **unfolding** EH_N*-def* NH_N*-def* **by** (*subst nn-integral-eq-integral*) (*auto*)

lemma R_N*-0* [*simp*]: $R_N\ 0\ u\ l = return\text{-}pmf\ 0$
  unfolding R_N*-def R-def SL-def* **by** (*auto simp add*: *steps.simps*)

lemma NR_N*-bounds*:
  fixes $u\ l{::}nat$
  shows $NR_N\ n\ u\ l \le l * n + u * NH_N\ n$
proof −
  have $NR_N\ n\ u\ l = \int^+ x.\ x\ \partial measure\text{-}pmf\ (R_N\ n\ u\ l)$
    unfolding NR_N*-def* R_N*-alt-def*
    by (*simp add*: *ennreal-of-nat-eq-real-of-nat*)
  also have $\ldots \le \int^+ x.\ x\ \partial(measure\text{-}pmf\ (map\text{-}pmf\ (\lambda f.\ l * n + u * Max_0\ (f\ `\ \{..<n\}))\ (SL_N\ n)))$
    using *of-nat-mono*[*OF steps-smaller-card-Max-fun'*[*of* $\{..<n\}$ *0 - u l*]] **unfolding** R_N*-alt-def*
    by (*cases* $n = 0$) (*auto intro*!: *nn-integral-mono*)
  also have $\ldots = l * n + u * NH_N\ n$
    unfolding NH_N*-def* H_N*-def H-def* SL_N*-def*
    by (*auto simp add*: *nn-integral-add nn-integral-cmult ennreal-of-nat-eq-real-of-nat ennreal-mult*)
  finally show $NR_N\ n\ u\ l \le l * n + u * NH_N\ n$
    by *simp*
qed

lemma NR_N*-recurrence*:
  assumes $0 < n\ p \in \{0<..<1\}$
  shows $NR_N\ n\ u\ l = (p * (l + NR_N\ (n - 1)\ u\ l) +$
$q * (u + (\sum k{<}n - 1.\ NR_N\ (k + 1)\ u\ l * (pmf\ (binomial\text{-}pmf\ (n - 1)\ q)\ k))))$
$/\ (1 - (q\ \hat{}\ n))$
proof −
  define B **where** $B = (\lambda n\ k.\ pmf\ (binomial\text{-}pmf\ n\ q)\ k)$
  have q: $q \in \{0<..<1\}$
    using *assms* **unfolding** *q-def* **by** *auto*
  then have $q\ \hat{}\ n < 1$

using *assms power-Suc-less-one* **by** (*induction n*) (*auto*)
then have *qn*: $q \hat{} n \in \{0<..<1\}$
using *assms q* **by** (*auto*)
have $NR_N\ n\ u\ l = p * (l + NR_N\ (n - 1)\ u\ l) +$
$q * (u + \int^+ k.\ NR_N\ (k + 1)\ u\ l\ \partial measure\text{-}pmf\ (binomial\text{-}pmf\ (n - 1)\ q))$
using *assms* **unfolding** NR_N*-def*
by(*subst* R_N*-recurrence*)
(*auto simp add*: *field-simps nn-integral-add q-def ennreal-of-nat-eq-real-of-nat*)
also have $(\int^+ m.\ NR_N\ (m + 1)\ u\ l\ \partial measure\text{-}pmf\ (binomial\text{-}pmf\ (n - 1)\ q)) =$
$(\sum k \le n - 1.\ NR_N\ (k + 1)\ u\ l * B\ (n - 1)\ k)$
using *assms* **unfolding** *B-def q-def*
by (*auto simp add*: *nn-integral-measure-pmf-finite*)
also have $\ldots = (\sum k \in \{..<n - 1\} \cup \{n - 1\}.\ NR_N\ (k + 1)\ u\ l * B\ (n - 1)\ k)$
by (*rule sum.cong*) (*auto*)
also have $\ldots = (\sum k < n - 1.\ NR_N\ (k + 1)\ u\ l * B\ (n - 1)\ k) + NR_N\ n\ u\ l * q \hat{} (n - 1)$
unfolding *B-def q-def* **using** *assms* **by** (*subst sum.union-disjoint*) (*auto*)
finally have $NR_N\ n\ u\ l = p * (l + NR_N\ (n - 1)\ u\ l) +$
$q * ((\sum k < n - 1.\ NR_N\ (k + 1)\ u\ l * B\ (n - 1)\ k) + u) +$
$NR_N\ n\ u\ l * (q \hat{} (n - 1)) * q$
using *assms* **by** (*auto simp add*: *field-simps numerals*)
also have $NR_N\ n\ u\ l * (q \hat{} (n - 1)) * q = (q \hat{} n) * NR_N\ n\ u\ l$
using *q power-minus-mult*[*of - q*] *assms*
by (*subst mult-ac*, *subst ennreal-mult*[*symmetric*], *auto simp add*: *mult-ac*)
finally have *1*: $NR_N\ n\ u\ l = p * (l + NR_N\ (n - 1)\ u\ l) +$
$q * (u + (\sum k < n - 1.\ NR_N\ (k + 1)\ u\ l * (B\ (n - 1)\ k))) +$
$(q \hat{} n) * NR_N\ n\ u\ l$
by (*simp add*: *add-ac*)
have $x - z = y$ **if** $x = y + z$ $z \neq \top$ **for** $x\ y\ z$::*ennreal*
using *that* **by** (*subst that*) (*auto*)
have $NR_N\ n\ u\ l \le l * n + u * NH_N\ n$
using NR_N*-bounds* **by** (*auto simp add*: *ennreal-of-nat-eq-real-of-nat*)
also have $NH_N\ n = EH_N\ n$
using *assms* NH_N-EH_N **by** *auto*
also have $(l * n) + u * ennreal\ (EH_N\ n) < \top$
by (*simp add*: *ennreal-mult-less-top of-nat-less-top*)
finally have *3*: $NR_N\ n\ u\ l \neq \top$
by *simp*
have *2*: $x = y\ /\ (1 - a)$ **if** $x = y + a * x$ **and** *t*: $x \neq \top$ $a \in \{0<..<1\}$ **for** x y::*ennreal*
and a::*real*
proof −
have $y = x - a * x$
using *t* **by** (*subst that*) (*auto simp add*: *ennreal-mult-eq-top-iff*)
also have $\ldots = x * (ennreal\ 1 - ennreal\ a)$

using *that* **by** (*auto simp add*: *mult-ac ennreal-right-diff-distrib*)
also have *ennreal 1* − *ennreal a* = *ennreal* (*1* − *a*)
using *that* **by** (*subst ennreal-minus*) (*auto*)
also have $x * (1 - a) / (1 - a) = x$
using *that ennreal-minus-eq-0 not-less* **by** (*subst mult-divide-eq-ennreal*) *auto*
finally show *?thesis*
by *simp*
qed
have $NR_N\ n\ u\ l = (p * (l + NR_N\ (n - 1)\ u\ l) +$
$q * (u + (\sum k{<}n - 1.\ NR_N\ (k + 1)\ u\ l * (B\ (n - 1)\ k))))$
$/\ (1 - (q\ \hat{}\ n))$
using *1 3 assms qn* **by** (*intro 2*) *auto*
then show *?thesis*
unfolding *B-def* **by** *simp*
qed

lemma NR_n-NH_N: $NR_N\ n\ u\ 0 = u * NH_N\ n$
proof −
have $NR_N\ n\ u\ 0 = \int^+ f.\ steps\ \{..{<}n\}\ f\ 0\ u\ 0\ \partial measure\text{-}pmf\ (SL_N\ n)$
unfolding NR_N-*def* R_N-*alt-def* **by** (*auto simp add*: *ennreal-of-nat-eq-real-of-nat*)
also have $\ldots = \int^+ f.\ of\text{-}nat\ u * of\text{-}nat\ (Max_0\ (f\ `\ \{..{<}n\}))\ \partial measure\text{-}pmf\ (SL_N\ n)$
by (*intro nn-integral-cong*) (*auto simp add*: *steps-height*)
also have $\ldots = u * NH_N\ n$
by (*auto simp add*: NH_N-*def* H_N-*def H-def* SL_N-*def ennreal-of-nat-eq-real-of-nat nn-integral-cmult*)
finally show *?thesis*
by *simp*
qed

lemma NR_N-*recurrence'*:
assumes $0 < n\ p \in \{0{<}..{<}1\}$
shows $NR_N\ n\ u\ l = (p * l + p * NR_N\ (n - 1)\ u\ l +$
$q * u + q * (\sum k{<}n - 1.\ NR_N\ (k + 1)\ u\ l * (pmf\ (binomial\text{-}pmf\ (n - 1)\ q)\ k)))$
$/\ (1 - (q\ \hat{}\ n))$
unfolding NR_N-*recurrence*[*OF assms*]
by (*auto simp add*: *field-simps ennreal-of-nat-eq-real-of-nat ennreal-mult' ennreal-mult''*)

lemma NR_N-*l-0*:
assumes $0 < n\ p \in \{0{<}..{<}1\}$
shows $NR_N\ n\ u\ 0 = (p * NR_N\ (n - 1)\ u\ 0 +$
$q * (u + (\sum k{<}n - 1.\ NR_N\ (k + 1)\ u\ 0 * (pmf\ (binomial\text{-}pmf\ (n - 1)\ q)\ k))))$
$/\ (1 - (q\ \hat{}\ n))$
unfolding NR_N-*recurrence*[*OF assms*] **by** (*simp*)

lemma NR_N-*u-0*:
  assumes $0 < n$ $p \in \{0<..<1\}$
  shows $NR_N\ n\ 0\ l = (p * (l + NR_N\ (n - 1)\ 0\ l) +$
$q * (\sum k<n - 1.\ NR_N\ (k + 1)\ 0\ l * (pmf\ (binomial\text{-}pmf\ (n - 1)\ q)\ k)))$
$/\ (1 - (q\ \hat{}\ n))$
  unfolding NR_N-*recurrence*[*OF assms*] **by** (*simp*)

lemma NR_N-*0*[*simp*]: $NR_N\ 0\ u\ l = 0$
  unfolding NR_N-*def* R_N-*def* *R-def* **by** (*auto*)

lemma NR_N-*1*:
  assumes $p \in \{0<..<1\}$
  shows $NR_N\ 1\ u\ l = (u * q + l * p)\ /\ p$
proof −
  have $NR_N\ 1\ u\ l = (ennreal\ p * of\text{-}nat\ l + ennreal\ q * of\text{-}nat\ u)\ /\ ennreal\ (1 - q)$
    using *assms* **by** (*subst* NR_N-*recurrence*) *auto*
  also have $(ennreal\ p * of\text{-}nat\ l + ennreal\ q * of\text{-}nat\ u) = (u * q + l * p)$
    using *assms q-def* **by** (*subst ennreal-plus*)
      (*auto simp add*: *field-simps ennreal-mult′ ennreal-of-nat-eq-real-of-nat*)
  also have ... $/\ ennreal\ (1 - q) = ennreal\ ((u * q + l * p)\ /\ (1 - q))$
    using *q-def assms* **by** (*intro divide-ennreal*) *auto*
  finally show *?thesis*
    unfolding *q-def* **by** *simp*
qed

lemma NR_N-NR_N-*l-0*:
  assumes n: $0 < n$ **and** p: $p \in \{0<..<1\}$ **and** $u \geq 1$
  shows $NR_N\ n\ u\ 0 = (u * q\ /\ (u * q + l * p)) * NR_N\ n\ u\ l$
  using n **proof** (*induction n rule*: *less-induct*)
  case (*less i*)
  have *1*: $0 < u * q$
    unfolding *q-def* **using** *assms* **by** *simp*
  moreover have $0 \leq l * p$
    using *assms* **by** *auto*
  ultimately have *2*: $0 < u * q + l * p$
    by *arith*
  define c **where** $c = ennreal\ (u * q\ /\ (u * q + l * p))$
  have [*simp*]: $c\ /\ c = 1$
  proof −
    have $u * q\ /\ (u * q + l * p) \neq 0$
      using *assms q-def 2* **by** *auto*
    then show *?thesis*
      unfolding *c-def* **using** *p q-def* **by** (*auto intro*!: *ennreal-divide-self*)
  qed
  show *?case*
  proof (*cases* $i = 1$)
    case *True*

```
  have c ∗ NR_N i u l = c ∗ ((u ∗ q + l ∗ p) / p)
    unfolding c-def True by (subst NR_N-1[OF p]) auto
  also have ... = ennreal ((u ∗ q / (u ∗ q + l ∗ p)) ∗ ((u ∗ q + l ∗ p) / p))
    unfolding c-def using assms q-def by (subst ennreal-mult'') auto
  also have (u ∗ q / (u ∗ q + l ∗ p)) ∗ ((u ∗ q + l ∗ p) / p) = u ∗ q / p
  proof −
    have I: (a / b) ∗ (b / c) = a / c if 0 < b for a b c::real
      using that by (auto)
    show ?thesis
      using 2 q-def by (intro I) auto
  qed
  also have ... = NR_N i u 0
    unfolding True c-def by (subst NR_N-1[OF p]) (auto)
  finally show ?thesis
    unfolding c-def using True by simp
next
  case False
  then have i: i > 1
    using less by auto
  define c where c = ennreal (u ∗ q / (u ∗ q + l ∗ p))
 define B where B = (∑k<i − 1. NR_N (k + 1) u l ∗ ennreal (pmf (binomial-pmf
(i − 1) q) k))
  have NR_N i u 0 = (p ∗ NR_N (i − 1) u 0 +
                q ∗ (u + (∑k<i − 1. NR_N (k + 1) u 0 ∗ (pmf (binomial-pmf
(i − 1) q) k))))
                / (1 − (q ^ i))
    using less assms by (subst NR_N-l-0) auto
  also have q ∗ (u + (∑k<i − 1. NR_N (k + 1) u 0 ∗ (pmf (binomial-pmf (i
− 1) q) k))) =
          q ∗ u + q ∗ (∑k<i − 1. NR_N (k + 1) u 0 ∗ (pmf (binomial-pmf (i
− 1) q) k))
    using assms q-def
    by (auto simp add: field-simps ennreal-of-nat-eq-real-of-nat ennreal-mult)
  also have NR_N (i − 1) u 0 = c ∗ NR_N (i − 1) u l
    unfolding c-def using less i by (intro less) (auto)
  also have (∑k<i − 1. NR_N (k + 1) u 0 ∗ ennreal (pmf (binomial-pmf (i −
1) q) k)) =
          (∑k<i − 1. c ∗ NR_N (k + 1) u l ∗ ennreal (pmf (binomial-pmf (i −
1) q) k))
    by (auto intro!: sum.cong simp add: less c-def)
  also have ... = c ∗ B
    unfolding B-def by (subst sum-distrib-left) (auto intro!: sum.cong mult-ac)
  also have q ∗ (c ∗ B) = c ∗ (q ∗ B)
    by (simp add: mult-ac)
  also have ennreal (q ∗ real u) = q ∗ u ∗ ((u ∗ q + l ∗ p) / (u ∗ q + l ∗ p))
    using assms 2 by (auto simp add: field-simps q-def)
  also have ... = c ∗ (real u ∗ q + real l ∗ p)
     unfolding c-def using 2 by (subst ennreal-mult''[symmetric]) (auto simp
add: mult-ac)
```

also have $c * ennreal\ (real\ u * q + real\ l * p) + c * (ennreal\ q * B) =$
$c * (ennreal\ (real\ u * q + real\ l * p) + (ennreal\ q * B))$
by (*auto simp add*: *field-simps*)
also have $ennreal\ p * (c * NR_N\ (i - 1)\ u\ l) = c * (ennreal\ p * NR_N\ (i - 1)\ u\ l)$
by (*simp add*: *mult-ac*)
also have $(c * (ennreal\ p * NR_N\ (i - 1)\ u\ l) + c * (ennreal\ (u * q + l * p) + ennreal\ q * B))$
$= c * ((ennreal\ p * NR_N\ (i - 1)\ u\ l) + (ennreal\ (u * q + l * p) + ennreal\ q * B))$
by (*auto simp add*: *field-simps*)
also have $c * (ennreal\ p * NR_N\ (i - 1)\ u\ l + (ennreal\ (u * q + l * p) + ennreal\ q * B))\ /\ ennreal\ (1 - q\ \hat{}\ i)$
$= c * ((ennreal\ p * NR_N\ (i - 1)\ u\ l + (ennreal\ (u * q + l * p) + ennreal\ q * B))\ /\ ennreal\ (1 - q\ \hat{}\ i))$
by (*auto simp add*: *ennreal-times-divide*)
also have $(ennreal\ p * NR_N\ (i - 1)\ u\ l + (ennreal\ (real\ u * q + real\ l * p) + ennreal\ q * B))\ /\ ennreal\ (1 - q\ \hat{}\ i)$
$= NR_N\ i\ u\ l$
apply(*subst* (*2*) NR_N*-recurrence'*)
using *i assms q-def* **by**
(*auto simp add*: *field-simps B-def ennreal-of-nat-eq-real-of-nat ennreal-mult' ennreal-mult''*)
finally show *?thesis*
unfolding *c-def* **by** *simp*
qed
qed

Assigning 1 as the cost for going up and/or left, we can now show the relation between the expected length of the reverse search path and the expected height.

definition EL_N **where** $EL_N\ n = measure\text{-}pmf.expectation\ (R_N\ n\ 1\ 1)\ real$

theorem EH_N-EL_{sp}:
assumes $p \in \{0<..<1\}$
shows $1\ /\ q * EH_N\ n = EL_N\ n$
proof −
have *1*: $ennreal\ (1\ /\ y * x) = r$ **if** $ennreal\ x = y * r\ x \geq 0\ y > 0$
for $x\ y$::*real* **and** r::*ennreal*
proof −
have $ennreal\ ((1\ /\ y) * x) = ennreal\ (1\ /\ y) * ennreal\ x$
using *that* **apply**(*subst ennreal-mult''*) **by** *auto*
also note *that*(*1*)
also have $ennreal\ (1\ /\ y) * (ennreal\ y * r) = ennreal\ ((1\ /\ y) * y) * r$
using *that* **by** (*subst ennreal-mult''*) (*auto simp add*: *mult-ac*)
also have $(1\ /\ y) * y = 1$
using *that* **by** (*auto*)
finally show *?thesis*

```
    by auto
  qed
  have EH_N n = NH_N n
    using NH_N-EH_N assms by auto
  also have NH_N n = NR_N n 1 0
    using NR_n-NH_N by auto
  also have NR_N n 1 0 = q * NR_N n 1 1 if n > 0
    using NR_N-NR_N-l-0[of - 1 1] that assms q-def by force
  finally have ennreal (EH_N n) = q * NR_N n 1 1 if n > 0
    using that by blast
  then have 1 / q * EH_N n = NR_N n 1 1 if n > 0
    using that assms q-def by (intro 1) (auto simp add: EH_N-def H_N-def H-def)
  moreover have 1 / q * EH_N n = NR_N n 1 1 if n = 0
    unfolding that by (auto simp add: EH_N-def H_N-def H-def)
  ultimately have 2: ennreal (1 / q * EH_N n) = NR_N n 1 1
    by blast
  also have NR_N n 1 1 = EL_N n
    using 2 assms EH_N-bounds unfolding EL_N-def NR_N-def
    by(subst nn-integral-eq-integral)
      (auto intro!: integrableI-nn-integral-finite[where x=EH_N n / q])
  finally show ?thesis
     using assms q-def ennreal-inj unfolding EL_N-def EH_N-def H_N-def H-def
SL-def
    by (auto)
qed

end

thm random-skip-list.EH_N-EL_sp[unfolded random-skip-list.q-def]
    random-skip-list.EH_N-bounds'[unfolded random-skip-list.q-def]


end
```

References

[1] R. Motwani and P. Raghavan. *Randomized algorithms.* Cambridge university press, 1995.

[2] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449. Springer, 1989.