Verification of the CVM algorithm with a New Recursive Analysis Technique

Emin Karayel, Derek Khu, Kuldeep S. Meel, Yong Kiam Tan,
and Seng Joe Watt

March 19, 2025

Abstract

In 2022, Chakraborty et al. [1] published a streaming algorithm (henceforth, the CVM algorithm) for the distinct elements problem, that deviated considerably from the state-of-the art, due to its simplicity and avoidance of standard derandomization techniques, while still maintaining a close to optimal logarithmic space complexity.

In this entry, we verify the CVM algorithm's correctness using a new technique which simplifies the analysis considerably compared to the orignal proof by Chakraborty et al. The main idea is based on a probabilistic invariant that allows us to derive concentration bounds using the Cramér–Chernoff method.

This new technique opens up the possible algorithm design space, and we introduce a new variant of the CVM algorithm, that is total, and also has an additional property in addition to concentration: unbiasedness. This means the expected result of the algorithm is exactly equal to the desired result. The latter is also a new property, that neither the original CVM algorithm nor classic algorithms for the distinct elements problem possess.

Contents

1 Preliminary Definitions and Results

theory *CVM-Preliminary*
  imports *HOL−Probability.SPMF*
begin

lemma *bounded-finite*:
  assumes ‹*finite S*›
  shows ‹*bounded* (*f* ‘ *S*)›
  using *assms* **by** (*intro finite-imp-bounded*) *auto*

lemma *of-bool-power*:
  assumes ‹*y* > *0*›
  shows ‹(*of-bool x*::*real*) ^(*y*::*nat*) = *of-bool x*›
  by (*simp add*: *assms*)

lemma *card-filter-mono*:
  assumes ‹*finite S*›
  shows ‹*card* (*Set.filter p S*) ≤ *card S*›
  by (*intro card-mono assms*) *auto*

fun *foldM* ::
  ‹('*a* ⇒ ('*b* ⇒ '*c*) ⇒ '*c*) ⇒ ('*b* ⇒ '*c*) ⇒ ('*d* ⇒ '*b* ⇒ '*a*) ⇒ '*d list* ⇒ '*b* ⇒ '*c*› **where**
  ‹*foldM* - *return'* - [] *val* = *return' val*› |
  ‹*foldM bind' return' f* (*x* # *xs*) *val* =
    bind' (*f x val*) (*foldM bind' return' f xs*)›

abbreviation *foldM-pmf* ::
  ‹('*a* ⇒ '*b* ⇒ '*b pmf*) ⇒ '*a list* ⇒ '*b* ⇒ '*b pmf*› **where**
  ‹*foldM-pmf* ≡ *foldM bind-pmf return-pmf*›

lemma *foldM-pmf-snoc*: ‹*foldM-pmf f* (*xs*@[*y*]) *val* = *bind-pmf* (*foldM-pmf f xs val*) (*f y*)›
  by (*induction xs arbitrary*:*val*)
    (*simp-all add*: *bind-return-pmf bind-return-pmf' bind-assoc-pmf cong*:*bind-pmf-cong*)

abbreviation *foldM-spmf*
  :: ‹('*a* ⇒ '*b* ⇒ '*b spmf*) ⇒ '*a list* ⇒ '*b* ⇒ '*b spmf*› **where**
  ‹*foldM-spmf* ≡ *foldM bind-spmf return-spmf*›

lemma *foldM-spmf-snoc*: ‹*foldM-spmf f* (*xs*@[*y*]) *val* = *bind-spmf* (*foldM-spmf f xs val*) (*f*
y)›
  by (*induction xs arbitrary*:*val*) (*simp-all cong*:*bind-spmf-cong*)

abbreviation ‹*prob-fail* ≡ (λ*x*. *pmf x None*)›

abbreviation ‹*fail-spmf* ≡ *return-pmf None*›

abbreviation *fails-or-satisfies* :: ‹('*a* ⇒ *bool*) ⇒ '*a option* ⇒ *bool*› **where**
  ‹*fails-or-satisfies* ≡ *case-option True*›

lemma *prob-fail-foldM-spmf-le* :
  fixes
    p :: *real* **and**

```
    P :: ‹'b ⇒ bool› and
    f :: ‹'a ⇒ 'b ⇒ 'b spmf›
  assumes
    ‹⋀ x y z. P y ⟹ z ∈ set-spmf (f x y) ⟹ P z›
    ‹⋀ x val. P val ⟹ prob-fail (f x val) ≤ p›
    ‹P val›
  shows ‹prob-fail (foldM-spmf f xs val) ≤ real (length xs) ∗ p›
using assms(3) proof (induction xs arbitrary: val)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)

  have p-ge-0: ‹p ≥ 0› using Cons(2) assms(2) order-trans[OF pmf-nonneg] by metis

  let ?val' = ‹f x val›
  let ?μ' = ‹measure-spmf ?val'›

  have
    ‹prob-fail (foldM-spmf f (x # xs) val) =
      prob-fail ?val' + ∫ val'. prob-fail (foldM-spmf f xs val') ∂ ?μ'›
    by (simp add: pmf-bind-spmf-None)
  also have ‹. . . ≤ p + ∫ -. length xs ∗ p ∂ ?μ'›
    using assms(1)[OF Cons(2)]
     by (intro add-mono integral-mono-AE iffD2[OF AE-measure-spmf-iff] ballI assms(2)
Cons
        measure-spmf.integrable-const measure-spmf.integrable-const-bound[where B=‹1›])
     (simp-all add:pmf-le-1)
  also have ‹. . . ≤ p + weight-spmf (f x val)∗ length xs ∗ p›
    by simp
  also have ‹. . . ≤ p + 1 ∗ real (length xs) ∗ p›
    by (intro add-mono mult-right-mono p-ge-0 weight-spmf-le-1) simp-all
  finally show ?case by (simp add: algebra-simps)
qed

lemma foldM-spmf-of-pmf-eq :
  shows ‹foldM-spmf (λx y. spmf-of-pmf (f x y)) xs = spmf-of-pmf ∘ foldM-pmf f xs›
  (is ?thesis-0)
    and ‹foldM-spmf (λx y. spmf-of-pmf (f x y)) xs val = spmf-of-pmf (foldM-pmf f xs
val)›
  (is ?thesis-1)
proof −
  show ?thesis-0
    by (induction xs) (simp-all add: spmf-of-pmf-bind)

  then show ?thesis-1 by simp
qed

end
```

2 Abstract Algorithm

This section verifies an abstract version of the CVM algorithm, where the subsampling step can be an arbitrary randomized algorithm fulfilling an expectation invariant.

The abstract algorithm is presented in Algorithm 1.

Algorithm 1 Abstract CVM algorithm.

Input: Stream elements $a_1, \ldots, a_l$, $0 < \varepsilon$, $0 < \delta < 1$, $1/2 \le f < 1$
Output: An estimate R, s.t., $\mathcal{P}\left(|R - |A|| > \varepsilon |A|\right) \le \delta$ where $A := \{a_1, \ldots, a_l\}$.

$\chi \leftarrow \{\}, p \leftarrow 1, n \ge \left\lceil \frac{12}{\varepsilon^2} \ln(\frac{3l}{\delta}) \right\rceil$
for $i \leftarrow 1$ to l **do**
  $b \xleftarrow{\$} \text{Ber}(p)$ ▷ insert a_i with probability p (and remove it otherwise)
  if b **then**
    $\chi \leftarrow \chi \cup \{a_i\}$
  else
    $\chi \leftarrow \chi - \{a_i\}$
  if $|\chi| = n$ **then**
    $\chi \xleftarrow{\$} \text{subsample}(\chi)$ ▷ abstract subsampling step
    $p \leftarrow pf$
return $\frac{|\chi|}{p}$ ▷ estimate cardinality of A

For the subsampling step we assume that it fulfills the following inequality:

$$\int_{\text{subsample}(\chi)} \left(\prod_{i \in S} g(i \in \omega) \right) d\omega \le \prod_{i \in S} \left(\int_{Ber(f)} g(\omega)\, d\omega \right) \quad (1)$$

for all non-negative functions g and $S \subseteq \chi$, where $\text{Ber}(p)$ denotes the Bernoulli-distribution.

The original CVM algorithm uses a subsampling step where each element of χ is retained independently with probability f. It is straightforward to see that this fulfills the above condition (with equality).

The new CVM algorithm variant proposed in this work uses a subsampling step where a random nf-sized subset of χ is kept. This also fulfills the above inequality, although this is harder to prove and will be explained in more detail in Section 4.

In this section, we will verify that the above abstract algorithm indeed fulfills the desired conditions on its estimate, as well as unbiasedness, i.e., that: $\mathbb{E}[R] = |A|$. The part that is not going to be verified in this section, is the fact that the algorithm keeps at most n elements in the state χ, because it is not unconditionally true, but will be ensured (by different means) for the concrete instantiations in the following sections.

An informal version of this proof is presented in Appendix A. For important lemmas and theorems, we include a reference to the corresponding statement in the appendix.

theory *CVM-Abstract-Algorithm*

```
imports
  HOL−Decision-Procs.Approximation
  CVM-Preliminary
  Finite-Fields.Finite-Fields-More-PMF
  Universal-Hash-Families.Universal-Hash-Families-More-Product-PMF
begin

unbundle no vec-syntax

datatype 'a state = State (state-χ: ‹'a set›) (state-p: real)

datatype 'a run-state = FinalState ‹'a list› | IntermState ‹'a list› ‹'a›

lemma run-state-induct:
  assumes ‹P (FinalState [])›
  assumes ‹⋀xs x. P (FinalState xs) ⟹ P (IntermState xs x)›
  assumes ‹⋀xs x. P (IntermState xs x) ⟹ P (FinalState (xs@[x]))›
  shows ‹P result›
proof −
  have ‹P (FinalState xs) ∧ P (IntermState xs x)› for xs x
    using assms by (induction xs rule:rev-induct) auto
  thus ?thesis by (cases result) auto
qed

locale cvm-algo-abstract =
  fixes n :: nat and f :: real and subsample :: ‹'a set ⇒ 'a set pmf›
  assumes n-gt-0: ‹n > 0›
  assumes f-range: ‹f ∈ {1/2..<1}›
  assumes subsample:
    ‹⋀U x. card U = n ⟹ x ∈ set-pmf (subsample U) ⟹ x ⊆ U›
  assumes subsample-inequality:
    ‹⋀g U S. S ⊆ U
      ⟹ card U = n
      ⟹ range g ⊆ {0::real..}
      ⟹ (∫ω. (∏s∈S. g(s ∈ ω)) ∂subsample U) ≤ (∏s∈S. (∫ω. g ω ∂bernoulli-pmf f))›
begin
```

Line 1 of Algorithm 1:

```
definition initial-state :: ‹'a state› where
  ‹initial-state ≡ State {} 1›
```

Lines 3–7:

```
fun step-1 :: ‹'a ⇒ 'a state ⇒ 'a state pmf› where
  ‹step-1 a (State χ p) =
    do {
      b ← bernoulli-pmf p;
      let χ = (if b then χ ∪ {a} else χ − {a});

      return-pmf (State χ p)
    }›
```

Lines 8–10:

```
fun step-2 :: ‹'a state ⇒ 'a state pmf› where
  ‹step-2 (State χ p) = do {
    if card χ = n
    then do {
      χ ← subsample χ;
      return-pmf (State χ (p∗f))
    } else do {
      return-pmf (State χ p)
    }
  }›

schematic-goal step-1-def: ‹step-1 x σ = ?x›
  by (subst state.collapse[symmetric], subst step-1.simps, rule refl)

schematic-goal step-2-def: ‹step-2 σ = ?x›
  by (subst state.collapse[symmetric], subst step-2.simps, rule refl)
```

Lines 1–10:

```
definition run-steps :: ‹'a list ⇒ 'a state pmf› where
  ‹run-steps xs ≡ foldM-pmf (λx σ. step-1 x σ ≫= step-2) xs initial-state›
```

Line 11:

```
definition estimate :: ‹'a state ⇒ real› where
  ‹estimate σ = card (state-χ σ) / state-p σ›

lemma run-steps-snoc: ‹run-steps (xs @[x]) = run-steps xs ≫= step-1 x ≫= step-2›
  unfolding run-steps-def foldM-pmf-snoc by (simp add:bind-assoc-pmf)

fun run-state-pmf where
  ‹run-state-pmf (FinalState xs) = run-steps xs› |
  ‹run-state-pmf (IntermState xs x) = run-steps xs ≫= step-1 x›

fun len-run-state where
  ‹len-run-state (FinalState xs) = length xs› |
  ‹len-run-state (IntermState xs x) = length xs›

fun run-state-set where
  ‹run-state-set (FinalState xs) = set xs› |
  ‹run-state-set (IntermState xs x) = set xs ∪ {x}›

lemma finite-run-state-set[simp]: ‹finite (run-state-set σ)› by (cases σ) auto

lemma subsample-finite-pmf:
  assumes ‹card U = n›
  shows ‹finite (set-pmf (subsample U))›
proof−
 have ‹finite (Pow U)› unfolding finite-Pow-iff using assms n-gt-0 card-gt-0-iff by blast
 from finite-subset[OF - this] show ?thesis using subsample[OF assms] by auto
qed

lemma finite-run-state-pmf: ‹finite (set-pmf (run-state-pmf ϱ))›
proof (induction ϱ rule:run-state-induct)
  case 1 thus ?case by (simp add:run-steps-def)
```

```
next
  case (2 xs x) thus ?case by (simp add:step-1-def Let-def)
next
  case (3 xs x)
  have a: ‹run-state-pmf (FinalState (xs@[x])) = run-state-pmf (IntermState xs x) ⤜
step-2›
    by (simp add:run-steps-snoc)
  show ?case unfolding a using 3 subsample-finite-pmf
    by (auto simp:step-2-def simp del:run-state-pmf.simps)
qed

lemma state-χ-run-state-pmf: ‹AE σ in run-state-pmf ρ. state-χ σ ⊆ run-state-set ρ›
proof (induction ρ rule:run-state-induct)
  case 1 thus ?case by (simp add:run-steps-def AE-measure-pmf-iff initial-state-def)
next
  case (2 xs x)
  then show ?case by (simp add:AE-measure-pmf-iff Let-def step-1-def) auto
next
  case (3 xs x)
  let ?p = ‹run-state-pmf (IntermState xs x)›
  have b: ‹run-state-pmf (FinalState (xs@[x])) = ?p ⤜ step-2›
    by (simp add:run-steps-snoc)
  show ?case unfolding b using subsample 3 by (simp add:AE-measure-pmf-iff step-2-def
Let-def) blast
qed

lemma state-χ-finite: ‹AE σ in run-state-pmf ρ. finite (state-χ σ)›
  using finite-subset[OF - finite-run-state-set]
  by (intro AE-mp[OF state-χ-run-state-pmf AE-I2]) auto

lemma state-p-range: ‹AE σ in run-state-pmf ρ. state-p σ ∈ {0<..1}›
proof (induction ρ rule:run-state-induct)
  case 1 thus ?case by (simp add:run-steps-def AE-measure-pmf-iff initial-state-def)
next
  case (2 xs x)
  then show ?case by (simp add:AE-measure-pmf-iff Let-def step-1-def)
next
  case (3 xs x)
  let ?p = ‹run-state-pmf (IntermState xs x)›
  have b: ‹run-state-pmf (FinalState (xs@[x])) = ?p ⤜ step-2›
    by (simp add:run-steps-snoc)
  have ‹x * f ≤ 1› if ‹x ∈ {0<..1}› for x using f-range that by (intro mult-le-one) auto
  thus ?case unfolding b using 3 f-range by (simp add:AE-measure-pmf-iff step-2-def
Let-def)
qed
```

Lemma 1:

```
lemma run-steps-preserves-expectation-le:
  fixes φ :: ‹real ⇒ bool ⇒ real›
  assumes phi :
    ‹⋀ x b. ⟦0 < x; x ≤ 1⟧ ⟹ φ x b ≥ 0›
    ‹⋀ p x. ⟦0 < p; p ≤ 1; 0 < x; x ≤ 1⟧ ⟹ (∫ ω. φ x ω ∂bernoulli-pmf p) ≤ φ (x / p)
True›
```

‹*mono-on* {*0..1*} (*λx. φ x False*)›
defines ‹*aux* ≡ *λ S σ.* (∏ *x* ∈ *S. φ* (*state-p σ*) (*x* ∈ *state-χ σ*))›
assumes ‹*S* ⊆ *run-state-set ϱ*›
shows ‹*measure-pmf.expectation* (*run-state-pmf ϱ*) (*aux S*) ≤ *φ 1 True* ^ *card S*›
using *assms*(*5*)
proof (*induction ϱ arbitrary*: *S rule*: *run-state-induct*)
case *1* **thus** *?case* **by** (*simp add*:*aux-def*)
next
case (*2 xs x*)

have ‹*finite* (*set-pmf* (*run-steps xs*))›
using *finite-run-state-pmf*[**where** *ϱ*=‹*FinalState xs*›] **by** *simp*
note [*simp*] = *integrable-measure-pmf-finite*[*OF this*]

have *fin-S*: ‹*finite S*› **using** *finite-run-state-set 2*(*2*) *finite-subset* **by** *auto*

have *a*: ‹*aux S ω* = *aux* (*S* − {*x*}) *ω* ∗ *aux* (*S* ∩ {*x*}) *ω*› **for** *ω* :: ‹*'a state*›
unfolding *aux-def* **using** *fin-S* **by** (*subst prod.union-disjoint*[*symmetric*]) (*auto intro*:*prod.cong*)

have *b*: ‹*finite* (*set-pmf* (*run-steps xs* ⪢= *step-1 x*))›
using *finite-run-state-pmf*[**where** *ϱ*=‹*IntermState xs x*›] **by** *simp*

have *c*:‹(∫ *u. aux* (*S*∩{*x*}) *u* ∂*step-1 x τ*) ≤ *φ 1 True*^(*card* (*S* ∩ {*x*}))› (**is** ‹*?L* ≤ *?R*›)
if ‹*τ* ∈ *set-pmf* (*run-steps xs*)› **for** *τ*
proof(*cases* ‹*x* ∈ *S*›)
case *True*
have *p-range*: ‹*state-p τ* ∈ {*0*<..*1*}›
using *state-p-range*[**where** *ϱ*=‹*FinalState xs*›] *that* **by** (*auto simp*: *AE-measure-pmf-iff*)
have ‹*?L* = *measure-pmf.expectation* (*bernoulli-pmf* (*state-p τ*)) (*λx. φ* (*state-p τ*) *x*)›
unfolding *step-1-def Let-def map-pmf-def*[*symmetric*] *integral-map-pmf aux-def* **using** *True*
by (*intro integral-cong-AE AE-pmfI*) *simp-all*
also have ‹... ≤ *φ* (*state-p τ* / *state-p τ*) *True*› **using** *p-range* **by** (*intro phi*(*2*)) *auto*
also have ‹... ≤ *φ 1 True*› **using** *p-range* **by** *simp*
also have ‹... = *φ 1 True* ^ *card* (*S* ∩ {*x*})› **using** *True* **by** *simp*
finally show *?thesis* **by** *simp*
next
case *False* **thus** *?thesis* **by** (*simp add*:*aux-def*)
qed

have *d*: ‹*aux* (*S*−{*x*}) *τ* ≥ *0*› **if** ‹*τ* ∈ *set-pmf* (*run-steps xs*)› **for** *τ*
using *state-p-range*[**where** *ϱ*=‹*FinalState xs*›] *that* **unfolding** *aux-def*
by (*intro prod-nonneg phi*(*1*) *power-le-one*) (*auto simp*: *AE-measure-pmf-iff*)

have ‹(∫ *τ. aux S τ* ∂(*bind-pmf* (*run-steps xs*) (*step-1 x*))) =
(∫ *τ.* (∫ *u. aux* (*S* − {*x*}) *τ* ∗ *aux* (*S* ∩ {*x*}) *u* ∂*step-1 x τ*) ∂*run-steps xs*)›
unfolding *integral-bind-pmf*[*OF bounded-finite*[*OF b*]] *a*
by (*intro integral-cong-AE AE-pmfI arg-cong2*[**where** *f*=‹(∗)›] *prod.cong*)
(*auto simp*:*step-1-def aux-def Let-def*)
also have ‹... = (∫ *τ. aux* (*S*−{*x*}) *τ* ∗ (∫ *u. aux* (*S*∩{*x*}) *u* ∂*step-1 x τ*) ∂*run-steps xs*)›
by *simp*

also have ‹... ≤ (∫ τ. *aux* (S−{x}) τ ∗ (φ *1 True*)^ *card* (S∩{x}) ∂*run-steps xs*)›
 by (*intro integral-mono-AE AE-pmfI mult-left-mono c d*) *auto*
also have ‹... = (φ *1 True*) ^ *card* (S∩{x}) ∗ (∫ τ. *aux* (S−{x}) τ ∂(*run-state-pmf* (*FinalState xs*)))›
 by *simp*
also have ‹... ≤ (φ *1 True*) ^ *card* (S∩{x}) ∗ (φ *1 True*)^ *card* (S − {x})›
 using *phi*(*1*) *2*(*2*) **by** (*intro mult-left-mono 2*(*1*)) *auto*
also have ‹... = (φ *1 True*) ^(*card* ((S∩{x}) ∪ (S−{x})))›
 using *fin-S* **by** (*subst card-Un-disjoint*) (*auto simp add*:*power-add*)
also have ‹... = (φ *1 True*) ^ *card S*› **by** (*auto intro*!:*arg-cong2*[**where** f=‹λx y. x ^ (*card y*)›])
finally show *?case* **by** *simp*

next
case (*3 xs x*)
define p **where** ‹p = *run-state-pmf* (*IntermState xs x*)›
have a: ‹*run-state-pmf* (*FinalState* (xs@[x])) = p ⪢= *step-2*›
 by (*simp add*:*run-steps-snoc p-def*)

have *fin-S*: ‹*finite S*› **using** *finite-run-state-set 3*(*2*) *finite-subset* **by** *auto*

have ‹*finite* (*set-pmf p*)›
 using *finite-run-state-pmf*[**where** ϱ=‹*IntermState xs x*›] **by** (*simp add*:*p-def*)
note [*simp*] = *integrable-measure-pmf-finite*[*OF this*]

have b:‹*finite* (*set-pmf* (p ⪢= *step-2*))›
 using *finite-run-state-pmf*[**where** ϱ=‹*FinalState* (xs@[x])›] a **by** *simp*

have c: ‹(∫ u. (∏ s∈S. φ (f ∗ *state-p* τ) (s ∈ u)) ∂*subsample* (*state-χ* τ)) ≤ *aux* S τ›
 (**is** ‹*?L* ≤ *?R*›) **if** *that′*: ‹*card*(*state-χ* τ) = n› ‹τ ∈ *set-pmf p*› **for** τ
proof −
 let *?q* = ‹*subsample* (*state-χ* τ)›
 let *?U* = ‹*state-χ* τ›

 define p′ **where** ‹p′ = f ∗ *state-p* τ›

 have d: ‹y ⊆ *state-χ* τ› **if** ‹y ∈ *set-pmf* (*subsample* (*state-χ* τ))› **for** y
  using *subsample*[*OF that′*(*1*)] *that* **by** *auto*

 have e: ‹*state-p* τ ∈ {*0*<..*1*}›
  using *that*(*2*) **unfolding** *p-def* **using** *state-p-range* **by** (*meson AE-measure-pmf-iff*)
 hence f: ‹p′ ∈ {*0*<..*1*}› **unfolding** *p′-def* **using** *f-range* **by** (*auto intro*:*mult-le-one*)

 have ‹*?L* = (∫ u. (∏ s∈(S−*?U*) ∪ (S∩*?U*). φ p′ (s∈u)) ∂*?q*)›
  using *fin-S p′-def* **by** (*intro integral-cong-AE prod.cong AE-pmfI*) *auto*
 also have ‹... = (∫ u. (∏ s∈S−*?U*. φ p′ (s∈u)) ∗ (∏ s∈(S∩*?U*). φ p′ (s∈u)) ∂*?q*)›
  using *fin-S* **by** (*subst prod.union-disjoint*) *auto*
 also have ‹... = (∫ u. (∏ s∈S−*?U*. φ p′ *False*) ∗ (∏ s∈(S∩*?U*). φ p′ (s∈u)) ∂*?q*)›
  using d **by** (*intro integral-cong-AE AE-pmfI arg-cong2*[**where** f=‹(∗)›] *prod.cong arg-cong2*[**where** f=‹φ›]) *auto*
 also have ‹... = (∏ s∈S−*?U*. φ p′ *False*) ∗ (∫ u. (∏ s∈S∩*?U*. φ p′ (s∈u)) ∂*?q*)›
  by *simp*
 also have ‹... ≤ (∏ s∈S−*?U*. φ p′ *False*) ∗ (∏ s∈S∩*?U*. (∫ u. φ p′ u ∂*bernoulli-pmf* f))›

using *that f* **by** (*intro mult-left-mono subsample-inequality prod-nonneg*) (*auto intro*!:*phi*(*1*))
also have ‹... $\leq$ ($\prod s{\in}S-?U.\ \varphi\ p'$ *False*) $*$ ($\prod s{\in}S\cap ?U.\ \varphi\ (p'/f)$ *True*)›
using *f f-range*
by (*intro mult-left-mono prod-mono phi*(*2*) *conjI integral-nonneg-AE AE-pmfI phi*(*1*) *prod-nonneg*)
auto
also have ‹... $\leq$ ($\prod s{\in}S-?U.\ \varphi$ (*state-p* τ) *False*) $*$ ($\prod s{\in}S\cap ?U.\ \varphi$ (*state-p* τ) *True*)›
using *e f f-range* **unfolding** *p'-def*
by (*intro mult-mono prod-mono conjI phi*(*1*) *mono-onD*[*OF phi*(*3*)] *prod-nonneg power-le-one*) *auto*
also have ‹... = ($\prod s{\in}S-?U.\ \varphi$(*state-p* τ) ($s \in ?U$)) $*$ ($\prod s{\in}S\cap ?U.\ \varphi$(*state-p* τ) ($s \in ?U$))›
by *simp*
also have ‹... = ($\prod s{\in}(S-?U)\cup(S\cap ?U).\ \varphi$(*state-p* τ) ($s \in ?U$))›
using *fin-S* **by** (*subst prod.union-disjoint*[*symmetric*]) (*auto*)
also have ‹... = *aux S* τ› **unfolding** *aux-def* **by** (*intro prod.cong*) *auto*
finally show *?thesis* **by** *simp*
qed

have ‹($\int \tau$. *aux S* τ ∂*run-state-pmf* (*FinalState* (*xs*@[*x*]))) = ($\int \tau$. *aux S* τ ∂*bind-pmf p step-2*)›
unfolding *a* **by** *simp*
also have ‹... = ($\int \tau$. ($\int u$. *aux S u* ∂*step-2* τ) ∂p)› **by** (*intro integral-bind-pmf bounded-finite b*)
also have ‹... = ($\int \tau$. (*if card*(*state-*χ τ) = *n then*
($\int u$. ($\prod s{\in}S.\ \varphi$ ($f *$ *state-p* τ) ($s \in u$)) ∂*subsample* (*state-*χ τ)) *else aux S* τ) ∂p)›
unfolding *step-2-def map-pmf-def*[*symmetric*] *Let-def aux-def*
by (*simp add*:*if-distrib if-distribR ac-simps cong*:*if-cong*)
also have ‹... $\leq$ ($\int \tau$. (*if card*(*state-*χ τ) $<$ *n then aux S* τ *else aux S* τ) ∂p)›
using *c* **by** (*intro integral-mono-AE AE-pmfI*) *auto*
also have ‹... = ($\int \tau$. *aux S* τ ∂p)› **by** *simp*
also have ‹... $\leq \varphi$ *1 True* ^ *card S*› **using** *3*(*2*) **unfolding** *p-def* **by** (*intro 3*(*1*)) *auto*
finally show *?case* **by** *simp*
qed

Lemma 2:

lemma *run-steps-preserves-expectation-le'* :
fixes *q* :: *real* **and** *h* :: ‹*real* $\Rightarrow$ *real*›
assumes *h*:
‹$0 < q$› ‹$q \leq 1$›
‹*concave-on* {*0* .. *1* / *q*} *h*›
‹$\bigwedge x.\ [\![0 \leq x;\ x * q \leq 1]\!] \Longrightarrow h\ x \geq 0$›
defines
‹*aux* $\equiv \lambda S\ \sigma.$ ($\prod x \in S.$ *of-bool* (*state-p* $\sigma \geq q$) $*$ *h* (*of-bool* ($x \in$ *state-*χ σ) / *state-p* σ))›

assumes ‹$S \subseteq$ *run-state-set* ϱ›
shows ‹($\int \tau$. *aux S* τ ∂*run-state-pmf* ϱ) $\leq$ (*h 1*) ^ *card S*› (**is** ‹*?L* $\leq$ *?R*›)
proof −
define φ **where** ‹φ = ($\lambda p\ e.$ *of-bool* ($q \leq p$) $*$ *h*(*of-bool e* / *p*))›
have *φ-1*: ‹$\varphi\ x\ b \geq 0$› **if** ‹$x > 0$› ‹$x \leq 1$› **for** *x b*
using *h*(*1*,*4*) *that* **unfolding** *φ-def* **by** *simp*

have *φ-2*: ‹*φ x True* ∗ *p* + *φ x False* ∗ (*1* − *p*) ≤ *φ* (*x* / *p*) *True*› **if** ‹*x* ∈ {*0*<..*1*}› ‹*p* ∈ {*0*<..*1*}›
for *x p*
proof (*cases* ‹ *1* / *x* ∈ {*0*..*1* / *q*}›)
case *True*
hence *a*:‹*q* ≤ *x*› **using** *that*(*1*) *h*(*1*) **by** (*simp add*:*divide-simps*)
also have ‹. . . ≤ *x*/*p*› **using** *that* **by** (*auto simp add*:*divide-simps*)
finally have *b*:‹*q* ≤ *x* / *p*› **by** *simp*
show *?thesis*
unfolding *φ-def* **using** *that concave-onD*[*OF h*(*3*) - - - *True*, **where** *t*=‹*p*› **and** *x*=‹*0*›] *a b h*(*1*)
by (*auto simp*:*algebra-simps*)
next
case *False*
hence *a*:‹*q* > *x*› **using** *that h*(*1*) **by** (*auto simp add*:*divide-simps*)
hence ‹*q* ≤ *x*/*p* ⟹ *0* ≤ *h* (*p* / *x*)› **using** *that* **by** (*intro h*(*4*)) (*auto simp*:*field-simps*)
thus *?thesis* **using** *a* **by** (*simp add*:*φ-def*)
qed
have *φ-3*: ‹*φ x False* ≤ *φ y False*› **if** ‹*x* ≤ *y*› **for** *x y*
using *that* **by** (*auto intro*:*h*(*4*) *simp add*:*φ-def*)

have ‹*?L* = (∫ *τ*. (∏ *x*∈*S*. *φ* (*state-p τ*) (*x* ∈ *state-χ τ*)) ∂*run-state-pmf ϱ*)›
unfolding *φ-def* **by** (*simp add*:*state-p-def aux-def*)
also have ‹. . . ≤ *φ 1 True*^*card S*› **using** *φ-1 φ-2 φ-3*
by (*intro run-steps-preserves-expectation-le assms*) (*auto intro*:*mono-onI*)
also have ‹. . . = *h 1* ^ *card S*› **using** *h* **unfolding** *φ-def* **by** *simp*
finally show *?thesis* **by** *simp*
qed

Analysis of the case where *n* ≤ *card* (*set xs*):

context
fixes *xs* :: ‹*'a list*›
begin

private abbreviation ‹*A* ≡ *real* (*card* (*set xs*))›

context
assumes *set-larger-than-n*: ‹*card* (*set xs*) ≥ *n*›
begin

private definition ‹*q* = *real n* / (*4* ∗ *card* (*set xs*))›

lemma *q-range*: ‹*q* ∈ {*0*<..*1*/*4*}›
using *set-larger-than-n n-gt-0* **unfolding** *q-def* **by** *auto*

lemma *mono-nonnegI*:
assumes ‹⋀*x*. *x* ∈ *I* ⟹ *h*′ *x* ≥ *0*›
assumes ‹⋀*x*. *x* ∈ *I* ⟹ (*h has-real-derivative* (*h*′ *x*)) (*at x*)›
assumes ‹*x* ∈ *I* ∩ {*0*..}› ‹*convex I*› ‹*0* ∈ *I*› ‹*h 0* ≥ *0*›
shows ‹*h x* ≥ *0*›
proof −
have *h-mono*: ‹*h x* ≤ *h y*› **if** *that*′: ‹*x* ≤ *y*› ‹*x* ∈ *I*› ‹*y* ∈ *I*› **for** *x y*
proof −

have ‹{*x..y*} ⊆ *I*› **unfolding** *closed-segment-eq-real-ivl1*[*OF that*(*1*),*symmetric*]
by (*intro closed-segment-subset assms that*)
from *subsetD*[*OF this*]
show *?thesis* **using** *assms*(*1*,*2*) **by** (*intro DERIV-nonneg-imp-nondecreasing*[*OF that*(*1*)]) *auto*
qed

have ‹*0* ≤ *h 0*› **using** *assms* **by** *simp*
also have ‹. . . ≤ *h x*› **using** *assms* **by** (*intro h-mono*) *auto*
finally show *?thesis* **by** *simp*
qed

lemma *upper-tail-bound-helper*:
assumes ‹*x* ∈ {*0*<..*1*::*real*}›
defines ‹*h* ≡ (λ*x*. − *q* ∗ x^2 / *3* − *ln* (*1* + *q* ∗ *x*) + *q* ∗ *ln* (*1* + *x*) ∗ (*1* + *x*))›
shows ‹*h x* ≥ *0*›
proof −
define *h'* **where** ‹*h' x* = −*2*∗*x*∗*q*/*3* − *q*/(*1*+*q*∗*x*) + *q*∗*ln*(*1*+*x*) + *q*› **for** *x* :: *real*

have *a*: ‹(*h has-real-derivative* (*h' x*)) (*at x*)› **if** ‹*x* ≥ *0*› ‹*x* ≤ *1*› **for** *x*
proof −
have ‹*0* < (*1*::*real*) + *0*› **by** *simp*
also have ‹. . . ≤ *1* + *q* ∗ *x*› **using** *that q-range* **by** (*intro add-mono mult-nonneg-nonneg*) *auto*
finally have ‹*0* < *1* + *q* ∗ *x*› **by** *simp*
thus *?thesis*
using *that q-range* **unfolding** *h-def h'-def power2-eq-square*
by (*auto intro*!:*derivative-eq-intros*)
qed

have *b*: ‹*h' x* ≥ *0*› **if** ‹*x* ≥ *0*› ‹*x* ≤ *1*› **for** *x*
proof −
have ‹*exp*(*2*∗*x*/*3*) = *exp*((*1*−*x*) $*_R$ *0* + *x* $*_R$ (*2*/*3*))› **by** *simp*
also have ‹. . . ≤ (*1*−*x*) ∗ *exp 0* + *x* ∗ *exp*(*2*/*3*)›
using *that* **by** (*intro convex-onD*[*OF exp-convex*]) *auto*
also have ‹. . . = *1* + *x* ∗ (*exp* (*2*/*3*)−*exp 0*)› **by** (*simp add*:*algebra-simps*)
also have ‹. . . ≤ *1* + *x* ∗ *1*› **by** (*intro that add-mono order.refl mult-left-mono*) (*approximation 5*)
finally have ‹*exp*(*2*∗*x*/*3*) ≤ *exp* (*ln* (*1*+*x*))› **using** *that* **by** *simp*
hence ‹*0* ≤ *ln* (*1*+*x*) − *2* ∗ *x* / *3*› **by** *simp*
also have ‹. . . = *ln* (*1*+*x*) + *1* − *2*∗*x*/*3* − *1*› **by** *simp*
also have ‹. . . ≤ *ln* (*1*+*x*) + *1* − *2*∗*x*/*3* − *1*/(*1*+*q*∗*x*)›
using *q-range that* **by** (*intro add-mono diff-mono*) (*auto simp*:*divide-simps*)
finally have *c*: ‹*0* ≤ *ln* (*1*+*x*) + *1* − *2*∗*x*/*3* − *1*/(*1*+*q*∗*x*)› **by** *simp*

have ‹*h' x* = *q* ∗ (−*2*∗*x*/*3* − *1*/(*1*+*q*∗*x*) + *ln* (*1*+*x*) + *1*)›
unfolding *h'-def* **by** (*simp add*:*algebra-simps*)
also have ‹. . . ≥ *0*› **using** *c q-range* **by** (*intro mult-nonneg-nonneg*) *auto*
finally show *?thesis* **by** *simp*
qed

show *?thesis* **by** (*rule mono-nonnegI*[**where** *I*=‹{*0*..*1*}›, *OF b a*]) (*use assms*(*1*) *h-def* **in** *simp-all*)

qed

private definition *ϑ* **where** ‹*ϑ t x* = *1* + *q* ∗ *x* ∗ (*exp* (*t* / *q*) − *1*)›

lemma *ϑ-concave*: ‹*concave-on* {*0..1* / *q*} (*ϑ t*)›
  unfolding *ϑ-def* **by** (*intro concave-on-linorderI*) (*auto simp*:*algebra-simps*)

lemma *ϑ-ge-exp-1*:
  assumes ‹*x* ∈ {*0..1*/*q*}›
  shows ‹*exp* (*t* ∗ *x*) ≤ *ϑ t x*›
proof −
  have ‹*exp* (*t* ∗ *x*) = *exp* ((*1*−*q*∗*x*) $*_R$ *0* + (*q*∗*x*) $*_R$ (*t*/*q*))› **using** *q-range* **by** *simp*
  also have ‹. . . ≤ (*1*−*q*∗*x*) ∗ *exp 0* + (*q*∗*x*) ∗ *exp* (*t*/*q*)› **using** *assms q-range*
    by (*intro convex-onD*[*OF exp-convex*]) (*auto simp*:*field-simps*)
  also have ‹. . . = *ϑ t x*› **unfolding** *ϑ-def* **by** (*simp add*:*algebra-simps*)
  finally show *?thesis* **by** *simp*
qed

lemma *ϑ-ge-exp*:
  assumes ‹*y* ≥ *q*›
  shows ‹*exp* (*t* / *y*) ≤ *ϑ t* (*1* / *y*)›
  using *assms ϑ-ge-exp-1*[**where** *x*=‹*1*/*y*› **and** *t*=*t*] *q-range* **by** (*auto simp*:*field-simps*)

lemma *ϑ-nonneg*:
  assumes ‹*x* ∈ {*0..1*/*q*}›
  shows ‹*ϑ t x* ≥ *0*› ‹*ϑ t x* > *0*›
proof −
  have ‹*0* < *exp* (*t* ∗ *x*)› **by** *simp*
  also have ‹. . . ≤ *ϑ t x*› **by** (*intro ϑ-ge-exp-1 assms*)
  finally show ‹*ϑ t x* > *0*› **by** *simp*
  thus ‹*ϑ t x* ≥ *0*› **by** *simp*
qed

lemma *ϑ-0*: ‹*ϑ t 0* = *1*› **unfolding** *ϑ-def* **by** *simp*

lemma *tail-bound-aux*:
  assumes ‹*run-state-set ϱ* ⊆ *set xs*› ‹*c* > *0*›
  defines ‹*A′* ≡ *real* (*card* (*run-state-set ϱ*))›
  shows ‹*measure* (*run-state-pmf ϱ*) {*ω*. *exp* (*t* ∗ *estimate ω*) ≥ *c* ∧ *state-p ω* ≥ *q*} ≤ *ϑ t*
1 powr A′/*c*›
    (**is** ‹*?L* ≤ *?R*›)
proof −
  let *?p* = ‹*run-state-pmf ϱ*›
  note [*simp*] = *integrable-measure-pmf-finite*[*OF finite-run-state-pmf*]

  let *?A′* = ‹*run-state-set ϱ*›
  let *?X* = ‹*λi ω*. *of-bool* (*i* ∈ *state-χ ω*) / *state-p ω*›

  have *a*: ‹*0* < *ϑ t 1*› **using** *q-range* **by** (*intro ϑ-nonneg*) *auto*

  have ‹*?L* ≤ 𝒫(*ω in ?p*. *of-bool*(*state-p ω* ≥ *q*) ∗ *exp* (*t*∗*estimate ω*) ≥ *c*)›
    by (*intro pmf-mono*) *auto*
  also have ‹. . . ≤ (∫*ω*. *of-bool*(*state-p ω* ≥ *q*) ∗ *exp* (*t*∗*estimate ω*) ∂*?p*) / *c*›

```
    by (intro integral-Markov-inequality-measure[where A=‹{}›] assms(2)) simp-all
  also have ‹... = (∫ω. of-bool(state-p ω ≥ q) * exp((∑ i∈?A'. t * ?X i ω)) ∂?p)/c›
    using state-χ-run-state-pmf[where ϱ=‹ϱ›] Int-absorb1
    unfolding sum-divide-distrib[symmetric] sum-distrib-left[symmetric] estimate-def
    by (intro integral-cong-AE arg-cong2[where f=‹(/)›])
      (auto simp:AE-measure-pmf-iff intro!:arg-cong[where f=‹card›])
  also have ‹... ≤ (∫ω. (∏ i ∈ ?A'. of-bool(state-p ω ≥ q) * exp(t * ?X i ω)) ∂?p) / c›
    unfolding exp-sum[OF finite-run-state-set] prod.distrib using assms(2)
    by (intro divide-right-mono integral-mono-AE AE-pmfI)
      (auto intro!:mult-nonneg-nonneg prod-nonneg)
  also have ‹... ≤ (∫ω. (∏ i ∈ ?A'. of-bool(state-p ω ≥ q) * ϑ t (?X i ω)) ∂?p) / c›
    using assms(2) ϑ-ge-exp ϑ-0 by (intro divide-right-mono integral-mono-AE AE-pmfI
prod-mono) auto
  also have ‹... ≤ ϑ t 1 ^ card ?A' / c› using q-range ϑ-concave assms(2)
    by (intro divide-right-mono run-steps-preserves-expectation-le' ϑ-nonneg)
      (auto intro!:ϑ-nonneg simp:field-simps)
  also have ‹... ≤ ?R›
    unfolding A'-def using card-mono[OF - assms(1)] assms(2) a
    by (subst powr-realpow) (auto intro!:power-increasing divide-right-mono)
  finally show ?thesis by simp
qed
```

Lemma 3:

```
lemma upper-tail-bound:
  assumes ‹ε ∈ {0<..1::real}›
  assumes ‹run-state-set ϱ ⊆ set xs›
  shows ‹measure (run-state-pmf ϱ) {ω. estimate ω ≥ (1+ε)*A ∧ state-p ω ≥ q} ≤
exp(−real n/12*ε^2)›
    (is ‹?L ≤ ?R›)
proof −
  let ?p = ‹run-state-pmf ϱ›
  define t where ‹t = q * ln (1+ε)›

  have t-gt-0: ‹t > 0› unfolding t-def using q-range assms(1) by auto

  have mono-exp-t: ‹strict-mono (λ(x::real). exp (t * x))›
    using t-gt-0 by (intro strict-monoI) auto

  have a: ‹ϑ t 1 = 1 + q * ε› using assms(1) unfolding ϑ-def t-def by simp
  have b: ‹ϑ t 1 ≥ 1› unfolding a using q-range assms(1) by auto

  have c: ‹ln (ϑ t 1) − t * (1 + ε) ≤ − q * ε^2 / 3›
    using upper-tail-bound-helper[OF assms(1)]
    unfolding a unfolding t-def by (simp add:algebra-simps)

  have ‹?L = measure ?p {ω. exp (t * estimate ω) ≥ exp (t*((1+ε)* A))∧state-p ω≥ q}›
    by (subst strict-mono-less-eq[OF mono-exp-t]) simp
  also have ‹... ≤ ϑ t 1 powr real (card (run-state-set ϱ)) / exp (t * ((1+ε)* A))›
    by (intro tail-bound-aux assms) auto
  also have ‹... ≤ ϑ t 1 powr A / exp (t * ((1+ε)* A))›
    using card-mono[OF finite-set assms(2)] b
    by (intro powr-mono divide-right-mono) auto
  also have ‹... = exp (A * (ln (ϑ t 1) − t * (1 + ε)))›
```

```
    using b unfolding powr-def by (simp add:algebra-simps exp-diff)
  also have ‹... ≤ exp (A * (−q * ε^2/3))›
    by (intro iffD2[OF exp-le-cancel-iff] mult-left-mono c) simp
  also have ‹... = ?R› using set-larger-than-n n-gt-0 unfolding q-def by auto
  finally show ?thesis by simp
qed
```

Lemma 4:

```
lemma low-p:
  shows ‹measure (run-steps xs) {σ. state-p σ < q} ≤ real (length xs) * exp(−real n/12)›
    (is ‹?L ≤ ?R›)
proof −
  define ϱ where ‹ϱ = FinalState xs›

  have ih: ‹run-state-set ϱ ⊆ set xs› unfolding ϱ-def by simp

  have ‹?L = measure (run-state-pmf ϱ) {ω. state-p ω < q}›
    unfolding ϱ-def run-state-pmf.simps by simp
  also have ‹... ≤ real (len-run-state ϱ) * exp(− real n/12)›
    using ih
  proof (induction ϱ rule:run-state-induct)
    case 1
    then show ?case using q-range by (simp add:run-steps-def initial-state-def)
  next
    case (2 ys x)
    let ?pmf = ‹run-state-pmf (IntermState ys x)›
    have a:‹run-state-set (FinalState ys) ⊆ set xs› using 2(2) by auto

    have ‹measure ?pmf {ω. state-p ω < q} = (∫σ. of-bool (state-p σ < q) ∂run-steps ys)›
      unfolding run-state-pmf.simps step-1-def Let-def by (simp add:measure-bind-pmf
indicator-def)
    also have ‹... = (∫σ. indicator {ω. (state-p ω < q)} σ ∂run-steps ys)›
      by (intro integral-cong-AE AE-pmfI) simp-all
    also have ‹... = measure (run-steps ys) {ω. (state-p ω < q)}› by simp
    also have ‹... ≤ real (len-run-state (IntermState ys x)) * exp (− real n / 12)›
      using 2(1)[OF a] by simp
    finally show ?case by simp
  next
    case (3 ys x)
    define p where ‹p = run-state-pmf (IntermState ys x)›

    have ‹finite (set-pmf p)› unfolding p-def by (intro finite-run-state-pmf)
    note [simp] = integrable-measure-pmf-finite[OF this]

    have a: ‹run-state-pmf (FinalState (ys@[x])) = p ⤜ step-2› (is ‹?pmf= -›)
      by (simp add:run-steps-snoc p-def)

    have b: ‹run-state-set (IntermState ys x) ⊆ set xs›
      using 3(2) by simp

    have c:‹measure (step-2 σ) {σ. state-p σ < q} ≤
      indicator {σ. state-p σ < q ∨ (card (state-χ σ) = n ∧ state-p σ ∈ {q..<q/f}) } σ›
      for σ :: ‹'a state›
```

using *f-range*
by (*simp add*:*step-2-def Let-def indicator-def map-pmf-def*[*symmetric*] *divide-simps*)

have *d*: ‹*2* ∗ *real* (*card* (*set xs*)) ≤ *real n* / *α*› **if** ‹*α* ∈ {*q*..<*q* / *f*}› **for** *α*
proof −
have ‹*α* ≤ *q* ∗ (*1*/*f*)› **using** *that* **by** *simp*
also have ‹. . . ≤ *q* ∗ *2*› **using** *q-range f-range* **by** (*intro mult-left-mono*) (*auto simp*:*divide-simps*)
finally have ‹*α* ≤ *2*∗*q*› **by** *simp*
hence ‹*α* ≤ *real n* / (*2* ∗ *real* (*card* (*set xs*)))›
using *set-larger-than-n n-gt-0* **unfolding** *q-def* **by** (*simp add*:*divide-simps*)
thus *?thesis*
using *set-larger-than-n n-gt-0 that q-range* **by** (*simp add*:*field-simps*)
qed

hence ‹*measure p* {*σ*. *card* (*state-χ σ*) = *n* ∧ *state-p σ* ∈ {*q*..<*q*/*f*}} ≤
measure p {*σ*. (*1*+*1*) ∗ *A* ≤ *estimate σ* ∧ *q* ≤ *state-p σ*}›
unfolding *estimate-def* **by** (*intro pmf-mono*) (*simp add*:*estimate-def*)
also have ‹. . . ≤ *exp* (− *real n* / *12* ∗ *1*^*2*)›
unfolding *p-def* **by** (*intro upper-tail-bound b*) *simp*
finally have *e*:
‹*measure p* {*σ*. *card* (*state-χ σ*) = *n* ∧ *state-p σ* ∈ {*q*..<*q*/*f*}} ≤ *exp* (− *real n* / *12*)›
by *simp*

have ‹*measure* (*run-state-pmf* (*FinalState* (*ys* @ [*x*]))) {*ω*. *state-p ω* < *q*} =
(∫ *s*. *measure* (*step-2 s*) {*ω*. *state-p ω* < *q*} ∂*p*)›
unfolding *a* **by** (*simp add*:*measure-bind-pmf*)
also have ‹. . . ≤ (∫ *s*. *indicator* {*ω*. *state-p ω* < *q* ∨ *card* (*state-χ ω*) = *n* ∧ *state-p ω* ∈ {*q*..<*q*/*f*}} *s* ∂*p*)›
by (*intro integral-mono-AE AE-pmfI c*) *simp-all*
also have ‹. . . = *measure p* {*ω*. *state-p ω* < *q* ∨ *card* (*state-χ ω*) = *n* ∧ *state-p ω* ∈ {*q*..<*q*/*f*}}›
by *simp*
also have ‹. . . ≤*measure p* {*ω*. *state-p ω*<*q*}+*measure p* {*ω*. *card*(*state-χ ω*)=*n*∧*state-p ω*∈{*q*..<*q*/*f*}}›
by (*intro pmf-add*) *auto*
also have ‹. . . ≤ *length ys* ∗ *exp* (− *real n* / *12*) + *exp* (− *real n* / *12*)›
using *3*(*1*)[*OF b*] **by** (*intro add-mono e*) (*simp add*:*p-def*)
also have ‹. . . = *real* (*len-run-state* (*FinalState* (*ys* @ [*x*]))) ∗ *exp* (− *real n* / *12*)›
by (*simp add*:*algebra-simps*)
finally show *?case* **by** *simp*
qed
also have ‹. . . = *real* (*length xs*) ∗ *exp*(− *real n*/*12*)› **by** (*simp add*:*ϱ-def*)
finally show *?thesis* **by** *simp*
qed

lemma *lower-tail-bound-helper*:
assumes ‹*x* ∈ {*0*<..<*1*::*real*}›
defines ‹*h* ≡ (*λx*. − *q* ∗ x^2 / *2* − *ln* (*1* − *q* ∗ *x*) + *q* ∗ *ln* (*1* − *x*) ∗ (*1* − *x*))›
shows ‹*h x* ≥ *0*›
proof −
define *h'* **where** ‹*h' x* = −*x*∗*q* + *q*/(*1*−*q*∗*x*) − *q*∗*ln*(*1*−*x*) − *q*› **for** *x*

```
  have a: ‹(h has-real-derivative (h' x)) (at x)› if ‹x ≥ 0› ‹x < 1› for x
  proof -
    have ‹q * x ≤ (1/4) * 1› using that q-range by (intro mult-mono) auto
    also have ‹... < 1› by simp
    finally have ‹q * x < 1› by simp
    thus ?thesis
      using that q-range unfolding h-def h'-def power2-eq-square
      by (auto intro!:derivative-eq-intros)
  qed

  have b: ‹h' x ≥ 0› if ‹x ≥ 0› ‹x < 1› for x
  proof -
    have ‹q * x ≤ (1/4) * 1› using that q-range by (intro mult-mono) auto
    also have ‹... < 1› by simp
    finally have a:‹q * x < 1› by simp

    have ‹0 ≤ - ln (1 - x) - x› using ln-one-minus-pos-upper-bound[OF that] by simp
    also have ‹... = - ln (1 - x) - 1 - x + 1› by simp
    also have ‹... ≤ - ln (1 - x) - 1 - x + 1 / (1 - q * x)›
      using a q-range that by (intro add-mono diff-mono) (auto simp:divide-simps)
    finally have b: ‹0 ≤ - ln (1 - x) - 1 - x + 1 / (1 - q * x)› by simp

    have ‹h' x = q * (-x + 1 / (1 - q * x) - ln (1 - x) - 1)›
      unfolding h'-def by (simp add:algebra-simps)
    also have ‹... ≥ 0›  using b q-range by (intro mult-nonneg-nonneg) auto
    finally show ?thesis by simp
  qed

  show ?thesis by (rule mono-nonnegI[where I=‹{0..<1}›, OF b a]) (use assms(1) h-def
in simp-all)
qed
```

Lemma 5:

```
lemma lower-tail-bound:
  assumes ‹ε ∈ {0<..<1::real}›
  shows ‹measure (run-steps xs) {ω. estimate ω ≤ (1-ε) * A ∧ state-p ω ≥ q} ≤ exp(-real
n/8*ε^2)›
    (is ‹?L ≤ ?R›)
proof -
  let ?p = ‹run-state-pmf (FinalState xs)›
  define t where ‹t = q * ln (1-ε)›

  have t-lt-0: ‹t < 0›
    unfolding t-def using q-range assms(1) by (intro mult-pos-neg ln-less-zero) auto

  have mono-exp-t: ‹exp (t * x) ≤ exp (t * y) ⟷ y ≤ x› for x y using t-lt-0 by auto

  have a: ‹ϑ t 1 = 1 - q * ε› using assms(1) unfolding ϑ-def t-def by simp
  have ‹ε * (q * 4) ≤ 1 * 1› using q-range assms(1) by (intro mult-mono) auto
  hence b: ‹ϑ t 1 ≥ 3/4› unfolding a by (auto simp:algebra-simps)

  have c: ‹ln (ϑ t 1) - t * (1 - ε) ≤ - q * ε^2 / 2›
    unfolding a unfolding t-def using lower-tail-bound-helper[OF assms(1)]
```

by (*simp add*:*divide-simps*)

have ‹*?L = measure ?p* {*ω. exp* (*t* ∗ *estimate ω*) ≥ *exp* (*t* ∗ ((*1*−*ε*) ∗ *A*)) ∧ *state-p ω* ≥ *q*}›
by (*subst mono-exp-t*) *simp*
also have ‹. . . ≤ *ϑ t 1 powr card* (*run-state-set* (*FinalState xs*)) / *exp* (*t* ∗ ((*1* − *ε*) ∗ *A*))›
by (*intro tail-bound-aux assms*) *auto*
also have ‹. . . ≤ *ϑ t 1 powr A* / *exp* (*t* ∗ ((*1* − *ε*) ∗ *A*))› **by** *simp*
also have ‹. . . = *exp* (*A* ∗ (*ln* (*ϑ t 1*) − *t* ∗ (*1* − *ε*)))›
using *b* **unfolding** *powr-def* **by** (*simp add*:*algebra-simps exp-add*[*symmetric*] *exp-diff*)
also have ‹. . . ≤ *exp* (*A* ∗ (− *q* ∗ *ε* ^ *2* / *2*))›
by (*intro iffD2*[*OF exp-le-cancel-iff*] *mult-left-mono c*) *simp*
also have ‹. . . = *?R*› **using** *set-larger-than-n n-gt-0* **unfolding** *q-def* **by** *auto*
finally show *?thesis* **by** *simp*
qed

lemma *correctness-aux*:
assumes ‹*ε* ∈ {*0*<..<*1*::*real*}› ‹*δ* ∈ {*0*<..<*1*::*real*}›
assumes ‹*real n* ≥ *12*/*ε*^*2* ∗ *ln* (*3*∗*real* (*length xs*) /*δ*)›
shows ‹*measure* (*run-steps xs*) {*ω.* |*estimate ω* − *A*| > *ε*∗*A* } ≤ *δ*›
(**is** ‹*?L* ≤ *?R*›)
proof −
let *?pmf* = ‹*run-steps xs*›
let *?pmf'* = ‹*run-state-pmf* (*FinalState xs*)›
let *?p* = ‹*state-p*›
let *?l* = ‹*real* (*length xs*)›
have *l-gt-0*: ‹*length xs* > *0*› **using** *set-larger-than-n n-gt-0* **by** *auto*
hence *l-ge-1*: ‹*?l* ≥ *1*› **by** *linarith*

have *a*:‹*ln* (*3* ∗ *real* (*length xs*) / *δ*) = − *ln* (*δ* / (*3* ∗ *?l*))›
using *l-ge-1 assms*(*2*) **by** (*subst* (*1 2*) *ln-div*) *auto*

have ‹*exp* (− *real n* / *12* ∗ *1*) ≤ *exp* (− *real n* / *12* ∗ *ε* ^ *2*)›
using *assms*(*1*) **by** (*intro iffD2*[*OF exp-le-cancel-iff*] *mult-left-mono-neg power-le-one*) *auto*
also have ‹. . . ≤ *δ* / (*3* ∗ *?l*)›
using *assms*(*1*−*3*) *l-ge-1* **unfolding** *a* **by** (*subst ln-ge-iff*[*symmetric*]) (*auto simp*:*divide-simps*)
finally have ‹*exp* (− *real n* / *12*) ≤ *δ* / (*3*∗*?l*)› **by** *simp*
hence *b*: ‹*?l* ∗ *exp* (− *real n* / *12*) ≤ *δ* / *3*› **using** *l-gt-0* **by** (*auto simp*:*field-simps*)

have ‹*exp*(−*real n*/*12* ∗ *ε*^*2*) ≤ *δ* / (*3*∗*?l*)›
using *assms*(*1*−*3*) *l-ge-1* **unfolding** *a* **by** (*subst ln-ge-iff*[*symmetric*]) (*auto simp*:*divide-simps*)
also have ‹. . . ≤ *δ* / *3*› **using** *assms*(*1*−*3*) *l-ge-1* **by** (*intro divide-left-mono*) *auto*
finally have *c*: ‹*exp*(−*real n*/*12* ∗ *ε*^*2*) ≤ *δ* / *3*› **by** *simp*

have ‹*exp*(−*real n*/*8* ∗ *ε*^*2*) ≤ *exp*(−*real n*/*12* ∗ *ε*^*2*)› **by** (*intro iffD2*[*OF exp-le-cancel-iff*]) *auto*
also have ‹. . . ≤ *δ*/*3*› **using** *c* **by** *simp*
finally have *d*: ‹*exp*(−*real n*/*8* ∗ *ε*^*2*) ≤ *δ* / *3*› **by** *simp*

have ‹*?L* ≤ *measure ?pmf* {*ω.* |*estimate ω* − *A*| ≥ *ε* ∗ *A*}› **by** (*intro pmf-mono*) *auto*
also have ‹. . . ≤ *measure ?pmf* {*ω.* |*estimate ω* − *A*| ≥ *ε*∗*A* ∧ *?p ω* ≥ *q*} + *measure*

```
?pmf {ω. ?p ω < q}›
    by (intro pmf-add) auto
  also have ‹... ≤ measure ?pmf {ω. (estimate ω ≤ (1−ε) ∗ A ∨ estimate ω ≥ (1+ε) ∗
A) ∧ ?p ω ≥ q} +
    ?l ∗ exp(−real n/12)›
   by (intro pmf-mono add-mono low-p) (auto simp:abs-real-def algebra-simps split:if-split-asm)
  also have ‹... ≤ measure ?pmf {ω. estimate ω ≤ (1−ε) ∗ A ∧ state-p ω ≥ q} +
    measure ?pmf' {ω. estimate ω ≥ (1+ε) ∗ A ∧ state-p ω ≥ q} + δ/3›
    unfolding run-state-pmf.simps by (intro add-mono pmf-add b) auto
  also have ‹... ≤ exp(−real n/8 ∗ ε ^ 2) + exp(−real n/12 ∗ ε ^ 2) + δ / 3›
    using assms(1) by (intro upper-tail-bound add-mono lower-tail-bound) auto
  also have ‹... ≤ δ / 3 +  δ / 3 + δ / 3› by (intro  add-mono d c) auto
  finally show ?thesis by simp
qed

end

lemma deterministic-phase:
  assumes ‹card (run-state-set σ) < n›
  shows ‹run-state-pmf σ = return-pmf (State (run-state-set σ) 1)›
  using assms
proof (induction σ rule:run-state-induct)
  case 1 thus ?case by (simp add:run-steps-def initial-state-def)
next
  case (2 xs x)
  have ‹card (set xs) < n› using 2(2) by (simp add:card-insert-if) presburger
  moreover have ‹bernoulli-pmf 1 = return-pmf True›
    by (intro pmf-eqI) (auto simp:bernoulli-pmf.rep-eq)
  ultimately show ?case using 2(1) by (simp add:step-1-def bind-return-pmf)
next
  case (3 xs x)
  let ?p = ‹run-state-pmf (IntermState xs x)›
  have a: ‹card (run-state-set (IntermState xs x)) < n› using 3(2) by simp
  have b: ‹run-state-pmf (FinalState (xs@[x])) = ?p ≫= step-2›
    by (simp add:run-steps-snoc)
  show ?case
    using 3(2) unfolding b 3(1)[OF a] by (simp add:step-2-def bind-return-pmf Let-def)
qed
```

Theorem 1:

```
theorem correctness:
  fixes ε δ :: real
  assumes ‹ε ∈ {0<..<1}› ‹δ ∈ {0<..<1}›
  assumes ‹real n ≥ 12 / ε² ∗ ln (3 ∗ real (length xs) / δ)›
  shows ‹measure (run-steps xs) {ω. |estimate ω − A| > ε ∗ A} ≤ δ›
proof (cases ‹card (set xs) ≥ n›)
  case True
  show ?thesis by (intro correctness-aux True assms)
next
  case False
  hence ‹run-steps xs = return-pmf (State (set xs) 1)›
    using deterministic-phase[where σ=‹FinalState xs›] by simp
  thus ?thesis using assms(1,2) by (simp add:indicator-def estimate-def not-less)
```

qed

lemma *p-pos*: ‹$\exists$ *M*∈{*0*<..*1*}. *AE* ω *in run-steps xs*. *state-p* $\omega \geq$ *M*›
proof −
have *fin*:‹*finite* (*set-pmf* (*run-steps xs*))›
using *finite-run-state-pmf*[**where** ϱ=‹*FinalState xs*›] **by** *simp*
define *M* **where** ‹*M* = (*MIN* $\sigma \in$ *set-pmf* (*run-steps xs*). *state-p* σ)›
have ‹*M* ∈ *state-p* ‘ *set-pmf* (*run-steps xs*)›
using *fin set-pmf-not-empty* **unfolding** *M-def* **by** (*intro Min-in*) *auto*
also have ‹. . . ⊆ {*0*<..*1*}›
using *state-p-range*[**where** ϱ=‹*FinalState xs*›]
by (*intro image-subsetI*) (*simp add*:*AE-measure-pmf-iff*)
finally have ‹*M* ∈ {*0*<..*1*}› **by** *simp*

moreover have ‹*AE* ω *in run-steps xs*. *state-p* $\omega \geq$ *M*›
using *fin* **unfolding** *AE-measure-pmf-iff M-def* **by** (*intro ballI Min-le*) *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *run-steps-expectation-sing*:
assumes *i*: ‹*i* ∈ *set xs*›
shows ‹*measure-pmf*.*expectation* (*run-steps xs*) ($\lambda\omega$. *of-bool* (*i* ∈ *state-*χ ω) / *state-p* ω) = *1*›
(**is** ‹*?L* = -›)
proof −
have ‹*finite* (*set-pmf* (*run-steps xs*))›
using *finite-run-state-pmf*[**where** ϱ=‹*FinalState xs*›] **by** *simp*
note *int* = *integrable-measure-pmf-finite*[*OF this*]

obtain *M* **where** ∗: ‹*AE* σ *in run-steps xs*. *M* ≤ *state-p* σ› **and** *M-range*: ‹*M* ∈ {*0*<..*1*}›
using *p-pos* **by** *blast*
then have ‹*?L* = ($\int\tau$. ($\prod x$∈{*i*}. *of-bool* (*M* ≤ *state-p* τ) ∗ (*of-bool* (*x* ∈ *state-*χ τ) / *state-p* τ))
∂*run-state-pmf* (*FinalState xs*))›
by (*auto intro*!: *integral-cong-AE*)

also have ‹. . . ≤ *1* ^ *card* {*i*}›
using *M-range i* **by** (*intro run-steps-preserves-expectation-le*′) (*auto simp*:*concave-on-iff*)
finally have *le*: ‹*?L* ≤ *1*› **by** *auto*

have *concave*: ‹*concave-on* {*0*..*1* / *M*} ((−) (*1* / *M* + *1*))›
unfolding *concave-on-iff*
using *M-range* **apply** (*clarsimp simp add*: *field-simps*)
by (*metis combine-common-factor distrib-right linear mult-1-left*)

have ‹*1* / *M* + *1* − *?L* = ($\int\omega$. *1* / *M* + *1* − *of-bool* (*i* ∈ *state-*χ ω) / *state-p* ω ∂*run-steps xs*)›
by (*auto simp*:*int*)
also have ‹. . . = ($\int\tau$. ($\prod x$∈{*i*}. *of-bool* (*M* ≤ *state-p* τ) ∗
(*1* / *M* + *1* − *of-bool* (*x* ∈ *state-*χ τ) / *state-p* τ)) ∂*run-state-pmf* (*FinalState xs*))›
using ∗ **by** (*auto intro*!: *integral-cong-AE*)
also have ‹. . . ≤ (*1* / *M* + *1* − *1*) ^ *card* {*i*}›
using *i M-range*

```
    by (intro run-steps-preserves-expectation-le'[OF - - concave]) (auto simp: field-simps)
  also have ‹... = 1 / M› by auto
  finally have ge:‹ −?L ≤ −1› by auto
  show ?thesis using le ge by auto
qed
```

Subsection A.3:

```
corollary unbiasedness:
  fixes σ :: ‹'a run-state›
  shows ‹measure-pmf.expectation (run-steps xs) estimate = real (card (set xs))›
    (is ‹?L = -›)
proof −
  have ‹finite (set-pmf (run-steps xs))›
    using finite-run-state-pmf[where ϱ=‹FinalState xs›] by simp
  note [simp] = integrable-measure-pmf-finite[OF this]

  have s: ‹AE ω in run-steps xs. state-χ ω ⊆ set xs›
    by (metis run-state-pmf.simps(1) run-state-set.simps(1) state-χ-run-state-pmf)

  have ‹?L = (∫ω. (∑i∈set xs. of-bool (i ∈ state-χ ω)) / state-p ω ∂run-steps xs)›
    unfolding estimate-def state-p-def[symmetric]
    by (auto intro!: integral-cong-AE intro: AE-mp[OF s] simp add: Int-absorb1)

  also have ‹... = (∫ω. (∑i∈set xs. of-bool (i ∈ state-χ ω) / state-p ω) ∂run-steps xs)›
    by (metis (no-types) sum-divide-distrib)
  also have ‹... = (∑i∈set xs. (∫ω. of-bool (i ∈ state-χ ω) / state-p ω ∂run-steps xs))›
    by (auto intro: Bochner-Integration.integral-sum)
  also have ‹... = (∑i∈set xs. 1)›
    using run-steps-expectation-sing by (auto cong:sum.cong)
  finally show ?thesis by auto
qed

end

end

end
```

3 The Original CVM Algorithm

In this section, we verify the algorithm as presented by Chakrabory et al. [1] (replicated, here, in Algorithm 2), with the following caveat:

In the original algorithm the elements are removed with probability $f := \frac{1}{2}$ in the subsampling step. The version verified here allows for any $f \in [\frac{1}{2}, e^{-1/12}]$.

Algorithm 2 Original CVM algorithm.

Input: Stream elements $a_1, \ldots, a_l$, $0 < \varepsilon$, $0 < \delta < 1$, f subsampling param.
Output: An estimate R, s.t., $\mathcal{P}\left(|R - |A|| > \varepsilon |A|\right) \leq \delta$ where $A := \{a_1, \ldots, a_l\}$.

1: $\chi \leftarrow \{\}, p \leftarrow 1, n \geq \left\lceil \frac{12}{\varepsilon^2} \ln(\frac{6l}{\delta}) \right\rceil$
2: **for** $i \leftarrow 1$ to l **do**
3:     $b \overset{\$}{\leftarrow} \mathrm{Ber}(p)$ ▷ insert a_i with probability p (and remove it otherwise)
4:     **if** b **then**
5:         $\chi \leftarrow \chi \cup \{a_i\}$
6:     **else**
7:         $\chi \leftarrow \chi - \{a_i\}$
8:     **if** $|\chi| = n$ **then**
9:         $\chi \overset{\$}{\leftarrow} \mathrm{subsample}(\chi)$ ▷ keep each element of χ indep. with prob. f
10:         $p \leftarrow pf$
11:     **if** $|\chi| = n$ **then**
12:         **return** $\perp$
13: **return** $\frac{|\chi|}{p}$ ▷ estimate cardinality of A

The first step of the proof is identical to the original proof [1], where the above algorithm is approximated by a second algorithm, where lines 11–12 are removed, i.e., the two algorithms behave identically, unless the very improbable event—where the subsampling step fails to remove any elements—occurs. It is possible to show that the total variational distance between the two algorithms is at most $\frac{\delta}{2}$.

In the second step, we verify that the probability that the second algorithm returns an estimate outside of the desired interval is also at most $\frac{\delta}{2}$. This, of course, works by noticing that it is an instance of the abstract algorithm we introduced in Section 2. In combination, we conclude a failure probability of δ for the unmodified version of the algorithm.

On the other hand, the fact that the number of elements in the buffer is at most n can be seen directly from Algorithm 2.

theory *CVM-Original-Algorithm*
  imports *CVM-Abstract-Algorithm*
begin

context
  fixes *f* :: *real*
  fixes *n* :: *nat*
  assumes *f-range*: ‹$f \in \{1/2..exp(-1/12)\}$›
  assumes *n-gt-0*: ‹$n > 0$›
begin

Line 1:

```
definition initial-state :: ‹'a state› where
  ‹initial-state = State {} 1›
```

Lines 3–7:

```
fun step-1 :: ‹'a ⇒ 'a state ⇒ 'a state spmf› where
  ‹step-1 a (State χ p) =
    do {
      b ← bernoulli-pmf p;
      let χ = (if b then χ ∪ {a} else χ − {a});

      return-spmf (State χ p)
    }›

definition subsample :: ‹'a set ⇒ 'a set spmf› where
  ‹subsample χ =
    do {
      keep-in-χ ← prod-pmf χ (λ-. bernoulli-pmf f);
      return-spmf (Set.filter keep-in-χ χ)
    }›
```

Lines 8–10:

```
fun step-2 :: ‹'a state ⇒ 'a state spmf› where
  ‹step-2 (State χ p) =
    do {
      if card χ = n then do {
        χ ← subsample χ;
        return-spmf (State χ (p ∗ f))
      } else
        return-spmf (State χ p)
    }›
```

Lines 11–12:

```
fun step-3 :: ‹'a state ⇒ 'a state spmf› where
  ‹step-3 (State χ p) =
    do {
      if card χ = n
      then fail-spmf
      else return-spmf (State χ p)
    }›
```

Lines 1–12:

```
definition run-steps :: ‹'a list ⇒ 'a state spmf› where
  ‹run-steps xs ≡ foldM-spmf (λx σ. step-1 x σ ⤜ step-2 ⤜ step-3) xs initial-state›
```

Line 13:

```
definition estimate :: ‹'a state ⇒ real› where
  ‹estimate σ = card (state-χ σ) / state-p σ›

definition run-algo :: ‹'a list ⇒ real spmf› where
  ‹run-algo xs = map-spmf estimate (run-steps xs)›
```

schematic-goal *step-1-m-def*: ‹*step-1 x σ = ?x*›
  by (*subst state.collapse*[*symmetric*], *subst step-1.simps*, *rule refl*)

schematic-goal *step-2-m-def*: ‹*step-2 σ = ?x*›
  by (*subst state.collapse*[*symmetric*], *subst step-2.simps*, *rule refl*)

schematic-goal *step-3-m-def*: ‹*step-3 σ = ?x*›
  by (*subst state.collapse*[*symmetric*], *subst step-3.simps*, *rule refl*)

lemma *ord-spmf-remove-step3*:
  ‹*ord-spmf* (=) (*step-1 x σ* ⤜ *step-2* ⤜ *step-3*) (*step-1 x σ* ⤜ *step-2*)›
proof −
  have ‹*ord-spmf* (=) (*step-2 x* ⤜ *step-3*) (*step-2 x*)› **for** *x* :: ‹′*a state*›
  proof −
    have ‹*ord-spmf* (=) (*step-2 x* ⤜ *step-3*) (*step-2 x* ⤜ *return-spmf*)›
      by (*intro bind-spmf-mono*′) (*simp-all add*:*step-3-m-def*)
    thus *?thesis* **by** *simp*
  qed
  thus *?thesis* **unfolding** *bind-spmf-assoc* **by** (*intro bind-spmf-mono*′) *simp-all*
qed

lemma *ord-spmf-run-steps*:
  ‹*ord-spmf* (=) (*run-steps xs*) (*foldM-spmf* (λ*x σ*. *step-1 x σ* ⤜ *step-2*) *xs initial-state*)›
  unfolding *run-steps-def*
proof (*induction xs rule*:*rev-induct*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*snoc x xs*)
  show *?case*
    unfolding *run-steps-def foldM-spmf-snoc*
    by (*intro ord-spmf-remove-step3 bind-spmf-mono*′ *snoc*)
qed

lemma *f-range-simple*: ‹$f \in \{1/2..<1\}$›
proof −
  have ‹$exp\ (-\ 1\ /\ 12) < (1::real)$› **by** (*approximation 5*)
  from *dual-order.strict-trans2*[*OF this*]
  show *?thesis* **using** *f-range* **by** *auto*
qed

Main result:

theorem *correctness*:
  fixes *xs* :: ‹′*a list*›
  assumes ‹$\varepsilon \in \{0<..<1\}$› ‹$\delta \in \{0<..<1\}$›
  assumes ‹$real\ n \geq 12\ /\ \varepsilon^2 * ln\ (6 * real\ (length\ xs)\ /\ \delta)$›
  defines ‹$A \equiv real\ (card\ (set\ xs))$›
  shows ‹$\mathcal{P}(\omega\ in\ run\text{-}algo\ xs.\ fails\text{-}or\text{-}satisfies\ (\lambda R.\ |R - A| > \varepsilon * A)\ \omega) \leq \delta$›
    (**is** ‹*?L* ≤ *?R*›)
proof −
  define *abs-subsample* **where**
    ‹*abs-subsample* χ = *map-pmf* (λω. *Set.filter* ω χ) (*prod-pmf* χ (λ-. *bernoulli-pmf f*))›

```
  for χ :: ‹'a set›

interpret abs:cvm-algo-abstract n f abs-subsample
  rewrites ‹abs.estimate = estimate›
proof −
  show abs:‹cvm-algo-abstract n f abs-subsample›
  proof (unfold-locales, goal-cases)
    case 1 thus ?case by (rule n-gt-0)
  next
    case 2 thus ?case using f-range-simple by auto
  next
    case (3 U x)
    then show ?case unfolding abs-subsample-def by auto
  next
    case (4 g χ S)
    hence fin-U: ‹finite χ› using n-gt-0 card-gt-0-iff by metis
    note conv = Pi-pmf-subset[OF this 4(1)]

    have ‹(∫ω. (∏s∈S. g (s ∈ ω)) ∂abs-subsample χ) =
      (∫ω. (∏s∈S. g (s ∈ χ ∧ ω s)) ∂prod-pmf χ (λ-. bernoulli-pmf f))›
      unfolding abs-subsample-def by (simp cong:prod.cong)
   also have ‹... = (∫ω. (∏s∈S. g (s ∈ χ ∧ ω s)) ∂prod-pmf S (λ-. bernoulli-pmf f))›
      unfolding conv by simp
    also have ‹... = (∏s∈S. (∫ω. g (s ∈ χ ∧ ω) ∂bernoulli-pmf f))›
      using fin-U finite-subset[OF 4(1)]
      by (intro expectation-prod-Pi-pmf integrable-measure-pmf-finite) auto
    also have ‹... = (∏s∈S. (∫ω. g ω ∂bernoulli-pmf f))›
      using 4(1) by (intro prod.cong refl) auto
    finally show ?case by simp
  qed
  show ‹cvm-algo-abstract.estimate = (estimate :: 'a state ⇒ real)›
    unfolding cvm-algo-abstract.estimate-def[OF abs] estimate-def by simp
qed

have a: ‹step-1 σ x = spmf-of-pmf (abs.step-1 σ x)› for σ x
 unfolding step-1-m-def abs.step-1-def Let-def spmf-of-pmf-def by (simp add:map-bind-pmf)

have b: ‹step-2 σ = map-pmf Some (abs.step-2 σ)› for σ
  unfolding step-2-m-def abs.step-2-def subsample-def abs-subsample-def Let-def
  by (simp add:map-bind-pmf bind-pmf-return-spmf)

have c: ‹abs.initial-state = initial-state›
  unfolding initial-state-def abs.initial-state-def by simp

have d: ‹subsample χ = spmf-of-pmf (abs-subsample χ)› for χ
  unfolding subsample-def abs-subsample-def map-pmf-def[symmetric]
  by (simp add:spmf-of-pmf-def map-pmf-comp)

define α :: real where ‹α = f ^ n›

have α-range: ‹α ∈ {0..1}›
  using f-range-simple unfolding α-def by (auto intro:power-le-one)
hence [simp]: ‹|α| ≤ 1› by auto
```

```
  have ‹(∫ x. (if card x = n then 1 else 0) ∂abs-subsample χ) ≤ α› (is ‹?L1 ≤ -›)
    if that': ‹card χ = n› for χ
  proof −
    have fin-U: ‹finite χ› using n-gt-0 that card-gt-0-iff by metis

    have ‹(∏ s∈χ. of-bool (s ∈ x)::real) = of-bool(card x = n)›
      if ‹x ∈ set-pmf (abs-subsample χ)› for x
    proof −
      have x-ran: ‹x ⊆ χ› using that unfolding abs-subsample-def by auto

      have ‹(∏ s∈χ. of-bool (s ∈ x)::real) = of-bool(x = χ)›
        using fin-U x-ran by (induction χ rule:finite-induct) auto
      also have ‹... = of-bool (card x = card χ)›
        using x-ran fin-U card-subset-eq by (intro arg-cong[where f=‹of-bool›]) blast
      also have ‹... = of-bool (card x = n)› using that' by simp
      finally show ?thesis by auto
    qed
    hence ‹?L1 = (∫ x. (∏ s ∈ χ. of-bool(s ∈ x)) ∂abs-subsample χ)›
      by (intro integral-cong-AE AE-pmfI) simp-all
    also have ‹... ≤ (∏ s ∈ χ. (∫ x. of-bool x ∂bernoulli-pmf f))›
      by (intro abs.subsample-inequality that) auto
    also have ‹... = f ^ card χ› using f-range-simple by simp
    also have ‹... = α› unfolding α-def that by simp
    finally show ?thesis by simp
  qed
  hence e: ‹pmf (step-2 σ ⤜ step-3) None ≤ α› for σ :: ‹'a state›
    using α-range unfolding step-2-m-def step-3-m-def d Let-def
    by (simp add:pmf-bind bind-pmf-return-spmf if-distrib if-distribR cong:if-cong)

  have ‹pmf (step-1 x σ ⤜ step-2 ⤜ step-3) None ≤ α› for σ and x :: 'a
  proof−
     have ‹pmf (step-1 x σ ⤜ step-2 ⤜ step-3) None ≤ 0 + (∫ -. α ∂ measure-spmf
(step-1 x σ))›
      unfolding bind-spmf-assoc pmf-bind-spmf-None[where p=‹step-1 x σ›]
        by (intro add-mono integral-mono-AE measure-spmf.integrable-const-bound[where
B=‹1›]
          iffD2[OF AE-measure-spmf-iff] ballI e)
          (simp-all add:pmf-le-1 step-1-m-def map-pmf-def[symmetric] pmf-map vimage-def
Let-def)
    also have ‹... ≤ α› using α-range by (simp add: mult-left-le-one-le weight-spmf-le-1)
    finally show ?thesis by simp
  qed
  hence ‹prob-fail (run-steps xs) ≤ length xs ∗ α›
    unfolding run-steps-def by (intro prob-fail-foldM-spmf-le[where P=‹λ-. True›]) auto
  also have ‹... ≤ δ / 2›
  proof (cases ‹xs = []›)
    case True
    thus ?thesis using assms(2) by auto
  next
    case False
    have ‹δ ≤ 6 ∗ 1› using assms(2) by simp
    also have ‹... ≤ 6 ∗ real (length xs)›
```

using *False* **by** (*intro mult-mono order.refl*) (*cases xs*, *auto*)
finally have [*simp*]: ‹ $\delta \leq 6 * real\ (length\ xs)$› **by** *simp*
have ‹$2 * real\ (length\ xs) * f\ \hat{}\ n \leq 2 * real\ (length\ xs) * exp\ (-1/12)\hat{}n$›
using *f-range* **by** (*intro mult-left-mono power-mono*) *auto*
also have ‹$\ldots = 2 * real\ (length\ xs) * exp\ (-real\ n\ /\ 12)$›
unfolding *exp-of-nat-mult*[*symmetric*] **by** *simp*
also have ‹$\ldots \leq 2 * real\ (length\ xs) * exp\ (-(12\ /\ \varepsilon\ \hat{}\ 2 * ln\ (6 * real\ (length\ xs)\ /\ \delta))/12)$›
using *assms*(*3*) **by** (*intro mult-left-mono iffD2*[*OF exp-le-cancel-iff*] *divide-right-mono*) *auto*
also have ‹$\ldots = 2 * real\ (length\ xs) * exp\ (-ln\ (6 * real\ (length\ xs)\ /\ \delta)\ /\ \varepsilon\hat{}2\ )$›
by *auto*
also have ‹$\ldots \leq 2 * real\ (length\ xs) * exp\ (-ln\ (6 * real\ (length\ xs)\ /\ \delta)\ /\ 1\ )$›
using *assms*(*1*,*2*) *False*
by (*intro mult-left-mono iffD2*[*OF exp-le-cancel-iff*] *divide-left-mono-neg power-le-one*)
(*auto intro*!:*ln-ge-zero simp*:*divide-simps*)
also have ‹$\ldots = 2 * real\ (length\ xs) * exp\ (ln\ (inverse\ (6 * real\ (length\ xs)\ /\ \delta)))$›
using *False assms*(*2*) **by** (*subst ln-inverse*[*symmetric*]) *auto*
also have ‹$\ldots = 2 * real\ (length\ xs)\ /\ (6 * real\ (length\ xs)\ /\ \delta)$›
using *assms*(*1*,*2*) *False* **by** (*subst exp-ln*) *auto*
also have ‹$\ldots = \delta\ /\ 3$› **using** *False assms*(*2*) **by** *auto*
also have ‹$\ldots \leq \delta$› **using** *assms*(*2*) **by** *auto*
finally have ‹$2 * real\ (length\ xs) * f\hat{}n \leq \delta$› **by** *simp*
thus *?thesis* **unfolding** *α-def* **by** *simp*
qed
finally have *f*:‹$prob\text{-}fail\ (run\text{-}steps\ xs) \leq \delta\ /\ 2$› **by** *simp*

have *g*:‹$spmf\text{-}of\text{-}pmf\ (abs.run\text{-}steps\ xs) = foldM\text{-}spmf\ (\lambda x\ \sigma.\ step\text{-}1\ x\ \sigma \gg= step\text{-}2)\ xs\ initial\text{-}state$›
unfolding *abs.run-steps-def foldM-spmf-of-pmf-eq*(*2*)[*symmetric*]
unfolding *spmf-of-pmf-def map-pmf-def c b a*
by (*simp add*:*bind-assoc-pmf bind-spmf-def bind-return-pmf*)

have ‹$?L \leq measure\ (run\text{-}steps\ xs)\ \{None\}\ +$
$measure\ (measure\text{-}spmf\ (run\text{-}steps\ xs))\ \{x.\ |estimate\ x - A| > \varepsilon * A\}$›
unfolding *run-algo-def measure-measure-spmf-conv-measure-pmf measure-map-pmf*
by (*intro pmf-add*) (*auto split*:*option.split-asm*)
also have ‹$\ldots \leq \delta\ /\ 2 + measure\ (measure\text{-}spmf\ (run\text{-}steps\ xs))\ \{x.\ |estimate\ x - A| > \varepsilon * A\}$›
unfolding *measure-pmf-single* **by** (*intro add-mono f order.refl*)
also have ‹$\ldots \leq \delta/2+measure(measure\text{-}spmf\ (spmf\text{-}of\text{-}pmf\ (abs.run\text{-}steps\ xs)))\ \{x.\ |estimate\ x-A|>\varepsilon*A\}$›
using *ord-spmf-eqD-emeasure*[*OF ord-spmf-run-steps*] **unfolding** *measure-spmf.emeasure-eq-measure g*
by (*intro add-mono*) *auto*
also have ‹$\ldots \leq \delta\ /\ 2 + measure\ (abs.run\text{-}steps\ xs)\ \{x.\ |estimate\ x - A| > \varepsilon * A\}$›
using *measure-spmf-map-pmf-Some spmf-of-pmf-def* **by** *auto*
also have ‹$\ldots \leq \delta\ /\ 2 + \delta\ /\ 2$›
using *assms*(*1−3*) **unfolding** *A-def* **by** (*intro add-mono abs.correctness*) *auto*
finally show *?thesis* **by** *simp*
qed

lemma *space-usage*:

```
  ‹AE σ in measure-spmf (run-steps xs). card (state-χ σ) < n ∧ finite (state-χ σ)›
proof (induction xs rule:rev-induct)
  case Nil thus ?case using n-gt-0 by (simp add:run-steps-def initial-state-def)
next
  case (snoc x xs)
  define p1 where ‹p1 = run-steps xs ≫= step-1 x›
  define p2 where ‹p2 = p1 ≫= step-2›
  define p3 where ‹p3 = p2 ≫= step-3›

  have a:‹run-steps (xs@[x]) = p3›
   unfolding run-steps-def p1-def p2-def p3-def foldM-spmf-snoc by (simp add:bind-assoc-pmf)

  have ‹card (state-χ σ) ≤ n ∧ finite (state-χ σ)› if ‹σ ∈ set-spmf p1› for σ
    using snoc that less-imp-le unfolding p1-def
    by (auto simp: step-1-m-def set-bind-spmf set-spmf-bind-pmf Let-def card-insert-if)+

  hence ‹card (state-χ σ) ≤ n ∧ finite (state-χ σ)› if ‹σ ∈ set-spmf p2› for σ
    using that card-filter-mono unfolding p2-def
    by (auto intro!:card-filter-mono simp:step-2-m-def set-bind-spmf set-spmf-bind-pmf
        subsample-def Let-def if-distrib)

  hence ‹card (state-χ σ) < n ∧ finite (state-χ σ)› if ‹σ ∈ set-spmf p3› for σ
    using that unfolding p3-def
    by (auto intro:le-neq-implies-less simp:step-3-m-def set-bind-spmf if-distrib)

  thus ?case unfolding a by simp
qed

end

end
```

4 The New Unbiased Algorithm

In this section, we introduce the new algorithm variant promised in the abstract.

The main change is to replace the subsampling step of the original algorithm, which removes each element of the buffer independently with probability f. Instead, we choose a random nf-subset of the buffer (see Algorithm 3). (This means f, n must be chosen, such that nf is an integer.)

Algorithm 3 New CVM algorithm.

Input: Stream elements $a_1, \ldots, a_l$, $0 < \varepsilon$, $0 < \delta < 1$, f subsampling param.
Output: An estimate R, s.t., $\mathcal{P}\left(|R - |A|| > \varepsilon |A|\right) \leq \delta$ where $A := \{a_1, \ldots, a_l\}$.

$\chi \leftarrow \{\}, p \leftarrow 1, n \geq \left\lceil \frac{12}{\varepsilon^2} \ln(\frac{3l}{\delta}) \right\rceil$
for $i \leftarrow 1$ to l **do**
    $b \xleftarrow{\$} \mathrm{Ber}(p)$ ▷ insert a_i with probability p (and remove it otherwise)
    if b **then**
        $\chi \leftarrow \chi \cup \{a_i\}$
    else
        $\chi \leftarrow \chi - \{a_i\}$
    if $|\chi| = n$ **then**
        $\chi \xleftarrow{\$} \mathrm{subsample}(\chi)$ ▷ Choose a random nf-subset of χ
        $p \leftarrow pf$
return $\frac{|\chi|}{p}$ ▷ estimate cardinality of A

The fact that this still preserves the required inequality for the subsampling operation (Eq. 1) follows from the negative associativity of permutation distributions [2, Th. 10].

(See also our formalization of the concept [3].)

Because the subsampling step always removes elements unconditionally, the second check, whether the subsampling succeeded of the original algorithm is not necessary anymore.

This improves the space usage of the algorithm, because the first reduction argument from Section 3 is now obsolete. Moreover the resulting algorithm is now unbiased, because it is an instance of the abstract algorithm of Section 2.

theory *CVM-New-Unbiased-Algorithm*
  imports
    CVM-Abstract-Algorithm
    Probabilistic-Prime-Tests.Generalized-Primality-Test
    Negative-Association.Negative-Association-Permutation-Distributions
begin

unbundle *no vec-syntax*

context
  fixes f :: *real* **and** n :: *nat*
  assumes *f-range*: ‹$f \in \{1/2..<1\}$› ‹$n * f \in \mathbb{N}$› **and** *n-gt-0*: ‹$n > 0$›
begin

```
definition ‹initial-state = State {} 1› — Setup initial state χ = ∅ and p = 1.
fun subsample where — Subsampling operation: Sample random nf subset.
  ‹subsample χ = pmf-of-set {S. S ⊆ χ ∧ card S = n ∗ f}›

fun step where — Loop body.
  ‹step a (State χ p) = do {
    b ← bernoulli-pmf p;
    let χ = (if b then χ ∪ {a} else χ − {a});

    if card χ = n then do {
      χ ← subsample χ;
      return-pmf (State χ (p ∗ f))
    } else do {
      return-pmf (State χ p)
    }
  }›

fun run-steps where — Iterate loop over stream xs.
  ‹run-steps xs = foldM-pmf step xs initial-state›

fun estimate where
  ‹estimate (State χ p) = card χ / p›

fun run-algo where — Run algorithm and estimate.
  ‹run-algo xs = map-pmf estimate (run-steps xs)›

definition ‹subsample-size = (THE x. real x = n ∗ f)›

declare subsample.simps [simp del]

lemma subsample-size-eq:
  ‹real subsample-size = n ∗ f›
proof −
  obtain a where a-def:‹real a = real n ∗ f› using f-range(2) by (metis Nats-cases)
  show ?thesis
    unfolding subsample-size-def using a-def
    by (rule theI2[where a=‹a›]) (use a-def in auto)
qed

lemma subsample-size:
  ‹subsample-size < n› ‹2 ∗ subsample-size ≥ n›
proof (goal-cases)
  case 1
  have ‹real subsample-size < real n›
    unfolding subsample-size-eq using f-range(1) n-gt-0 by auto
  thus ?case by simp
next
  case 2
  have ‹real n ≤ 2 ∗ real subsample-size›
    using f-range(1) n-gt-0 unfolding subsample-size-eq by auto
  thus ?case by simp
qed
```

lemma *subsample-finite-nonempty*:
  assumes ‹*card U* = *n*›
  shows
    ‹{*T*. *T* ⊆ *U* ∧ *card T* = *subsample-size*} ≠ {}› (**is** ‹*?C* ≠ {}›)
    ‹*finite* {*T*. *T* ⊆ *U* ∧ *card T* = *subsample-size*}›
    ‹*subsample U* = *pmf-of-set* {*T*. *T* ⊆ *U* ∧ *card T* = *subsample-size*}›
    ‹*finite* (*set-pmf* (*subsample U*))›
proof −
  have *fin-U*: ‹*finite U*› **using** *assms subsample-size*
    by (*meson card-gt-0-iff le0 order-le-less-trans order-less-le-trans*)
  have *a*: ‹*card U choose subsample-size* > *0*›
    using *subsample-size assms* **by** (*intro zero-less-binomial*) *auto*
  show *b*:‹*subsample U* = *pmf-of-set ?C*›
    using *subsample-size-eq* **unfolding** *subsample.simps*
    by (*intro arg-cong*[**where** *f*=‹*pmf-of-set*›] *Collect-cong*) *auto*
  with *assms subsample-size* **have** ‹*card ?C* > *0*›
    using *n-subsets*[*OF fin-U*] **by** *simp*
  thus ‹*?C* ≠ {}› ‹*finite ?C*› **using** *card-gt-0-iff* **by** *blast+*
  thus ‹*finite* (*set-pmf* (*subsample U*))› **unfolding** *b* **by** *auto*
qed

lemma *int-prod-subsample-eq-prod-int*:
  fixes *g* :: ‹*bool* ⇒ *real*›
  assumes ‹*card U* = *n*› ‹*S* ⊆ *U*› ‹*range g* ⊆ {*0*..}›
  shows ‹(∫ ω. (∏ *s*∈*S*. *g*(*s* ∈ ω)) ∂*subsample U*) ≤ (∏ *s*∈*S*. (∫ ω. *g* ω ∂*bernoulli-pmf f*))›
(**is** ‹*?L* ≤ *?R*›)
proof −
  define η **where** ‹η ≡ *if g True* ≥ *g False then Fwd else Rev*›

  have *fin-U*: ‹*finite U*› **using** *assms subsample-size*
    by (*meson card-gt-0-iff le0 order-le-less-trans order-less-le-trans*)

  note *subsample* = *subsample-finite-nonempty*[*OF assms*(*1*)]

  note [*simp*] = *integrable-measure-pmf-finite*[*OF subsample*(*4*)]

  let *?C* = ‹{*T*. *T* ⊆ *U* ∧ *card T* = *subsample-size*}›

  have *subssample-size-le-card-U*: ‹*subsample-size* ≤ *card U*›
    using *subsample-size* **unfolding** *assms*(*1*) **by** *simp*

  have ‹*measure-pmf.neg-assoc* (*subsample U*) (λ*s* ω. (*s* ∈ ω)) *U*›
    using *subssample-size-le-card-U* **unfolding** *subsample*
    by (*intro n-subsets-distribution-neg-assoc fin-U*)

  hence *na*: ‹*measure-pmf.neg-assoc* (*subsample U*) (λ*s* ω. (*s* ∈ ω)) *S*›
    using *measure-pmf.neg-assoc-subset*[*OF assms*(*2*)] **by** *auto*

  have *fin-S*: ‹*finite S*› **using** *assms*(*2*) *fin-U finite-subset* **by** *auto*
  note *na-imp-prod-mono* = *has-int-thatD*(*2*)[*OF measure-pmf.neg-assoc-imp-prod-mono*[*OF fin-S na*]]

  have *g-borel*: ‹*g* ∈ *borel-measurable borel*› **by** (*intro borel-measurable-continuous-onI*)

simp
  have *g-mono-aux*: ‹*g x* $\le\ge_\eta$ *g y*› **if** ‹*x* ≤ *y*› **for** *x y*
    unfolding *η-def* **using** *that* **by** *simp* (*smt* (*verit*, *best*))
  have *g-mono*: ‹*monotone* (≤) ($\le\ge_\eta$) *g*›
    by (*intro monotoneI*) (*auto simp*:*dir-le-refl intro*!:*g-mono-aux*)

  have *a*: ‹*map-pmf* (λω. *s* ∈ ω) (*subsample U*) = *bernoulli-pmf f*› **if** ‹*s* ∈ *U*› **for** *s*
  proof −
    have ‹*measure* (*pmf-of-set ?C*) {*x*. *s* ∈ *x*} = *real subsample-size* / *card U*›
      by (*intro n-subsets-prob subssample-size-le-card-U that fin-U*)
    also have ‹. . . = *f*› **unfolding** *subsample-size-eq assms*(*1*) **using** *n-gt-0* **by** *auto*
    finally have ‹*measure* (*pmf-of-set ?C*) {*x*. *s* ∈ *x*} = *f*› **by** *simp*
    thus *?thesis*
      unfolding *subsample* **by** (*intro eq-bernoulli-pmfI*) (*simp add*: *pmf-map vimage-def*)
  qed

  have ‹*?L* ≤ (∏ *s*∈*S*. (∫ ω. *g* (*s* ∈ ω) ∂*subsample U*))›
    by (*intro na-imp-prod-mono*[*OF - g-mono*] *g-borel assms*(*3*)) *auto*
  also have ‹. . . = (∏ *s*∈*S*. (∫ ω. *g* ω ∂*map-pmf* (λω. *s* ∈ ω) (*subsample U*)))› **by** *simp*
  also have ‹. . . = *?R*› **using** *a assms*(*2*) **by** (*intro prod.cong refl*) (*metis in-mono*)
  finally show *?thesis* **.**
qed

schematic-goal *step-n-def*: ‹*step x* σ = *?x*›
  by (*subst state.collapse*[*symmetric*], *subst step.simps*, *rule refl*)

interpretation *abs*: *cvm-algo-abstract n f subsample*
  rewrites ‹*abs.run-steps* = *run-steps*› **and** ‹*abs.estimate* = *estimate*›
proof −
  show *abs*:‹*cvm-algo-abstract n f subsample*›
  proof (*unfold-locales*, *goal-cases*)
    case *1* **thus** *?case* **using** *subsample-size* **by** *auto*
  next
    case *2* **thus** *?case* **using** *f-range* **by** *auto*
  next
    case (*3 U x*) **thus** *?case* **using** *subsample-finite-nonempty*[*OF 3*(*1*)] **by** *simp*
  next
    case (*4 g U S*) **thus** *?case* **by** (*intro int-prod-subsample-eq-prod-int*) *auto*
  qed
  have *a*:‹(λ*x* σ. *cvm-algo-abstract.step-1 x* σ ≫= *cvm-algo-abstract.step-2 n f subsample*)
= *step*›
    unfolding *cvm-algo-abstract.step-1-def*[*OF abs*] *cvm-algo-abstract.step-2-def*[*OF abs*]
step-n-def
    by (*intro ext*) (*simp add*: *bind-assoc-pmf Let-def bind-return-pmf Set.remove-def cong*:*if-cong*)
  have *c*:‹*cvm-algo-abstract.initial-state* = *initial-state*›
    unfolding *cvm-algo-abstract.initial-state-def*[*OF abs*] *initial-state-def* **by** *auto*
  show ‹*cvm-algo-abstract.run-steps n f subsample* = *run-steps*›
    unfolding *cvm-algo-abstract.run-steps-def*[*OF abs*] *run-steps.simps a c* **by** *simp*
  show ‹*cvm-algo-abstract.estimate* = *estimate*›
    unfolding *cvm-algo-abstract.estimate-def*[*OF abs*]
    by (*intro ext*) (*metis estimate.simps state.collapse*)
qed

```
theorem unbiasedness: ‹measure-pmf.expectation (run-algo xs) id = card (set xs)›
  unfolding run-algo.simps integral-map-pmf using abs.unbiasedness by simp

theorem correctness:
  assumes ‹ε ∈ {0<..<1::real}› ‹δ ∈ {0<..<1::real}›
  assumes ‹real n ≥ 12 / ε² ∗ ln (3 ∗ real (length xs) / δ)›
  defines ‹A ≡ real (card (set xs))›
  shows ‹𝒫(R in run-algo xs. |R − A| > ε ∗ A) ≤ δ›
  using assms(3) unfolding A-def using abs.correctness[OF assms(1,2)] by auto

lemma space-usage:
  ‹AE σ in run-steps xs. card (state-χ σ) < n ∧ finite (state-χ σ)›
proof −
  define ϱ where ‹ϱ = FinalState xs›
  have ‹card (state-χ σ) < n + (case ϱ of FinalState - ⇒ 0 | IntermState - - ⇒ 1)›
    if ‹σ ∈ set-pmf (abs.run-state-pmf ϱ)› for σ
    using that
  proof (induction ϱ arbitrary:σ rule:run-state-induct)
    case 1
    then show ?case using n-gt-0 by (simp add:initial-state-def)
  next
    case (2 xs x)
    have ‹card (state-χ τ) < n ∧ finite (state-χ τ)›
      if ‹τ ∈ set-pmf (abs.run-state-pmf (FinalState xs))› for τ
     using 2(1) abs.state-χ-finite[where ϱ=‹FinalState xs›] that by (simp add:AE-measure-pmf-iff)
    thus ?case
     using 2(2) unfolding abs.step-1-def abs.run-state-pmf.simps Let-def map-pmf-def[symmetric]
      by (force simp:card-insert-if remove-def)
  next
    case (3 xs x)
    define p where ‹p = abs.run-state-pmf (IntermState xs x)›

    have a: ‹abs.run-state-pmf (FinalState (xs@[x])) = p ≫= abs.step-2›
      by (simp add:p-def abs.run-steps-snoc del:run-steps.simps)

    have b:‹card χ < card (state-χ τ)›
      if ‹card (state-χ τ) = n› ‹χ ∈ set-pmf (subsample (state-χ τ))› ‹τ ∈ set-pmf p› for
χ τ
    proof −
      from subsample-finite-nonempty[OF that(1)]
      have ‹card χ = subsample-size› using that unfolding subsample-def by auto
      thus ?thesis using subsample-size(1) that by auto
    qed

    have ‹card (state-χ τ) < n ∨ card (state-χ τ) = n› ‹finite (state-χ τ)›
      if ‹τ ∈ set-pmf p› for τ
      using 3(1) abs.state-χ-finite[where ϱ=‹IntermState xs x›] that unfolding p-def
      by (auto simp:AE-measure-pmf-iff less-Suc-eq)
    hence ‹card (state-χ σ) < n›
       using 3(2) unfolding a abs.step-2-def Let-def by (auto intro!:b simp:if-distrib
if-distribR)
    thus ?case by simp
  qed
```

thus *?thesis* **using** *abs.state-χ-finite*[**where** *ϱ*=‹*FinalState xs*›] **unfolding** *ϱ-def*
by (*simp add*:*AE-measure-pmf-iff*)
qed

end

end

References

[1] S. Chakraborty, N. V. Vinodchandran, and K. S. Meel. Distinct elements in streams: An algorithm for the (text) book. In S. Chechik, G. Navarro, E. Rotenberg, and G. Herman, editors, *ESA*, volume 244 of *LIPIcs*, pages 34:1–34:6. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[2] D. P. Dubhashi, V. Priebe, and D. Ranjan. Negative dependence through the fkg inequality. *BRICS Report Series*, 3, 1996.

[3] E. Karayel. Negatively associated random variables. *Archive of Formal Proofs*, January 2025. https://isa-afp.org/entries/Negative_Association.html, Formal proof development.

A Informal Proof

This section includes an informal version of the proof for the tail bounds and unbiasedness of the abstract algorithm (Algorithm 1) for interested readers.

This means we assume the subsample(χ) operation fulfills Eq. 1 and always returns a subset of χ.

Notation: For a finite set S, the probability space of uniformly sampling from the set is denoted by $U(S)$, i.e., for each $s \in S$ we have $\mathcal{P}_{U(S)}(s) = |S|^{-1}$. We write $\mathrm{Ber}(p)$ for the Bernoulli probability space, over the set $\{0, 1\}$, i.e., $P_{\mathrm{Ber}(p)}(\{1\}) = p$. $I(P)$ is the indicator function for a predicate P, i.e., $I(\text{true}) = 1$ and $I(\text{false}) = 0$.

Like in the formalization, we will denote the first five lines of the loop (3–7) as step 1, the last four lines (8–10) as step 2. For the distribution of the state of the algorithm after processing i elements of the sequence, we will write Ω_i. The elements of the probability spaces are pairs composed of a set and the number of subsampling steps, representing χ and p respectively.

For example: $\Omega_0 = U(\{(\emptyset, 1)\})$ is the initial state, $\Omega_1 = U(\{(\{a_1\}, 1)\})$, etc., and Ω_l denotes the final state. We introduce χ and p as random variables defined over such probability spaces Ω, in particular, χ (resp. p) is the projection to the first (resp. second) component.

The state of the algorithm after processing only step 1 of the i-th loop iteration is denoted by Ω'_i. So the sequence of states is represented by the distributions $\Omega_0, \Omega'_1, \Omega_1, \ldots, \Omega'_l, \Omega_l$.

A.1 Loop Invariant

After these preliminaries, we can go to the main proof, whose core is a probabilistic loop invariant for Algorithm 1 that can be verified inductively.

Lemma 1. Let $\varphi : (0, 1] \times \{0, 1\} \to \mathbb{R}_{\geq 0}$ be a function, fulfilling the following conditions:

1. $(1 - \alpha)\varphi(x, 0) + \alpha\varphi(x, 1) \leq \varphi(x/\alpha, 1)$ for all $0 < \alpha < 1$, $0 < x \leq 1$, and
2. $\varphi(x, 0) \leq \varphi(y, 0)$ for all $0 < x < y \leq 1$.

Then for all $k \in \{0, \ldots, l\}$, $S \subseteq \{a_1, .., a_k\}$, $\Omega \in \{\Omega_k, \Omega'_k\}$:

$$\mathbb{E}_{\Omega}\left[\prod_{s \in S} \varphi(p, I(s \in \chi))\right] \leq \varphi(1, 1)^{|S|}$$

Proof. We show the result using induction over k. Note that we show the statement for arbitrary S, i.e., the induction statements are:

$$P(k) \quad :\leftrightarrow \quad \left(\forall S \subseteq \{a_1, .., a_k\}.\ \mathbb{E}_{\Omega_k}\left[\prod_{s \in S} \varphi(p, I(s \in \chi))\right] \leq \varphi(1, 1)^{|S|}\right)$$

$$Q(k) \quad :\leftrightarrow \quad \left(\forall S \subseteq \{a_1, .., a_k\}.\ \mathbb{E}_{\Omega'_k}\left[\prod_{s \in S} \varphi(p, I(s \in \chi))\right] \leq \varphi(1, 1)^{|S|}\right)$$

and we will show $P(0), Q(1), P(1), Q(2), P(2), \ldots, Q(l), P(l)$ successively.

Induction start $P(0)$**:**
We have $S \subseteq \emptyset$, and hence

$$\mathbb{E}_{\Omega_0}\left[\prod_{s\in S}\varphi(p, I(s\in\chi))\right] = \mathbb{E}_{\Omega_0}[1] = 1 \leq \varphi(1,1)^0.$$

Induction step $P(k) \to Q(k+1)$**:**
Let $S \subseteq \{a_1, \ldots, a_{k+1}\}$ and define $S' := S - \{a_{k+1}\}$. Note that Ω'_{k+1} can be constructed from Ω_k as a compound distribution, where a_{k+1} is included in the buffer, with the probability p, which is itself a random variable over the space Ω_k.
In particular, for example:

$$\mathcal{P}_{\Omega'_{k+1}}(P(\chi, p)) = \int_{\Omega_k}\int_{\mathrm{Ber}(p(\omega))} P(\chi(\omega) - \{a_{k+1}\} \cup f(\tau), p(\omega))\, d\tau\, d\omega$$

for all predicates P where we define $f(1) = \{a_{k+1}\}$ and $f(0) = \emptyset$.
We distinguish the two cases $a_{k+1} \in S$ and $a_{k+1} \notin S$. If $a_{k+1} \in S$:

$$\begin{aligned}
& \mathbb{E}_{\Omega'_{k+1}}\left[\prod_{s\in S}\varphi(p, I(s\in\chi))\right] \\
= \;& \textstyle\int_{\Omega_k}\left(\prod_{s\in S'}\varphi(p, I(s\in\chi))\right)\int_{\mathrm{Ber}(p(\omega))}\varphi(p,\tau)\, d\tau\, d\omega \\
= \;& \textstyle\int_{\Omega_k}\left(\prod_{s\in S'}\varphi(p, I(s\in\chi))\right)((1-p)\varphi(p,0) + p\varphi(p,1))\, d\omega \\
\underset{\text{Cond 1}}{\leq} \;& \textstyle\int_{\Omega_k}\left(\prod_{s\in S'}\varphi(p, I(s\in\chi))\right)\varphi(1,1)\, d\omega \\
\underset{\text{IH}}{\leq} \;& \varphi(1,1)^{|S'|}\varphi(1,1) = \varphi(1,1)^{|S|}
\end{aligned}$$

If $a_{k+1} \notin S$ then $S' = S$ and:

$$\mathbb{E}_{\Omega'_{k+1}}\left[\prod_{s\in S}\varphi(p, I(s\in\chi))\right] = \int_{\Omega_k}\prod_{s\in S}\varphi(p, I(s\in\chi))\, d\omega \leq_{\mathrm{IH}} \varphi(1,1)^{|S'|} = \varphi(1,1)^{|S|}$$

Induction step $Q(k+1) \to P(k+1)$**:**
Let $S \subseteq \{a_1, \ldots, a_{k+1}\}$.
Let us again note that Ω_{k+1} is a compound distribution over Ω'_{k+1}. In general, for all predicates P:

$$\mathcal{P}_{\Omega_{k+1}}(P(\chi, p)) =$$
$$\int_{\Omega'_{k+1}} I(|\chi(\omega)| < n)P(\chi(\omega), p(\omega)) + I(|\chi(\omega)| = n)\int_{\mathrm{subsample}(\chi(\omega))} P(\tau, fp(\omega))\, d\tau\, d\omega.$$

With this we can can now verify the induction step:

$$
\begin{aligned}
& \mathbb{E}_{\Omega_{k+1}}\left[\prod_{s\in S}\varphi(p,I(s\in\chi))\right] \\
= \; & \int_{\Omega'_{k+1}} I(|\chi|<n)\prod_{s\in S}\varphi(p,I(s\in\chi))\,d\omega \\
+ \; & \int_{\Omega'_{k+1}} I(|\chi|=n)\prod_{s\in S\setminus\chi(\omega)}\varphi(pf,0)\int_{\text{subsample}(\chi)}\prod_{s\in S\cap\chi}\varphi(pf,I(s\in\tau))d\tau\,d\omega \\
\leq \; & \int_{\Omega'_{k+1}} I(|\chi|<n)\prod_{s\in S}\varphi(p,I(s\in\chi))\,d\omega && \text{Eq. 1} \\
+ \; & \int_{\Omega'_{k+1}} I(|\chi|=n)\prod_{s\in S\setminus\chi(\omega)}\varphi(pf,0)\prod_{s\in S\cap\chi}\int_{\text{Ber(f)}}\varphi(pf,\tau)d\tau\,d\omega \\
\leq \; & \int_{\Omega'_{k+1}} I(|\chi|<n)\prod_{s\in S}\varphi(p,I(s\in\chi))\,d\omega && \text{Cond 2} \\
+ \; & \int_{\Omega'_{k+1}} I(|\chi|=n)\prod_{s\in S\setminus\chi(\omega)}\varphi(p,0)\prod_{s\in S\cap\chi}((1-f)\varphi(pf,0)+f\varphi(pf,1))\,d\omega \\
\leq \; & \int_{\Omega'_{k+1}} I(|\chi|<n)\prod_{s\in S}\varphi(p,I(s\in\chi))\,d\omega && \text{Cond 1} \\
+ \; & \int_{\Omega'_{k+1}} I(|\chi|=n)\prod_{s\in S\setminus\chi(\omega)}\varphi(p,0)\prod_{s\in S\cap\chi}\varphi(p,1)\,d\omega \\
= \; & \int_{\Omega'_{k+1}} I(|\chi|<n)\prod_{s\in S}\varphi(p,I(s\in\chi))\,d\omega \\
+ \; & \int_{\Omega'_{k+1}} I(|\chi|=n)\prod_{s\in S}\varphi(p,I(s\in\chi))\,d\omega \\
= \; & \mathbb{E}_{\Omega'_{k+1}}\left[\prod_{s\in S}\varphi(p,I(s\in\chi))\right]\leq\varphi(1,1)^{|S|} && \text{IH}
\end{aligned}
$$

□

A corollary and more practical version of the previous lemma is:

Lemma 2. Let $q\leq 1$ and $h:[0,q^{-1}]\to\mathbb{R}_{\geq 0}$ be concave then for all $k\in\{0,\ldots,l\}$, $S\subseteq\{a_1,..,a_k\}$, $\Omega\in\{\Omega_k,\Omega'_k\}$:

$$\mathbb{E}_{\Omega}\left[\prod_{s\in S}I(p>q)h(p^{-1}I(s\in\chi))\right]\leq h(1)^{|S|}$$

Proof. Follows from Lemma 1 for $\varphi(p,\tau):=I(p>q)h(\tau p^{-1})$. We just need to check Conditions 1 and 2. Indeed,

$$
\begin{aligned}
(1-\alpha)\varphi(x,0)+\alpha\varphi(x,1) &= (1-\alpha)I(x>q)h(0)+\alpha I(x>q)h(x^{-1}) \\
&\leq I(x>q)h(\alpha x^{-1})\leq I(x>q\alpha)h(\alpha x^{-1})=\varphi(x/\alpha,1)
\end{aligned}
$$

for $0<\alpha<1$ and $0<x\leq 1$, where we used that $x>q$ implies $x>q\alpha$; and

$$\varphi(x,0)=I(x>q)h(0)\leq I(y>q)h(0)=\varphi(y,0)$$

for $0<x<y\leq 1$, where we used that $x>q$ implies $y>q$. □

It should be noted that this is a probabilistic recurrence relation, but the main innovation is that we establish a relation, with respect to general classes of functions of the state variables.

A.2 Concentration

Let us now see how we can obtain concentration bounds using Lemma 2, i.e., that the result of the algorithm is concentrated around the cardinality of $A = \{a_1, \dots, a_l\}$. This will be done using the Cramér–Chernoff method for the probability that the estimate is above $(1+\varepsilon)|A|$ (resp. below $(1-\varepsilon)|A|$) assuming p is not too small and a tail estimate for p being too small.

It should be noted that concentration is trivial, if $|A| < n$, i.e., if we never need to do sub-sampling, so we assume $|A| \geq n$.

Define $q := n/(4|A|)$ and notice that $q \leq \frac{1}{4}$.

Let us start with the upper tail bound:

Lemma 3. For any $\Omega \in \{\Omega_0, \dots, \Omega_l\} \cup \{\Omega'_1, \dots, \Omega'_l\}$ and $0 < \varepsilon \leq 1$:

$$L := \mathcal{P}_\Omega \left(p^{-1}|\chi| \geq (1+\varepsilon)|A| \wedge p \geq q\right) \leq \exp\left(-\frac{n}{12}\varepsilon^2\right)$$

Proof. By assumption there exists a k such that $\Omega \in \{\Omega_k, \Omega'_k\}$. Let $A' = A \cap \{a_1, \dots, a_k\}$. Moreover, we define:

$$t := q\ln(1+\varepsilon)$$
$$h(x) := 1 + qx(e^{t/q} - 1)$$

To get a tail estimate, we use the Cramér–Chernoff method:

$$\begin{aligned}
L \quad &\underset{t>0}{\leq} \quad \mathcal{P}_\Omega\left(\exp(tp^{-1}|\chi|) \geq \exp(t(1+\varepsilon)|A|) \wedge p \geq q\right) \\
&\leq \quad \mathcal{P}_\Omega\left(I(p \geq q)\exp(tp^{-1}|\chi|) \geq \exp(t(1+\varepsilon)|A|)\right) \\
&\underset{\text{Markov}}{\leq} \quad \exp(-t(1+\varepsilon)|A|)\,\mathbb{E}_\Omega\left[I(p \geq q)\exp(tp^{-1}|\chi|)\right] \\
&\leq \quad \exp(-t(1+\varepsilon)|A|)\,\mathbb{E}_\Omega\left[\prod_{s \in A'} I(p \geq q)\exp(tp^{-1}I(s \in \chi))\right] \\
&\leq \quad \exp(-t(1+\varepsilon)|A|)\,\mathbb{E}_\Omega\left[\prod_{s \in A'} I(p \geq q)h(p^{-1}I(s \in \chi))\right] \\
&\underset{\text{Le. 2}}{\leq} \quad \exp(-t(1+\varepsilon)|A|)h(1)^{|A'|} \\
&\underset{h(1)\geq 1}{\leq} \quad (\exp(\ln(h(1)) - t(1+\varepsilon)))^{|A|}
\end{aligned}$$

So we just need to show that (using $|A| = \frac{n}{4q}$):

$$\ln(h(1)) - t(1+\varepsilon) \leq \frac{-q\varepsilon^2}{3}$$

The latter can be established by analyzing the function

$$f(\varepsilon) := -\ln(1+q\varepsilon) + q\ln(1+\varepsilon)(1+\varepsilon) - \frac{q\varepsilon^2}{3} = -\ln(h(1)) + t(1+\varepsilon) - \frac{q\varepsilon^2}{3}.$$

For which it is easy to check $f(0) = 0$ and the derivative with respect to ε is non-negative in the range $0 \leq q \leq 1/4$ and $0 < \varepsilon \leq 1$, i.e., $f(\varepsilon) \geq 0$. ☐

Using the previous result we can also estimate bounds for p becoming too small:

Lemma 4.
$$\mathcal{P}_{\Omega_l}(p < q) \le l \exp\left(-\frac{n}{12}\right)$$

Proof. We will use a similar strategy as in the Bad_2 bound from the original CVM paper [1]. Let j be maximal, s.t., $q \le f^j$. Hence $f^{j+1} < q$ and:

$$f^j \le 2ff^j < 2q = \frac{n}{2|A|}. \tag{2}$$

First, we bound the probability of jumping from $p = f^j$ to $p = f^{j+1}$ at a specific point in the algorithm, e.g., while processing k stream elements. It can only happen if $|\chi| = n$, $p = f^j$ in Ω'_k. Then

$$\begin{aligned}
\mathcal{P}_{\Omega'_k}(|\chi| \ge n \wedge p = f^j) &\le \mathcal{P}(p^{-1}|\chi| \ge f^{-j}n \wedge p \ge q) \\
&\underset{\text{Eq. 2}}{\le} \mathcal{P}(p^{-1}|\chi| \ge 2|A| \wedge p \ge q) \\
&\underset{\text{Le. 3}}{\le} \exp(-n/12)
\end{aligned}$$

The probability that this happens ever in the entire process is then at most l times the above which proves the lemma. ☐

Lemma 5. Let $0 < \varepsilon < 1$ then:

$$L := \mathcal{P}_{\Omega_l}(p^{-1}|\chi| \le (1-\varepsilon)|A| \wedge p \ge q) \le \exp\left(-\frac{n}{8}\varepsilon^2\right)$$

Proof. Let us define

$$\begin{aligned}
t &:= q\ln(1-\varepsilon) < 0 \\
h(x) &:= 1 + qx(e^{t/q} - 1)
\end{aligned}$$

Note that $h(x) \ge 0$ for $0 \le x \le q^{-1}$ (can be checked by verifying it is true for $h(0)$ and $h(q^{-1})$ and the fact that the function is affine.)
With these definitions we again follow the Cramér–Chernoff method:

$$\begin{aligned}
L &\underset{t<0}{=} \mathcal{P}_{\Omega_l}\left(\exp(tp^{-1}|\chi|) \ge \exp(t(1-\varepsilon)|A|) \wedge p \ge q\right) \\
&\le \mathcal{P}_{\Omega_l}\left(I(p \ge q)\exp(tp^{-1}|\chi|) \ge \exp(t(1-\varepsilon)|A|) \wedge p > q\right) \\
&\underset{\text{Markov}}{\le} \exp(-t(1-\varepsilon)|A|)\,\mathbb{E}_\Omega\left[I(p \ge q)\exp(tp^{-1}|\chi|)\right] \\
&= \exp(-t(1-\varepsilon)|A|)\,\mathbb{E}_\Omega\left[\prod_{s\in A} I(p \ge q)\exp(tp^{-1}I(s \in \chi))\right] \\
&\le \exp(-t(1-\varepsilon)|A|)\,\mathbb{E}_\Omega\left[\prod_{s\in A} I(p \ge q)h(p^{-1}I(s \in \chi))\right] \\
&\underset{\text{Le. 2}}{\le} \exp(-t(1-\varepsilon)|A|)(h(1))^{|A|} \\
&= \exp(\ln(h(1)) - t(1-\varepsilon))^{|A|}
\end{aligned}$$

Substituting t and h and using $|A| = \frac{n}{4q}$, we can see that the lemma is true if

$$f(\varepsilon) := q\ln(1-\varepsilon)(1-\varepsilon) - \ln(1-q\varepsilon) - \frac{q}{2}\varepsilon^2 = t(1-\varepsilon) - \ln(h(1)) - \frac{q}{2}\varepsilon^2$$

is non-negative for $0 \le q \le \frac{1}{4}$ and $0 < \varepsilon < 1$. This can be verified by checking that $f(0) = 0$ and that the derivative with respect to ε is non-negative. □

We can now establish the concentration result:

Theorem 1. *Let* $0 < \varepsilon < 1$ *and* $0 < \delta < 1$ *and* $n \ge \frac{12}{\varepsilon^2}\ln\left(\frac{3l}{\delta}\right)$ *then:*

$$L = \mathcal{P}_{\Omega_l}\left(|p^{-1}|\chi| - |A|| \ge \varepsilon|A|\right) \le \delta$$

Proof. Note that the theorem is trivial if $|A| < n$. If not:

$$\begin{aligned} L &\le \mathcal{P}_{\Omega_l}\left(|p^{-1}|\chi| \le (1-\varepsilon)|A| \wedge p \ge q\right) \\ &+ \mathcal{P}_{\Omega_l}\left(|p^{-1}|\chi| \ge (1+\varepsilon)|A| \wedge p \ge q\right) + \mathcal{P}_{\Omega_l}\left(p < q\right) \\ &\underset{Le.\ 3\text{–}5}{\le} \exp\left(-\frac{n}{8}\varepsilon^2\right) + \exp\left(-\frac{n}{12}\varepsilon^2\right) + l\exp\left(-\frac{n}{12}\right) \\ &\le \frac{\delta}{3} + \frac{\delta}{3} + \frac{\delta}{3} \end{aligned}$$

□

A.3 Unbiasedness

Let M be large enough such that $p^{-1} \le M$ a.s. (e.g., we can choose $M = f^{-l}$). Then we can derive from Lemma 2 using $h(x) = x$ and $h(x) = M + 1 - x$ that for all $s \in A$:

$$\begin{aligned} \mathbb{E}_{\Omega_l}[p^{-1}I(s \in \chi)] &= \mathbb{E}_{\Omega_l}[I(p \ge M^{-1})p^{-1}I(s \in \chi)] \le 1 \\ \mathbb{E}_{\Omega_l}[M + 1 - p^{-1}I(s \in \chi)] &= \mathbb{E}_{\Omega_l}[I(p \ge M^{-1})(M + 1 - p^{-1}I(s \in \chi))] \le M \end{aligned}$$

which implies $\mathbb{E}_{\Omega_l}[p^{-1}I(s \in \chi)] = 1$. By linearity of expectation we conclude

$$\mathbb{E}_{\Omega_l}[p^{-1}|\chi|] = \sum_{s \in A} \mathbb{E}_{\Omega_l}[p^{-1}I(s \in \chi)] = |A|.$$