Approximate Model Counting

Yong Kiam Tan and Jiong Yang

April 20, 2024

Abstract

Approximate model counting is the task of approximating the number of solutions to an input formula. This entry formalizes ApproxMC, an algorithm due to Chakraborty et al. [1] with a probably approximately correct (PAC) guarantee, i.e., ApproxMC returns a multiplicative $(1 + \varepsilon)$-factor approximation of the model count with probability at least $1-\delta$, where $\varepsilon > 0$ and $0 < \delta \leq 1$. The algorithmic specification is further refined to a verified certificate checker that can be used to validate the results of untrusted ApproxMC implementations (assuming access to trusted randomness).

Contents

1 Preliminary probability/UHF lemmas

This section proves some simplified/specialized forms of lemmas that will be used in the algorithm's analysis later.

theory *ApproxMCPreliminaries* **imports**
Frequency-Moments.Probability-Ext
Concentration-Inequalities.Bienaymes-Identity
Concentration-Inequalities.Paley-Zygmund-Inequality
begin

lemma *card-inter-sum-indicat-real*:
assumes *finite A*
shows *card* $(A \cap B) =$ *sum* (*indicat-real B*) *A*
⟨*proof*⟩

lemma *card-dom-ran*:
assumes *finite D*
shows *card* $\{w.$ *dom* $w = D \wedge$ *ran* $w \subseteq R\} =$ *card R* ^ *card D*
⟨*proof*⟩

lemma *finite-set-pmf-expectation-sum*:
fixes $f :: 'a \Rightarrow 'c \Rightarrow 'b$::{*banach*, *second-countable-topology*}
assumes *finite* (*set-pmf A*)
shows *measure-pmf.expectation A* ($\lambda x.$ *sum* (f x) T) =
($\sum i{\in}T.$ *measure-pmf.expectation A* ($\lambda x.$ f x i))
⟨*proof*⟩

lemma (**in** *prob-space*) *k-universal-prob-unif*:
assumes *k-universal k H D R*
assumes $w \in D$ $\alpha \in R$
shows *prob* $\{s \in$ *space M*. H w $s = \alpha\} = 1$ / *card R*
⟨*proof*⟩

lemma *k-universal-expectation-eq*:
assumes *p*: *finite* (*set-pmf p*)
assumes *ind*: *prob-space.k-universal p k H D R*
assumes *S*: *finite S* $S \subseteq D$
assumes *a*: $\alpha \in R$
shows
prob-space.expectation p
($\lambda s.$ *real* (*card* ($S \cap \{w.$ H w $s = \alpha\}$))) =
real (*card S*) / *card R*
⟨*proof*⟩

lemma (**in** *prob-space*) *two-universal-indep-var*:
assumes *k-universal 2 H D R*
assumes $w \in D$ $w' \in D$ $w \neq w'$

shows *indep-var*
borel
($\lambda x.$ *indicat-real* $\{w.\ H\ w\ x = \alpha\}$ w)
borel
($\lambda x.$ *indicat-real* $\{w.\ H\ w\ x = \alpha\}$ w')
$\langle proof \rangle$

lemma *two-universal-variance-bound*:
assumes p: *finite* (*set-pmf* p)
assumes *ind*: *prob-space.k-universal* (*measure-pmf* p) *2* H D R
assumes S: *finite* S $S \subseteq D$
assumes a: $\alpha \in R$
shows
measure-pmf.variance p
($\lambda s.$ *real* (*card* ($S \cap \{w.\ H\ w\ s = \alpha\}$))) $\leq$
measure-pmf.expectation p
($\lambda s.$ *real* (*card* ($S \cap \{w.\ H\ w\ s = \alpha\}$)))
$\langle proof \rangle$

lemma (**in** *prob-space*) *k-universal-mono*:
assumes $k' \leq k$
assumes*k-universal* k H D R
shows*k-universal* k' H D R
$\langle proof \rangle$

lemma *finite-set-pmf-expectation-add*:
assumes *finite* (*set-pmf* S)
shows *measure-pmf.expectation* S ($\lambda x.$ (($f\ x$)::*real*) $+$ $g\ x$) $=$
measure-pmf.expectation S f $+$ *measure-pmf.expectation* S g
$\langle proof \rangle$

lemma *finite-set-pmf-expectation-add-const*:
assumes *finite* (*set-pmf* S)
shows *measure-pmf.expectation* S ($\lambda x.$ (($f\ x$)::*real*) $+$ g) $=$
measure-pmf.expectation S f $+$ g
$\langle proof \rangle$

lemma *finite-set-pmf-expectation-diff*:
assumes *finite* (*set-pmf* S)
shows *measure-pmf.expectation* S ($\lambda x.$ (($f\ x$)::*real*) $-$ $g\ x$) $=$
measure-pmf.expectation S f $-$ *measure-pmf.expectation* S g
$\langle proof \rangle$

lemma *spec-paley-zygmund-inequality*:
assumes *fin*: *finite* (*set-pmf* p)
assumes *Zpos*: $\bigwedge z.\ Z\ z \geq 0$
assumes t: $\vartheta \leq 1$

shows
$(measure\text{-}pmf.variance\ p\ Z + (1-\vartheta)\hat{}2 * (measure\text{-}pmf.expectation\ p\ Z)\hat{}2) *$
$measure\text{-}pmf.prob\ p\ \{z.\ Z\ z > \vartheta * measure\text{-}pmf.expectation\ p\ Z\}$
$\geq$
$(1-\vartheta)\hat{}2 * (measure\text{-}pmf.expectation\ p\ Z)\hat{}2$
⟨*proof*⟩

lemma *spec-chebyshev-inequality*:
assumes *fin*: *finite* (*set-pmf p*)
assumes *pvar*: $measure\text{-}pmf.variance\ p\ Y > 0$
assumes *k*: $k > 0$
shows
measure-pmf.prob p
$\{y.\ (Y\ y - measure\text{-}pmf.expectation\ p\ Y)\hat{}2 \geq$
$k\hat{}2 * measure\text{-}pmf.variance\ p\ Y\} \leq 1\ /\ k\hat{}2$
⟨*proof*⟩

end

2 Random XORs

The goal of this section is to prove that, for a randomly sampled XOR X from a set of variables V:

1. the probability of an assignment w satisfying X is $\frac{1}{2}$;
2. for any distinct assignments w, w' the probability of both satisfying X is equal to $\frac{1}{4}$ (2-wise independence); and
3. for any distinct assignments w, w', w'' the probability of all three satisfying X is equal to $\frac{1}{8}$ (3-wise independence).

theory *RandomXOR* **imports**
ApproxMCPreliminaries
Universal-Hash-Families.Universal-Hash-Families-More-Product-PMF
Monad-Normalisation.Monad-Normalisation
begin

A random XOR constraint is modeled as a random subset of variables and a randomly chosen RHS bit.

definition *random-xor* :: $'a\ set \Rightarrow ('a\ set \times bool)\ pmf$
where *random-xor V* =
pair-pmf (*pmf-of-set* (*Pow V*)) (*bernoulli-pmf* (*1/2*))

lemma *pmf-of-set-Pow-fin-map*:
assumes *V*:*finite V*
shows *pmf-of-set* (*Pow V*) =
map-pmf ($\lambda b.\ \{x \in V.\ b\ x = Some\ True\}$)

```
  (Pi-pmf V def (λ-. map-pmf Some (bernoulli-pmf (1 / 2))))
⟨proof⟩
```

```
lemma random-xor-from-bits:
  assumes V:finite V
  shows random-xor V =
    pair-pmf
      (map-pmf (λb. {x ∈ V. b x = Some True})
        (Pi-pmf V def (λ-. map-pmf Some (bernoulli-pmf (1/2)))))
      (bernoulli-pmf (1/2))
  ⟨proof⟩
```

```
fun satisfies-xor :: ('a set × bool) ⇒ 'a set ⇒ bool
  where satisfies-xor (x,b) ω =
    even (card (ω ∩ x) + of-bool b)
```

```
lemma satisfies-xor-inter:
  shows satisfies-xor ( ω ∩ x ,b) ω = satisfies-xor (x,b) ω
  ⟨proof⟩
```

```
lemma prob-bernoulli-bind-pmf:
  assumes 0 ≤ p p ≤ 1
  assumes finite E
  shows measure-pmf.prob
   (bernoulli-pmf p ≫= x) E =
  p ∗ (measure-pmf.prob (x True) E) +
  (1 − p) ∗ (measure-pmf.prob (x False) E)
  ⟨proof⟩
```

```
lemma set-pmf-random-xor:
  assumes V: finite V
  shows set-pmf (random-xor V) = (Pow V) × UNIV
  ⟨proof⟩
```

```
lemma pmf-of-set-prod:
  assumes P ≠ {} Q ≠ {}
  assumes finite P finite Q
   shows pmf-of-set (P × Q) = pair-pmf (pmf-of-set P) (pmf-of-set
Q)
  ⟨proof⟩
```

```
lemma random-xor-pmf-of-set:
  assumes V:finite V
  shows random-xor V  = pmf-of-set ((Pow V) × UNIV)
  ⟨proof⟩
```

lemma *prob-random-xor-with-set-pmf*:
assumes *V*: *finite V*
shows *prob-space.prob* (*random-xor V*) {*c*. *P c*} =
prob-space.prob (*random-xor V*) {*c*. *fst c* ⊆ *V* ∧ *P c*}
⟨*proof*⟩

lemma *prob-set-parity*:
assumes *measure-pmf.prob M*
{*c*. *P c*} = *q*
shows *measure-pmf.prob M*
{*c*. *P c* = *b*} = (*if b then q else 1* − *q*)
⟨*proof*⟩

lemma *satisfies-random-xor*:
assumes *V*: *finite V*
shows *prob-space.prob* (*random-xor V*)
{*c*. *satisfies-xor c ω*} = *1* / *2*
⟨*proof*⟩

lemma *satisfies-random-xor-parity*:
assumes *V*: *finite V*
shows *prob-space.prob* (*random-xor V*)
{*c*. *satisfies-xor c ω* = *b*} = *1* / *2*
⟨*proof*⟩

2.1 Independence properties of random XORs

lemma *pmf-of-set-powerset-split*:
assumes *S* ⊆ *V finite V*
shows
map-pmf (λ(*x*,*y*). *x* ∪ *y*)
(*pmf-of-set* (*Pow S* × *Pow* (*V* − *S*))) =
pmf-of-set (*Pow V*)
⟨*proof*⟩

lemma *pmf-of-set-Pow-sing*:
shows*pmf-of-set* (*Pow* {*x*}) =
bernoulli-pmf (*1* / *2*) ≫=
(λ*b*. *return-pmf* (*if b then* {*x*} *else* {}))
⟨*proof*⟩

lemma *pmf-of-set-sing-coin-flip*:
assumes *finite V*
shows *pmf-of-set* (*Pow* {*x*} × *Pow V*) =
map-pmf (λ(*r*,*c*). (*if c then* {*x*} *else* {}, *r*)) (*random-xor V*)
⟨*proof*⟩
including *monad-normalisation*
⟨*proof*⟩

lemma *measure-pmf-prob-dependent-product-bound-eq*:
assumes *countable A* $\bigwedge$*i. countable* (*B i*)
assumes $\bigwedge$*a. a* ∈ *A* ⟹ *measure-pmf.prob N* (*B a*) = *r*
shows *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) =
measure-pmf.prob M A ∗ *r*
⟨*proof*⟩

lemma *measure-pmf-prob-dependent-product-bound-eq′*:
assumes *countable* (*A* ∩ *set-pmf M*) $\bigwedge$*i. countable* (*B i* ∩ *set-pmf N*)
assumes $\bigwedge$*a. a* ∈ *A* ∩ *set-pmf M* ⟹ *measure-pmf.prob N* (*B a* ∩ *set-pmf N*) = *r*
shows *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) = *measure-pmf.prob M A* ∗ *r*
⟨*proof*⟩

lemma *single-var-parity-coin-flip*:
assumes *x* ∈ ω *finite* ω
assumes *finite a x* ∉ *a*
shows *measure-pmf.prob* (*pmf-of-set* (*Pow* {*x*}))
{*y. even* (*card* ((*a* ∪ *y*) ∩ ω)) = *b*} = *1/2*
⟨*proof*⟩

lemma *prob-pmf-of-set-nonempty-parity*:
assumes *V*: *finite V*
assumes *x* ∈ ω ω ⊆ *V*
assumes $\bigwedge$*c. c* ∈ *E* ⟷ *c* − {*x*} ∈ *E*
shows *prob-space.prob* (*pmf-of-set* (*Pow V*))
(*E* ∩ {*c. even* (*card* (*c* ∩ ω)) = *b*}) =
1 / *2* ∗ *prob-space.prob* (*pmf-of-set* (*Pow* (*V* − {*x*}))) *E*
⟨*proof*⟩

lemma *prob-random-xor-split*:
assumes *V*: *finite V*
shows *prob-space.prob* (*random-xor V*) *E* =
1 / *2* ∗ *prob-space.prob* (*pmf-of-set* (*Pow V*)) {*e.* (*e*,*True*) ∈ *E*} +
1 / *2* ∗ *prob-space.prob* (*pmf-of-set* (*Pow V*)) {*e.* (*e*,*False*) ∈ *E*}
⟨*proof*⟩

lemma *prob-random-xor-nonempty-parity*:
assumes *V*: *finite V*
assumes ω: *x* ∈ ω ω ⊆ *V*
assumes *E*: $\bigwedge$*c. c* ∈ *E* ⟷ (*fst c* − {*x*},*snd c*) ∈ *E*
shows *prob-space.prob* (*random-xor V*)
(*E* ∩ {*c. satisfies-xor c* ω = *b*}) =

1 / 2 $*$ *prob-space.prob* (*random-xor* (V − {x})) E
⟨*proof*⟩

lemma *pair-satisfies-random-xor-parity-1*:
assumes V:*finite* V
assumes x: $x \notin \omega$ $x \in \omega'$
assumes ω: $\omega \subseteq V$ $\omega' \subseteq V$
shows *prob-space.prob* (*random-xor* V)
{c. *satisfies-xor* c ω = b ∧ *satisfies-xor* c ω' = b'} = *1 / 4*
⟨*proof*⟩

lemma *pair-satisfies-random-xor-parity*:
assumes V:*finite* V
assumes ω: $\omega \neq \omega'$ $\omega \subseteq V$ $\omega' \subseteq V$
shows *prob-space.prob* (*random-xor* V)
{c. *satisfies-xor* c ω = b ∧ *satisfies-xor* c ω' = b'} = *1 / 4*
⟨*proof*⟩

lemma *prob-pmf-of-set-nonempty-parity-UNIV*:
assumes *finite* V
assumes $x \in \omega$ $\omega \subseteq V$
shows *prob-space.prob* (*pmf-of-set* (*Pow* V))
{c. *even* (*card* ($c \cap \omega$)) = b} = *1 / 2*
⟨*proof*⟩

lemma *prob-Pow-split*:
assumes $\omega \subseteq V$ *finite* V
shows *prob-space.prob* (*pmf-of-set* (*Pow* V))
{x. P ($\omega \cap x$) ∧ Q (($V - \omega$) ∩ x)} =
prob-space.prob (*pmf-of-set* (*Pow* ω))
{x. P x} $*$
prob-space.prob (*pmf-of-set* (*Pow* ($V - \omega$)))
{x. Q x}
⟨*proof*⟩

lemma *disjoint-prob-pmf-of-set-nonempty*:
assumes ω: $x \in \omega$ $\omega \subseteq V$
assumes ω': $x' \in \omega'$ $\omega' \subseteq V$
assumes $\omega \cap \omega' = \{\}$
assumes V: *finite* V
shows *prob-space.prob* (*pmf-of-set* (*Pow* V))
{c. *even* (*card* ($\omega \cap c$)) = b ∧ *even* (*card* ($\omega' \cap c$)) = b'} = *1 / 4*
⟨*proof*⟩

lemma *measure-pmf-prob-product-finite-set-pmf*:
assumes *finite* (*set-pmf* M) *finite* (*set-pmf* N)
shows *measure-pmf.prob* (*pair-pmf* M N) ($A \times B$) =
measure-pmf.prob M A $*$ *measure-pmf.prob* N B

⟨*proof*⟩

lemma *prob-random-xor-split-space*:
  assumes $\omega \subseteq V$ *finite* V
  shows *prob-space.prob* (*random-xor* V)
    $\{(x,b).\ P\ (\omega \cap x)\ b \wedge Q\ ((V - \omega) \cap x)\} =$
  prob-space.prob (*random-xor* ω)
    $\{(x,b).\ P\ x\ b\} *$
  prob-space.prob (*pmf-of-set* (*Pow* $(V - \omega)$))
    $\{x.\ Q\ x\}$
⟨*proof*⟩
    including *monad-normalisation*
    ⟨*proof*⟩

lemma *three-disjoint-prob-random-xor-nonempty*:
  assumes ω: $\omega \neq \{\}\ \omega \subseteq V$
  assumes ω': $\omega' \neq \{\}\ \omega' \subseteq V$
  assumes I: $I \subseteq V$
  assumes *int*: $I \cap \omega = \{\}\ I \cap \omega' = \{\}\ \omega \cap \omega' = \{\}$
  assumes V: *finite* V
  shows *prob-space.prob* (*random-xor* V)
    $\{c.$ *satisfies-xor* $c\ I = b \wedge$
      even (*card* ($\omega \cap$ *fst* c)) $= b' \wedge$
      even (*card* ($\omega' \cap$ *fst* c)) $= b''\} = 1\ /\ 8$
⟨*proof*⟩

lemma *three-disjoint-prob-pmf-of-set-nonempty*:
  assumes ω: $x \in \omega\ \omega \subseteq V$
  assumes ω': $x' \in \omega'\ \omega' \subseteq V$
  assumes ω'': $x'' \in \omega''\ \omega'' \subseteq V$
  assumes *int*: $\omega \cap \omega' = \{\}\ \omega' \cap \omega'' = \{\}\ \omega'' \cap \omega = \{\}$
  assumes V: *finite* V
  shows *prob-space.prob* (*pmf-of-set* (*Pow* V))
    $\{c.$ *even* (*card* ($\omega \cap c$)) $= b \wedge$ *even* (*card* ($\omega' \cap c$)) $= b' \wedge$ *even* (*card* ($\omega'' \cap c$)) $= b''\} = 1\ /\ 8$
⟨*proof*⟩

lemma *four-disjoint-prob-random-xor-nonempty*:
  assumes ω: $\omega \neq \{\}\ \omega \subseteq V$
  assumes ω': $\omega' \neq \{\}\ \omega' \subseteq V$
  assumes ω'': $\omega'' \neq \{\}\ \omega'' \subseteq V$
  assumes I: $I \subseteq V$
  assumes *int*: $I \cap \omega = \{\}\ I \cap \omega' = \{\}\ I \cap \omega'' = \{\}$
   $\omega \cap \omega' = \{\}\ \omega' \cap \omega'' = \{\}\ \omega'' \cap \omega = \{\}$
  assumes V: *finite* V
  shows *prob-space.prob* (*random-xor* V)
    $\{c.$ *satisfies-xor* $c\ I = b0 \wedge$
      even (*card* ($\omega \cap$ *fst* c)) $= b \wedge$

even (*card* ($\omega' \cap$ *fst* *c*)) = b' ∧
even (*card* ($\omega'' \cap$ *fst* *c*)) = b''} = *1* / *16*
⟨*proof*⟩

lemma *three-satisfies-random-xor-parity-1*:
assumes *V*:*finite* *V*
assumes ω: $\omega \subseteq V$ $\omega' \subseteq V$ $\omega'' \subseteq V$
assumes *x*: $x \notin \omega$ $x \notin \omega'$ $x \in \omega''$
assumes *d*: $\omega \neq \omega'$
shows *prob-space.prob* (*random-xor* *V*)
{*c*.
satisfies-xor *c* ω = *b* ∧
satisfies-xor *c* ω' = b' ∧
satisfies-xor *c* ω'' = b''} = *1* / *8*
⟨*proof*⟩

lemma *split-boolean-eq*:
shows(*A* ⟷ *B*) = (*b* ⟷ *I*) ∧
(*B* ⟷ *C*) = (b' ⟷ *I*) ∧
(*C* ⟷ *A*) = (b'' ⟷ *I*)
⟷
I = *odd*(*of-bool* *b* + *of-bool* b' + *of-bool* b'') ∧
(*A* = *True* ∧
B = (b'=b'') ∧
C = (*b* = b') ∨
A = *False* ∧
B = ($b' \neq b''$) ∧
C = ($b \neq b'$))
⟨*proof*⟩

lemma *three-satisfies-random-xor-parity*:
assumes *V*:*finite* *V*
assumes ω:
$\omega \neq \omega'$ $\omega \neq \omega''$ $\omega' \neq \omega''$
$\omega \subseteq V$ $\omega' \subseteq V$ $\omega'' \subseteq V$
shows *prob-space.prob* (*random-xor* *V*)
{*c*. *satisfies-xor* *c* ω = *b* ∧
satisfies-xor *c* ω' = b' ∧
satisfies-xor *c* ω'' = b''} = *1* / *8*
⟨*proof*⟩

2.2 Independence for repeated XORs

We can lift the previous result to a list of independent sampled XORs.

definition *random-xors* :: *'a set* ⇒ *nat* ⇒
(*nat* ⇀ *'a set* × *bool*) *pmf*
where *random-xors* *V* *n* =
Pi-pmf {..<(*n*::*nat*)} *None*

(λ-. *map-pmf Some* (*random-xor V*))

lemma *random-xors-set*:
assumes *V*:*finite V*
shows
PiE-dflt {..<*n*} *None*
(*set-pmf* ∘ (λ-. *map-pmf Some* (*random-xor V*))) =
{*xors*. *dom xors* = {..<*n*} ∧
ran xors ⊆ (*Pow V*) × *UNIV*} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *random-xors-eq*:
assumes *V*:*finite V*
shows*random-xors V n* =
pmf-of-set
{*xors*. *dom xors* = {..<*n*} ∧ *ran xors* ⊆ (*Pow V*) × *UNIV*}
⟨*proof*⟩

definition *xor-hash* ::
('*a* ⇀ *bool*) ⇒
(*nat* ⇀ ('*a set* × *bool*)) ⇒
(*nat* ⇀ *bool*)
where *xor-hash* ω *xors* =
(*map-option*
(λ*xor*. *satisfies-xor xor* {*x*. ω *x* = *Some True*}) ∘ *xors*)

lemma *finite-map-set-nonempty*:
assumes *R* ≠ {}
shows
{*xors*.
dom xors = *D* ∧ *ran xors* ⊆ *R*} ≠ {}
⟨*proof*⟩

lemma *random-xors-set-pmf*:
assumes *V*: *finite V*
shows
set-pmf (*random-xors V n*) =
{*xors*. *dom xors* = {..<*n*} ∧
ran xors ⊆ (*Pow V*) × *UNIV*}
⟨*proof*⟩

lemma *finite-random-xors-set-pmf*:
assumes *V*: *finite V*
shows
finite (*set-pmf* (*random-xors V n*))
⟨*proof*⟩

lemma *map-eq-1*:
  assumes *dom f* = *dom g*
  assumes $\bigwedge x.\ x \in$ *dom f* $\Longrightarrow$ *the* (*f x*) = *the* (*g x*)
  shows *f* = *g*
  ⟨*proof*⟩

lemma *xor-hash-eq-iff*:
  assumes *dom* α = {..<*n*}
  shows*xor-hash* ω *x* = α $\longleftrightarrow$
  (*dom x* = {..<*n*} $\wedge$
  ($\forall$ *i*. *i* < *n* $\longrightarrow$
    ($\exists$ *xor*. *x i* = *Some xor* $\wedge$
    satisfies-xor xor {*x*. ω *x* = *Some True*} = *the* (α *i*))
  ))
⟨*proof*⟩

lemma *xor-hash-eq-PiE-dflt*:
  assumes *dom* α = {..<*n*}
  shows
    {*xors*. *xor-hash* ω *xors* = α} =
    PiE-dflt {..<*n*} *None*
      (λ*i*. *Some* ‘
        {*xor*. *satisfies-xor xor* {*x*. ω *x* = *Some True*} = *the* (α *i*)})
⟨*proof*⟩

lemma *prob-random-xors-xor-hash*:
  assumes *V*: *finite V*
  assumes α: *dom* α = {..<*n*}
  shows
  measure-pmf.prob (*random-xors V n*)
    {*xors*. *xor-hash* ω *xors* = α} = *1* / *2* ^ *n*
⟨*proof*⟩

lemma *PiE-dflt-inter*:
  shows*PiE-dflt A dflt B* $\cap$ *PiE-dflt A dflt B'* =
    PiE-dflt A dflt (λ*b*. *B b* $\cap$ *B' b*)
  ⟨*proof*⟩

lemma *random-xors-xor-hash-pair*:
  assumes *V*: *finite V*
  assumes α: *dom* α = {..<*n*}
  assumes α': *dom* α' = {..<*n*}
  assumes ω: *dom* ω = *V*
  assumes ω': *dom* ω' = *V*
  assumes *neq*: $\omega \neq \omega'$
  shows
  measure-pmf.prob (*random-xors V n*)
    {*xors*. *xor-hash* ω *xors* = α $\wedge$ *xor-hash* ω' *xors* = α'} =
    1 / *4* ^ *n*

⟨*proof*⟩

lemma *random-xors-xor-hash-three*:
  assumes *V*: *finite V*
  assumes α: *dom* α = {..<*n*}
  assumes α': *dom* α' = {..<*n*}
  assumes α'': *dom* α'' = {..<*n*}
  assumes ω: *dom* ω = *V*
  assumes ω': *dom* ω' = *V*
  assumes ω': *dom* ω'' = *V*
  assumes *neq*: $\omega \neq \omega'$ $\omega' \neq \omega''$ $\omega'' \neq \omega$
  shows
  measure-pmf.prob (*random-xors V n*)
    {*xors*.
      xor-hash ω *xors* = α
      $\wedge$ *xor-hash* ω' *xors* = α'
      $\wedge$ *xor-hash* ω'' *xors* = α''} =
    1 / *8* ^ *n*
⟨*proof*⟩

end

3 Random XOR hash family

This section defines a hash family based on random XORs and proves that this hash family is 3-universal.

theory *RandomXORHashFamily* **imports**
  RandomXOR
begin

lemma *finite-dom*:
  assumes *finite V*
  shows *finite* {*w* :: *'a* ⇀ *bool*. *dom w* = *V*}
⟨*proof*⟩

lemma *xor-hash-eq-dom*:
  assumes *xor-hash* ω *xors* = α
  shows *dom xors* = *dom* α
  ⟨*proof*⟩

lemma *prob-random-xors-xor-hash-indicat-real*:
  assumes *V*: *finite V*
  shows
  measure-pmf.prob (*random-xors V n*)
    {*xors*. *xor-hash* ω *xors* = α} =
     indicat-real {α::*nat* ⇀ *bool*. *dom* α = {*0*..<*n*}} α /
     real (*card* {α::*nat* ⇀ *bool*. *dom* α = {*0*..<*n*}})
⟨*proof*⟩

lemma *xor-hash-family-uniform*:
  assumes V: *finite* V
  assumes *dom* $\omega = V$
  shows *prob-space.uniform-on*
    (*random-xors* V n)
    (*xor-hash* i) $\{\alpha.\ dom\ \alpha = \{0..<n\}\}$
  ⟨*proof*⟩

lemma *random-xors-xor-hash-pair-indicat*:
  assumes V: *finite* V
  assumes ω: *dom* $\omega = V$
  assumes ω': *dom* $\omega' = V$
  assumes *neq*: $\omega \neq \omega'$
  shows
  measure-pmf.prob (*random-xors* V n)
    $\{$*xors*.
      xor-hash ω *xors* $= \alpha \wedge$ *xor-hash* ω' *xors* $= \alpha'\} =$
  (*measure-pmf.prob* (*random-xors* V n)
    $\{$*xors*.
      xor-hash ω *xors* $= \alpha\} *$
   measure-pmf.prob (*random-xors* V n)
    $\{$*xors*.
      xor-hash ω' *xors* $= \alpha'\}$)
⟨*proof*⟩

lemma *prod-3-expand*:
  assumes $a \neq b$ $b \neq c$ $c \neq a$
  shows$(\prod \omega \in \{a,\ b,\ c\}.\ f\ \omega) = f\ a * (f\ b * f\ c)$
  ⟨*proof*⟩

lemma *random-xors-xor-hash-three-indicat*:
  assumes V: *finite* V
  assumes ω: *dom* $\omega = V$
  assumes ω': *dom* $\omega' = V$
  assumes ω'': *dom* $\omega'' = V$
  assumes *neq*: $\omega \neq \omega'$ $\omega' \neq \omega''$ $\omega'' \neq \omega$
  shows
  measure-pmf.prob (*random-xors* V n)
    $\{$*xors*.
      xor-hash ω *xors* $= \alpha$
    $\wedge$ *xor-hash* ω' *xors* $= \alpha'$
    $\wedge$ *xor-hash* ω'' *xors* $= \alpha''\} =$
  (*measure-pmf.prob* (*random-xors* V n)
    $\{$*xors*.
      xor-hash ω *xors* $= \alpha\} *$
   measure-pmf.prob (*random-xors* V n)
    $\{$*xors*.
      xor-hash ω' *xors* $= \alpha'\} *$

measure-pmf.prob (*random-xors* *V* *n*)
{*xors*.
xor-hash ω'' *xors* = α''})
⟨*proof*⟩

lemma *xor-hash-3-indep*:
assumes *V*: *finite V*
assumes *J*: *card J* ≤ *3 J* ⊆ {α. *dom* α = *V*}
shows
measure-pmf.prob (*random-xors V n*)
{*xors*. ∀ω∈*J*. *xor-hash* ω *xors* = *f* ω} =
(∏ω∈*J*.
measure-pmf.prob (*random-xors V n*)
{*xors*. *xor-hash* ω *xors* = *f* ω})
⟨*proof*⟩

lemma *xor-hash-3-wise-indep*:
assumes *finite V*
shows *prob-space.k-wise-indep-vars*
(*random-xors V n*) *3*
(λ-. *Universal-Hash-Families-More-Independent-Families.discrete*)
xor-hash
{α. *dom* α = *V*}
⟨*proof*⟩

theorem *xor-hash-family-3-universal*:
assumes *finite V*
shows*prob-space.k-universal*
(*random-xors V n*) *3 xor-hash*
{α::'*a* ⇀ *bool*. *dom* α = *V*}
{α::*nat* ⇀ *bool*. *dom* α = {*0*..<*n*}}
⟨*proof*⟩

corollary *xor-hash-family-2-universal*:
assumes *finite V*
shows*prob-space.k-universal*
(*random-xors V n*) *2 xor-hash*
{α::'*a* ⇀ *bool*. *dom* α = *V*}
{α::*nat* ⇀ *bool*. *dom* α = {*0*..<*n*}}
⟨*proof*⟩

end

4 ApproxMCCore definitions

This section defines the ApproxMCCore locale and various failure events to be used in its probabilistic analysis. The definitions closely follow Section 4.2 of Chakraborty et al. [1]. Some nonprobabilistic properties of the events are proved, most notably, the event inclusions of Lemma 3 [1]. Note that "events" here refer to subsets of hash functions.

theory *ApproxMCCore* **imports**
  ApproxMCPreliminaries
begin

type-synonym *'a assg* = *'a* ⇀ *bool*

definition *restr* :: *'a set* ⇒ (*'a* ⇒ *bool*) ⇒ *'a assg*
  where *restr S w* = (λ*x. if x* ∈ *S then Some* (*w x*) *else None*)

lemma *restrict-eq-mono*:
  assumes *x* ⊆ *y*
  assumes *f* |‘ *y* = *g* |‘ *y*
  shows *f* |‘ *x* = *g* |‘ *x*
  ⟨*proof*⟩

definition *proj* :: *'a set* ⇒ (*'a* ⇒ *bool*) *set* ⇒ *'a assg set*
  where *proj S W* = *restr S* ‘ *W*

lemma *card-proj*:
  assumes *finite S*
  shows *finite* (*proj S W*) *card* (*proj S W*) ≤ *2* ^ *card S*
⟨*proof*⟩

lemma *proj-mono*:
  assumes *x* ⊆ *y*
  shows *proj w x* ⊆ *proj w y*
  ⟨*proof*⟩

definition *aslice* :: *nat* ⇒ *nat assg* ⇒ *nat assg*
  where *aslice i a* = *a* |‘ {..<*i*}

lemma *aslice-eq*:
  assumes *i* ≥ *n*
  assumes *dom a* = {..<*n*}
  shows *aslice i a* = *aslice n a*
  ⟨*proof*⟩

definition *hslice* :: *nat* ⇒
  (*'a assg* ⇒ *nat assg*) ⇒ (*'a assg* ⇒ *nat assg*)
  where *hslice i h* = *aslice i* ∘ *h*

```
locale ApproxMCCore =
  fixes W :: ('a ⇒ bool) set
  fixes S :: 'a set
  fixes ε :: real
  fixes α :: nat assg
  fixes thresh :: nat
  assumes α: dom α = {0..<card S − 1}
  assumes ε: ε > 0
  assumes thresh:
    thresh > 4
    card (proj S W) ≥ thresh
  assumes S: finite S
begin

lemma finite-proj-S:
  shows finite (proj S W)
  ⟨proof⟩

definition μ :: nat ⇒ real
  where μ i = card (proj S W) / 2 ^ i


definition card-slice ::
  ('a assg ⇒ nat assg) ⇒
  nat ⇒ nat
  where card-slice h i =
    card (proj S W ∩ {w. hslice i h w = aslice i α})

lemma card-slice-anti-mono:
  assumes i ≤ j
  shows card-slice h j ≤ card-slice h i
⟨proof⟩

lemma hslice-eq:
  assumes n ≤ i
  assumes ⋀w. dom (h w) = {..<n}
  shows hslice i h = hslice n h
  ⟨proof⟩

lemma card-slice-lim:
  assumes card S − 1 ≤ i
  assumes ⋀w. dom (h w) = {..<(card S − 1)}
  shows card-slice h i = card-slice h (card S − 1)
  ⟨proof⟩


definition T :: nat ⇒
  ('a assg ⇒ nat assg) set
```

where $T\ i = \{h.\ \mathit{card\text{-}slice}\ h\ i < \mathit{thresh}\}$

lemma *T-mono*:
assumes $i \leq j$
shows $T\ i \subseteq T\ j$
$\langle \mathit{proof} \rangle$

lemma *μ-anti-mono*:
assumes $i \leq j$
shows $\mu\ i \geq \mu\ j$
$\langle \mathit{proof} \rangle$

lemma *card-proj-witnesses*:
$\mathit{card}\ (\mathit{proj}\ S\ W) > 0$
$\langle \mathit{proof} \rangle$

lemma *μ-strict-anti-mono*:
assumes $i < j$
shows $\mu\ i > \mu\ j$
$\langle \mathit{proof} \rangle$

lemma *μ-gt-zero*:
shows $\mu\ i > 0$
$\langle \mathit{proof} \rangle$

definition $L :: \mathit{nat} \Rightarrow$
$('a\ \mathit{assg} \Rightarrow \mathit{nat}\ \mathit{assg})\ \mathit{set}$
where
$L\ i =$
$\{h.\ \mathit{real}\ (\mathit{card\text{-}slice}\ h\ i) < \mu\ i\ /\ (1 + \varepsilon)\}$

definition $U :: \mathit{nat} \Rightarrow$
$('a\ \mathit{assg} \Rightarrow \mathit{nat}\ \mathit{assg})\ \mathit{set}$
where
$U\ i =$
$\{h.\ \mathit{real}\ (\mathit{card\text{-}slice}\ h\ i) \geq \mu\ i * (1 + \varepsilon\ /\ (1 + \varepsilon))\}$

definition *approxcore* ::
$('a\ \mathit{assg} \Rightarrow \mathit{nat}\ \mathit{assg}) \Rightarrow$
$\mathit{nat} \times \mathit{nat}$
where
$\mathit{approxcore}\ h =$
$(\mathit{case}\ \mathit{List.find}$
$(\lambda i.\ h \in T\ i)\ [1..<\mathit{card}\ S]\ \mathit{of}$
$\mathit{None} \Rightarrow (2\ \hat{}\ \mathit{card}\ S,\ 1)$
$\mid \mathit{Some}\ m \Rightarrow$
$(2\ \hat{}\ m,\ \mathit{card\text{-}slice}\ h\ m))$

definition *approxcore-fail* ::
 $('a\ assg \Rightarrow nat\ assg)\ set$
 where *approxcore-fail* =
 $\{h.$
 $let\ (cells,sols) = approxcore\ h\ in$
 $cells * sols \notin$
 $\{\ card\ (proj\ S\ W)\ /\ (1 + \varepsilon)\ ..$
 $(1 + \varepsilon::real) * card\ (proj\ S\ W)\}$
 $\}$

lemma *T0-empty*:
 shows $T\ 0 = \{\}$
 ⟨*proof*⟩

lemma *L0-empty*:
 shows $L\ 0 = \{\}$
⟨*proof*⟩

lemma *U0-empty*:
 shows $U\ 0 = \{\}$
⟨*proof*⟩

lemma *real-divide-pos-left*:
 assumes $(0::real) < a$
 assumes $a * b < c$
 shows $b < c\ /\ a$
 ⟨*proof*⟩

lemma *real-divide-pos-right*:
 assumes $a > (0::real)$
 assumes $b < a * c$
 shows $b\ /\ a < c$
 ⟨*proof*⟩

lemma *failure-imp*:
 shows *approxcore-fail* $\subseteq$
 $(\bigcup i\in\{1..<card\ S\}.$
 $(T\ i - T\ (i-1)) \cap (L\ i \cup U\ i)) \cup$
 $-T\ (card\ S - 1)$
⟨*proof*⟩

lemma *smallest-nat-exists*:
 assumes $P\ i\ \neg P\ (0::nat)$
 obtains m **where** $m \le i\ P\ m\ \neg P\ (m-1)$
 ⟨*proof*⟩

lemma *mstar-non-zero*:
  shows $\neg\ \mu\ 0 * (1 + \varepsilon\ /\ (1 + \varepsilon)) \leq thresh$
⟨*proof*⟩

lemma *real-div-less*:
  assumes $c > 0$
  assumes $a \leq b * (c{::}nat)$
  shows $real\ a\ /\ real\ c \leq b$
  ⟨*proof*⟩

lemma *mstar-exists*:
  obtains m **where**
    $\mu\ (m - 1) * (1 + \varepsilon\ /\ (1 + \varepsilon)) > thresh$
    $\mu\ m * (1 + \varepsilon\ /\ (1 + \varepsilon)) \leq thresh$
    $m \leq card\ S - 1$
⟨*proof*⟩

definition *mstar* :: *nat*
  where $mstar = (@m.$
    $\mu\ (m - 1) * (1 + \varepsilon\ /\ (1 + \varepsilon)) > thresh\ \wedge$
    $\mu\ m * (1 + \varepsilon\ /\ (1 + \varepsilon)) \leq thresh\ \wedge$
    $m \leq card\ S - 1)$

lemma *mstar-prop*:
  shows
    $\mu\ (mstar - 1) * (1 + \varepsilon\ /\ (1 + \varepsilon)) > thresh$
    $\mu\ mstar * (1 + \varepsilon\ /\ (1 + \varepsilon)) \leq thresh$
    $mstar \leq card\ S - 1$
  ⟨*proof*⟩

lemma *O1-lem*:
  assumes $i \leq m$
  shows $(T\ i - T\ (i-1)) \cap (L\ i \cup U\ i) \subseteq T\ m$
  ⟨*proof*⟩

lemma *O1*:
  shows $(\bigcup i{\in}\{1..mstar-3\}.$
    $(T\ i - T\ (i-1)) \cap (L\ i \cup U\ i)) \subseteq T\ (mstar-3)$
  ⟨*proof*⟩

lemma *T-anti-mono-neg*:
  assumes $i \leq j$
  shows $-\ T\ j \subseteq -\ T\ i$
  ⟨*proof*⟩

lemma *O2-lem*:

assumes $mstar < i$
shows $(T\ i - T\ (i-1)) \cap (L\ i \cup U\ i) \subseteq -T\ mstar$
⟨*proof*⟩

lemma *O2*:
shows $(\bigcup i \in \{mstar..<card\ S\}.$
$(T\ i - T\ (i-1)) \cap (L\ i \cup U\ i)) \cup$
$-T\ (card\ S - 1) \subseteq L\ mstar \cup U\ mstar$
⟨*proof*⟩

lemma *O3*:
assumes $i \leq mstar - 1$
shows $(T\ i - T\ (i-1)) \cap (L\ i \cup U\ i) \subseteq L\ i$
⟨*proof*⟩

lemma *union-split-lem*:
assumes x: $x \in (\bigcup i \in \{1..<n::nat\}.\ P\ i)$
shows $x \in (\bigcup i \in \{1..m-3\}.\ P\ i) \cup$
$P\ (m-2) \cup$
$P\ (m-1) \cup$
$(\bigcup i \in \{m..<n\}.\ P\ i)$
⟨*proof*⟩

lemma *union-split*:
$(\bigcup i \in \{1..<n::nat\}.\ P\ i) \subseteq$
$(\bigcup i \in \{1..m-3\}.\ P\ i) \cup$
$P\ (m-2) \cup$
$P\ (m-1) \cup$
$(\bigcup i \in \{m..<n\}.\ P\ i)$
⟨*proof*⟩

lemma *failure-bound*:
shows *approxcore-fail* $\subseteq$
$T\ (mstar-3) \cup$
$L\ (mstar-2) \cup$
$L\ (mstar-1) \cup$
$(L\ mstar \cup U\ mstar)$
⟨*proof*⟩

end

end

5 ApproxMCCore analysis

This section analyzes ApproxMCCore with respect to a universal hash family. The proof follows Lemmas 1 and 2 from Chakraborty et al. [1].

theory *ApproxMCCoreAnalysis* **imports**
ApproxMCCore
begin

definition *Hslice* :: *nat* ⇒
('*a assg* ⇒ '*b* ⇒ *nat assg*) ⇒ ('*a assg* ⇒ '*b* ⇒ *nat assg*)
where *Hslice i H* = (λ*w s. aslice i* (*H w s*))

context *prob-space*
begin

lemma *indep-vars-prefix*:
assumes *indep-vars* (λ-. *count-space UNIV*) *H J*
shows *indep-vars* (λ-. *count-space UNIV*) (*Hslice i H*) *J*
⟨*proof*⟩

lemma *assg-nonempty-dom*:
shows
(λ*x. if x* < *i then Some True else None*) ∈
{α::*nat assg. dom* α = {*0*..<*i*}}
⟨*proof*⟩

lemma *card-dom-ran-nat-assg*:
shows *card* {α::*nat assg. dom* α = {*0*..<*n*}} = *2*^*n*
⟨*proof*⟩

lemma *card-nat-assg-le*:
assumes *i* ≤ *n*
shows *card* {α::*nat assg. dom* α = {*0*..<*n*}} =
2^(*n*−*i*) ∗ *card* {α::*nat assg. dom* α = {*0*..<*i*}}
⟨*proof*⟩

lemma *empty-nat-assg-slice-notin*:
assumes *i* ≤ *n*
assumes *dom* β ≠ {*0*..<*i*}
shows{α::*nat assg. dom* α = {*0*..<*n*} ∧ *aslice i* α = β} = {}
⟨*proof*⟩

lemma *restrict-map-dom*:
shows α |' *dom* α = α
⟨*proof*⟩

lemma *aslice-refl*:
assumes *dom* α = {..<*i*}
shows *aslice i* α = α
⟨*proof*⟩

lemma *bij-betw-with-inverse*:

assumes f ‘ $A \subseteq B$
assumes $\bigwedge x.\ x \in A \Longrightarrow g\ (f\ x) = x$
assumes g ‘ $B \subseteq A$
assumes $\bigwedge x.\ x \in B \Longrightarrow f\ (g\ x) = x$
shows *bij-betw f A B*
⟨*proof*⟩

lemma *card-nat-assg-slice*:
assumes $i \leq n$
assumes $dom\ \beta = \{0..<i\}$
shows$card\ \{\alpha{::}nat\ assg.\ dom\ \alpha = \{0..<n\} \wedge aslice\ i\ \alpha = \beta\} =$
$2\ \hat{}\ (n-i)$
⟨*proof*⟩

lemma *finite-dom*:
assumes *finite V*
shows $finite\ \{w :: 'a \rightharpoonup bool.\ dom\ w = V\}$
⟨*proof*⟩

lemma *universal-prefix-family-from-hash*:
assumes *M*: $M = measure\text{-}pmf\ p$
assumes *kH*: $k\text{-}universal\ k\ H\ D\ \{\alpha{::}nat\ assg.\ dom\ \alpha = \{0..<n\}\}$
assumes *i*: $i \leq n$
shows $k\text{-}universal\ k\ (Hslice\ i\ H)\ D\ \{\alpha.\ dom\ \alpha = \{0..<i\}\}$
⟨*proof*⟩

end

context *ApproxMCCore*
begin

definition *pivot* :: *real*
where $pivot = 9.84 * (\ 1 + 1\ /\ \varepsilon\ )\hat{}2$

context
assumes *thresh*: $thresh \geq (1 + \varepsilon\ /\ (1 + \varepsilon)) * pivot$
begin

lemma *aux-1*:
assumes *fin*:*finite* (*set-pmf p*)
assumes σ: $\sigma > 0$
assumes *exp*: $\mu\ i = measure\text{-}pmf.expectation\ p\ Y$
assumes *var*: $\sigma\hat{}2 = measure\text{-}pmf.variance\ p\ Y$
assumes *var-bound*: $\sigma\hat{}2 \leq \mu\ i$
shows
$measure\text{-}pmf.prob\ p\ \{y.\ |\ Y\ y - \mu\ i\ | \geq \varepsilon\ /\ (1 + \varepsilon) * \mu\ i\ \}$
$\leq (1 + \varepsilon)\hat{}2\ /\ (\varepsilon\hat{}2 * \mu\ i)$
⟨*proof*⟩

lemma *analysis-1-1*:
assumes *p*: *finite* (*set-pmf p*)
assumes *ind*: *prob-space.k-universal* (*measure-pmf p*) *2 H*
$\{\alpha::'a\ assg.\ dom\ \alpha = S\}$
$\{\alpha::nat\ assg.\ dom\ \alpha = \{0..<card\ S - 1\}\}$
assumes *i*: $i \leq card\ S - 1$
shows
measure-pmf.prob p
$\{s.\ |\ card\text{-}slice\ ((\lambda w.\ H\ w\ s))\ i - \mu\ i\ | \geq \varepsilon\ /\ (1 + \varepsilon) * \mu\ i\}$
$\leq (1 + \varepsilon)\hat{}2\ /\ (\varepsilon\hat{}2 * \mu\ i)$
⟨*proof*⟩

lemma *analysis-1-2*:
assumes *p*: *finite* (*set-pmf p*)
assumes *ind*: *prob-space.k-universal* (*measure-pmf p*) *2 H*
$\{\alpha::'a\ assg.\ dom\ \alpha = S\}$
$\{\alpha::nat\ assg.\ dom\ \alpha = \{0..<card\ S - 1\}\}$
assumes *i*: $i \leq card\ S - 1$
assumes β: $\beta \leq 1$
shows *measure-pmf.prob p*
$\{s.\ real(card\text{-}slice\ ((\lambda w.\ H\ w\ s))\ i) \leq \beta * \mu\ i\ \}$
$\leq 1\ /\ (1 + (1 - \beta)\hat{}2 * \mu\ i)$
⟨*proof*⟩

lemma *shift-μ*:
assumes $k \leq i$
shows$\mu\ i * 2\hat{}k = \mu\ (i-k)$
⟨*proof*⟩

lemma *analysis-2-1*:
assumes *p*: *finite* (*set-pmf p*)
assumes *ind*: *prob-space.k-universal* (*measure-pmf p*) *2 H*
$\{\alpha::'a\ assg.\ dom\ \alpha = S\}$
$\{\alpha::nat\ assg.\ dom\ \alpha = \{0..<card\ S - 1\}\}$
assumes *ε-up*: $\varepsilon \leq 1$
shows
measure-pmf.prob (*map-pmf* ($\lambda s\ w.\ H\ w\ s$) *p*) (*T* (*mstar*−*3*))
$\leq 1\ /\ 62.5$
⟨*proof*⟩

lemma *analysis-2-1'*:
assumes *p*: *finite* (*set-pmf p*)
assumes *ind*: *prob-space.k-universal* (*measure-pmf p*) *2 H*
$\{\alpha::'a\ assg.\ dom\ \alpha = S\}$

{α::nat assg. dom α = {0..<card S − 1}}
shows
measure-pmf.prob (map-pmf (λs w. H w s) p) (T (mstar−3))
≤ 1 / 10.84
⟨*proof*⟩

lemma *analysis-2-2*:
assumes *p*: *finite (set-pmf p)*
assumes *ind*: *prob-space.k-universal (measure-pmf p) 2 H*
{α::'a assg. dom α = S}
{α::nat assg. dom α = {0..<card S − 1}}
shows
measure-pmf.prob (map-pmf (λs w. H w s) p) (L (mstar−2)) ≤ 1 / 20.68
⟨*proof*⟩

lemma *analysis-2-3*:
assumes *p*: *finite (set-pmf p)*
assumes *ind*: *prob-space.k-universal (measure-pmf p) 2 H*
{α::'a assg. dom α = S}
{α::nat assg. dom α = {0..<card S − 1}}
shows
measure-pmf.prob
(map-pmf (λs w. H w s) p) (L (mstar−1)) ≤ 1 / 10.84
⟨*proof*⟩

lemma *analysis-2-4*:
assumes *p*: *finite (set-pmf p)*
assumes *ind*: *prob-space.k-universal (measure-pmf p) 2 H*
{α::'a assg. dom α = S}
{α::nat assg. dom α = {0..<card S − 1}}
shows
measure-pmf.prob (map-pmf (λs w. H w s) p) (L mstar ∪ U mstar) ≤ 1 / 4.92
⟨*proof*⟩

lemma *analysis-3*:
assumes *p*: *finite (set-pmf p)*
assumes *ind*: *prob-space.k-universal (measure-pmf p) 2 H*
{α::'a assg. dom α = S}
{α::nat assg. dom α = {0..<card S − 1}}
assumes *ε-up*: *ε ≤ 1*
shows
measure-pmf.prob (map-pmf (λs w. H w s) p)
approxcore-fail ≤ 0.36

⟨*proof*⟩

lemma *analysis-3'*:
assumes *p*: *finite* (*set-pmf p*)
assumes *ind*: *prob-space.k-universal* (*measure-pmf p*) *2 H*
{α::'*a assg. dom* α = *S*}
{α::*nat assg. dom* α = {*0..<card S* − *1*}}
shows
measure-pmf.prob (*map-pmf* (λ*s w. H w s*) *p*)
approxcore-fail ≤ *0.44*
⟨*proof*⟩
end

end

end

6 ApproxMC definition and analysis

This section puts together preceding results to formalize the PAC guarantee of ApproxMC.

theory *ApproxMCAnalysis* **imports**
ApproxMCCoreAnalysis
RandomXORHashFamily
Median-Method.Median
begin

lemma *replicate-pmf-Pi-pmf*:
assumes *distinct ls*
shows *replicate-pmf* (*length ls*) *P* =
map-pmf (λ*f. map f ls*)
(*Pi-pmf* (*set ls*) *def* (λ-. *P*))
⟨*proof*⟩

lemma *replicate-pmf-Pi-pmf'*:
assumes *finite V*
shows *replicate-pmf* (*card V*) *P* =
map-pmf (λ*f. map f* (*sorted-list-of-set V*))
(*Pi-pmf V def* (λ-. *P*))
⟨*proof*⟩

definition *map-of-default*::('*a* × '*b*) *list* ⇒ '*b* ⇒ '*a* ⇒ '*b*
where *map-of-default ls def* =
(*let m* = *map-of ls in*
(λ*x. case m x of None* ⇒ *def* | *Some v* ⇒ *v*))

lemma *Pi-pmf-replicate-pmf*:

assumes *finite V*
shows
(Pi-pmf V def (λ-. p)) =
map-pmf (λbs.
map-of-default (zip (sorted-list-of-set V) bs) def)
(replicate-pmf (card V) p)
⟨*proof*⟩

lemma *proj-inter-neutral*:
assumes $\bigwedge w.\ w \in B \longleftrightarrow restr\ S\ w \in C$
shows $proj\ S\ (A \cap B) = proj\ S\ A \cap C$
⟨*proof*⟩

An abstract spec of ApproxMC for any Boolean theory. This locale must be instantiated with a theory implementing the two the functions below (and satisfying the assumption linking them).

locale *ApproxMC =*
fixes *sols* :: *'fml ⇒ ('a ⇒ bool) set*
fixes *enc-xor* :: *'a set × bool ⇒ 'fml ⇒ 'fml*
assumes *sols-enc-xor*:
$\bigwedge F\ xor.\ finite\ (fst\ xor) \Longrightarrow$
sols (enc-xor xor F) =
sols F ∩ {ω. satisfies-xor xor {x. ω x}}
begin

definition *compute-thresh* :: *real ⇒ nat*
where *compute-thresh ε =*
nat ⌈1 + 9.84 ∗ (1 + ε / (1 + ε)) ∗ (1 + 1 / ε)^2⌉

definition *fix-t* :: *real ⇒ nat*
where *fix-t δ =*
nat ⌈ln (1 / δ) /(2 ∗ (0.5−0.36)^2)⌉

definition *raw-median-bound* :: *real ⇒ nat ⇒ real*
where *raw-median-bound α t =*
(∑ i = 0..t div 2.
(t choose i) ∗ (1 / 2 + α) ^ i ∗ (1 / 2 − α) ^ (t − i))

definition *compute-t* :: *real ⇒ nat ⇒ nat*
where *compute-t δ n =*
(if raw-median-bound 0.14 n < δ then n
else fix-t δ)

definition *size-xor* ::
'fml ⇒ 'a set ⇒
(nat ⇒ ('a set × bool) option) ⇒
nat ⇒ nat
where *size-xor F S xorsf i = (*

```
    let xors = map (the ∘ xorsf) [0..<i] in
    let Fenc = fold enc-xor xors F in
      card (proj S (sols Fenc))
    )
```

definition *check-xor* ::
  'fml ⇒ 'a set ⇒
  nat ⇒
  (nat ⇒ ('a set × bool) option) ⇒
  nat ⇒ bool
  where *check-xor F S thresh xorsf i =*
  (size-xor F S xorsf i < thresh)

definition *approxcore-xors* ::
  'fml ⇒ 'a set ⇒
  nat ⇒
  (nat ⇒ ('a set × bool) option) ⇒
  nat
  where
    approxcore-xors F S thresh xorsf =
    (case List.find
      (check-xor F S thresh xorsf) [1..<card S] of
      None ⇒ 2 ^ card S
    | Some m ⇒
      (2 ^ m ∗ size-xor F S xorsf m))

definition *approxmccore* :: *'fml ⇒ 'a set ⇒ nat ⇒ nat pmf*
where *approxmccore F S thresh =*
  map-pmf (approxcore-xors F S thresh) (random-xors S (card S − 1))

definition *approxmc* :: *'fml ⇒ 'a set ⇒ real ⇒ real ⇒ nat ⇒ nat pmf*
  where *approxmc F S ε δ n = (*
    let thresh = compute-thresh ε in
    if card (proj S (sols F)) < thresh then
      return-pmf (card (proj S (sols F)))
    else
      let t = compute-t δ n in
      map-pmf (median t)
        (prod-pmf {0..<t::nat} (λi. approxmccore F S thresh))
)

lemma *median-commute*:
  assumes $t \geq 1$
  shows *(real ∘ median t) = (λw::nat ⇒ nat. median t (real ∘ w))*
  ⟨*proof*⟩

lemma *median-default*:
  shows *median t y = median t (λx. if x < t then y x else def)*

⟨*proof*⟩

definition *default-α*::*'a set* ⇒ *nat assg*
where *default-α S i* = (*if i* < *card S* − *1 then Some True else None*)

lemma *dom-default-α*:
dom (*default-α S*) = {*0*..<*card S* −*1*}
⟨*proof*⟩

lemma *compute-thresh-bound-4*:
assumes *ε* > *0*
shows*4* < *compute-thresh ε*
⟨*proof*⟩

lemma *satisfies-xor-with-domain*:
assumes *fst x* ⊆ *S*
shows *satisfies-xor x* {*x*. *w x*} ⟷
satisfies-xor x ({*x*. *w x*} ∩ *S*)
⟨*proof*⟩

lemma *approxcore-xors-eq*:
assumes *thresh*:
thresh = *compute-thresh ε*
thresh ≤ *card* (*proj S* (*sols F*))
assumes *ε*: *ε* > (*0*::*real*) *ε* ≤ *1*
assumes *S*: *finite S*
shows *measure-pmf*.*prob* (*random-xors S* (*card S* − *1*))
{*xors*. *real* (*approxcore-xors F S thresh xors*) ∈
{*real* (*card* (*proj S* (*sols F*))) / (*1* + *ε*)..
(*1* + *ε*) ∗ *real* (*card* (*proj S* (*sols F*)))}} ≥ *0.64*
⟨*proof*⟩

lemma *compute-t-ge1*:
assumes *0* < *δ δ* < *1*
shows*compute-t δ n* ≥ *1*
⟨*proof*⟩

lemma *success-arith-bound*:
assumes *s* ≤ (*f* ::*nat*)
assumes *p* ≤ (*1*::*real*) *q* ≤ *p 1* /*2* ≤ *q*
shows *p* ^ *s* ∗ (*1* − *p*) ^ *f* ≤ *q* ^ *s* ∗ (*1* − *q*) ^ *f*
⟨*proof*⟩

lemma *prob-binomial-pmf-upto-mono*:
assumes *1*/*2* ≤ *q q* ≤ *p p* ≤ *1*
shows
measure-pmf.*prob* (*binomial-pmf n p*) {..*n div 2*} ≤
measure-pmf.*prob* (*binomial-pmf n q*) {..*n div 2*}

⟨*proof*⟩

theorem *approxmc-sound*:
  assumes δ: $\delta > 0$ $\delta < 1$
  assumes ε: $\varepsilon > 0$ $\varepsilon \leq 1$
  assumes *S*: *finite S*
  shows *measure-pmf.prob* (*approxmc F S* ε δ *n*)
      {*c*. *real c* ∈
        {*real* (*card* (*proj S* (*sols F*))) / (*1* + ε)..
          (*1* + ε) ∗ *real* (*card* (*proj S* (*sols F*)))}}
      ≥ *1* − δ
⟨*proof*⟩

To simplify further analyses, we can remove the (required) upper bound on epsilon.

definition *mk-eps* :: *real* ⇒ *real*
  where *mk-eps* ε = (*if* ε > *1 then 1 else* ε)

definition *approxmc′*::
  ′fml ⇒ *′a set* ⇒
  real ⇒ *real* ⇒ *nat* ⇒ *nat pmf*
  where *approxmc′ F S* ε δ *n* =
    approxmc F S (*mk-eps* ε) δ *n*

corollary *approxmc′-sound*:
  assumes δ: $\delta > 0$ $\delta < 1$
  assumes ε: $\varepsilon > 0$
  assumes *S*: *finite S*
  shows *prob-space.prob* (*approxmc′ F S* ε δ *n*)
      {*c*. *real c* ∈
        {*real* (*card* (*proj S* (*sols F*))) / (*1* + ε)..
          (*1* + ε) ∗ *real* (*card* (*proj S* (*sols F*)))}}
      ≥ *1* − δ
⟨*proof*⟩

This shows we can lift all randomness to the top-level (i.e., eagerly sample it).

definition *approxmc-map*::
  ′fml ⇒ *′a set* ⇒
  real ⇒ *real* ⇒ *nat* ⇒
  (*nat* ⇒ *nat* ⇒ (*′a set* × *bool*) *option*) ⇒
  nat
  where *approxmc-map F S* ε δ *n xorsFs* = (
    let ε = *mk-eps* ε *in*
    let thresh = *compute-thresh* ε *in*
    if card (*proj S* (*sols F*)) < *thresh then*
      card (*proj S* (*sols F*))
    else

let t = compute-t δ n in
median t (approxcore-xors F S thresh ∘ xorsFs))

lemma *approxmc-map-eq*:
shows
map-pmf (approxmc-map F S ε δ n)
(Pi-pmf {0..<compute-t δ n} def
(λi. random-xors S (card S − 1))) =
approxmc′ F S ε δ n
⟨*proof*⟩

end

end

7 Certificate-based ApproxMC

This turns the randomized algorithm into an executable certificate checker

theory *CertCheck*
imports *ApproxMCAnalysis*

begin

7.1 ApproxMC with lists instead of sets

type-synonym *′a xor = ′a list × bool*

definition *satisfies-xorL* :: *′a xor ⇒ (′a ⇒ bool) ⇒ bool*
where *satisfies-xorL xb ω =*
even (sum-list (map (of-bool ∘ ω) (fst xb)) +
of-bool (snd xb)::nat)

definition *sublist-bits*::*′a list ⇒ bool list ⇒ ′a list*
where *sublist-bits ls bs =*
map fst (filter snd (zip ls bs))

definition *xor-from-bits*::
′a list ⇒ bool list × bool ⇒ ′a xor
where *xor-from-bits V xsb =*
(sublist-bits V (fst xsb), snd xsb)

locale *ApproxMCL =*
fixes *sols* :: *′fml ⇒ (′a ⇒ bool) set*
fixes *enc-xor* :: *′a xor ⇒ ′fml ⇒ ′fml*
assumes *sols-enc-xor*:
⋀F xor.

sols (*enc-xor xor F*) =
sols F ∩ {ω. *satisfies-xorL xor* ω}
begin

definition *list-of-set* ::′*a set* ⇒ ′*a list*
where *list-of-set x* = (@*ls*. *set ls* = *x* ∧ *distinct ls*)

definition *xor-conc* :: ′*a set* × *bool* ⇒ ′*a xor*
where *xor-conc xsb* = (*list-of-set* (*fst xsb*), *snd xsb*)

definition *enc-xor-conc* :: ′*a set* × *bool* ⇒ ′*fml* ⇒ ′*fml*
where *enc-xor-conc* = *enc-xor* ∘ *xor-conc*

lemma *distinct-count-list*:
assumes *distinct ls*
shows *count-list ls x* = *of-bool* (*x* ∈ *set ls*)
⟨*proof*⟩

lemma *list-of-set*:
assumes *finite x*
shows *distinct* (*list-of-set x*) *set* (*list-of-set x*) = *x*
⟨*proof*⟩

lemma *count-list-list-of-set*:
assumes *finite x*
shows *count-list* (*list-of-set x*) *y* = *of-bool* (*y* ∈ *x*)
⟨*proof*⟩

lemma *satisfies-xorL-xor-conc*:
assumes *finite x*
shows*satisfies-xorL* (*xor-conc* (*x*,*b*)) ω ⟷ *satisfies-xor* (*x*,*b*) {*x*. ω *x*}
⟨*proof*⟩

sublocale *appmc*: *ApproxMC sols enc-xor-conc*
⟨*proof*⟩

definition *size-xorL* ::
′*fml* ⇒ ′*a list* ⇒
(*nat* ⇒ *bool list* × *bool*) ⇒
nat ⇒ *nat*
where *size-xorL F S xorsl i* = (
let xors = *map* (*xor-from-bits S* ∘ *xorsl*) [*0*..<*i*] *in*
let Fenc = *fold enc-xor xors F in*
card (*proj* (*set S*) (*sols Fenc*)))

definition *check-xorL* ::
′*fml* ⇒ ′*a list* ⇒
nat ⇒

(*nat* ⇒ *bool list* × *bool*) ⇒
nat ⇒ *bool*
where *check-xorL F S thresh xorsl i* =
(*size-xorL F S xorsl i* < *thresh*)

definition *approxcore-xorsL* ::
′*fml* ⇒ ′*a list* ⇒
nat ⇒
(*nat* ⇒ (*bool list* × *bool*)) ⇒
nat
where
approxcore-xorsL F S thresh xorsl =
(*case List.find*
(*check-xorL F S thresh xorsl*) [*1..<length S*] *of*
None ⇒ *2* ^ *length S*
| *Some m* ⇒
(*2* ^ *m* ∗ *size-xorL F S xorsl m*))

definition *xor-abs* :: ′*a xor* ⇒ ′*a set* × *bool*
where *xor-abs xsb* = (*set* (*fst xsb*), *snd xsb*)

lemma *sols-fold-enc-xor*:
assumes *list-all2* (λ*x y*.
∀ *w*. *satisfies-xorL x w* = *satisfies-xorL y w*) *xs ys*
assumes *sols F* = *sols G*
shows *sols* (*fold enc-xor xs F*) = *sols* (*fold enc-xor ys G*)
⟨*proof*⟩

lemma *satisfies-xor-xor-abs*:
assumes *distinct x*
shows*satisfies-xor* (*xor-abs* (*x*,*b*)) {*x*. ω *x*} ⟷ *satisfies-xorL* (*x*,*b*)
ω
⟨*proof*⟩

lemma *xor-conc-xor-abs-rel*:
assumes *distinct* (*fst x*)
shows*satisfies-xorL* (*xor-conc* (*xor-abs x*)) *w* ⟷
satisfies-xorL x w
⟨*proof*⟩

lemma *sorted-sublist-bits*:
assumes *sorted V*
shows*sorted* (*sublist-bits V bs*)
⟨*proof*⟩

lemma *distinct-sublist-bits*:
assumes *distinct V*
shows*distinct* (*sublist-bits V bs*)
⟨*proof*⟩

lemma *distinct-fst-xor-from-bits*:
  assumes *distinct V*
  shows *distinct* (*fst* (*xor-from-bits V bs*))
  ⟨*proof*⟩

lemma *size-xorL*:
  assumes $\bigwedge j.\ j < i \Longrightarrow$
    xorsf j =
    Some (*xor-abs* (*xor-from-bits S* (*xorsl j*)))
  assumes *distinct S*
  shows *size-xorL F S xorsl i* =
    appmc.size-xor F (*set S*) *xorsf i*
  ⟨*proof*⟩

lemma *fold-enc-xor-more*:
  assumes $x \in$ *sols* (*fold enc-xor* (*xs* @ *rev ys*) *F*)
  shows $x \in$ *sols* (*fold enc-xor xs F*)
  ⟨*proof*⟩

lemma *size-xorL-anti-mono*:
  assumes $x \leq y$ *distinct S*
  shows *size-xorL F S xorsl x* $\geq$ *size-xorL F S xorsl y*
⟨*proof*⟩

lemma *find-upto-SomeI*:
  assumes $\bigwedge i.\ a \leq i \Longrightarrow i < x \Longrightarrow \neg P\ i$
  assumes $P\ x\ a \leq x\ x < b$
  shows *find P* [*a*..<*b*] = *Some x*
⟨*proof*⟩

lemma *check-xorL*:
  assumes $\bigwedge j.\ j < i \Longrightarrow$
    xorsf j =
    Some (*xor-abs* (*xor-from-bits S* (*xorsl j*)))
  assumes *distinct S*
  shows *check-xorL F S thresh xorsl i* =
    appmc.check-xor F (*set S*) *thresh xorsf i*
  ⟨*proof*⟩

lemma *approxcore-xorsL*:
  assumes $\bigwedge j.\ j <$ *length S* − *1* $\Longrightarrow$
    xorsf j =
    Some (*xor-abs* (*xor-from-bits S* (*xorsl j*)))
  assumes *S*: *distinct S*
  shows *approxcore-xorsL F S thresh xorsl* =
    appmc.approxcore-xors F (*set S*) *thresh xorsf*
⟨*proof*⟩

definition *approxmc-mapL*::
  'fml ⇒ *'a list* ⇒
  real ⇒ *real* ⇒ *nat* ⇒
  (*nat* ⇒ *nat* ⇒ (*bool list* × *bool*)) ⇒
  nat
  where *approxmc-mapL F S ε δ n xorsLs* = (
    let ε = appmc.mk-eps ε in
    let thresh = appmc.compute-thresh ε in
    if card (proj (set S) (sols F)) < thresh then
      card (proj (set S) (sols F))
    else
      let t = appmc.compute-t δ n in
      median t (approxcore-xorsL F S thresh ∘ xorsLs))

definition *random-xorB* :: *nat* ⇒ (*bool list* × *bool*) *pmf*
  where *random-xorB n* =
    pair-pmf
      (*replicate-pmf n* (*bernoulli-pmf* (*1/2*)))
      (*bernoulli-pmf* (*1/2*))

lemma *approxmc-mapL*:
  assumes ⋀*i j. j < length S − 1* ⟹
    xorsFs i j =
    Some (*xor-abs* (*xor-from-bits S* (*xorsLs i j*)))
  assumes *S*: *distinct S*
  shows
    approxmc-mapL F S ε δ n xorsLs =
    appmc.approxmc-map F (*set S*) *ε δ n xorsFs*
⟨*proof*⟩

lemma *approxmc-mapL'*:
  assumes *S*: *distinct S*
  shows
    approxmc-mapL F S ε δ n xorsLs =
    appmc.approxmc-map F (*set S*) *ε δ n*
      (*λi j. if j < length S − 1*
        then Some (*xor-abs* (*xor-from-bits S* (*xorsLs i j*)))
        else None)
  ⟨*proof*⟩

lemma *bits-to-random-xor*:
  assumes *distinct S*
  shows *map-pmf*
    (*λx. xor-abs* (*xor-from-bits S x*))
    (*random-xorB* (*length S*)) =
    random-xor (*set S*)
⟨*proof*⟩

lemma *Pi-pmf-map-pmf-Some*:
  assumes *finite S*
  shows *Pi-pmf S None* (λ-. *map-pmf Some p*) =
    map-pmf (λ*f v*. *if v* ∈ *S then Some* (*f v*) *else None*)
    (*Pi-pmf S def* (λ-. *p*))
⟨*proof*⟩

lemma *bits-to-random-xors*:
  assumes *distinct S*
  shows
    map-pmf
    (λ*f j*.
      if j < *n*
      then Some (*xor-abs* (*xor-from-bits S* (*f j*)))
      else None)
    (*Pi-pmf* {..<(*n*::*nat*)} *def* (λ-. *random-xorB* (*length S*))) =
    random-xors (*set S*) *n*
  ⟨*proof*⟩

lemma *bits-to-all-random-xors*:
  assumes *distinct S*
  assumes (λ*j*. *if j* < *n*
          then Some (*xor-abs* (*xor-from-bits S* (*def1 j*)))
          else None) = *def*
  shows
    map-pmf
    ((∘) (λ*f j*. *if j* < *n*
              then Some (*xor-abs* (*xor-from-bits S* (*f j*)))
              else None))
    (*Pi-pmf* {*0*..<(*m*::*nat*)} *def1*
      (λ-.
        Pi-pmf {..<(*n*::*nat*)} *def2* (λ-. *random-xorB* (*length S*)))) =
    Pi-pmf {*0*..<*m*} *def*
        (λ*i*. *random-xors* (*set S*) *n*)
  ⟨*proof*⟩

definition *random-seed-xors*::*nat* ⇒ *nat* ⇒ (*nat* ⇒ *nat* ⇒ *bool list* × *bool*) *pmf*
  where *random-seed-xors t l* =
    (*prod-pmf* {*0*..<*t*}
      (λ-. *prod-pmf* {..<*l*−*1*} (λ-. *random-xorB l*)))

lemma *approxmcL-sound*:
  assumes δ: δ > *0* δ < *1*
  assumes ε: ε > *0*
  assumes *S*: *distinct S*
  shows
    prob-space.prob

(*map-pmf* (*approxmc-mapL F S ε δ n*)
  (*random-seed-xors* (*appmc.compute-t δ n*) (*length S*)))
{*c. real c* ∈
  {*real* (*card* (*proj* (*set S*) (*sols F*))) / (*1* + *ε*)..
    (*1* + *ε*) ∗ *real* (*card* (*proj* (*set S*) (*sols F*)))}}
≥ *1* − *δ*
⟨*proof*⟩

lemma *approxmcL-sound′*:
  assumes *δ*: *δ* > *0 δ* < *1*
  assumes *ε*: *ε* > *0*
  assumes *S*: *distinct S*
  shows
    prob-space.prob
      (*map-pmf* (*approxmc-mapL F S ε δ n*)
        (*random-seed-xors* (*appmc.compute-t δ n*) (*length S*)))
      {*c. real c* ∉
        {*real* (*card* (*proj* (*set S*) (*sols F*))) / (*1* + *ε*)..
          (*1* + *ε*) ∗ *real* (*card* (*proj* (*set S*) (*sols F*)))}} ≤ *δ*
⟨*proof*⟩

end

7.2 ApproxMC certificate checker

definition *str-of-bool* :: *bool* ⇒ *String.literal*
  where *str-of-bool b* = (
  if b then STR ''true'' else STR ''false'')

fun *str-of-nat-aux* :: *nat* ⇒ *char list* ⇒ *char list*
  where *str-of-nat-aux n acc* = (
  let c = *char-of-integer* (*of-nat* (*48* + *n mod 10*)) *in*
  if n < *10 then c # acc*
  else str-of-nat-aux (*n div 10*) (*c # acc*))

definition *str-of-nat* :: *nat* ⇒ *String.literal*
  where *str-of-nat n* = *String.implode* (*str-of-nat-aux n* [])

type-synonym *′a sol* = (*′a* × *bool*) *list*

definition *canon-map-of* :: (*′a* × *bool*) *list* ⇒ (*′a* ⇒ *bool*)
  where *canon-map-of ls* =
    (*let m* = *map-of ls in*
    (*λx. case m x of None* ⇒ *False* | *Some b* ⇒ *b*))

lemma *canon-map-of*[*code*]:

shows *canon-map-of ls* =
  (*let m* = *Mapping.of-alist ls in*
   Mapping.lookup-default False m)
⟨*proof*⟩

definition *proj-sol* :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool list*
  where *proj-sol S w* = *map w S*

The following extended locale assumes additional support for syntactically working with solutions

locale *CertCheck* = *ApproxMCL sols enc-xor*
  for *sols* :: *'fml* ⇒ (*'a* ⇒ *bool*) *set*
  and *enc-xor* :: *'a list* × *bool* ⇒ *'fml* ⇒ *'fml* +
  fixes *check-sol* :: *'fml* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
  fixes *ban-sol* :: *'a sol* ⇒ *'fml* ⇒ *'fml*
  assumes *sols-ban-sol*:
    ⋀*F vs.*
      sols (*ban-sol vs F*) =
      sols F ∩ {*ω. map ω* (*map fst vs*) ≠ *map snd vs*}
  assumes *check-sol*:
    ⋀*F w. check-sol F w* ⟷ *w* ∈ *sols F*
begin

Assuming parameter access to an UNSAT checking oracle

context
  fixes *check-unsat* :: *'fml* ⇒ *bool*
begin

Throughout this checker, INL indicates error, INR indicates success

definition *check-BSAT-sols*::
  'fml ⇒ *'a list* ⇒ *nat* ⇒ (*'a* ⇒ *bool*) *list* ⇒ *String.literal* + *unit*
where *check-BSAT-sols F S thresh cms* = (
  let ps = *map* (*proj-sol S*) *cms in*
  let b1 = *list-all* (*check-sol F*) *cms in*
  let b2 = *distinct ps in*
  let b3 =
    (*length cms* < *thresh* ⟶
      check-unsat (*fold ban-sol* (*map* (*zip S*) *ps*) *F*)) *in*
    if b1 ∧ *b2* ∧ *b3 then Inr* ()
    else Inl (*STR ''checks −−−''* +
      STR '' all valid sols: '' + *str-of-bool b1* +
      STR '', all distinct sols: '' + *str-of-bool b2* +
      STR '', unsat check (< *thresh sols*)*: ''* + *str-of-bool b3*)
)

definition *BSAT* ::
  'fml ⇒ *'a list* ⇒ *nat* ⇒ (*'a* ⇒ *bool*) *list* ⇒ *String.literal* + *nat*
where *BSAT F S thresh xs* = (

```
  case check-BSAT-sols F S thresh xs of
    Inl err ⇒ Inl err
  | Inr - ⇒ Inr (length xs)
)
```

definition *size-xorL-cert* ::

```
  'fml ⇒ 'a list ⇒ nat ⇒
  (nat ⇒ (bool list × bool)) ⇒ nat ⇒
  (('a ⇒ bool) list) ⇒ String.literal + nat
  where size-xorL-cert F S thresh xorsl i xs = (
    let xors = map (xor-from-bits S ∘ xorsl) [0..<i]  in
    let Fenc = fold enc-xor xors F in
    BSAT Fenc S thresh xs
  )
```

fun *approxcore-xorsL-cert* ::

```
  'fml ⇒ 'a list ⇒ nat ⇒
  nat × ('a ⇒ bool) list × ('a ⇒ bool) list ⇒
  (nat ⇒ (bool list × bool))
  ⇒ String.literal + nat
where approxcore-xorsL-cert F S thresh (m,cert1,cert2) xorsl = (
  if 1 ≤ m ∧ m ≤ length S
  then
    case size-xorL-cert F S thresh xorsl (m−1) cert1 of
      Inl err ⇒ Inl (STR ''cert1 '' + err)
    | Inr n ⇒
    if n ≥ thresh
    then
      if m = length S
      then Inr (2 ^ length S)
      else
        case size-xorL-cert F S thresh xorsl m cert2 of
          Inl err ⇒ Inl (STR ''cert2 '' + err)
        | Inr c ⇒
          if c < thresh then Inr (2 ^ m ∗ c)
          else Inl (STR ''too many solutions at m added XORs'')
    else Inl (STR ''too few solutions at m−1 added XORs'')
  else
     Inl (STR ''invalid value of m, need 1 <= m <= |S|''))
```

definition *find-t* :: *real* ⇒ *nat*

```
where find-t δ = (
  case find (λi. appmc.raw-median-bound 0.14 i < δ) [0..<256] of
    Some m ⇒ m
  | None ⇒ appmc.fix-t δ
  )
```

```
fun fold-approxcore-xorsL-cert::
  'fml ⇒ 'a list ⇒ nat ⇒
  nat ⇒ nat ⇒
  (nat ⇒ (nat × ('a ⇒ bool) list × ('a ⇒ bool) list)) ⇒
  (nat ⇒ nat ⇒ (bool list × bool))
  ⇒ String.literal + (nat list)
  where
  fold-approxcore-xorsL-cert F S thresh t 0 cs xorsLs = Inr []
| fold-approxcore-xorsL-cert F S thresh t (Suc i) cs xorsLs = (
    let it = t − Suc i in
    case approxcore-xorsL-cert F S thresh (cs it) (xorsLs it) of
      Inl err ⇒ Inl (STR ''round '' + str-of-nat it + STR '' '' + err)
    | Inr n ⇒
      (case fold-approxcore-xorsL-cert F S thresh t i cs xorsLs of
          Inl err ⇒ Inl err
        | Inr ns ⇒ Inr (n # ns)))

definition calc-median::
  'fml ⇒ 'a list ⇒ nat ⇒ nat ⇒
  (nat ⇒ (nat × ('a ⇒ bool) list × ('a ⇒ bool) list)) ⇒
  (nat ⇒ nat ⇒ (bool list × bool)) ⇒
  String.literal + nat
  where calc-median F S thresh t ms xorsLs = (
      case fold-approxcore-xorsL-cert F S thresh t t ms xorsLs of
        Inl err ⇒ Inl err
      | Inr ls ⇒ Inr (sort ls ! (t div 2))
  )

fun certcheck::
  'fml ⇒ 'a list ⇒
  real ⇒ real ⇒
  (('a ⇒ bool) list ×
  (nat ⇒ (nat × ('a ⇒ bool) list × ('a ⇒ bool) list))) ⇒
  (nat ⇒ nat ⇒ (bool list × bool)) ⇒
  String.literal + nat
  where certcheck F S ε δ (m0,ms) xorsLs = (
    let ε = appmc.mk-eps ε in
    let thresh = appmc.compute-thresh ε in
    case BSAT F S thresh m0 of Inl err ⇒ Inl err
    | Inr Y ⇒
    if Y < thresh then Inr Y
    else
      let t = find-t δ in
      calc-median F S thresh t ms xorsLs)


context
  assumes check-unsat: ⋀F. check-unsat F ⟹ sols F = {}
```

begin

lemma *sols-fold-ban-sol*:
  shows*sols* (*fold ban-sol ls F*) =
  sols F ∩ {ω. (∀ *vs* ∈ *set ls*. *map* ω (*map fst vs*) ≠ *map snd vs*)}
⟨*proof*⟩

lemma *inter-cong-right*:
  assumes ⋀*x*. *x* ∈ *A* ⟹ *x* ∈ *B* ⟷ *x* ∈ *C*
  shows *A* ∩ *B* = *A* ∩ *C*
  ⟨*proof*⟩

lemma *proj-sol-canon-map-of*:
  assumes *distinct S length S* = *length w*
  shows *proj-sol S* (*canon-map-of* (*zip S w*)) = *w*
  ⟨*proof*⟩

lemma *proj-sol-cong*:
  assumes *restr* (*set S*) *A* = *restr* (*set S*) *B*
  shows *proj-sol S A* = *proj-sol S B*
  ⟨*proof*⟩

lemma *canon-map-of-map-of*:
  assumes *length S* = *length x*
  assumes *canon-map-of* (*zip S x*) ∈ *A*
  shows *map-of* (*zip S x*) ∈ *proj* (*set S*) *A*
⟨*proof*⟩

lemma *proj-proj-sol-map-of-zip-1*:
  assumes *distinct S length S* = *length w*
  assumes *w*: *w* ∈ *rdb*
  shows
    map-of (*zip S w*) ∈
    proj (*set S*) {ω. *proj-sol S* ω ∈ *rdb*}
  ⟨*proof*⟩

lemma *proj-proj-sol-map-of-zip-2*:
  assumes ⋀*bs*. *bs* ∈ *rdb* ⟹ *length bs* = *length S*
  assumes *w*: *w* ∈ *proj* (*set S*) {ω. *proj-sol S* ω ∈ *rdb*}
  shows
    w ∈ (*map-of* ∘ *zip S*) ‘ *rdb*
⟨*proof*⟩

lemma *proj-proj-sol-map-of-zip*:
  assumes *distinct S*
  assumes ⋀*bs*. *bs* ∈ *rdb* ⟹ *length bs* = *length S*
  shows
    proj (*set S*) {ω. *proj-sol S* ω ∈ *rdb*} =
    (*map-of* ∘ *zip S*) ‘ *rdb*

⟨proof⟩

definition *ban-proj-sol* :: $'a$ *list* ⇒ ($'a$ ⇒ *bool*) *list* ⇒ $'fml$ ⇒ $'fml$
where *ban-proj-sol S xs F* =
fold ban-sol (*map* (*zip S* ∘ *proj-sol S*) *xs*) *F*

lemma *check-sol-imp-proj*:
assumes $w \in$ *sols F*
shows *map-of* (*zip S* (*proj-sol S w*)) ∈ *proj* (*set S*) (*sols F*)
⟨proof⟩

lemma *checked-BSAT-lower*:
assumes *S*: *distinct S*
assumes *check-BSAT-sols F S thresh xs* = *Inr* ()
shows *length xs* ≤ *card* (*proj* (*set S*) (*sols F*))
length xs < *thresh* ⟹
card (*proj* (*set S*) (*sols F*)) = *length xs*
⟨proof⟩

lemma *good-BSAT*:
assumes *distinct S*
assumes *BSAT F S thresh xs* = *Inr n*
shows *n* ≤ *card* (*proj* (*set S*) (*sols F*))
n < *thresh* ⟹
card (*proj* (*set S*) (*sols F*)) = *n*
⟨proof⟩

lemma *size-xorL-cert*:
assumes *distinct S*
assumes *size-xorL-cert F S thresh xorsl i xs* = *Inr n*
shows
size-xorL F S xorsl i ≥ *n*
n < *thresh* ⟶ *size-xorL F S xorsl i* = *n*
⟨proof⟩

lemma *approxcore-xorsL-cert*:
assumes *S*: *distinct S*
assumes *approxcore-xorsL-cert F S thresh mc xorsl* = *Inr n*
shows *approxcore-xorsL F S thresh xorsl* = *n*
⟨proof⟩

lemma *fold-approxcore-xorsL-cert*:
assumes *S*: *distinct S*
assumes *i* ≤ *t*
assumes *fold-approxcore-xorsL-cert F S thresh t i cs xorsLs* = *Inr*
ns
shows *map* (*approxcore-xorsL F S thresh* ∘ *xorsLs*) [*t*−*i*..<*t*] = *ns*
⟨proof⟩

lemma *calc-median*:
  assumes *S*: *distinct S*
  assumes *calc-median F S thresh t ms xorsLs* = *Inr n*
  shows *median t* (*approxcore-xorsL F S thresh* ∘ *xorsLs*) = *n*
  ⟨*proof*⟩

lemma *compute-t-find-t*[*simp*]:
  shows *appmc.compute-t* δ (*find-t* δ) = *find-t* δ
  ⟨*proof*⟩

lemma *certcheck*:
  assumes *distinct S*
  assumes *certcheck F S* ε δ (*m0*,*ms*) *xorsLs* = *Inr n*
  shows *approxmc-mapL F S* ε δ (*find-t* δ) *xorsLs* = *n*
  ⟨*proof*⟩

lemma *certcheck′*:
  assumes *distinct S*
  assumes ¬*isl* (*certcheck F S* ε δ *m xorsLs*)
  shows *projr* (*certcheck F S* ε δ *m xorsLs*) =
    approxmc-mapL F S ε δ (*find-t* δ) *xorsLs*
  ⟨*proof*⟩

lemma *certcheck-sound*:
  assumes δ: $\delta > 0$ $\delta < 1$
  assumes ε: $\varepsilon > 0$
  assumes *S*: *distinct S*
  shows
    measure-pmf.prob
      (*map-pmf* (λ*r*. *certcheck F S* ε δ (*f r*) *r*)
        (*random-seed-xors* (*find-t* δ) (*length S*)))
      {*c*. ¬*isl c* ∧
        real (*projr c*) ∉
        {*real* (*card* (*proj* (*set S*) (*sols F*))) / (*1* + ε)..
          (*1* + ε) ∗ *real* (*card* (*proj* (*set S*) (*sols F*)))}} ≤ δ
⟨*proof*⟩

lemma *certcheck-promise-complete*:
  assumes δ: $\delta > 0$ $\delta < 1$
  assumes ε: $\varepsilon > 0$
  assumes *S*: *distinct S*
  assumes *r*: ⋀*r*.
    r ∈ *set-pmf* (*random-seed-xors* (*find-t* δ) (*length S*)) ⟹
    ¬*isl* (*certcheck F S* ε δ (*f r*) *r*)
  shows
    measure-pmf.prob
      (*map-pmf* (λ*r*. *certcheck F S* ε δ (*f r*) *r*)

    (*random-seed-xors* (*find-t* δ) (*length* *S*)))
  {*c*. *real* (*projr* *c*) $\in$
    {*real* (*card* (*proj* (*set* *S*) (*sols* *F*))) / (*1* + ε)..
      (*1* + ε) $*$ *real* (*card* (*proj* (*set* *S*) (*sols* *F*)))}} $\geq$ *1* $-$ δ
⟨*proof*⟩

end

lemma *certcheck-code*[*code*]:
  certcheck F S ε δ (*m0*,*ms*) *xorsLs* = (
  if δ > *0* $\wedge$ δ < *1* $\wedge$ ε > *0* $\wedge$ *distinct S then*
    (*let* ε = *appmc.mk-eps* ε *in*
    let thresh = *appmc.compute-thresh* ε *in*
    case BSAT F S thresh m0 of Inl err $\Rightarrow$ *Inl err*
    | *Inr Y* $\Rightarrow$
    if Y < *thresh then Inr Y*
    else
      let t = *find-t* δ *in*
      calc-median F S thresh t ms xorsLs)
  else Code.abort (*STR ''invalid inputs''*)
    (λ-. *certcheck F S* ε δ (*m0*,*ms*) *xorsLs*))
⟨*proof*⟩

end

end

end

8 ApproxMC certification for CNF-XOR

This concretely instantiates the locales with a syntax and semantics for CNF-XOR, giving us a certificate checker for approximate counting in this theory.

theory *CertCheck-CNF-XOR* **imports**
  ApproxMCAnalysis
  CertCheck
  HOL.String HOL−Library.Code-Target-Numeral
  Show.Show-Real
begin

This follows CryptoMiniSAT's CNF-XOR formula syntax. A clause is a list of literals (one of which must be satisfied). An XOR constraint has the form $l_1 + l_2 + \ldots + l_n = 1$ where addition is taken over F_2. Syntactically, they are specified by the list of LHS literals. Variables are natural numbers (in practice, variable 0 is never used)

datatype *lit* = *Pos nat* | *Neg nat*
type-synonym *clause* = *lit list*
type-synonym *cmsxor* = *lit list*
type-synonym *fml* = *clause list* × *cmsxor list*

type-synonym *assignment* = *nat* ⇒ *bool*

definition *sat-lit* :: *assignment* ⇒ *lit* ⇒ *bool* **where**
  sat-lit w l = (*case l of Pos x* ⇒ *w x* | *Neg x* ⇒ ¬*w x*)

definition *sat-clause* :: *assignment* ⇒ *clause* ⇒ *bool* **where**
  sat-clause w C = (∃ *l* ∈ *set C*. *sat-lit w l*)

definition *sat-cmsxor* :: *assignment* ⇒ *cmsxor* ⇒ *bool* **where**
  sat-cmsxor w C = *odd* ((*sum-list* (*map* (*of-bool* ∘ (*sat-lit w*)) *C*))::*nat*)

definition *sat-fml* :: *assignment* ⇒ *fml* ⇒ *bool*
  where
  sat-fml w f = (
    (∀ *C* ∈ *set* (*fst f*). *sat-clause w C*) ∧
    (∀ *C* ∈ *set* (*snd f*). *sat-cmsxor w C*))

definition *sols* :: *fml* ⇒ *assignment set*
  where *sols f* = {*w*. *sat-fml w f*}

lemma *sat-fml-cons*[*simp*]:
  shows
  sat-fml w (*FC*, *x* # *FX*) ⟷
  sat-fml w (*FC*,*FX*) ∧ *sat-cmsxor w x*
  sat-fml w (*c* # *FC*, *FX*) ⟷
  sat-fml w (*FC*,*FX*) ∧ *sat-clause w c*
  ⟨*proof*⟩

fun *enc-xor* :: *nat xor* ⇒ *fml* ⇒ *fml*
  where
    enc-xor (*x*,*b*) (*FC*,*FX*) = (
    if b then (*FC*, *map Pos x* # *FX*)
    else
      case x of
        [] ⇒ (*FC*,*FX*)
      | (*v*#*vs*) ⇒ (*FC*, (*Neg v* # *map Pos vs*) # *FX*))

lemma *sols-enc-xor*:
  shows *sols* (*enc-xor* (*x*,*b*) (*FC*,*FX*)) =
        sols (*FC*,*FX*) ∩ {ω. *satisfies-xorL* (*x*,*b*) ω}
  ⟨*proof*⟩

definition *check-sol* :: *fml* ⇒ (*nat* ⇒ *bool*) ⇒ *bool*
where *check-sol fml w* = (
list-all (*list-ex* (*sat-lit w*)) (*fst fml*) ∧
list-all (*sat-cmsxor w*) (*snd fml*))

definition *ban-sol* :: (*nat* × *bool*) *list* ⇒ *fml* ⇒ *fml*
where *ban-sol vs fml* =
((*map* (λ(*v*,*b*). *if b then Neg v else Pos v*) *vs*)#*fst fml*, *snd fml*)

lemma *check-sol-sol*:
shows *w* ∈ *sols F* ⟷
check-sol F w
⟨*proof*⟩

lemma *ban-sat-clause*:
shows *sat-clause w* (*map* (λ(*v*, *b*). *if b then Neg v else Pos v*) *vs*)
⟷
map w (*map fst vs*) ≠ *map snd vs*
⟨*proof*⟩

lemma *sols-ban-sol*:
shows*sols* (*ban-sol vs F*) =
sols F ∩
{ω. *map* ω (*map fst vs*) ≠ *map snd vs*}
⟨*proof*⟩

global-interpretation *CertCheck-CNF-XOR* :
CertCheck sols enc-xor check-sol ban-sol
defines
random-seed-xors = *CertCheck-CNF-XOR.random-seed-xors* **and**
fix-t = *CertCheck-CNF-XOR.appmc.fix-t* **and**
find-t = *CertCheck-CNF-XOR.find-t* **and**
BSAT = *CertCheck-CNF-XOR.BSAT* **and**
check-BSAT-sols = *CertCheck-CNF-XOR.check-BSAT-sols* **and**
size-xorL-cert = *CertCheck-CNF-XOR.size-xorL-cert* **and**
approxcore-xorsL = *CertCheck-CNF-XOR.approxcore-xorsL* **and**
fold-approxcore-xorsL-cert = *CertCheck-CNF-XOR.fold-approxcore-xorsL-cert*
and
approxcore-xorsL-cert = *CertCheck-CNF-XOR.approxcore-xorsL-cert*
and
calc-median = *CertCheck-CNF-XOR.calc-median* **and**
certcheck = *CertCheck-CNF-XOR.certcheck*
⟨*proof*⟩

8.1 Blasting XOR constraints to CNF

This formalizes the usual linear conversion from CNF-XOR into CNF. It is not necessary to use this conversion for solvers that support CNF-XOR formulas natively.

definition *negate-lit* :: *lit* ⇒ *lit*
  where *negate-lit l* = (*case l of Pos x* ⇒ *Neg x* | *Neg x* ⇒ *Pos x*)

fun *xor-clauses* :: *cmsxor* ⇒ *bool* ⇒ *clause list*
  where
    xor-clauses [] *b* = (*if b then* [[]] *else* [])
  | *xor-clauses* (*x*#*xs*) *b* =
    (*let p-x* = *xor-clauses xs b in*
     let n-x = *xor-clauses xs* (¬*b*) *in*
       map (λ*c*. *x* # *c*) *p-x* @ *map* (λ*c*. *negate-lit x* # *c*) *n-x*)

lemma *sat-cmsxor-nil*[*simp*]:
  shows¬ (*sat-cmsxor w* [])
  ⟨*proof*⟩

lemma *sat-cmsxor-cons*:
  shows *sat-cmsxor w* (*x* # *xs*) =
    (*if sat-lit w x then* ¬ (*sat-cmsxor w xs*) *else sat-cmsxor w xs*)
  ⟨*proof*⟩

lemma *sat-cmsxor-append*:
  shows *sat-cmsxor w* (*xs* @ *ys*) =
    (*if sat-cmsxor w xs then* ¬ (*sat-cmsxor w ys*) *else sat-cmsxor w ys*)
⟨*proof*⟩

definition *sat-clauses*:: *assignment* ⇒ *clause list* ⇒ *bool*
  where *sat-clauses w cs* = (∀ *c* ∈ *set cs*. *sat-clause w c*)

lemma *sat-clauses-append*:
  shows *sat-clauses w* (*xs* @ *ys*) =
    (*sat-clauses w xs* ∧ *sat-clauses w ys*)
  ⟨*proof*⟩

lemma *sat-clauses-map*:
  shows *sat-clauses w* (*map* ((#) *x*) *cs*) =
    (*sat-lit w x* ∨ *sat-clauses w cs*)
  ⟨*proof*⟩

lemma *sat-lit-negate-lit*[*simp*]:
  sat-lit w (*negate-lit l*) = (¬*sat-lit w l*)
  ⟨*proof*⟩

lemma *sols-xor-clauses*:

shows
sat-clauses w (*xor-clauses xs b*) $\longleftrightarrow$
(*sat-cmsxor w xs* = *b*)
⟨*proof*⟩

definition *var-lit* :: *lit* ⇒ *nat*
where *var-lit l* = (*case l of Pos x* ⇒ *x* | *Neg x* ⇒ *x*)

definition *var-lits* :: *lit list* ⇒ *nat*
where *var-lits ls* = *fold max* (*map var-lit ls*) *0*

lemma *sat-lit-same*:
assumes $\bigwedge x.\ x \leq$ *var-lit l* $\Longrightarrow$ *w x* = *w′ x*
shows *sat-lit w l* = *sat-lit w′ l*
⟨*proof*⟩

lemma *var-lits-eq*:
var-lits ls = *Max* (*set* (*0* # *map var-lit ls*))
⟨*proof*⟩

lemma *sat-lits-same*:
assumes $\bigwedge x.\ x \leq$ *var-lits c* $\Longrightarrow$ *w x* = *w′ x*
shows *sat-clause w c* = *sat-clause w′ c*
⟨*proof*⟩

lemma *le-var-lits-in*:
assumes *y* ∈ *set ys v* ≤ *var-lit y*
shows *v* ≤ *var-lits ys*
⟨*proof*⟩

lemma *sat-cmsxor-same*:
assumes $\bigwedge x.\ x \leq$ *var-lits xs* $\Longrightarrow$ *w x* = *w′ x*
shows *sat-cmsxor w xs* = *sat-cmsxor w′ xs*
⟨*proof*⟩

lemma *sat-cmsxor-split*:
assumes *u*: *var-lits xs* < *u var-lits ys* < *u*
assumes *w′*: *w′* = (λ*x*. *if x* = *u then* ¬ *sat-cmsxor w xs else w x*)
shows
(*sat-cmsxor w* (*xs* @ *ys*) =
(*sat-cmsxor w′* (*Pos u* # *xs*) ∧
sat-cmsxor w′ (*Neg u* # *ys*)))
⟨*proof*⟩

fun *split-xor* ::*nat* ⇒ *cmsxor* ⇒ *cmsxor list* × *nat* ⇒ *cmsxor list* × *nat*

where *split-xor k xs* (*acc*,*u*) = (
  if length xs ≤ *k* + *3 then* (*xs* # *acc*, *u*)
  else (
    let xs1 = *take* (*k* + *2*) *xs in*
    let xs2 = *drop* (*k* + *2*) *xs in*
    split-xor k (*Neg u* # *xs2*) ((*Pos u* # *xs1*) # *acc*, *u*+*1*)
)
)

declare *split-xor.simps*[*simp del*]

lemma *split-xor-bound*:
  assumes *split-xor k xs* (*acc*,*u*) = (*acc*′,*u*′)
  shows $u \le u'$
  ⟨*proof*⟩

lemma *var-lits-append*:
  shows *var-lits xs* ≤ *var-lits* (*xs* @ *ys*)
    var-lits ys ≤ *var-lits* (*xs* @ *ys*)
  ⟨*proof*⟩

lemma *fold-max-eq*:
  assumes $i \le u$
  shows *fold max ls u* = *max u* (*fold max ls* (*i*::*nat*))
  ⟨*proof*⟩

lemma *split-xor-sound*:
  assumes *sat-cmsxor w xs* ⋀*x*. *x* ∈ *set acc* ⟹ *sat-cmsxor w x*
  assumes *u*: *var-lits xs* < *u* ⋀*x*. *x* ∈ *set acc* ⟹ *var-lits x* < *u*
  assumes *split-xor k xs* (*acc*,*u*) = (*acc*′,*u*′)
  obtains *w*′ **where**
    ⋀*x*. *x* < *u* ⟹ *w x* = *w*′ *x*
    ⋀*x*. *x* ∈ *set acc*′ ⟹ *sat-cmsxor w*′ *x*
    ⋀*x*. *x* ∈ *set acc*′ ⟹ *var-lits x* < *u*′
  ⟨*proof*⟩

definition *split-xors* ::*nat* ⇒ *nat* ⇒ *cmsxor list* ⇒ *cmsxor list*
where *split-xors k u xs* = *fst* (*fold* (*split-xor k*) *xs* ([],*u*))

lemma *split-xors-sound*:
  assumes ⋀*x*. *x* ∈ *set xs* ⟹ *sat-cmsxor w x*
      ⋀*x*. *x* ∈ *set acc* ⟹ *sat-cmsxor w x*
  assumes *u*: ⋀*x*. *x* ∈ *set xs* ⟹ *var-lits x* < *u*
    ⋀*x*. *x* ∈ *set acc* ⟹ *var-lits x* < *u*
  assumes *fold* (*split-xor k*) *xs* (*acc*,*u*) = (*acc*′,*u*′)
  obtains *w*′ **where**
    ⋀*x*. *x* < *u* ⟹ *w x* = *w*′ *x*
    ⋀*x*. *x* ∈ *set acc*′ ⟹ *sat-cmsxor w*′ *x*

$\bigwedge x.\ x \in set\ acc' \Longrightarrow var\text{-}lits\ x < u'$
⟨*proof*⟩

definition *var-fml* :: *fml* ⇒ *nat*
where *var-fml f* =
max (*fold max* (*map var-lits* (*fst f*)) *0*)
(*fold max* (*map var-lits* (*snd f*)) *0*)

lemma *var-fml-eq*:
var-fml f =
max (*Max* (*set* (*0* # *map var-lits* (*fst f*))))
(*Max* (*set* (*0* # *map var-lits* (*snd f*))))
⟨*proof*⟩

definition *split-fml* :: *nat* ⇒ *fml* ⇒ *fml*
where *split-fml k f* = (
let u = *var-fml f* + *1 in*
(*fst f*, (*split-xors k u* (*snd f*)))
)

lemma *var-lits-var-fml*:
shows $\bigwedge x.\ x \in set\ (snd\ F) \Longrightarrow var\text{-}lits\ x \leq var\text{-}fml\ F$
$\bigwedge x.\ x \in set\ (fst\ F) \Longrightarrow var\text{-}lits\ x \leq var\text{-}fml\ F$
⟨*proof*⟩

lemma *split-fml-satisfies*:
assumes *sat-fml w F*
obtains *w'* **where** *sat-fml w'* (*split-fml k F*)
⟨*proof*⟩

lemma *split-fml-sols*:
assumes *sols* (*split-fml k F*) = {}
shows *sols F* = {}
⟨*proof*⟩

definition *blast-xors* :: *cmsxor list* ⇒ *clause list*
where *blast-xors xors* = *concat* (*map* (λ*x*. *xor-clauses x True*) *xors*)

definition *blast-fml* :: *fml* ⇒ *clause list*
where *blast-fml f* =
fst f @ *blast-xors* (*snd f*)

lemma *sat-clauses-concat*:
sat-clauses w (*concat xs*) ⟷
(∀ *x* ∈ *set xs*. *sat-clauses w x*)
⟨*proof*⟩

lemma *blast-xors-sound*:

assumes $(\bigwedge x.\ x \in set\ xors \Longrightarrow sat\text{-}cmsxor\ w\ x)$
shows $sat\text{-}clauses\ w\ (blast\text{-}xors\ xors)$
⟨*proof*⟩

lemma *blast-fml-sound*:
assumes $sat\text{-}fml\ w\ F$
shows $sat\text{-}fml\ w\ (blast\text{-}fml\ F,[])$
⟨*proof*⟩

definition $blast\text{-}split\text{-}fml :: fml \Rightarrow clause\ list$
where $blast\text{-}split\text{-}fml\ f = blast\text{-}fml\ (split\text{-}fml\ 1\ f)$

lemma *blast-split-fml-sols*:
assumes $sols\ (blast\text{-}split\text{-}fml\ F,[]) = \{\}$
shows $sols\ F = \{\}$
⟨*proof*⟩

definition *certcheck-blast*::
$(clause\ list \Rightarrow bool) \Rightarrow$
$fml \Rightarrow nat\ list \Rightarrow$
$real \Rightarrow real \Rightarrow$
$((nat \Rightarrow bool)\ list \times$
$(nat \Rightarrow (nat \times (nat \Rightarrow bool)\ list \times (nat \Rightarrow bool)\ list))) \Rightarrow$
$(nat \Rightarrow nat \Rightarrow (bool\ list \times bool)) \Rightarrow$
$String.literal + nat$
where $certcheck\text{-}blast\ check\text{-}unsat\ F\ S\ \varepsilon\ \delta\ m0ms =$
$certcheck\ (check\text{-}unsat \circ blast\text{-}split\text{-}fml)\ F\ S\ \varepsilon\ \delta\ m0ms$

corollary *certcheck-blast-sound*:
assumes $\bigwedge F.\ check\text{-}unsat\ F \Longrightarrow sols\ (F,\ []) = \{\}$
assumes $0 < \delta\ \delta < 1$
assumes $0 < \varepsilon$
assumes *distinct S*
shows
$measure\text{-}pmf.prob$
$(map\text{-}pmf\ (\lambda r.\ certcheck\text{-}blast\ check\text{-}unsat\ F\ S\ \varepsilon\ \delta\ (f\ r)\ r)$
$(random\text{-}seed\text{-}xors\ (find\text{-}t\ \delta)\ (length\ S)))$
$\{c.\ \neg\ isl\ c\ \wedge$
$real\ (projr\ c) \notin$
$\{real\ (card\ (proj\ (set\ S)\ (sols\ F)))\ /\ (1 + \varepsilon)..$
$(1 + \varepsilon) * real\ (card\ (proj\ (set\ S)\ (sols\ F)))\}\} \leq \delta$
⟨*proof*⟩

corollary *certcheck-blast-promise-complete*:
assumes $\bigwedge F.\ check\text{-}unsat\ F \Longrightarrow sols\ (F,\ []) = \{\}$
assumes $0 < \delta\ \delta < 1$
assumes $0 < \varepsilon$
assumes *distinct S*

assumes *r*: $\bigwedge r$.
  $r \in$ *set-pmf* (*random-seed-xors* (*find-t* δ) (*length* *S*)) $\Longrightarrow$
  $\neg$*isl* (*certcheck-blast check-unsat F S* ε δ (*f r*) *r*)
shows
  measure-pmf.prob
    (*map-pmf* (λr. *certcheck-blast check-unsat F S* ε δ (*f r*) *r*)
      (*random-seed-xors* (*find-t* δ) (*length S*)))
    {*c*. *real* (*projr c*) $\in$
      {*real* (*card* (*proj* (*set S*) (*sols F*))) / (*1* + ε)..
        (*1* + ε) $*$ *real* (*card* (*proj* (*set S*) (*sols F*)))}} $\geq$ *1* $-$ δ
⟨*proof*⟩

8.2 Export code for a SML implementation.

definition *real-of-int* :: *integer* $\Rightarrow$ *real*
  where *real-of-int n* = *real* (*nat-of-integer n*)

definition *real-mult* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real*
  where *real-mult n m* = *n* $*$ *m*

definition *real-div* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real*
  where *real-div n m* = *n* / *m*

definition *real-plus* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real*
  where *real-plus n m* = *n* + *m*

definition *real-minus* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real*
  where *real-minus n m* = *n* $-$ *m*

declare [[*code abort*: *fix-t*]]

export-code
  length
  nat-of-integer int-of-integer
  integer-of-nat integer-of-int
  real-of-int real-mult real-div real-plus real-minus
  quotient-of

  Pos Neg
  CertCheck-CNF-XOR.appmc.compute-thresh
  find-t certcheck
  certcheck-blast
  in *SML*

end

References

[1] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In S. Kambhampati, editor, *IJCAI*, pages 3569–3576. IJCAI/AAAI Press, 2016.