

Approximate Model Counting

Yong Kiam Tan and Jiong Yang

March 23, 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	2
2	Random XORs	4
2.1	Independence properties of random XORs	6
2.2	Independence for repeated XORs	10
3	Random XOR hash family	13
4	ApproxMCCore definitions	15
5	ApproxMCCore analysis	21
6	ApproxMC definition and analysis	26
7	Certificate-based ApproxMC	31
7.1	ApproxMC with lists instead of sets	31
7.2	ApproxMC certificate checker	37
8	ApproxMC certification for CNF-XOR	44
8.1	Blasting XOR constraints to CNF	46
8.2	Export code for a SML implementation.	52

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.Paley-Zygmund-Inequality
begin

lemma *card-inter-sum-indicat-real*:
assumes *finite A*
shows $\text{card } (A \cap B) = \text{sum } (\text{indicat-real } B) A$
<proof>

lemma *card-dom-ran*:
assumes *finite D*
shows $\text{card } \{w. \text{dom } w = D \wedge \text{ran } w \subseteq R\} = \text{card } R \wedge \text{card } D$
<proof>

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

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

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

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*
 $(\lambda s. \text{real } (\text{card } (S \cap \{w. H w s = \alpha\}))) \leq$
measure-pmf.expectation *p*
 $(\lambda s. \text{real } (\text{card } (S \cap \{w. H w s = \alpha\})))$
 <proof>

lemma (*in* *prob-space*) *k-universal-mono:*

assumes $k' \leq k$
assumes *k-universal* *k H D R*
shows *k-universal* $k' H D R$
 <proof>

lemma *finite-set-pmf-expectation-add:*

assumes *finite* (*set-pmf* *S*)
shows *measure-pmf.expectation* *S* $(\lambda x. ((f x)::\text{real}) + g x) =$
measure-pmf.expectation *S* *f* + *measure-pmf.expectation* *S* *g*
 <proof>

lemma *finite-set-pmf-expectation-add-const:*

assumes *finite* (*set-pmf* *S*)
shows *measure-pmf.expectation* *S* $(\lambda x. ((f x)::\text{real}) + g) =$
measure-pmf.expectation *S* *f* + *g*
 <proof>

lemma *finite-set-pmf-expectation-diff:*

assumes *finite* (*set-pmf* *S*)
shows *measure-pmf.expectation* *S* $(\lambda x. ((f x)::\text{real}) - g x) =$
measure-pmf.expectation *S* *f* - *measure-pmf.expectation* *S* *g*
 <proof>

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

$$\begin{aligned}
& (\text{measure-pmf.variance } p \ Z + (1-\vartheta)^2 * (\text{measure-pmf.expectation } \\
& p \ Z)^2) * \\
& \text{measure-pmf.prob } p \ \{z. \ Z \ z > \vartheta * \text{measure-pmf.expectation } p \ Z\} \\
& \geq \\
& (1-\vartheta)^2 * (\text{measure-pmf.expectation } p \ Z)^2 \\
\langle \text{proof} \rangle
\end{aligned}$$

lemma *spec-chebyshev-inequality*:

assumes *fin*: *finite (set-pmf p)*

assumes *pvar*: *measure-pmf.variance p Y > 0*

assumes *k*: *k > 0*

shows

measure-pmf.prob p

$\{y. (Y \ y - \text{measure-pmf.expectation } p \ Y)^2 \geq$
 $k^2 * \text{measure-pmf.variance } p \ Y\} \leq 1 / k^2$

$\langle \text{proof} \rangle$

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

Frequency-Moments.Product-PMF-Ext

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 ($\lambda b. \{x \in V. b \ x = \text{Some True}\}$)

(Pi-pmf V def ($\lambda \cdot. \text{map-pmf Some (bernoulli-pmf (1 / 2))$))

<proof>

lemma *random-xor-from-bits:*

assumes $V:finite\ V$

shows $random-xor\ V =$

pair-pmf

$(map-pmf\ (\lambda b. \{x \in V. b\ x = Some\ True\}))$

$(Pi-pmf\ V\ def\ (\lambda-. map-pmf\ Some\ (bernoulli-pmf\ (1/2))))$

$(bernoulli-pmf\ (1/2))$

<proof>

fun *satisfies-xor* :: $('a\ set \times bool) \Rightarrow 'a\ set \Rightarrow bool$

where *satisfies-xor* $(x,b)\ \omega =$

even $(card\ (\omega \cap x) + of-bool\ b)$

lemma *satisfies-xor-inter:*

shows $satisfies-xor\ (\omega \cap x, b)\ \omega = satisfies-xor\ (x, b)\ \omega$

<proof>

lemma *prob-bernoulli-bind-pmf:*

assumes $0 \leq p \leq 1$

assumes *finite E*

shows *measure-pmf.prob*

$(bernoulli-pmf\ p \gg x)\ E =$

$p * (measure-pmf.prob\ (x\ True)\ E) +$

$(1 - p) * (measure-pmf.prob\ (x\ False)\ E)$

<proof>

lemma *set-pmf-random-xor:*

assumes $V: finite\ V$

shows $set-pmf\ (random-xor\ V) = (Pow\ V) \times UNIV$

<proof>

lemma *pmf-of-set-prod:*

assumes $P \neq \{\}\ Q \neq \{\}$

assumes *finite P finite Q*

shows $pmf-of-set\ (P \times Q) = pair-pmf\ (pmf-of-set\ P)\ (pmf-of-set\ Q)$

<proof>

lemma *random-xor-pmf-of-set:*

assumes $V:finite\ V$

shows $random-xor\ V = pmf-of-set\ ((Pow\ V) \times UNIV)$

<proof>

lemma *prob-random-xor-with-set-pmf*:
assumes V : *finite* V
shows $\text{prob-space.prob (random-xor } V) \{c. P\ c\} =$
 $\text{prob-space.prob (random-xor } V) \{c. \text{fst } c \subseteq V \wedge P\ c\}$
 $\langle \text{proof} \rangle$

lemma *prob-set-parity*:
assumes $\text{measure-pmf.prob } M$
 $\{c. P\ c\} = q$
shows $\text{measure-pmf.prob } M$
 $\{c. P\ c = b\} = (\text{if } b \text{ then } q \text{ else } 1 - q)$
 $\langle \text{proof} \rangle$

lemma *satisfies-random-xor*:
assumes V : *finite* V
shows $\text{prob-space.prob (random-xor } V)$
 $\{c. \text{satisfies-xor } c\ \omega\} = 1 / 2$
 $\langle \text{proof} \rangle$

lemma *satisfies-random-xor-parity*:
assumes V : *finite* V
shows $\text{prob-space.prob (random-xor } V)$
 $\{c. \text{satisfies-xor } c\ \omega = b\} = 1 / 2$
 $\langle \text{proof} \rangle$

2.1 Independence properties of random XORs

lemma *pmf-of-set-powerset-split*:
assumes $S \subseteq V$ *finite* V
shows
 $\text{map-pmf } (\lambda(x,y). x \cup y)$
 $(\text{pmf-of-set (Pow } S \times \text{Pow (} V - S))) =$
 $\text{pmf-of-set (Pow } V)$
 $\langle \text{proof} \rangle$

lemma *pmf-of-set-Pow-sing*:
shows $\text{pmf-of-set (Pow } \{x\}) =$
 $\text{bernoulli-pmf (} 1 / 2 \text{)} \gg$
 $(\lambda b. \text{return-pmf (if } b \text{ then } \{x\} \text{ else } \{\}))$
 $\langle \text{proof} \rangle$

lemma *pmf-of-set-sing-coin-flip*:
assumes *finite* V
shows $\text{pmf-of-set (Pow } \{x\} \times \text{Pow } V) =$
 $\text{map-pmf } (\lambda(r,c). (\text{if } c \text{ then } \{x\} \text{ else } \{\}, r)) (\text{random-xor } V)$
 $\langle \text{proof} \rangle$
including *monad-normalisation*
 $\langle \text{proof} \rangle$

lemma *measure-pmf-prob-dependent-product-bound-eq*:
assumes *countable A* \wedge *i. countable (B i)*
assumes $\wedge a. a \in A \implies \text{measure-pmf.prob } N (B a) = r$
shows $\text{measure-pmf.prob (pair-pmf } M N) (\text{Sigma } A B) =$
 $\text{measure-pmf.prob } M A * r$
 $\langle \text{proof} \rangle$

lemma *measure-pmf-prob-dependent-product-bound-eq'*:
assumes *countable (A \cap set-pmf M)* \wedge *i. countable (B i \cap set-pmf N)*
assumes $\wedge a. a \in A \cap \text{set-pmf } M \implies \text{measure-pmf.prob } N (B a \cap \text{set-pmf } N) = r$
shows $\text{measure-pmf.prob (pair-pmf } M N) (\text{Sigma } A B) = \text{measure-pmf.prob } M A * r$
 $\langle \text{proof} \rangle$

lemma *single-var-parity-coin-flip*:
assumes *x \in ω finite ω*
assumes *finite a x \notin a*
shows $\text{measure-pmf.prob (pmf-of-set (Pow \{x\}))}$
 $\{y. \text{even (card ((a \cup y) \cap ω))} = b\} = 1/2$
 $\langle \text{proof} \rangle$

lemma *prob-pmf-of-set-nonempty-parity*:
assumes *V: finite V*
assumes *x \in ω $\omega \subseteq V$*
assumes $\wedge c. c \in E \longleftrightarrow c - \{x\} \in E$
shows $\text{prob-space.prob (pmf-of-set (Pow V))}$
 $(E \cap \{c. \text{even (card (c \cap ω))} = b\}) =$
 $1 / 2 * \text{prob-space.prob (pmf-of-set (Pow (V - \{x\}))) } E$
 $\langle \text{proof} \rangle$

lemma *prob-random-xor-split*:
assumes *V: finite V*
shows $\text{prob-space.prob (random-xor V) } E =$
 $1 / 2 * \text{prob-space.prob (pmf-of-set (Pow V)) } \{e. (e, \text{True}) \in E\} +$
 $1 / 2 * \text{prob-space.prob (pmf-of-set (Pow V)) } \{e. (e, \text{False}) \in E\}$
 $\langle \text{proof} \rangle$

lemma *prob-random-xor-nonempty-parity*:
assumes *V: finite V*
assumes $\omega: x \in \omega \omega \subseteq V$
assumes *E: $\wedge c. c \in E \longleftrightarrow (\text{fst } c - \{x\}, \text{snd } c) \in E$*
shows $\text{prob-space.prob (random-xor V)}$
 $(E \cap \{c. \text{satisfies-xor } c \omega = b\}) =$
 $1 / 2 * \text{prob-space.prob (random-xor (V - \{x\})) } E$

\langle proof \rangle

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)

$\{c. \text{satisfies-xor } c \ \omega = b \wedge \text{satisfies-xor } c \ \omega' = b'\} = 1 / 4$

\langle proof \rangle

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)

$\{c. \text{satisfies-xor } c \ \omega = b \wedge \text{satisfies-xor } c \ \omega' = b'\} = 1 / 4$

\langle proof \rangle

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))

$\{c. \text{even } (\text{card } (c \cap \omega)) = b\} = 1 / 2$

\langle proof \rangle

lemma *prob-Pow-split:*

assumes $\omega \subseteq V$ finite V

shows *prob-space.prob* (*pmf-of-set* (*Pow* V))

$\{x. P \ (\omega \cap x) \wedge Q \ ((V - \omega) \cap x)\} =$

prob-space.prob (*pmf-of-set* (*Pow* ω))

$\{x. P \ x\} *$

prob-space.prob (*pmf-of-set* (*Pow* ($V - \omega$))))

$\{x. Q \ x\}$

\langle proof \rangle

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))

$\{c. \text{even } (\text{card } (\omega \cap c)) = b \wedge \text{even } (\text{card } (\omega' \cap c)) = b'\} = 1 / 4$

\langle proof \rangle

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 $M \ A * \text{measure-pmf.prob } N \ B$

\langle proof \rangle

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* ($V - \omega$)))
 $\{x. Q x\}$
 \langle *proof* \rangle
including *monad-normalisation*
 \langle *proof* \rangle

lemma *three-disjoint-prob-random-xor-nonempty*:

assumes $\omega: \omega \neq \{\}$ $\omega \subseteq V$
assumes $\omega': \omega' \neq \{\}$ $\omega' \subseteq V$
assumes $I: I \subseteq V$
assumes *int*: $I \cap \omega = \{\}$ $I \cap \omega' = \{\}$ $\omega \cap \omega' = \{\}$
assumes $V: \textit{finite } V$
shows *prob-space.prob* (*random-xor* V)
 $\{c. \textit{satisfies-xor } c I = b \wedge$
 $\textit{even} (\textit{card} (\omega \cap \textit{fst } c)) = b' \wedge$
 $\textit{even} (\textit{card} (\omega' \cap \textit{fst } c)) = b''\} = 1 / 8$
 \langle *proof* \rangle

lemma *three-disjoint-prob-pmf-of-set-nonempty*:

assumes $\omega: x \in \omega$ $\omega \subseteq V$
assumes $\omega': x' \in \omega'$ $\omega' \subseteq V$
assumes $\omega'': x'' \in \omega''$ $\omega'' \subseteq V$
assumes *int*: $\omega \cap \omega' = \{\}$ $\omega' \cap \omega'' = \{\}$ $\omega'' \cap \omega = \{\}$
assumes $V: \textit{finite } V$
shows *prob-space.prob* (*pmf-of-set* (*Pow* V))
 $\{c. \textit{even} (\textit{card} (\omega \cap c)) = b \wedge \textit{even} (\textit{card} (\omega' \cap c)) = b' \wedge \textit{even}$
 $(\textit{card} (\omega'' \cap c)) = b''\} = 1 / 8$
 \langle *proof* \rangle

lemma *four-disjoint-prob-random-xor-nonempty*:

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

$even (card (\omega'' \cap fst\ c)) = b''\} = 1 / 16$
 <proof>

lemma *three-satisfies-random-xor-parity-1*:

assumes $V:finite\ V$
assumes $\omega: \omega \subseteq V\ \omega' \subseteq V\ \omega'' \subseteq V$
assumes $x: x \notin \omega\ x \notin \omega'\ x \in \omega''$
assumes $d: \omega \neq \omega'$
shows $prob\ space.\ prob\ (random\ xor\ V)$
 $\{c.$
 $\text{satisfies-xor } c\ \omega = b \wedge$
 $\text{satisfies-xor } c\ \omega' = b' \wedge$
 $\text{satisfies-xor } c\ \omega'' = b''\} = 1 / 8$

<proof>

lemma *split-boolean-eq*:

shows $(A \longleftrightarrow B) = (b \longleftrightarrow I) \wedge$
 $(B \longleftrightarrow C) = (b' \longleftrightarrow I) \wedge$
 $(C \longleftrightarrow A) = (b'' \longleftrightarrow I)$
 \longleftrightarrow
 $I = odd(of\ bool\ b + of\ bool\ b' + of\ bool\ b'') \wedge$
 $(A = True \wedge$
 $B = (b' = b'') \wedge$
 $C = (b = b') \vee$
 $A = False \wedge$
 $B = (b' \neq b'') \wedge$
 $C = (b \neq b'))$

<proof>

lemma *three-satisfies-random-xor-parity*:

assumes $V:finite\ V$
assumes $\omega:$
 $\omega \neq \omega'\ \omega \neq \omega''\ \omega' \neq \omega''$
 $\omega \subseteq V\ \omega' \subseteq V\ \omega'' \subseteq V$
shows $prob\ space.\ prob\ (random\ xor\ V)$
 $\{c.\ \text{satisfies-xor } c\ \omega = b \wedge$
 $\text{satisfies-xor } c\ \omega' = b' \wedge$
 $\text{satisfies-xor } c\ \omega'' = b''\} = 1 / 8$

<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 \Rightarrow nat \Rightarrow

$(nat \rightarrow 'a\ set \times bool)\ pmf$

where *random-xors* $V\ n =$

$Pi\ pmf\ \{..<(n::nat)\}\ None$

$(\lambda\cdot.\ map\ pmf\ Some\ (random\ xor\ V))$

lemma *random-xors-set*:

assumes $V:finite\ V$

shows

$PiE\text{-dflt}\ \{..\lt n\}\ None$

$(set\text{-pmf}\ \circ\ (\lambda\cdot. map\text{-pmf}\ Some\ (random\text{-xor}\ V))) =$

$\{xors.\ dom\ xors = \{..\lt n\} \wedge$

$ran\ xors \subseteq (Pow\ V) \times UNIV\}$ (**is** $?lhs = ?rhs$)

$\langle proof \rangle$

lemma *random-xors-eq*:

assumes $V:finite\ V$

shows $random\text{-xors}\ V\ n =$

$pmf\text{-of}\text{-set}$

$\{xors.\ dom\ xors = \{..\lt n\} \wedge ran\ xors \subseteq (Pow\ V) \times UNIV\}$

$\langle proof \rangle$

definition *xor-hash* ::

$('a \rightarrow bool) \Rightarrow$

$(nat \rightarrow ('a\ set \times bool)) \Rightarrow$

$(nat \rightarrow bool)$

where $xor\text{-hash}\ \omega\ xors =$

$(map\text{-option}$

$(\lambda xor.\ satisfies\text{-xor}\ xor\ \{x.\ \omega\ x = Some\ True\}) \circ xors)$

lemma *finite-map-set-nonempty*:

assumes $R \neq \{\}$

shows

$\{xors.\$

$dom\ xors = D \wedge ran\ xors \subseteq R\} \neq \{\}$

$\langle proof \rangle$

lemma *random-xors-set-pmf*:

assumes $V:finite\ V$

shows

$set\text{-pmf}\ (random\text{-xors}\ V\ n) =$

$\{xors.\ dom\ xors = \{..\lt n\} \wedge$

$ran\ xors \subseteq (Pow\ V) \times UNIV\}$

$\langle proof \rangle$

lemma *finite-random-xors-set-pmf*:

assumes $V:finite\ V$

shows

$finite\ (set\text{-pmf}\ (random\text{-xors}\ V\ n))$

$\langle proof \rangle$

lemma *map-eq-1*:

assumes $\text{dom } f = \text{dom } g$
assumes $\bigwedge x. x \in \text{dom } f \implies \text{the } (f x) = \text{the } (g x)$
shows $f = g$
 <proof>

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

lemma *xor-hash-eq-PiE-dfft*:
assumes $\text{dom } \alpha = \{..<n\}$
shows
 $\{\text{xors}. \text{xor-hash } \omega \text{ xors} = \alpha\} =$
 $\text{PiE-dfft } \{..<n\} \text{ None}$
 $(\lambda i. \text{Some } ')$
 $\{\text{xor}. \text{satisfies-xor } \text{xor } \{x. \omega x = \text{Some } \text{True}\} = \text{the } (\alpha i)\}$
 <proof>

lemma *prob-random-xors-xor-hash*:
assumes $V: \text{finite } V$
assumes $\alpha: \text{dom } \alpha = \{..<n\}$
shows
 $\text{measure-pmf.prob } (\text{random-xors } V n)$
 $\{\text{xors}. \text{xor-hash } \omega \text{ xors} = \alpha\} = 1 / 2 \wedge n$
 <proof>

lemma *PiE-dfft-inter*:
shows $\text{PiE-dfft } A \text{ dfft } B \cap \text{PiE-dfft } A \text{ dfft } B' =$
 $\text{PiE-dfft } A \text{ dfft } (\lambda b. B b \cap B' b)$
 <proof>

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

```

lemma random-xors-xor-hash-three:
  assumes  $V$ : finite  $V$ 
  assumes  $\alpha$ :  $\text{dom } \alpha = \{..<n\}$ 
  assumes  $\alpha'$ :  $\text{dom } \alpha' = \{..<n\}$ 
  assumes  $\alpha''$ :  $\text{dom } \alpha'' = \{..<n\}$ 
  assumes  $\omega$ :  $\text{dom } \omega = V$ 
  assumes  $\omega'$ :  $\text{dom } \omega' = V$ 
  assumes  $\omega''$ :  $\text{dom } \omega'' = V$ 
  assumes neq:  $\omega \neq \omega' \wedge \omega' \neq \omega'' \wedge \omega'' \neq \omega$ 
  shows
     $\text{measure-pmf.prob } (\text{random-xors } V \ n)$ 
       $\{xors.$ 
         $\text{xor-hash } \omega \ xors = \alpha$ 
         $\wedge \text{xor-hash } \omega' \ xors = \alpha'$ 
         $\wedge \text{xor-hash } \omega'' \ xors = \alpha''\} =$ 
         $1 / 8 \wedge n$ 
   $\langle\text{proof}\rangle$ 
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  $\{w :: 'a \rightarrow \text{bool. dom } w = V\}$ 
   $\langle\text{proof}\rangle$ 

```

```

lemma xor-hash-eq-dom:
  assumes  $\text{xor-hash } \omega \ xors = \alpha$ 
  shows  $\text{dom } xors = \text{dom } \alpha$ 
   $\langle\text{proof}\rangle$ 

```

```

lemma prob-random-xors-xor-hash-indicat-real:
  assumes  $V$ : finite  $V$ 
  shows
     $\text{measure-pmf.prob } (\text{random-xors } V \ n)$ 
       $\{xors. \text{xor-hash } \omega \ xors = \alpha\} =$ 
       $\text{indicat-real } \{\alpha :: \text{nat} \rightarrow \text{bool. dom } \alpha = \{0..<n\}\} \alpha /$ 
       $\text{real } (\text{card } \{\alpha :: \text{nat} \rightarrow \text{bool. dom } \alpha = \{0..<n\}\})$ 
   $\langle\text{proof}\rangle$ 

```

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

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

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

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

$\{xors.$
 $\quad xor\text{-hash } \omega'' \ xors = \alpha''\}$
 $\langle proof \rangle$

lemma *xor-hash-3-indep:*

assumes V : *finite* V

assumes J : $card\ J \leq 3 \ J \subseteq \{\alpha. dom\ \alpha = V\}$

shows

$measure\text{-pmf}.\text{prob } (random\text{-xors } V\ n)$
 $\{xors. \forall \omega \in J. xor\text{-hash } \omega \ xors = f\ \omega\} =$
 $(\prod_{\omega \in J}.$
 $\quad measure\text{-pmf}.\text{prob } (random\text{-xors } V\ n)$
 $\quad \{xors. xor\text{-hash } \omega \ xors = f\ \omega\})$

$\langle proof \rangle$

lemma *xor-hash-3-wise-indep:*

assumes *finite* V

shows *prob-space.k-wise-indep-vars*

$(random\text{-xors } V\ n)\ 3$

$(\lambda-. Universal\text{-Hash-Families-More-Independent-Families}.\text{discrete})$

xor-hash

$\{\alpha. dom\ \alpha = V\}$

$\langle proof \rangle$

theorem *xor-hash-family-3-universal:*

assumes *finite* V

shows *prob-space.k-universal*

$(random\text{-xors } V\ n)\ 3\ xor\text{-hash}$

$\{\alpha::'a \rightarrow bool. dom\ \alpha = V\}$

$\{\alpha::nat \rightarrow bool. dom\ \alpha = \{0..<n\}\}$

$\langle proof \rangle$

corollary *xor-hash-family-2-universal:*

assumes *finite* V

shows *prob-space.k-universal*

$(random\text{-xors } V\ n)\ 2\ xor\text{-hash}$

$\{\alpha::'a \rightarrow bool. dom\ \alpha = V\}$

$\{\alpha::nat \rightarrow bool. dom\ \alpha = \{0..<n\}\}$

$\langle proof \rangle$

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 non-probabilistic 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  $\rightarrow$  bool
```

```
definition restr :: 'a set  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a assg
  where restr S w = ( $\lambda x$ . if x  $\in$  S then Some (w x) else None)
```

```
lemma restrict-eq-mono:
  assumes x  $\subseteq$  y
  assumes f |' y = g |' y
  shows f |' x = g |' x
   $\langle$ proof $\rangle$ 
```

```
definition proj :: 'a set  $\Rightarrow$  ('a  $\Rightarrow$  bool) set  $\Rightarrow$  '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)  $\leq$  2  $\wedge$  card S
   $\langle$ proof $\rangle$ 
```

```
lemma proj-mono:
  assumes x  $\subseteq$  y
  shows proj w x  $\subseteq$  proj w y
   $\langle$ proof $\rangle$ 
```

```
definition aslice :: nat  $\Rightarrow$  nat assg  $\Rightarrow$  nat assg
  where aslice i a = a |' {..i}
```

```
lemma aslice-eq:
  assumes i  $\geq$  n
  assumes dom a = {..n}
  shows aslice i a = aslice n a
   $\langle$ proof $\rangle$ 
```

```
definition hslice :: nat  $\Rightarrow$ 
  ('a assg  $\Rightarrow$  nat assg)  $\Rightarrow$  ('a assg  $\Rightarrow$  nat assg)
  where hslice i h = aslice i  $\circ$  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) / 2i

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  $\bigwedge w. \text{dom } (h w) = \{..n\}$ 
  shows hslice i h = hslice n h
  ⟨proof⟩

lemma card-slice-lim:
  assumes card S - 1 ≤ i
  assumes  $\bigwedge w. \text{dom } (h w) = \{..(card S - 1)\}$ 
  shows card-slice h i = card-slice h (card S - 1)
  ⟨proof⟩

definition T :: nat ⇒
  ('a assg ⇒ nat assg) set
  where T i = {h. card-slice h i < thresh}

```

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

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

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

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

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

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

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

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

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 + ε) ..
 (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** $m \leq i$ $P\ m \neg P\ (m-1)$
 <proof>

lemma *mstar-non-zero*:
shows $\neg \mu 0 * (1 + \varepsilon / (1 + \varepsilon)) \leq \text{thresh}$
 $\langle \text{proof} \rangle$

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

lemma *mstar-exists*:
obtains m **where**
 $\mu (m - 1) * (1 + \varepsilon / (1 + \varepsilon)) > \text{thresh}$
 $\mu m * (1 + \varepsilon / (1 + \varepsilon)) \leq \text{thresh}$
 $m \leq \text{card } S - 1$
 $\langle \text{proof} \rangle$

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

lemma *mstar-prop*:
shows
 $\mu (mstar - 1) * (1 + \varepsilon / (1 + \varepsilon)) > \text{thresh}$
 $\mu mstar * (1 + \varepsilon / (1 + \varepsilon)) \leq \text{thresh}$
 $mstar \leq \text{card } S - 1$
 $\langle \text{proof} \rangle$

lemma *O1-lem*:
assumes $i \leq m$
shows $(T i - T (i-1)) \cap (L i \cup U i) \subseteq T m$
 $\langle \text{proof} \rangle$

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

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

lemma *O2-lem*:
assumes $mstar < i$

shows $(T\ i - T\ (i-1)) \cap (L\ i \cup U\ i) \subseteq -T\ mstar$
 $\langle proof \rangle$

lemma *O2*:

shows $(\bigcup_{i \in \{mstar..<card\ S\}} (T\ i - T\ (i-1)) \cap (L\ i \cup U\ i)) \cup$
 $-T\ (card\ S - 1) \subseteq L\ mstar \cup U\ mstar$
 $\langle proof \rangle$

lemma *O3*:

assumes $i \leq mstar - 1$
shows $(T\ i - T\ (i-1)) \cap (L\ i \cup U\ i) \subseteq L\ i$
 $\langle proof \rangle$

lemma *union-split-lem*:

assumes $x: x \in (\bigcup_{i \in \{1..<n::nat\}}. P\ i)$
shows $x \in (\bigcup_{i \in \{1..m-3\}}. P\ i) \cup$
 $P\ (m-2) \cup$
 $P\ (m-1) \cup$
 $(\bigcup_{i \in \{m..<n\}}. P\ i)$
 $\langle proof \rangle$

lemma *union-split*:

$(\bigcup_{i \in \{1..<n::nat\}}. P\ i) \subseteq$
 $(\bigcup_{i \in \{1..m-3\}}. P\ i) \cup$
 $P\ (m-2) \cup$
 $P\ (m-1) \cup$
 $(\bigcup_{i \in \{m..<n\}}. P\ i)$
 $\langle proof \rangle$

lemma *failure-bound*:

shows *approxcore-fail* \subseteq
 $T\ (mstar-3) \cup$
 $L\ (mstar-2) \cup$
 $L\ (mstar-1) \cup$
 $(L\ mstar \cup U\ mstar)$
 $\langle proof \rangle$

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* \Rightarrow
 ('a *assg* \Rightarrow 'b \Rightarrow *nat assg*) \Rightarrow ('a *assg* \Rightarrow 'b \Rightarrow *nat assg*)
where *Hslice* *i* *H* = ($\lambda w s. aslice\ i\ (H\ w\ s)$)

context *prob-space*
begin

lemma *indep-vars-prefix*:
assumes *indep-vars* ($\lambda-. count-space\ UNIV$) *H* *J*
shows *indep-vars* ($\lambda-. count-space\ UNIV$) (*Hslice* *i* *H*) *J*
 <proof>

lemma *assg-nonempty-dom*:
shows
 ($\lambda x. if\ x < i\ then\ Some\ True\ else\ None$) \in
 { $\alpha::nat\ assg. dom\ \alpha = \{0..<i\}$ }
 <proof>

lemma *card-dom-ran-nat-assg*:
shows *card* { $\alpha::nat\ assg. dom\ \alpha = \{0..<n\}$ } = $2^{\wedge}n$
 <proof>

lemma *card-nat-assg-le*:
assumes $i \leq n$
shows *card* { $\alpha::nat\ assg. dom\ \alpha = \{0..<n\}$ } =
 $2^{\wedge}(n-i) * card\ \{\alpha::nat\ assg. dom\ \alpha = \{0..<i\}\}$
 <proof>

lemma *empty-nat-assg-slice-notin*:
assumes $i \leq n$
assumes $dom\ \beta \neq \{0..<i\}$
shows { $\alpha::nat\ assg. dom\ \alpha = \{0..<n\} \wedge aslice\ i\ \alpha = \beta$ } = {}
 <proof>

lemma *restrict-map-dom*:
shows $\alpha \upharpoonright' dom\ \alpha = \alpha$
 <proof>

lemma *aslice-refl*:
assumes $dom\ \alpha = \{..<i\}$
shows *aslice* *i* $\alpha = \alpha$
 <proof>

lemma *bij-betw-with-inverse*:
assumes $f \upharpoonright' A \subseteq B$

assumes $\bigwedge x. x \in A \implies g (f x) = x$
assumes $g ' B \subseteq A$
assumes $\bigwedge x. x \in B \implies f (g x) = x$
shows *bij-betw f A B*
 <proof>

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

lemma *finite-dom*:
assumes *finite V*
shows *finite {w :: 'a \rightarrow bool. dom w = V}*
 <proof>

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

end

context *ApproxMCCore*
begin

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

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

lemma *aux-1*:
assumes $\text{fin}: \text{finite } (\text{set-pmf } p)$
assumes $\sigma: \sigma > 0$
assumes $\text{exp}: \mu i = \text{measure-pmf.expectation } p Y$
assumes $\text{var}: \sigma^{\wedge 2} = \text{measure-pmf.variance } p Y$
assumes $\text{var-bound}: \sigma^{\wedge 2} \leq \mu i$
shows
 $\text{measure-pmf.prob } p \{y. | Y y - \mu i | \geq \varepsilon / (1 + \varepsilon) * \mu i\}$
 $\leq (1 + \varepsilon)^{\wedge 2} / (\varepsilon^{\wedge 2} * \mu i)$
 <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
 $measure-pmf.prob\ p$
 $\{s. | card-slice\ ((\lambda w. H\ w\ s))\ i - \mu\ i | \geq \varepsilon / (1 + \varepsilon) * \mu\ i\}$
 $\leq (1 + \varepsilon)^2 / (\varepsilon^2 * \mu\ i)$
 $\langle proof \rangle$

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
 $\{s. real(card-slice\ ((\lambda w. H\ w\ s))\ i) \leq \beta * \mu\ i\}$
 $\leq 1 / (1 + (1 - \beta)^2 * \mu\ i)$
 $\langle proof \rangle$

lemma shift- μ :
assumes $k \leq i$
shows $\mu\ i * 2^k = \mu\ (i-k)$
 $\langle proof \rangle$

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
 $measure-pmf.prob\ (map-pmf\ (\lambda s\ w. H\ w\ s)\ p)\ (T\ (mstar-3))$
 $\leq 1 / 62.5$
 $\langle proof \rangle$

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.* $dom\ \alpha = \{0..<card\ S - 1\}\}$

shows
 $measure\text{-}pmf.\text{prob} (\text{map}\text{-}pmf (\lambda s w. H w s) p) (T (mstar-3))$
 $\leq 1 / 10.84$
 $\langle proof \rangle$

lemma analysis-2-2:
assumes $p: finite (set\text{-}pmf p)$
assumes $ind: prob\text{-}space.k\text{-universal} (measure\text{-}pmf p) 2 H$
 $\{\alpha::'a \text{ assg. } dom \alpha = S\}$
 $\{\alpha::nat \text{ assg. } dom \alpha = \{0..<card S - 1\}\}$
shows
 $measure\text{-}pmf.\text{prob} (\text{map}\text{-}pmf (\lambda s w. H w s) p) (L (mstar-2)) \leq 1$
 $/ 20.68$
 $\langle proof \rangle$

lemma analysis-2-3:
assumes $p: finite (set\text{-}pmf p)$
assumes $ind: prob\text{-}space.k\text{-universal} (measure\text{-}pmf p) 2 H$
 $\{\alpha::'a \text{ assg. } dom \alpha = S\}$
 $\{\alpha::nat \text{ assg. } dom \alpha = \{0..<card S - 1\}\}$
shows
 $measure\text{-}pmf.\text{prob} (\text{map}\text{-}pmf (\lambda s w. H w s) p) (L (mstar-1)) \leq 1 / 10.84$
 $\langle proof \rangle$

lemma analysis-2-4:
assumes $p: finite (set\text{-}pmf p)$
assumes $ind: prob\text{-}space.k\text{-universal} (measure\text{-}pmf p) 2 H$
 $\{\alpha::'a \text{ assg. } dom \alpha = S\}$
 $\{\alpha::nat \text{ assg. } dom \alpha = \{0..<card S - 1\}\}$
shows
 $measure\text{-}pmf.\text{prob} (\text{map}\text{-}pmf (\lambda s w. H w s) p) (L mstar \cup U mstar)$
 $\leq 1 / 4.92$
 $\langle proof \rangle$

lemma analysis-3:
assumes $p: finite (set\text{-}pmf p)$
assumes $ind: prob\text{-}space.k\text{-universal} (measure\text{-}pmf p) 2 H$
 $\{\alpha::'a \text{ assg. } dom \alpha = S\}$
 $\{\alpha::nat \text{ assg. } dom \alpha = \{0..<card S - 1\}\}$
assumes $\varepsilon\text{-up}: \varepsilon \leq 1$
shows
 $measure\text{-}pmf.\text{prob} (\text{map}\text{-}pmf (\lambda s w. H w s) p)$
 $approxcore\text{-}fail \leq 0.36$
 $\langle proof \rangle$

```

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 (λs 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 (λf. 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 (λf. 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
    (λx. case m x of None ⇒ def | Some v ⇒ v))

```

```

lemma Pi-pmf-replicate-pmf:
  assumes finite V

```

shows
 $(Pi\text{-pmf } V \text{ def } (\lambda\cdot. p)) =$
 $map\text{-pmf } (\lambda bs.$
 $map\text{-of-default } (zip \text{ (sorted-list-of-set } V) bs) \text{ def})$
 $(replicate\text{-pmf } (card \text{ } V) p)$
 $\langle proof \rangle$

lemma *proj-inter-neutral*:
assumes $\bigwedge w. w \in B \longleftrightarrow restr \text{ } S \text{ } w \in C$
shows $proj \text{ } S \text{ } (A \cap B) = proj \text{ } S \text{ } A \cap C$
 $\langle proof \rangle$

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 $sols :: 'fml \Rightarrow ('a \Rightarrow bool) \text{ set}$
fixes $enc\text{-xor} :: 'a \text{ set} \times bool \Rightarrow 'fml \Rightarrow 'fml$
assumes *sols-enc-xor*:
 $\bigwedge F \text{ xor. finite } (fst \text{ } xor) \implies$
 $sols \text{ } (enc\text{-xor } xor \text{ } F) =$
 $sols \text{ } F \cap \{\omega. satisfies\text{-xor } xor \text{ } \{x. \omega \text{ } x\}\}$
begin

definition *compute-thresh* :: $real \Rightarrow nat$
where *compute-thresh* $\varepsilon =$
 $nat \lceil 1 + 9.84 * (1 + \varepsilon / (1 + \varepsilon)) * (1 + 1 / \varepsilon)^2 \rceil$

definition *fix-t* :: $real \Rightarrow nat$
where *fix-t* $\delta =$
 $nat \lceil \ln (1 / \delta) / (2 * (0.5 - 0.36)^2) \rceil$

definition *raw-median-bound* :: $real \Rightarrow nat \Rightarrow real$
where *raw-median-bound* $\alpha \text{ } t =$
 $(\sum i = 0..t \text{ div } 2.$
 $(t \text{ choose } i) * (1 / 2 + \alpha) ^ i * (1 / 2 - \alpha) ^ (t - i))$

definition *compute-t* :: $real \Rightarrow nat \Rightarrow nat$
where *compute-t* $\delta \text{ } n =$
 $(if \text{ raw-median-bound } 0.14 \text{ } n < \delta \text{ then } n$
 $else \text{ fix-t } \delta)$

definition *size-xor* ::
 $'fml \Rightarrow 'a \text{ set} \Rightarrow$
 $(nat \Rightarrow ('a \text{ set} \times bool) \text{ option}) \Rightarrow$
 $nat \Rightarrow nat$
where *size-xor* $F \text{ } S \text{ } xorsf \text{ } i = ($
 $let \text{ } xors = map \text{ } (the \circ xorsf) \text{ } [0..<i] \text{ in}$

```

    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 $(\text{real} \circ \text{median } t) = (\lambda w::\text{nat} \Rightarrow \text{nat}. \text{median } t (\text{real} \circ w))$
 ⟨proof⟩

lemma *median-default*:
shows $\text{median } t y = \text{median } t (\lambda x. \text{if } x < t \text{ then } y \text{ else } \text{def})$
 ⟨proof⟩

definition *default- α :*'a set \Rightarrow nat assg
where *default- α* S $i =$ (if $i < \text{card } S - 1$ then Some True else None)

lemma *dom-default- α :*
dom (*default- α* S) = $\{0..<\text{card } S - 1\}$
 <proof>

lemma *compute-thresh-bound-4:*
assumes $\varepsilon > 0$
shows $4 < \text{compute-thresh } \varepsilon$
 <proof>

lemma *satisfies-xor-with-domain:*
assumes $\text{fst } x \subseteq S$
shows *satisfies-xor* $x \{x. w x\} \longleftrightarrow$
satisfies-xor $x (\{x. w x\} \cap S)$
 <proof>

lemma *approxcore-xors-eq:*
assumes *thresh:*
 $\text{thresh} = \text{compute-thresh } \varepsilon$
 $\text{thresh} \leq \text{card } (\text{proj } S (\text{sols } F))$
assumes $\varepsilon: \varepsilon > (0::\text{real}) \ \varepsilon \leq 1$
assumes $S: \text{finite } S$
shows *measure-pmf.prob* (*random-xors* S ($\text{card } S - 1$))
 $\{xors. \text{real } (\text{approxcore-xors } F \ S \ \text{thresh } xors) \in$
 $\{\text{real } (\text{card } (\text{proj } S (\text{sols } F))) / (1 + \varepsilon)..$
 $(1 + \varepsilon) * \text{real } (\text{card } (\text{proj } S (\text{sols } F)))\}\} \geq 0.64$
 <proof>

lemma *compute-t-ge1:*
assumes $0 < \delta \ \delta < 1$
shows *compute-t* $\delta \ n \geq 1$
 <proof>

lemma *success-arith-bound:*
assumes $s \leq (f :: \text{nat})$
assumes $p \leq (1::\text{real}) \ q \leq p \ 1/2 \leq q$
shows $p^s * (1 - p)^f \leq q^s * (1 - q)^f$
 <proof>

lemma *prob-binomial-pmf-upto-mono:*
assumes $1/2 \leq q \ q \leq p \ p \leq 1$
shows
measure-pmf.prob (*binomial-pmf* n p) $\{..n \text{ div } 2\} \leq$
measure-pmf.prob (*binomial-pmf* n q) $\{..n \text{ div } 2\}$
 <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)
 $\{c. \text{real } c \in$
 $\{\text{real } (\text{card } (\text{proj } S (\text{sols } F))) / (1 + \varepsilon)..$
 $(1 + \varepsilon) * \text{real } (\text{card } (\text{proj } S (\text{sols } F)))\}$
 $\geq 1 - \delta$
 $\langle \text{proof} \rangle$

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

definition *mk-eps* :: *real* \Rightarrow *real*
where *mk-eps* $\varepsilon = (\text{if } \varepsilon > 1 \text{ then } 1 \text{ else } \varepsilon)$

definition *approxmc'* ::
'fml \Rightarrow *'a set* \Rightarrow
real \Rightarrow *real* \Rightarrow *nat* \Rightarrow *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)
 $\{c. \text{real } c \in$
 $\{\text{real } (\text{card } (\text{proj } S (\text{sols } F))) / (1 + \varepsilon)..$
 $(1 + \varepsilon) * \text{real } (\text{card } (\text{proj } S (\text{sols } F)))\}$
 $\geq 1 - \delta$
 $\langle \text{proof} \rangle$

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

definition *approxmc-map* ::
'fml \Rightarrow *'a set* \Rightarrow
real \Rightarrow *real* \Rightarrow *nat* \Rightarrow
(*nat* \Rightarrow *nat* \Rightarrow (*'a set* \times *bool*) *option*) \Rightarrow
nat
where *approxmc-map* F S ε δ n *xorsFs* = (
 $\text{let } \varepsilon = \text{mk-eps } \varepsilon \text{ in}$
 $\text{let thresh} = \text{compute-thresh } \varepsilon \text{ in}$
 $\text{if } \text{card } (\text{proj } S (\text{sols } F)) < \text{thresh} \text{ then}$
 $\text{card } (\text{proj } S (\text{sols } F))$
 else
 $\text{let } t = \text{compute-t } \delta \text{ } n \text{ in}$

$median\ t\ (approxcore\text{-}xors\ F\ S\ thresh\ \circ\ xorsFs)$

lemma *approxmc-map-eq*:

shows

$map\text{-}pmf\ (approxmc\text{-}map\ F\ S\ \varepsilon\ \delta\ n)$
 $(Pi\text{-}pmf\ \{0..<compute\text{-}t\ \delta\ n\}\ def$
 $(\lambda i. random\text{-}xors\ S\ (card\ S - 1))) =$
 $approxmc'\ F\ S\ \varepsilon\ \delta\ n$

$\langle proof \rangle$

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 $'a\ xor = 'a\ list \times bool$

definition *satisfies-xorL* :: $'a\ xor \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$

where *satisfies-xorL* $xb\ \omega =$

$even\ (sum\text{-}list\ (map\ (of\text{-}bool\ \circ\ \omega)\ (fst\ xb)) +$
 $of\text{-}bool\ (snd\ xb)::nat)$

definition *sublist-bits*:: $'a\ list \Rightarrow bool\ list \Rightarrow 'a\ list$

where *sublist-bits* $ls\ bs =$

$map\ fst\ (filter\ snd\ (zip\ ls\ bs))$

definition *xor-from-bits*::

$'a\ list \Rightarrow bool\ list \times bool \Rightarrow 'a\ xor$

where *xor-from-bits* $V\ xsb =$

$(sublist\text{-}bits\ V\ (fst\ xsb),\ snd\ xsb)$

locale *ApproxMCL* =

fixes *sols* :: $'fml \Rightarrow ('a \Rightarrow bool)\ set$

fixes *enc-xor* :: $'a\ xor \Rightarrow 'fml \Rightarrow 'fml$

assumes *sols-enc-xor*:

$\bigwedge F\ xor.$

$sols\ (enc\text{-}xor\ xor\ F) =$

$sols F \cap \{\omega. \text{satisfies-xorL } xor \ \omega\}$
begin

definition *list-of-set* :: 'a set \Rightarrow 'a list
where *list-of-set* $x = (@ls. \text{set } ls = x \wedge \text{distinct } ls)$

definition *xor-conc* :: 'a set \times bool \Rightarrow 'a xor
where *xor-conc* $xsb = (\text{list-of-set } (fst \ xsb), \text{snd } xsb)$

definition *enc-xor-conc* :: 'a set \times bool \Rightarrow 'fml \Rightarrow 'fml
where *enc-xor-conc* $= \text{enc-xor} \circ \text{xor-conc}$

lemma *distinct-count-list*:
assumes *distinct* ls
shows *count-list* $ls \ x = \text{of-bool } (x \in \text{set } ls)$
 $\langle \text{proof} \rangle$

lemma *list-of-set*:
assumes *finite* x
shows *distinct* $(\text{list-of-set } x)$ *set* $(\text{list-of-set } x) = x$
 $\langle \text{proof} \rangle$

lemma *count-list-list-of-set*:
assumes *finite* x
shows *count-list* $(\text{list-of-set } x) \ y = \text{of-bool } (y \in x)$
 $\langle \text{proof} \rangle$

lemma *satisfies-xorL-xor-conc*:
assumes *finite* x
shows *satisfies-xorL* $(\text{xor-conc } (x,b)) \ \omega \longleftrightarrow \text{satisfies-xor } (x,b) \ \{x. \ \omega\}$
 $\langle \text{proof} \rangle$

sublocale *appmc*: *ApproxMC* *sols* *enc-xor-conc*
 $\langle \text{proof} \rangle$

definition *size-xorL* ::
'fml \Rightarrow 'a list \Rightarrow
 $(\text{nat} \Rightarrow \text{bool list} \times \text{bool}) \Rightarrow$
 $\text{nat} \Rightarrow \text{nat}$
where *size-xorL* $F \ S \ \text{xorsl} \ i = ($
 $\text{let } \text{xors} = \text{map } (\text{xor-from-bits } S \circ \text{xorsl}) \ [0..<i] \ \text{in}$
 $\text{let } \text{Fenc} = \text{fold } \text{enc-xor } \ \text{xors} \ F \ \text{in}$
 $\text{card } (\text{proj } (\text{set } S) (\text{sols } \text{Fenc}))$

definition *check-xorL* ::
'fml \Rightarrow 'a list \Rightarrow
 $\text{nat} \Rightarrow$
 $(\text{nat} \Rightarrow \text{bool list} \times \text{bool}) \Rightarrow$

$nat \Rightarrow bool$
where $check\text{-}xorL\ F\ S\ thresh\ xorSl\ i =$
 $(size\text{-}xorL\ F\ S\ xorSl\ i < thresh)$

definition $approxcore\text{-}xorSl ::$
 $'fml \Rightarrow 'a\ list \Rightarrow$
 $nat \Rightarrow$
 $(nat \Rightarrow (bool\ list \times bool)) \Rightarrow$
 nat
where
 $approxcore\text{-}xorSl\ F\ S\ thresh\ xorSl =$
 $(case\ List.find$
 $(check\text{-}xorL\ F\ S\ thresh\ xorSl)\ [1..<length\ S]\ of$
 $None \Rightarrow 2 \wedge length\ S$
 $| Some\ m \Rightarrow$
 $(2 \wedge m * size\text{-}xorL\ F\ S\ xorSl\ m))$

definition $xor\text{-}abs :: 'a\ xor \Rightarrow 'a\ set \times bool$
where $xor\text{-}abs\ xsb = (set\ (fst\ xsb),\ snd\ xsb)$

lemma $sols\text{-}fold\text{-}enc\text{-}xor:$
assumes $list\text{-}all2\ (\lambda x\ y.$
 $\forall w. satisfies\text{-}xorL\ x\ w = satisfies\text{-}xorL\ y\ w)\ xs\ ys$
assumes $sols\ F = sols\ G$
shows $sols\ (fold\ enc\text{-}xor\ xs\ F) = sols\ (fold\ enc\text{-}xor\ ys\ G)$
 $\langle proof \rangle$

lemma $satisfies\text{-}xor\text{-}xor\text{-}abs:$
assumes $distinct\ x$
showssatisfies\text{-}xor $(xor\text{-}abs\ (x,b))\ \{x.\ \omega\ x\} \longleftrightarrow satisfies\text{-}xorL\ (x,b)$
 ω
 $\langle proof \rangle$

lemma $xor\text{-}conc\text{-}xor\text{-}abs\text{-}rel:$
assumes $distinct\ (fst\ x)$
showssatisfies\text{-}xorL $(xor\text{-}conc\ (xor\text{-}abs\ x))\ w \longleftrightarrow$
 $satisfies\text{-}xorL\ x\ w$
 $\langle proof \rangle$

lemma $sorted\text{-}sublist\text{-}bits:$
assumes $sorted\ V$
showssorted $(sublist\text{-}bits\ V\ bs)$
 $\langle proof \rangle$

lemma $distinct\text{-}sublist\text{-}bits:$
assumes $distinct\ V$
showsdistinct $(sublist\text{-}bits\ V\ bs)$
 $\langle proof \rangle$

lemma *distinct-fst-xor-from-bits*:
assumes *distinct V*
shows *distinct (fst (xor-from-bits V bs))*
 \langle *proof* \rangle

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

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

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

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

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

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

definition *approxmc-mapL*::

fml \Rightarrow *'a list* \Rightarrow
real \Rightarrow *real* \Rightarrow *nat* \Rightarrow
(nat \Rightarrow *nat* \Rightarrow (*bool list* \times *bool*)) \Rightarrow
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* \circ *xorsLs*))

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

lemma *approxmc-mapL*:
assumes $\bigwedge i j. j < \text{length } S - 1 \implies$
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*
 \langle *proof* \rangle

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*
($\lambda i j. \text{if } j < \text{length } S - 1$
then *Some* (*xor-abs* (*xor-from-bits* *S* (*xorsLs i j*)))
else *None*)
 \langle *proof* \rangle

lemma *bits-to-random-xor*:
assumes *distinct S*
shows *map-pmf*
($\lambda x. \text{xor-abs}$ (*xor-from-bits* *S* *x*))
(*random-xorB* (*length S*)) =
random-xor (*set S*)
 \langle *proof* \rangle

lemma *Pi-pmf-map-pmf-Some*:

assumes *finite S*
shows $Pi\text{-pmf } S \text{ None } (\lambda\cdot. \text{map-pmf } \text{Some } p) =$
 $\text{map-pmf } (\lambda f v. \text{if } v \in S \text{ then } \text{Some } (f v) \text{ else } \text{None})$
 $(Pi\text{-pmf } S \text{ def } (\lambda\cdot. p))$
 $\langle\text{proof}\rangle$

lemma *bits-to-random-xors:*

assumes *distinct S*
shows
 map-pmf
 $(\lambda f j.$
 $\text{if } j < n$
 $\text{then } \text{Some } (\text{xor-abs } (\text{xor-from-bits } S (f j)))$
 $\text{else } \text{None})$
 $(Pi\text{-pmf } \{..\langle n::\text{nat}\rangle\} \text{ def } (\lambda\cdot. \text{random-xorB } (\text{length } S))) =$
 $\text{random-xors } (\text{set } S) n$
 $\langle\text{proof}\rangle$

lemma *bits-to-all-random-xors:*

assumes *distinct S*
assumes $(\lambda j. \text{if } j < n$
 $\text{then } \text{Some } (\text{xor-abs } (\text{xor-from-bits } S (\text{def1 } j)))$
 $\text{else } \text{None}) = \text{def}$
shows
 map-pmf
 $((\circ) (\lambda f j. \text{if } j < n$
 $\text{then } \text{Some } (\text{xor-abs } (\text{xor-from-bits } S (f j)))$
 $\text{else } \text{None}))$
 $(Pi\text{-pmf } \{0..\langle m::\text{nat}\rangle\} \text{ def1}$
 $(\lambda\cdot.$
 $Pi\text{-pmf } \{..\langle n::\text{nat}\rangle\} \text{ def2 } (\lambda\cdot. \text{random-xorB } (\text{length } S)))) =$
 $Pi\text{-pmf } \{0..\langle m\rangle\} \text{ def}$
 $(\lambda i. \text{random-xors } (\text{set } S) n)$
 $\langle\text{proof}\rangle$

definition *random-seed-xors::nat \Rightarrow nat \Rightarrow (nat \Rightarrow nat \Rightarrow bool list \times bool) pmf*

where *random-seed-xors t l =*
 $(\text{prod-pmf } \{0..\langle t\rangle\}$
 $(\lambda\cdot. \text{prod-pmf } \{..\langle l-1\rangle\} (\lambda\cdot. \text{random-xorB } l)))$

lemma *approxmL-sound:*

assumes $\delta: \delta > 0 \ \delta < 1$
assumes $\varepsilon: \varepsilon > 0$
assumes *S: distinct S*
shows
 prob-space.prob
 $(\text{map-pmf } (\text{approxmc-mapL } F S \varepsilon \delta n))$

```

      (random-seed-xors (appmc.compute-t  $\delta$   $n$ ) (length  $S$ )))
    {c. real  $c \in$ 
      {real (card (proj (set  $S$ ) (sols  $F$ ))) / (1 +  $\varepsilon$ )..
      (1 +  $\varepsilon$ ) * real (card (proj (set  $S$ ) (sols  $F$ )))}}
     $\geq 1 - \delta$ 
  <proof>

```

lemma *approxmcL-sound'*:

assumes δ : $\delta > 0$ $\delta < 1$

assumes ε : $\varepsilon > 0$

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 $c \notin$

{real (card (proj (set S) (sols F))) / (1 + ε)..

(1 + ε) * real (card (proj (set S) (sols F)))}} $\leq \delta$

<proof>

end

7.2 ApproxMC certificate checker

definition *str-of-bool* :: *bool* \Rightarrow *String.literal*

where *str-of-bool* $b =$ (

if b then *STR* "true" else *STR* "false")

fun *str-of-nat-aux* :: *nat* \Rightarrow *char list* \Rightarrow *char list*

where *str-of-nat-aux* n $acc =$ (

let $c =$ *char-of-integer* (*of-nat* ($48 + n \bmod 10$)) in

if $n < 10$ then $c \# acc$

else *str-of-nat-aux* ($n \div 10$) ($c \# acc$))

definition *str-of-nat* :: *nat* \Rightarrow *String.literal*

where *str-of-nat* $n =$ *String.implode* (*str-of-nat-aux* n [])

type-synonym *'a sol* = (*'a* \times *bool*) *list*

definition *canon-map-of* :: (*'a* \times *bool*) *list* \Rightarrow (*'a* \Rightarrow *bool*)

where *canon-map-of* $ls =$

(let $m =$ *map-of* ls in

($\lambda x.$ case m x of *None* \Rightarrow *False* | *Some* $b \Rightarrow b$))

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  $\Rightarrow$  'a list  $\Rightarrow$  nat  $\Rightarrow$ 
  nat  $\Rightarrow$  nat  $\Rightarrow$ 
  (nat  $\Rightarrow$  (nat  $\times$  ('a  $\Rightarrow$  bool) list  $\times$  ('a  $\Rightarrow$  bool) list))  $\Rightarrow$ 
  (nat  $\Rightarrow$  nat  $\Rightarrow$  (bool list  $\times$  bool))
 $\Rightarrow$  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  $\Rightarrow$  Inl (STR "round " + str-of-nat it + STR " " + err)
  | Inr n  $\Rightarrow$ 
    (case fold-approxcore-xorsL-cert F S thresh t i cs xorsLs of
      Inl err  $\Rightarrow$  Inl err
    | Inr ns  $\Rightarrow$  Inr (n # ns)))

```

```

definition calc-median::
  'fml  $\Rightarrow$  'a list  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$ 
  (nat  $\Rightarrow$  (nat  $\times$  ('a  $\Rightarrow$  bool) list  $\times$  ('a  $\Rightarrow$  bool) list))  $\Rightarrow$ 
  (nat  $\Rightarrow$  nat  $\Rightarrow$  (bool list  $\times$  bool))  $\Rightarrow$ 
  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  $\Rightarrow$  Inl err
  | Inr ls  $\Rightarrow$  Inr (sort ls ! (t div 2))
  )

```

```

fun certcheck::
  'fml  $\Rightarrow$  'a list  $\Rightarrow$ 
  real  $\Rightarrow$  real  $\Rightarrow$ 
  (('a  $\Rightarrow$  bool) list  $\times$ 
  (nat  $\Rightarrow$  (nat  $\times$  ('a  $\Rightarrow$  bool) list  $\times$  ('a  $\Rightarrow$  bool) list)))  $\Rightarrow$ 
  (nat  $\Rightarrow$  nat  $\Rightarrow$  (bool list  $\times$  bool))  $\Rightarrow$ 
  String.literal + nat
where certcheck F S  $\varepsilon$   $\delta$  (m0,ms) xorsLs = (
  let  $\varepsilon$  = appmc.mk-eps  $\varepsilon$  in
  let thresh = appmc.compute-thresh  $\varepsilon$  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  $\delta$  in
      calc-median F S thresh t ms xorsLs)

```

```

context
assumes check-unsat:  $\bigwedge F. \text{check-unsat } F \implies \text{sols } F = \{\}$ 
begin

```


lemma *sols-fold-ban-sol*:
shows $sols (fold\ ban\ sol\ ls\ F) =$
 $sols\ F \cap \{\omega. (\forall vs \in set\ ls. map\ \omega\ (map\ fst\ vs) \neq map\ snd\ vs)\}$
 $\langle proof \rangle$

lemma *inter-cong-right*:
assumes $\bigwedge x. x \in A \implies x \in B \longleftrightarrow x \in C$
shows $A \cap B = A \cap C$
 $\langle proof \rangle$

lemma *proj-sol-canon-map-of*:
assumes $distinct\ S\ length\ S = length\ w$
shows $proj\ sol\ S (canon\ map\ of\ (zip\ S\ w)) = w$
 $\langle proof \rangle$

lemma *proj-sol-cong*:
assumes $restr\ (set\ S)\ A = restr\ (set\ S)\ B$
shows $proj\ sol\ S\ A = proj\ sol\ S\ B$
 $\langle proof \rangle$

lemma *canon-map-of-map-of*:
assumes $length\ S = length\ x$
assumes $canon\ map\ of\ (zip\ S\ x) \in A$
shows $map\ of\ (zip\ S\ x) \in proj\ (set\ S)\ A$
 $\langle proof \rangle$

lemma *proj-proj-sol-map-of-zip-1*:
assumes $distinct\ S\ length\ S = length\ w$
assumes $w: w \in rdb$
shows
 $map\ of\ (zip\ S\ w) \in$
 $proj\ (set\ S)\ \{\omega. proj\ sol\ S\ \omega \in rdb\}$
 $\langle proof \rangle$

lemma *proj-proj-sol-map-of-zip-2*:
assumes $\bigwedge bs. bs \in rdb \implies length\ bs = length\ S$
assumes $w: w \in proj\ (set\ S)\ \{\omega. proj\ sol\ S\ \omega \in rdb\}$
shows
 $w \in (map\ of \circ zip\ S) \text{ ` } rdb$
 $\langle proof \rangle$

lemma *proj-proj-sol-map-of-zip*:
assumes $distinct\ S$
assumes $\bigwedge bs. bs \in rdb \implies length\ bs = length\ S$
shows
 $proj\ (set\ S)\ \{\omega. proj\ sol\ S\ \omega \in rdb\} =$
 $(map\ of \circ zip\ S) \text{ ` } rdb$
 $\langle proof \rangle$

definition *ban-proj-sol* :: 'a list \Rightarrow ('a \Rightarrow bool) list \Rightarrow 'fml \Rightarrow 'fml
where *ban-proj-sol* *S* *xs* *F* =
 fold *ban-sol* (map (zip *S* \circ *proj-sol* *S*) *xs*) *F*

lemma *check-sol-imp-proj*:
assumes $w \in \text{sols } F$
shows $\text{map-of } (\text{zip } S (\text{proj-sol } S w)) \in \text{proj } (\text{set } S) (\text{sols } F)$
 $\langle \text{proof} \rangle$

lemma *checked-BSAT-lower*:
assumes *S*: *distinct* *S*
assumes *check-BSAT-sols* *F* *S* *thresh* *xs* = *Inr* ()
shows $\text{length } xs \leq \text{card } (\text{proj } (\text{set } S) (\text{sols } F))$
 $\text{length } xs < \text{thresh} \implies$
 $\text{card } (\text{proj } (\text{set } S) (\text{sols } F)) = \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *good-BSAT*:
assumes *distinct* *S*
assumes *BSAT* *F* *S* *thresh* *xs* = *Inr* *n*
shows $n \leq \text{card } (\text{proj } (\text{set } S) (\text{sols } F))$
 $n < \text{thresh} \implies$
 $\text{card } (\text{proj } (\text{set } S) (\text{sols } F)) = n$
 $\langle \text{proof} \rangle$

lemma *size-xorL-cert*:
assumes *distinct* *S*
assumes *size-xorL-cert* *F* *S* *thresh* *xorsl* *i* *xs* = *Inr* *n*
shows
 $\text{size-xorL } F S \text{ xorsl } i \geq n$
 $n < \text{thresh} \longrightarrow \text{size-xorL } F S \text{ xorsl } i = n$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *fold-approxcore-xorsL-cert*:
assumes *S*: *distinct* *S*
assumes $i \leq t$
assumes *fold-approxcore-xorsL-cert* *F* *S* *thresh* *t* *i* *cs* *xorsLs* = *Inr* *ns*
shows $\text{map } (\text{approxcore-xorsL } F S \text{ thresh } \circ \text{xorsLs}) [t-i..<t] = \text{ns}$
 $\langle \text{proof} \rangle$

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* \circ $xorsLs$) = n
 \langle *proof* \rangle

lemma *compute-t-find-t[simp]*:
shows *apPMC.compute-t* δ (*find-t* δ) = *find-t* δ
 \langle *proof* \rangle

lemma *certcheck*:
assumes *distinct* S
assumes *certcheck* $F S \varepsilon \delta$ ($m0,ms$) $xorsLs = Inr n$
shows *approxmc-mapL* $F S \varepsilon \delta$ (*find-t* δ) $xorsLs = n$
 \langle *proof* \rangle

lemma *certcheck'*:
assumes *distinct* S
assumes $\neg isl$ (*certcheck* $F S \varepsilon \delta$ m $xorsLs$)
shows *projr* (*certcheck* $F S \varepsilon \delta$ m $xorsLs$) =
approxmc-mapL $F S \varepsilon \delta$ (*find-t* δ) $xorsLs$
 \langle *proof* \rangle

lemma *certcheck-sound*:
assumes δ : $\delta > 0$ $\delta < 1$
assumes ε : $\varepsilon > 0$
assumes S : *distinct* S
shows
measure-pmf.prob
map-pmf ($\lambda r. \text{certcheck } F S \varepsilon \delta (f r) r$)
random-seed-xors (*find-t* δ) (*length* S))
 $\{c. \neg isl c \wedge$
 $real (projr c) \notin$
 $\{real (card (proj (set S) (sols F))) / (1 + \varepsilon)..$
 $(1 + \varepsilon) * real (card (proj (set S) (sols F)))\}\} \leq \delta$
 \langle *proof* \rangle

lemma *certcheck-promise-complete*:
assumes δ : $\delta > 0$ $\delta < 1$
assumes ε : $\varepsilon > 0$
assumes S : *distinct* S
assumes r : $\bigwedge r.$
 $r \in \text{set-pmf } (\text{random-seed-xors } (\text{find-t } \delta) (\text{length } S)) \implies$
 $\neg isl (\text{certcheck } F S \varepsilon \delta (f r) r)$
shows
measure-pmf.prob
map-pmf ($\lambda r. \text{certcheck } F S \varepsilon \delta (f r) r$)
random-seed-xors (*find-t* δ) (*length* S))

```

      {c. real (projr c) ∈
       {real (card (proj (set S) (sols F))) / (1 + ε)..
        (1 + ε) * real (card (proj (set S) (sols F)))}} ≥ 1 - δ
⟨proof⟩

```

end

```

lemma certcheck-code[code]:
  certcheck F S ε δ (m0,ms) xorsLs = (
    if δ > 0 ∧ δ < 1 ∧ ε > 0 ∧ 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 ⇒ 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)
    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 $l_1 + l_2 + \dots + l_n = 1$ 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* :: $fml \Rightarrow (nat \Rightarrow bool) \Rightarrow bool$

where *check-sol fml w* = (
list-all (*list-ex* (*sat-lit w*)) (*fst fml*) \wedge
list-all (*sat-cmsxor w*) (*snd fml*))

definition *ban-sol* :: $(nat \times bool) list \Rightarrow fml \Rightarrow fml$

where *ban-sol vs fml* =
 ((*map* ($\lambda(v,b). \text{if } b \text{ then } Neg \ v \text{ else } Pos \ v$) *vs*) # *fst fml, snd fml*)

lemma *check-sol-sol*:

shows $w \in \text{sols } F \longleftrightarrow$
check-sol F w
 $\langle \text{proof} \rangle$

lemma *ban-sat-clause*:

shows *sat-clause w* (*map* ($\lambda(v, b). \text{if } b \text{ then } Neg \ v \text{ else } Pos \ v$) *vs*)
 \longleftrightarrow
map w (*map fst vs*) \neq *map snd vs*
 $\langle \text{proof} \rangle$

lemma *sols-ban-sol*:

shows *sols* (*ban-sol vs F*) =
sols F \cap
 { $\omega. \text{map } \omega$ (*map fst vs*) \neq *map snd vs*}
 $\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

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* \Rightarrow *lit*

where *negate-lit* *l* = (case *l* of *Pos* *x* \Rightarrow *Neg* *x* | *Neg* *x* \Rightarrow *Pos* *x*)

fun *xor-clauses* :: *cmsxor* \Rightarrow *bool* \Rightarrow *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* (\neg *b*) in

map ($\lambda c. x \# c$) *p-x* @ map ($\lambda c. \text{negate-lit } x \# c$) *n-x*)

lemma *sat-cmsxor-nil[simp]*:

shows \neg (*sat-cmsxor* *w* [])

\langle *proof* \rangle

lemma *sat-cmsxor-cons*:

shows *sat-cmsxor* *w* (*x* # *xs*) =

(if *sat-lit* *w* *x* then \neg (*sat-cmsxor* *w* *xs*) else *sat-cmsxor* *w* *xs*)

\langle *proof* \rangle

lemma *sat-cmsxor-append*:

shows *sat-cmsxor* *w* (*xs* @ *ys*) =

(if *sat-cmsxor* *w* *xs* then \neg (*sat-cmsxor* *w* *ys*) else *sat-cmsxor* *w* *ys*)

\langle *proof* \rangle

definition *sat-clauses*:: *assignment* \Rightarrow *clause list* \Rightarrow *bool*

where *sat-clauses* *w* *cs* = ($\forall c \in \text{set } cs. \text{sat-clause } w c$)

lemma *sat-clauses-append*:

shows *sat-clauses* *w* (*xs* @ *ys*) =

(*sat-clauses* *w* *xs* \wedge *sat-clauses* *w* *ys*)

\langle *proof* \rangle

lemma *sat-clauses-map*:

shows *sat-clauses* *w* (map ((#) *x*) *cs*) =

(*sat-lit* *w* *x* \vee *sat-clauses* *w* *cs*)

\langle *proof* \rangle

lemma *sat-lit-negate-lit[simp]*:

sat-lit *w* (*negate-lit* *l*) = (\neg *sat-lit* *w* *l*)

\langle *proof* \rangle

lemma *sols-xor-clauses*:

shows

sat-clauses *w* (*xor-clauses* *xs* *b*) \longleftrightarrow

(*sat-cmsxor* *w* *xs* = *b*)

\langle *proof* \rangle

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

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

lemma *sat-lit-same*:
assumes $\bigwedge x. x \leq \text{var-lit } l \implies w\ x = w'\ x$
shows *sat-lit* *w l* = *sat-lit* *w' l*
 $\langle \text{proof} \rangle$

lemma *var-lits-eq*:
var-lits *ls* = *Max (set (0 # map var-lit ls))*
 $\langle \text{proof} \rangle$

lemma *sat-lits-same*:
assumes $\bigwedge x. x \leq \text{var-lits } c \implies w\ x = w'\ x$
shows *sat-clause* *w c* = *sat-clause* *w' c*
 $\langle \text{proof} \rangle$

lemma *le-var-lits-in*:
assumes $y \in \text{set } ys\ v \leq \text{var-lit } y$
shows $v \leq \text{var-lits } ys$
 $\langle \text{proof} \rangle$

lemma *sat-cmsxor-same*:
assumes $\bigwedge x. x \leq \text{var-lits } xs \implies w\ x = w'\ x$
shows *sat-cmsxor* *w xs* = *sat-cmsxor* *w' xs*
 $\langle \text{proof} \rangle$

lemma *sat-cmsxor-split*:
assumes $u: \text{var-lits } xs < u\ \text{var-lits } ys < u$
assumes $w': w' = (\lambda x. \text{if } x = u \text{ then } \neg \text{sat-cmsxor } w\ xs \text{ else } w\ x)$
shows
 $(\text{sat-cmsxor } w\ (xs @ ys) =$
 $(\text{sat-cmsxor } w'\ (\text{Pos } u \# xs) \wedge$
 $\text{sat-cmsxor } w'\ (\text{Neg } u \# ys)))$
 $\langle \text{proof} \rangle$

fun *split-xor* :: *nat* \Rightarrow *cmsxor* \Rightarrow *cmsxor list* \times *nat* \Rightarrow *cmsxor list* \times *nat*
where *split-xor* *k xs (acc,u)* = (
if length xs $\leq k + 3$ *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  $u \leq u'$ 
<proof>

```

lemma *var-lits-append*:

```

shows  $\text{var-lits } xs \leq \text{var-lits } (xs @ ys)$ 
 $\text{var-lits } ys \leq \text{var-lits } (xs @ ys)$ 
<proof>

```

lemma *fold-max-eq*:

```

assumes  $i \leq u$ 
shows  $\text{fold max } ls \ u = \text{max } u \ (\text{fold max } ls \ (i::\text{nat}))$ 
<proof>

```

lemma *split-xor-sound*:

```

assumes  $\text{sat-cmsxor } w \ xs \ \wedge \ x \in \text{set } acc \implies \text{sat-cmsxor } w \ x$ 
assumes  $u: \text{var-lits } xs < u \ \wedge \ x \in \text{set } acc \implies \text{var-lits } x < u$ 
assumes split-xor k xs (acc,u) = (acc',u')
obtains  $w'$  where
 $\wedge x. x < u \implies w \ x = w' \ x$ 
 $\wedge x. x \in \text{set } acc' \implies \text{sat-cmsxor } w' \ x$ 
 $\wedge x. x \in \text{set } acc' \implies \text{var-lits } x < u'$ 
<proof>

```

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

lemma *split-xors-sound*:

```

assumes  $\wedge x. x \in \text{set } xs \implies \text{sat-cmsxor } w \ x$ 
 $\wedge x. x \in \text{set } acc \implies \text{sat-cmsxor } w \ x$ 
assumes  $u: \wedge x. x \in \text{set } xs \implies \text{var-lits } x < u$ 
 $\wedge x. x \in \text{set } acc \implies \text{var-lits } x < u$ 
assumes  $\text{fold (split-xor k) } xs \ (acc,u) = (acc',u')$ 
obtains  $w'$  where
 $\wedge x. x < u \implies w \ x = w' \ x$ 
 $\wedge x. x \in \text{set } acc' \implies \text{sat-cmsxor } w' \ x$ 
 $\wedge x. x \in \text{set } acc' \implies \text{var-lits } x < u'$ 
<proof>

```

definition *var-fml* :: *fml* \Rightarrow *nat*
where *var-fml* *f* =
 \max ($\text{fold } \max$ ($\text{map } \text{var-lits}$ ($\text{fst } f$)) 0)
 \quad ($\text{fold } \max$ ($\text{map } \text{var-lits}$ ($\text{snd } f$)) 0)

lemma *var-fml-eq*:
 $\text{var-fml } f =$
 \max (Max (set (0 # $\text{map } \text{var-lits}$ ($\text{fst } f$))))
 \quad (Max (set (0 # $\text{map } \text{var-lits}$ ($\text{snd } f$))))
 $\langle \text{proof} \rangle$

definition *split-fml* :: *nat* \Rightarrow *fml* \Rightarrow *fml*
where *split-fml* *k* *f* = (
 $\text{let } u = \text{var-fml } f + 1$ in
 $(\text{fst } f, (\text{split-xors } k \ u \ (\text{snd } f)))$
 $)$

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

lemma *split-fml-satisfies*:
assumes *sat-fml* *w* *F*
obtains *w'* **where** *sat-fml* *w'* (*split-fml* *k* *F*)
 $\langle \text{proof} \rangle$

lemma *split-fml-sols*:
assumes *sols* (*split-fml* *k* *F*) = {}
shows *sols* *F* = {}
 $\langle \text{proof} \rangle$

definition *blast-xors* :: *cmsxor* *list* \Rightarrow *clause* *list*
where *blast-xors* *xors* = concat ($\text{map } (\lambda x. \text{xor-clauses } x \ \text{True})$ *xors*)

definition *blast-fml* :: *fml* \Rightarrow *clause* *list*
where *blast-fml* *f* =
 $\text{fst } f \ @ \ \text{blast-xors } (\text{snd } f)$

lemma *sat-clauses-concat*:
 $\text{sat-clauses } w \ (\text{concat } xs) \longleftrightarrow$
 $(\forall x \in \text{set } xs. \text{sat-clauses } w \ x)$
 $\langle \text{proof} \rangle$

lemma *blast-xors-sound*:
assumes $(\bigwedge x. x \in \text{set } \text{xors} \Longrightarrow \text{sat-cmsxor } w \ x)$
shows *sat-clauses* *w* (*blast-xors* *xors*)
 $\langle \text{proof} \rangle$

lemma *blast-fml-sound*:
assumes *sat-fml w F*
shows *sat-fml w (blast-fml F, [])*
 $\langle proof \rangle$

definition *blast-split-fml* :: *fml* \Rightarrow *clause list*
where *blast-split-fml f* = *blast-fml (split-fml 1 f)*

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

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-blast check-unsat F S* ε δ *m0ms* =
certcheck (check-unsat \circ *blast-split-fml) F S* ε δ *m0ms*

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

corollary *certcheck-blast-promise-complete*:
assumes $\bigwedge F. check-unsat F \Longrightarrow sols (F, []) = \{\}$
assumes $0 < \delta \delta < 1$
assumes $0 < \varepsilon$
assumes *distinct S*
assumes *r*: $\bigwedge r.$
 $r \in set-pmf (random-seed-xors (find-t \delta) (length S)) \Longrightarrow$
 $\neg isl (certcheck-blast check-unsat F S \varepsilon \delta (f r) r)$
shows

```

measure-pmf.prov
  (map-pmf ( $\lambda r.$  certcheck-blast check-unsat  $F S \in \delta$  ( $f r$ )  $r$ )
    (random-seed-xors (find-t  $\delta$ ) (length  $S$ )))
  { $c.$  real (projr  $c$ )  $\in$ 
    {real (card (proj (set  $S$ ) (sols  $F$ ))) / ( $1 + \epsilon$ )..
      ( $1 + \epsilon$ ) * real (card (proj (set  $S$ ) (sols  $F$ )))}}  $\geq 1 - \delta$ 
  <proof>

```

8.2 Export code for a SML implementation.

```

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

```

```

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, *IJ-CAI*, pages 3569–3576. IJCAI/AAAI Press, 2016.