Abstract

This AFP entry defines a probabilistic while operator based on sub-probability mass functions and formalises zero-one laws and variant rules for probabilistic loop termination. As applications, we implement probabilistic algorithms for the Bernoulli, geometric and arbitrary uniform distributions that only use fair coin flips, and prove them correct and terminating with probability 1.

Contents

1 Miscellaneous library additions 2
2 Probabilistic while loop 5
3 Rules for probabilistic termination 6
   3.1 0/1 termination laws 6
   3.2 Variant rule 7
4 Distributions built from coin flips 8
   4.1 The Bernoulli distribution 8
   4.2 The geometric distribution 9
   4.3 Arbitrary uniform distributions 10

theory While-SPMF imports
  MFMC-Countable, Rel-PMF-Characterisation
  HOL-Types-To-Sets, Types-To-Sets
  HOL-Library, Complete-Partial-Order2
begin

This theory defines a probabilistic while combinator for discrete (sub-)probabilities and formalises rules for probabilistic termination similar to those by Hurd [1] and McIver and Morgan [3].
1 Miscellaneous library additions

fun map-option-set :: ('a ⇒ 'b option set) ⇒ 'a option ⇒ 'b option set
where
  map-option-set f None = {None}
  | map-option-set f (Some x) = f x

lemma None-in-map-option-set:
  None ∈ map-option-set f x ⇔ None ∈ Set.bind (set-option x) f ∨ x = None
⟨proof⟩

lemma None-in-map-option-set-None [intro!]: None ∈ map-option-set f None
⟨proof⟩

lemma None-in-map-option-set-Some [intro!]: None ∈ f x ⇒ None ∈ map-option-set f (Some x)
⟨proof⟩

lemma Some-in-map-option-set [intro!]: Some y ∈ f x ⇒ Some y ∈ map-option-set f (Some x)
⟨proof⟩

lemma map-option-set-singleton [simp]: map-option-set (λx. {f x}) y = {Option.bind y f}
⟨proof⟩

lemma Some-eq-bind-conv: Some y = Option.bind x f ⇔ (∃ z. x = Some z ∧ f z = Some y)
⟨proof⟩

lemma map-option-set-bind: map-option-set f (Option.bind x g) = map-option-set (map-option-set f ∘ g) x
⟨proof⟩

lemma Some-in-map-option-set-conv: Some y ∈ map-option-set f x ⇔ (∃ z. x = Some z ∧ Some y ∈ f z)
⟨proof⟩

interpretation rel-spmf-characterisation ⟨proof⟩
hide-fact (open) rel-pmf-measureI

lemma Sup-conv-fun-lub: Sup = fun-lub Sup
⟨proof⟩

lemma le-conv-fun-ord: (≤) = fun-ord (≤)
⟨proof⟩

lemmas parallel-fixp-induct-2-1 = parallel-fixp-induct-uc
of - - - - case-prod - curry λ. x - λ. x,

where \( P = \lambda f \ g. \ P \ (\text{curry } f) \ g \),

unfolded case-prod-curry curry-case-prod curry-K,

\[
\text{OF - - - - - - refl refl}
\]

for \( P \)

lemma monotone-Pair:
\[
\begin{array}{l}
\text{\[ [ \text{monotone ord ord } a \ f; \text{monotone ord ord } b \ g ] \]}
\Rightarrow \text{monotone ord (rel-prod ord ord) (λx. (f x, g x))}
\end{array}
\]

⟨proof⟩

lemma cont-Pair:
\[
\begin{array}{l}
\text{\[ [ \text{cont lab ord ord } a \ f; \text{cont lab ord ord } b \ g ] \]}
\Rightarrow \text{cont lab ord (prod-lab ord ord) (rel-prod ord ord) (λx. (f x, g x))}
\end{array}
\]

⟨proof⟩

lemma mcont-Pair:
\[
\begin{array}{l}
\text{\[ [ \text{mcont lab ord ord } a \ f; \text{mcont lab ord ord } b \ g ] \]}
\Rightarrow \text{mcont lab ord (prod-lab ord ord) (rel-prod ord ord) (λx. (f x, g x))}
\end{array}
\]

⟨proof⟩

lemma mono2mono-emeasure-spmf [THEN lfp.mono2mono]:
\[ \text{shows monotone-emeasure-spmf:}
\begin{array}{l}
\text{monotone (ord-spmf (=)) (≤) (λp. emeasure (measure-spmf p))}
\end{array}
\]

⟨proof⟩

lemma cont-emeasure-spmf: cont lub-spmf (ord-spmf (=)) Sup (≤) (λp. emeasure (measure-spmf p))

⟨proof⟩

lemma mcont2mcont-emeasure-spmf [THEN lfp.mcont2mcont, cont-intro]:
\[ \text{shows mcont-emeasure-spmf: mcont lub-spmf (ord-spmf (=)) Sup (≤) (λp. emeasure (measure-spmf p))}
\]

⟨proof⟩

lemma mcont2mcont-emeasure-spmf': [THEN lfp.mcont2mcont, cont-intro]:
\[ \text{shows mcont-emeasure-spmf': mcont lub-spmf (ord-spmf (=)) Sup (≤) (λp. emeasure (measure-spmf p) A)}
\]

⟨proof⟩

lemma mcont-bind-pmf [cont-intro]:
\[ \text{assumes g:} \bigwedge y. \text{mcont lub ord lab-spmf (ord-spmf (=)) (g y)}
\]
\[ \text{shows mcont lub ord lab-spmf (ord-spmf (=)) (λx. bind-pmf p (λy. g y x))}
\]

⟨proof⟩

lemma ennreal-less-top-iff: \( x < \top \iff x \neq (\top :: \text{ennreal}) \)

⟨proof⟩

lemma type-definition-Domainp:
fixes Rep Abs A T
assumes type: type-definition Rep Abs A
assumes T-def: T ≡ (λ(x::'a) (y::'b). x = Rep y)
shows Domainp T = (λx. x ∈ A)
⟨proof⟩

context includes lifting-syntax begin

lemma weight-spmf-parametric [transfer-rule]:
(rel-spmf A ===> (=)) weight-spmf weight-spmf
⟨proof⟩

lemma lossless-spmf-parametric [transfer-rule]:
(rel-spmf A ===> (=)) lossless-spmf lossless-spmf
⟨proof⟩

lemma UNIV-parametric-pred: rel-pred R UNIV UNIV
⟨proof⟩

end

lemma bind-spmf-spmf-of-set:
∀A. [finite A; A ≠ {}] ⇒ bind-spmf (spmfofset A) = bind-pmf (pmfofset A)
⟨proof⟩

lemma set-pmf-bind-spmf: set-pmf (bind-spmf M f) = set-pmf M ≧ map-option-set (set-pmf o f)
⟨proof⟩

lemma set-pmf-spmf-of-set:
set-pmf (spmfofset A) = (if finite A ∧ A ≠ {} then Some ' A else {None})
⟨proof⟩

definition measure-measure-spmf :: 'a spmf ⇒ 'a set ⇒ real
where [simp]: measure-measure-spmf p = measure (measure-spmf p)

lemma measure-measure-spmf-parametric [transfer-rule]:
includes lifting-syntax shows
(rel-spmf A ===> rel-pred A ===> (=)) measure-measure-spmf measure-measure-spmf
⟨proof⟩

lemma of-nat-le-one-cancel-iff [simp]:
fixed n :: nat shows real n ≤ 1 ⇔ n ≤ 1
⟨proof⟩

lemma of-int-ceiling-less-add-one [simp]: of-int ⌈r⌉ < r + 1
⟨proof⟩

lemma lessThan-subset-Collect: {..<x} ⊆ Collect P ⇔ (∀ y<x. P y)

4
lemma spmf-ub-tight:
  assumes ab: \( \forall x. \text{spmf} \ p \ x \leq f \ x \)
  and sum: \( (\int_{x. \text{f} \ x} \text{dcount-space} \ \text{UNIV}) = \text{weight-spmf} \ p \)
  shows \( \text{spmf} \ p \ x = f \ x \)

2 Probabilistic while loop

locale loop-spmf =
  fixes guard :: 'a \Rightarrow \text{bool}
  and body :: 'a \Rightarrow 'a \text{ spmf}
begin

context notes [[\text{function-internals}]] begin

partial-function (\text{spmf}) \text{while} :: 'a \Rightarrow 'a \text{ spmf}
where \text{while} \ s = (\text{if} \ \text{guard} \ s \ \text{then} \ \text{bind-spmf} \ (\text{body} \ s) \ \text{while} \ \text{else} \ \text{return-spmf} \ s)
end

lemma while-fixp-induct [case-names adm bottom step]:
  assumes \text{spmf}. \text{admissible} \ P
  and \ P \ (\lambda\text{while}. \ \text{return-pmf} \ \text{None})
  and \( \text{while}'. \ P \text{ while}' \ \Rightarrow \ P \ (\lambda s. \ \text{if} \ \text{guard} \ s \ \text{then} \ \text{body} \ s \ \gg \ \text{while}' \ \text{else} \ \text{return-spmf} \ s)\)
  shows \ P \text{ while}
  (proof)

lemma while-simps:
  \text{guard} \ s \ \Rightarrow \ \text{while} \ s = \\text{bind-spmf} \ (\text{body} \ s) \ \text{while}
  \neg \ \text{guard} \ s \ \Rightarrow \ \text{while} \ s = \\text{return-spmf} \ s
  (proof)

end

lemma while-spmf-parametric [\text{transfer-rule}]:
  includes lifting-syntax shows
  \((S \Longrightarrow (\equiv)) \Longrightarrow (S \Longrightarrow \text{rel-spmf} \ S) \Longrightarrow S \Longrightarrow \text{rel-spmf} \ S)\)
loop-spmf.\text{while} \ loop-spmf.\text{while}
  (proof)

lemma loop-spmf-while-cong:
  [ guard = guard'; \( \\forall s. \ \text{guard}' \ s \ \Rightarrow \ \text{body} \ s = \text{body}' \ s \)]
  \Rightarrow \ loop-spmf.\text{while} \ \text{guard} \ \text{body} = \\text{loop-spmf.while} \ \text{guard'} \ \text{body}'
  (proof)
3 Rules for probabilistic termination

context loop-spmf begin

3.1 0/1 termination laws

lemma termination-0-1-immediate:
  assumes p: \( \forall s. \text{guard } s \Rightarrow \text{spmf } (\text{map-spmf guard } (\text{body } s)) \) False \( \leq p \)
  and p-pos: 0 \( < p \)
  and lossless: \( \forall s. \text{guard } s \Rightarrow \text{lossless-spmf } (\text{body } s) \)
  shows lossless-spmf (while s)
⟨proof⟩

primrec iter :: nat \( \Rightarrow \) 'a \( \Rightarrow \) 'a spmf
where
  iter 0 s = return-spmf s
| iter (Suc n) s = (if guard s then bind-spmf (body s) (iter n) else return-spmf s)

lemma iter-unguarded [simp]: \( \neg \) guard s \( \Rightarrow \) iter n s = return-spmf s
⟨proof⟩

lemma iter-bind-iter: bind-spmf (iter m s) (iter n) = iter (m + n) s
⟨proof⟩

lemma iter-Suc2: iter (Suc n) s = bind-spmf (iter n s) (\( \lambda s. \) if guard s then body s else return-spmf s)
⟨proof⟩

lemma lossless-iter: (\( \forall s. \) guard s \( \Rightarrow \) lossless-spmf (body s)) \( \Rightarrow \) lossless-spmf (iter n s)
⟨proof⟩

lemma iter-mono-emeasure1:
  emeasure (measure-spmf (iter n s)) \( \{ s. \neg \text{guard } s \} \) \( \leq \) emeasure (measure-spmf (iter (Suc n) s)) \( \{ s. \neg \text{guard } s \} \)
  (is ?lhs \( \leq \) ?rhs)
⟨proof⟩

lemma weight-while-conv-iter:
  weight-spmf (while s) = (SUP n. measure (measure-spmf (iter n s)) \( \{ s. \neg \text{guard } s \} \))
  (is ?lhs = ?rhs)
⟨proof⟩

lemma termination-0-1:
  assumes p: \( \forall s. \) guard s \( \Rightarrow \) p \( \leq \) weight-spmf (while s)
  and p-pos: 0 \( < p \)
  and lossless: \( \forall s. \) guard s \( \Rightarrow \) lossless-spmf (body s)
  shows lossless-spmf (while s)
⟨proof⟩
lemma termination-0-1-immediate-invar:
  fixes I :: 's ⇒ bool
  assumes p: ∀s. [ guard s; I s ] −→ spmf (map-spmf guard (body s)) False ≥ p
  and p-pos: 0 < p
  and lossless: ∀s. [ guard s; I s ] −→ lossless-spmf (body s)
  and invar: ∀s s'. [ s' ∈ set-spmf (body s); I s; guard s ] −→ I s'
  and I: I s
  shows lossless-spmf (loop-spmf.while guard body s)
  including lifting-syntax
⟨proof⟩
end

lemma termination-0-1-invar:
  fixes I :: 's ⇒ bool
  assumes p: ∀s. [ guard s; I s ] −→ p ≤ weight-spmf (loop-spmf.while guard body s)
  and p-pos: 0 < p
  and lossless: ∀s. [ guard s; I s ] −→ lossless-spmf (body s)
  and invar: ∀s s'. [ s' ∈ set-spmf (body s); I s; guard s ] −→ I s'
  and I: I s
  shows lossless-spmf (loop-spmf.while guard body s)
  including lifting-syntax
⟨proof⟩
end

3.2 Variant rule

class loop-spmf begin

lemma termination-variant:
  fixes bound :: nat
  assumes bound: ∀s. guard s −→ f s ≤ bound
  and step: ∀s. [ guard s; I s ] −→ p ≤ spmf (map-spmf (λs'. f s' < f s) (body s)) True
  and p-pos: 0 < p
  and lossless: ∀s. guard s −→ lossless-spmf (body s)
  shows lossless-spmf (while s)
⟨proof⟩
end

lemma termination-variant-invar:
  fixes bound :: nat and I :: 's ⇒ bool
  assumes bound: ∀s. [ guard s; I s ] −→ f s ≤ bound
  and step: ∀s. [ guard s; I s ] −→ p ≤ spmf (map-spmf (λs'. f s' < f s) (body s)) True
  and p-pos: 0 < p
  and lossless: ∀s. [ guard s; I s ] −→ lossless-spmf (body s)
  and invar: ∀s s'. [ s' ∈ set-spmf (body s); I s; guard s ] −→ I s'

and \( I: s \)

shows lossless-spmf (loop-spmf.while guard body s)

including lifting-syntax

⟨proof⟩

end

4 Distributions built from coin flips

4.1 The Bernoulli distribution

theory Bernoulli imports HOL−Probability.Probability begin

lemma zero-lt-num [simp]: \( 0 < (\text{numeral } n :: \cdot :: \{\text{canonically-ordered-monoid-add, semiring-char-0}\}) \)

⟨proof⟩

lemma ennreal-mult-numeral: ennreal \( x \) * numeral \( n = \) ennreal \( (x * \text{numeral } n) \)

⟨proof⟩

lemma one-plus-ennreal: \( 0 \leq x \Rightarrow 1 + \) ennreal \( x = \) ennreal \( (1 + x) \)

⟨proof⟩

We define the Bernoulli distribution as a least fixpoint instead of a loop because this avoids the need to add a condition flag to the distribution, which we would have to project out at the end again. As the direct termination proof is so simple, we do not bother to prove it equivalent to a while loop.

partial-function (spmfd) bernoulli :: real ⇒ bool spmf where

beroulli \( p \) = do \( b \leftarrow \) coin-spmf;
if \( b \) then return-spmf \( (p \geq \frac{1}{2}) \)
else if \( p < \frac{1}{2} \) then bernoulli \( (2 * p) \)
else bernoulli \( (2 * p - 1) \)

lemma pmf-bernoulli-None: pmf (bernoulli \( p \)) None = 0

⟨proof⟩

lemma lossless-bernoulli [simp]: lossless-spmf (bernoulli \( p \))

⟨proof⟩

lemma [simp]; assumes \( 0 \leq p p \leq 1 \)

shows bernoulli-True: spmf (bernoulli \( p \)) True = \( p \) (is ?True)

and bernoulli-False: spmf (bernoulli \( p \)) False = \( 1 - p \) (is ?False)

⟨proof⟩

lemma bernoulli-neg [simp]:

assumes \( p \leq 0 \)

shows bernoulli \( p \) = return-spmf False
lemma bernoulli-pos [simp]:
  assumes $1 \leq p$
  shows bernoulli $p = \text{return-spmf True}$

context begin interpretation pmf-as-function ⟨proof⟩
lemma bernoulli-eq-bernoulli-pmf:
  bernoulli $p = \text{spmf-of-pmf (bernoulli-pmf p)}$
⟨proof⟩
end

4.2 The geometric distribution

theory Geometric imports
  Bernoulli
  While-SPMF
begin

We define the geometric distribution as a least fixpoint, which is more elegant
than as a loop. To prove probabilistic termination, we prove it equivalent
to a loop and use the proof rules for probabilistic termination.

context notes [[function-internals]] begin
partial-function (spmf) geometric-spmf :: real ⇒ nat spmf where
  geometric-spmf $p = \{$
  $b \leftarrow \text{bernoulli } p;$
  $\quad \text{if } b \text{ then } \text{return-spmf 0 else map-spmf } (\{+\} 1) (\text{geometric-spmf } p)$
$\}$
end

lemma geometric-spmf-fixp-induct [case-names adm bottom step]:
  assumes spmf.admissible $P$
  and $P (\lambda \text{geometric-spmf. return-pmf None})$
  and $\forall \text{geometric-spmf}. \text{ P geometric-spmf }' \implies P (\lambda p. \text{ bernoulli } p \gg (\lambda b. \text{ if } b \text{ then } \text{return-spmf 0 else map-spmf } (\{+\} 1) (\text{geometric-spmf }' p)))$
  shows $P \text{ geometric-spmf}$
(proof)

lemma spmf-geometric-nonpos: $p \leq 0 \implies \text{geometric-spmf } p = \text{return-pmf None}$
(proof)

lemma spmf-geometric-ge-1: $1 \leq p \implies \text{geometric-spmf } p = \text{return-spmf 0}$
(proof)

context
  fixes $p :: \text{real}$
and body :: bool × nat ⇒ (bool × nat) spmf

defines [simp]: body ≡ λ(b, x). map-spmf (λb'. (¬ b', x + (if b' then 0 else 1)))

(bernoulli p)

begin

interpretation loop-spmf fst body
rewrites body ≡ λ(b, x). map-spmf (λb'. (¬ b', x + (if b' then 0 else 1)))
(bernoulli p)
(proof)

lemma geometric-spmf-conv-while:
shows geometric-spmf p = map-spmf snd (while (True, 0))
(proof)

lemma lossless-geometric [simp]: lossless-spmf (geometric-spmf p) ⟷ p > 0
(proof)

end

lemma spmf-geometric:
assumes p: 0 < p p < 1
shows spmf (geometric-spmf p) n = (1 − p) ^ n * p (is ?lhs n = ?rhs n)
(proof)

end

4.3 Arbitrary uniform distributions

theory Fast-Dice-Roll imports
  Bernoulli
  While-SPMF

begin

This formalisation follows the ideas by Jérémie Lumbroso [2].

lemma sample-bits-fusion:
  fixes v :: nat
  assumes 0 < v
  shows
  bind-pmf (pmf-of-set {..<v}) (λc. bind-pmf (pmf-of-set UNIV) (λb. f (2 * c + (if b then 1 else 0)))) =
  bind-pmf (pmf-of-set {..<2 * v}) f
  (is ?lhs = ?rhs)
(proof)

lemma sample-bits-fusion2:
  fixes v :: nat
  assumes 0 < v
  shows
  bind-pmf (pmf-of-set UNIV) (λb. bind-pmf (pmf-of-set {..<v}) (λc. f (c + v *
context fixes n :: nat notes [[function-internals]] begin

The check for $n \leq v$ should be done already at the start of the loop. Otherwise we do not see why this algorithm should be optimal (when we start with $v = n$ and $c = n - 1$, then it can go round a few loops before it returns something).

We define the algorithm as a least fixpoint. To prove termination, we later show that it is equivalent to a while loop which samples bitstrings of a given length, which could in turn be implemented as a loop. The fixpoint formulation is more elegant because we do not need to nest any loops.

partial-function (spmfs) fast-dice-roll :: nat ⇒ nat ⇒ nat spmf
where
fast-dice-roll v c =
(if $v \geq n$ then if $c < n$ then return-spmf c else fast-dice-roll ($v - n$) ($c - n$)
else do {
  $b \leftarrow$ coin-spmf;
  fast-dice-roll ($2 \ast v$) ($2 \ast c + (if b then 1 else 0)$)
})

lemma fast-dice-roll-fixp-induct [case-names adm bottom step]:
assumes spmf. admissible (λfast-dice-roll. P (curry fast-dice-roll))
and $P$ (λv c. return-spmf None)
and $fdr. P fdr \Rightarrow P$ (λv c. if $v \geq n$ then if $c < n$ then return-spmf c else fdr ($v - n$) ($c - n$)
else bind-spmf coin-spmf (λb. fdr ($2 \ast v$) ($2 \ast c + (if b then 1 else 0)$)))
shows $P$ fast-dice-roll
⟨proof⟩

definition fast-uniform :: nat spmf
where fast-uniform = fast-dice-roll 1 0

lemma spmf-fast-dice-roll-ub:
assumes $0 < v$
shows $\text{spmfs (bind-pmf (pmf-of-set \{..<v\}) (fast-dice-roll v)) x \leq (if x < n then 1 / n else 0)}$
⟨is lhs \leq rhs⟩
⟨proof⟩

lemma spmf-fast-uniform-ub:
$\text{spmfs fast-uniform x \leq (if x < n then 1 / n else 0)}$
⟨proof⟩

lemma fast-dice-roll-0 [simp]: fast-dice-roll 0 c = return-pmf None
⟨proof⟩
To prove termination, we fold all the iterations that only double into one big step

**definition** `fdr-step :: nat ⇒ nat ⇒ (nat × nat) spmf`

**where**

\[
\text{fdr-step } v \ c =
\]

(if \(v = 0\) then return-pmf None

else let \(x = 2 ^ \lceil \log 2 (\max 1 n) - \log 2 v \rceil \) in

map-spmf (λbs. (x * v, x * c + bs)) (spmf-of-set {..<x})

**lemma** `fdr-step-unfold`:

\[
\text{fdr-step } v \ c =
\]

(if \(v = 0\) then return-pmf None

else if \(n \leq v\) then return-spmf (v, c)

else do

\(b ← \text{coin-spmf};\)

\(\text{fdr-step } (2 * v) (2 * c + (if b then 1 else 0))\)

(is ?lhs = ?rhs is - = (if - then - else ?else))

⟨proof⟩

**lemma** `lossless-fdr-step` [simp]:

\[
\text{lossless-spmf } (fdr-step v c) ←→ v > 0
\]

⟨proof⟩

**lemma** `fast-dice-roll-alt-conv-while`:

\[
\text{fdr-alt } v \ c =
\]

map-spmf snd (bind-spmf (fdr-step v c) (loop-spmf.while (λ(v, c). n ≤ c) (λ(v, c). fdr-step (v - n) (c - n))))

⟨proof⟩

**lemma** `fast-dice-roll-alt`:

\[
\text{fast-dice-roll-alt } \triangleq \text{fast-dice-roll}
\]

⟨proof⟩

**lemma** `fast-dice-roll-n0`:

12
assumes \( n = 0 \)
shows \( \text{fast-dice-roll } v \ c = \text{return-pmf } \text{None} \)
\hfill (proof)

lemma lossless-fast-uniform \[\text{simp; lossless-spmf } \text{fast-uniform } \iff n > 0\]
\hfill (proof)

lemma spmf-fast-uniform: \( \text{spmf } \text{fast-uniform } x = (\text{if } x < n \text{ then } 1 / n \text{ else } 0) \)
\hfill (proof)

end

lemma fast-uniform-cone-uniform: \( \text{fast-uniform } n = \text{spmf-of-set } \{..<n\} \)
\hfill (proof)

end

theory Resampling imports
While-SPMF
begin

lemma ord-spmf-lossless:
assumes ord-spmf \((=) p q \text{ lossless-spmf } p\)
shows \( p = q \)
\hfill (proof)

context notes \[[\text{function-internals}]\]
begin

partial-function \((\text{spmf})\) resample :: \(a\ \text{set} \Rightarrow a \text{ set} \Rightarrow a\ \text{spmf}\) where
\(\text{resample } A\ B = \text{bind-spmf } (\text{spmf-of-set } A) (\lambda x. \text{if } x \in B \text{ then return-spmf } x \text{ else resample } A\ B)\)

end

lemmas resample-fixp-induct\[\text{case-names adm bottom step} = \text{resample.fixp-induct}\]

context
\hfill (proof)

fixes \(A:: a\ \text{set}\)
and \(B:: a\ \text{set}\)
begin

interpretation loop-spmf \(\lambda x. x \notin B \lambda -. \text{spmf-of-set } A\)
\hfill (proof)

lemma resample-coro-while: \(\text{resample } A\ B = \text{bind-spmf } (\text{spmf-of-set } A) \text{ while}\)
\hfill (proof)

context
\hfill (proof)
assumes \(A:: \text{finite } A\)
and $B: B \subseteq A B \neq \{\}$

begin

private lemma A-nonempty: $A \neq \{\}$
⟨proof⟩ lemma B-finite: finite $B$
⟨proof⟩

lemma lossless-resample: lossless-spmf (resample $A B$)
⟨proof⟩

lemma resample-le-sample:
ord-spmf (=) (resample $A B$) (spm-to-set $B$)
⟨proof⟩

lemma resample-eq-sample: resample $A B$ = spmf-of-set $B$
⟨proof⟩

end

end

end

References

