Probabilistic while loop

Andreas Lochbihler

April 20, 2020

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 ....................................... 14

4 Distributions built from coin flips 16
  4.1 The Bernoulli distribution ....................... 16
  4.2 The geometric distribution ....................... 19
  4.3 Arbitrary uniform distributions .................. 21

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
by(cases x) simp-all

lemma None-in-map-option-set-None [intro!]: None ∈ map-option-set f None
by simp

lemma None-in-map-option-set-Some [intro!]: None ∈ f x ⇒ None ∈ map-option-set f (Some x)
by simp

lemma Some-in-map-option-set [intro!]: Some y ∈ map-option-set f x ⇒ Some y ∈ f (Some x)
by simp

lemma map-option-set-singleton [simp]: map-option-set (λx. {f x}) y = {Option.bind y f}
by(cases y) simp-all

lemma Some-eq-bind-conv: Some y = Option.bind x f ⇐⇒ (∃ z. x = Some z ∧ f z = Some y)
by(cases x) auto

lemma map-option-set-bind: map-option-set f (Option.bind x g) = map-option-set (map-option-set f o g) x
by(cases x) simp-all

lemma Some-in-map-option-set-conv: Some y ∈ map-option-set f x ⇐⇒ (∃ z. x = Some z ∧ Some y ∈ f z)
by(cases x) auto

interpretation rel-spmf-characterisation by unfold-locales(rule rel-pmf-measureI)
hide-fact (open) rel-pmf-measureI

lemma Sup-conv-fun-lub: Sup = fun-lub Sup
by(auto simp add: Sup-fun-def fun-eq-iff fun-lub-def intro: arg-cong[where f=Sup])

lemma le-conv-fun-ord: (≤) = fun-ord (≤)
by(auto simp add: fun-eq-iff fun-ord-def le-fun-def)

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

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

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

\( \text{OF} - - - - - - \text{refl refl} \)

for \( P \)

**lemma monotone-Pair:**

\[
\begin{align*}
\text{monotone ord orda f ; monotone ord ordb g} \\
\Rightarrow \text{monotone ord (rel-prod orda ordb) (λx. (f x, g x))}
\end{align*}
\]

by (simp add: monotone-def)

**lemma cont-Pair:**

\[
\begin{align*}
\text{cont lub ord luba orda f ; cont lub ord lubb ordb g} \\
\Rightarrow \text{cont lub ord (prod-lub luba lubb) (rel-prod orda ordb) (λx. (f x, g x))}
\end{align*}
\]

by (rule contI) (auto simp add: prod-lub-def image-image dest: contD)

**lemma mcont-Pair:**

\[
\begin{align*}
\text{mcont lub ord luba orda f ; mcont lub ord lubb ordb g} \\
\Rightarrow \text{mcont lub ord (prod-lub luba lubb) (rel-prod orda ordb) (λx. (f x, g x))}
\end{align*}
\]

by (rule mcontI) (simp-all add: monotone-Pair mcont-mono cont-Pair)

**lemma mono2mono-emeasure-spmf**

\[
\text{shows monotone-emeasure-spmf : monotone (ord-spmf (=)) (≤) (λp. emeasure (measure-spmf p))}
\]

by (rule monotoneI le-funI ord-spmf-eqD-emeasure+)

**lemma cont-emeasure-spmf**:

\[
\text{cont lub-spmf (ord-spmf (=)) Sup (≤) (λp. emeasure (measure-spmf p))}
\]

by (rule contI) (simp add: emeasure-lub-spmf fun-eq-iff image-comp)

**lemma mcont2mcont-emeasure-spmf**

\[
\text{shows mcont-emeasure-spmf : mcont lub-spmf (ord-spmf (=)) Sup (≤) (λp. emeasure (measure-spmf p))}
\]

by (simp add: mcont-def monotone-emeasure-spmf cont-emeasure-spmf)

**lemma mcont2mcont-emeasure-spmf'**

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

using mcont-emeasure-spmf [unfolded Sup-conv-fun-lub le-conv-fun-ord] by (subst (asm) mcont-fun-lub-apply) blast

**lemma mcont-bind-pmf**

\[
\text{assumes g: \( \forall y. \ \text{mcont luba orda lub-spmf (ord-spmf (=)) (g y) \) }
\]

\text{shows mcont luba lub-spmf (ord-spmf (=)) (λx. bind-pmf p (λy. g y x))}

using mcont-bind-spmf [where \( f = \lambda. \ \text{spmfn-of-pmf p} \ \text{and g=g, OF - assms} \)] by (simp)

**lemma ennreal-less-top-iff**:

\[
x < T \iff x \neq (T :: ennreal)
\]

by (cases x) simp-all
lemma type-definition-Domainp:
fixes Rep Abs A T
assumes type: type-definition Rep Abs A
assumes T-def: \( T \equiv (\lambda (x::'a) (y::'b). x = \text{Rep } y) \)
shows Domainp \( T = (\lambda x. x \in A) \)
proof –
interpret type-definition Rep Abs A by (rule type)
qed

countext includes lifting-syntax begin

lemma weight-spmf-parametric [transfer-rule]:
(\text{rel-spmf } A ===> (=)) weight-spmf weight-spmf
by (simp add: rel-fun-def rel-spmf-weightD)

lemma lossless-spmf-parametric [transfer-rule]:
(\text{rel-spmf } A ===> (=)) lossless-spmf lossless-spmf
by (simp add: rel-fun-def lossless-spmf-def rel-spmf-weightD)

lemma UNIV-parametric-pred: rel-pred R UNIV UNIV
by (auto intro!: rel-predI)
end

lemma bind-spmf-spmf-of-set:
\( \forall A. \{ \text{finite } A; A \neq \{\} \} \implies \text{bind-spmf } (\text{spmfof-set } A) = \text{bind-pmf } (\text{pmfof-set } A) \)

lemma set-pmf-bind-spmf: set-pmf (bind-spmf M f) = set-pmf (\map option-set (set-pmf f))
by (auto 3 simp add: bind-spmf-def split: option.splits intro: rev-bexI)

lemma set-pmf-spmf-of-set:
\( \text{set-pmf } (\text{spmfof-set } A) = (\text{if finite } A \land A \neq \{\} \text{ then Some } 'A \text{ else } \{\text{None}\}) \)

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

lemma measure-measure-spmf-parametric [transfer-rule]:
includes lifting-syntax shows
(\text{rel-spmf } A ===> rel-pred A ===> (=)) measure-measure-spmf measure-measure-spmf
unfolding measure-measure-spmf-def [abs-def] by (rule measure-spmf-parametric)

lemma of-nat-le-one-cancel-iff [simp]:
fixes n :: nat shows real n \leq 1 \longleftrightarrow n \leq 1
by linarith
lemma of-int-ceiling-less-add-one [simp]: of-int ⌈r⌉ < r + 1
by linarith

lemma lessThan-subset-Collect: {..<x} ⊆ Collect P ↔ (∀ y < x. P y)
by(auto simp add: lessThan-def)

lemma spmf-ub-tight:
assumes ab: ∀ x. spmf p x ≤ f x
and sum: (∫ y. f y ∅ count-space UNIV) = weight-spmf p
shows spmf p x = f x
proof -
  have [rule-format]: ∀ x. f x ≤ spmf p x
  proof (rule ccontr)
  assume ¬ ?thesis
  then obtain x where x: spmf p x < f x by (auto simp add: not-le)
  have *: (∫ y. ennreal (f y) * indicator (- {x}) y ∅ count-space UNIV) ≠ ⊤
by (rule neq-top-trans[where y=weight-spmf p], simp)
intro!: nn-integral-mono split: split-indicator
  have weight-spmf p = (∫ y. ennreal (f y) * indicator (- {x}) y ∅ count-space UNIV)
  intros simp: nn-integral-cong split: split-indicator intro: enreal-leI
also have ... ≤ (∫ y. ennreal (f y) * indicator (- {x}) y ∅ count-space UNIV)
also have ... < (∫ y. ennreal (f y) * indicator (- {x}) y ∅ count-space UNIV)
using ub by (intro add-mono nn-integral-mono)(auto split: split-indicator intro: enreal-leI)
also have ... = (∫ y. ennreal (f y) ∅ count-space UNIV)
using * x by (simp add: enreal-less-iff)
also have ... = (∫ y. ennreal (f y) ∅ count-space UNIV)
by (subst nn-integral-add[symmetric])(auto intro: nn-integral-cong split: split-indicator)
also have ... = weight-spmf p using sum by simp
finally show False by simp
qed
from this[of x] ub[of x] show ?thesis by simp
qed

2 Probabilistic while loop

locale loop-spmf =
fixes guard :: 'a ⇒ bool
and body :: 'a ⇒ 'a spmf
begin
context
notes [[function-internals]] begin


partial-function (spmf) while :: 'a ⇒ 'a spmf
where while s = (if guard s then bind-spmf (body s) while else return-spmf s)
end

lemma while-fixp-induct [case-names adm bottom step]:
  assumes spmf.admissible P
  and P (λwhile. return-pmf None)
  and (λwhile'. P while' =⇒ P (λs. if guard s then body s >>= while' else return-spmf s))
  shows P while
  using assms by (rule while.fixp-induct)

lemma while-simps:
  guard s =⇒ while s = bind-spmf (body s) while
  ~ guard s =⇒ while s = return-spmf s
by (rewrite while.simps; simp; fail)+
end

lemma while-spmf-parametric [transfer-rule]:
  includes lifting-syntax shows
  ((S ===> (=)) ===> (S ===> rel-spmf S) ===> S ===> rel-spmf S)
  loop-spmf.while loop-spmf.while
unfold loop-spmf.while-def[abs-def]
apply (rule rel-fun1)
apply (rule rel-fun1)
apply (rule fixp-spmf-parametric[OF loop-spmf.while.monot loop-spmf.while.mono])
subgoal premises by transfer-prover
done

lemma loop-spmf-while-cong:
  [[ guard = guard'; \s. guard' s =⇒ body s = body' s ]]
  =⇒ loop-spmf.while guard body = loop-spmf.while guard' body'
unfold loop-spmf.while-def[abs-def] by (simp cong: if-cong)

3 Rules for probabilistic termination

context loop-spmf begin

3.1 0/1 termination laws

lemma termination-0-1-immediate:
  assumes p: \s. guard s =⇒ spmf (map-spmf guard (body s)) False ≥ p
  and p-pos: 0 < p
  and lossless: \s. guard s =⇒ lossless-spmf (body s)
  shows lossless-spmf (while s)
proof –
have \( \forall s. \) lossless-spmf (while s)

proof (rule ccontr)

  assume \( \neg \) thesis

  then obtain s where s: \( \neg \) lossless-spmf (while s) by blast

  hence True: guard s by (simp add: while.simps split: if-split-_asm)

  from \( p[\text{OF this}] \) have p-le-1: \( p \leq 1 \) using pmf-le-1 by (rule order_trans)

  have new-bound: \( p \ast (1 - k) + k \leq \text{weight-spmf} \) (while s)

  if \( k: 0 \leq k \leq 1 \) and k-le: \( \forall s. k \leq \text{weight-spmf} \) (while s) for k s

  proof (cases guard s)

  case False

    have p * (1 - k) + k \( \leq 1 \) * (1 - k) + k using p-le-1 k by (intro

    mult-right-mono add-mono; simp)

    also have \( \ldots \leq 1 \) by simp

  finally show ?thesis using False by (simp add: while.simps)

next

  case True

  let \( ?M = \lambda s. \text{measure-spmf} \) (body s)

  have bounded: \( \int s''. \text{weight-spmf} \) (while s'') \( \partial \?M s'' \) \( \leq 1 \) for s''

    using integral-nonneg-AE[of \( \lambda s''. \) weight-spmf (while s'') \( \partial \?M s'' \)]

    by (auto simp add: weight-spmf-nonneg weight-spmf-le-1 intro!: measure-spmf.nn-integral-le-const

    integral-real-bounded)

    have p \( \leq \) measure (?M s) \{ s', \( \neg \) guard s' \} using p[OF True]

    by (simp add: spmf-cone.measure-spmf measure-map-spmf vimage_def)

    hence p * (1 - k) + k \( \leq \) measure (?M s) \{ s', \( \neg \) guard s' \} * (1 - k) + k

    using k by (intro add-mono mult-right-mono)(simp-all)

    also have \( \ldots = \int s'. \) indicator \{ s', \( \neg \) guard s' \} \( s' \ast (1 - k) + k \ \partial \?M s'' \)

    using True by (simp add: ennreal-less-top-iff lossless lossless-weight-spmfD)

    also have \( \ldots = \int s'. \) indicator \{ s', \( \neg \) guard s' \} \( s' + \) indicator \{ s', guard s' \}

    \( s' \ast k \ \partial \?M s'' \)

    by (rule Bochner-Integration.integral-cong)(simp-all split: split-indicator)

    also have \( \ldots \leq \int s'. \) indicator \{ s', \( \neg \) guard s' \} \( s' + \) indicator \{ s', guard s' \}

    \( s' \ast \int s'''. \text{weight-spmf} \) (while s''') \( \partial \?M s''' \partial \?M s'' \)

    using k bounded

    by (intro integral-mono integrable-add measure-spmf.integrable-const-bound[where

    B=1] add-mono mult-left-mono)

    (simp-all add: weight-spmf-nonneg weight-spmf-le-1 mult-le-one k-le split:

    split-indicator)

    also have \( \ldots = \int s'. \) (if \( \neg \) guard s' then 1 else \( \int s'''. \) weight-spmf (while s''')) \( \partial \?M s'''' \)

    \( \partial \?M s'''' \)

    by (rule Bochner-Integration.integral-cong)(simp-all split: split-indicator)

    also have \( \ldots = \int s'. \) weight-spmf (while s') \( \partial \text{measure-spmf} \) (body s)

    by (rule Bochner-Integration.integral-cong; simp add: while.simps weight-bind-spmf

    o-def)

    also have \( \ldots = \text{weight-spmf} \) (while s) using True
\begin{verbatim}
  by(simp add: while.simps weight-bind-spmf o_def)
finally show \textit{thesis}.
qed

define k where \( k \equiv \inf s. \text{weight-spmf} (\text{while } s) \)
define k' where \( k' \equiv p \cdot (1 - k) + k \)
from s have \( \text{weight-spmf} (\text{while } s) < 1 \)
  using \text{weight-spmf[le-1]}[\text{of while } s]
by(simp add: lossless-spmf-def)
then have \( k < 1 \)
unfolding k-def by(rewrite cINF-less-iff)(auto intro!:bdd-belowI2 weight-spmf-nonneg)

have \( 0 \leq k \)
  unfolding k-def by(auto intro!:cINF-greatest simp add: weight-spmf-nonneg)
moreover have \( k \leq \text{weight-spmf} (\text{while } s) \)
  for s
unfolding k-def by(rule cINF-lower)(auto intro!:bdd-belowI2 weight-spmf-nonneg)
ultimately have \( \forall s. k' \leq \text{weight-spmf} (\text{while } s) \)
unfolding k'-def by(rule new-bound)

have \( k < k' \)
  using p-pos \( \langle k < 1 \rangle \)
by(auto simp add:k'-def)
finally show False
by simp
qed

thus \textit{thesis} by blast
qed

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

lemma iter-unguarded [simp]: \( \neg \text{guard } s \Rightarrow \text{iter } n \ s = \text{return-spmf } s \)
  by(induction n arbitrary: s simp-all)

lemma iter-bind-iter: \( \text{bind-spmf } (\text{iter } m \ s) \text{ (iter } n) = \text{iter } (m + n) \ s \)
  by(induction m arbitrary: s simp-all)

lemma iter-Suc2: \( \text{iter } (\text{Suc } n) \ s = \text{bind-spmf } (\text{iter } n \ s) \text{ (\text{\lambda s. if guard } s \text{ then body } s \text{ else return-spmf } s)}) \)
using iter-bind-iter[of n s 1, symmetric]
by(simp del: iter.simps)(rule bind-spmf-cong; simp cong: bind-spmf-cong)

lemma lossless-iter: \( \forall s. \neg \text{guard } s \Rightarrow \text{lossless-spmf } (\text{body } s) \Rightarrow \text{lossless-spmf } (\text{iter } n \ s) \)
  by(induction n arbitrary: s simp-all)

lemma iter-mono-emeasure1:
  \( \text{emeasure } (\text{measure-spmf } (\text{iter } n \ s)) \{ s. \neg \text{guard } s \} \leq \text{emeasure } (\text{measure-spmf } (\text{iter } (\text{Suc } n) \ s)) \{ s. \neg \text{guard } s \} \)
(is \text{?lhs} \leq \text{?rhs})
proof(cases guard s)
\end{verbatim}
case True
have ?lhs = emeasure (measure-spmf (bind-spmf (iter n s) return-spmf)) {s. ¬ guard s} by simp
also have ... = ∫⁺ s’. emeasure (measure-spmf (return-spmf s’)) {s. ¬ guard s} by (simp del: bind-return-spmf add: measure-spmf-bind o-def emeasure-bind)
also have ... ≤ ∫⁺ s’. emeasure (measure-spmf (if guard s’ then body s’ else return-spmf s’)) {s. ¬ guard s} by (simp add: measure-spmf-return-spmf)
also have ... = ?rhs by (simp add: iter-Suc2 measure-spmf-bind o-def emeasure-bind)
finally show ?thesis . qed simp

lemma weight-while-conv-iter: weight-spmf (while s) = (SUP n. measure (measure-spmf (iter n s)) {s. ¬ guard s})
(is ?lhs = ?rhs)
proof (rule antisym)
have emeasure (measure-spmf (while s)) UNIV ≤ (SUP n. emeasure (measure-spmf (iter n s)) {s. ¬ guard s})
(is - ≤ (SUP n. ?f n s))
proof (induction arbitrary: s rule: while-fixp-induct)
case adm show ?case by simp
case bottom show ?case by simp
case (step while’)
show ?case (is ?lhs’ ≤ ?rhs’)
proof(cases guard s)
case True
have inc: inseq ?f by (rule incseq-Suc le-funI iter-mono-emeasure1)
also have ... ≤ (SUP n. ?f n s) by (simp add: measure-spmf-bind o-def emeasure-bind)
also have ... = (SUP n. ∫⁺ s’. ?f n s’ ∂ measure-spmf (body s)) using inc
by (rule nn-integral-monotone-convergence-SUP simp-all)
also have ... = (SUP n. ?f (Suc n) s) using True
by (simp add: measure-spmf-bind o-def emeasure-bind)
also have ... ≤ (SUP n. ?f n s) by (rule SUP-mono)
finally show ?thesis .

next
  case False
then have \(\text{lhs}' = \text{emeasure} (\text{measure-spmf} (\text{iter} 0 s)) \{s. \neg \text{guard } s\}\)
  by(simp add: measure-spmf-return-spmf)
also have \(\ldots \leq \text{rhs}'\) by(rule SUP-upper) simp
finally show \(\text{thesis}\).
qed

also have \(\ldots = \text{ennreal} (\text{SUP } n. \text{measure} (\text{measure-spmf} (\text{iter } n s)) \{s. \neg \text{guard } s\})\)
  by(subst ennreal-SUP) fold measure-spmf.emeasure-eq-measure, auto simp add:
  not-less measure-spmf.subprob-emeasure-le-1 intro!: exI [where x=1]
also have \(0 \leq (\text{SUP } n. \text{measure} (\text{measure-spmf} (\text{iter } n s)) \{s. \neg \text{guard } s\})\)
  by(rule cSUP-upper2) (auto intro!: bdd-aboveI [where M=1] simp add:
    measure-spmf.space-measure-spmf)
ultimately show \(\text{lhs} \leq \text{rhs}\) by(simp add: measure-spmf.emeasure-eq-measure
    space-measure-spmf)

show \(\text{rhs} \leq \text{lhs}\)
proof(rule cSUP-least)
  show \(\text{measure} (\text{measure-spmf} (\text{iter } n s)) \{s. \neg \text{guard } s\} \leq \text{weight-spmf} (\text{while } s)\) \(\{\text{is } \varphi (n s s) \leq -\}\) for \(n\)
proof(induction \(n\) arbitrary: \(s\))
case 0
  show \(\text{thesis}\) by(simp add: measure-spmf-return-spmf measure-return while-simps split:
    split-indicator)
next
case (Suc \(n\))
  show \(\text{thesis}\)
  proof(cases \(\text{guard } s\))
case True
    have \(\varphi (\text{Suc } n) s = \int^+ s' . \varphi n s' \text{ measure-spmf} (\text{body } s)\)
    using True unfolding measure-spmf.emeasure-eq-measure[symmetric]
    by(simp add: measure-spmf-bind o-def emeasure-bind[where \(N=\text{measure-spmf}\)
      space-measure-spmf Pi-def space-subprob-algebra]
    also have \(\ldots \leq \int^+ s' . \text{weight-spmf} (\text{while } s') \text{ measure-spmf} (\text{body } s)\)
    by(rule nn-integral-mono ennreal-leI Suc.IH)+
    also have \(\ldots = \text{weight-spmf} (\text{while } s)\)
    using True unfolding measure-spmf.emeasure-eq-measure[symmetric]
    space-measure-spmf
    by(simp add: while-simps measure-spmf-bind o-def emeasure-bind[where \(N=\text{measure-spmf}\]
      space-measure-spmf Pi-def space-subprob-algebra]
    finally show \(\text{thesis}\) by(simp)
next
case False then show \(\text{thesis}\)
  by(simp add: measure-spmf-return-spmf measure-return while-simps split:
    split-indicator)
qed
qed
qed
lemma termination-0-1:
assumes \( p: \forall s. \text{guard} \ s \Longrightarrow p \leq \text{weight-spmf} \ (\text{while} \ s) \)
and \( p\text{-pos}: 0 < p \)
and lossless: \( \forall s. \text{guard} \ s \Longrightarrow \text{lossless-spmf} \ (\text{body} \ s) \)
shows lossless-spmf (while s)
unfolding lossless-spmf-def

proof (rule antisym)
let ?\( X = \{s. \neg \text{guard} \ s\}\)
show weight-spmf (while s) \( \leq 1 \)
by (rule weight-spmf-le-1)
define \( p' \) where \( p' \equiv p / 2 \)
have \( p'\text{-pos}: p' > 0 \) and \( p' < p \)
using p-pos by (simp-all add: \( p'	ext{-def} \))

have \( \exists n. p' < \text{measure} \ (\text{measure-spmf} \ (\text{iter} \ n \ s) ) \) ?\( X \) if guard s for s
using p[OF that \( \langle p' < p \rangle \)]
unfolding weight-while-conv-iter
by (subst (asm) le-cSUP-iff)(auto intro: measure-spmf.subprob-measure-le-1)
then obtain \( N \) where \( p': p' \leq \text{measure} \ (\text{measure-spmf} \ (\text{iter} \ (N \ s) ) ) \) ?\( X \) if guard s for s
using p by atomize-elim (rule choice, force dest: order.strict-implies-order)

interpret fuse: loop-spmf guard \( \lambda s. \text{iter} \ (N \ s) \) s.

have \( 1 = \text{weight-spmf} \ (\text{fuse} \ . \text{while} \ s) \)
by (rule lossless-weight-spmfD[ symmetric])
(rule fuse.termination-0-1-immediate; auto simp add: spmf-map vimage-def
intro: \( p'\text{-pos} \) lossless-iter lossless)
also have \( \ldots \leq \bigcup n. \text{measure} \ (\text{measure-spmf} \ (\text{iter} \ n \ s) ) \) ?\( X \)
unfolding fuse. weight-while-cone-iter
proof (rule cSUP-least)
fix \( n \)
have \( \text{emeasure} \ (\text{measure-spmf} \ (\text{fuse} \ . \text{iter} \ n \ s) ) \) ?\( X \) \( \leq (\text{SUP} \ n. \text{emeasure} \ (\text{measure-spmf} \ (\text{iter} \ (N \ s) ) ) ) \) ?\( X \)
proof (induction \( n \) arbitrary: \( s \))
  case 0 show ?case by (auto intro!: SUP-upper2[where \( i=0 \)])
next
  case \( \text{Suc} \ n \)
  have inc: incseq \( (\lambda n \ s'. \text{emeasure} \ (\text{measure-spmf} \ (\text{iter} \ n \ s' ) ) ) \) ?\( X \)
  by (rule incseq-SucI le-funI iter-mono-emeasure I+)

  have \( \text{emeasure} \ (\text{measure-spmf} \ (\text{fuse} \ . \text{iter} \ (\text{Suc} \ n) \ s) ) ) \) ?\( X \) \( = \text{emeasure} \ (\text{measure-spmf} \ (\text{iter} \ (N \ s) s \gg fuse \ . \text{iter} \ n \ )) \) ?\( X \)
  by simp
  also have \( \ldots = \int^+ s'. \text{emeasure} \ (\text{measure-spmf} \ (\text{fuse} \ . \text{iter} \ n \ s' ) ) \) ?\( X \)
  \text{measure-spmf} \ (\text{iter} \ (N \ s) s)
  by (simp add: measure-spmf-bind o-def emeasure-bind[where \( N=\text{measure-spmf} \) \text{- space-measure-spmf Pi-def space-subprob-algebra}]
  \text{also have} \( \ldots \leq \int^+ s'. (\text{SUP} \ n. \text{emeasure} \ (\text{measure-spmf} \ (\text{iter} \ n \ s' ) ) ) \) ?\( X \)
\text{measure-spmf} \ (\text{iter} \ (N \ s) s)

by (rule nn-integral-mono Suc.IH) +
also have ... = (SUP n. \int + s' emeasure (measure-spmf (iter n s')) ?X
\partial measure-spmf (iter (N s s)))
  by (rule nn-integral-monotone-convergence-SUP[OF inc]) simp
also have ... = (SUP n. emeasure (measure-spmf (bind-spmf (iter (N s s))
(iter n))) ?X)
by (simp add: measure-spmf-bind o-def emeasure-bind[where N=measure-spmf
space-measure-spmf Pi-def space-subprob-algebra]
also have ... = (SUP n. emeasure (measure-spmf (iter (N s + n) s)) ?X)
by (simp add: iter-bind-iter)
also have ... \leq (SUP n. emeasure (measure-spmf (iter n s)) ?X) by (rule
SUP-mono) auto
finally show ?case .
qed
also have ... = ennreal (SUP n. measure (measure-spmf (iter n s)) ?X)
by (subt ennreal-SUP)[fold measure-spmf.emeasure-eq-measure, auto simp
add: not-less measure-spmf.subprob-emeasure-le-1 intro!: exI[where x=1])
also have \text{0 \leq (SUP n. measure (measure-spmf (iter n s)) ?X)
by (rule SUP-upper2)[auto intro!:bdd-aboveI[where M=1] simp add: measure-spmf.subprob-emeasure-le-1]
ultimately show measure (measure-spmf (fuse.iter n s)) ?X \leq ...}
by (simp add: measure-spmf.emeasure-eq-measure)
qed
finally show \text{1 \leq weight-spmf (while s) unfolding weight-while-conv-iter} .
qed
end

lemma termination-0-1-immediate-invar:
fixes \text{I :: 's = \Rightarrow bool}
assumes \text{p: \\\true s. \\[ guard s; I s \\] \Rightarrow spmf (map-spmf guard (body s)) False \geq p}
and \text{p-pos: 0 < p}
and \text{lossless: \\\true s. \\[ guard s; I s \\] \Rightarrow lossless-spmf (body s)}
and \text{invar: \\\true s s'. \\[ s' \in set-spmf (body s); I s; guard s \\] \Rightarrow I s'}
and \text{i: I s}
shows \text{lossless-spmf (loop-spmf.while guard body s)}
including lifting-syntax

proof –
{ \text{assume \text{\exists (Rep :: 's' = 's) Abs. type-definition Rep Abs {s, I s}}} } \text{then  obtain Rep :: 's' = 's and Abs where td: type-definition Rep Abs {s, I s)}}
  by blast
then interpret \text{td: type-definition Rep Abs {s, I s} .}
define \text{cr where cr \equiv \lambda x y. x = Rep y}
have \text{[transfer-rule]: bi-unique cr right-total cr using td cr-def by (rule typedef-bi-unique
typedef-right-total)+}
  have \text{[transfer-domain-rule]: Domainp cr = I using type-definition-Domainp[OF
td cr-def] by simp}
define \text{guard' where guard' \equiv (Rep ----> id) guard}
have \text{[transfer-rule]: (cr ----> (=)) guard guard' by (simp add: rel-fun-def}
\textbf{cr-def guard' def)}
\begin{verbatim}
    define body1 where body1 \equiv \lambda s. if guard s then body s else return-pmf None
    define body1' where body1' \equiv (\text{Rep} \longrightarrow \text{map-spmf} \text{ Abs}) body1
    have [transfer-rule]: (cr \Longrightarrow \text{rel-spmf cr}) body1 body1'
    by (auto simp add: \text{rel-fun-def} body1' def body1 def \text{cr-def spmf-rel-map td. Rep[simplified]}
        invar td.\text{Abs-inverse intro}: \text{rel-spmf-refII})
    define s' where s' \equiv Abs s
    have [transfer-rule]: cr s s' by (simp add: s' def cr-def I td.\text{Abs-inverse})
\end{verbatim}

\begin{verbatim}
    have \forall s. guard' s \Longrightarrow p \leq \text{spmf} (\text{map-spmf} \text{ guard'} (body1' s)) False
    by (transfer fixing: p)(simp add: body1-def p)
    moreover note p-pos
    moreover have \forall s. guard' s \Longrightarrow \text{lossless-spmf} (body1' s) by transfer(simp add: lossless-body1-def)
    ultimately have \text{lossless-spmf} (\text{loop-spmf.while guard'} body1' s') by (rule loop-spmf.termination-0-1-immediate)
    hence \text{lossless-spmf} (\text{loop-spmf.while guard body1 s}) by transfer }
from this[cancel-type-definition] I show \?thesis by (auto cong: loop-spmf.while-cong)
qued
\end{verbatim}

\textbf{lemma termination-0-1-invar:}
\begin{verbatim}
fixes I :: 's \Rightarrow bool
assumes p: \forall s. [\text{guard s; I s}] \Longrightarrow p \leq \text{weight-spmf} (\text{loop-spmf.while guard body s})
    and p-pos: 0 < p
    and lossless: \forall s. [guard s; I s] \Longrightarrow \text{lossless-spmf} (body s)
    and invar: \forall s s'. [s' \in \text{set-spmf} (body s); I s; guard s] \Longrightarrow I s'
    and I; I s
shows \text{lossless-spmf} (\text{loop-spmf.while guard body s})
including lifting-syntax
proof
\begin{verbatim}
{ assume \exists (\text{Rep} :: 's \Rightarrow 's) \text{Abs. type-definition Rep Abs} \{s, I s\}
    then obtain \text{Rep} :: 's \Rightarrow 's and \text{Abs where td: type-definition Rep Abs} \{s, I s\}
    by blast
    then interpret td: \text{type-definition Rep Abs} \{s, I s\}.
    define cr where cr \equiv \lambda x y. x = \text{Rep} y
    have [transfer-rule]: bi-unique cr right-total cr using td cr-def by (rule typedef-bi-unique typedef-right-total)+
    have [transfer-domain-rule]: Domainp cr = I using type-definition-Domainp[OF
td cr-def] by simp
\end{verbatim}
\begin{verbatim}
    define guard' where guard' \equiv (\text{Rep} \longrightarrow id) guard
    have [transfer-rule]: (cr \Longrightarrow (\_)) guard guard' by (simp add: rel-fun-def cr-def guard' def)
    define body1 where body1 \equiv \lambda s. if guard s then body s else return-pmf None
    define body1' where body1' \equiv (\text{Rep} \longrightarrow \text{map-spmf} \text{ Abs}) body1
    have [transfer-rule]: (cr \Longrightarrow \text{rel-spmf cr}) body1 body1'
    by (auto simp add: \text{rel-fun-def} body1' def body1 def cr-def spmf-rel-map td.\text{Rep[simplified]}
        invar td.\text{Abs-inverse intro}: \text{rel-spmf-refII})
\end{verbatim}
\end{verbatim}
define \( s' \) where \( s' \equiv \text{Abs } s \)

have [transfer-rule]: \( \text{cr } s s' \) by (simp add: \( s' \text{-def cr-def I td.Abs-inverse} \))

interpret \( \text{loop-spmf guard' body1'} \).

\[ \text{note UNIV-parametric-pred[transfer-rule]} \]

have \( \forall s. \text{guard' } s \Rightarrow p \leq \text{weight-spmf (while } s) \)

unfolding measure-measure-spmf-def[Symmetric] space-measure-spmf

by (transfer fixing: \( p \))(simp add: body1-def p[simplified space-measure-spmf]

cong: loop-spmf-while-cong)

moreover note \( p \text{-pos} \)

moreover have \( \forall s. \text{guard' } s \Rightarrow \text{lossless-spmf (body1' } s) \) by transfer

ultimately have \( \text{lossless-spmf (while } s) \)

by (rule termination-0-1)

hence \( \text{lossless-spmf (loop-spmf.while guard body1 s) by transfer } \}

from this[cancel-type-definition] I show \( ?\text{thesis} \) by (auto cong: loop-spmf-while-cong)

case 3.2 Variant rule

context \( \text{loop-spmf begin} \)

\( \text{lemma termination-variant:} \)

\( \text{fixes bound :: nat} \)

\( \text{assumes bound: } \forall s. \text{guard } s \Rightarrow f s \leq \text{bound} \)

\( \text{and step: } \forall s. \text{guard } s \Rightarrow p \leq \text{spmf (map-spmf (\( \lambda s'. f s' < f s \) (body } s)) \text{True} \)

\( \text{and p-pos: } 0 < p \)

\( \text{and lossless: } \forall s. \text{guard } s \Rightarrow \text{lossless-spmf (body } s) \)

\( \text{shows lossless-spmf (while } s) \)

\( \text{proof –} \)

\( \text{define } p' \text{ and } n \text{ where } p' \equiv \text{min } p 1 \text{ and } n \equiv \text{bound } + 1 \)

\( \text{have } p'-\text{pos: } 0 < p' \text{ and } p'-\text{le-1: } p' \leq 1 \)

\( \text{and step': guard } s \Rightarrow p' \leq \text{measure (measure-spmf (body } s)) \{ s'. f s' < f s \} \)

for \( s \)

\( \text{using } p\text{-pos step[of } s \text{] by (simp-all add: } p'\text{-def spmf\text{-map vimage-def})} \)

\( \text{have } p' \cdot n \leq \text{weight-spmf (while } s) \text{ if } f s < n \text{ for } s \text{ using that} \)

\( \text{proof(induction } n \text{ arbitrary: } s) \)

\( \text{case } 0 \text{ thus } ?\text{case by simp} \)

next

\( \text{case } (\text{Suc } n) \)

\( \text{show } ?\text{case} \)

\( \text{proof(cases guard } s) \)

\( \text{case False} \)

\( \text{hence weight-spmf (while } s) = 1 \text{ by (simp add: while.simps)} \)

\( \text{thus } ?\text{thesis using } p'-\text{le-1 } p\text{-pos} \)

\( \text{by simp(meson less-eq-real-def mult-le-one } p'\text{-pos power-le-one zero-le-power)} \)

next

\( \text{case True} \)

\( \text{let } ?M = \text{measure-spmf (body } s) \)

14
have \( p' \cdot \text{Suc } n \leq \{ \int s'. \text{ indicator } \{ s'. f s' < f s \} s' \cdot \partial?M \} \cdot p' \cdot n \)

using \( \text{step}[\text{OF True } p'-\text{pos } \text{by simp add: multi-right-mono}] \)
also have \( \ldots = \{ \int s'. \text{ indicator } \{ s'. f s' < f s \} s' \cdot p' \cdot n \cdot \partial?M \} \) by simp
also have \( \ldots \leq \{ \int s'. \text{ indicator } \{ s'. f s' < f s \} s' \cdot \text{ weight-spmf } (\text{while } s') \cdot \partial?M \} \)

using \( \text{Suc.prems } p'-\text{le-1 } p'-\text{pos } \text{by (intro integral-mono)}(\text{auto simp add: Suc.IH } \text{power-le-one weight-spmf-le-1}) \)
split: \( \text{split-indicator intro: measure-spmf.integrable-const-bound \{where } B=1\} \)
also have \( \ldots \leq \ldots + \{ \int s'. \text{ indicator } \{ s'. f s' \geq f s \} s' \cdot \text{ weight-spmf } (\text{while } s') \cdot \partial?M \} \)

by (simp add: integral-nonneg-AE weight-spmf-nonneg)
also have \( \ldots = \text{ weight-spmf } \) by (auto simp add: \text{symmetric})
(auto intro!: \text{Bochner-Integration.integral-add\{symmetric\}})
\( B=1 \) weight-spmf-le-1 split: \( \text{split-indicator} \)
also have \( \ldots = \text{ weight-spmf } \) by (simp add: weight-bind-spmf o-def)

finally show \( \text{thesis} \).

qed

moreover have \( 0 < p' \cdot n \) using \( p'-\text{pos } \text{by simp} \)
ultimately show \( \text{thesis } \text{using lossless} \)

proof (rule \text{termination-0-1-invar})

show \( f s < n \) if guard \( s \) guard \( s \longrightarrow f s < n \) for \( s \) using that by simp
show guard \( s \longrightarrow f s < n \) using bound[of \( s \)] by (auto simp add: \text{n-def})

show guard \( s' \longrightarrow f s' < n \) for \( s' \) using bound[of \( s' \)] by (clarsimp simp add: \text{n-def})

qed

qed

end

lemma \text{termination-variant-invar}:

fixes bound :: \text{nat} and \( I :: s \Rightarrow \text{bool} \)
assumes bound:: \( \text{bound } \cdot [s. \text{ guard } s; I s ] \Rightarrow f s \leq \text{bound} \)
and step: \( \forall s. [\text{ guard } s; I s ] \Rightarrow p \leq \text{spmfn} (\text{map-spmfn } (\lambda s'. f s' < f s ) (\text{body } s)) \) True
and \( p-\text{pos} 0 < p \)
and lossless: \( \forall s. [\text{ guard } s; I s ] \Rightarrow \text{lossless-spmf } (\text{body } s) \)
and invar: \( \forall s s'. [s' \in \text{set-spmf } (\text{body } s); I s; \text{ guard } s ] \Rightarrow I s' \)
and I: \( I s \)
shows \( \text{lossless-spmf } \) (loop-spmf.\text{while } guard body s)

including \text{lifting-syntax} 

proof –

\{ assume \exists (\text{Rep } :: 's' \Rightarrow 's) \text{Abs. type-definition Rep Abs } \{ s, I s \}
then obtain Rep :: 's' \Rightarrow 's' and Abs \text{where } td: \text{type-definition Rep Abs } \{ s, I s \} \text{ by blast }
then interpret td: \text{type-definition Rep Abs } \{ s, I s \} ,
define cr where cr \equiv \lambda x y. x = \text{Rep } y \}

15
have [transfer-rule]: bi-unique cr right-total cr using td cr-def by (rule typedef-bi-unique typedef-right-total)+

have [transfer-domain-rule]: Domainp cr = I using type-definition-Domainp[OF td cr-def] by simp

define guard’ where guard’ ≡ (Rep ----> id) guard

have [transfer-rule]: (cr ===>(=)) guard guard’ by (simp add: rel-fan-def cr-def guard’-def)

define body1 where body1 ≡ λs. if guard s then body s else return-pmf None

define body1’ where body1’ ≡ (Rep ----> map-spmf Abs) body1

have [transfer-rule]: (cr ===> rel-spmf cr) body1 body1’

by (auto simp add: rel-fan-def body1’-def body1-def cr-def spmf-rel-map td.Rep[simplified])

invar td.Abs-inverse intra!: rel-spmf-reflI

define s’ where s’ ≡ Abs s

have [transfer-rule]: cr s s’ by (simp add: s’-def cr-def I td.Abs-inverse)

define f’ where f’ ≡ (Rep ----> id) f

have [transfer-rule]: (cr ===> (=)) f’ f by (simp add: rel-fan-def cr-def f’-def)

have ∀s. guard’ s ===> f’ s ≤ bound by (transfer fixing: bound)(rule bound)

moreover have ∀s. guard’ s ===> p ≤ spmf (map-spmf (λs’. f’ s’ < f’ s) (body1’s)) True

by (transfer fixing: p)(simp add: step body1-def)

note this p-pos

moreover have ∀s. guard’ s ===> lossless-spmf (body1’s)

by (transfer(simp add: body1-def lossless)

ultimately have lossless-spmf (loop-spmf.while guard’ body1’ s) by (rule loop-spmf.termination-variant)

hence lossless-spmf (loop-spmf.while guard body1 s) by transfer

from this[cancel-type-definition] I show ?thesis by (auto cong: loop-spmf-while-cong)

qed

4 Distributions built from coin flips

4.1 The Bernoulli distribution

theory Bernoulli imports HOL-Probability.Probability begin

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

by (metis not-gr-zero zero-neq-numeral)

lemma ennreal-mult-numeral: ennreal x * numeral n = ennreal (x * numeral n)

by (simp add: ennreal-mult"

lemma one-plus-ennreal: 0 ≤ x ==\> 1 + ennreal x = ennreal (1 + x)

by simp
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.

```plaintext
partial-function (spmf) bernoulli :: real ⇒ bool spmf
where bernoulli p = do { b ← coin-spmf;
  if b then return-spmf (p ≥ 1 / 2)
  else if p < 1 / 2 then bernoulli (2 * p)
  else bernoulli (2 * p - 1) }

lemma pmf-bernoulli-None: pmf (bernoulli p) None = 0
  proof – have ereal (pmf (bernoulli p) None) ≤ (INF n∈UNIV. ereal (1 / 2 ^ n))
    proof(rule INF-greatest)
      show ereal (pmf (bernoulli p) None) ≤ ereal (1 / 2 ^ n) for n
        proof(induction n arbitrary: p)
          case (Suc n)
            show ?case using Suc.IH[of 2 * p] Suc.IH[of 2 * p - 1]
        qed(simp add: pmf-le-1)
    qed
    also have ... = ereal 0
      proof(rule LIMSEQ-unique)
        show (λn. ereal (1 / 2 ^ n)) −−−−→ ... by(rule LIMSEQ-INF)(simp add: field-simps decseq-SucI)
        show (λn. ereal (1 / 2 ^ n)) −−−−→ ereal 0 by(simp add: LIMSEQ-divide-realpow-zero)
      qed
    finally show ?thesis by simp
  qed

lemma lossless-bernoulli [simp]: lossless-spmf (bernoulli p)
  by(simp add: lossless-iff-pmf-None pmf-bernoulli-None)

lemma [simp]: assumes 0 ≤ p p ≤ 1
  shows bernoulli-True: spmf (bernoulli p) True = p (is ?True)
  and bernoulli-False: spmf (bernoulli p) False = 1 - p (is ?False)
  proof –
    { have ennreal (spmf (bernoulli p) b) ≤ ennreal (if b then p else 1 - p) for b
        using assms
      proof(induction arbitrary: p rule: bernoulli.fixp-induct[case-names adm bottom step])
        case adm show ?case by(rule cont-intro)+
      next
        case (step bernoulli’ p)
          by(auto simp add: UNIV-bool max-def divide-le-pos1 ennreal ennreal-mult-numeral
```

qed simp }

note this[of True] this[of False]
moreover have spmf (bernoulli p) True + spmf (bernoulli p) False = 1
  by(simp add: spmf-False-conv-True)
ultimately show ?True ?False using assms by(auto simp add: ennreal-le-iff2)
qed

lemma bernoulli-neg [simp]:
  assumes p ≤ 0
  shows bernoulli p = return-spmf False
proof –
  from assms have ord-spmf (=) (bernoulli p) (return-spmf False)
  proof(induction arbitrary: p rule: bernoulli_fixp-induct\[case-names adm bottom step])
    case (step bernoulli′ p)
    show ?case using step.prems step.IH[of 2 * p]
      by(auto simp add: ord-spmf-return-spmf2 set-bind-spmf bind-UNION field-simps)
    qed simp-all
  from ord-spmf-eq-leD[OF this, of True] have spmf (bernoulli p) True = 0 by simp
  moreover then have spmf (bernoulli p) False = 1 by(simp add: spmf-False-conv-True)
  ultimately show ?thesis by(auto intro: spmf-eqI split: split-indicator)
  qed

lemma bernoulli-pos [simp]:
  assumes 1 ≤ p
  shows bernoulli p = return-spmf True
proof –
  from assms have ord-spmf (=) (bernoulli p) (return-spmf True)
  proof(induction arbitrary: p rule: bernoulli_fixp-induct\[case-names adm bottom step])
    case (step bernoulli′ p)
    show ?case using step.prems step.IH[of 2 * p - 1]
      by(auto simp add: ord-spmf-return-spmf2 set-bind-spmf bind-UNION field-simps)
    qed simp-all
  from ord-spmf-eq-leD[OF this, of False] have spmf (bernoulli p) False = 0 by simp
  moreover then have spmf (bernoulli p) True = 1 by(simp add: spmf-False-conv-True)
  ultimately show ?thesis by(auto intro: spmf-eqI split: split-indicator)
  qed

context begin interpretation pmf-as-function .
lemma bernoulli-eq-bernoulli-pmf:
  bernoulli p = spmf-of-pmf (bernoulli-pmf p)
by(rule spmf-eqI; simp)(transfer; auto simp add: max-def min-def)
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.

case notes [[function-internals]] begin

partial-function (spmf) geometric-spmf :: real ⇒ nat spmf where
geometric-spmf p = do 
  b ← bernoulli p;
  if b then return-spmf 0 else map-spmf (+) (geometric-spmf p)
end

lemma geometric-spmf-fixp-induct [case-names adm bottom step]:
assumes spmf.admissible P and P (λ geometric-spmf. return-pmf None)
and P geometric-spmf' ⇒ P (λp. bernoulli p ϱ= (λb. if b then return-spmf 0 else map-spmf (+) 1 (geometric-spmf' p)))
shows P geometric-spmf
using assms by (rule geometric-spmf.fixp-induct)

lemma spmf-geometric-nonpos: p ≤ 0 ⇒ geometric-spmf p = return-pmf None
by (induction rule: geometric-spmf-fixp-induct) simp-all

lemma spmf-geometric-ge-1: 1 ≤ p ⇒ geometric-spmf p = return-spmf 0
by (simp add: geometric-spmf.simps)

case fixes p :: 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)
by (fact body-def)

lemma geometric-spmf-conv-while:
shows geometric-spmf p = map-spmf snd (while True, 0)
proof −
have map-spmf ((+) x) (geometric-spmf p) = map-spmf snd (while (True, x))
(is ?lhs = ?rhs) for x
proof (rule spmf.leq-antisym)
  show ord-spmf (=) ?lhs ?rhs
    proof (induction arbitrary: x rule: geometric-spmf-fixp-induct)
      case adm show ?case by simp
      case bottom show ?case by simp
      case (step geometric)
      show ?case using step.IH[of Suc x]
        apply (rewite while.simps)
        apply (clarsimp simp add: map-spmf-bind-spmf bind-map-spmf intro: !: ord-spmf-bind-reflI)
        apply (rewite while.simps)
        apply (clarsimp simp add: spmf.map-comp o_def)
      done
    qed
  have ord-spmf (=) ?rhs ?lhs
    and ord-spmf (=) (map-spmf snd (while (False, x))) (return-spmf x)
    proof (induction arbitrary: x and x rule: while-fixp-induct)
      case adm show ?case by simp
      case bottom case 1 show ?case by simp
      case bottom case 2 show ?case by simp
      next
        case (step while)
        case 1 show ?case using step.IH(1)[of Suc x] step.IH(2)[of x]
        case 2 show ?case by simp
        qed
      then show ord-spmf (=) ?rhs ?lhs by
      qed
      from this[of 0] show ?thesis by (simp cong: map-spmf-cong)
    qed

lemma lossless-geometric [simp]: lossless-spmf (geometric-spmf p) \iff p > 0
proof (cases 0 < p \land p < 1)
  case True
  let ?body = \lambda (b, x :: nat). map-spmf (\lambda b'. (\neg b', x + (if b' then 0 else 1)))
  have lossless-spmf (while (True, 0))
    proof (rule termination-0-1-immediate)
      have \{ x, x \} = \{ True \} by auto
      then show p \leq spmf (map-spmf fst (?body s)) False for s :: bool \times nat using
        True
          by (cases s)(clarsimp simp add: spmf.map-comp o_def spmf-map vimage-def spmf-conv-measure-spmf[ symmetric])
        show 0 < p using True by simp
    qed (clarsimp)
  with True show ?thesis by (simp add: geometric-spmf-conv-while)
    qed (auto simp add: spmf-geometric-nonpos spmf-geometric-ge-1)
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 (rule spmf-ub-tight)
fix n
have ennreal (?lhs n) ≤ ennreal (?rhs n) using p
proof (induction arbitrary: n rule: geometric-spmf-fixp-induct)
case adm show ?case by (rule cont-intro)+
case bottom show ?case by simp
case (step geometric-spmf')
then show ?case by (cases n)
(simp-all add: ennreal-spmf-bind nn-integral-measure-spmf UNIV-bool
nn-integral-count-space-finite ennreal-mult spmf-map vimage-def mult
assoc spmf-conv-measure-spmf [symmetric]
 mult-mono split: split-indicator)
qed
then show ?lhs n ≤ ?rhs n using p by (simp)
next
have (∑ i. ennreal (p * (1 - p) ^ i)) = ennreal (p * (1 / (1 - (1 - p))))
using p
by (intro suminf-ennreal-eq sums-mult geometric-sums) auto
then show (∑ x. ennreal ((1 - p) ^ x * p)) = weight-spmf (geometric-spmf p)
using lossless-geometric[of p] unfolding lossless-spmf-def
by (simp add: nn-integral-count-space-nat field-simps)
qed

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
have ?lhs = bind-pmf (map-pmf (λ(c, b). (2 * c + (if b then 1 else 0))) (pair-pmf
(pmffof-set {..<v}) (pmf-of-set UNIV))) f

end
\((is\ - = bind-pmf \ (map-pmf \ ?f\ -)\ -)\)
by\((simp \ add: \ pair-pmf-def \ bind-map-pmf \ bind-assoc-pmf \ bind-return-pmf)\)
also have \(map-pmf \ ?f\ \ (pair-pmf \ (pmf-of-set \ \{..<v\}) \ (pmf-of-set \ UNIV)) = pmf-of-set \ \{..<2 * v\}\)
\((is \ \$l = \ ?r \ is \ map-pmf \ ?f \ ?p = -)\)
proof\((rule \ pmf-eqI)\)
fix \(i :: \ nat\)
have \([simp]: \ inj \ ?f \ by(auto \ simp \ add: \ inj-on-def) \ arith+\)
define \(i' \ where \ i' \equiv \ i \ div \ 2\)
define \(b \ where \ b \equiv \ odd \ i\)
have \(i = \ ?f \ (i', \ b) \ by(simp \ add: \ i'-def \ b-def)\)
show \(pmf \ ?l \ i = pmf \ ?r \ i\)
by\((subst \ i; \ subst \ pmf-map-inj')(simp-all \ add: \ pmf-pair \ i'-def \ assms \ lessThan-empty-iff \ split: \ split-indicator)\)
qed
finally show \(?thesis\).
qed

lemma sample-bits-fusion2:
fixes \(v :: \ nat\)
assumes \(0 < v\)
shows \(bind-pmf \ (pmf-of-set \ UNIV) \ (\lambda \ b. \ bind-pmf \ (pmf-of-set \ \{..<v\}) \ (\lambda c. \ f \ (c + v * (if \ b \ then \ 1 \ else \ 0)))) = bind-pmf \ (pmf-of-set \ \{..<2 * v\}) \ f\)
\((is \ ?lhs = ?rhs)\)
proof
have \(?lhs = bind-pmf \ (map-pmf \ (\lambda (c, \ b). \ (c + v * (if \ b \ then \ 1 \ else \ 0)))) \ (pair-pmf \ (pmf-of-set \ \{..<v\}) \ (pmf-of-set \ UNIV))\) \(f\)
\((is \ - = bind-pmf \ (map-pmf \ ?f\ -)\ -)\)
unfolding \(pair-pmf-def\) by\((subst \ bind-commute-pmf)(simp \ add: \ bind-map-pmf \ bind-assoc-pmf \ bind-return-pmf)\)
also have \(map-pmf \ ?f \ (pair-pmf \ (pmf-of-set \ \{..<v\}) \ (pmf-of-set \ UNIV)) = pmf-of-set \ \{..<2 * v\}\)
\((is \ ?l = \ ?r \ is \ map-pmf \ ?f \ ?p = -)\)
proof\((rule \ pmf-eqI)\)
fix \(i :: \ nat\)
have \([simp]: \ inj \ ?f \ by(auto \ simp \ add: \ inj-on-def)\)
define \(i' \ where \ i' \equiv \ if \ i \ ge \ v \ then \ i - v \ else \ i\)
define \(b \ where \ b \equiv \ i \ ge \ v\)
have \(i = \ ?f \ (i', \ b) \ by(simp \ add: \ i'-def \ b-def)\)
show \(pmf \ ?l \ i = pmf \ ?r \ i\)
proof\((cases \ i < 2 * v)\)
case \(True\)
thus \(?thesis\)
by\((subst \ i; \ subst \ pmf-map-inj')(auto \ simp \ add: \ pmf-pair \ i'-def \ assms \ lessThan-empty-iff \ split: \ split-indicator)\)
next
case \(False\)
hence $i \notin \text{set-pmf}$.\footnote{Using \texttt{assms} by\(\texttt{(auto simp add: lessThan-empty-iff split: if-split-asm)}\) thus \texttt{?thesis} by\(\texttt{(simp add: set-pmf-iff del: set-map-pmf)}\)

\texttt{qed}

\texttt{qed}

finally show \texttt{?thesis}.

\texttt{qed}

\texttt{context fixes} $n :: \text{nat}$ \texttt{notes} \texttt{[[function-internals]] \begin{quote}

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.

\texttt{partial-function} (\texttt{spmf}) \texttt{fast-dice-roll :: nat} $\Rightarrow$ \texttt{nat} $\Rightarrow$ \texttt{nat} \texttt{spmf}

\texttt{where}

\texttt{fast-dice-roll \ v \ c =}

\texttt{(if v \geq \ n \ then \ if \ c \ < \ n \ then \ return-spmf \ c \ else \ fast-dice-roll \ (v - n) \ (c - n)}

\texttt{else \ do}

\texttt{b <- coin-spmf;

fast-dice-roll \ (2 * v) \ (2 * c + (if b \ then \ 1 \ else \ 0)) \ )

\texttt{})

\texttt{lemma fast-dice-roll-fixp-induct [case-names adm bottom step]:}

\texttt{assumes spmf. admitsible \(\lambda\)\texttt{fast-dice-roll}. \texttt{P} \ (\texttt{curry fast-dice-roll})}

\texttt{and } \texttt{P} \ (\lambda \ v \ c. \ \texttt{return-spmf} \ \texttt{None})

\texttt{and } \texttt{fdr.} \texttt{P} \ \texttt{fdr} \Rightarrow \texttt{P} \ (\lambda \ v \ c. \ \texttt{if} \ v \geq \ n \ \texttt{then} \ \texttt{if} \ c \ < \ n \ \texttt{then} \ \texttt{return-spmf} \ c \ \texttt{else} \ \texttt{fdr} \ (v - n) \ (c - n)}

\texttt{else} \texttt{bind-spmf} \ \texttt{coin-spmf} \ (\lambda \ \texttt{fdr}. \texttt{(2 * v) \ (2 * c + (if b \ then \ 1 \ else \ 0)))}}

\texttt{shows \texttt{P} \ \texttt{fast-dice-roll}}

\texttt{using \texttt{assms} by\(\texttt{(rule fast-dice-roll.fixp-induct)}\)

\texttt{definition fast-uniform :: \texttt{nat} \texttt{spmf}}

\texttt{where} \texttt{fast-uniform = fast-dice-roll 1 0}

\texttt{lemma spmf-fast-dice-roll-ub:}

\texttt{assumes 0 < v}

\texttt{shows spmf \ (bind-pmf \ (pmf-of-set \ \{..<v\}) \ \texttt{fast-dice-roll} \ v)) \ x \leq (if x < n \ then \ 1 / n \ else \ 0)}

\texttt{(is ?lhs \leq ?rhs)}

\texttt{proof}

\texttt{have ennreal ?lhs \leq ennreal ?rhs \ using \texttt{assms}}

\texttt{proof(induction arbitrary: v x rule: fast-dice-roll-fixp-induct)}

\texttt{case adm thus ?case}

\texttt{by\(\texttt{(rule cont-intro ccpo-class.admissible-leI)}\)+ \texttt{simp-all}

\texttt{23}
case bottom thus \( \text{case by simp} \)

\begin{enumerate}
\item \text{case (step fdr)}
\item \text{show \( \text{?case (is \( \forall b s \leq \forall e s) \)}}\)
\item \text{proof\( (cases n \leq v) \)}}\)
\item \text{case le: True}
\item \text{then have \( \forall b s = \text{spmf (bind-pmf (pmf-of-set \{..<v\}) (e c < n then return-spmf c else fdr (v n) (c n))) x} \)}}\)
\item \text{by simp}
\item \text{also have \( \ldots = (f * c \cdot \text{indicator (if } x < n \text{ then } \{x\} \text{ else } \{\}) c \cdot \text{measure-pmf (pmf-of-set \{..<v\}) + (f * c \cdot \text{indicator } \{n \..< v\} c \prime \cdot \text{spmf (fdr (v n) (c n)) x} \prime \text{ measure-pmf (pmf-of-set \{..<v\})} \)}}\)
\item \text{(is ?then = ?found + ?continue) \textbf{using step.prems}}
\item \text{by\( (\text{subst nn-integral-add symmetric}) (\text{auto simp add: ennreal-pmf-bind AE-measure-pmf-iff lessThan-empty-iff split: split-indicator intro!: nn-integral-cong-AE}) \)}}\)
\item \text{also have \( ?\text{found = (if } x < n \text{ then 1 else 0) / v using step.prems le} \)}}\)
\item \text{by\( (\text{auto simp add: measure-pmf_evaluate-measure measure-pmf-of-set lessThan-empty-iff Bounded-Int-singleton}) \)}}\)
\item \text{also have \( \ldots = (f * c \cdot \text{indicator } \{n \..< v\} c \prime \cdot \text{measure-pmf (pmf-of-set \{..<v\})} \text{ pmf-of-set \{..<v\})} (\lambda c \cdot \text{fdr (v n) (c n)}) x) \)}}\)
\item \text{using le \textbf{step.prems}}
\item \text{by\( (\text{subst ennreal-pmf-bind}) (\text{auto simp add: ennreal-mult symmetric nn-integral-measure-pmf nn-integral-0-iff-AE AE-count-space nn-integral-mult symmetric split: split-indicator}) \)}}\)
\item \text{also \{}
\item \text{assume \: \: \: \text{* n < v}}
\item \text{then have \text{pmf-of-set \{n..<v\} = map-pmf ((\*) n) (pmf-of-set \{..<v n\})} \)}}\)
\item \text{by\( (\text{subst map-pmf-of-set-inj}) (\text{auto 4 simp add: inj-on-def lessThan-empty-iff intro!: arg-cong where f=pmf-of-set intro: rev-image-eq where x=\-- n) diff-less-mono}) \)}}\)
\item \text{also have \text{bind-pmf \ldots = (\lambda c \cdot \text{fdr (v n) (c n)}) = bind-pmf (pmf-of-set \{..<v n\}) (fdr (v n))} \)}}\)
\item \text{by\( (\text{simp add: bind-map-pmf}) \)}}\)
\item \text{also have \text{ennreal (spmf \ldots x) < (if } x < n \text{ then 1 / n else 0) by\( (\text{rule step.IH}) (\text{simp add: \*}) \)}}\)
\item \text{also note calculation \}}\)
\item \text{then have \text{\leq ennreal ((v n) / v) * (if } x < n \text{ then 1 / n else 0)} \textbf{using le} \)}}\)
\item \text{by\( (\text{cases n=n}) (\text{auto split del: if-split intro: divide-right-mono mult-left-mono}) \)}}\)
\item \text{also have \( \ldots = (v n) / v * (if } x < n \text{ then 1 / n else 0) by\( (\text{simp add: ennreal-mult symmetric}) \)}}\)
\item \text{\textbf{finally show \text{?thesis using le by (auto simp add: add-mono field-simps of-nat-diff ennreal-plus symmetric simp del: ennreal-plus)}}} \}
\item \text{next}
\item \text{case \text{False}}
\item \text{then have \text{\forall b s = spmf (bind-pmf (pmf-of-set \{..<v\}) (\lambda c. bind-pmf (pmf-of-set UNIV) (\lambda b. fdr (2 v) (2 c + (if b then 1 else 0)))) x)}}\)
\end{enumerate}

24
by (simp add: bind-spmf-spmf-of-set)
also have \( \ldots = \text{spmf (bind-pmf (pmf-of-set \{..<2 \ast v\}) (fdr (2 \ast v)))} \) x
using step.prems
  by (simp add: sample-bits-fusion[symmetric])
also have \( \ldots \leq \text{?rhs} \) using step.prems by (intro step.IH) simp
finally show ?thesis .
qed
qed
thus ?thesis by simp
qed

lemma spmf-fast-uniform-ub:
  \( \text{spmf fast-uniform x} \leq (\text{if x < n then 1 / n else 0}) \)
proof
  have \( \{..\text{Suc 0}\} = \{0\} \) by auto
  then show ?thesis using spmf-fast-dice-roll-ub[of 1 x]
  by (simp add: fast-uniform-def pmf-of-set-singleton bind-return-pmf split: if-split_asm)
qed

lemma fast-dice-roll-0 [simp]: 
  \( \text{fast-dice-roll 0 c} = \text{return-pmf None} \)

To prove termination, we fold all the iterations that only double into one big step

definition fdr-step :: nat ⇒ nat ⇒ (nat × nat) spmf
where
  fdr-step v c =
  (if v = 0 then return-pmf None
   else let x = \(2 \ast (\text{nat \ceil{\log 2 (max 1 n) - \log 2 v}})\) in
     map-spmf (λbs. (x \ast v, x \ast c + bs)) (spmf-of-set \{..<x\}))

lemma fdr-step-unfold:
  fdr-step v c =
  (if v = 0 then return-pmf None
   else if n ≤ v then return-spmf (v, c)
   else do {b ← \text{coin-spmf};
              fdr-step (2 \ast v) (2 \ast c + (if b then 1 else 0))}
           (is ?lhs = ?rhs is - = (if - then - else ?else))
proof(cases v = 0)
  case v: False
  define x where x ≡ λv :: nat. 2 \ast (\text{nat \ceil{\log 2 (max 1 n) - \log 2 v}}) :: nat
  have x-pos: x v > 0 by (simp add: x-def)
  show ?thesis
  proof(cases n ≤ v)
    case le: True
    hence x v = 1 using v by (simp add: x-def log-le)
    moreover have \{..<1\} = \{0 :: nat\} by auto
  qed
ultimately show \textbf{thesis} using le v by(simp add: fdr-step-def spmf-of-set-singleton)

next
\begin{itemize}
  \item \textbf{case} \texttt{less}: False
  \item \textbf{hence} even: even \((x v)\) using \(v\) by(simp add: \texttt{x-def})
  \item with \texttt{x-pos} have \(x\geq 1\) by(cases \(x v = 1\)) auto
  \item have \(*: x (2*v) = x v \text{ div } 2\) using \(v\) less unfolding \(\texttt{x-def}\)
    \begin{itemize}
      \item apply(simp add: log-mult diff-add-ev-diff-diff-swap)
      \item apply(rewrite in - = 2 ^ \Pi \text{ div } \leq add-diff-inverse2[symmetric, where \(b = 1\)])
      \item apply (simp add: Suc-leI)
      \item apply(simp del: Suc-pred)
    \end{itemize}
  \item done
\end{itemize}

have \(?lhs = \text{map-spmf} (\lambda b. (x v * v, x v * c + bs)) (\text{spmf-of-set} \{..<x v\})

using \(*by(sim\ add: \text{fdr-step-def x-def Let-def})\)

also from even have \(= \text{bind-pmf} (\text{pmf-of-set} \{..<2 \ast (x v \text{ div } 2)\}) (\lambda b. \text{return-spmf} (x v * v, x v * c + bs))\)

by(simp add: map-spmf-conv-bind-spmf bind-spmf-spmf-of-set \texttt{x-pos lessThan-empty-iff})

also have \(= \text{bind-spmf} \text{coin-spmf} (\lambda b. \text{bind-spmf} (\text{spmf-of-set} \{..<x v \text{ div } 2\}))\)

\(\lambda c'. \text{return-spmf} (x v * v, x v * c + c' + (x v \text{ div } 2) * (if b then 1 else 0))\))

using \(x\geq 1\)

by(simp add: sample-bits-fusion2[symmetric] bind-spmf-spmf-of-set lessThan-empty-iff add_assoc)

also have \(= \text{bind-spmf} \text{coin-spmf} (\lambda b. \text{map-spmf} (\lambda b. (x (2 * v) * (2 * v), x (2 * v) * (2 * c + (if b then 1 else 0)) + bs)) (\text{spmf-of-set} \{..< (x (2 * v))\})\)

using \(*by(sim\ add: \text{map-spmf-conv-bind-spmf algebra-simps})\)

also have \(= \?\text{lhs} using \;v\; by(simp add: \text{fdr-step-def Let-def x-def})\)

finally show \(?\text{thesis}\)

qed

\textbf{qed}(simp add: \text{fdr-step-def})

\textbf{lemma} \text{fdr-step-induct} [case-names \text{fdr-step}]:
\((\forall v c. (\lambda b. [v \neq 0; v < n] \implies P (2 * v) (2 * c + (if b then 1 else 0))) \implies P v c)\)
\(\implies P v c\)

\textbf{apply} induction-schema

\textbf{apply} \text{pat-completeness}

\textbf{apply} (relation Wellfounded.measure (\lambda (v, c). n - v))

\textbf{apply} \text{simp-all}

\textbf{done}

\textbf{partial-function} (spmf) \text{fdr-alt} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \text{ spmf}

\textbf{where}
\text{fdr-alt} v c = do 
(\(v', c'\) \leftarrow \text{fdr-step} v c;
if \(c' < n\) then \text{return-spmf} c' else \text{fdr-alt} (v' - n) (c' - n) \}

\textbf{lemma} \text{fast-dice-roll-alt}: \text{fdr-alt} = \text{fast-dice-roll}

\textbf{proof}(intro ext)
show \( fdr-alt \ v \ c = \text{fast-dice-roll} \ v \ c \) for \( v \ c \)

**proof** (rule \( \text{spmf.\ leq-antisym} \))

show \( \text{ord-spmf} \ (\ldotp) \ (fdr-alt \ v \ c) \ (\text{fast-dice-roll} \ v \ c) \)

**proof** (induction arbitrary: \( v \ c \) rule: \( fdr-alt,\fixp-induct\{\text{case-names adm bottom step}\})

case adm show \( ?\)case by simp

case bottom show \( ?\)case by simp

case (step \( fdra \)) show \( ?\)case

**proof** (induction \( v \ c \) rule: \( \text{fdr-step-induct} \))

case inner: (\( \text{fdr-step} \ v \ c \)) show \( ?\)case

apply (rewrite \( \text{fdr-step-unfold} \))
apply (rewrite \( \text{fast-dice-roll} \).\ simps)
apply (auto intro: \( \text{ord-spmf\\-bind-reflI} \) simp add: \( \text{Let-def inner.IH step.IH} \))
done

qed

qed

have \( \text{ord-spmf} \ (\ldotp) \ (\text{fast-dice-roll} \ v \ c) \ (fdr-alt \ v \ c) \) and \( \text{fast-dice-roll} \ 0 \ c = \text{return-pmf} \ \text{None} \)

**proof** (induction arbitrary: \( v \ c \) rule: \( \text{fast-dice-roll\-fixp-induct} \))

case adm thus \( ?\)case by simp

case bottom case 1 thus \( ?\)case by simp

case bottom case 2 thus \( ?\)case by simp

case (step \( fdr \)) case 1 show \( ?\)case

apply (rewrite \( \text{fdr-alt\\-simps} \))
apply (rewrite \( \text{fdr-step-unfold} \))
apply (clarsimp simp add: \( \text{Let-def} \))
apply (auto intro: \( \text{ord-spmf\\-bind-reflI} \) simp add: \( \text{fdr-alt\\-simps[symmetric]} \) step.IH rel-pmf-return-pmf2 set-pmf-bind-spmf o-def set-pmf-spmf\-of-set split: \( \text{if-split-asm} \))
done

case step case 2 from step.IH show \( ?\)case by (simp add: \( \text{Let-def bind-eq-return-pmf\-None} \))

qed
then show \( \text{ord-spmf} \ (\ldotp) \ (\text{fast-dice-roll} \ v \ c) \ (fdr-alt \ v \ c) \) by −

qed

qed

**lemma** \( \text{lossless-fdr-step [simp]} \): \( \text{lossless-spmf} \ (\text{fdr-step} \ v \ c) \leftrightarrow v > 0 \)

by (simp add: \( \text{fdr-step\-def Let-def lessThan-empty-iff} \))

**lemma** \( \text{fast-dice-roll\-alt\-conv-while} \):

\[
\text{fdr-alt} \ v \ c = \text{map-spmf} \ \text{snd} \ (\text{bind-spmf} \ (\text{fdr-step} \ v \ c) \ (\text{loop-spmf\ \while} \ (\lambda (v, c). \ n \leq c) \ (\lambda (v, c). \text{fdr-step} (v - n) (c - n))))
\]

**proof** (induction arbitrary: \( v \ c \) rule: \( \text{parallel-fixp-induct-2-1[OF partial-function-definitions-spmf partial-function-definitions-spmf fdr-alt,\ mono loop-spmf \while mono fdr-alt,\ mono loop-spmf \while-def, case-names adm bottom step]} \))

case adm show \( ?\)case by (simp)

case bottom show \( ?\)case by simp
case (step fdr while)
  by(auto simp add: map-spmf-bind-spmf o_def intro!: bind-spmf-cong[OF refl])

qed

lemma lossless-fast-dice-roll:
  assumes c < v v ≤ n
  shows lossless-spmf (fast-dice-roll v c)
proof(cases v < n)
  case True
  have invar: ?I (v', c') if step: (v', c') ∈ set-spmf (fdr-step (v - n) (c - n))
    using v' c' ≤ v v c
  finally show c... by(auto)

  show c... by(auto)

proof(clarsimp)
  define x where x = nat [log 2 (max 1 n) - log 2 (v - n)]
  have **: -1 < log 2 (real n / real (v - n)) by(rule less_le_trans[where y=0])(use I c in(auto))

  from I c step obtain bs where v': v' = 2 ^ x * (v - n)
    and c'; c' = 2 ^ x * (c - n) + bs
    and bs: bs < 2 ^ x
    unfolding fdr-step-def x-def[symmetric] by(auto simp add: Let-def)
  have 2 ^ x * (c - n) + bs < 2 ^ x * (c - n + 1) unfolding distrib_left using
    bs by(intro add-strict-left-mono) simp
  also have ... ≤ 2 ^ x * (v - n) using I c by(intro mult-strict-left-mono) auto
  finally show c' < v' using c' v' by simp

  have v' = 2 powr x * (v - n) by(simp add: powr-realpow v')
  also have ... < 2 powr (log 2 (max 1 n) - log 2 (v - n) + 1) * (v - n)
    using ** I c by(intro mult-strict-right-mono)(auto simp add: x-def log-divide)
  also have ... ≤ 2 * n unfolding powr-add using I c
    by(simp add: log-divide[symmetric] max-def)
  finally show v' < 2 * n using c' by(simp del: of-nat-add)

  have log 2 (n / (v - n)) ≤ x using I c ** by(auto simp add: x-def log-divide max-def)
  hence 2 powr log 2 (n / (v - n)) ≤ 2 powr x by(rule powr-mono) simp
  also have 2 powr log 2 (n / (v - n)) = n / (v - n) using I c by(simp)
  finally have n ≤ real (2 ^ x * (v - n)) using I c by(simp add: field-simps powr-realpow)
    then show n ≤ v' by(simp add: v' del: of-nat-mult)
    qed

  have loop: lossless-spmf (loop-spmf.while (λ(v, c). n ≤ c) (λ(v, c). fdr-step (v - n) (c - n)) (v, c))
    if c < 2 * n and n ≤ v and c < v and v < 2 * n
    for v c
proof (rule termination-variant-invar; clarify)
  fix v c
  assume I: ?I (v, c) and c: n ≤ c
  show ?I (v, c) ≤ n using I c by auto

  define x where x = nat [log 2 (max 1 n) − log 2 (v − n)]
  define p :: real where p = 1 / (2 * n)

  from I c have n: 0 < n and v: n < v by auto
  from I c n have x-pos: x > 0 by (auto simp add: x-def max-def)

  have **: −1 < log 2 (real n / real (v − n)) by (rule less-le-trans[where y=0])(use I c in (auto))
  then have x ≤ log 2 (real n) + 1 using v n
    by (auto simp add: x-def log-divide[symmetric] max-def field-simps intro:
          order-trans[OF of-int-ceiling-le-add-one])
  hence 2 powr x ≤ 2 powr ... by (rule powr-mono) simp
  hence p ≤ 1 / 2 ^ x unfolding powr-add using n
    by (subt (asm) powr-realpow, simp)(subst (asm) powr-log-cancel; simp-all
       add: p-def field-simps)
  also
    let ?X = {c', n ≤ 2 ^ x * (c − n) + c'} → n + (2 ^ x * (c − n) + c') − 2
    ^ x * (v − n) < n + c − v
  have n + c * 2 ^ x − v * 2 ^ x < c + n − v using I c
  proof (cases n + c * 2 ^ x ≥ v * 2 ^ x)
    case True
    have (int c − v) * 2 ^ x < (int c − v) * 1
      using x-pos I c by (intro mult-strict-left-mono-neg) simp-all
    then have int n + c * 2 ^ x − v * 2 ^ x < c + int n − v by (simp add:
                                                          algebra-simps)
    also have ... = int (c + n − v) using I c by auto
    also have int n + c * 2 ^ x − v * 2 ^ x = int (n + c * 2 ^ x − v * 2 ^ x)
      using True that by (simp add: of-nat-diff)
  finally show ?thesis by simp
  qed auto
  then have {... << 2 ^ x} ∩ {?X ≠ {}} using that n v
    by (auto simp add: disjoint-eq-subset-Compl Collect-neg-eq[symmetric] lessThan-subset-Collect
           algebra-simps intro: exI[where x=0])
  then have 0 < card {... << 2 ^ x} ∩ {?X} by (simp add: card-gt-0-iff)
  hence 1 / / 2 ^ x ≤ ... / 2 ^ x by (simp add: field-simps)
  finally show p ≤ spmf (map-spmf (λxs′. ¨f s′ < ¨f (v, c)) (fdr-step (v − n))
          (c − n))) True
    using I c unfolding fdr-step-def x-def[symmetric]
    by (clarsimp simp add: Let-def spmf.map-comp o-def spmf-map measure-spmf-of-set
          vimage-def p-def)

  show lossless-spmf (fdr-step (v − n) (c − n)) using I c by simp
  show ?I (v', c') if step: (v', c') ∈ set-spmf (fdr-step (v − n) (c − n)) for v'
     c'
using that by (rule invar) (use I c in auto)
next
  show \((0 :: real) < 1 / (2 * n)\) using by (simp)
  show \(?I (v, c)\) using by simp
qed
show \(?thesis using assms True\)
  by (auto simp add: fast-dice-roll-alt[symmetric] fast-dice-roll-alt-conv-while intro: loop dest: invar[of - - n + v n + c, simplified])
next
  case False
  with assms have \(v = n\) by (simp)
thus \(?thesis using assms by(subst fast-dice-roll.simps) simp\)
qed

lemma fast-dice-roll-n0:
  assumes \(n = 0\)
  shows fast-dice-roll v c = return-pmf None
  by (induction arbitrary: v c rule: fast-dice-roll-fixp-induct)
  (simp-all add: assms)

lemma lossless-fast-uniform:
  lossless-spmf fast-uniform \(\leftrightarrow n > 0\)
proof (cases \(n = 0\))
  case True
  then show \(?thesis using fast-dice-roll-n0 unfolding fast-uniform-def by(simp)\)
next
  case False
  then show \(?thesis by(simp add: fast-uniform-def lossless-fast-dice-roll)\)
qed

lemma spmf-fast-uniform:
  spmf fast-uniform x = (if x < n then 1 / n else 0)
proof (cases \(n > 0\))
  case True
  show \(?thesis using spmf-fast-uniform-ub\)
    by (rule spmf-ub-tight)
    (auto simp add: nn-integral-count-space-indicator simp del: nn-integral-const intro: nn-integral-cong)
    also have \(= 1\) using \(n\) by (simp add: field-simps ennreal-of-nat-eq-real-of-nat ennreal-mult[symmetric])
    also have \(= weight-spmf fast-uniform using lossless-fast-uniform n unfolding lossless-spmf-def by simp\)
    finally show \((\sum^{+} x. ennreal (if x < n then 1 / n else 0)) = \ldots \). 
    qed
next
  case False
  with fast-dice-roll-n0[of 1 0] show \(?thesis unfolding fast-uniform-def by(simp)\)
qed
end
lemma fast-uniform-conv-uniform: fast-uniform n = spmf-of-set {..<n}
by (rule spmf-eqI) (simp add: spmf-fast-uniform spmf-of-set)
end

theory Resampling imports While-SPMF begin

lemma ord-spmf-lossless:
assumes ord-spmf (=) p q lossless-spmf p
shows p = q
unfolding pmf.rel-eq[symmetric] using assms(1)
by (rule pmf.rel-mono-strong) (use assms(2) in {auto elim: ord-option.cases simp add: lossless-iff-set-pmf-None})

context notes [[function-internals]] begin

partial-function (spmf) resample :: 'a set ⇒ 'a set ⇒ 'a spmf where
resample A B = bind-spmf (spmf-of-set A) (λ x. if x ∈ B then return-spmf x else resample A B)
end

lemmas resample-fixp-induct[case-names adm bottom step] = resample.fixp-induct

context fixes A :: 'a set and B :: 'a set begin
interpretation loop-spmf λ x. x /∈ B λ -. spmf-of-set A .

lemma resample-conv-while: resample A B = bind-spmf (spmf-of-set A) while
proof (induction rule: parallel-fixp-induct-2-1[OF partial-function-definations-spmf partial-function-definations-spmf resample.mono while.mono resample-def while-def, case-names adm bottom step])
case adm show ?case by simp
case bottom show ?case by simp
case (step resample' while') then show ?case by (simp add: z3-rule(33) cong del: if-cong)
qed

context
assumes A: finite A
and B: B ⊆ A B ≠ {}
begin
private lemma A-nonempty: A ≠ {}
  using B by blast

private lemma B-finite: finite B
  using A B by (blast intro: finite-subset)

lemma lossless-resample: lossless-spmf (resample A B)
proof
  from B have [simp]: A ∩ B ≠ {} by auto
  have lossless-spmf (while x) for x
    by (rule termination-0-1-immediate) [where p = card (A ∩ B) / card A]
    simp-all add: spmf-map vimage-def measure-spmf-of-set field-simps A-nonempty
    A not-le card-gt-0-iff B
  then show ?thesis by (clarsimp simp add: resample-conv-while A A-nonempty)
qed

lemma resample-le-sample: ord-spmf (=) (resample A B) (spmf-of-set B)
proof (induction rule: resample-fixp-induct)
case adm show ?case by simp
  case bottom show ?case by simp
  case (step resample')
    note [simp] = B-finite A
    show ?case
      proof (rule ord-pmf-increaseI)
        fix x
        let ?f = λx. if x ∈ B then return-spmf x else resample' A B
        have spmf (bind-spmf (spmf-of-set A) ?f) x =
          (∑n∈B ∪ (A − B). if n ∈ B then (if n = x then 1 else 0) / card A else spmf (resample' A B) x / card A)
          using B
        also have ... = (∑n∈B. if n = x then 1 else 0) / card A + (∑n∈A − B. spmf (resample' A B) x / card A)
          by (subst sum.union-disjoint) (auto)
        also have ... ≤ (if x ∈ B then 1 / card A else 0) + card (A − B) / card A * spmf (resample' A B) x
          by (simp cong: sum.cong add: if-distrib) [where f = λx. x / -] cong: if-cong
        also have ... = spmf (spmf-of-set B) x using B
          by (simp add: spmf-of-set field-simps A-nonempty card-Diff-subset card-mono of-nat-diff)
        finally show spmf (bind-spmf (spmf-of-set A) ?f) x ≤ ... .
      qed simp
    qed
qeq
lemma resample-eq-sample: resample A B = spmf-of-set B
  using resample-le-sample lossless-resample by (rule ord-spmf-lossless)
end
end
end

References

