Probabilistic while loop

Andreas Lochbihler

April 11, 2026

Abstract

This AFP entry defines a probabilistic while operator based on subprobability 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

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* ∈ *f x* ⟹ *Some y* ∈ *map-option-set f* (*Some x*)
by *simp*

lemma *map-option-set-singleton* [*simp*]: *map-option-set* (λ*x*. {*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* ∘ *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 $\lambda x.\ x$ - $\lambda x.\ x$,
where $P{=}\lambda f\ g.\ P\ (curry\ f)\ g$,
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - - - refl refl]
for P

lemma *monotone-Pair*:
$[\![$ *monotone ord orda f*; *monotone ord ordb g* $]\!]$
$\Longrightarrow$ *monotone ord* (*rel-prod orda ordb*) ($\lambda x.$ ($f\ x$, $g\ x$))
by(*simp add*: *monotone-def*)

lemma *cont-Pair*:
$[\![$ *cont lub ord luba orda f*; *cont lub ord lubb ordb g* $]\!]$
$\Longrightarrow$ *cont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) ($\lambda x.$ ($f\ x$, $g\ x$))
by(*rule contI*)(*auto simp add*: *prod-lub-def image-image dest*!: *contD*)

lemma *mcont-Pair*:
$[\![$ *mcont lub ord luba orda f*; *mcont lub ord lubb ordb g* $]\!]$
$\Longrightarrow$ *mcont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) ($\lambda x.$ ($f\ x$, $g\ x$))
by(*rule mcontI*)(*simp-all add*: *monotone-Pair mcont-mono cont-Pair*)

lemma *mono2mono-emeasure-spmf* [*THEN lfp.mono2mono*]:
shows *monotone-emeasure-spmf*:
monotone (*ord-spmf* (=)) ($\leq$) ($\lambda p.$ *emeasure* (*measure-spmf p*))
by(*rule monotoneI le-funI ord-spmf-eqD-emeasure*)+

lemma *cont-emeasure-spmf*: *cont lub-spmf* (*ord-spmf* (=)) *Sup* ($\leq$) ($\lambda p.$ *emeasure* (*measure-spmf p*))
by (*rule contI*) (*simp add*: *emeasure-lub-spmf fun-eq-iff image-comp*)

lemma *mcont2mcont-emeasure-spmf* [*THEN lfp.mcont2mcont*, *cont-intro*]:
shows *mcont-emeasure-spmf*: *mcont lub-spmf* (*ord-spmf* (=)) *Sup* ($\leq$) ($\lambda p.$ *emeasure* (*measure-spmf p*))
by(*simp add*: *mcont-def monotone-emeasure-spmf cont-emeasure-spmf*)

lemma *mcont2mcont-emeasure-spmf'* [*THEN lfp.mcont2mcont*, *cont-intro*]:
shows *mcont-emeasure-spmf'*: *mcont lub-spmf* (*ord-spmf* (=)) *Sup* ($\leq$) ($\lambda p.$ *emeasure* (*measure-spmf p*) *A*)
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* [*cont-intro*]:
assumes *g*: $\bigwedge y.$ *mcont luba orda lub-spmf* (*ord-spmf* (=)) ($g\ y$)
shows *mcont luba orda lub-spmf* (*ord-spmf* (=)) ($\lambda x.$ *bind-pmf p* ($\lambda y.$ $g\ y\ x$))
using *mcont-bind-spmf*[**where** $f{=}\lambda$-. *spmf-of-pmf p* **and** $g{=}g$, *OF - assms*] **by**(*simp*)

lemma *ennreal-less-top-iff*: $x < \top \longleftrightarrow x \neq (\top :: ennreal)$
by(*cases x*) *simp-all*

lemma *type-definition-Domainp*:
fixes *Rep Abs A T*
assumes *type*: *type-definition Rep Abs A*
assumes *T-def*: $T \equiv (\lambda(x::'a)\ (y::'b).\ x = Rep\ y)$
shows $Domainp\ T = (\lambda x.\ x \in A)$
proof −
interpret *type-definition Rep Abs A* **by**(*rule type*)
show *?thesis* **unfolding** *Domainp-iff*[*abs-def*] *T-def fun-eq-iff* **by**(*metis Abs-inverse Rep*)
qed

context includes *lifting-syntax* **begin**

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

lemma *lossless-spmf-parametric* [*transfer-rule*]:
(*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*:
$\bigwedge A.\ [\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow bind\text{-}spmf\ (spmf\text{-}of\text{-}set\ A) = bind\text{-}pmf\ (pmf\text{-}of\text{-}set\ A)$
by(*simp add*: *spmf-of-set-def fun-eq-iff del*: *spmf-of-pmf-pmf-of-set*)

lemma *set-pmf-bind-spmf*: $set\text{-}pmf\ (bind\text{-}spmf\ M\ f) = set\text{-}pmf\ M \gg\!= map\text{-}option\text{-}set\ (set\text{-}pmf \circ f)$
by(*auto 4 3 simp add*: *bind-spmf-def split*: *option.splits intro*: *rev-bexI*)

lemma *set-pmf-spmf-of-set*:
$set\text{-}pmf\ (spmf\text{-}of\text{-}set\ A) = (if\ finite\ A \wedge A \neq \{\}\ then\ Some\ `\ A\ else\ \{None\})$
by(*simp add*: *spmf-of-set-def spmf-of-pmf-def del*: *spmf-of-pmf-pmf-of-set*)

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**
(*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 *ub*: ⋀*x*. *spmf p x* ≤ *f x*
  and *sum*: (∫⁺ *x*. *f x* ∂*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*)(*auto simp add*: *sum*[*symmetric*]
intro!: *nn-integral-mono split*: *split-indicator*)

    have *weight-spmf p* = ∫⁺ *y*. *spmf p y* ∂*count-space UNIV*
     by(*simp add*: *nn-integral-spmf space-measure-spmf measure-spmf*.*emeasure-eq-measure*)
    also have ... = (∫⁺ *y*. *ennreal* (*spmf p y*) ∗ *indicator* (− {*x*}) *y* ∂*count-space*
UNIV) +
      (∫⁺ *y*. *spmf p y* ∗ *indicator* {*x*} *y* ∂*count-space UNIV*)
     by(*subst nn-integral-add*[*symmetric*])(*auto intro*!: *nn-integral-cong split*: *split-indicator*)
    also have ... ≤ (∫⁺ *y*. *ennreal* (*f y*) ∗ *indicator* (− {*x*}) *y* ∂*count-space UNIV*)
+ *spmf p x*
      using *ub* **by**(*intro add-mono nn-integral-mono*)(*auto split*: *split-indicator intro*:
ennreal-leI)
    also have ... < (∫⁺ *y*. *ennreal* (*f y*) ∗ *indicator* (− {*x*}) *y* ∂*count-space UNIV*)
+ (∫⁺ *y*. *f y* ∗ *indicator* {*x*} *y* ∂*count-space UNIV*)
      using ∗ *x* **by**(*simp add*: *ennreal-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 ⋀*while'*. *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*
unfolding *loop-spmf.while-def*[*abs-def*]
apply(*rule rel-funI*)
apply(*rule rel-funI*)
apply(*rule fixp-spmf-parametric*[*OF loop-spmf.while.mono loop-spmf.while.mono*])
subgoal premises [*transfer-rule*] **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'*
unfolding *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 ∀ *s. lossless-spmf* (*while s*)
proof(*rule ccontr*)
assume ¬ *?thesis*
then obtain *s* **where** *s*: ¬ *lossless-spmf* (*while s*) **by** *blast*
hence *True*: *guard s* **by**(*simp add*: *while.simps split*: *if-split-asm*)

from *p*[*OF this*] **have** *p-le-1*: $p \leq 1$ **using** *pmf-le-1* **by**(*rule order-trans*)
have *new-bound*: $p * (1 - k) + k \leq$ *weight-spmf* (*while s*)
if *k*: $0 \leq k$ $k \leq 1$ **and** *k-le*: $\bigwedge s.\ k \leq$ *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 ... ≤ 1 **by** *simp*
finally show *?thesis* **using** *False* **by**(*simp add*: *while.simps*)
next
case *True*
let *?M* = λ*s. measure-spmf* (*body s*)
have *bounded*: |∫ *s''. weight-spmf* (*while s''*) ∂*?M s'*| ≤ *1* **for** *s'*
using *integral-nonneg-AE*[*of* λ*s''. weight-spmf* (*while s''*) *?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* ≤ *measure* (*?M s*) {*s'*. ¬ *guard s'*} **using** *p*[*OF True*]
by(*simp add*: *spmf-conv-measure-spmf measure-map-spmf vimage-def*)
hence *p* ∗ (*1* − *k*) + *k* ≤ *measure* (*?M s*) {*s'*. ¬ *guard s'*} ∗ (*1* − *k*) + *k*
using *k* **by**(*intro add-mono mult-right-mono*)(*simp-all*)
also have ... = ∫ *s'. indicator* {*s'*. ¬ *guard s'*} *s'* ∗ (*1* − *k*) + *k* ∂*?M s*
using *True* **by**(*simp add*: *ennreal-less-top-iff lossless lossless-weight-spmfD*)
also have ... = ∫ *s'. indicator* {*s'*. ¬ *guard s'*} *s'* + *indicator* {*s'*. *guard s'*} *s'* ∗ *k* ∂*?M s*
by(*rule Bochner-Integration.integral-cong*)(*simp-all split*: *split-indicator*)
also have ... = ∫ *s'. indicator* {*s'*. ¬ *guard s'*} *s'* + *indicator* {*s'*. *guard s'*} *s'* ∗ ∫ *s''. k* ∂*?M s'* ∂*?M s*
by(*rule Bochner-Integration.integral-cong*)(*auto simp add*: *lossless lossless-weight-spmfD split*: *split-indicator*)
also have ... ≤ ∫ *s'. indicator* {*s'*. ¬ *guard s'*} *s'* + *indicator* {*s'*. *guard s'*} *s'* ∗ ∫ *s''. weight-spmf* (*while s''*) ∂*?M s'* ∂*?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 ... = ∫ *s'*. (*if* ¬ *guard s'* *then 1 else* ∫ *s''. weight-spmf* (*while s''*) ∂*?M s'*) ∂*?M s*
by(*rule Bochner-Integration.integral-cong*)(*simp-all split*: *split-indicator*)
also have ... = ∫ *s'. weight-spmf* (*while s'*) ∂*measure-spmf* (*body s*)
by(*rule Bochner-Integration.integral-cong*; *simp add*: *while.simps weight-bind-spmf o-def*)
also have ... = *weight-spmf* (*while s*) **using** *True*

```
      by(simp add: while.simps weight-bind-spmf o-def)
    finally show ?thesis .
  qed

  define k where k ≡ INF s. weight-spmf (while s)
  define k' where k' ≡ p ∗ (1 − k) + k
  from s have weight-spmf (while s) < 1
    using weight-spmf-le-1[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 ≤ k unfolding k-def by(auto intro: cINF-greatest simp add: weight-spmf-nonneg)
  moreover from ‹k < 1› have k ≤ 1 by simp
  moreover have k ≤ weight-spmf (while s) for s unfolding k-def
    by(rule cINF-lower)(auto intro!: bdd-belowI2 weight-spmf-nonneg)
  ultimately have ⋀s. k' ≤ weight-spmf (while s)
    unfolding k'-def by(rule new-bound)
  hence k' ≤ k unfolding k-def by(auto intro: cINF-greatest)
  also have k < k' using p-pos ‹k < 1› by(auto simp add: k'-def)
  finally show False by simp
 qed
 thus ?thesis by blast
qed

primrec iter :: nat ⇒ 'a ⇒ '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]: ¬ guard s ⟹ iter n s = return-spmf s
  by(induction n) simp-all

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

lemma iter-Suc2: iter (Suc n) s = bind-spmf (iter n s) (λs. if guard s then body
s 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: (⋀s. guard s ⟹ lossless-spmf (body s)) ⟹ lossless-spmf
(iter n s)
  by(induction n arbitrary: s) simp-all

lemma iter-mono-emeasure1:
   emeasure (measure-spmf (iter n s)) {s. ¬ guard s} ≤ emeasure (measure-spmf
(iter (Suc n) s)) {s. ¬ guard s}
  (is ?lhs ≤ ?rhs)
proof(cases guard s)
```

case *True*
have *?lhs* = *emeasure* (*measure-spmf* (*bind-spmf* (*iter n s*) *return-spmf*)) {*s*. ¬ *guard s*} **by** *simp*
also have ... = $\int^+$ *s′*. *emeasure* (*measure-spmf* (*return-spmf s′*)) {*s*. ¬ *guard s*} ∂*measure-spmf* (*iter n s*)
by(*simp del*: *bind-return-spmf add*: *measure-spmf-bind o-def emeasure-bind*[**where** *N*=*measure-spmf* -] *space-measure-spmf Pi-def space-subprob-algebra*)
also have ... ≤ $\int^+$ *s′*. *emeasure* (*measure-spmf* (*if guard s′ then body s′ else return-spmf s′*)) {*s*. ¬ *guard s*} ∂*measure-spmf* (*iter n s*)
by(*rule nn-integral-mono*)(*simp add*: *measure-spmf-return-spmf*)
also have ... = *?rhs*
by(*simp add*: *iter-Suc2 measure-spmf-bind o-def emeasure-bind*[**where** *N*=*measure-spmf* -] *space-measure-spmf Pi-def space-subprob-algebra del*: *iter.simps*)
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*: *incseq ?f* **by**(*rule incseq-SucI le-funI iter-mono-emeasure1*)+

from *True* **have** *?lhs′* = $\int^+$ *s′*. *emeasure* (*measure-spmf* (*while′ s′*)) *UNIV* ∂*measure-spmf* (*body s*)
by(*simp add*: *measure-spmf-bind o-def emeasure-bind*[**where** *N*=*measure-spmf* -] *space-measure-spmf Pi-def space-subprob-algebra*)
also have ... ≤ $\int^+$ *s′*. (*SUP n*. *?f n s′*) ∂*measure-spmf* (*body s*)
by(*rule nn-integral-mono*)(*rule step.IH*)
also have ... = (*SUP n*. $\int^+$ *s′*. *?f n s′* ∂*measure-spmf* (*body s*)) **using** *inc*
by(*subst 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*[**where** *N*=*measure-spmf* -] *space-measure-spmf Pi-def space-subprob-algebra*)
also have ... ≤ (*SUP n*. *?f n s*)
by(*rule SUP-mono*)(*auto intro*: *exI*[**where** *x*=*Suc* -])
finally show *?thesis* **.**
next
case *False*

then have *?lhs'* = *emeasure* (*measure-spmf* (*iter 0 s*)) {*s*. ¬ *guard s*}
by(*simp add*: *measure-spmf-return-spmf*)
also have ‹. . . ≤ *?rhs'*› **by**(*rule SUP-upper*) *simp*
finally show *?thesis* **.**
qed
qed
also have . . . = *ennreal* (*SUP n*. *measure* (*measure-spmf* (*iter n s*)) {*s*. ¬ *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* ≤ (*SUP n*. *measure* (*measure-spmf* (*iter n s*)) {*s*. ¬ *guard s*})
by(*rule cSUP-upper2*)(*auto intro*!: *bdd-aboveI*[**where** *M=1*] *simp add*: *measure-spmf.subprob-measure-le-1*)
ultimately show *?lhs* ≤ *?rhs* **by**(*simp add*: *measure-spmf.emeasure-eq-measure space-measure-spmf*)

show *?rhs* ≤ *?lhs*
proof(*rule cSUP-least*)
show *measure* (*measure-spmf* (*iter n s*)) {*s*. ¬ *guard s*} ≤ *weight-spmf* (*while s*) (**is** *?f n s* ≤ -) **for** *n*
proof(*induction n arbitrary*: *s*)
case *0* **show** *?case*
by(*simp add*: *measure-spmf-return-spmf measure-return while-simps split*: *split-indicator*)
next
case (*Suc n*)
show *?case*
proof(*cases guard s*)
case *True*
have *?f* (*Suc n*) *s* = $\int^+$ *s'*. *?f n s'* ∂*measure-spmf* (*body s*)
using *True* **unfolding** *measure-spmf.emeasure-eq-measure*[*symmetric*]
by(*simp add*: *measure-spmf-bind o-def emeasure-bind*[**where** *N=measure-spmf* -] *space-measure-spmf Pi-def space-subprob-algebra*)
also have . . . ≤ $\int^+$ *s'*. *weight-spmf* (*while s'*) ∂*measure-spmf* (*body s*)
by(*rule nn-integral-mono ennreal-leI Suc.IH*)+
also have . . . = *weight-spmf* (*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=measure-spmf* -] *space-measure-spmf Pi-def space-subprob-algebra*)
finally show *?thesis* **by**(*simp*)
next
case *False* **then show** *?thesis*
by(*simp add*: *measure-spmf-return-spmf measure-return while-simps split*: *split-indicator*)
qed
qed
qed *simp*
qed

```
lemma termination-0-1:
  assumes p: ⋀s. guard s ⟹ p ≤ weight-spmf (while s)
    and p-pos: 0 < p
    and lossless: ⋀s. guard s ⟹ lossless-spmf (body s)
  shows lossless-spmf (while s)
  unfolding lossless-spmf-def
proof(rule antisym)
  let ?X = {s. ¬ guard s}
  show weight-spmf (while s) ≤ 1 by(rule weight-spmf-le-1)

  define p' where p' ≡ p / 2
  have p'-pos: p' > 0 and p' < p using p-pos by(simp-all add: p'-def)

  have ∃ n. p' < measure (measure-spmf (iter n s)) ?X if guard s for s using
p[OF that] ‹p' < p›
    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' ≤ measure (measure-spmf (iter (N s) 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 λs. iter (N s) s .

  have 1 = weight-spmf (fuse.while s)
    by(rule lossless-weight-spmfD[symmetric])
      (rule fuse.termination-0-1-immediate; auto simp add: spmf-map vimage-def
intro: p' p'-pos lossless-iter lossless)
  also have ... ≤ (⨆n. measure (measure-spmf (iter n s)) ?X)
    unfolding fuse.weight-while-conv-iter
  proof(rule cSUP-least)
    fix n
    have emeasure (measure-spmf (fuse.iter n s)) ?X ≤ (SUP n. emeasure (measure-spmf
(iter n s)) ?X)
    proof(induction n arbitrary: s)
      case 0 show ?case by(auto intro!: SUP-upper2[where i=0])
    next
      case (Suc n)
      have inc: incseq (λn s'. emeasure (measure-spmf (iter n s')) ?X)
        by(rule incseq-SucI le-funI iter-mono-emeasure1)+

      have emeasure (measure-spmf (fuse.iter (Suc n) s)) ?X = emeasure (measure-spmf
(iter (N s) s ⤜ fuse.iter n)) ?X
        by simp
      also have ... = ∫⁺ s'. emeasure (measure-spmf (fuse.iter n s')) ?X ∂measure-spmf
(iter (N s) s)
        by(simp add: measure-spmf-bind o-def emeasure-bind[where N=measure-spmf
-] space-measure-spmf Pi-def space-subprob-algebra)
      also have ... ≤ ∫⁺ s'. (SUP n. emeasure (measure-spmf (iter n s')) ?X)
```

∂measure-spmf (iter (N s) s)
by(*rule nn-integral-mono Suc.IH*)+
also have … = (*SUP n.* $\int^{+}$ *s′. emeasure (measure-spmf (iter n s′)) ?X ∂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 … ≤ (*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(*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* ≤ (*SUP n. measure (measure-spmf (iter n s)) ?X*)
by(*rule cSUP-upper2*)(*auto intro*!: *bdd-aboveI*[**where** *M*=*1*] *simp add*: *measure-spmf.subprob-measure-le-1*)
ultimately show *measure (measure-spmf (fuse.iter n s)) ?X* ≤ …
by(*simp add*: *measure-spmf.emeasure-eq-measure*)
qed *simp*
finally show *1* ≤ *weight-spmf (while s)* **unfolding** *weight-while-conv-iter* **.**
qed

end

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 −
{ **assume** ∃(*Rep* :: *′s′ ⇒ ′s*) *Abs. type-definition Rep Abs* {*s. I s*}
then obtain *Rep* :: *′s′ ⇒ ′s* **and** *Abs* **where** *td*: *type-definition Rep Abs* {*s. I s*} **by** *blast*
then interpret *td*: *type-definition Rep Abs* {*s. I s*} **.**
define *cr* **where** *cr* ≡ λ*x y. x* = *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*

define guard′ **where** guard′ ≡ (Rep ---> id) guard
have [transfer-rule]: (cr ===> (=)) guard guard′ **by**(simp add: rel-fun-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-fun-def body1′-def body1-def cr-def spmf-rel-map td.Rep[simplified] invar td.Abs-inverse intro!: 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)

have ⋀s. guard′ s ⟹ p ≤ spmf (map-spmf guard′ (body1′ s)) False
by(transfer fixing: p)(simp add: body1-def p)
moreover note p-pos
moreover have ⋀s. guard′ s ⟹ lossless-spmf (body1′ s) **by** transfer(simp add: lossless body1-def)
ultimately have lossless-spmf (loop-spmf.while guard′ body1′ s′) **by**(rule loop-spmf.termination-0-1-immediate)
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

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−
{ assume ∃(Rep :: ′s′ ⇒ ′s) Abs. type-definition Rep Abs {s. I s}
then obtain Rep :: ′s′ ⇒ ′s **and** Abs **where** td: type-definition Rep Abs {s. I s} **by** blast
then interpret td: type-definition Rep Abs {s. I s} **.**
define cr **where** cr ≡ λx y. x = 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

define guard′ **where** guard′ ≡ (Rep ---> id) guard
have [transfer-rule]: (cr ===> (=)) guard guard′ **by**(simp add: rel-fun-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-fun-def body1′-def body1-def cr-def spmf-rel-map td.Rep*[*simplified*] *invar td.Abs-inverse intro*!: *rel-spmf-reflI*)
define s' **where** $s' \equiv Abs\ s$
have [*transfer-rule*]: $cr\ s\ s'$ **by**(*simp add*: *s′-def cr-def I td.Abs-inverse*)

interpret *loop-spmf guard′ body1′* **.**

note *UNIV-parametric-pred*[*transfer-rule*]
have $\bigwedge s.\ guard'\ s \Longrightarrow p \leq weight\text{-}spmf\ (while\ s)$
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-pos*
moreover have $\bigwedge s.\ guard'\ s \Longrightarrow lossless\text{-}spmf\ (body1'\ s)$ **by** *transfer*(*simp add*: *lossless body1-def*)
ultimately have $lossless\text{-}spmf\ (while\ s')$ **by**(*rule termination-0-1*)
hence $lossless\text{-}spmf\ (loop\text{-}spmf.while\ guard\ body1\ s)$ **by** *transfer* **}**
from *this*[*cancel-type-definition*] *I* **show** *?thesis* **by**(*auto cong*: *loop-spmf-while-cong*)
qed

3.2 Variant rule

context *loop-spmf* **begin**

lemma *termination-variant*:
fixes *bound* :: *nat*
assumes *bound*: $\bigwedge s.\ guard\ s \Longrightarrow f\ s \leq bound$
and *step*: $\bigwedge s.\ guard\ s \Longrightarrow p \leq spmf\ (map\text{-}spmf\ (\lambda s'.\ f\ s' < f\ s)\ (body\ s))\ True$
and *p-pos*: $0 < p$
and *lossless*: $\bigwedge s.\ guard\ s \Longrightarrow lossless\text{-}spmf\ (body\ s)$
shows $lossless\text{-}spmf\ (while\ s)$
proof −
define p' **and** n **where** $p' \equiv min\ p\ 1$ **and** $n \equiv bound + 1$
have *p′-pos*: $0 < p'$ **and** *p′-le-1*: $p' \leq 1$
and *step′*: $guard\ s \Longrightarrow p' \leq measure\ (measure\text{-}spmf\ (body\ s))\ \{s'.\ f\ s' < f\ s\}$ **for** *s*
using *p-pos step*[*of s*] **by**(*simp-all add*: *p′-def spmf-map vimage-def*)
have $p'\ \widehat{}\ n \leq weight\text{-}spmf\ (while\ s)$ **if** $f\ s < n$ **for** *s* **using** *that*
proof(*induction n arbitrary*: *s*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
show *?case*
proof(*cases guard s*)
case *False*
hence $weight\text{-}spmf\ (while\ s) = 1$ **by**(*simp add*: *while.simps*)
thus *?thesis* **using** *p′-le-1 p-pos*
by *simp*(*meson less-eq-real-def mult-le-one p′-pos power-le-one zero-le-power*)
next

case *True*
let *?M* = *measure-spmf* (*body s*)
have *p′ ^ Suc n* ≤ (∫ *s′*. *indicator* {*s′*. *f s′* < *f s*} *s′* ∂*?M*) ∗ *p′ ^ n*
using *step′*[*OF True*] *p′-pos* **by**(*simp add*: *mult-right-mono*)
also have ... = (∫ *s′*. *indicator* {*s′*. *f s′* < *f s*} *s′* ∗ *p′ ^ n* ∂*?M*) **by** *simp*
also have ... ≤ (∫ *s′*. *indicator* {*s′*. *f s′* < *f s*} *s′* ∗ *weight-spmf* (*while s′*) ∂*?M*)
using *Suc.prems p′-le-1 p′-pos*
by(*intro integral-mono*)(*auto simp add*: *Suc.IH power-le-one weight-spmf-le-1 split*: *split-indicator intro*!: *measure-spmf.integrable-const-bound*[**where** *B=1*])
also have ... ≤ ... + (∫ *s′*. *indicator* {*s′*. *f s′* ≥ *f s*} *s′* ∗ *weight-spmf* (*while s′*) ∂*?M*)
by(*simp add*: *integral-nonneg-AE weight-spmf-nonneg*)
also have ... = ∫ *s′*. *weight-spmf* (*while s′*) ∂*?M*
by(*subst Bochner-Integration.integral-add*[*symmetric*])
(*auto intro*!: *Bochner-Integration.integral-cong measure-spmf.integrable-const-bound*[**where** *B=1*] *weight-spmf-le-1 split*: *split-indicator*)
also have ... = *weight-spmf* (*while s*)
using *True* **by**(*subst* (*1 2*) *while.simps*)(*simp add*: *weight-bind-spmf o-def*)
finally show *?thesis* **.**
qed
qed
moreover have *0* < *p′ ^ n* **using** *p′-pos* **by** *simp*
ultimately show *?thesis* **using** *lossless*
proof(*rule termination-0-1-invar*)
show *f s* < *n* **if** *guard s guard s* ⟶ *f s* < *n* **for** *s* **using** *that* **by** *simp*
show *guard s* ⟶ *f s* < *n* **using** *bound*[*of s*] **by**(*auto simp add*: *n-def*)
show *guard s′* ⟶ *f s′* < *n* **for** *s′* **using** *bound*[*of s′*] **by**(*clarsimp simp add*: *n-def*)
qed
qed

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* (λ*s′*. *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*: *I s*
shows *lossless-spmf* (*loop-spmf.while guard body s*)
including *lifting-syntax*
proof −
{ **assume** ∃(*Rep* :: *′s′* ⇒ *′s*) *Abs*. *type-definition Rep Abs* {*s*. *I s*}
then obtain *Rep* :: *′s′* ⇒ *′s* **and** *Abs* **where** *td*: *type-definition Rep Abs* {*s*. *I s*} **by** *blast*

then interpret *td*: *type-definition Rep Abs* {*s. I s*} **.**
define *cr* **where** *cr* ≡ *λx y. x* = *Rep y*
have [*transfer-rule*]: *bi-unique cr right-total cr* **using** *td cr-def* **by**(*rule type-def-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-fun-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-fun-def body1′-def body1-def cr-def spmf-rel-map td.Rep*[*simplified*] *invar td.Abs-inverse intro*!: *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-fun-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

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* < (*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.

```
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]
      by(subst bernoulli.simps)(simp add: UNIV-bool max-def field-simps spmf-of-pmf-pmf-of-set[symmetric]
pmf-bind-pmf-of-set ennreal-pmf-bind nn-integral-pmf-of-set del: spmf-of-pmf-pmf-of-set)
    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)
```

show *?case* **using** *step.prems step.IH*[*of 2 ∗ p*] *step.IH*[*of 2 ∗ p − 1*]
by(*auto simp add*: *UNIV-bool max-def divide-le-posI-ennreal ennreal-mult-numeral numeral-mult-ennreal field-simps spmf-of-pmf-pmf-of-set*[*symmetric*] *ennreal-pmf-bind nn-integral-pmf-of-set one-plus-ennreal simp del*: *spmf-of-pmf-pmf-of-set ennreal-plus*)
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

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 = *do* {
    b ← *bernoulli p*;
    if b then return-spmf 0 else map-spmf ((+) *1*) (*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 ⋀*geometric-spmf'*. *P geometric-spmf'* ⟹ *P* (λ*p*. *bernoulli p* ⤜ (λ*b*. *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*)

context
  fixes *p* :: *real*
  and *body* :: *bool* × *nat* ⇒ (*bool* × *nat*) *spmf*
  defines [*simp*]: *body* ≡ λ(*b*, *x*). *map-spmf* (λ*b'*. (¬ *b'*, *x* + (*if b' then 0 else 1*))) (*bernoulli p*)
begin

interpretation *loop-spmf fst body*
  rewrites *body* ≡ λ(*b*, *x*). *map-spmf* (λ*b'*. (¬ *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(rewrite while.simps)
      apply(clarsimp simp add: map-spmf-bind-spmf bind-map-spmf intro!: ord-spmf-bind-reflI)
        apply(rewrite 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]
        by(rewrite geometric-spmf.simps)(clarsimp simp add: map-spmf-bind-spmf
bind-map-spmf spmf.map-comp o-def intro!: ord-spmf-bind-reflI)
      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) ⟷ p > 0
proof(cases 0 < p ∧ p < 1)
  case True
  let ?body = λ(b, x :: nat). map-spmf (λb′. (¬ b′, x + (if b′ then 0 else 1)))
(bernoulli p)
  have lossless-spmf (while (True, 0))
  proof(rule termination-0-1-immediate)
    have {x. x} = {True} by auto
    then show p ≤ spmf (map-spmf fst (?body s)) False for s :: bool × nat using
True
      by(cases s)(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*)

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(*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*[*symmetri* *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*] *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*}) (λ*c*. *bind-pmf* (*pmf-of-set UNIV*) (λ*b*. *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* (*pmf-of-set* {..<*v*}) (*pmf-of-set UNIV*))) *f*
(**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′* ≡ *i div 2*
define *b* **where** *b* ≡ *odd i*
have *i*: *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*) (λ*b*. *bind-pmf* (*pmf-of-set* {..<*v*}) (λ*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* (λ(*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-on ?f* ({..<*v*} × *UNIV*) **by**(*auto simp add*: *inj-on-def*)
define *i′* **where** *i′* ≡ *if i* ≥ *v then i* − *v else i*
define *b* **where** *b* ≡ *i* ≥ *v*
have *i*: *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 $i \notin$ *set-pmf ?l* $i \notin$ *set-pmf ?r*
using *assms* **by**(*auto simp add*: *lessThan-empty-iff split*: *if-split-asm*)
thus *?thesis* **by**(*simp add*: *set-pmf-iff del*: *set-map-pmf*)
qed
qed
finally show *?thesis* **.**
qed

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 (*spmf*) *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 $(2 * v)$ $(2 * c + ($*if b then 1 else 0*$))$ })

lemma *fast-dice-roll-fixp-induct* [*case-names adm bottom step*]:
assumes *spmf.admissible* (λ*fast-dice-roll*. *P* (*curry fast-dice-roll*))
and *P* (λ*v c*. *return-pmf None*)
and ⋀*fdr*. *P fdr* ⟹ *P* (λ*v c*. *if* $v \geq n$ *then if* $c < n$ *then return-spmf c else fdr* $(v - n)$ $(c - n)$
else bind-spmf coin-spmf (λ*b*. *fdr* $(2 * v)$ $(2 * c + ($*if b then 1 else 0*$))$))
shows *P fast-dice-roll*
using *assms* **by**(*rule fast-dice-roll.fixp-induct*)

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

lemma *spmf-fast-dice-roll-ub*:
assumes $0 < v$
shows *spmf* (*bind-pmf* (*pmf-of-set* {..<*v*}) (*fast-dice-roll v*)) $x \leq$ (*if* $x < n$ *then* $1 / n$ *else* 0)
(**is** *?lhs* ≤ *?rhs*)
proof −
have *ennreal ?lhs* ≤ *ennreal ?rhs* **using** *assms*
proof(*induction arbitrary*: *v x rule*: *fast-dice-roll-fixp-induct*)

case *adm* **thus** *?case*
by(*rule cont-intro ccpo-class.admissible-leI*)+ *simp-all*
case *bottom* **thus** *?case* **by** *simp*
case (*step fdr*)
show *?case* (**is** *?lhs* ≤ *?rhs*)
proof(*cases n* ≤ *v*)
case *le*: *True*
then have *?lhs* = *spmf* (*bind-pmf* (*pmf-of-set* {..<*v*}) (λ*c*. *if c* < *n then return-spmf c else fdr* (*v* − *n*) (*c* − *n*))) *x*
by *simp*
also have ... = (∫ + *c'*. *indicator* (*if x* < *n then* {*x*} *else* {}) *c'* ∂*measure-pmf* (*pmf-of-set* {..<*v*})) +
(∫ + *c'*. *indicator* {*n* ..< *v*} *c'* ∗ *spmf* (*fdr* (*v* − *n*) (*c'* − *n*)) *x* ∂*measure-pmf* (*pmf-of-set* {..<*v*}))
(**is** *?then* = *?found* + *?continue*) **using** *step.prems*
by(*subst nn-integral-add*[*symmetric*])(*auto simp add*: *ennreal-pmf-bind AE-measure-pmf-iff lessThan-empty-iff split*: *split-indicator intro*!: *nn-integral-cong-AE*)
also have *?found* = (*if x* < *n then 1 else 0*) / *v* **using** *step.prems le*
by(*auto simp add*: *measure-pmf.emeasure-eq-measure measure-pmf-of-set lessThan-empty-iff Iio-Int-singleton*)
also have *?continue* = (∫ + *c'*. *indicator* {*n* ..< *v*} *c'* ∗ *1* / *v* ∗ *spmf* (*fdr* (*v* − *n*) (*c'* − *n*)) *x* ∂*count-space UNIV*)
using *step.prems* **by**(*auto simp add*: *nn-integral-measure-pmf lessThan-empty-iff ennreal-mult*[*symmetric*] *intro*!: *nn-integral-cong split*: *split-indicator*)
also have ... = (*if v* = *n then 0 else ennreal* ((*v* − *n*) / *v*) ∗ *spmf* (*bind-pmf* (*pmf-of-set* {*n*..<*v*}) (λ*c'*. *fdr* (*v* − *n*) (*c'* − *n*))) *x*)
using *le step.prems*
by(*subst ennreal-pmf-bind*)(*auto simp add*: *ennreal-mult*[*symmetric*] *nn-integral-measure-pmf nn-integral-0-iff-AE AE-count-space nn-integral-cmult*[*symmetric*] *split*: *split-indicator*)
also {
assume ∗: *n* < *v*
then have *pmf-of-set* {*n*..<*v*} = *map-pmf* ((+) *n*) (*pmf-of-set* {..<*v* − *n*})
by(*subst map-pmf-of-set-inj*)(*auto 4 3 simp add*: *inj-on-def lessThan-empty-iff intro*!: *arg-cong*[**where** *f*=*pmf-of-set*] *intro*: *rev-image-eqI*[**where** *x*=- − *n*] *diff-less-mono*)
also have *bind-pmf* ... (λ*c'*. *fdr* (*v* − *n*) (*c'* − *n*)) = *bind-pmf* (*pmf-of-set* {..<*v* − *n*}) (*fdr* (*v* − *n*))
by(*simp add*: *bind-map-pmf*)
also have *ennreal* (*spmf* ... *x*) ≤ (*if x* < *n then 1* / *n else 0*)
by(*rule step.IH*)(*simp add*: ∗)
also note *calculation* }
then have ... ≤ *ennreal* ((*v* − *n*) / *v*) ∗ (*if x* < *n then 1* / *n else 0*) **using** *le*
by(*cases v* = *n*)(*auto split del*: *if-split intro*: *divide-right-mono mult-left-mono*)
also have ... = (*v* − *n*) / *v* ∗ (*if x* < *n then 1* / *n else 0*) **by**(*simp add*: *ennreal-mult*[*symmetric*])
finally show *?thesis* **using** *le* **by**(*auto simp add*: *add-mono field-simps of-nat-diff ennreal-plus*[*symmetric*] *simp del*: *ennreal-plus*)
next
case *False*

then have *?lhs* = *spmf* (*bind-pmf* (*pmf-of-set* {..<*v*}) (λ*c*. *bind-pmf* (*pmf-of-set* *UNIV*) (λ*b*. *fdr* (*2* ∗ *v*) (*2* ∗ *c* + (*if b then 1 else 0*))))) *x*
by(*simp add*: *bind-spmf-spmf-of-set*)
also have ... = *spmf* (*bind-pmf* (*pmf-of-set* {..<*2* ∗ *v*}) (*fdr* (*2* ∗ *v*))) *x* **using** *step.prems*
by(*simp add*: *sample-bits-fusion*[*symmetric*])
also have ... ≤ *?rhs* **using** *step.prems* **by**(*intro step.IH*) *simp*
finally show *?thesis* **.**
qed
qed
thus *?thesis* **by** *simp*
qed

lemma *spmf-fast-uniform-ub*:
spmf fast-uniform x ≤ (*if x* < *n then 1* / *n else 0*)
proof −
have {..<*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*]: *fast-dice-roll 0 c* = *return-pmf None*
by(*induction arbitrary*: *c rule*: *fast-dice-roll-fixp-induct*)(*simp-all add*: *bind-eq-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* ^(*nat* ⌈*log 2* (*max 1 n*) − *log 2 v*⌉) *in*
map-spmf (λ*bs*. (*x* ∗ *v*, *x* ∗ *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 ← *coin-spmf*;
fdr-step (*2* ∗ *v*) (*2* ∗ *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* ≡ λ*v* :: *nat*. *2* ^(*nat* ⌈*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-mono)
    moreover have {..<1} = {0 :: nat} by auto
   ultimately show ?thesis using le v by(simp add: fdr-step-def spmf-of-set-singleton)
  next
    case less: False
    hence even: even (x v) using v by(simp add: x-def)
    with x-pos have x-ge-1: x v > 1 by(cases x v = 1) auto
    have *: x (2 * v) = x v div 2 using v less unfolding x-def
      apply(simp add: log-mult diff-add-eq-diff-diff-swap)
     apply(rewrite in - = 2 ^ ¤ div - le-add-diff-inverse2[symmetric, where b=1])
       apply (simp add: Suc-leI)
      apply(simp del: Suc-pred)
      done

    have ?lhs = map-spmf (λbs. (x v * v, x v * c + bs)) (spmf-of-set {..<x v})
      using v by(simp add: fdr-step-def x-def Let-def)
    also from even have ... = bind-pmf (pmf-of-set {..<2 * (x v div 2)}) (λbs.
return-spmf (x v * v, x v * c + bs))
     by(simp add: map-spmf-conv-bind-spmf bind-spmf-spmf-of-set x-pos lessThan-empty-iff)
    also have ... = bind-spmf coin-spmf (λb. bind-spmf (spmf-of-set {..<x v div
2})
      (λc'. return-spmf (x v * v, x v * c + c' + (x v div 2) * (if b then 1 else 0))))
      using x-ge-1
     by(simp add: sample-bits-fusion2[symmetric] bind-spmf-spmf-of-set lessThan-empty-iff
add.assoc)
    also have ... = bind-spmf coin-spmf (λb. map-spmf (λbs. (x (2 * v) * (2 *
v), x (2 * v) * (2 * c + (if b then 1 else 0)) + bs)) (spmf-of-set {..<x (2 * v)}))
      using * even by(simp add: map-spmf-conv-bind-spmf algebra-simps)
    also have ... = ?rhs using v less by(simp add: fdr-step-def Let-def x-def)
    finally show ?thesis .
  qed
qed(simp add: fdr-step-def)

lemma fdr-step-induct [case-names fdr-step]:
  (⋀v c. (⋀b. ⟦v ≠ 0; v < n⟧ ⟹ P (2 * v) (2 * c + (if b then 1 else 0))) ⟹ P
v c)
  ⟹ P v c
apply induction-schema
apply pat-completeness
apply(relation Wellfounded.measure (λ(v, c). n − v))
apply simp-all
done

partial-function (spmf) fdr-alt :: nat ⇒ nat ⇒ nat spmf
where
  fdr-alt v c = do {
    (v', c') ← fdr-step v c;
    if c' < n then return-spmf c' else fdr-alt (v' − n) (c' − n) }
```

```
lemma fast-dice-roll-alt: fdr-alt = fast-dice-roll
proof(intro ext)
  show fdr-alt v c = fast-dice-roll v c for v c
  proof(rule spmf.leq-antisym)
    show ord-spmf (=) (fdr-alt v c) (fast-dice-roll v c)
    proof(induction arbitrary: v c rule: fdr-alt.fixp-induct[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: fdr-step-induct)
        case inner: (fdr-step v c)
        show ?case
          apply(rewrite fdr-step-unfold)
          apply(rewrite fast-dice-roll.simps)
         apply(auto intro!: ord-spmf-bind-reflI simp add: Let-def inner.IH step.IH)
          done
      qed
    qed
    have ord-spmf (=) (fast-dice-roll v c) (fdr-alt v c)
      and fast-dice-roll 0 c = return-pmf None
    proof(induction arbitrary: v c rule: 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 fdr-alt.simps)
        apply(rewrite fdr-step-unfold)
        apply(clarsimp simp add: Let-def)
         apply(auto intro!: ord-spmf-bind-reflI simp add: fdr-alt.simps[symmetric]
step.IH rel-pmf-return-pmf2 set-pmf-bind-spmf o-def set-pmf-spmf-of-set split: if-split-asm)
        done
     case step case 2 from step.IH show ?case by(simp add: Let-def bind-eq-return-pmf-None)
    qed
    then show ord-spmf (=) (fast-dice-roll v c) (fdr-alt v c) by −
  qed
qed

lemma lossless-fdr-step [simp]: lossless-spmf (fdr-step v c) ⟷ v > 0
by(simp add: fdr-step-def Let-def lessThan-empty-iff)

lemma fast-dice-roll-alt-conv-while:
  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(induction arbitrary: v c rule: parallel-fixp-induct-2-1[OF partial-function-definitions-spmf
partial-function-definitions-spmf fdr-alt.mono loop-spmf.while.mono fdr-alt-def 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*)
show *?case* **using** *step.IH*
by(*auto simp add*: *map-spmf-bind-spmf o-def intro*!: *bind-spmf-cong*[*OF refl*])
qed

lemma *lossless-fast-dice-roll*:
assumes $c < v$ $v \leq n$
shows *lossless-spmf* (*fast-dice-roll v c*)
proof(*cases* $v < n$)
case *True*
let $?I = \lambda(v, c).\ c < v \wedge n \leq v \wedge v < 2 * n$
let $?f = \lambda(v, c).$ *if* $n \leq c$ *then* $n + c - v + 1$ *else* 0
have *invar*: $?I\ (v', c')$ **if** *step*: $(v', c') \in$ *set-spmf* (*fdr-step* $(v - n)$ $(c - n)$)
and *I*: $c < v$ $n \leq v$ $v < 2 * n$ **and** *c*: $n \leq c$ **for** v' c' v c
proof(*clarsimp*; *safe*)
define *x* **where** $x =$ *nat* $\lceil log\ 2\ (max\ 1\ n) - log\ 2\ (v - n)\rceil$
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': $v' = 2\ \hat{}\ x * (v - n)$
and c': $c' = 2\ \hat{}\ x * (c - n) + bs$
and *bs*: $bs < 2\ \hat{}\ x$
unfolding *fdr-step-def x-def*[*symmetric*] **by**(*auto simp add*: *Let-def*)
have $2\ \hat{}\ x * (c - n) + bs < 2\ \hat{}\ x * (c - n + 1)$ **unfolding** *distrib-left* **using** *bs*
by(*intro add-strict-left-mono*) *simp*
also have $\ldots \leq 2\ \hat{}\ x * (v - n)$ **using** *I c* **by**(*intro mult-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 $\ldots < 2$ *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 $\ldots \leq 2 * n$ **unfolding** *powr-add* **using** *I c*
by (*simp add*:*powr-diff*)
finally show $v' < 2 * n$ **using** c' **by**(*simp del*: *of-nat-add*)

have $log\ 2\ (n\ /\ (v - n)) \leq x$ **using** *I c* $**$ **by**(*auto simp add*: *x-def log-divide max-def*)
hence 2 *powr* $log\ 2\ (n\ /\ (v - n)) \leq 2$ *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 $n \leq real\ (2\ \hat{}\ x * (v - n))$ **using** *I c* **by**(*simp add*: *field-simps powr-realpow*)
then show $n \leq v'$ **by**(*simp add*: v' *del*: *of-nat-mult*)
qed

have *loop*: *lossless-spmf* (*loop-spmf.while* $(\lambda(v, c).\ n \leq c)$ $(\lambda(v, c).$ *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 ?f (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 v n have x-pos: x > 0 by(auto simp add: x-def max-def)

    have log 2 (real n / (real v − real n)) ≤ log 2 (real n) + 1
     by (smt (verit, best) log-divide log-less-zero-cancel-iff n nat-less-real-le of-nat-0-less-iff
v)
    moreover have ∗∗: −1 < log 2 (real n / real (v − n))
      by(rule less-le-trans[where y=0])(use I c in ‹auto›)
    ultimately have x ≤ log 2 (real n) + 1 using v n
      by (simp add: x-def max-def field-simps log-divide)
     (smt (verit, best) ceiling-correct log-divide log-less-zero-cancel-iff nat-less-real-le
of-nat-0-less-iff)
    hence 2 powr x ≤ 2 powr . . . by(rule powr-mono) simp
    hence p ≤ 1 / 2 ^ x unfolding powr-add using n
     by(subst (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 (λs'. ?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** *that* **by**(*simp*)
show *?I* (*v*, *c*) **using** *that* **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* [*simp*]: *lossless-spmf fast-uniform* ⟷ *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 *n*: *True*
show *?thesis* **using** *spmf-fast-uniform-ub*
proof(*rule spmf-ub-tight*)
have $(\sum^+ x.$ *ennreal* (*if x* < *n then 1* / *n else 0*)) = $(\sum^+ x{\in}\{..{<}n\}.\ 1\ /\ n)$
by(*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*)) = . . . **.**
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*) (*λx. 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 λx. 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-definitions-spmf partial-function-definitions-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* = λ*x*. *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*
     by(*auto simp add*: *spmf-bind integral-spmf-of-set sum-divide-distrib if-distrib*[**where** *f*=λ*p*. *spmf p* - / -] *cong*: *if-cong intro*!: *sum.cong split*: *split-indicator-asm*)
    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*=λ*x*. *x* / -] *cong*: *if-cong*)
    also have . . . ≤ (*if x* ∈ *B then 1* / *card A else 0*) + *card* (*A* − *B*) / *card A* ∗ *spmf* (*spmf-of-set B*) *x*
      by(*intro add-left-mono mult-left-mono step.IH*[*THEN ord-spmf-eq-leD*]) *simp*
    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*) $x \leq \ldots$ **.**
qed *simp*
qed

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

[1] J. Hurd. A formal approach to probabilistic termination. In *TPHOLs 2002*, volume 2410 of *LNCS*, pages 230–245. Springer, 2002.

[2] J. Lumbroso. Optimal discrete uniform generation from coin flips, and applications. *CoRR*, abs/1304.1916, 2013.

[3] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems.* Monographs in Computer Science. Springer, 2005.