

Probabilistic while loop

Andreas Lochbihler

March 24, 2023

Abstract

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

Contents

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

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

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

1 Miscellaneous library additions

fun *map-option-set* :: ('a ⇒ 'b option set) ⇒ 'a option ⇒ 'b option set

where

map-option-set *f* *None* = {*None*}
| *map-option-set* *f* (*Some* *x*) = *f* *x*

lemma *None-in-map-option-set*:

None ∈ *map-option-set* *f* *x* ↔ *None* ∈ *Set.bind* (*set-option* *x*) *f* ∨ *x* = *None*
<proof>

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

<proof>

lemma *None-in-map-option-set-Some* [intro!]: *None* ∈ *f* *x* ⇒ *None* ∈ *map-option-set*

f (*Some* *x*)

<proof>

lemma *Some-in-map-option-set* [intro!]: *Some* *y* ∈ *f* *x* ⇒ *Some* *y* ∈ *map-option-set*

f (*Some* *x*)

<proof>

lemma *map-option-set-singleton* [simp]: *map-option-set* ($\lambda x. \{f\ x\}$) *y* = {*Option.bind*

y *f*}

<proof>

lemma *Some-eq-bind-conv*: *Some* *y* = *Option.bind* *x* *f* ↔ (∃ *z*. *x* = *Some* *z* ∧ *f*

z = *Some* *y*)

<proof>

lemma *map-option-set-bind*: *map-option-set* *f* (*Option.bind* *x* *g*) = *map-option-set*

(*map-option-set* *f* ∘ *g*) *x*

<proof>

lemma *Some-in-map-option-set-conv*: *Some* *y* ∈ *map-option-set* *f* *x* ↔ (∃ *z*. *x* =

Some *z* ∧ *Some* *y* ∈ *f* *z*)

<proof>

interpretation *rel-spmf-characterisation* <proof>

hide-fact (open) *rel-pmf-measureI*

lemma *Sup-conv-fun-lub*: *Sup* = *fun-lub* *Sup*

<proof>

lemma *le-conv-fun-ord*: (≤) = *fun-ord* (≤)

<proof>

lemmas *parallel-fixp-induct-2-1* = *parallel-fixp-induct-uc*[

of - - - case-prod - curry $\lambda x. x - \lambda x. x$,
where $P = \lambda f g. P (\text{curry } f) g$,
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - refl refl]
for P

lemma *monotone-Pair:*

$\llbracket \text{monotone ord orda } f; \text{monotone ord ordb } g \rrbracket$
 $\implies \text{monotone ord (rel-prod orda ordb) } (\lambda x. (f x, g x))$
 $\langle \text{proof} \rangle$

lemma *cont-Pair:*

$\llbracket \text{cont lub ord luba orda } f; \text{cont lub ord lubb ordb } g \rrbracket$
 $\implies \text{cont lub ord (prod-lub luba lubb) (rel-prod orda ordb) } (\lambda x. (f x, g x))$
 $\langle \text{proof} \rangle$

lemma *mcont-Pair:*

$\llbracket \text{mcont lub ord luba orda } f; \text{mcont lub ord lubb ordb } g \rrbracket$
 $\implies \text{mcont lub ord (prod-lub luba lubb) (rel-prod orda ordb) } (\lambda x. (f x, g x))$
 $\langle \text{proof} \rangle$

lemma *mono2mono-emeasure-spmf [THEN lfp.mono2mono]:*

shows *monotone-emeasure-spmf:*
 $\text{monotone (ord-spmf (=)) } (\leq) (\lambda p. \text{emeasure (measure-spmf } p))$
 $\langle \text{proof} \rangle$

lemma *cont-emeasure-spmf: cont lub-spmf (ord-spmf (=)) Sup (\leq) ($\lambda p. \text{emeasure (measure-spmf } p)$)*

$\langle \text{proof} \rangle$

lemma *mcont2mcont-emeasure-spmf [THEN lfp.mcont2mcont, cont-intro]:*

shows *mcont-emeasure-spmf: mcont lub-spmf (ord-spmf (=)) Sup (\leq) ($\lambda p. \text{emeasure (measure-spmf } p)$)*
 $\langle \text{proof} \rangle$

lemma *mcont2mcont-emeasure-spmf' [THEN lfp.mcont2mcont, cont-intro]:*

shows *mcont-emeasure-spmf': mcont lub-spmf (ord-spmf (=)) Sup (\leq) ($\lambda p. \text{emeasure (measure-spmf } p) A$)*
 $\langle \text{proof} \rangle$

lemma *mcont-bind-pmf [cont-intro]:*

assumes $g: \bigwedge y. \text{mcont luba orda lub-spmf (ord-spmf (=)) } (g y)$
shows $\text{mcont luba orda lub-spmf (ord-spmf (=)) } (\lambda x. \text{bind-pmf } p (\lambda y. g y x))$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *type-definition-Domainp:*

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

context includes *lifting-syntax begin*

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

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

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

lemma *bind-spmf-spmf-of-set:*
∧A. [finite A; A ≠ {}] ==> bind-spmf (spmof-of-set A) = bind-pmf (pmf-of-set A)
 ⟨*proof*⟩

lemma *set-pmf-bind-spmf: set-pmf (bind-spmf M f) = set-pmf M ≫= map-option-set (set-pmf ∘ f)*
 ⟨*proof*⟩

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

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

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

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

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

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

<proof>

lemma *spmf-ub-tight*:

assumes *ub*: $\bigwedge x. \text{spmf } p \ x \leq f \ x$

and *sum*: $(\int^+ x. f \ x \ \partial \text{count-space } UNIV) = \text{weight-spmf } p$

shows $\text{spmf } p \ x = f \ x$

<proof>

2 Probabilistic while loop

locale *loop-spmf* =

fixes *guard* :: 'a \Rightarrow bool

and *body* :: 'a \Rightarrow 'a *spmf*

begin

context notes *[[function-internals]] begin*

partial-function (*spmf*) *while* :: 'a \Rightarrow '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* ($\lambda \text{while. return-pmf None}$)

and $\bigwedge \text{while}'. P \ \text{while}' \Longrightarrow P \ (\lambda s. \text{if } \text{guard } s \text{ then } \text{body } s \gg \text{while}' \text{ else } \text{return-spmf } s)$

shows *P while*

<proof>

lemma *while-simps*:

guard *s* \Longrightarrow *while* *s* = *bind-spmf* (*body* *s*) *while*

\neg *guard* *s* \Longrightarrow *while* *s* = *return-spmf* *s*

<proof>

end

lemma *while-spmf-parametric* [*transfer-rule*]:

includes *lifting-syntax* **shows**

$((S \text{====>} (=)) \text{====>} (S \text{====>} \text{rel-spmf } S) \text{====>} S \text{====>} \text{rel-spmf } S)$

loop-spmf.while *loop-spmf.while*

<proof>

lemma *loop-spmf-while-cong*:

$\llbracket \text{guard} = \text{guard}'; \bigwedge s. \text{guard}' \ s \Longrightarrow \text{body } s = \text{body}' \ s \rrbracket$

$\Longrightarrow \text{loop-spmf.while } \text{guard} \ \text{body} = \text{loop-spmf.while } \text{guard}' \ \text{body}'$

<proof>

3 Rules for probabilistic termination

context *loop-spmf* begin

3.1 0/1 termination laws

lemma *termination-0-1-immediate*:

assumes $p: \bigwedge s. \text{guard } s \implies \text{spmf } (\text{map-spmf } \text{guard } (\text{body } s)) \text{ False} \geq p$

and $p\text{-pos}: 0 < p$

and $\text{lossless}: \bigwedge s. \text{guard } s \implies \text{lossless-spmf } (\text{body } s)$

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

<proof>

primrec *iter* :: $\text{nat} \Rightarrow 'a \Rightarrow 'a \text{ spmf}$

where

$\text{iter } 0 \ s = \text{return-spmf } s$

| $\text{iter } (\text{Suc } n) \ s = (\text{if } \text{guard } s \text{ then } \text{bind-spmf } (\text{body } s) (\text{iter } n) \text{ else } \text{return-spmf } s)$

lemma *iter-unguarded [simp]*: $\neg \text{guard } s \implies \text{iter } n \ s = \text{return-spmf } s$

<proof>

lemma *iter-bind-iter*: $\text{bind-spmf } (\text{iter } m \ s) (\text{iter } n) = \text{iter } (m + n) \ s$

<proof>

lemma *iter-Suc2*: $\text{iter } (\text{Suc } n) \ s = \text{bind-spmf } (\text{iter } n \ s) (\lambda s. \text{if } \text{guard } s \text{ then } \text{body } s \text{ else } \text{return-spmf } s)$

<proof>

lemma *lossless-iter*: $(\bigwedge s. \text{guard } s \implies \text{lossless-spmf } (\text{body } s)) \implies \text{lossless-spmf } (\text{iter } n \ s)$

<proof>

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 $?lhs \leq ?rhs$)

<proof>

lemma *weight-while-conv-iter*:

$\text{weight-spmf } (\text{while } s) = (\text{SUP } n. \text{measure } (\text{measure-spmf } (\text{iter } n \ s)) \ \{s. \neg \text{guard } s\})$

(is $?lhs = ?rhs$)

<proof>

lemma *termination-0-1*:

assumes $p: \bigwedge s. \text{guard } s \implies p \leq \text{weight-spmf } (\text{while } s)$

and $p\text{-pos}: 0 < p$

and $\text{lossless}: \bigwedge s. \text{guard } s \implies \text{lossless-spmf } (\text{body } s)$

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

<proof>

end

lemma *termination-0-1-immediate-invar*:

fixes $I :: 's \Rightarrow \text{bool}$
assumes $p: \bigwedge s. \llbracket \text{guard } s; I \ s \rrbracket \Longrightarrow \text{spmf } (\text{map-spmf } \text{guard } (\text{body } s)) \ \text{False} \geq p$
and $p\text{-pos}: 0 < p$
and $\text{lossless}: \bigwedge s. \llbracket \text{guard } s; I \ s \rrbracket \Longrightarrow \text{lossless-spmf } (\text{body } s)$
and $\text{invar}: \bigwedge s \ s'. \llbracket s' \in \text{set-spmf } (\text{body } s); I \ s; \text{guard } s \rrbracket \Longrightarrow I \ s'$
and $I: I \ s$
shows $\text{lossless-spmf } (\text{loop-spmf}.\text{while } \text{guard } \text{body } s)$
including *lifting-syntax*
<proof>

lemma *termination-0-1-invar*:

fixes $I :: 's \Rightarrow \text{bool}$
assumes $p: \bigwedge s. \llbracket \text{guard } s; I \ s \rrbracket \Longrightarrow p \leq \text{weight-spmf } (\text{loop-spmf}.\text{while } \text{guard } \text{body } s)$
and $p\text{-pos}: 0 < p$
and $\text{lossless}: \bigwedge s. \llbracket \text{guard } s; I \ s \rrbracket \Longrightarrow \text{lossless-spmf } (\text{body } s)$
and $\text{invar}: \bigwedge s \ s'. \llbracket s' \in \text{set-spmf } (\text{body } s); I \ s; \text{guard } s \rrbracket \Longrightarrow I \ s'$
and $I: I \ s$
shows $\text{lossless-spmf } (\text{loop-spmf}.\text{while } \text{guard } \text{body } s)$
including *lifting-syntax*
<proof>

3.2 Variant rule

context *loop-spmf* **begin**

lemma *termination-variant*:

fixes $\text{bound} :: \text{nat}$
assumes $\text{bound}: \bigwedge s. \text{guard } s \Longrightarrow f \ s \leq \text{bound}$
and $\text{step}: \bigwedge s. \text{guard } s \Longrightarrow p \leq \text{spmf } (\text{map-spmf } (\lambda s'. f \ s' < f \ s) (\text{body } s)) \ \text{True}$
and $p\text{-pos}: 0 < p$
and $\text{lossless}: \bigwedge s. \text{guard } s \Longrightarrow \text{lossless-spmf } (\text{body } s)$
shows $\text{lossless-spmf } (\text{while } s)$
<proof>

end

lemma *termination-variant-invar*:

fixes $\text{bound} :: \text{nat}$ **and** $I :: 's \Rightarrow \text{bool}$
assumes $\text{bound}: \bigwedge s. \llbracket \text{guard } s; I \ s \rrbracket \Longrightarrow f \ s \leq \text{bound}$
and $\text{step}: \bigwedge s. \llbracket \text{guard } s; I \ s \rrbracket \Longrightarrow p \leq \text{spmf } (\text{map-spmf } (\lambda s'. f \ s' < f \ s) (\text{body } s)) \ \text{True}$
and $p\text{-pos}: 0 < p$
and $\text{lossless}: \bigwedge s. \llbracket \text{guard } s; I \ s \rrbracket \Longrightarrow \text{lossless-spmf } (\text{body } s)$
and $\text{invar}: \bigwedge s \ s'. \llbracket s' \in \text{set-spmf } (\text{body } s); I \ s; \text{guard } s \rrbracket \Longrightarrow I \ s'$

```

and  $I$ :  $I$   $s$ 
shows lossless-spmf (loop-spmf.while guard body s)
including lifting-syntax
⟨proof⟩

end

```

4 Distributions built from coin flips

4.1 The Bernoulli distribution

```

theory Bernoulli imports HOL-Probability.Probability begin

```

```

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

```

```

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

```

```

lemma one-plus-ennreal:  $0 \leq x \implies 1 + \text{ennreal } x = \text{ennreal } (1 + x)$ 
  ⟨proof⟩

```

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

```

partial-function (spmf) bernoulli ::  $\text{real} \Rightarrow \text{bool } \text{spmf}$  where
  bernoulli  $p = \text{do}$  {
     $b \leftarrow \text{coin-spmf}$ ;
    if  $b$  then return-spmf ( $p \geq 1 / 2$ )
    else if  $p < 1 / 2$  then bernoulli ( $2 * p$ )
    else bernoulli ( $2 * p - 1$ )
  }

```

```

lemma pmf-bernoulli-None:  $\text{pmf } (\text{bernoulli } p) \text{ None} = 0$ 
  ⟨proof⟩

```

```

lemma lossless-bernoulli [simp]: lossless-spmf (bernoulli p)
  ⟨proof⟩

```

```

lemma [simp]: assumes  $0 \leq p \leq 1$ 
  shows bernoulli-True:  $\text{spmf } (\text{bernoulli } p) \text{ True} = p$  (is ?True)
  and bernoulli-False:  $\text{spmf } (\text{bernoulli } p) \text{ False} = 1 - p$  (is ?False)
  ⟨proof⟩

```

```

lemma bernoulli-neg [simp]:
  assumes  $p \leq 0$ 
  shows bernoulli p = return-spmf False

```


<proof>

lemma *bernoulli-pos* [*simp*]:
 assumes $1 \leq p$
 shows *bernoulli* $p = \text{return-spmf True}$
<proof>

context begin interpretation *pmf-as-function* *<proof>*

lemma *bernoulli-eq-bernoulli-pmf*:
 bernoulli $p = \text{spmf-of-pmf (bernoulli-pmf } p)$
<proof>
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* \Rightarrow *nat* *spmf* **where**
 geometric-spmf $p = \text{do } \{$
 $b \leftarrow \text{bernoulli } p;$
 if b *then* $\text{return-spmf } 0$ *else* $\text{map-spmf } ((+) 1) (\text{geometric-spmf } p)$
 }
end

lemma *geometric-spmf-fixp-induct* [*case-names adm bottom step*]:

assumes *spmf.admissible* P
 and $P (\lambda \text{geometric-spmf}. \text{return-pmf None})$
 and $\bigwedge \text{geometric-spmf}'. P \text{geometric-spmf}' \Longrightarrow P (\lambda p. \text{bernoulli } p \ggg (\lambda b. \text{if } b$
then $\text{return-spmf } 0$ *else* $\text{map-spmf } ((+) 1) (\text{geometric-spmf}' p))$
 shows $P \text{geometric-spmf}$
<proof>

lemma *spmf-geometric-nonpos*: $p \leq 0 \Longrightarrow \text{geometric-spmf } p = \text{return-pmf None}$
<proof>

lemma *spmf-geometric-ge-1*: $1 \leq p \Longrightarrow \text{geometric-spmf } p = \text{return-spmf } 0$
<proof>

context

fixes $p :: \text{real}$

```

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

```

```

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

```

```

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

```

```

lemma lossless-geometric [simp]: lossless-spmf (geometric-spmf p) ↔ p > 0
  ⟨proof⟩

```

end

```

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

```

end

4.3 Arbitrary uniform distributions

```

theory Fast-Dice-Roll imports
  Bernoulli
  While-SPMF
begin

```

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

```

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

```

```

lemma sample-bits-fusion2:
  fixes v :: nat
  assumes 0 < v
  shows
    bind-pmf (pmf-of-set UNIV) (λb. bind-pmf (pmf-of-set {..v}) (λc. f (c + v *

```

```

(if b then 1 else 0))) =
  bind-pmf (pmf-of-set {.. $2 * v$ }) f
  (is ?lhs = ?rhs)
⟨proof⟩

```

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* :: $nat \Rightarrow nat \Rightarrow nat$ *spmf*
where

```

fast-dice-roll v c =
  (if v ≥ n then if c < n then return-spmf c else fast-dice-roll (v - n) (c - n)
   else do {
     b ← 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 ( $\lambda$ fast-dice-roll. P (curry fast-dice-roll))
and P ( $\lambda$ v c. return-pmf None)
and  $\bigwedge$ fdr. P fdr  $\implies$  P ( $\lambda$ v c. if v ≥ n then if c < n then return-spmf c else fdr
(v - n) (c - n)
  else bind-spmf coin-spmf ( $\lambda$ b. fdr (2 * v) (2 * c + (if b then 1 else 0))))
shows P fast-dice-roll
⟨proof⟩

```

definition *fast-uniform* :: nat *spmf*

where *fast-uniform* = *fast-dice-roll* 1 0

lemma *spmf-fast-dice-roll-ub*:

```

assumes  $0 < v$ 
shows 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⟩

```

lemma *spmf-fast-uniform-ub*:

```

spmf fast-uniform  $x \leq$  (if  $x < n$  then 1 / n else 0)
⟨proof⟩

```

lemma *fast-dice-roll-0* [*simp*]: *fast-dice-roll* 0 c = *return-pmf None*

⟨proof⟩

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

definition $fdr\text{-step} :: nat \Rightarrow nat \Rightarrow (nat \times nat) \text{ spmf}$

where

$fdr\text{-step } v \ c =$
(if $v = 0$ *then* $\text{return-pmf } None$
else let $x = 2 \wedge (nat \lceil \log 2 (max \ 1 \ n) - \log 2 \ v \rceil)$ *in*
 $\text{map-spmf } (\lambda bs. (x * v, x * c + bs)) (\text{spmf-of-set } \{..<x\})$)

lemma $fdr\text{-step-unfold}$:

$fdr\text{-step } v \ c =$
(if $v = 0$ *then* $\text{return-pmf } None$
else if $n \leq v$ *then* $\text{return-spmf } (v, c)$
else do {
 $b \leftarrow \text{coin-spmf}$;
 $fdr\text{-step } (2 * v) \ (2 * c + (\text{if } b \text{ then } 1 \text{ else } 0))$ }
(is ?lhs = ?rhs is - = (if - then - else ?else))
 $\langle \text{proof} \rangle$

lemma $fdr\text{-step-induct}$ [*case-names* $fdr\text{-step}$]:

$(\bigwedge v \ c. (\bigwedge b. \llbracket v \neq 0; v < n \rrbracket \implies P \ (2 * v) \ (2 * c + (\text{if } b \text{ then } 1 \text{ else } 0))) \implies P \ v \ c)$
 $\implies P \ v \ c$
 $\langle \text{proof} \rangle$

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

where

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma $\text{fast-dice-roll-alt-conv-while}$:

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

lemma $\text{lossless-fast-dice-roll}$:

assumes $c < v \ v \leq n$
shows $\text{lossless-spmf } (\text{fast-dice-roll } v \ c)$
 $\langle \text{proof} \rangle$

lemma fast-dice-roll-n0 :

```

assumes  $n = 0$ 
shows  $\text{fast-dice-roll } v \ c = \text{return-pmf } \text{None}$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{lossless-fast-uniform [simp]: lossless-spmf fast-uniform} \longleftrightarrow n > 0$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{spm-f-fast-uniform: spmf fast-uniform } x = (\text{if } x < n \text{ then } 1 / n \text{ else } 0)$ 
 $\langle \text{proof} \rangle$ 

end

lemma  $\text{fast-uniform-conv-uniform: fast-uniform } n = \text{spm-f-of-set } \{..<n\}$ 
 $\langle \text{proof} \rangle$ 

end

theory Resampling imports
  While-SPMF
begin

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

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

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

end

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

context
  fixes  $A :: 'a \ \text{set}$ 
  and  $B :: 'a \ \text{set}$ 
begin

interpretation  $\text{loop-spmf } \lambda x. \ x \notin B \ \lambda -. \ \text{spm-f-of-set } A \ \langle \text{proof} \rangle$ 

lemma  $\text{resample-conv-while: resample } A \ B = \text{bind-spmf } (\text{spm-f-of-set } A) \ \text{while}$ 
 $\langle \text{proof} \rangle$ 

context
  assumes  $A: \text{finite } A$ 

```

```

and  $B: B \subseteq A \ B \neq \{\}$ 
begin

private lemma A-nonempty:  $A \neq \{\}$ 
   $\langle proof \rangle$  lemma B-finite: finite B
   $\langle proof \rangle$ 

lemma lossless-resample: lossless-spmf (resample A B)
   $\langle proof \rangle$ 

lemma resample-le-sample:
  ord-spmf (=) (resample A B) (spmf-of-set B)
   $\langle proof \rangle$ 

lemma resample-eq-sample: resample A B = spmf-of-set B
   $\langle proof \rangle$ 

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. Lombroso. 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.