Probabilistic while loop

Andreas Lochbihler

March 19, 2025

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

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* :: $('a \Rightarrow 'b\ option\ set) \Rightarrow 'a\ option \Rightarrow 'b\ option\ set$
where
  $map\text{-}option\text{-}set\ f\ None = \{None\}$
| $map\text{-}option\text{-}set\ f\ (Some\ x) = f\ x$

lemma *None-in-map-option-set*:
  $None \in map\text{-}option\text{-}set\ f\ x \longleftrightarrow None \in Set.bind\ (set\text{-}option\ x)\ f \lor x = None$
⟨*proof*⟩

lemma *None-in-map-option-set-None* [*intro*!]: $None \in map\text{-}option\text{-}set\ f\ None$
⟨*proof*⟩

lemma *None-in-map-option-set-Some* [*intro*!]: $None \in f\ x \Longrightarrow None \in map\text{-}option\text{-}set\ f\ (Some\ x)$
⟨*proof*⟩

lemma *Some-in-map-option-set* [*intro*!]: $Some\ y \in f\ x \Longrightarrow Some\ y \in map\text{-}option\text{-}set\ f\ (Some\ x)$
⟨*proof*⟩

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

lemma *Some-eq-bind-conv*: $Some\ y = Option.bind\ x\ f \longleftrightarrow (\exists z.\ x = Some\ z \land f\ z = Some\ y)$
⟨*proof*⟩

lemma *map-option-set-bind*: $map\text{-}option\text{-}set\ f\ (Option.bind\ x\ g) = map\text{-}option\text{-}set\ (map\text{-}option\text{-}set\ f \circ g)\ x$
⟨*proof*⟩

lemma *Some-in-map-option-set-conv*: $Some\ y \in map\text{-}option\text{-}set\ f\ x \longleftrightarrow (\exists z.\ x = Some\ z \land Some\ y \in f\ z)$
⟨*proof*⟩

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

lemma *Sup-conv-fun-lub*: $Sup = fun\text{-}lub\ Sup$
  ⟨*proof*⟩

lemma *le-conv-fun-ord*: $(\le) = fun\text{-}ord\ (\le)$
  ⟨*proof*⟩

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

of - - - - case-prod - curry $\lambda x.\ x$ - $\lambda x.\ x$,
where $P = \lambda f\ g.\ P\ (curry\ f)\ g$,
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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *mono2mono-emeasure-spmf* [*THEN lfp.mono2mono*]:
shows *monotone-emeasure-spmf*:
monotone (*ord-spmf* (=)) ($\leq$) ($\lambda p.$ *emeasure* (*measure-spmf p*))
⟨*proof*⟩

lemma *cont-emeasure-spmf*: *cont lub-spmf* (*ord-spmf* (=)) *Sup* ($\leq$) ($\lambda p.$ *emeasure* (*measure-spmf p*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *ennreal-less-top-iff*: $x < \top \longleftrightarrow x \neq (\top :: ennreal)$
⟨*proof*⟩

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*⟩

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*:
$\bigwedge A.$ ⟦ *finite A*; $A \neq \{\}$ ⟧ $\Longrightarrow$ *bind-spmf* (*spmf-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 (*spmf-of-set A*) = (*if finite A* ∧ $A \neq \{\}$ *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* $n \le 1 \longleftrightarrow n \le 1$
⟨*proof*⟩

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

lemma *lessThan-subset-Collect*: $\{..<x\} \subseteq$ *Collect P* $\longleftrightarrow (\forall y<x.\ P\ y)$

⟨*proof*⟩

lemma *spmf-ub-tight*:
assumes *ub*: $\bigwedge x.\ spmf\ p\ x \le f\ x$
and *sum*: $(\int^{+} x.\ f\ x\ \partial count\text{-}space\ UNIV) = weight\text{-}spmf\ p$
shows $spmf\ p\ x = f\ x$
⟨*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* (λ*while. return-pmf None*)
and $\bigwedge while'.\ P\ while' \Longrightarrow P\ (\lambda s.\ if\ guard\ s\ then\ body\ s \gg= while'\ else\ return\text{-}spmf\ s)$
shows *P while*
⟨*proof*⟩

lemma *while-simps*:
$guard\ s \Longrightarrow while\ s = bind\text{-}spmf\ (body\ s)\ while$
$\neg\ guard\ s \Longrightarrow while\ s = return\text{-}spmf\ s$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *loop-spmf-while-cong*:
$[\![\ guard = guard';\ \bigwedge s.\ guard'\ s \Longrightarrow body\ s = body'\ s\ ]\!]$
$\Longrightarrow loop\text{-}spmf.while\ guard\ body = loop\text{-}spmf.while\ guard'\ body'$
⟨*proof*⟩

3 Rules for probabilistic termination

context *loop-spmf* **begin**

3.1 0/1 termination laws

lemma *termination-0-1-immediate*:
assumes *p*: $\bigwedge s.\ guard\ s \Longrightarrow spmf\ (map\text{-}spmf\ guard\ (body\ s))\ False \geq p$
and *p-pos*: $0 < p$
and *lossless*: $\bigwedge s.\ guard\ s \Longrightarrow lossless\text{-}spmf\ (body\ s)$
shows *lossless-spmf (while s)*
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *termination-0-1*:
assumes *p*: $\bigwedge s.\ guard\ s \Longrightarrow p \leq weight\text{-}spmf\ (while\ s)$
and *p-pos*: $0 < p$
and *lossless*: $\bigwedge s.\ guard\ s \Longrightarrow lossless\text{-}spmf\ (body\ s)$
shows *lossless-spmf (while s)*
⟨*proof*⟩

end

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

lemma *termination-0-1-invar*:
fixes $I :: 's \Rightarrow bool$
assumes *p*: $\bigwedge s.\ [\![\ guard\ s;\ I\ s\ ]\!] \Longrightarrow p \leq weight\text{-}spmf\ (loop\text{-}spmf.while\ guard\ body\ s)$
and *p-pos*: $0 < p$
and *lossless*: $\bigwedge s.\ [\![\ guard\ s;\ I\ s\ ]\!] \Longrightarrow lossless\text{-}spmf\ (body\ s)$
and *invar*: $\bigwedge s\ s'.\ [\![\ s' \in set\text{-}spmf\ (body\ s);\ I\ s;\ guard\ s\ ]\!] \Longrightarrow I\ s'$
and *I*: $I\ s$
shows $lossless\text{-}spmf\ (loop\text{-}spmf.while\ guard\ body\ s)$
including *lifting-syntax*
⟨*proof*⟩

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*⟩

end

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

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* < (*numeral n* :: - :: {*canonically-ordered-monoid-add*, *semiring-char-0*})
⟨*proof*⟩

lemma *ennreal-mult-numeral*: *ennreal x* ∗ *numeral n* = *ennreal* (*x* ∗ *numeral n*)
⟨*proof*⟩

lemma *one-plus-ennreal*: *0* ≤ *x* ⟹ *1* + *ennreal x* = *ennreal* (*1* + *x*)
⟨*proof*⟩

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

```
partial-function (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*⟩

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

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*⟩

lemma *bernoulli-neg* [*simp*]:
assumes *p* ≤ *0*
shows *bernoulli p* = *return-spmf False*

⟨*proof*⟩

lemma *bernoulli-pos* [*simp*]:
  assumes $1 \leq p$
  shows *bernoulli p = return-spmf True*
⟨*proof*⟩

context begin interpretation *pmf-as-function* ⟨*proof*⟩
lemma *bernoulli-eq-bernoulli-pmf*:
  bernoulli p = 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* ⇒ *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 $\bigwedge$*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*
  ⟨*proof*⟩

lemma *spmf-geometric-nonpos*: $p \leq 0$ ⟹ *geometric-spmf p = return-pmf None*
  ⟨*proof*⟩

lemma *spmf-geometric-ge-1*: $1 \leq p$ ⟹ *geometric-spmf p = return-spmf 0*
  ⟨*proof*⟩

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*)
⟨*proof*⟩

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

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

end

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

end

4.3 Arbitrary uniform distributions

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

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

lemma *sample-bits-fusion*:
fixes *v* :: *nat*
assumes *0* < *v*
shows
bind-pmf (*pmf-of-set* {..<*v*}) (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*⟩

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*⟩

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* ≥ *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* (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* ≥ *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*
⟨*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* ≤ (*if x* < *n then 1* / *n else 0*)
 (**is** *?lhs* ≤ *?rhs*)
⟨*proof*⟩

lemma *spmf-fast-uniform-ub*:
 spmf fast-uniform x ≤ (*if x* < *n then 1* / *n else 0*)
⟨*proof*⟩

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

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

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

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

lemma *fdr-step-induct* [*case-names fdr-step*]:
  $(\bigwedge v\ c.\ (\bigwedge b.\ [\![v \neq 0;\ v < n]\!] \Longrightarrow P\ (2 * v)\ (2 * c + (\textit{if}\ b\ \textit{then}\ 1\ \textit{else}\ 0))) \Longrightarrow P$
$v\ c)$
  $\Longrightarrow P\ v\ c$
⟨*proof*⟩

partial-function (*spmf*) *fdr-alt* :: $nat \Rightarrow nat \Rightarrow nat\ spmf$
where
  $\textit{fdr-alt}\ v\ c = \textit{do}\ \{$
    $(v',\ c') \leftarrow \textit{fdr-step}\ v\ c;$
    $\textit{if}\ c' < n\ \textit{then return-spmf}\ c'\ \textit{else fdr-alt}\ (v' - n)\ (c' - n)\ \}$

lemma *fast-dice-roll-alt*: *fdr-alt* = *fast-dice-roll*
⟨*proof*⟩

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

lemma *fast-dice-roll-alt-conv-while*:
  $\textit{fdr-alt}\ v\ c =$
  $\textit{map-spmf snd}\ (\textit{bind-spmf}\ (\textit{fdr-step}\ v\ c)\ (\textit{loop-spmf.while}\ (\lambda(v,\ c).\ n \leq c)\ (\lambda(v,$
$c).\ \textit{fdr-step}\ (v - n)\ (c - n))))$
⟨*proof*⟩

lemma *lossless-fast-dice-roll*:
  assumes $c < v\ v \leq n$
  shows *lossless-spmf* (*fast-dice-roll* $v\ c$)
⟨*proof*⟩

lemma *fast-dice-roll-n0*:

assumes *n* = *0*
shows *fast-dice-roll v c* = *return-pmf None*
⟨*proof*⟩

lemma *lossless-fast-uniform* [*simp*]: *lossless-spmf fast-uniform* ⟷ *n* > *0*
⟨*proof*⟩

lemma *spmf-fast-uniform*: *spmf fast-uniform x* = (*if x* < *n then 1* / *n else 0*)
⟨*proof*⟩

end

lemma *fast-uniform-conv-uniform*: *fast-uniform n* = *spmf-of-set* {..<*n*}
⟨*proof*⟩

end

theory *Resampling* **imports**
While-SPMF
begin

lemma *ord-spmf-lossless*:
assumes *ord-spmf* (=) *p q lossless-spmf p*
shows *p* = *q*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *resample-conv-while*: *resample A B* = *bind-spmf* (*spmf-of-set A*) *while*
⟨*proof*⟩

context
assumes *A*: *finite A*

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

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

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

lemma *resample-le-sample*:
ord-spmf (=) (*resample A B*) (*spmf-of-set B*)
⟨*proof*⟩

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

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.