Executable Randomized Algorithms

Emin Karayel and Manuel Eberl

September 13, 2023

Abstract

In Isabelle, randomized algorithms are usually represented using probability mass functions (PMFs), with which it is possible to verify their correctness, particularly properties about the distribution of their result. However, that approach does not provide a way to generate executable code for such algorithms. In this entry, we introduce a new monad for randomized algorithms, for which it is possible to generate code and simultaneously reason about the correctness of randomized algorithms. The latter works by a Scott-continuous monad morphism between the newly introduced random monad and PMFs. On the other hand, when supplied with an external source of random coin flips, the randomized algorithms can be executed.

Contents

1 Introduction **1**

2 τ-Additivity **2**

3 Coin Flip Space **3**

4 Randomized Algorithms (Internal Representation) **11**

5 Randomized Algorithms **18**
5.1 Almost surely terminating randomized algorithms 22

6 Tracking Randomized Algorithms **23**

7 Tracking SPMFs **26**

8 Dice Roll **30**

9 A Pseudo-random Number Generator **32**

10 Basic Randomized Algorithms **33**

1 Introduction

In Isabelle, randomized algorithms are usually represented using probability mass functions (PMFs). (These are distributions on the discrete σ-algebra, i.e., pure point measures.) That representation allows the verification of the correctness of randomized algorithms, for example the expected value of their result, moments or other probabilistic properties. However, it is not directly possible to execute a randomized algorithm modelled as a PMF.

In this work, we introduce a representation of randomized algorithms as a parser monad over an external arbitrary source of random coin flips, modelled using a lazy infinite stream of booleans. Using for example a PRG or some other mechanism, like a hardware RNG to supply the coin flips, the generated code for the monad can be executed.

$$
\begin{tikzcd}
\text{'a random-alg} \arrow[rr, "\text{spmf-of-ra}"] \arrow[d, "\text{track-random-bits}"] \arrow[drr, "\text{tspmf-of-ra}"] & & \text{'a spmf} \\
\text{('a} \times \text{nat) random-alg} \arrow[rr, "\text{spmf-of-ra}"] & & \text{'a tspmf} \arrow[u, "\text{map-spmf fst}"']
\end{tikzcd}
$$

Figure 1: Scott-continuous monad morphisms verified in this work.

Then we introduce a monad morphism between such algorithms and the corresponding PMF, i.e., the PMF representing the distribution of the randomized algorithm under the idealized assumption that the coin flips are independent and unbiased, such that correctness properties can still be verified.

In the presence of loops and possible likelihood of non-termination, the resulting PMF maybe an SPMF (a finite measure space with total measure less than 1). (Internally these are just PMFs over the `option` type, where `None` represents non-termination.) If a randomized algorithm terminates almost surely, the weight of the SPMF will be 1.

With this framework, it is also possible to reason about the number of coin-flips consumed by the algorithm. The latter is itself a distribution, where for example the average count of used coin-flips is represented as the expectation of that distribution. To facilitate the latter, we introduce a second monad morphism, between randomized algorithm and a resource monad on top of the SPMF monad. Indeed the latter describes the joint-distribution of the result of a randomized algorithm and the number of used coin flips. (It is easy to construct examples where the individual marginal distributions are not enough, for example when the number of coin-flips used in intermediate steps of the algorithm depend on parameters.)

Figure 1 summarizes the Scott-continuous monad morphisms verified in this work. In particular:

- *spmf-of-ra*: Morphism between randomized algorithms and the distribution of their result. (Section 5)
- *track-coin-usage*: Morphism between randomized algorithms and randomized algorithms that track their coin flip usage. The result is still executable. (Section 6)
- *tspmf-of-ra*: Morpshism between randomized algorithms and the joint-distribution of their result and coin-flip usage. (Section 7)

In addition to that we also introduce the monad morphism pmf-of-ra which returns a PMF instead of an SPMF. It is defined for algorithms that terminate unconditionally or almost surely.

Section 10 contains some examples showing how to use this library, as well as randomized algorithms for standard probability distributions.

Section 8 contains an extended example with verification of correctness, as well as bounds on the the average coin-flip usage for a dice roll algorithm. (It is a specialization of an algorithm presented by Hao and Hoshi [4].)

2 τ-Additivity

theory *Tau-Additivity*
  imports *HOL−Analysis.Regularity*
begin

In this section we show τ-additivity for measures, that are compatible with a second-countable topology. This will be essential for the verification of the Scott-continuity of the monad morphisms. To understand the property, let us recall that for general countable chains of measurable sets, it is possible to deduce that the supremum of the measures of the sets is equal to the measure of the union of the family:

$$\mu\left(\bigcup \mathcal{X}\right) = \sup_{X \in \mathcal{X}} \mu(X)$$

this is shown in *SUP-emeasure-incseq*.

It is possible to generalize that to arbitrary chains [1] of open sets for some measures without the restriction of countability, such measures are called τ-additive [3].

In the following this property is derived for measures that are at least borel (i.e. every open set is measurable) in a complete second-countable topology. The result is an immediate consequence of inner-regularity. The latter is already verified in *HOL−Analysis.Regularity*.

definition *op-stable op F* = ($\forall$ *x y*. *x* $\in$ *F* $\wedge$ *y* $\in$ *F* $\longrightarrow$ *op x y* $\in$ *F*)

lemma *op-stableD*:
  assumes *op-stable op F*
  assumes *x* $\in$ *F* *y* $\in$ *F*
  shows *op x y* $\in$ *F*
  ⟨*proof*⟩

lemma *tau-additivity-aux*:
  fixes *M*::'*a*::{*second-countable-topology*, *complete-space*} *measure*
  assumes *sb*: *sets M* = *sets borel*
  assumes *fin*: *emeasure M* (*space M*) $\neq \infty$
  assumes *of*: $\bigwedge$*a*. *a* $\in$ *A* $\Longrightarrow$ *open a*
  assumes *ud*: *op-stable* ($\cup$) *A*
  shows *emeasure M* ($\bigcup$ *A*) = (*SUP a* $\in$ *A*. *emeasure M a*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *chain-imp-union-stable*:
  assumes *Complete-Partial-Order.chain* ($\subseteq$) *F*
  shows *op-stable* ($\cup$) *F*
⟨*proof*⟩

theorem *tau-additivity*:
  fixes *M* :: '*a*::{*second-countable-topology*, *complete-space*} *measure*
  assumes *sb*: $\bigwedge$*x*. *open x* $\Longrightarrow$ *x* $\in$ *sets M*
  assumes *fin*: *emeasure M* (*space M*) $\neq \infty$
  assumes *of*: $\bigwedge$*a*. *a* $\in$ *A* $\Longrightarrow$ *open a*
  assumes *ud*: *op-stable* ($\cup$) *A*
  shows *emeasure M* ($\bigcup$ *A*) = (*SUP a* $\in$ *A*. *emeasure M a*) (**is** *?L* = *?R*)
⟨*proof*⟩

end

3 Coin Flip Space

In this section, we introduce the coin flip space, an infinite lazy stream of booleans and introduce a probability measure and topology for the space.

theory *Coin-Space*
  imports
    HOL−Probability.Probability
    HOL−Library.Code-Lazy
begin

[1] More generally families closed under pairwise unions.

lemma *stream-eq-iff*:
  assumes $\bigwedge i.\ x\ !!\ i = y\ !!\ i$
  shows $x = y$
$\langle proof \rangle$

Notation for the discrete σ-algebra:

abbreviation *discrete-sigma-algebra*
  where *discrete-sigma-algebra* $\equiv$ *count-space UNIV*

bundle *discrete-sigma-algebra-notation*
begin
  notation *discrete-sigma-algebra* $(\mathcal{D})$
end

bundle *no-discrete-sigma-algebra-notation*
begin
  no-notation *discrete-sigma-algebra* $(\mathcal{D})$
end

unbundle *discrete-sigma-algebra-notation*

lemma *map-prod-measurable*[*measurable*]:
  assumes $f \in M \rightarrow_M M'$
  assumes $g \in N \rightarrow_M N'$
  shows *map-prod f g* $\in M \bigotimes_M N \rightarrow_M M' \bigotimes_M N'$
  $\langle proof \rangle$

lemma *measurable-sigma-sets-with-exception*:
  fixes $f :: 'a \Rightarrow 'b :: countable$
  assumes $\bigwedge x.\ x \neq d \Longrightarrow f -` \{x\} \cap space\ M \in sets\ M$
  shows $f \in M \rightarrow_M$ *count-space UNIV*
$\langle proof \rangle$

lemma *restr-empty-eq*: *restrict-space* M $\{\}$ = *restrict-space* N $\{\}$
  $\langle proof \rangle$

lemma (**in** *prob-space*) *distr-stream-space-snth* [*simp*]:
  assumes *sets* M = *sets* N
  shows *distr* (*stream-space* M) N ($\lambda xs.$ *snth xs n*) = M
$\langle proof \rangle$

lemma (**in** *prob-space*) *distr-stream-space-shd* [*simp*]:
  assumes *sets* M = *sets* N
  shows *distr* (*stream-space* M) N *shd* = M
  $\langle proof \rangle$

lemma *shift-measurable*:
  assumes *set* $x \subseteq$ *space* M
  shows ($\lambda bs.\ x$ @− bs) $\in$ *stream-space* $M \rightarrow_M$ *stream-space* M
$\langle proof \rangle$

lemma (**in** *sigma-finite-measure*) *restrict-space-pair-lift*:
  assumes $A' \in sets\ A$
  shows *restrict-space* A $A' \bigotimes_M M$ = *restrict-space* $(A \bigotimes_M M)$ $(A' \times space\ M)$ (**is** *?L* = *?R*)
$\langle proof \rangle$

lemma *to-stream-comb-seq-eq*:
  to-stream (*comb-seq n x y*) = *stake n* (*to-stream x*) @− *to-stream y*

⟨*proof*⟩

lemma *to-stream-snth*: *to-stream* ((!!) *x*) = *x*
⟨*proof*⟩

lemma *snth-to-stream*: *snth* (*to-stream x*) = *x*
⟨*proof*⟩

lemma (**in** *prob-space*) *branch-stream-space*:
$(\lambda(x,\ y).\ \mathit{stake}\ n\ x\ @\!-\ y) \in \mathit{stream\text{-}space}\ M \bigotimes_M \mathit{stream\text{-}space}\ M \rightarrow_M \mathit{stream\text{-}space}\ M$
$\mathit{distr}\ (\mathit{stream\text{-}space}\ M \bigotimes_M \mathit{stream\text{-}space}\ M)\ (\mathit{stream\text{-}space}\ M)\ (\lambda(x,y).\ \mathit{stake}\ n\ x@\!-\!y)$
$= \mathit{stream\text{-}space}\ M$ (**is** *?L* = *?R*)
⟨*proof*⟩

The type for the coin flip space is isomorphic to *bool stream*. Nevertheless, we introduce it as a separate type to be able to introduce a topology and mark it as a lazy type for code-generation:

codatatype *coin-stream* = *Coin* (*chd*:*bool*) (*ctl*:*coin-stream*)

code-lazy-type *coin-stream*

primcorec *from-coins* :: *coin-stream* ⇒ *bool stream* **where**
from-coins coins = *chd coins* ## (*from-coins* (*ctl coins*))

primcorec *to-coins* :: *bool stream* ⇒ *coin-stream* **where**
to-coins str = *Coin* (*shd str*) (*to-coins* (*stl str*))

lemma *to-from-coins*: *to-coins* (*from-coins x*) = *x*
⟨*proof*⟩

lemma *from-to-coins*: *from-coins* (*to-coins x*) = *x*
⟨*proof*⟩

lemma *bij-to-coins*: *bij to-coins*
⟨*proof*⟩

lemma *bij-from-coins*: *bij from-coins*
⟨*proof*⟩

definition *cshift* **where** *cshift x y* = *to-coins* (*x* @− *from-coins y*)
definition *cnth* **where** *cnth x n* = *from-coins x* !! *n*
definition *ctake* **where** *ctake n x* = *stake n* (*from-coins x*)
definition *cdrop* **where** *cdrop n x* = *to-coins* (*sdrop n* (*from-coins x*))
definition *rel-coins* **where** *rel-coins x y* = (*to-coins x* = *y*)
definition *cprefix* **where** *cprefix x y* ⟷ *ctake* (*length x*) *y* = *x*
definition *cconst* **where** *cconst x* = *to-coins* (*sconst x*)

context
includes *lifting-syntax*
begin

lemma *bi-unique-rel-coins* [*transfer-rule*]: *bi-unique rel-coins*
⟨*proof*⟩

lemma *bi-total-rel-coins* [*transfer-rule*]: *bi-total rel-coins*
⟨*proof*⟩

lemma *cnth-transfer* [*transfer-rule*]: (*rel-coins* ===> (=) ===> (=)) *snth cnth*

⟨*proof*⟩

lemma *cshift-transfer* [*transfer-rule*]: ((=) ===> *rel-coins* ===> *rel-coins*) *shift cshift*
⟨*proof*⟩

lemma *ctake-transfer* [*transfer-rule*]: ((=) ===> *rel-coins* ===> (=)) *stake ctake*
⟨*proof*⟩

lemma *cdrop-transfer* [*transfer-rule*]: ((=) ===> *rel-coins* ===> *rel-coins*) *sdrop cdrop*
⟨*proof*⟩

lemma *chd-transfer* [*transfer-rule*]: (*rel-coins* ===> (=)) *shd chd*
⟨*proof*⟩

lemma *ctl-transfer* [*transfer-rule*]: (*rel-coins* ===> *rel-coins*) *stl ctl*
⟨*proof*⟩

lemma *cconst-transfer* [*transfer-rule*]: ((=) ===> *rel-coins*) *sconst cconst*
⟨*proof*⟩

end

lemma *coins-eq-iff*:
assumes $\bigwedge i.$ *cnth x i* = *cnth y i*
shows *x* = *y*
⟨*proof*⟩

lemma *length-ctake* [*simp*]: *length* (*ctake n x*) = *n*
⟨*proof*⟩

lemma *ctake-nth*[*simp*]: *m* < *n* ⟹ *ctake n s* ! *m* = *cnth s m*
⟨*proof*⟩

lemma *ctake-cdrop*: *cshift* (*ctake n s*) (*cdrop n s*) = *s*
⟨*proof*⟩

lemma *cshift-append*[*simp*]: *cshift* (*p*@*q*) *s* = *cshift p* (*cshift q s*)
⟨*proof*⟩

lemma *cshift-empty*[*simp*]: *cshift* [] *xs* = *xs*
⟨*proof*⟩

lemma *ctake-null*[*simp*]: *ctake 0 xs* = []
⟨*proof*⟩

lemma *ctake-Suc*[*simp*]: *ctake* (*Suc n*) *s* = *chd s* # *ctake n* (*ctl s*)
⟨*proof*⟩

lemma *cdrop-null*[*simp*]: *cdrop 0 s* = *s*
⟨*proof*⟩

lemma *cdrop-Suc*[*simp*]: *cdrop* (*Suc n*) *s* = *cdrop n* (*ctl s*)
⟨*proof*⟩

lemma *chd-shift*[*simp*]: *chd* (*cshift xs s*) = (*if xs* = [] *then chd s else hd xs*)
⟨*proof*⟩

lemma *ctl-shift*[*simp*]: *ctl* (*cshift xs s*) = (*if xs* = [] *then ctl s else cshift* (*tl xs*) *s*)

⟨*proof*⟩

lemma *shd-sconst*[*simp*]: $chd\ (cconst\ x) = x$
⟨*proof*⟩

lemma *take-ctake*: $take\ n\ (ctake\ m\ s) = ctake\ (min\ n\ m)\ s$
⟨*proof*⟩

lemma *ctake-add*[*simp*]: $ctake\ m\ s\ @\ ctake\ n\ (cdrop\ m\ s) = ctake\ (m + n)\ s$
⟨*proof*⟩

lemma *cdrop-add*[*simp*]: $cdrop\ m\ (cdrop\ n\ s) = cdrop\ (n + m)\ s$
⟨*proof*⟩

lemma *cprefix-iff*: $cprefix\ x\ y \longleftrightarrow (\forall\, i < length\ x.\ cnth\ y\ i = x\ !\ i)$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

A non-empty shift is not idempotent:

lemma *empty-if-shift-idem*:
assumes $\bigwedge cs.\ cshift\ h\ cs = cs$
shows $h = []$
⟨*proof*⟩

Stream version of *prefix-length-prefix*:

lemma *cprefix-length-prefix*:
assumes $length\ x \le length\ y$
assumes $cprefix\ x\ bs\ cprefix\ y\ bs$
shows $prefix\ x\ y$
⟨*proof*⟩

lemma *same-prefix-not-parallel*:
assumes $cprefix\ x\ bs\ cprefix\ y\ bs$
shows $\neg(x \parallel y)$
⟨*proof*⟩

lemma *ctake-shift*:
$ctake\ m\ (cshift\ xs\ ys) = (if\ m \le length\ xs\ then\ take\ m\ xs\ else\ xs\ @\ ctake\ (m - length\ xs)\ ys)$
⟨*proof*⟩

lemma *ctake-shift-small* [*simp*]: $m \le length\ xs \Longrightarrow ctake\ m\ (cshift\ xs\ ys) = take\ m\ xs$
and *ctake-shift-big* [*simp*]:
$m \ge length\ xs \Longrightarrow ctake\ m\ (cshift\ xs\ ys) = xs\ @\ ctake\ (m - length\ xs)\ ys$
⟨*proof*⟩

lemma *cdrop-shift*:
$cdrop\ m\ (cshift\ xs\ ys) = (if\ m \le length\ xs\ then\ cshift\ (drop\ m\ xs)\ ys\ else\ cdrop\ (m - length\ xs)\ ys)$
⟨*proof*⟩

lemma *cdrop-shift-small* [*simp*]:
$m \le length\ xs \Longrightarrow cdrop\ m\ (cshift\ xs\ ys) = cshift\ (drop\ m\ xs)\ ys$
and *cdrop-shift-big* [*simp*]:
$m \ge length\ xs \Longrightarrow cdrop\ m\ (cshift\ xs\ ys) = cdrop\ (m - length\ xs)\ ys$
⟨*proof*⟩

Infrastructure for building coin streams:

primcorec $cmap\text{-}iterate :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow coin\text{-}stream$
where

cmap-iterate m f s = Coin (m s) (cmap-iterate m f (f s))

lemma *cmap-iterate*: *cmap-iterate m f s = to-coins (smap m (siterate f s))*
⟨*proof*⟩

definition *build-coin-gen* :: $('a \Rightarrow bool\ list) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow coin\text{-}stream$
where
build-coin-gen m f s = cmap-iterate (hd ∘ fst)
$(\lambda(r,s').\ (if\ tl\ r = []\ then\ (m\ s',\ f\ s')\ else\ (tl\ r,\ s')))\ (m\ s,\ f\ s)$

lemma *build-coin-gen-aux*:
fixes $f :: 'a \Rightarrow 'b\ stream$
assumes $\bigwedge x.\ (\exists\ n\ y.\ n \neq [] \wedge f\ x = n@-f\ y \wedge g\ x = n@-g\ y)$
shows $f\ x = g\ x$
⟨*proof*⟩

lemma *build-coin-gen*:
assumes $\bigwedge x.\ m\ x \neq []$
shows *build-coin-gen m f s = to-coins (flat (smap m (siterate f s)))*
⟨*proof*⟩

Measure space for coin streams:

definition *coin-space* :: *coin-stream measure*
where *coin-space = embed-measure (stream-space (measure-pmf (pmf-of-set UNIV))) to-coins*

bundle *coin-space-notation*
begin
notation *coin-space* ($\mathcal{B}$)
end

bundle *no-coin-space-notation*
begin
no-notation *coin-space* ($\mathcal{B}$)
end

unbundle *coin-space-notation*

lemma *space-coin-space*: *space* $\mathcal{B}$ = *UNIV*
⟨*proof*⟩

lemma *B-t-eq-distr*: $\mathcal{B}$ = *distr (stream-space (pmf-of-set UNIV))* $\mathcal{B}$ *to-coins*
⟨*proof*⟩

lemma *from-coins-measurable*: *from-coins* $\in \mathcal{B} \to_M$ *(stream-space (pmf-of-set UNIV))*
⟨*proof*⟩

lemma *to-coins-measurable*: *to-coins* $\in$ *(stream-space (pmf-of-set UNIV))* $\to_M \mathcal{B}$
⟨*proof*⟩

lemma *chd-measurable*: *chd* $\in \mathcal{B} \to_M \mathcal{D}$
⟨*proof*⟩

lemma *cnth-measurable*: $(\lambda xs.\ cnth\ xs\ i) \in \mathcal{B} \to_M \mathcal{D}$
⟨*proof*⟩

lemma *B-eq-distr*:
stream-space (pmf-of-set UNIV) = distr $\mathcal{B}$ *(stream-space (pmf-of-set UNIV)) from-coins*
(**is** *?L = ?R*)

⟨*proof*⟩

lemma *B-t-finite*: *emeasure* $\mathcal{B}$ (*space* $\mathcal{B}$) = *1*
⟨*proof*⟩

interpretation *coin-space*: *prob-space coin-space*
⟨*proof*⟩

lemma *distr-shd*: *distr* $\mathcal{B}$ $\mathcal{D}$ *chd* = *pmf-of-set UNIV* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *cshift-measurable*: *cshift x* ∈ $\mathcal{B} \rightarrow_M \mathcal{B}$
⟨*proof*⟩

lemma *cdrop-measurable*: *cdrop x* ∈ $\mathcal{B} \rightarrow_M \mathcal{B}$
⟨*proof*⟩

lemma *ctake-measurable*: *ctake k* ∈ $\mathcal{B} \rightarrow_M \mathcal{D}$
⟨*proof*⟩

lemma *branch-coin-space*:
(λ(*x*, *y*). *cshift* (*ctake n x*) *y*) ∈ $\mathcal{B} \bigotimes_M \mathcal{B} \rightarrow_M \mathcal{B}$
distr ($\mathcal{B} \bigotimes_M \mathcal{B}$) $\mathcal{B}$ (λ(*x*,*y*). *cshift* (*ctake n x*) *y*) = $\mathcal{B}$ (**is** *?L* = *?R*)
⟨*proof*⟩

definition *from-coins-t* :: *coin-stream* ⇒ (*nat* ⇒ *bool discrete*)
where *from-coins-t* = *snth* ∘ *smap discrete* ∘ *from-coins*

definition *to-coins-t* :: (*nat* ⇒ *bool discrete*) ⇒ *coin-stream*
where *to-coins-t* = *to-coins* ∘ *smap of-discrete* ∘ *to-stream*

lemma *from-to-coins-t*:
from-coins-t (*to-coins-t x*) = *x*
⟨*proof*⟩

lemma *to-from-coins-t*:
to-coins-t (*from-coins-t x*) = *x*
⟨*proof*⟩

lemma *bij-to-coins-t*: *bij to-coins-t*
⟨*proof*⟩

lemma *bij-from-coins-t*: *bij from-coins-t*
⟨*proof*⟩

instantiation *coin-stream* :: *topological-space*
begin
definition *open-coin-stream* :: *coin-stream set* ⇒ *bool*
where *open-coin-stream U* = *open* (*from-coins-t* ‘ *U*)

instance ⟨*proof*⟩
end

definition *coin-stream-basis*
where *coin-stream-basis* = (λ*x*. *Collect* (*cprefix x*)) ‘ *UNIV*

lemma *image-collect-eq*: *f* ‘ {*x*. *A* (*f x*)} = {*x*. *A x*} ∩ *range f*
⟨*proof*⟩

lemma *coin-stream-basis*: *topological-basis coin-stream-basis*
⟨*proof*⟩

lemma *coin-steam-open*: *open* {*xs. cprefix x xs*}
⟨*proof*⟩

instance *coin-stream* :: *second-countable-topology*
⟨*proof*⟩

instantiation *coin-stream* :: *uniformity-dist*
begin
definition *dist-coin-stream* :: *coin-stream* ⇒ *coin-stream* ⇒ *real*
where *dist-coin-stream x y* = *dist* (*from-coins-t x*) (*from-coins-t y*)

definition *uniformity-coin-stream* :: (*coin-stream* × *coin-stream*) *filter*
where *uniformity-coin-stream* = (*INF e*∈{*0* <..}. *principal* {(*x*, *y*). *dist x y* < *e*})

instance ⟨*proof*⟩
end

lemma *in-from-coins-iff*: *x* ∈ *from-coins-t* ‘ *U* ⟷ (*to-coins-t x* ∈ *U*)
⟨*proof*⟩

instantiation *coin-stream* :: *metric-space*
begin
instance ⟨*proof*⟩
end

lemma *from-coins-t-u-continuous*: *uniformly-continuous-on UNIV from-coins-t*
⟨*proof*⟩

lemma *to-coins-t-u-continuous*: *uniformly-continuous-on UNIV to-coins-t*
⟨*proof*⟩

lemma *to-coins-t-continuous*: *continuous-on UNIV to-coins-t*
⟨*proof*⟩

instance *coin-stream* :: *complete-space*
⟨*proof*⟩

lemma *at-least-borelI*:
assumes *topological-basis K*
assumes *countable K*
assumes *K* ⊆ *sets M*
assumes *open U*
shows *U* ∈ *sets M*
⟨*proof*⟩

lemma *measurable-sets-coin-space*:
assumes *f* ∈ *measurable* $\mathcal{B}$ *A*
assumes *Collect P* ∈ *sets A*
shows {*xs. P* (*f xs*)} ∈ *sets* $\mathcal{B}$
⟨*proof*⟩

lemma *coin-space-is-borel-measure*:
assumes *open U*
shows *U* ∈ *sets* $\mathcal{B}$

⟨*proof*⟩

This is the upper topology on $'a$ *option* with the natural partial order on $'a$ *option*.

definition *option-ud* :: $'a$ *option topology*
  where *option-ud* = *topology* ($\lambda S.\ S{=}UNIV \vee None \notin S$)

lemma *option-ud-topology*: *istopology* ($\lambda S.\ S{=}UNIV \vee None \notin S$) (**is** *istopology ?T*)
⟨*proof*⟩

lemma *openin-option-ud*: *openin option-ud S* $\longleftrightarrow$ ($S = UNIV \vee None \notin S$)
  ⟨*proof*⟩

lemma *topspace-option-ud*: *topspace option-ud* = *UNIV*
⟨*proof*⟩

lemma *contionuos-into-option-udI*:
  assumes $\bigwedge x.$ *openin X* (f −‘ {*Some x*} ∩ *topspace X*)
  shows *continuous-map X option-ud f*
⟨*proof*⟩

lemma *map-option-continuous*:
  continuous-map option-ud option-ud (*map-option f*)
  ⟨*proof*⟩

end

4 Randomized Algorithms (Internal Representation)

theory *Randomized-Algorithm-Internal*
  imports
    HOL−Probability.Probability
    Coin-Space
    Tau-Additivity
    Zeta-Function.Zeta-Library

begin

This section introduces the internal representation for randomized algorithms. For ease of use, we will introduce in Section 5 a **typedef** for the monad which is easier to work with.

This is the inverse of *set-option*

definition *the-elem-opt* :: $'a$ *set* ⇒ $'a$ *option*
  where *the-elem-opt S* = (*if Set.is-singleton S then Some* (*the-elem S*) *else None*)

lemma *the-elem-opt-empty*[*simp*]: *the-elem-opt* {} = *None*
  ⟨*proof*⟩

lemma *the-elem-opt-single*[*simp*]: *the-elem-opt* {*x*} = *Some x*
  ⟨*proof*⟩

definition *at-most-one* :: $'a$ *set* ⇒ *bool*
  where *at-most-one S* $\longleftrightarrow$ ($\forall x\ y.\ x \in S \wedge y \in S \longrightarrow x = y$)

lemma *at-most-one-cases*[*consumes 1*]:
  assumes *at-most-one S*
  assumes *P* {*the-elem S*}
  assumes *P* {}

shows $P\ S$
⟨*proof*⟩

lemma *the-elem-opt-Some-iff*[*simp*]: $\textit{at-most-one}\ S \Longrightarrow \textit{the-elem-opt}\ S = \textit{Some}\ x \longleftrightarrow S = \{x\}$
⟨*proof*⟩

lemma *the-elem-opt-None-iff*[*simp*]: $\textit{at-most-one}\ S \Longrightarrow \textit{the-elem-opt}\ S = \textit{None} \longleftrightarrow S = \{\}$
⟨*proof*⟩

The following is the fundamental type of the randomized algorithms, which are represented as functions that take an infinite stream of coin flips and return the unused suffix of coin-flips together with the result. We use the *'a option* type to be able to introduce the denotational semantics for the monad.

type-synonym $'a\ \textit{random-alg-int} = \textit{coin-stream} \Rightarrow ('a \times \textit{coin-stream})\ \textit{option}$

The *return-rai* combinator, does not consume any coin-flips and thus returns the entire stream together with the result.

definition $\textit{return-rai} :: 'a \Rightarrow 'a\ \textit{random-alg-int}$
where $\textit{return-rai}\ x\ bs = \textit{Some}\ (x,\ bs)$

The *bind-rai* combinator passes the coin-flips to the first algorithm, then passes the remaining coin flips to the second function, and returns the unused coin-flips from both steps.

definition $\textit{bind-rai} :: 'a\ \textit{random-alg-int} \Rightarrow ('a \Rightarrow 'b\ \textit{random-alg-int}) \Rightarrow 'b\ \textit{random-alg-int}$
where $\textit{bind-rai}\ m\ f\ bs =$
do {
$(r,\ bs') \leftarrow m\ bs;$
$f\ r\ bs'$
}

adhoc-overloading *Monad-Syntax.bind bind-rai*

The *coin-rai* combinator consumes one coin-flip and return it as the result, while the tail of the coin flips are returned as unused.

definition $\textit{coin-rai} :: \textit{bool random-alg-int}$
where $\textit{coin-rai}\ bs = \textit{Some}\ (\textit{chd}\ bs,\ \textit{ctl}\ bs)$

This representation is similar to the model proposed by Hurd [5] [2]. It is also closely related to the construction of parser monads in functional languages [6].

We also had following alternatives considered, with various advantages and drawbacks:

- *Returning the count of used coin flips:* Instead of returning a suffix of the input stream a randomized algorithm could also return the number of used coin flips, which then would allow the definition of the bind function, in a way that performs the appropriate shift in the stream according to the returned number. An advantage of this model, is that it makes the number of used coin-flips immediately available. (As we will see below, this is still possible even in the formalized model, albeit with some more work.) The main disadvantage of this model is that in scenarios, where the coin-flips cannot be computed in a random-access way, it leads to performance degradation. Indeed it is easy to construct example algorithms, which incur asymptotically quadratic slow-down compared to the formalized model.
- *Trees of coin-flips:* Another model we were considering is to require an infinite tree of coin-flips as input instead of a stream. Here the idea is that each bind operation

[2] Although we were not aware of the technical report, when initially considering this representation.

would pass the left sub-tree to the first algorithm and the right sub-tree to the second algorithm. This model has the dis-advantage that the resulting ‘’monad”, does not fulfill the associativity law. Moreover many PRG’s are designed and tested in the streaming sense, and there is not a lot of research into the performance of PRGs with tree structured output. (A related idea was to still use a stream as input, and split it into two sub-streams for example by the parity of the stream position. This alternative also suffers from the lack of associativity problem and may lead to a lot of unused coin flips.)

Another reason for using the formalized representation is compatibility with linear types [1], if support for them are introduced in Isabelle in future.

Monad laws:

lemma *return-bind-rai*: $\mathit{bind\text{-}rai}\ (\mathit{return\text{-}rai}\ x)\ g = g\ x$
⟨*proof*⟩

lemma *bind-rai-assoc*: $\mathit{bind\text{-}rai}\ (\mathit{bind\text{-}rai}\ f\ g)\ h = \mathit{bind\text{-}rai}\ f\ (\lambda x.\ \mathit{bind\text{-}rai}\ (g\ x)\ h)$
⟨*proof*⟩

lemma *bind-return-rai*: $\mathit{bind\text{-}rai}\ m\ \mathit{return\text{-}rai} = m$
⟨*proof*⟩

definition *wf-on-prefix* :: $'a\ \mathit{random\text{-}alg\text{-}int} \Rightarrow \mathit{bool\ list} \Rightarrow\ 'a \Rightarrow \mathit{bool}$ **where**
$\mathit{wf\text{-}on\text{-}prefix}\ f\ p\ r = (\forall cs.\ f\ (\mathit{cshift}\ p\ cs) = \mathit{Some}\ (r,cs))$

definition *wf-random* :: $'a\ \mathit{random\text{-}alg\text{-}int} \Rightarrow \mathit{bool}$ **where**
$\mathit{wf\text{-}random}\ f \longleftrightarrow (\forall bs.$
$\quad \mathit{case}\ f\ bs\ \mathit{of}$
$\quad\quad \mathit{None} \Rightarrow \mathit{True}\ |$
$\quad\quad \mathit{Some}\ (r,bs') \Rightarrow (\exists p.\ \mathit{cprefix}\ p\ bs \wedge \mathit{wf\text{-}on\text{-}prefix}\ f\ p\ r))$

definition *range-rm* :: $'a\ \mathit{random\text{-}alg\text{-}int} \Rightarrow\ 'a\ \mathit{set}$
where $\mathit{range\text{-}rm}\ f = \mathit{Some}\ -`\ (\mathit{range}\ (\mathit{map\text{-}option}\ \mathit{fst} \circ f))$

lemma *in-range-rmI*:
assumes $r\ bs = \mathit{Some}\ (y,\ n)$
shows $y \in \mathit{range\text{-}rm}\ r$
⟨*proof*⟩

definition *distr-rai* :: $'a\ \mathit{random\text{-}alg\text{-}int} \Rightarrow\ 'a\ \mathit{option\ measure}$
where $\mathit{distr\text{-}rai}\ f = \mathit{distr}\ \mathcal{B}\ \mathcal{D}\ (\mathit{map\text{-}option}\ \mathit{fst} \circ f)$

lemma *wf-randomI*:
assumes $\bigwedge bs.\ f\ bs \neq \mathit{None} \Longrightarrow (\exists p\ r.\ \mathit{cprefix}\ p\ bs \wedge \mathit{wf\text{-}on\text{-}prefix}\ f\ p\ r)$
shows $\mathit{wf\text{-}random}\ f$
⟨*proof*⟩

lemma *wf-on-prefix-bindI*:
assumes $\mathit{wf\text{-}on\text{-}prefix}\ m\ p\ r$
assumes $\mathit{wf\text{-}on\text{-}prefix}\ (f\ r)\ q\ s$
shows $\mathit{wf\text{-}on\text{-}prefix}\ (m \gg= f)\ (p@q)\ s$
⟨*proof*⟩

lemma *wf-bind*:
assumes $\mathit{wf\text{-}random}\ m$
assumes $\bigwedge x.\ x \in \mathit{range\text{-}rm}\ m \Longrightarrow \mathit{wf\text{-}random}\ (f\ x)$
shows $\mathit{wf\text{-}random}\ (m \gg= f)$

⟨*proof*⟩

lemma *wf-return*:
  wf-random (*return-rai* x)
⟨*proof*⟩

lemma *wf-coin*:
  wf-random (*coin-rai*)
⟨*proof*⟩

definition *ptree-rm* :: $'a$ *random-alg-int* $\Rightarrow$ *bool list set*
  where *ptree-rm* f = $\{p.\ \exists\, r.$ *wf-on-prefix* $f\ p\ r\}$

definition *eval-rm* :: $'a$ *random-alg-int* $\Rightarrow$ *bool list* $\Rightarrow$ $'a$ **where**
  eval-rm $f\ p$ = *fst* (*the* (f (*cshift* p (*cconst False*))))

lemma *eval-rmD*:
  assumes *wf-on-prefix* $f\ p\ r$
  shows *eval-rm* $f\ p = r$
  ⟨*proof*⟩

lemma *wf-on-prefixD*:
  assumes *wf-on-prefix* $f\ p\ r$
  assumes *cprefix* p *bs*
  shows f *bs* = *Some* (*eval-rm* $f\ p$, *cdrop* (*length* p) *bs*)
⟨*proof*⟩

lemma *prefixes-parallel-helper*:
  assumes $p \in$ *ptree-rm* f
  assumes $q \in$ *ptree-rm* f
  assumes *prefix* $p\ q$
  shows $p = q$
⟨*proof*⟩

lemma *prefixes-parallel*:
  assumes $p \in$ *ptree-rm* f
  assumes $q \in$ *ptree-rm* f
  shows $p = q \lor p \parallel q$
  ⟨*proof*⟩

lemma *prefixes-singleton*:
  assumes $p \in \{p.\ p \in$ *ptree-rm* $f \land$ *cprefix* p *bs*$\}$
  shows $\{p \in$ *ptree-rm* $f.$ *cprefix* p *bs*$\} = \{p\}$
⟨*proof*⟩

lemma *prefixes-at-most-one*:
  at-most-one $\{p \in$ *ptree-rm* $f.$ *cprefix* $p\ x\}$
  ⟨*proof*⟩

definition *consumed-prefix* f *bs* = *the-elem-opt* $\{p \in$ *ptree-rm* $f.$ *cprefix* p *bs*$\}$

lemma *wf-random-alt*:
  assumes *wf-random* f
  shows f *bs* = *map-option* ($\lambda p.$ (*eval-rm* $f\ p$, *cdrop* (*length* p) *bs*)) (*consumed-prefix* f *bs*)
⟨*proof*⟩

lemma *range-rm-alt*:
  assumes *wf-random* f

shows *range-rm f* = *eval-rm f* ‘ *ptree-rm f* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *consumed-prefix-some-iff*:
consumed-prefix f bs = *Some p* ⟷ (*p* ∈ *ptree-rm f* ∧ *cprefix p bs*)
⟨*proof*⟩

definition *consumed-bits* **where**
consumed-bits f bs = *map-option length* (*consumed-prefix f bs*)

definition *used-bits-distr* :: *'a random-alg-int* ⇒ *nat option measure*
where *used-bits-distr f* = *distr* $\mathcal{B}$ $\mathcal{D}$ (*consumed-bits f*)

lemma *wf-random-alt2*:
assumes *wf-random f*
shows *f bs* = *map-option* (λ*n*. (*eval-rm f* (*ctake n bs*), *cdrop n bs*)) (*consumed-bits f bs*)
(**is** *?L* = *?R*)
⟨*proof*⟩

lemma *consumed-prefix-none-iff*:
assumes *wf-random f*
shows *f bs* = *None* ⟷ *consumed-prefix f bs* = *None*
⟨*proof*⟩

lemma *consumed-bits-inf-iff*:
assumes *wf-random f*
shows *f bs* = *None* ⟷ *consumed-bits f bs* = *None*
⟨*proof*⟩

lemma *consumed-bits-enat-iff*:
consumed-bits f bs = *Some n* ⟷ *ctake n bs* ∈ *ptree-rm f* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *consumed-bits-measurable*: *consumed-bits f* ∈ $\mathcal{B} \rightarrow_M \mathcal{D}$
⟨*proof*⟩

lemma *R-sets*:
assumes *wf*:*wf-random f*
shows {*bs*. *f bs* = *None*} ∈ *sets* $\mathcal{B}$ {*bs*. *f bs* ≠ *None*} ∈ *sets* $\mathcal{B}$
⟨*proof*⟩

lemma *countable-range*:
assumes *wf*:*wf-random f*
shows *countable* (*range-rm f*)
⟨*proof*⟩

lemma *consumed-prefix-continuous*:
continuous-map euclidean option-ud (*consumed-prefix f*)
⟨*proof*⟩

Randomized algorithms are continuous with respect to the product topology on the domain and the upper topology on the range.

lemma *f-continuous*:
assumes *wf*:*wf-random f*
shows *continuous-map euclidean option-ud* (*map-option fst* ∘ *f*)
⟨*proof*⟩

lemma *none-measure-subprob-algebra*:

$return\ \mathcal{D}\ None \in space\ (subprob\text{-}algebra\ \mathcal{D})$
⟨*proof*⟩

context
fixes f :: $'a\ random\text{-}alg\text{-}int$
fixes R
assumes wf: $wf\text{-}random\ f$
defines $R \equiv restrict\text{-}space\ \mathcal{B}\ \{bs.\ f\ bs \neq None\}$
begin

lemma *the-f-measurable*: $the \circ f \in R \to_M \mathcal{D} \bigotimes_M \mathcal{B}$
⟨*proof*⟩

lemma *distr-rai-measurable*: $map\text{-}option\ fst \circ f \in \mathcal{B} \to_M \mathcal{D}$
⟨*proof*⟩

lemma *distr-rai-subprob-space*:
$distr\text{-}rai\ f \in space\ (subprob\text{-}algebra\ \mathcal{D})$
⟨*proof*⟩

lemma *fst-the-f-measurable*: $fst \circ the \circ f \in R \to_M \mathcal{D}$
⟨*proof*⟩

lemma *prob-space-distr-rai*:
$prob\text{-}space\ (distr\text{-}rai\ f)$
⟨*proof*⟩

This is the central correctness property for the monad. The returned stream of coins is independent of the result of the randomized algorithm.

lemma *remainder-indep*:
$distr\ R\ (\mathcal{D} \bigotimes_M \mathcal{B})\ (the \circ f) = distr\ R\ \mathcal{D}\ (fst \circ the \circ f) \bigotimes_M \mathcal{B}$
⟨*proof*⟩

end

lemma *distr-rai-bind*:
assumes $wf\text{-}m$: $wf\text{-}random\ m$
assumes $wf\text{-}f$: $\bigwedge x.\ x \in range\text{-}rm\ m \Longrightarrow wf\text{-}random\ (f\ x)$
shows $distr\text{-}rai\ (m \gg\!= f) = distr\text{-}rai\ m \gg\!=$
$(\lambda x.\ if\ x \in Some\ `\ range\text{-}rm\ m\ then\ distr\text{-}rai\ (f\ (the\ x))\ else\ return\ \mathcal{D}\ None)$
(**is** $?L = ?RHS$)
⟨*proof*⟩

lemma *return-discrete*: $return\ \mathcal{D}\ x = return\text{-}pmf\ x$
⟨*proof*⟩

lemma *distr-rai-return*: $distr\text{-}rai\ (return\text{-}rai\ x) = return\ \mathcal{D}\ (Some\ x)$
⟨*proof*⟩

lemma *distr-rai-return'*: $distr\text{-}rai\ (return\text{-}rai\ x) = return\text{-}spmf\ x$
⟨*proof*⟩

lemma *distr-rai-coin*: $distr\text{-}rai\ coin\text{-}rai = coin\text{-}spmf$ (**is** $?L = ?R$)
⟨*proof*⟩

definition *ord-rai* :: $'a\ random\text{-}alg\text{-}int \Rightarrow 'a\ random\text{-}alg\text{-}int \Rightarrow bool$
where $ord\text{-}rai = fun\text{-}ord\ (flat\text{-}ord\ None)$

definition *lub-rai* :: $'a$ *random-alg-int set* $\Rightarrow$ $'a$ *random-alg-int*
  where *lub-rai* = *fun-lub* (*flat-lub None*)

lemma *random-alg-int-pd-fact*:
  partial-function-definitions ord-rai lub-rai
  ⟨*proof*⟩

interpretation *random-alg-int-pd*: *partial-function-definitions ord-rai lub-rai*
  ⟨*proof*⟩

lemma *wf-lub-helper*:
  assumes *ord-rai f g*
  assumes *wf-on-prefix f p r*
  shows *wf-on-prefix g p r*
⟨*proof*⟩

lemma *wf-lub*:
  assumes *Complete-Partial-Order.chain ord-rai R*
  assumes $\bigwedge r.\ r \in R \Longrightarrow$ *wf-random r*
  shows *wf-random* (*lub-rai R*)
⟨*proof*⟩

lemma *ord-rai-mono*:
  assumes *ord-rai f g*
  assumes $\neg$ (*P None*)
  assumes *P* (*f bs*)
  shows *P* (*g bs*)
  ⟨*proof*⟩

lemma *lub-rai-empty*:
  lub-rai {} = *Map.empty*
  ⟨*proof*⟩

lemma *distr-rai-lub*:
  assumes $F \neq \{\}$
  assumes *Complete-Partial-Order.chain ord-rai F*
  assumes *wf-f*: $\bigwedge f.\ f \in F \Longrightarrow$ *wf-random f*
  assumes *None* $\notin A$
  shows *emeasure* (*distr-rai* (*lub-rai F*)) *A* = (*SUP f* $\in$ *F*. *emeasure* (*distr-rai f*) *A*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *distr-rai-ord-rai-mono*:
  assumes *wf-random f wf-random g ord-rai f g*
  assumes *None* $\notin A$
  shows *emeasure* (*distr-rai f*) *A* $\leq$ *emeasure* (*distr-rai g*) *A* (**is** *?L* $\leq$ *?R*)
⟨*proof*⟩

lemma *distr-rai-None*: *distr-rai* (λ-. *None*) = *measure-pmf* (*return-pmf* (*None* :: $'a$ *option*))
⟨*proof*⟩

lemma *bind-rai-mono*:
  assumes *ord-rai f1 f2* $\bigwedge y.$ *ord-rai* (*g1 y*) (*g2 y*)
  shows *ord-rai* (*bind-rai f1 g1*) (*bind-rai f2 g2*)
⟨*proof*⟩

end

5 Randomized Algorithms

This section introduces the *random-alg* monad, that can be used to represent executable randomized algorithms. It is a type-definition based on the internal representation from Section 4 with the wellformedness restriction.

Additionally, we introduce the *spmf-of-ra* morphism, which represent the distribution of a randomized algorithm, under the assumption that the coin flips are independent and unbiased.

We also show that it is a Scott-continuous monad-morphism and introduce transfer theorems, with which it is possible to establish the corresponding SPMF of a randomized algorithms, even in the case of (possibly infinite) loops.

theory *Randomized-Algorithm*
  imports
    Randomized-Algorithm-Internal
begin

A stronger variant of *pmf-eqI*.

lemma *pmf-eq-iff-le*:
  fixes $p\ q :: 'a\ pmf$
  assumes $\bigwedge x.\ pmf\ p\ x \leq pmf\ q\ x$
  shows $p = q$
⟨*proof*⟩

The following is a stronger variant of *ord-spmf-eq-pmf-None-eq*

lemma *eq-iff-ord-spmf*:
  assumes $weight\text{-}spmf\ p \geq weight\text{-}spmf\ q$
  assumes $ord\text{-}spmf\ (=)\ p\ q$
  shows $p = q$
⟨*proof*⟩

lemma *wf-empty*: $wf\text{-}random\ (\lambda\text{-}.\ None)$
  ⟨*proof*⟩

typedef $'a\ random\text{-}alg = \{(r :: 'a\ random\text{-}alg\text{-}int).\ wf\text{-}random\ r\}$
  ⟨*proof*⟩

setup-lifting *type-definition-random-alg*

lift-definition $return\text{-}ra :: 'a \Rightarrow 'a\ random\text{-}alg$ **is** *return-rai*
  ⟨*proof*⟩

lift-definition $coin\text{-}ra :: bool\ random\text{-}alg$ **is** *coin-rai*
  ⟨*proof*⟩

lift-definition $bind\text{-}ra :: 'a\ random\text{-}alg \Rightarrow ('a \Rightarrow 'b\ random\text{-}alg) \Rightarrow 'b\ random\text{-}alg$ **is** *bind-rai*
  ⟨*proof*⟩

adhoc-overloading *Monad-Syntax.bind bind-ra*

Monad laws:

lemma *return-bind-ra*:
  $bind\text{-}ra\ (return\text{-}ra\ x)\ g = g\ x$
  ⟨*proof*⟩

lemma *bind-ra-assoc*:

bind-ra (*bind-ra f g*) *h* = *bind-ra f* (λx. *bind-ra* (*g x*) *h*)
⟨*proof*⟩

lemma *bind-return-ra*:
bind-ra m return-ra = *m*
⟨*proof*⟩

lift-definition *lub-ra* :: *'a random-alg set* ⇒ *'a random-alg* **is**
(λF. *if Complete-Partial-Order.chain ord-rai F then lub-rai F else* (λx. *None*))
⟨*proof*⟩

lift-definition *ord-ra* :: *'a random-alg* ⇒ *'a random-alg* ⇒ *bool* **is** *ord-rai* ⟨*proof*⟩

lift-definition *run-ra* :: *'a random-alg* ⇒ *coin-stream* ⇒ *'a option* **is**
(λf *s*. *map-option fst* (*f s*)) ⟨*proof*⟩

context
begin

interpretation *pmf-as-measure* ⟨*proof*⟩

lemma *distr-rai-is-pmf*:
assumes *wf-random f*
shows
prob-space (*distr-rai f*) (**is** *?A*)
sets (*distr-rai f*) = *UNIV* (**is** *?B*)
AE x in distr-rai f. *measure* (*distr-rai f*) {*x*} ≠ *0* (**is** *?C*)
⟨*proof*⟩

lift-definition *spmf-of-ra* :: *'a random-alg* ⇒ *'a spmf* **is** *distr-rai*
⟨*proof*⟩

lemma *used-bits-distr-is-pmf*:
assumes *wf-random f*
shows
prob-space (*used-bits-distr f*) (**is** *?A*)
sets (*used-bits-distr f*) = *UNIV* (**is** *?B*)
AE x in used-bits-distr f. *measure* (*used-bits-distr f*) {*x*} ≠ *0* (**is** *?C*)
⟨*proof*⟩

lift-definition *coin-usage-of-ra-aux* :: *'a random-alg* ⇒ *nat spmf* **is** *used-bits-distr*
⟨*proof*⟩

definition *coin-usage-of-ra*
where *coin-usage-of-ra p* = *map-pmf* (*case-option* ∞ *enat*) (*coin-usage-of-ra-aux p*)

end

lemma *wf-rep-rand-alg*:
wf-random (*Rep-random-alg f*)
⟨*proof*⟩

lemma *set-pmf-spmf-of-ra*:
set-pmf (*spmf-of-ra f*) ⊆ *Some* ‘ *range-rm* (*Rep-random-alg f*) ∪ {*None*}
⟨*proof*⟩

lemma *spmf-of-ra-return*: *spmf-of-ra* (*return-ra x*) = *return-spmf x*
⟨*proof*⟩

lemma *spmf-of-ra-coin*: *spmf-of-ra coin-ra* = *coin-spmf*
⟨*proof*⟩

lemma *spmf-of-ra-bind*:
spmf-of-ra (*bind-ra f g*) = *bind-spmf* (*spmf-of-ra f*) (λx. *spmf-of-ra* (*g x*)) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *spmf-of-ra-mono*:
assumes *ord-ra f g*
shows *ord-spmf* (=) (*spmf-of-ra f*) (*spmf-of-ra g*)
⟨*proof*⟩

lemma *spmf-of-ra-lub-ra-empty*:
spmf-of-ra (*lub-ra* {}) = *return-pmf None* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *spmf-of-ra-lub-ra*:
fixes *A* :: *'a random-alg set*
assumes *Complete-Partial-Order.chain ord-ra A*
shows *spmf-of-ra* (*lub-ra A*) = *lub-spmf* (*spmf-of-ra* ‘ *A*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *rep-lub-ra*:
assumes *Complete-Partial-Order.chain ord-ra F*
shows *Rep-random-alg* (*lub-ra F*) = *lub-rai* (*Rep-random-alg* ‘ *F*)
⟨*proof*⟩

lemma *partial-function-image-improved*:
fixes *ord*
assumes $\bigwedge A$. *Complete-Partial-Order.chain ord* (*f* ‘ *A*) $\Longrightarrow$ *l1* (*f* ‘ *A*) = *f* (*l2 A*)
assumes *partial-function-definitions ord l1*
assumes *inj f*
shows *partial-function-definitions* (*img-ord f ord*) *l2*
⟨*proof*⟩

lemma *random-alg-pfd*: *partial-function-definitions ord-ra lub-ra*
⟨*proof*⟩

interpretation *random-alg-pf*: *partial-function-definitions ord-ra lub-ra*
⟨*proof*⟩

abbreviation *mono-ra* ≡ *monotone* (*fun-ord ord-ra*) *ord-ra*

lemma *bind-mono-aux-ra*:
assumes *ord-ra f1 f2* $\bigwedge y$. *ord-ra* (*g1 y*) (*g2 y*)
shows *ord-ra* (*bind-ra f1 g1*) (*bind-ra f2 g2*)
⟨*proof*⟩

lemma *bind-mono-ra* [*partial-function-mono*]:
assumes *mono-ra B* **and** $\bigwedge y$. *mono-ra* (*C y*)
shows *mono-ra* (λf. *bind-ra* (*B f*) (λy. *C y f*))
⟨*proof*⟩

definition *map-ra* :: (*'a* ⇒ *'b*) ⇒ *'a random-alg* ⇒ *'b random-alg*
where *map-ra f p* = *p* ⤜ (λx. *return-ra* (*f x*))

lemma *spmf-of-ra-map*: *spmf-of-ra* (*map-ra f p*) = *map-spmf f* (*spmf-of-ra p*)

⟨*proof*⟩

lemmas *spmf-of-ra-simps* =
spmf-of-ra-return spmf-of-ra-bind spmf-of-ra-coin spmf-of-ra-map

lemma *map-mono-ra* [*partial-function-mono*]:
assumes *mono-ra B*
shows *mono-ra* (*λf. map-ra g* (*B f*))
⟨*proof*⟩

definition *rel-spmf-of-ra* :: *'a spmf ⇒ 'a random-alg ⇒ bool* **where**
rel-spmf-of-ra q p ⟷ *q* = *spmf-of-ra p*

lemma *admissible-rel-spmf-of-ra*:
ccpo.admissible (*prod-lub lub-spmf lub-ra*) (*rel-prod* (*ord-spmf* (=)) *ord-ra*) (*case-prod rel-spmf-of-ra*)
(**is** *ccpo.admissible ?lub ?ord ?P*)
⟨*proof*⟩

lemma *admissible-rel-spmf-of-ra-cont* [*cont-intro*]:
fixes *ord*
shows ⟦ *mcont lub ord lub-spmf* (*ord-spmf* (=)) *f*; *mcont lub ord lub-ra ord-ra g* ⟧
⟹ *ccpo.admissible lub ord* (*λx. rel-spmf-of-ra* (*f x*) (*g x*))
⟨*proof*⟩

lemma *mcont2mcont-spmf-of-ra*[*THEN spmf.mcont2mcont*, *cont-intro*]:
shows *mcont-spmf-of-sampler*: *mcont lub-ra ord-ra lub-spmf* (*ord-spmf* (=)) *spmf-of-ra*
⟨*proof*⟩

context
includes *lifting-syntax*
begin

lemma *fixp-ra-parametric*[*transfer-rule*]:
assumes *f*: ⋀*x. mono-spmf* (*λf. F f x*)
and *g*: ⋀*x. mono-ra* (*λf. G f x*)
and *param*: ((*A* ===> *rel-spmf-of-ra*) ===> *A* ===> *rel-spmf-of-ra*) *F G*
shows (*A* ===> *rel-spmf-of-ra*) (*spmf.fixp-fun F*) (*random-alg-pf.fixp-fun G*)
⟨*proof*⟩

lemma *return-ra-tranfer*[*transfer-rule*]: ((=) ===> *rel-spmf-of-ra*) *return-spmf return-ra*
⟨*proof*⟩

lemma *bind-ra-tranfer*[*transfer-rule*]:
(*rel-spmf-of-ra* ===> ((=) ===> *rel-spmf-of-ra*) ===> *rel-spmf-of-ra*) *bind-spmf bind-ra*
⟨*proof*⟩

lemma *coin-ra-tranfer*[*transfer-rule*]:
rel-spmf-of-ra coin-spmf coin-ra
⟨*proof*⟩

lemma *map-ra-tranfer*[*transfer-rule*]:
((=) ===> *rel-spmf-of-ra* ===> *rel-spmf-of-ra*) *map-spmf map-ra*
⟨*proof*⟩

end

declare [[*function-internals*]]

⟨*ML*⟩

5.1 Almost surely terminating randomized algorithms

definition *terminates-almost-surely* :: *'a random-alg* ⇒ *bool*
 where *terminates-almost-surely f* ⟷ *lossless-spmf* (*spmf-of-ra f*)

definition *pmf-of-ra* :: *'a random-alg* ⇒ *'a pmf* **where**
 pmf-of-ra p = *map-pmf the* (*spmf-of-ra p*)

lemma *pmf-of-spmf*: *map-pmf the* (*spmf-of-pmf x*) = *x*
 ⟨*proof*⟩

definition *coin-pmf* :: *bool pmf* **where** *coin-pmf* = *pmf-of-set UNIV*

lemma *pmf-of-ra-coin*: *pmf-of-ra* (*coin-ra*) = *coin-pmf* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *pmf-of-ra-return*: *pmf-of-ra* (*return-ra x*) = *return-pmf x*
 ⟨*proof*⟩

lemma *pmf-of-ra-bind*:
 assumes *terminates-almost-surely f*
 shows *pmf-of-ra* (*f* ⤜ *g*) = *pmf-of-ra f* ⤜ (*λx. pmf-of-ra* (*g x*)) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *pmf-of-ra-map*:
 assumes *terminates-almost-surely m*
 shows *pmf-of-ra* (*map-ra f m*) = *map-pmf f* (*pmf-of-ra m*)
 ⟨*proof*⟩

lemma *terminates-almost-surely-return*:
 terminates-almost-surely (*return-ra x*)
 ⟨*proof*⟩

lemma *terminates-almost-surely-coin*:
 terminates-almost-surely coin-ra
 ⟨*proof*⟩

lemma *terminates-almost-surely-bind*:
 assumes *terminates-almost-surely f*
 assumes $\bigwedge x.$ *x* ∈ *set-pmf* (*pmf-of-ra f*) ⟹ *terminates-almost-surely* (*g x*)
 shows *terminates-almost-surely* (*f* ⤜ *g*)
⟨*proof*⟩

lemma *terminates-almost-surely-map*:
 assumes *terminates-almost-surely p*
 shows *terminates-almost-surely* (*map-ra f p*)
 ⟨*proof*⟩

lemmas *pmf-of-ra-simps* =
 pmf-of-ra-return pmf-of-ra-bind pmf-of-ra-coin pmf-of-ra-map

lemmas *terminates-almost-surely-intros* =
 terminates-almost-surely-return
 terminates-almost-surely-bind
 terminates-almost-surely-coin
 terminates-almost-surely-map

end

6 Tracking Randomized Algorithms

This section introduces the *track-random-bits* monad morphism, which converts a randomized algorithm to one that tracks the number of used coin-flips. The resulting algorithm can still be executed. This morphism is useful for testing and debugging. For the verification of coin-flip usage, the morphism *tspmf-of-ra* introduced in Section 7 is more useful.

theory *Tracking-Randomized-Algorithm*
  imports *Randomized-Algorithm*
begin

definition *track-random-bits* :: *'a random-alg-int* ⇒ (*'a* × *nat*) *random-alg-int*
  where *track-random-bits f bs* =
    do {
      (*r*,*bs'*) ← *f bs*;
      n ← *consumed-bits f bs*;
      Some ((*r*,*n*),*bs'*)
    }

lemma *track-random-bits-Some-iff*:
  assumes *track-random-bits f bs* ≠ *None*
  shows *f bs* ≠ *None*
  ⟨*proof*⟩

lemma *track-random-bits-alt*:
  assumes *wf-random f*
  shows *track-random-bits f bs* =
    map-option (λ*p*. ((*eval-rm f p*, *length p*), *cdrop* (*length p*) *bs*)) (*consumed-prefix f bs*)
⟨*proof*⟩

lemma *track-rb-coin*:
  track-random-bits coin-rai = *coin-rai* ≫= (λ*b*. *return-rai* (*b*,*1*)) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *track-rb-return*: *track-random-bits* (*return-rai x*) = *return-rai* (*x*,*0*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *consumed-prefix-imp-wf*:
  assumes *consumed-prefix m bs* = *Some p*
  shows *wf-on-prefix m p* (*eval-rm m p*)
⟨*proof*⟩

lemma *consumed-prefix-imp-prefix*:
  assumes *consumed-prefix m bs* = *Some p*
  shows *cprefix p bs*
  ⟨*proof*⟩

lemma *consumed-prefix-bindI*:
  assumes *consumed-prefix m bs* = *Some p*
  assumes *consumed-prefix* (*f* (*eval-rm m p*)) (*cdrop* (*length p*) *bs*) = *Some q*
  shows *consumed-prefix* (*m* ≫= *f*) *bs* = *Some* (*p*@*q*)
⟨*proof*⟩

lemma *track-rb-bind*:

assumes *wf-random m*
assumes $\bigwedge x.\ x \in$ *range-rm m* $\Longrightarrow$ *wf-random* $(f\ x)$
shows *track-random-bits* $(m \gg= f) =$ *track-random-bits m* $\gg=$
$(\lambda(r,n).$ *track-random-bits* $(f\ r) \gg= (\lambda(r',m).$ *return-rai* $(r',n+m)))$ (**is** *?L = ?R*)
⟨*proof*⟩

lemma *track-random-bits-mono*:
assumes *wf-random f wf-random g*
assumes *ord-rai f g*
shows *ord-rai* (*track-random-bits f*) (*track-random-bits g*)
⟨*proof*⟩

lemma *wf-track-random-bits*:
assumes *wf-random f*
shows *wf-random* (*track-random-bits f*)
⟨*proof*⟩

lemma *track-random-bits-lub-rai*:
assumes *Complete-Partial-Order.chain ord-rai A*
assumes $\bigwedge r.\ r \in A \Longrightarrow$ *wf-random r*
shows *track-random-bits* (*lub-rai A*) = *lub-rai* (*track-random-bits ‘ A*) (**is** *?L = ?R*)
⟨*proof*⟩

lemma *untrack-random-bits*:
assumes *wf-random f*
shows *track-random-bits f* $\gg= (\lambda x.$ *return-rai* (*fst x*)$)$ = *f* (**is** *?L = ?R*)
⟨*proof*⟩

lift-definition *track-coin-use* :: *'a random-alg* $\Rightarrow$ (*'a* $\times$ *nat*) *random-alg*
is *track-random-bits*
⟨*proof*⟩

definition *bind-tra* ::
(*'a* $\times$ *nat*) *random-alg* $\Rightarrow$ (*'a* $\Rightarrow$ (*'b* $\times$ *nat*) *random-alg*) $\Rightarrow$ (*'b* $\times$ *nat*) *random-alg*
where *bind-tra m f = do* {
$(r,k) \leftarrow m$;
$(s,l) \leftarrow (f\ r)$;
return-ra $(s,\ k+l)$
}

definition *coin-tra* :: (*bool* $\times$ *nat*) *random-alg*
where *coin-tra = do* {
$b \leftarrow$ *coin-ra*;
return-ra $(b,1)$
}

definition *return-tra* :: *'a* $\Rightarrow$ (*'a* $\times$ *nat*) *random-alg*
where *return-tra x = return-ra* $(x,0)$

adhoc-overloading *Monad-Syntax.bind bind-tra*

Monad laws:

lemma *return-bind-tra*:
bind-tra (*return-tra x*) *g = g x*
⟨*proof*⟩

lemma *bind-tra-assoc*:
bind-tra (*bind-tra f g*) *h = bind-tra f* $(\lambda x.$ *bind-tra* $(g\ x)\ h)$

⟨*proof*⟩

lemma *bind-return-tra*:
bind-tra m return-tra = *m*
⟨*proof*⟩

lemma *track-coin-use-bind*:
fixes m :: *'a random-alg*
fixes f :: *'a* ⇒ *'b random-alg*
shows *track-coin-use* (m ≫= f) = *track-coin-use m* ≫= (λr. *track-coin-use* (f r))
(**is** *?L* = *?R*)
⟨*proof*⟩

lemma *track-coin-use-coin*: *track-coin-use coin-ra* = *coin-tra* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *track-coin-use-return*: *track-coin-use* (*return-ra x*) = *return-tra x* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *track-coin-use-lub*:
assumes *Complete-Partial-Order.chain ord-ra A*
shows *track-coin-use* (*lub-ra A*) = *lub-ra* (*track-coin-use* ‘ *A*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *track-coin-use-mono*:
assumes *ord-ra f g*
shows *ord-ra* (*track-coin-use f*) (*track-coin-use g*)
⟨*proof*⟩

lemma *bind-mono-tra-aux*:
assumes *ord-ra f1 f2* $\bigwedge y$. *ord-ra* (*g1 y*) (*g2 y*)
shows *ord-ra* (*bind-tra f1 g1*) (*bind-tra f2 g2*)
⟨*proof*⟩

lemma *bind-tra-mono* [*partial-function-mono*]:
assumes *mono-ra B* **and** $\bigwedge y$. *mono-ra* (*C y*)
shows *mono-ra* (λf. *bind-tra* (*B f*) (λy. *C y f*))
⟨*proof*⟩

lemma *track-coin-use-empty*:
track-coin-use (*lub-ra* {}) = (*lub-ra* {}) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *untrack-coin-use*:
map-ra fst (*track-coin-use f*) = *f* (**is** *?L* = *?R*)
⟨*proof*⟩

definition *rel-track-coin-use* :: (*'a* × *nat*) *random-alg* ⇒ *'a random-alg* ⇒ *bool* **where**
rel-track-coin-use q p ⟷ *q* = *track-coin-use p*

lemma *admissible-rel-track-coin-use*:
ccpo.admissible (*prod-lub lub-ra lub-ra*) (*rel-prod ord-ra ord-ra*) (*case-prod rel-track-coin-use*)
(**is** *ccpo.admissible ?lub ?ord ?P*)
⟨*proof*⟩

lemma *admissible-rel-track-coin-use-cont* [*cont-intro*]:
fixes *ord*
shows ⟦ *mcont lub ord lub-ra ord-ra f*; *mcont lub ord lub-ra ord-ra g* ⟧

$\Longrightarrow$ *ccpo.admissible lub ord* ($\lambda x.$ *rel-track-coin-use* (f x) (g x))
⟨*proof*⟩

lemma *mcont-track-coin-use*:
mcont lub-ra ord-ra lub-ra ord-ra track-coin-use
⟨*proof*⟩

lemmas *mcont2mcont-track-coin-use* = *mcont-track-coin-use*[*THEN random-alg-pf.mcont2mcont*]

context includes *lifting-syntax*
begin

lemma *fixp-track-coin-use-parametric*[*transfer-rule*]:
assumes f: $\bigwedge x.$ *mono-ra* ($\lambda f.$ F f x)
and g: $\bigwedge x.$ *mono-ra* ($\lambda f.$ G f x)
and *param*: ((A ===> *rel-track-coin-use*) ===> A ===> *rel-track-coin-use*) F G
shows (A ===> *rel-track-coin-use*) (*random-alg-pf.fixp-fun* F) (*random-alg-pf.fixp-fun* G)
⟨*proof*⟩

lemma *return-ra-tranfer*[*transfer-rule*]: ((=) ===> *rel-track-coin-use*) *return-tra return-ra*
⟨*proof*⟩

lemma *bind-ra-tranfer*[*transfer-rule*]:
(*rel-track-coin-use* ===> ((=) ===> *rel-track-coin-use*) ===> *rel-track-coin-use*) *bind-tra bind-ra*
⟨*proof*⟩

lemma *coin-ra-tranfer*[*transfer-rule*]:
rel-track-coin-use coin-tra coin-ra
⟨*proof*⟩

end

end

7 Tracking SPMFs

This section introduces tracking SPMFs — this is a resource monad on top of SPMFs, we also introduce the Scott-continous monad morphism *tspmf-of-ra*, with which it is possible to reason about the joint-distribution of a randomized algorithm's result and used coin-flips.

An example application of the results in this theory can be found in Section 8.

theory *Tracking-SPMF*
imports *Tracking-Randomized-Algorithm*
begin

type-synonym *'a tspmf* = (*'a* × *nat*) *spmf*

definition *return-tspmf* :: *'a* ⇒ *'a tspmf* **where**
return-tspmf x = *return-spmf* (x,*0*)

definition *coin-tspmf* :: *bool tspmf* **where**
coin-tspmf = *pair-spmf coin-spmf* (*return-spmf 1*)

definition *bind-tspmf* :: *'a tspmf* ⇒ (*'a* ⇒ *'b tspmf*) ⇒ *'b tspmf* **where**
bind-tspmf f g = *bind-spmf f* ($\lambda(r,c).$ *map-spmf* (*apsnd* ((+) c)) (g r))

adhoc-overloading *Monad-Syntax.bind bind-tspmf*

Monad laws:

lemma *return-bind-tspmf*:
  bind-tspmf (return-tspmf x) g = g x
  ⟨*proof*⟩

lemma *bind-tspmf-assoc*:
  bind-tspmf (bind-tspmf f g) h = bind-tspmf f (λx. bind-tspmf (g x) h)
  ⟨*proof*⟩

lemma *bind-return-tspmf*:
  bind-tspmf m return-tspmf = m
  ⟨*proof*⟩

lemma *bind-mono-tspmf-aux*:
  assumes *ord-spmf (=) f1 f2* $\bigwedge$*y. ord-spmf (=) (g1 y) (g2 y)*
  shows *ord-spmf (=) (bind-tspmf f1 g1) (bind-tspmf f2 g2)*
  ⟨*proof*⟩

lemma *bind-mono-tspmf [partial-function-mono]*:
  assumes *mono-spmf B* **and** $\bigwedge$*y. mono-spmf (C y)*
  shows *mono-spmf (λf. bind-tspmf (B f) (λy. C y f))*
  ⟨*proof*⟩

definition *ord-tspmf* :: *'a tspmf ⇒ 'a tspmf ⇒ bool* **where**
  ord-tspmf = ord-spmf (λx y. fst x = fst y ∧ snd x ≥ snd y)

bundle *ord-tspmf-notation*
begin
  notation *ord-tspmf* ((-/ $\leq_R$ -) [*51*, *51*] *50*)
end

bundle *no-ord-tspmf-notation*
begin
  no-notation *ord-tspmf* ((-/ $\leq_R$ -) [*51*, *51*] *50*)
end

unbundle *ord-tspmf-notation*

definition *coin-usage-of-tspmf* :: *'a tspmf ⇒ enat pmf*
  where *coin-usage-of-tspmf = map-pmf (λx. case x of None ⇒ ∞ | Some y ⇒ enat (snd y))*

definition *expected-coin-usage-of-tspmf* :: *'a tspmf ⇒ ennreal*
  where
    expected-coin-usage-of-tspmf p = ($\int^+$*x. x ∂(map-pmf ennreal-of-enat (coin-usage-of-tspmf p)))*

definition *expected-coin-usage-of-ra* **where**
  expected-coin-usage-of-ra p = $\int^+$*x. x ∂(map-pmf ennreal-of-enat (coin-usage-of-ra p))*

definition *result* :: *'a tspmf ⇒ 'a spmf*
  where *result = map-spmf fst*

lemma *coin-usage-of-tspmf-alt-def*:
  coin-usage-of-tspmf p = map-pmf (λx. case x of None ⇒ ∞ | Some y ⇒ enat y) (map-spmf snd p)
  ⟨*proof*⟩

lemma *coin-usage-of-tspmf-bind-return*:
coin-usage-of-tspmf (*bind-tspmf* *f* (λx. *return-tspmf* (*g* *x*))) = (*coin-usage-of-tspmf* *f*)
⟨*proof*⟩

lemma *coin-usage-of-tspmf-mono*:
assumes *ord-tspmf* *p* *q*
shows *measure* (*coin-usage-of-tspmf* *p*) $\{..k\}$ $\leq$ *measure* (*coin-usage-of-tspmf* *q*) $\{..k\}$
⟨*proof*⟩

lemma *coin-usage-of-tspmf-mono-rev*:
assumes *ord-tspmf* *p* *q*
shows *measure* (*coin-usage-of-tspmf* *q*) $\{x.\ x > k\}$ $\leq$ *measure* (*coin-usage-of-tspmf* *p*) $\{x.\ x > k\}$
(**is** *?L* $\leq$ *?R*)
⟨*proof*⟩

lemma *expected-coin-usage-of-tspmf*:
expected-coin-usage-of-tspmf *p* = ($\sum k$. *ennreal* (*measure* (*coin-usage-of-tspmf* *p*) $\{x.\ x >$ *enat* $k\}$)) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *ord-tspmf-min*: *ord-tspmf* (*return-pmf* *None*) *p*
⟨*proof*⟩

lemma *ord-tspmf-refl*: *ord-tspmf* *p* *p*
⟨*proof*⟩

lemma *ord-tspmf-trans*[*trans*]:
assumes *ord-tspmf* *p* *q* *ord-tspmf* *q* *r*
shows *ord-tspmf* *p* *r*
⟨*proof*⟩

lemma *ord-tspmf-map-spmf*:
assumes $\bigwedge x.\ x \leq f\ x$
shows *ord-tspmf* (*map-spmf* (*apsnd* *f*) *p*) *p*
⟨*proof*⟩

lemma *ord-tspmf-bind-pmf*:
assumes $\bigwedge x$. *ord-tspmf* (*f* *x*) (*g* *x*)
shows *ord-tspmf* (*bind-pmf* *p* *f*) (*bind-pmf* *p* *g*)
⟨*proof*⟩

lemma *ord-tspmf-bind-tspmf*:
assumes $\bigwedge x$. *ord-tspmf* (*f* *x*) (*g* *x*)
shows *ord-tspmf* (*bind-tspmf* *p* *f*) (*bind-tspmf* *p* *g*)
⟨*proof*⟩

definition *use-coins* :: *nat* $\Rightarrow$ *'a tspmf* $\Rightarrow$ *'a tspmf*
where *use-coins* *k* = *map-spmf* (*apsnd* ((+) *k*))

lemma *use-coins-add*:
use-coins *k* (*use-coins* *s* *f*) = *use-coins* (*k*+*s*) *f*
⟨*proof*⟩

lemma *coin-tspmf-split*:
fixes *f* :: *bool* $\Rightarrow$ *'b tspmf*
shows (*coin-tspmf* $\ggg$ *f*) = *use-coins* *1* (*coin-spmf* $\ggg$ *f*)

⟨*proof*⟩

lemma *ord-tspmf-use-coins*:
ord-tspmf (*use-coins k p*) *p*
⟨*proof*⟩

lemma *ord-tspmf-use-coins-2*:
assumes *ord-tspmf p q*
shows *ord-tspmf* (*use-coins k p*) (*use-coins k q*)
⟨*proof*⟩

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

lemma *result-bind*:
result (*bind-tspmf f g*) = *result f* $\gg=$ ($\lambda x.$ *result* (*g x*))
⟨*proof*⟩

lemma *result-return*:
result (*return-tspmf x*) = *return-spmf x*
⟨*proof*⟩

lemma *result-coin*:
result (*coin-tspmf*) = *coin-spmf*
⟨*proof*⟩

definition *tspmf-of-ra* :: *'a random-alg* ⇒ *'a tspmf* **where**
tspmf-of-ra = *spmf-of-ra* ∘ *track-coin-use*

lemma *tspmf-of-ra-coin*: *tspmf-of-ra coin-ra* = *coin-tspmf*
⟨*proof*⟩

lemma *tspmf-of-ra-return*: *tspmf-of-ra* (*return-ra x*) = *return-tspmf x*
⟨*proof*⟩

lemma *tspmf-of-ra-bind*:
tspmf-of-ra (*bind-ra m f*) = *bind-tspmf* (*tspmf-of-ra m*) ($\lambda x.$ *tspmf-of-ra* (*f x*))
⟨*proof*⟩

lemmas *tspmf-of-ra-simps* = *tspmf-of-ra-bind tspmf-of-ra-return tspmf-of-ra-coin*

lemma *tspmf-of-ra-mono*:
assumes *ord-ra f g*
shows *ord-spmf* (=) (*tspmf-of-ra f*) (*tspmf-of-ra g*)
⟨*proof*⟩

lemma *tspmf-of-ra-lub*:
assumes *Complete-Partial-Order.chain ord-ra A*
shows *tspmf-of-ra* (*lub-ra A*) = *lub-spmf* (*tspmf-of-ra* ‘ *A*) (**is** *?L* = *?R*)
⟨*proof*⟩

definition *rel-tspmf-of-ra* :: *'a tspmf* ⇒ *'a random-alg* ⇒ *bool* **where**
rel-tspmf-of-ra q p ⟷ *q* = *tspmf-of-ra p*

lemma *admissible-rel-tspmf-of-ra*:
ccpo.admissible (*prod-lub lub-spmf lub-ra*) (*rel-prod* (*ord-spmf* (=)) *ord-ra*) (*case-prod rel-tspmf-of-ra*)

(**is** *ccpo.admissible ?lub ?ord ?P*)
⟨*proof*⟩

lemma *admissible-rel-tspmf-of-ra-cont* [*cont-intro*]:
fixes *ord*
shows ⟦ *mcont lub ord lub-spmf* (*ord-spmf* (=)) *f*; *mcont lub ord lub-ra ord-ra g* ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *rel-tspmf-of-ra* (*f x*) (*g x*))
⟨*proof*⟩

lemma *mcont-tspmf-of-ra*:
mcont lub-ra ord-ra lub-spmf (*ord-spmf* (=)) *tspmf-of-ra*
⟨*proof*⟩

lemmas *mcont2mcont-tspmf-of-ra* = *mcont-tspmf-of-ra*[*THEN spmf.mcont2mcont*]

context includes *lifting-syntax*
begin

lemma *fixp-rel-tspmf-of-ra-parametric*[*transfer-rule*]:
assumes *f*: ⋀*x*. *mono-spmf* (λ*f*. *F f x*)
and *g*: ⋀*x*. *mono-ra* (λ*f*. *G f x*)
and *param*: ((*A* ===> *rel-tspmf-of-ra*) ===> *A* ===> *rel-tspmf-of-ra*) *F G*
shows (*A* ===> *rel-tspmf-of-ra*) (*spmf.fixp-fun F*) (*random-alg-pf.fixp-fun G*)
⟨*proof*⟩

lemma *return-ra-tranfer*[*transfer-rule*]: ((=) ===> *rel-tspmf-of-ra*) *return-tspmf return-ra*
⟨*proof*⟩

lemma *bind-ra-tranfer*[*transfer-rule*]:
(*rel-tspmf-of-ra* ===> ((=) ===> *rel-tspmf-of-ra*) ===> *rel-tspmf-of-ra*) *bind-tspmf bind-ra*
⟨*proof*⟩

lemma *coin-ra-tranfer*[*transfer-rule*]:
rel-tspmf-of-ra coin-tspmf coin-ra
⟨*proof*⟩

end

lemma *spmf-of-tspmf*:
result (*tspmf-of-ra f*) = *spmf-of-ra f*
⟨*proof*⟩

lemma *coin-usage-of-tspmf-correct*:
coin-usage-of-tspmf (*tspmf-of-ra p*) = *coin-usage-of-ra p* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *expected-coin-usage-of-tspmf-correct*:
expected-coin-usage-of-tspmf (*tspmf-of-ra p*) = *expected-coin-usage-of-ra p*
⟨*proof*⟩

end

8 Dice Roll

theory *Dice-Roll*
imports *Tracking-SPMF*
begin

The following is a dice roll algorithm for an arbitrary number of sides n. Besides correctness we also show that the expected number of coin flips is at most $log\ 2\ n + 2$. It is a specialization of the algorithm presented by Hao and Hoshi [4]. [3]

lemma *floor-le-ceil-minus-one*:
  fixes $x\ y$ :: *real*
  shows $x < y \Longrightarrow \lfloor x \rfloor \leq \lceil y \rceil - 1$
  ⟨*proof*⟩

lemma *combine-spmf-set-coin-spmf*:
  fixes f :: *nat* ⇒ *'a spmf*
  fixes k :: *nat*
  shows *pmf-of-set* $\{..<2\hat{}k\}$ ≫= ($\lambda l.$ *coin-spmf* ≫=($\lambda b.$ f ($2*l+$ *of-bool* b))) =
    pmf-of-set $\{..<2\hat{}(k+1)\}$ ≫= f (**is** $?L = ?R$)
⟨*proof*⟩

lemma *count-ints-in-range*:
  real (*card* $\{x.$ *of-int* $x \in \{u..v\}\}$) $\geq v-u-1$ (**is** $?L \geq ?R$)
⟨*proof*⟩

partial-function (*random-alg*) *dice-roll-step-ra* :: *real* ⇒ *real* ⇒ *int random-alg*
  where *dice-roll-step-ra* $l\ h$ = (
    if $\lfloor l \rfloor = \lceil l+h \rceil - 1$ *then*
      return-ra $\lfloor l \rfloor$
    else
      do { $b \leftarrow$ *coin-ra*; *dice-roll-step-ra* (l + ($h/2$) ∗ *of-bool* b) ($h/2$) }
)

definition *dice-roll-ra* n = *map-ra nat* (*dice-roll-step-ra* 0 (*of-nat* n))

partial-function (*spmf*) *drs-tspmf* :: *real* ⇒ *real* ⇒ *int tspmf*
  where *drs-tspmf* $l\ h$ = (
    if $\lfloor l \rfloor = \lceil l+h \rceil - 1$ *then*
      return-tspmf $\lfloor l \rfloor$
    else
      do { $b \leftarrow$ *coin-tspmf*; *drs-tspmf* (l + ($h/2$) ∗ *of-bool* b) ($h/2$) }
)

definition *dice-roll-tspmf* n = *drs-tspmf* 0 (*of-nat* n) ≫= ($\lambda x.$ *return-tspmf* (*nat* x))

lemma *drs-tspmf*: *drs-tspmf* $l\ u$ = *tspmf-of-ra* (*dice-roll-step-ra* $l\ u$)
⟨*proof*⟩
  include *lifting-syntax*
  ⟨*proof*⟩

lemma *dice-roll-ra-tspmf*: *tspmf-of-ra* (*dice-roll-ra* n) = *dice-roll-tspmf* n
  ⟨*proof*⟩

lemma *dice-roll-step-tspmf-lb*:
  assumes $h > 0$
  shows *coin-tspmf* ≫= ($\lambda b.$ *drs-tspmf* (l + ($h/2$) ∗ *of-bool* b) ($h/2$)) $\leq_R$ *drs-tspmf* $l\ h$
⟨*proof*⟩

abbreviation *coins* $k \equiv$ *pmf-of-set* $\{..<(2::nat)\hat{}k\}$

lemma *dice-roll-step-tspmf-expand*:
  assumes $h > 0$

[3] An interesting alternative algorithm, which we did not formalized here, has been introduced by Lambruso [7].

shows *coins k* $\gg=$ ($\lambda l.$ *use-coins k* (*drs-tspmf* (*real l*$*$*h*) *h*)) $\leq_R$ *drs-tspmf 0* ($h*2\hat{\ }k$)
⟨*proof*⟩

lemma *dice-roll-step-tspmf-approx*:
fixes k :: *nat*
assumes $h > (0::real)$
defines $f \equiv (\lambda l.$ *if* $\lfloor l*h \rfloor = \lceil (l+1)*h \rceil - 1$ *then Some* ($\lfloor l*h \rfloor$,k) *else None*)
shows *map-pmf f* (*coins k*) $\leq_R$ *drs-tspmf 0* ($h*2\hat{\ }k$) (**is** $?L \leq_R ?R$)
⟨*proof*⟩

lemma *dice-roll-step-spmf-approx*:
fixes k :: *nat*
assumes $h > (0::real)$
defines $f \equiv (\lambda l.$ *if* $\lfloor l*h \rfloor = \lceil (l+1)*h \rceil - 1$ *then Some* ($\lfloor l*h \rfloor$) *else None*)
shows *ord-spmf* (=) (*map-pmf f* (*coins k*)) (*result* (*drs-tspmf 0* ($h*2\hat{\ }k$)))
(**is** *ord-spmf - ?L ?R*)
⟨*proof*⟩

lemma *spmf-dice-roll-step-lb*:
assumes $j < n$
shows *spmf* (*result* (*drs-tspmf 0* (*of-nat n*))) (*of-nat j*) $\geq 1/n$ (**is** $?L \geq ?R$)
⟨*proof*⟩

lemma *dice-roll-correct-aux*:
assumes $n > 0$
shows *result* (*drs-tspmf 0* (*of-nat n*)) = *spmf-of-set* $\{0..<n\}$
⟨*proof*⟩

theorem *dice-roll-correct*:
assumes $n > 0$
shows
result (*dice-roll-tspmf n*) = *spmf-of-set* $\{..<n\}$ (**is** $?L = ?R$)
spmf-of-ra (*dice-roll-ra n*) = *spmf-of-set* $\{..<n\}$
⟨*proof*⟩

lemma *dice-roll-consumption-bound*:
assumes $n > 0$
shows *measure* (*coin-usage-of-tspmf* (*dice-roll-tspmf n*)) $\{x.\ x >$ *enat k* $\} \leq$ *real* $n/2\hat{\ }k$
(**is** $?L \leq ?R$)
⟨*proof*⟩

lemma *dice-roll-expected-consumption-aux*:
assumes $n > (0::nat)$
shows *expected-coin-usage-of-tspmf* (*dice-roll-tspmf n*) $\leq$ *log 2 n* + 2 (**is** $?L \leq ?R$)
⟨*proof*⟩

theorem *dice-roll-coin-usage*:
assumes $n > (0::nat)$
shows *expected-coin-usage-of-ra* (*dice-roll-ra n*) $\leq$ *log 2 n* + 2 (**is** $?L \leq ?R$)
⟨*proof*⟩

end

9 A Pseudo-random Number Generator

In this section we introduce a PRG, that can be used to generate random bits. It is an implementation of O'Neil's Permuted congruential generator [9] (specifically PCG-XSH-

RR). In empirical tests it ranks high [2, 10] while having a low implementation complexity. This is for easy testing purposes only, the generated code can be run with any source of random bits.

theory *Permuted-Congruential-Generator*
  imports
    HOL−Library.Word
    Coin-Space
begin

The following are default constants from the reference implementation [8].

definition *pcg-mult* :: *64 word*
  where *pcg-mult* = *6364136223846793005*
definition *pcg-increment* :: *64 word*
  where *pcg-increment* = *1442695040888963407*

fun *pcg-rotr* :: *32 word* ⇒ *nat* ⇒ *32 word*
  where *pcg-rotr x r* = *Bit-Operations.or* (*drop-bit r x*) (*push-bit* (*32*−*r*) *x*)

fun *pcg-step* :: *64 word* ⇒ *64 word*
  where *pcg-step state* = *state* ∗ *pcg-mult* + *pcg-increment*

Based on [9, Section 6.3.1]:

fun *pcg-get* :: *64 word* ⇒ *32 word*
  where *pcg-get state* =
    (*let count* = *unsigned* (*drop-bit 59 state*);
         x   = *xor* (*drop-bit 18 state*) *state*
      in pcg-rotr (*ucast* (*drop-bit 27 x*)) *count*)

fun *pcg-init* :: *64 word* ⇒ *64 word*
  where *pcg-init seed* = *pcg-step* (*seed* + *pcg-increment*)

definition *to-bits* :: *32 word* ⇒ *bool list*
  where *to-bits x* = *map* (λ*k*. *bit x k*) [*0*..<*32*]

definition *random-coins*
  where *random-coins seed* = *build-coin-gen* (*to-bits* ∘ *pcg-get*) *pcg-step* (*pcg-init seed*)

end

10 Basic Randomized Algorithms

This section introduces a few randomized algorithms for well-known distributions. These both serve as building blocks for more complex algorithms and as examples describing how to use the framework.

theory *Basic-Randomized-Algorithms*
  imports
    Randomized-Algorithm
    Probabilistic-While.Bernoulli
    Probabilistic-While.Geometric
    Permuted-Congruential-Generator
begin

A simple example: Here we define a randomized algorithm that can sample uniformly from *pmf-of-set* $\{..<2^n\}$. (The same problem for general ranges is discussed in Section 8).

fun *binary-dice-roll* :: *nat* ⇒ *nat random-alg*

where
binary-dice-roll 0 = return-ra 0 |
binary-dice-roll (*Suc n*) =
do { *h* ← *binary-dice-roll n*;
c ← *coin-ra*;
return-ra (*of-bool c* + *2* ∗ *h*)
}

Because the algorithm terminates unconditionally it is easy to verify that *binary-dice-roll* terminates almost surely:

lemma *binary-dice-roll-terminates*: *terminates-almost-surely* (*binary-dice-roll n*)
⟨*proof*⟩

The corresponding PMF can be written as:

fun *binary-dice-roll-pmf* :: *nat* ⇒ *nat pmf*
where
binary-dice-roll-pmf 0 = return-pmf 0 |
binary-dice-roll-pmf (*Suc n*) =
do { *h* ← *binary-dice-roll-pmf n*;
c ← *coin-pmf*;
return-pmf (*of-bool c* + *2* ∗ *h*)
}

To verify that the distribution of the result of *binary-dice-roll* is *binary-dice-roll-pmf* we can rely on the *pmf-of-ra-simps* simp rules and the *terminates-almost-surely-intros* introduction rules:

lemma *pmf-of-ra* (*binary-dice-roll n*) = *binary-dice-roll-pmf n*
⟨*proof*⟩

Let us now consider an algorithm that does not terminate unconditionally but just almost surely:

partial-function (*random-alg*) *binary-geometric* :: *nat* ⇒ *nat random-alg*
where
binary-geometric n =
do { *c* ← *coin-ra*;
if c then (*return-ra n*) *else binary-geometric* (*n+1*)
}

This is necessary for running randomized algorithms defined with the **partial-function** directive:

declare *binary-geometric.simps*[*code*]

In this case, we need to map to an SPMF:

partial-function (*spmf*) *binary-geometric-spmf* :: *nat* ⇒ *nat spmf*
where
binary-geometric-spmf n =
do { *c* ← *coin-spmf*;
if c then (*return-spmf n*) *else binary-geometric-spmf* (*n+1*)
}

We use the transfer rules for *spmf-of-ra* to show the correspondence:

lemma *binary-geometric-ra-correct*:
spmf-of-ra (*binary-geometric x*) = *binary-geometric-spmf x*
⟨*proof*⟩
include *lifting-syntax*
⟨*proof*⟩

Bernoulli distribution: For this example we show correspondence with the already existing definition of *bernoulli* SPMF.

partial-function (*random-alg*) *bernoulli-ra* :: *real* ⇒ *bool random-alg* **where**
  bernoulli-ra p = *do* {
    b ← *coin-ra*;
    if b then return-ra ($p \geq 1 / 2$)
    else if $p < 1 / 2$ *then bernoulli-ra* ($2 * p$)
    else bernoulli-ra ($2 * p - 1$)
  }

declare *bernoulli-ra.simps*[*code*]

The following is a different technique to show equivalence of an SPMF with a randomized algorithm. It only works if the SPMF has weight 1. First we show that the SPMF is a lower bound:

lemma *bernoulli-ra-correct-aux*: *ord-spmf* (=) (*bernoulli x*) (*spmf-of-ra* (*bernoulli-ra x*))
⟨*proof*⟩

Then relying on the fact that the SPMF has weight one, we can derive equivalence:

lemma *bernoulli-ra-correct*: *bernoulli x* = *spmf-of-ra* (*bernoulli-ra x*)
  ⟨*proof*⟩

Because *bernoulli p* is a lossless SPMF equivalent to *spmf-of-pmf* (*bernoulli-pmf p*) it is also possible to express the above, without referring to SPMFs:

lemma
  terminates-almost-surely (*bernoulli-ra p*)
  bernoulli-pmf p = *pmf-of-ra* (*bernoulli-ra p*)
  ⟨*proof*⟩

context
  includes *lifting-syntax*
begin

lemma *bernoulli-ra-transfer* [*transfer-rule*]:
  ((=) ===> *rel-spmf-of-ra*) *bernoulli bernoulli-ra*
  ⟨*proof*⟩

end

Using the randomized algorithm for the Bernoulli distribution, we can introduce one for the general geometric distribution:

partial-function (*random-alg*) *geometric-ra* :: *real* ⇒ *nat random-alg* **where**
  geometric-ra p = *do* {
    b ← *bernoulli-ra p*;
    if b then return-ra 0 else map-ra ((+) *1*) (*geometric-ra p*)
  }
declare *geometric-ra.simps*[*code*]

lemma *geometric-ra-correct*: *spmf-of-ra* (*geometric-ra x*) = *geometric-spmf x*
⟨*proof*⟩
  include *lifting-syntax*
  ⟨*proof*⟩

Replication of a distribution

fun *replicate-ra* :: *nat* ⇒ *'a random-alg* ⇒ *'a list random-alg*
  where

  replicate-ra 0 f = *return-ra* [] |
  replicate-ra (*Suc n*) *f* = *do* { *xh* ← *f*; *xt* ← *replicate-ra n f*; *return-ra* (*xh*#*xt*) }

fun *replicate-spmf* :: *nat* ⇒ *'a spmf* ⇒ *'a list spmf*
 where
  replicate-spmf 0 f = *return-spmf* [] |
  replicate-spmf (*Suc n*) *f* = *do* { *xh* ← *f*; *xt* ← *replicate-spmf n f*; *return-spmf* (*xh*#*xt*) }

lemma *replicate-ra-correct*: *spmf-of-ra* (*replicate-ra n f*) = *replicate-spmf n* (*spmf-of-ra f*)
 ⟨*proof*⟩

lemma *replicate-spmf-of-pmf*: *replicate-spmf n* (*spmf-of-pmf f*) = *spmf-of-pmf* (*replicate-pmf n f*)
 ⟨*proof*⟩

Binomial distribution

definition *binomial-ra* :: *nat* ⇒ *real* ⇒ *nat random-alg*
 where *binomial-ra n p* = *map-ra* (*length* ∘ *filter id*) (*replicate-ra n* (*bernoulli-ra p*))

lemma
 assumes $p \in \{0..1\}$
 shows *spmf-of-ra* (*binomial-ra n p*) = *spmf-of-pmf* (*binomial-pmf n p*)
⟨*proof*⟩

Running randomized algorithms: Here we use the PRG introduced in Section 9.

value *run-ra* (*binomial-ra 10 0.5*) (*random-coins 42*)

value *run-ra* (*replicate-ra 20* (*bernoulli-ra 0.3*)) (*random-coins 42*)

end

References

[1] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[2] K. Bhattacharjee, K. Maity, and S. Das. A search for good pseudo-random number generators: Survey and empirical studies. *Comput. Sci. Rev.*, 45:100471, 2018.

[3] D. H. Fremlin. *Measure theory*, volume 4. Torres Fremlin, 2000.

[4] T. S. Hao and M. Hoshi. Interval algorithm for random number generation. *IEEE Transactions on Information Theory*, 43(2):599–611, 1997.

[5] J. Hurd. Formal verification of probabilistic algorithms. Technical report, University of Cambridge, Computer Laboratory, 2003.

[6] G. Hutton and E. Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437444, 1998.

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

[8] M. E. O'Neill. PCG random number generation, minimal C edition.

[9] M. E. O'Neill. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, Sept. 2014.

[10] M. Singh, P. Singh, and P. Kumar. An empirical study of non-cryptographically secure pseudorandom number generators. In *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*, pages 1–6, 2020.