

Executable Randomized Algorithms

Emin Karayel and Manuel Eberl

February 6, 2026

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	5
4	Randomized Algorithms (Internal Representation)	22
5	Randomized Algorithms	42
5.1	Almost surely terminating randomized algorithms	51
6	Tracking Randomized Algorithms	52
7	Tracking SPMFs	61
8	Dice Roll	67
9	A Pseudo-random Number Generator	76
10	Basic Randomized Algorithms	77

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.

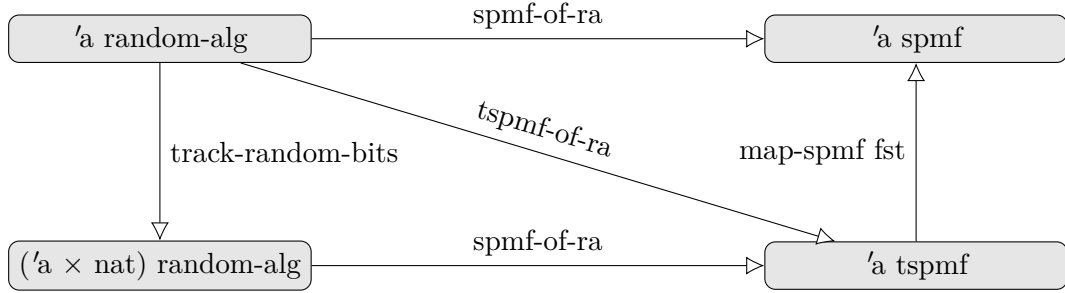


Figure 1: Scott-continuous monad morphisms verified in this work.

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 may be 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:

- *spmof-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)
- *tspmof-of-ra*: Morphism 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 chains¹ 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 $op x y \in F$

using *assms unfolding op-stable-def* **by** *auto*

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*: $\bigwedge a. a \in A \implies open a$

assumes *ud*: *op-stable* $(\bigcup) A$

shows *emeasure* $M (\bigcup A) = (SUP a \in A. emeasure M a)$ (**is** $?L = ?R$)

proof (*cases* $A \neq \{\}$)

case *True*

have *open* $(\bigcup A)$ **using** *of* **by** *auto*

hence $\bigcup A \in sets borel$ **by** *simp*

hence *usets*: $\bigcup A \in sets M$ **using** *assms(1)* **by** *simp*

have $0:a \in sets borel$ **if** $a \in A$ **for** a

using *of that* **by** *simp*

have $1:\bigcup T \in A$ **if** *finite* $T T \neq \{\}$ $T \subseteq A$ **for** T

using *that op-stableD[OF ud]* **by** (*induction T rule:finite-ne-induct*) *auto*

have $2:emeasure M K \leq ?R$ **if** *K-def*: *compact* $K K \subseteq \bigcup A$ **for** K

proof (*cases* $K \neq \{\}$)

case *True*

obtain T **where** *T-def*: $K \subseteq \bigcup T T \subseteq A$ *finite* T

using *compactE[OF K-def of]* *that* **by** *metis*

have *T-ne*: $T \neq \{\}$ **using** *T-def(1)* *True* **by** *auto*

define t **where** $t = \bigcup T$

have *t-in*: $t \in A$

unfolding *t-def* **by** (*intro 1 T-ne T-def*)

have $K \subseteq t$

unfolding *t-def* **using** *T-def* **by** *simp*

hence *emeasure* $M K \leq emeasure M t$

using $0 sb t-in$ **by** (*intro emeasure-mono*) *auto*

also have $\dots \leq ?R$

using *t-in* **by** (*intro cSup-upper*) *auto*

finally show *?thesis*

¹More generally families closed under pairwise unions.

by *simp*
 next
 case *False*
 hence $K = \{\}$ by *simp*
 thus *?thesis* by *simp*
 qed

 have $?L = (\text{SUP } K \in \{K. K \subseteq \bigcup A \wedge \text{compact } K\}, \text{emeasure } M K)$
 using *usets unfolding sb* by (*intro inner-regular[OF sb fin]*) *auto*
 also have $\dots \leq ?R$
 using *2* by (*intro cSup-least*) *auto*
 finally have $?L \leq ?R$ by *simp*
 moreover have $\text{emeasure } M a \leq \text{emeasure } M (\bigcup A)$ if $a \in A$ for a
 using *that* by (*intro emeasure-mono usets*) *auto*
 hence $?R \leq ?L$
 using *True* by (*intro cSup-least*) *auto*
 ultimately show *?thesis* by *auto*
 next
 case *False*
 thus *?thesis* by (*simp add:bot-ennreal*)
 qed

lemma *chain-imp-union-stable*:

assumes *Complete-Partial-Order.chain* (\subseteq) F
 shows *op-stable* (\cup) F

proof –

have $x \cup y \in F$ if $x \in F$ $y \in F$ for x y

proof (*cases x ⊆ y*)

case *True*

then show *?thesis* using *that sup.absorb2[OF True]* by *simp*

next

case *False*

hence $0 : y \subseteq x$

using *assms that unfolding Complete-Partial-Order.chain-def* by *auto*

then show *?thesis* using *that sup.absorb1[OF 0]* by *simp*

qed

thus *?thesis*

unfolding *op-stable-def* by *auto*

qed

theorem *tau-additivity*:

fixes $M :: 'a :: \{\text{second-countable-topology, complete-space}\}$ *measure*

assumes *sb*: $\bigwedge x. \text{open } x \implies x \in \text{sets } M$

assumes *fin*: $\text{emeasure } M (\text{space } M) \neq \infty$

assumes *of*: $\bigwedge a. a \in A \implies \text{open } a$

assumes *ud*: *op-stable* (\cup) A

shows $\text{emeasure } M (\bigcup A) = (\text{SUP } a \in A. \text{emeasure } M a)$ (is $?L = ?R$)

proof –

have $UNIV \in \text{sets } M$

using *open-UNIV sb* by *auto*

hence $\text{space-}M[\text{simp}]: \text{space } M = UNIV$

using *sets.sets-into-space* by *blast*

have *id-borel*: $(\lambda x. x) \in M \rightarrow_M \text{borel}$

using *sb* by (*intro borel-measurableI*) *auto*

have *open* ($\bigcup A$) using *of* by *auto*

hence *usets*: $(\bigcup A) \in \text{sets borel}$ by *simp*

```

define N where N = distr M borel ( $\lambda x. x$ )
have sets-N: sets N = sets borel
  unfolding N-def by simp
have fin-N: emeasure N (space N)  $\neq \infty$ 
  using fin id-borel unfolding N-def
  by (subst emeasure-distr) auto

have ?L = emeasure N ( $\bigcup A$ )
  unfolding N-def by (subst emeasure-distr[OF id-borel usets]) auto
also have ... = (SUP a  $\in A$ . emeasure N a)
  by (intro tau-additivity-aux sets-N of ud fin-N) auto
also have ... = (SUP a  $\in A$ . emeasure M ( $(\lambda x. x) - 'a \cap \text{space } M$ ))
  unfolding N-def using of
  by (intro arg-cong[where f=Sup] image-cong emeasure-distr id-borel) auto
also have ... = ?R by simp
finally show ?thesis by simp
qed

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

  lemma stream-eq-iff:
    assumes  $\bigwedge i. x !! i = y !! i$ 
    shows  $x = y$ 
  proof -
    have  $x = \text{smap id } x$  by (simp add: stream.map-id)
    also have ... =  $y$  using assms unfolding smap-alt by auto
    finally show ?thesis by simp
  qed

```

Notation for the discrete σ -algebra:

```

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

open-bundle discrete-sigma-algebra-syntax
begin
notation discrete-sigma-algebra ( $\langle \mathcal{D} \rangle$ )
end

```

```

lemma map-prod-measurable[measurable]:
  assumes  $f \in M \rightarrow_M M'$ 
  assumes  $g \in N \rightarrow_M N'$ 
  shows  $\text{map-prod } f g \in M \otimes_M N \rightarrow_M M' \otimes_M N'$ 
  using assms by (subst measurable-pair-iff) simp

```

```

lemma measurable-sigma-sets-with-exception:
  fixes  $f :: 'a \Rightarrow 'b :: \text{countable}$ 

```

assumes $\bigwedge x. x \neq d \implies f - \{x\} \cap \text{space } M \in \text{sets } M$
shows $f \in M \rightarrow_M \text{count-space } UNIV$
proof –
define $A :: 'b \text{ set set where } A = (\lambda x. \{x\}) - UNIV$

have $0: \text{sets } (\text{count-space } UNIV) = \text{sigma-sets } (UNIV :: 'b \text{ set}) A$
unfolding $A\text{-def}$ **by** $(\text{subst sigma-sets-singletons}) \text{ auto}$

have $1: f - \{x\} \cap \text{space } M \in \text{sets } M$ **for** x
proof $(\text{cases } x = d)$
case $True$
have $f - \{d\} \cap \text{space } M = \text{space } M - (\bigcup y \in UNIV - \{d\}. f - \{y\} \cap \text{space } M)$
by $(\text{auto simp add:set-eq-iff})$
also have $\dots \in \text{sets } M$
using assms
by $(\text{intro sets.compl-sets sets.countable-UN}) \text{ auto}$
finally show $?thesis$
using $True$ **by** simp

next
case $False$
then show $?thesis$ **using** assms **by** simp
qed

hence $1: \bigwedge y. y \in A \implies f - y \cap \text{space } M \in \text{sets } M$
unfolding $A\text{-def}$ **by** auto

thus $?thesis$
by $(\text{intro measurable-sigma-sets}[OF 0]) \text{ simp-all}$
qed

lemma $\text{restr-empty-eq: restrict-space } M \{\} = \text{restrict-space } N \{\}$
by $(\text{intro measure-eqI}) (\text{auto simp add:sets-restrict-space})$

lemma $(\text{in prob-space}) \text{distr-stream-space-snth} [\text{simp}]$:
assumes $\text{sets } M = \text{sets } N$
shows $\text{distr } (\text{stream-space } M) N (\lambda xs. \text{snth } xs \ n) = M$
proof –
have $\text{distr } (\text{stream-space } M) N (\lambda xs. \text{snth } xs \ n) = \text{distr } (\text{stream-space } M) M (\lambda xs. \text{snth } xs \ n)$
by $(\text{rule distr-cong}) (\text{use assms in auto})$
also have $\dots = \text{distr } (Pi_M UNIV (\lambda i. M)) M (\lambda f. f \ n)$
by $(\text{subst stream-space-eq-distr, subst distr-distr}) (\text{auto simp: to-stream-def o-def})$
also have $\dots = M$
by $(\text{intro distr-PiM-component prob-space-axioms}) \text{ auto}$
finally show $?thesis .$
qed

lemma $(\text{in prob-space}) \text{distr-stream-space-shd} [\text{simp}]$:
assumes $\text{sets } M = \text{sets } N$
shows $\text{distr } (\text{stream-space } M) N \text{shd} = M$
using $\text{distr-stream-space-snth}[OF \text{assms, of } 0]$ **by** $(\text{simp del: distr-stream-space-snth})$

lemma shift-measurable :
assumes $\text{set } x \subseteq \text{space } M$
shows $(\lambda bs. x @ - bs) \in \text{stream-space } M \rightarrow_M \text{stream-space } M$
proof –
have $(\lambda bs. (x @ - bs) !! n) \in (\text{stream-space } M) \rightarrow_M M$ **for** n
proof $(\text{cases } n < \text{length } x)$
case $True$
have $(\lambda bs. (x @ - bs) !! n) = (\lambda bs. x ! n)$

```

    using True by simp
  also have ... ∈ stream-space M →M M
    using assms True by (intro measurable-const) auto
  finally show ?thesis by simp
next
case False
have (λbs. (x @- bs) !! n) = (λbs. bs !! (n - length x))
  using False by simp
also have ... ∈ (stream-space M) →M M
  by (intro measurable-snth)
finally show ?thesis by simp
qed
thus ?thesis
  by (intro measurable-stream-space2) auto
qed

lemma (in sigma-finite-measure) restrict-space-pair-lift:
  assumes A' ∈ sets A
  shows restrict-space A A' ⊗M M = restrict-space (A ⊗M M) (A' × space M) (is ?L = ?R)
proof -
  let ?X = ((∩) (A' × space M) ' {a × b | a b. a ∈ sets A ∧ b ∈ sets M})
  have 0: A' ⊆ space A
    using assms sets.sets-into-space by blast

  have ?X ⊆ {a × b | a b. a ∈ sets (restrict-space A A') ∧ b ∈ sets M}
  proof (rule image-subsetI)
    fix x assume x ∈ {a × b | a b. a ∈ sets A ∧ b ∈ sets M}
    then obtain u v where uv-def: x = u × v u ∈ sets A v ∈ sets M
      by auto
    have 1: u ∩ A' ∈ sets (restrict-space A A')
      using uv-def(2) unfolding sets-restrict-space by auto
    have v ⊆ space M
      using uv-def(3) sets.sets-into-space by auto
    hence A' × space M ∩ x = (u ∩ A') × v
      unfolding uv-def(1) by auto
    also have ... ∈ {a × b | a b. a ∈ sets (restrict-space A A') ∧ b ∈ sets M}
      using 1 uv-def(3) by auto

    finally show A' × space M ∩ x ∈ {a × b | a b. a ∈ sets (restrict-space A A') ∧ b ∈ sets M}
      by simp
  qed
  moreover have {a × b | a b. a ∈ sets (restrict-space A A') ∧ b ∈ sets M} ⊆ ?X
  proof (rule subsetI)
    fix x assume x ∈ {a × b | a b. a ∈ sets (restrict-space A A') ∧ b ∈ sets M}
    then obtain u v where uv-def: x = u × v u ∈ sets (restrict-space A A') v ∈ sets M
      by auto

    have x = (A' × space M) ∩ x
      unfolding uv-def(1) using uv-def(2,3) sets.sets-into-space
      by (intro Int-absorb1[symmetric]) (auto simp add:sets-restrict-space)
    moreover have u ∈ sets A using uv-def(2) assms unfolding sets-restrict-space by blast
    hence x ∈ {a × b | a b. a ∈ sets A ∧ b ∈ sets M}
      unfolding uv-def(1) using uv-def(3) by auto
    ultimately show x ∈ ?X
      by simp
  qed
  ultimately have 2: ?X = {a × b | a b. a ∈ sets (restrict-space A A') ∧ b ∈ sets M} by simp

```

have sets ?R = sigma-sets (A' × space M) ((∩) (A' × space M) ' {a × b | a b. a ∈ sets A ∧ b ∈ sets M})
unfolding sets-restrict-space sets-pair-measure **using** assms sets.sets-into-space
by (intro sigma-sets-Int sigma-sets.Basic) auto
also have ... = sets (restrict-space A A' ⊗_M M)
unfolding sets-pair-measure space-restrict-space Int-absorb2[OF 0] sets-restrict-space 2
by auto
finally have 3:sets (restrict-space (A ⊗_M M) (A' × space M)) = sets (restrict-space A A' ⊗_M M)
by simp

have 4: emeasure (restrict-space A A' ⊗_M M) S = emeasure (restrict-space (A ⊗_M M) (A' × space M)) S
(is ?L1 = ?R1) if 5:S ∈ sets (restrict-space A A' ⊗_M M) **for** S
proof –
have Pair x - ' S = {} **if** x ∉ A' x ∈ space A **for** x
using that 5 **by** (auto simp add:3[symmetric] sets-restrict-space)
hence 5: emeasure M (Pair x - ' S) = 0 **if** x ∉ A' x ∈ space A **for** x
using that **by** auto
have ?L1 = (∫⁺ x. emeasure M (Pair x - ' S) ∂restrict-space A A')
by (intro emeasure-pair-measure-alt[OF that])
also have ... = (∫⁺ x ∈ A'. emeasure M (Pair x - ' S) ∂A)
using assms **by** (intro nn-integral-restrict-space) auto
also have ... = (∫⁺ x. emeasure M (Pair x - ' S) ∂A)
using 5 **by** (intro nn-integral-cong) force
also have ... = emeasure (A ⊗_M M) S
using that assms **by** (intro emeasure-pair-measure-alt[symmetric])
(auto simp add:3[symmetric] sets-restrict-space)
also have ... = ?R1
using assms that **by** (intro emeasure-restrict-space[symmetric])
(auto simp add:3[symmetric] sets-restrict-space)
finally show ?thesis **by** simp
qed

show ?thesis **using** 3 4
by (intro measure-eqI) auto
qed

lemma to-stream-comb-seq-eq:
to-stream (comb-seq n x y) = stake n (to-stream x) @- to-stream y
unfolding comb-seq-def to-stream-def
by (intro stream-eq-iff) simp

lemma to-stream-snth: to-stream (!! x) = x
by (intro ext stream-eq-iff) (simp add:to-stream-def)

lemma snth-to-stream: snth (to-stream x) = x
by (intro ext) (simp add:to-stream-def)

lemma (in prob-space) branch-stream-space:
(λ(x, y). stake n x @- y) ∈ stream-space M ⊗_M stream-space M →_M stream-space M
distr (stream-space M ⊗_M stream-space M) (stream-space M) (λ(x, y). stake n x @- y)
= stream-space M **(is ?L = ?R)**

proof –
let ?T = stream-space M
let ?S = PiM UNIV (λ-. M)

interpret S: sequence-space M

by *standard*

show $0:(\lambda(x, y). \text{stake } n \ x \ @- \ y) \in ?T \otimes_M ?T \rightarrow_M ?T$
 by *simp*

have $?L = \text{distr } (\text{distr } ?S \ ?T \ \text{to-stream} \otimes_M \text{distr } ?S \ ?T \ \text{to-stream}) \ ?T \ (\lambda(x, y). \text{stake } n \ x @- y)$
 by *(subst (1 2) stream-space-eq-distr) simp*

also have $\dots = \text{distr } (\text{distr } (?S \otimes_M ?S) \ (?T \otimes_M ?T) \ (\lambda(x, y). (\text{to-stream } x, \text{to-stream } y)))$
 $\ ?T \ (\lambda(x, y). \text{stake } n \ x \ @- \ y)$
 using *prob-space-imp-sigma-finite[OF prob-space-stream-space]*
 by *(intro arg-cong2[where f=($\lambda x y. \text{distr } x \ ?T \ y$)] pair-measure-distr)*
(simp-all flip:stream-space-eq-distr)

also have $\dots = \text{distr } (?S \otimes_M ?S) \ ?T \ ((\lambda(x, y). \text{stake } n \ x @- y) \circ (\lambda(x, y). (\text{to-stream } x, \text{to-stream } y)))$
 by *(intro distr-distr 0) (simp add: measurable-pair-iff)*

also have $\dots = \text{distr } (?S \otimes_M ?S) \ ?T \ ((\lambda(x, y). \text{stake } n \ (\text{to-stream } x) \ @- \ \text{to-stream } y))$
 by *(simp add: comp-def case-prod-beta')*

also have $\dots = \text{distr } (?S \otimes_M ?S) \ ?T \ (\text{to-stream} \circ (\lambda(x, y). \text{comb-seq } n \ x \ y))$
 using *to-stream-comb-seq-eq[symmetric]*
 by *(intro arg-cong2[where f=($\lambda x y. \text{distr } x \ ?T \ y$)] ext) auto*

also have $\dots = \text{distr } (\text{distr } (?S \otimes_M ?S) \ ?S \ (\lambda(x, y). \text{comb-seq } n \ x \ y)) \ ?T \ \text{to-stream}$
 by *(intro distr-distr[symmetric] measurable-comb-seq) simp*

also have $\dots = \text{distr } ?S \ ?T \ \text{to-stream}$
 by *(subst S.PiM-comb-seq) simp*

also have $\dots = ?R$
 unfolding *stream-space-eq-distr[symmetric]* by *simp*

finally show $?L = ?R$
 by *simp*

qed

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* \Rightarrow *bool stream* **where**
from-coins coins = *chd coins ## (from-coins (ctl coins))*

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

lemma *to-from-coins*: *to-coins (from-coins x)* = *x*
 by *(rule coin-stream.coinduct[where R=($\lambda x y. x = \text{to-coins } (\text{from-coins } y)$)] simp-all)*

lemma *from-to-coins*: *from-coins (to-coins x)* = *x*
 by *(rule stream.coinduct[where R=($\lambda x y. x = \text{from-coins } (\text{to-coins } y)$)] simp-all)*

lemma *bij-to-coins*: *bij to-coins*
 by *(intro bij-betwI[where g=from-coins] to-from-coins from-to-coins) auto*

lemma *bij-from-coins*: *bij from-coins*
 by *(intro bij-betwI[where g=to-coins] to-from-coins from-to-coins) auto*

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* $n\ x = \text{to-coins } (\text{sdrop } n\ (\text{from-coins } x))$

definition *rel-coins* **where** *rel-coins* $x\ y = (\text{to-coins } x = y)$

definition *cprefix* **where** *cprefix* $x\ y \longleftrightarrow \text{ctake } (\text{length } x)\ y = x$

definition *cconst* **where** *cconst* $x = \text{to-coins } (\text{sconst } x)$

context

includes *lifting-syntax*

begin

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

unfolding *rel-coins-def* **using** *inj-onD*[*OF* *bij-is-inj*[*OF* *bij-to-coins*]]

by (*intro* *bi-uniqueI* *left-uniqueI* *right-uniqueI*) *auto*

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

unfolding *rel-coins-def* **using** *from-to-coins* *to-from-coins*

by (*intro* *bi-totalI* *left-totalI* *right-totalI*) *auto*

lemma *cnth-transfer* [*transfer-rule*]: (*rel-coins* $\implies (=) \implies (=)$) *snth* *cnth*

unfolding *rel-coins-def* *cnth-def* *rel-fun-def* **by** (*auto* *simp:from-to-coins*)

lemma *cshift-transfer* [*transfer-rule*]: ($(=) \implies \text{rel-coins} \implies \text{rel-coins}$) *shift* *cshift*

unfolding *rel-coins-def* *cshift-def* *rel-fun-def* **by** (*auto* *simp:from-to-coins*)

lemma *ctake-transfer* [*transfer-rule*]: ($(=) \implies \text{rel-coins} \implies (=)$) *stake* *ctake*

unfolding *rel-coins-def* *ctake-def* *rel-fun-def* **by** (*auto* *simp:from-to-coins*)

lemma *cdrop-transfer* [*transfer-rule*]: ($(=) \implies \text{rel-coins} \implies \text{rel-coins}$) *sdrop* *cdrop*

unfolding *rel-coins-def* *cdrop-def* *rel-fun-def* **by** (*auto* *simp:from-to-coins*)

lemma *chd-transfer* [*transfer-rule*]: (*rel-coins* $\implies (=)$) *shd* *chd*

unfolding *rel-coins-def* *rel-fun-def* **by** (*auto* *simp:from-to-coins*)

lemma *ctl-transfer* [*transfer-rule*]: (*rel-coins* $\implies \text{rel-coins}$) *stl* *ctl*

unfolding *rel-coins-def* *rel-fun-def* **by** (*auto* *simp:from-to-coins*)

lemma *cconst-transfer* [*transfer-rule*]: ($(=) \implies \text{rel-coins}$) *sconst* *cconst*

unfolding *rel-coins-def* *cconst-def* *rel-fun-def* **by** (*auto* *simp:from-to-coins*)

end

lemma *coins-eq-iff*:

assumes $\bigwedge i. \text{cnth } x\ i = \text{cnth } y\ i$

shows $x = y$

proof –

have $(\forall i. \text{cnth } x\ i = \text{cnth } y\ i) \longrightarrow x = y$

by *transfer* (*use* *stream-eq-iff* **in** *auto*)

thus *?thesis* **using** *assms* **by** *simp*

qed

lemma *length-ctake* [*simp*]: *length* (*ctake* $n\ x$) = n

by *transfer* (*rule* *length-stake*)

lemma *ctake-nth*[*simp*]: $m < n \implies \text{ctake } n\ s\ !\ m = \text{cnth } s\ m$

by *transfer* (*rule* *stake-nth*)

lemma *ctake-cdrop*: *cshift* (*ctake* $n\ s$) (*cdrop* $n\ s$) = s

by *transfer* (*rule* *stake-sdrop*)

lemma *cshift-append[simp]*: $cshift (p@q) s = cshift p (cshift q s)$
by transfer (rule *shift-append*)

lemma *cshift-empty[simp]*: $cshift [] xs = xs$
by transfer *simp*

lemma *ctake-null[simp]*: $ctake 0 xs = []$
by transfer *simp*

lemma *ctake-Suc[simp]*: $ctake (Suc n) s = chd s \# ctake n (ctl s)$
by transfer *simp*

lemma *cdrop-null[simp]*: $cdrop 0 s = s$
by transfer *simp*

lemma *cdrop-Suc[simp]*: $cdrop (Suc n) s = cdrop n (ctl s)$
by transfer *simp*

lemma *chd-shift[simp]*: $chd (cshift xs s) = (if xs = [] then chd s else hd xs)$
by transfer *simp*

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

lemma *shd-sconst[simp]*: $chd (cconst x) = x$
by transfer *simp*

lemma *take-ctake*: $take n (ctake m s) = ctake (min n m) s$
by transfer (rule *take-stake*)

lemma *ctake-add[simp]*: $ctake m s @ ctake n (cdrop m s) = ctake (m + n) s$
by transfer (rule *stake-add*)

lemma *cdrop-add[simp]*: $cdrop m (cdrop n s) = cdrop (n + m) s$
by transfer (rule *sdrop-add*)

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

proof –

have $?L \longleftrightarrow ctake (length x) y = x$

unfolding *cprefix-def* **by** *simp*

also have $\dots \longleftrightarrow (\forall i < length x. (ctake (length x) y) ! i = x ! i)$

by (*simp add: list-eq-iff-nth-eq*)

also have $\dots \longleftrightarrow ?R$

by (*intro all-cong*) *simp*

finally show *?thesis* **by** *simp*

qed

A non-empty shift is not idempotent:

lemma *empty-if-shift-idem*:

assumes $\bigwedge cs. cshift h cs = cs$

shows $h = []$

proof (*cases h*)

case *Nil*

then show *?thesis* **by** *simp*

next

case (*Cons hh ht*)

have $[hh] = ctake 1 (cshift (hh\#ht) (cconst (\neg hh)))$

by *simp*

also have ... = *ctake* 1 (*cconst* (\neg *hh*))
using *assms* **unfolding** *Cons* **by** *simp*
also have ... = [\neg *hh*] **by** *simp*
finally show *?thesis* **by** *simp*
qed

Stream version of *prefix-length-prefix*:

lemma *cprefix-length-prefix*:
assumes *length* *x* \leq *length* *y*
assumes *cprefix* *x* *bs* *cprefix* *y* *bs*
shows *prefix* *x* *y*
proof –
have *take* (*length* *x*) *y* = *take* (*length* *x*) (*ctake* (*length* *y*) *bs*)
using *assms*(3) **unfolding** *cprefix-def* **by** *simp*
also have ... = *ctake* (*length* *x*) *bs*
unfolding *take-ctake* **using** *assms* **by** *simp*
also have ... = *x*
using *assms*(2) **unfolding** *cprefix-def* **by** *simp*
finally have *take* (*length* *x*) *y* = *x*
by *simp*
thus *?thesis*
by (*metis take-is-prefix*)
qed

lemma *same-prefix-not-parallel*:
assumes *cprefix* *x* *bs* *cprefix* *y* *bs*
shows \neg (*x* || *y*)
using *assms* *cprefix-length-prefix*
by (*cases length x \leq length y*) *auto*

lemma *ctake-shift*:
ctake *m* (*cshift* *xs* *ys*) = (if *m* \leq *length* *xs* then *take* *m* *xs* else *xs* @ *ctake* (*m* – *length* *xs*) *ys*)
proof (*induction m arbitrary: xs*)
case (*Suc* *m* *xs*)
thus *?case*
by (*cases xs*) *auto*
qed *auto*

lemma *ctake-shift-small* [*simp*]: *m* \leq *length* *xs* \implies *ctake* *m* (*cshift* *xs* *ys*) = *take* *m* *xs*
and *ctake-shift-big* [*simp*]:
m \geq *length* *xs* \implies *ctake* *m* (*cshift* *xs* *ys*) = *xs* @ *ctake* (*m* – *length* *xs*) *ys*
by (*subst ctake-shift; simp*)+

lemma *cdrop-shift*:
cdrop *m* (*cshift* *xs* *ys*) = (if *m* \leq *length* *xs* then *cshift* (*drop* *m* *xs*) *ys* else *cdrop* (*m* – *length* *xs*) *ys*)
proof (*induction m arbitrary: xs*)
case (*Suc* *m* *xs*)
thus *?case*
by (*cases xs*) *auto*
qed *auto*

lemma *cdrop-shift-small* [*simp*]:
m \leq *length* *xs* \implies *cdrop* *m* (*cshift* *xs* *ys*) = *cshift* (*drop* *m* *xs*) *ys*
and *cdrop-shift-big* [*simp*]:
m \geq *length* *xs* \implies *cdrop* *m* (*cshift* *xs* *ys*) = *cdrop* (*m* – *length* *xs*) *ys*
by (*subst cdrop-shift; simp*)+

Infrastructure for building coin streams:

primcorec *cmap-iterate* :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ coin-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 (*rule coin-stream.coinduct*

[**where** $R = (\lambda x s y s. (\exists x. x s = \text{cmap-iterate } m f x \wedge y s = \text{to-coins } (\text{smap } m (\text{siterate } f x))))$])

show $\exists x. \text{cmap-iterate } m f s = \text{cmap-iterate } m f x \wedge$

$\text{to-coins } (\text{smap } m (\text{siterate } f s)) = \text{to-coins } (\text{smap } m (\text{siterate } f x))$

by (*intro exI[where x=s] refl conjI*)

next

fix *xs ys*

assume $\exists x. x s = \text{cmap-iterate } m f x \wedge y s = \text{to-coins } (\text{smap } m (\text{siterate } f x))$

then obtain *x* **where** $0 : x s = \text{cmap-iterate } m f x \wedge y s = \text{to-coins } (\text{smap } m (\text{siterate } f x))$

by *auto*

have *chd xs = chd ys*

unfolding *0* **by** (*subst cmap-iterate.ctr, subst siterate.ctr*) *simp*

moreover have *ctl xs = cmap-iterate m f (f x)*

unfolding *0* **by** (*subst cmap-iterate.ctr*) *simp*

moreover have *ctl ys = to-coins (smap m (siterate f (f x)))*

unfolding *0* **by** (*subst siterate.ctr*) *simp*

ultimately show

$\text{chd } x s = \text{chd } y s \wedge (\exists x. \text{ctl } x s = \text{cmap-iterate } m f x \wedge \text{ctl } y s = \text{to-coins } (\text{smap } m (\text{siterate } f x)))$

by *auto*

qed

definition *build-coin-gen* :: ('a ⇒ bool list) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ coin-stream

where

build-coin-gen m f s = *cmap-iterate (hd ∘ fst)*

$(\lambda (r, s'). (\text{if } \text{tl } r = [] \text{ then } (m s', f s') \text{ else } (\text{tl } r, s')) (m s, f s))$

lemma *build-coin-gen-aux*:

fixes *f* :: 'a ⇒ '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 (*rule stream.coinduct[where R=(λxs ys. (∃x n. xs = n @- (f x) ∧ ys = n @- (g x)))]*)

show $\exists y n. f x = n @- (f y) \wedge g x = n @- (g y)$

by (*intro exI[where x=x] exI[where x=[]] auto*)

next

fix *xs ys* :: 'b stream

assume $\exists x n. x s = n @- (f x) \wedge y s = n @- (g x)$

hence $\exists x n. n \neq [] \wedge x s = n @- (f x) \wedge y s = n @- (g x)$

using *assms* **by** (*metis shift.simps(1)*)

then obtain *x n* **where** $0 : x s = n @- (f x) \wedge y s = n @- (g x) \wedge n \neq []$

by *auto*

have *shd xs = shd ys*

using *0* **by** *simp*

moreover have *stl xs = tl n @- (f x) stl ys = tl n @- (g x)*

using *0* **by** *auto*

ultimately show *shd xs = shd ys* $\wedge (\exists x n. \text{stl } x s = n @- (f x) \wedge \text{stl } y s = n @- (g x))$

by *auto*

qed

lemma *build-coin-gen*:

assumes $\bigwedge x. m x \neq []$

shows $build\text{-}coin\text{-}gen\ m\ f\ s = to\text{-}coins\ (flat\ (smap\ m\ (siterate\ f\ s)))$

proof –

let $?g = (\lambda(r, s').\ if\ tl\ r = []\ then\ (m\ s',\ f\ s')\ else\ (tl\ r,\ s'))$

have $liter: smap\ (hd\ \circ\ fst)\ (siterate\ ?g\ (bs,\ x)) =$
 $bs\ @-\ (smap\ (hd\ \circ\ fst)\ (siterate\ ?g\ (m\ x,\ f\ x)))$ **if** $bs \neq []$ **for** $x\ bs$
using *that*

proof (*induction bs rule:list-nonempty-induct*)
case (*single y*)
then show $?case$ **by** (*subst siterate.ctr*) *simp*

next
case (*cons y ys*)
then show $?case$ **by** (*subst siterate.ctr*) (*simp add:comp-def*)

qed

have $smap(hd\ \circ\ fst)\ (siterate\ ?g\ (m\ x,\ f\ x)) = m\ x\ @-\ smap(hd\ \circ\ fst)\ (siterate\ ?g\ (m\ (f\ x),\ f\ (f\ x)))$
for x **by** (*subst liter[OF assms]*) *auto*

moreover have $flat\ (smap\ m\ (siterate\ f\ x)) = m\ x\ @-\ flat\ (smap\ m\ (siterate\ f\ (f\ x)))$ **for** x
by (*subst siterate.ctr*) (*simp add:flat-Stream[OF assms]*)

ultimately have $\exists n\ y.\ n \neq [] \wedge$
 $smap\ (hd\ \circ\ fst)\ (siterate\ ?g\ (m\ x,\ f\ x)) = n\ @-\ smap\ (hd\ \circ\ fst)\ (siterate\ ?g\ (m\ y,\ f\ y)) \wedge$
 $flat\ (smap\ m\ (siterate\ f\ x)) = n\ @-\ flat\ (smap\ m\ (siterate\ f\ y))$ **for** x
by (*intro exI[where x=m x] exI[where x=f x] conjI assms*)

hence $smap\ (hd\ \circ\ fst)\ (siterate\ ?g\ (m\ s',\ f\ s')) = flat\ (smap\ m\ (siterate\ f\ s'))$ **for** s'
by (*rule build-coin-gen-aux[where f=($\lambda x.\ smap\ (hd\ \circ\ fst)\ (siterate\ ?g\ (m\ x,\ f\ x))$)]*)

thus *?thesis*
unfolding *build-coin-gen-def cmap-iterate* **by** *simp*

qed

Measure space for coin streams:

definition *coin-space* :: *coin-stream measure*

where $coin\text{-}space = embed\text{-}measure\ (stream\text{-}space\ (measure\text{-}pmf\ (pmf\text{-}of\text{-}set\ UNIV)))\ to\text{-}coins$

open-bundle *coin-space-syntax*

begin

notation *coin-space* ($\langle \mathcal{B} \rangle$)

end

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

using *bij-is-surj[OF bij-to-coins]*

unfolding *coin-space-def space-embed-measure space-stream-space* **by** *simp*

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

unfolding *coin-space-def* **by** (*intro embed-measure-eq-distr bij-is-inj[OF bij-to-coins]*)

lemma *from-coins-measurable*: $from\text{-}coins \in \mathcal{B} \rightarrow_M (stream\text{-}space\ (pmf\text{-}of\text{-}set\ UNIV))$

unfolding *coin-space-def* **by** (*intro measurable-embed-measure1*) (*simp add:from-to-coins*)

lemma *to-coins-measurable*: $to\text{-}coins \in (stream\text{-}space\ (pmf\text{-}of\text{-}set\ UNIV)) \rightarrow_M \mathcal{B}$

unfolding *coin-space-def*

by (*intro measurable-embed-measure2 bij-is-inj[OF bij-to-coins]*)

lemma *chd-measurable*: $chd \in \mathcal{B} \rightarrow_M \mathcal{D}$

proof –

have $0:chd\ (to\text{-}coins\ x) = shd\ x$ **for** x

using *chd-transfer* **unfolding** *rel-fun-def* **by** *auto*

thus *?thesis*

unfolding *coin-space-def* **by** (*intro measurable-embed-measure1*) *simp*
qed

lemma *cnth-measurable*: $(\lambda xs. \text{cnth } xs \ i) \in \mathcal{B} \rightarrow_M \mathcal{D}$

unfolding *coin-space-def cnth-def* **by** (*intro measurable-embed-measure1*) (*simp add:from-to-coins*)

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 –

let ?S = *stream-space* (*pmf-of-set UNIV*)

have ?R = *distr* (*distr* ?S \mathcal{B} *to-coins*) ?S *from-coins*

using *B-t-eq-distr* **by** *simp*

also have ... = *distr* ?S ?S (*from-coins* \circ *to-coins*)

by (*intro distr-distr to-coins-measurable from-coins-measurable*)

also have ... = *distr* ?S ?S *id*

unfolding *id-def comp-def from-to-coins* **by** *simp*

also have ... = ?L

unfolding *id-def* **by** *simp*

finally show ?thesis

by *simp*

qed

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

proof –

let ?S = *stream-space* (*pmf-of-set* (*UNIV::bool set*))

have 1 = *emeasure* ?S (*space* ?S)

by (*intro prob-space.emeasure-space-1[symmetric] prob-space.prob-space-stream-space prob-space-measure-pmf*)

also have ... = *emeasure* \mathcal{B} (*from-coins* – (*space* (*stream-space* (*pmf-of-set UNIV*))) \cap *space* \mathcal{B})

by (*subst B-eq-distr*) (*intro emeasure-distr from-coins-measurable sets.top*)

also have ... = *emeasure* \mathcal{B} (*space* \mathcal{B})

unfolding *space-coin-space space-stream-space vimage-def* **by** *simp*

finally show ?thesis **by** *simp*

qed

interpretation *coin-space*: *prob-space coin-space*

using *B-t-finite* **by** *standard*

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

proof –

have ?L = *distr* (*stream-space* (*measure-pmf* (*pmf-of-set UNIV*))) \mathcal{D} (*chd* \circ *to-coins*)

by (*subst B-t-eq-distr*) (*intro distr-distr to-coins-measurable chd-measurable*)

also have ... = *distr* (*stream-space* (*measure-pmf* (*pmf-of-set UNIV*))) \mathcal{D} *shd*

using *chd-transfer* **unfolding** *rel-fun-def rel-coins-def* **by** (*simp add:comp-def*)

also have ... = ?R

using *coin-space.distr-stream-space-shd* **by** *auto*

finally show ?thesis **by** *simp*

qed

lemma *cshift-measurable*: *cshift* $x \in \mathcal{B} \rightarrow_M \mathcal{B}$

proof –

have (*to-coins* \circ *shift* x \circ *from-coins*) $\in \mathcal{B} \rightarrow_M \mathcal{B}$

by (*intro measurable-comp[OF from-coins-measurable] measurable-comp[OF to-coins-measurable] shift-measurable*) *auto*

thus ?thesis

unfolding *cshift-def* **by** (*simp add:comp-def*)

qed

lemma *cdrop-measurable*: $cdrop\ x \in \mathcal{B} \rightarrow_M \mathcal{B}$

proof –

have $(to\text{-coins} \circ sdrop\ x \circ from\text{-coins}) \in \mathcal{B} \rightarrow_M \mathcal{B}$

by $(intro\ measurable\text{-comp}[OF\ from\text{-coins}\text{-measurable}]\ measurable\text{-comp}[OF\ \text{-}\ to\text{-coins}\text{-measurable}]\ shift\text{-measurable})\ auto$

thus *?thesis*

unfolding *cdrop-def* by $(simp\ add:\text{comp-def})$

qed

lemma *ctake-measurable*: $ctake\ k \in \mathcal{B} \rightarrow_M \mathcal{D}$

proof –

have $stake\ k \circ from\text{-coins} \in \mathcal{B} \rightarrow_M \mathcal{D}$

by $(intro\ measurable\text{-comp}[OF\ from\text{-coins}\text{-measurable}])\ simp$

thus *?thesis*

unfolding *ctake-def* by $(simp\ add:\text{comp-def})$

qed

lemma *branch-coin-space*:

$(\lambda(x, y).\ cshift\ (ctake\ n\ x)\ y) \in \mathcal{B} \otimes_M \mathcal{B} \rightarrow_M \mathcal{B}$

$distr\ (\mathcal{B} \otimes_M \mathcal{B})\ \mathcal{B}\ (\lambda(x, y).)\ cshift\ (ctake\ n\ x)\ y) = \mathcal{B}\ (is\ ?L = ?R)$

proof –

let *?M* = *stream-space* (*measure-pmf* (*pmf-of-set UNIV*))

let *?f* = $(\lambda(x, y).)\ stake\ n\ x\ @-\ y)$

let *?g* = *map-prod from-coins from-coins*

have $(\lambda(x, y).)\ cshift\ (ctake\ n\ x)\ y) = to\text{-coins} \circ (?f \circ ?g)$

by $(simp\ add:\text{comp-def}\ cshift\text{-def}\ ctake\text{-def}\ case\text{-prod-beta}')$

also have $\dots \in \mathcal{B} \otimes_M \mathcal{B} \rightarrow_M \mathcal{B}$

by $(intro\ measurable\text{-comp}[OF\ \text{-}\ to\text{-coins}\text{-measurable}]\ measurable\text{-comp}[\mathbf{where}\ N=(?M \otimes_M ?M)])$

map-prod-measurable from-coins-measurable prob-space.branch-stream-space(1)
prob-space-measure-pmf)

finally show $(\lambda(x, y).)\ cshift\ (ctake\ n\ x)\ y) \in \mathcal{B} \otimes_M \mathcal{B} \rightarrow_M \mathcal{B}$

by *simp*

have $distr\ (\mathcal{B} \otimes_M \mathcal{B})\ (?M \otimes_M ?M)\ ?g = (distr\ \mathcal{B}\ ?M\ from\text{-coins} \otimes_M distr\ \mathcal{B}\ ?M\ from\text{-coins})$

unfolding *map-prod-def* using *prob-space-measure-pmf*

by $(intro\ pair\text{-measure}\text{-distr}[symmetric]\ from\text{-coins}\text{-measurable})\ (auto\ intro!:$

prob-space-imp-sigma-finite prob-space.prob-space-stream-space simp:B-eg-distr[symmetric])

also have $\dots = ?M \otimes_M ?M$

unfolding *B-eg-distr[symmetric]* by *simp*

finally have $0: distr\ (\mathcal{B} \otimes_M \mathcal{B})\ (?M \otimes_M ?M)\ ?g = (?M \otimes_M ?M)$

by *simp*

have $?L = distr\ (\mathcal{B} \otimes_M \mathcal{B})\ \mathcal{B}\ (to\text{-coins} \circ ?f \circ ?g)$

unfolding *cshift-def ctake-def* by $(simp\ add:\text{comp-def}\ map\text{-prod-def}\ case\text{-prod-beta}')$

also have $\dots = distr\ (distr\ (\mathcal{B} \otimes_M \mathcal{B})\ (?M \otimes_M ?M)\ ?g)\ \mathcal{B}\ (to\text{-coins} \circ ?f)$

by $(intro\ distr\text{-distr}[symmetric]\ map\text{-prod}\text{-measurable}\ from\text{-coins}\text{-measurable}\ measurable\text{-comp}[OF\ \text{-}\ to\text{-coins}\text{-measurable}]\ prob\text{-space}\text{-measure}\text{-pmf})\ simp$

also have $\dots = distr\ (?M \otimes_M ?M)\ \mathcal{B}\ (to\text{-coins} \circ ?f)$

unfolding *0* by *simp*

also have $\dots = distr\ (distr\ (?M \otimes_M ?M)\ ?M\ ?f)\ \mathcal{B}\ to\text{-coins}$

by $(intro\ distr\text{-distr}[symmetric]\ to\text{-coins}\text{-measurable})\ simp$

also have $\dots = distr\ ?M\ \mathcal{B}\ to\text{-coins}$

by $(subst\ prob\text{-space.branch}\text{-stream}\text{-space}(2))\ (auto\ intro:\text{prob}\text{-space}\text{-measure}\text{-pmf})$

also have $\dots = ?R$

using *B-t-eq-distr* **by** *simp*
finally show $?L = ?R$
by *simp*
qed

definition *from-coins-t* :: *coin-stream* \Rightarrow (*nat* \Rightarrow *bool discrete*)
where *from-coins-t* = *snth* \circ *smap discrete* \circ *from-coins*

definition *to-coins-t* :: (*nat* \Rightarrow *bool discrete*) \Rightarrow *coin-stream*
where *to-coins-t* = *to-coins* \circ *smap of-discrete* \circ *to-stream*

lemma *from-to-coins-t*:
from-coins-t (*to-coins-t* *x*) = *x*
unfolding *to-coins-t-def from-coins-t-def*
by (*intro ext*) (*simp add:snth-to-stream from-to-coins of-discrete-inverse*)

lemma *to-from-coins-t*:
to-coins-t (*from-coins-t* *x*) = *x*
unfolding *to-coins-t-def from-coins-t-def*
by (*simp add:to-stream-snth to-from-coins comp-def discrete-inverse stream.map-comp stream.map-ident*)

lemma *bij-to-coins-t*: *bij to-coins-t*
by (*intro bij-betwI[where g=from-coins-t] to-from-coins-t from-to-coins-t*) *auto*

lemma *bij-from-coins-t*: *bij from-coins-t*
by (*intro bij-betwI[where g=to-coins-t] to-from-coins-t from-to-coins-t*) *auto*

instantiation *coin-stream* :: *topological-space*
begin

definition *open-coin-stream* :: *coin-stream set* \Rightarrow *bool*
where *open-coin-stream* *U* = *open* (*from-coins-t* ‘ *U*)

instance proof
show *open* (*UNIV* :: *coin-stream set*)
using *bij-is-surj[OF bij-from-coins-t] unfolding open-coin-stream-def* **by** *simp*
show *open* (*S* \cap *T*) **if** *open S open T* **for** *S T* :: *coin-stream set*
using *that unfolding open-coin-stream-def image-Int[OF bij-is-inj[OF bij-from-coins-t]]*
by *auto*
show *open* (\bigcup *K*) **if** $\forall S \in K. \textit{open } S$ **for** *K* :: *coin-stream set set*
using *that unfolding open-coin-stream-def image-Union*
by *auto*
qed
end

definition *coin-stream-basis*
where *coin-stream-basis* = ($\lambda x. \textit{Collect} (\textit{cprefix } x)$) ‘ *UNIV*

lemma *image-collect-eq*: f ‘ $\{x. A (f x)\} = \{x. A x\} \cap \textit{range } f$
by *auto*

lemma *coin-stream-basis*: *topological-basis coin-stream-basis*

proof –
have *bij-betw* ($\lambda x. (!) (\textit{smap discrete } x)$) *UNIV UNIV*
by (*intro bij-betwI[where g=smap of-discrete \circ to-stream]*) (*simp-all add:to-stream-snth snth-to-stream stream.map-comp comp-def of-discrete-inverse discrete-inverse stream.map-ident*)
hence $\exists \textit{range} (\lambda x. (!) (\textit{smap discrete } x)) = \textit{UNIV}$

using *bij-is-surj* by *auto*

obtain $K :: (\text{nat} \Rightarrow \text{bool} \text{ discrete}) \text{ set set}$ **where**

K-countable: *countable K* **and** *K-top-basis*: *topological-basis K* **and**

K-cylinder: $\forall k \in K. \exists X. (k = \text{PiE UNIV } X) \wedge (\forall i. \text{open } (X \ i)) \wedge \text{finite } \{i. X \ i \neq \text{UNIV}\}$

using *product-topology-countable-basis* by *auto*

have *from-coins-cprefix*: $\text{from-coins-t } \{xs. \text{cprefix } p \ xs\} =$

$\text{PiE UNIV } (\lambda i. \text{if } i < \text{length } p \text{ then } \{\text{discrete } (p \ ! \ i)\} \text{ else UNIV})$ (**is** $?L = ?R$) **for** p

proof –

have $2:\text{from-coins } \{xs. \text{cprefix } p \ xs\} = \{f. \forall i < \text{length } p. f \ ! \ i = p \ ! \ i\}$

unfolding *cprefix-iff cnth-def* **using** *bij-is-surj*[*OF* *bij-from-coins*]

by (*subst image-collect-eq*) *auto*

have $\text{from-coins-t } \{xs. \text{cprefix } p \ xs\} = (\text{snth} \circ \text{smap } \text{discrete}) (\text{from-coins } \{xs. \text{cprefix } p \ xs\})$

unfolding *from-coins-t-def image-image* **by** *simp*

also have $\dots = (\text{snth} \circ \text{smap } \text{discrete}) \{f. \forall i < \text{length } p. f \ ! \ i = p \ ! \ i\}$

unfolding 2 **by** *simp*

also have $\dots = (\lambda x. \text{snth } (\text{smap } \text{discrete } x)) \{$

$f. \forall i < \text{length } p. (\text{smap } \text{discrete } f) \ ! \ i = \text{discrete } (p \ ! \ i)\}$

by (*simp add:discrete-inject*)

also have $\dots = \{x. \forall i < \text{length } p. x \ i = \text{discrete } (p \ ! \ i)\} \cap \text{range } (\lambda x. (!) (\text{smap } \text{discrete } x))$

by (*intro image-collect-eq*)

also have $\dots = \{x. \forall i < \text{length } p. x \ i = \text{discrete } (p \ ! \ i)\}$

unfolding 3 **by** *simp*

also have $\dots = \text{PiE UNIV } (\lambda i. \text{if } i < \text{length } p \text{ then } \{\text{discrete } (p \ ! \ i)\} \text{ else UNIV})$

unfolding *PiE-def Pi-def* **by** *auto*

finally show $?thesis$

by *simp*

qed

have *open U* **if** $0:U \in \text{coin-stream-basis}$ **for** U

proof –

obtain p **where** $U\text{-eq}:U = \{xs. \text{cprefix } p \ xs\}$ **using** 0 **unfolding** *coin-stream-basis-def* **by**

auto

show $?thesis$

unfolding *open-coin-stream-def U-eq from-coins-cprefix*

by (*intro open-PiE*) (*auto intro:open-discrete*)

qed

moreover have $\exists B \in \text{coin-stream-basis}. x \in B \wedge B \subseteq U$ **if** *open U* $x \in U$ **for** $U \ x$

proof –

have *open* ($\text{from-coins-t } U$) $\text{from-coins-t } x \in \text{from-coins-t } U$

using that **unfolding** *open-coin-stream-def* **by** *auto*

then obtain B **where** $B: B \in K \text{ from-coins-t } x \in B \ B \subseteq \text{from-coins-t } U$

using *topological-basisE*[*OF* *K-top-basis*] **by** *blast*

obtain X **where** $X: B = \text{PiE UNIV } X$ **and** *fin-X*: $\text{finite } \{i. X \ i \neq \text{UNIV}\}$

using *K-cylinder B(1)* **by** *auto*

define Z **where** $Z \ i = (X \ i \neq \text{UNIV})$ **for** i

define n **where** $n = (\text{if } \{i. X \ i \neq \text{UNIV}\} \neq \{\} \text{ then } \text{Suc } (\text{Max } \{i. X \ i \neq \text{UNIV}\}) \text{ else } 0)$

have $i < n$ **if** $Z \ i$ **for** i

using *fin-X that less-Suc-eq-le* **unfolding** *n-def Z-def*[*symmetric*] **by** (*auto split:if-split-asm*)

hence *X-univ*: $X \ i = \text{UNIV}$ **if** $i \geq n$ **for** i

using that *leD* **unfolding** *Z-def* **by** *auto*

define R **where** $R = \{xs. \text{cprefix } (\text{ctake } n \ x) \ xs\}$

have $\{\text{discrete } (\text{ctake } n \ x \ ! \ i)\} \subseteq X \ i$ **if** $i < n$ **for** i

proof –

have $\{\text{discrete } (\text{ctake } n \ x \ ! \ i)\} = \{\text{discrete } (\text{cnth } x \ i)\}$ **using that**

```

    by simp
  also have ... = {from-coins-t x i}
    unfolding from-coins-t-def cnth-def by simp
  also have ...  $\subseteq$  X i
    using B(2) unfolding X PiE-def Pi-def by auto
  finally show ?thesis
    by simp
qed
hence from-coins-t ' R  $\subseteq$  PiE UNIV X
  using X-univ unfolding R-def from-coins-cprefix
  by (intro PiE-mono) auto
moreover have ...  $\subseteq$  from-coins-t ' U
  using B(3) X by simp
ultimately have from-coins-t ' R  $\subseteq$  from-coins-t ' U
  by simp
hence R  $\subseteq$  U
  using bij-is-inj[OF bij-from-coins-t]
  by (simp add: inj-image-eq-iff subset-image-iff)
moreover have R  $\in$  coin-stream-basis x  $\in$  R
  unfolding R-def coin-stream-basis-def by (auto simp:cprefix-def)
ultimately show ?thesis
  by auto
qed
ultimately show ?thesis
  by (intro topological-basisI) auto
qed

lemma coin-stream-open: open {xs. cprefix x xs}
  by (intro topological-basis-open[OF coin-stream-basis]) (simp add:coin-stream-basis-def)

instance coin-stream :: second-countable-topology
proof
  show  $\exists$  (B :: coin-stream set set). countable B  $\wedge$  open = generate-topology B
    by (intro exI[where x=coin-stream-basis] topological-basis-imp-subbasis conjI
        coin-stream-basis) (simp add:coin-stream-basis-def)
qed

instantiation coin-stream :: uniformity-dist
begin
definition dist-coin-stream :: coin-stream  $\Rightarrow$  coin-stream  $\Rightarrow$  real
  where dist-coin-stream x y = dist (from-coins-t x) (from-coins-t y)

definition uniformity-coin-stream :: (coin-stream  $\times$  coin-stream) filter
  where uniformity-coin-stream = (INF e $\in$ {0 <..}. principal {(x, y). dist x y < e})

instance proof
  show uniformity = (INF e $\in$ {0 <..}. principal {(x, y). dist (x::coin-stream) y < e})
    unfolding uniformity-coin-stream-def by simp
qed
end

lemma in-from-coins-iff: x  $\in$  from-coins-t ' U  $\iff$  (to-coins-t x  $\in$  U)
  using to-from-coins-t from-to-coins-t by (simp add:image-iff) metis

instantiation coin-stream :: metric-space
begin
instance proof
  show open U = ( $\forall$  x $\in$ U.  $\forall$  F (x', y) in uniformity. x' = x  $\implies$  y  $\in$  U) for U :: coin-stream set

```

proof –
have $open\ U \longleftrightarrow open\ (from\ coins\ t\ \text{' } U)$
unfolding $open\ coin\ stream\ def$ **by** $simp$
also have $\dots \longleftrightarrow (\forall x \in U. \exists e > 0. \forall y. dist\ (from\ coins\ t\ x)\ y < e \longrightarrow y \in from\ coins\ t\ \text{' } U)$
unfolding $fun\ open\ ball\ aux$ **by** $auto$
also have $\dots \longleftrightarrow (\forall x \in U. \exists e > 0. \forall y \in to\ coins\ t\ \text{' } UNIV. dist\ x\ y < e \longrightarrow y \in U)$
unfolding $dist\ coin\ stream\ def$ **by** $(intro\ ball\ cong\ refl\ ex\ cong)$
(simp add: from-to-coins-t in-from-coins-iff)
also have $\dots \longleftrightarrow (\forall x \in U. \exists e > 0. \forall y. dist\ x\ y < e \longrightarrow y \in U)$
using $bij\ is\ surj[OF\ bij\ to\ coins\ t]$ **by** $simp$
finally have $open\ U = (\forall x \in U. \exists e > 0. \forall y. dist\ x\ y < e \longrightarrow y \in U)$
by $simp$
thus *?thesis*
unfolding $eventually\ uniformity\ metric$ **by** $simp$
qed
show $(dist\ x\ y = 0) = (x = y)$ **for** $x\ y :: coin\ stream$
unfolding $dist\ coin\ stream\ def$ **by** $(metis\ dist\ eq\ 0\ iff\ to\ from\ coins\ t)$
show $dist\ x\ y \leq dist\ x\ z + dist\ y\ z$ **for** $x\ y\ z :: coin\ stream$
unfolding $dist\ coin\ stream\ def$ **by** $(intro\ dist\ triangle2)$
qed
end

lemma $from\ coins\ t\ u\ continuous: uniformly\ continuous\ on\ UNIV\ from\ coins\ t$
unfolding $uniformly\ continuous\ on\ def\ dist\ coin\ stream\ def$ **by** $auto$

lemma $to\ coins\ t\ u\ continuous: uniformly\ continuous\ on\ UNIV\ to\ coins\ t$
unfolding $uniformly\ continuous\ on\ def\ dist\ coin\ stream\ def\ from\ to\ coins\ t$ **by** $auto$

lemma $to\ coins\ t\ continuous: continuous\ on\ UNIV\ to\ coins\ t$
using $to\ coins\ t\ u\ continuous\ uniformly\ continuous\ imp\ continuous$ **by** $auto$

instance $coin\ stream :: complete\ space$

proof

show $convergent\ X$ **if** $Cauchy\ X$ **for** $X :: nat \Rightarrow coin\ stream$

proof –

have $Cauchy\ (from\ coins\ t \circ X)$

using $uniformly\ continuous\ imp\ Cauchy\ continuous[unfolded\ Cauchy\ continuous\ on\ def]$
 $from\ coins\ t\ u\ continuous\ that$ **by** $auto$

hence $convergent\ (from\ coins\ t \circ X)$

by $(rule\ Cauchy\ convergent)$

then obtain x **where** $(from\ coins\ t \circ X) \longrightarrow x$

unfolding $convergent\ def$ **by** $auto$

moreover have $isCont\ to\ coins\ t\ x$

using $to\ coins\ t\ continuous\ continuous\ on\ eq\ continuous\ within$ **by** $blast$

ultimately have $(to\ coins\ t \circ from\ coins\ t \circ X) \longrightarrow to\ coins\ t\ x$

using $isCont\ tendsto\ compose$ **by** $(auto\ simp\ add: comp\ def)$

thus $convergent\ X$

unfolding $convergent\ def\ comp\ def\ to\ from\ coins\ t$ **by** $auto$

qed

qed

lemma $at\ least\ borell:$

assumes $topological\ basis\ K$

assumes $countable\ K$

assumes $K \subseteq sets\ M$

assumes $open\ U$

shows $U \in sets\ M$

proof –

obtain K' **where** K' -range: $K' \subseteq K$ **and** $\bigcup K' = U$
using $assms(1,4)$ **unfolding** *topological-basis-def* **by** *blast*
hence $U = \bigcup K'$ **by** *simp*
also have $\dots \in sets\ M$
using K' -range $assms(2,3)$ *countable-subset*
by (*intro sets.countable-Union*) *auto*
finally show *?thesis* **by** *simp*
qed

lemma *measurable-sets-coin-space*:
assumes $f \in measurable\ \mathcal{B}\ A$
assumes $Collect\ P \in sets\ A$
shows $\{xs.\ P\ (f\ xs)\} \in sets\ \mathcal{B}$
proof –
have $\{xs.\ P\ (f\ xs)\} = f\ -' \ Collect\ P \cap\ space\ \mathcal{B}$
unfolding *vimage-def space-coin-space* **by** *simp*
also have $\dots \in sets\ \mathcal{B}$
by (*intro measurable-sets[OF assms(1,2)]*)
finally show *?thesis* **by** *simp*
qed

lemma *coin-space-is-borel-measure*:
assumes *open U*
shows $U \in sets\ \mathcal{B}$
proof –
have $0:countable\ coin-stream-basis$
unfolding *coin-stream-basis-def* **by** *simp*

have $cnth\ sets: \{xs.\ cnth\ xs\ i = v\} \in sets\ \mathcal{B}$ **for** $i\ v$
by (*intro measurable-sets-coin-space[OF cnth-measurable]*) *auto*

have $\{xs.\ cprefix\ x\ xs\} \in sets\ \mathcal{B}$ **for** x
proof (*cases x ≠ []*)
case *True*
have $\{xs.\ cprefix\ x\ xs\} = (\bigcap i < length\ x.\ \{xs.\ cnth\ xs\ i = x\ !\ i\})$
unfolding *cprefix-iff* **by** *auto*
also have $\dots \in sets\ \mathcal{B}$
using *cnth-sets True*
by (*intro sets.countable-INT image-subsetI*) *auto*
finally show *?thesis* **by** *simp*
next
case *False*
hence $\{xs.\ cprefix\ x\ xs\} = space\ \mathcal{B}$
unfolding *cprefix-iff space-coin-space* **by** *simp*
also have $\dots \in sets\ \mathcal{B}$
by *simp*
finally show *?thesis* **by** *simp*
qed
hence $1:coin-stream-basis \subseteq sets\ \mathcal{B}$
unfolding *coin-stream-basis-def* **by** *auto*
show *?thesis*
using *at-least-borelI[OF coin-stream-basis 0 1 assms]* **by** *simp*
qed

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 -
  have ?T ( $U \cap V$ ) if ?T U ?T V for U V using that by auto
  moreover have ?T ( $\bigcup K$ ) if  $\bigwedge U. U \in K \implies ?T U$  for K using that by auto
  ultimately show ?thesis unfolding istopology-def by auto
qed

lemma openin-option-ud: openin option-ud S  $\longleftrightarrow$  ( $S = UNIV \vee None \notin S$ )
  unfolding option-ud-def by (subst topology-inverse'[OF option-ud-topology]) auto

lemma topspace-option-ud: topspace option-ud = UNIV
proof -
  have  $UNIV \subseteq$  topspace option-ud by (intro openin-subset) (simp add:openin-option-ud)
  thus ?thesis by auto
qed

lemma contionuos-into-option-udI:
  assumes  $\bigwedge x. openin X (f -' \{Some x\} \cap topspace X)$ 
  shows continuous-map X option-ud f
proof -
  have openin X  $\{x \in topspace X. f x \in U\}$  if openin option-ud U for U
  proof (cases  $U = UNIV$ )
    case True
    then show ?thesis by simp
  next
    case False
    define V where  $V = the ' U$ 
    have  $None \notin U$ 
      using that False unfolding openin-option-ud by simp
    hence  $Some ' V = id ' U$ 
      unfolding V-def image-image id-def
      by (intro image-cong refl) (metis option.exhaust-sel)
    hence  $U = Some ' V$  by simp
    hence  $\{x \in topspace X. f x \in U\} = (\bigcup v \in V. f -' \{Some v\} \cap topspace X)$  by auto
    moreover have openin X  $(\bigcup v \in V. f -' \{Some v\} \cap topspace X)$ 
      using assms by (intro openin-Union) auto
    ultimately show ?thesis by auto
  qed
  thus ?thesis
    unfolding continuous-map topspace-option-ud by auto
qed

lemma map-option-continuous:
  continuous-map option-ud option-ud (map-option f)
  by (intro contionuos-into-option-udI) (simp add:topspace-option-ud vimage-def openin-option-ud)

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 \Rightarrow 'a option

where *the-elem-opt* $S = (\text{if } \text{Set.is-singleton } S \text{ then } \text{Some } (\text{the-elem } S) \text{ else } \text{None})$

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

unfolding *the-elem-opt-def is-singleton-def* **by** (*simp split:if-split-asm*)

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

unfolding *the-elem-opt-def* **by** *simp*

definition *at-most-one* :: 'a set \Rightarrow bool

where *at-most-one* $S \iff (\forall x y. x \in S \wedge y \in S \longrightarrow x = y)$

lemma *at-most-one-cases[consumes 1]*:

assumes *at-most-one* S

assumes $P \{ \text{the-elem } S \}$

assumes $P \{ \}$

shows $P S$

proof (*cases* $S = \{ \}$)

case *True*

then show *?thesis* **using** *assms* **by** *auto*

next

case *False*

then obtain x **where** $x \in S$ **by** *auto*

hence $S = \{x\}$ **using** *assms(1)* **unfolding** *at-most-one-def* **by** *auto*

thus *?thesis* **using** *assms(2)* **by** *simp*

qed

lemma *the-elem-opt-Some-iff[simp]*: *at-most-one* $S \implies \text{the-elem-opt } S = \text{Some } x \iff S = \{x\}$

by (*induction* S *rule:at-most-one-cases*) *auto*

lemma *the-elem-opt-None-iff[simp]*: *at-most-one* $S \implies \text{the-elem-opt } S = \text{None} \iff S = \{ \}$

by (*induction* S *rule:at-most-one-cases*) *auto*

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 random-*alg-int* = coin-stream \Rightarrow ('a \times coin-stream) option

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

definition *return-rai* :: 'a \Rightarrow 'a random-*alg-int*

where *return-rai* $x \text{ bs} = \text{Some } (x, \text{bs})$

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 *bind-rai* :: 'a random-*alg-int* \Rightarrow ('a \Rightarrow 'b random-*alg-int*) \Rightarrow 'b random-*alg-int*

where *bind-rai* $m f \text{ bs} =$

do {

```

    (r, bs') ← m bs;
  f r bs'
}

```

ad hoc-overloading *Monad-Syntax.bind* \equiv *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 *coin-rai* :: *bool random-alg-int*
where *coin-rai* bs = *Some (chd bs, 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 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*: *bind-rai (return-rai x) g = g x*
unfolding *bind-rai-def return-rai-def* **by** *simp*

lemma *bind-rai-assoc*: *bind-rai (bind-rai f g) h = bind-rai f (λx. bind-rai (g x) h)*
unfolding *bind-rai-def* **by** (*simp add:case-prod-beta'*)

lemma *bind-return-rai*: *bind-rai m return-rai = m*
unfolding *bind-rai-def return-rai-def* **by** *simp*

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

definition *wf-random* :: '*a random-alg-int* \Rightarrow *bool* **where**

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

$wf\text{-random } f \longleftrightarrow (\forall bs.$
case $f\ bs$ *of*
 $None \Rightarrow True \mid$
 $Some (r, bs') \Rightarrow (\exists p. cprefix\ p\ bs \wedge wf\text{-on-prefix } f\ p\ r))$

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

lemma $in\text{-}range\text{-}rmI$:

assumes $r\ bs = Some (y, n)$

shows $y \in range\text{-}rm\ r$

proof –

have $Some (y, n) \in range\ r$

using $assms[symmetric]$ **by** $auto$

thus $?thesis$

unfolding $range\text{-}rm\text{-}def$ **using** $fun.set\text{-}map$ **by** $force$

qed

definition $distr\text{-}rai :: 'a\ random\text{-}alg\text{-}int \Rightarrow 'a\ option\ measure$

where $distr\text{-}rai\ f = distr\ \mathcal{B}\ \mathcal{D} (map\ option\ fst \circ f)$

lemma $wf\text{-}randomI$:

assumes $\bigwedge bs. f\ bs \neq None \implies (\exists p\ r. cprefix\ p\ bs \wedge wf\text{-on-prefix } f\ p\ r)$

shows $wf\text{-}random\ f$

proof –

have $\exists p. cprefix\ p\ bs \wedge wf\text{-on-prefix } f\ p\ r$ **if** $0:f\ bs = Some (r, bs')$ **for** $bs\ r\ bs'$

proof –

obtain $p\ r'$ **where** $1:cprefix\ p\ bs$ **and** $2:wf\text{-on-prefix } f\ p\ r'$

using $assms\ 0$ **by** $force$

have $f\ bs = f (cshift\ p (cdrop (length\ p)\ bs))$

using 1 **unfolding** $cprefix\text{-}def$ **by** $(metis\ ctake\text{-}cdrop)$

also have $\dots = Some (r', cdrop (length\ p)\ bs)$

using 2 **unfolding** $wf\text{-on-prefix}\text{-}def$ **by** $auto$

finally have $f\ bs = Some (r', cdrop (length\ p)\ bs)$

by $simp$

hence $r = r'$ **using** 0 **by** $simp$

thus $?thesis$ **using** $1\ 2$ **by** $auto$

qed

thus $?thesis$

unfolding $wf\text{-}random\text{-}def$ **by** $(auto\ split:option.split)$

qed

lemma $wf\text{-on-prefix}\text{-}bindI$:

assumes $wf\text{-on-prefix } m\ p\ r$

assumes $wf\text{-on-prefix } (f\ r)\ q\ s$

shows $wf\text{-on-prefix } (m \gg= f) (p@q)\ s$

proof –

have $(m \gg= f) (cshift (p@q)\ cs) = Some (s, cs)$ **for** cs

proof –

have $(m \gg= f) (cshift (p@q)\ cs) = (m \gg= f) (cshift\ p (cshift\ q\ cs))$

by $simp$

also have $\dots = (f\ r) (cshift\ q\ cs)$

using $assms$ **unfolding** $wf\text{-on-prefix}\text{-}def\ bind\text{-}rai\text{-}def$ **by** $simp$

also have $\dots = Some (s, cs)$

using $assms$ **unfolding** $wf\text{-on-prefix}\text{-}def$ **by** $simp$

finally show $?thesis$ **by** $simp$

qed

thus $?thesis$

unfolding *wf-on-prefix-def* **by** *simp*
qed

lemma *wf-bind*:

assumes *wf-random* *m*

assumes $\bigwedge x. x \in \text{range-}rm\ m \implies \text{wf-random}\ (f\ x)$

shows *wf-random* ($m \ggg f$)

proof (*rule wf-randomI*)

fix *bs*

assume ($m \ggg f$) *bs* \neq *None*

then obtain *x bs' y bs''* **where** *1*: $m\ bs = \text{Some}\ (x, bs')$ **and** *2*: $f\ x\ bs' = \text{Some}\ (y, bs'')$

unfolding *bind-rai-def* **by** (*cases m bs*) *auto*

hence *wf*: *wf-random* (*f x*)

by (*intro assms(2) in-range-rmI*) *auto*

obtain *p* **where** *5*:*wf-on-prefix* *m p x* **and** *3*:*cprefix* *p bs*

using *assms(1) 1* **unfolding** *wf-random-def* **by** (*auto split:option.split-asm*)

have *4*:*bs* = *cshift* *p* (*cdrop* (*length p*) *bs*)

using *3* **unfolding** *cprefix-def* **by** (*metis ctake-cdrop*)

hence $m\ bs = \text{Some}\ (x, \text{cdrop}\ (\text{length}\ p)\ bs)$

using *5* **unfolding** *wf-on-prefix-def* **by** *metis*

hence $bs' = \text{cdrop}\ (\text{length}\ p)\ bs$

using *1* **by** *auto*

hence *6*:*bs* = *cshift* *p bs'*

using *4* **by** *auto*

obtain *q* **where** *7*:*wf-on-prefix* (*f x*) *q y* **and** *8*:*cprefix* *q bs'*

using *wf 2* **unfolding** *wf-random-def* **by** (*auto split:option.split-asm*)

have *cprefix* (*p@q*) *bs*

unfolding *6* **using** *8* **unfolding** *cprefix-def* **by** *auto*

moreover have *wf-on-prefix* ($m \ggg f$) (*p@q*) *y*

by (*intro wf-on-prefix-bindI[OF 5] 7*)

ultimately show $\exists p\ r. \text{cprefix}\ p\ bs \wedge \text{wf-on-prefix}\ (m \ggg f)\ p\ r$

by *auto*

qed

lemma *wf-return*:

wf-random (*return-rai x*)

proof (*rule wf-randomI*)

fix *bs* **assume** *return-rai x bs* \neq *None*

have *wf-on-prefix* (*return-rai x*) \square *x*

unfolding *wf-on-prefix-def return-rai-def* **by** *auto*

moreover have *cprefix* \square *bs*

unfolding *cprefix-def* **by** *auto*

ultimately show $\exists p\ r. \text{cprefix}\ p\ bs \wedge \text{wf-on-prefix}\ (\text{return-rai}\ x)\ p\ r$

by *auto*

qed

lemma *wf-coin*:

wf-random (*coin-rai*)

proof (*rule wf-randomI*)

fix *bs* **assume** *coin-rai bs* \neq *None*

have *wf-on-prefix* *coin-rai* [*chd bs*] (*chd bs*)

unfolding *wf-on-prefix-def coin-rai-def* **by** *auto*

moreover have *cprefix* [*chd bs*] *bs*

unfolding *cprefix-def* **by** *auto*

ultimately show $\exists p\ r. \text{cprefix}\ p\ bs \wedge \text{wf-on-prefix}\ \text{coin-rai}\ p\ r$

by *auto*
qed

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*
using *assms* **unfolding** *wf-on-prefix-def* *eval-rm-def* **by** *auto*

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 –
have *0:bs* = *cshift* *p* (*cdrop* (*length* *p*) *bs*)
using *assms*(2) **unfolding** *cprefix-def* **by** (*metis* *ctake-cdrop*)
hence *f* *bs* = *Some* (*r*, *cdrop* (*length* *p*) *bs*)
using *assms*(1) *0* **unfolding** *wf-on-prefix-def* **by** *metis*
thus *?thesis*
using *eval-rmD*[*OF* *assms*(1)] **by** *simp*
qed

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

proof –
obtain *h* where *0:q* = *p@h*
using *assms*(3) *prefixE* **that** **by** *auto*
obtain *r1* where *1:wf-on-prefix* *f* *p* *r1*
using *assms*(1) **unfolding** *ptree-rm-def* **by** *auto*
obtain *r2* where *2:wf-on-prefix* *f* *q* *r2*
using *assms*(2) **unfolding** *ptree-rm-def* **by** *auto*
have *x* = *cshift* *h* *x* **for** *x* :: *coin-stream*
proof –
have *Some* (*r2*, *x*) = *f* (*cshift* *q* *x*)
using *2* **unfolding** *wf-on-prefix-def* **by** *auto*
also have ... = *f* (*cshift* *p* (*cshift* *h* *x*))
using *0* **by** *auto*
also have ... = *Some* (*r1*, *cshift* *h* *x*)
using *1* **unfolding** *wf-on-prefix-def* **by** *auto*
finally show *x* = *cshift* *h* *x*
by *simp*
qed
hence *h* = []
using *empty-if-shift-idem* **by** *simp*
thus *?thesis* using *0* **by** *simp*
qed

lemma *prefixes-parallel*:
assumes *p* \in *ptree-rm* *f*
assumes *q* \in *ptree-rm* *f*

shows $p = q \vee p \parallel q$
 using *prefixes-parallel-helper* *assms* **by** *blast*

lemma *prefixes-singleton*:

assumes $p \in \{p. p \in \text{ptree-rm } f \wedge \text{cprefix } p \text{ } bs\}$
 shows $\{p \in \text{ptree-rm } f. \text{cprefix } p \text{ } bs\} = \{p\}$

proof

have $q = p$ **if** $q \in \text{ptree-rm } f \text{ cprefix } q \text{ } bs$ **for** q
 using *same-prefix-not-parallel* *assms* *prefixes-parallel* **that** **by** *blast*
 thus $\{p \in \text{ptree-rm } f. \text{cprefix } p \text{ } bs\} \subseteq \{p\}$
by (*intro subsetI*) *simp*

next

show $\{p\} \subseteq \{p \in \text{ptree-rm } f. \text{cprefix } p \text{ } bs\}$
 using *assms* **by** *auto*

qed

lemma *prefixes-at-most-one*:

at-most-one $\{p \in \text{ptree-rm } f. \text{cprefix } p \text{ } x\}$
 unfolding *at-most-one-def* **using** *same-prefix-not-parallel* *prefixes-parallel* **by** *blast*

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

lemma *wf-random-alt*:

assumes *wf-random* f
 shows $f \text{ } bs = \text{map-option } (\lambda p. (\text{eval-rm } f \text{ } p, \text{cdrop } (\text{length } p) \text{ } bs)) (\text{consumed-prefix } f \text{ } bs)$

proof (*cases* $f \text{ } bs$)

case *None*

have *False* **if** $p\text{-in}$: $p \in \text{ptree-rm } f$ **and** $p\text{-pref}$: $\text{cprefix } p \text{ } bs$ **for** p

proof –

obtain r **where** wf : $wf\text{-on-prefix } f \text{ } p \text{ } r$ **using** *that* $p\text{-in}$ **unfolding** *ptree-rm-def* **by** *auto*

have $bs = \text{cshift } p (\text{cdrop } (\text{length } p) \text{ } bs)$

using $p\text{-pref}$ **unfolding** *cprefix-def* **by** (*metis* *ctake-cdrop*)

hence $f \text{ } bs \neq \text{None}$

using wf **unfolding** *wf-on-prefix-def*

by (*metis* *option.simps(3)*)

thus *False* **using** *None* **by** *simp*

qed

hence $0:\{p \in \text{ptree-rm } f. \text{cprefix } p \text{ } bs\} = \{\}$

by *auto*

show *?thesis* **unfolding** *0 None* *consumed-prefix-def* **by** *simp*

next

case (*Some* a)

moreover obtain $r \text{ } cs$ **where** $a = (r, cs)$ **by** (*cases* a) *auto*

ultimately have $f \text{ } bs = \text{Some } (r, cs)$ **by** *simp*

hence $\exists p. \text{cprefix } p \text{ } bs \wedge wf\text{-on-prefix } f \text{ } p \text{ } r$

using *assms(1)* **unfolding** *wf-random-def* **by** (*auto* *split:option.split-asm*)

then obtain p **where** sp : $\text{cprefix } p \text{ } bs$ **and** wf : $wf\text{-on-prefix } f \text{ } p \text{ } r$

by *auto*

hence $p \in \{p \in \text{ptree-rm } f. \text{cprefix } p \text{ } bs\}$

unfolding *ptree-rm-def* **by** *auto*

hence $0:\{p \in \text{ptree-rm } f. \text{cprefix } p \text{ } bs\} = \{p\}$

using *prefixes-singleton* **by** *auto*

show *?thesis* **unfolding** *0 wf-on-prefixD[OF wf sp]* *consumed-prefix-def* **by** *simp*

qed

lemma *range-rm-alt*:

assumes *wf-random* f

shows $\text{range-rm } f = \text{eval-rm } f \text{ ' } \text{ptree-rm } f \text{ (is ?L = ?R)}$

proof –

have $0 : \text{cprefix } p \text{ (cshift } p \text{ (const False)) for } p$
 unfolding *cprefix-def* **by** *auto*
have $?L = \{x. \exists bs. \text{map-option (eval-rm } f) \text{ (consumed-prefix } f \text{ } bs) = \text{Some } x\}$
 unfolding *range-rm-def comp-def* **by** (*subst wf-random-alt[OF assms]*)
 (*simp add:map-option.compositionality comp-def vimage-def image-iff eq-commute*)
also have $\dots = \{x. \exists p \text{ } bs. x = \text{eval-rm } f \text{ } p \wedge \text{consumed-prefix } f \text{ } bs = \text{Some } p\}$
 unfolding *map-option-eq-Some*
 by (*intro Collect-cong*) *metis*
also have $\dots = \{x. \exists p. p \in \text{ptree-rm } f \wedge x = \text{eval-rm } f \text{ } p\}$
 unfolding *consumed-prefix-def the-elem-opt-Some-iff[OF prefixes-at-most-one]*
 using 0 *prefixes-singleton*
 by (*intro Collect-cong*) *blast*
also have $\dots = ?R$
 by *auto*
finally show *?thesis*
 by *simp*
qed

lemma *consumed-prefix-some-iff*:

$\text{consumed-prefix } f \text{ } bs = \text{Some } p \longleftrightarrow (p \in \text{ptree-rm } f \wedge \text{cprefix } p \text{ } bs)$

proof –

have $p \in \text{ptree-rm } f \implies \text{cprefix } p \text{ } bs \implies x \in \text{ptree-rm } f \implies \text{cprefix } x \text{ } bs \implies x = p$ **for** x
 using *same-prefix-not-parallel prefixes-parallel* **by** *blast*
thus *?thesis*
 unfolding *consumed-prefix-def the-elem-opt-Some-iff[OF prefixes-at-most-one]*
 by *auto*
qed

definition *consumed-bits* **where**

$\text{consumed-bits } f \text{ } bs = \text{map-option length (consumed-prefix } f \text{ } bs)$

definition *used-bits-distr* $:: 'a \text{ random-alg-int} \Rightarrow \text{nat option measure}$

where $\text{used-bits-distr } f = \text{distr } \mathcal{B} \ \mathcal{D} \text{ (consumed-bits } f)$

lemma *wf-random-alt2*:

assumes *wf-random* f

shows $f \text{ } bs = \text{map-option } (\lambda n. (\text{eval-rm } f \text{ (ctake } n \text{ } bs), \text{cdrop } n \text{ } bs)) \text{ (consumed-bits } f \text{ } bs)$
 (*is ?L = ?R*)

proof –

have $0 : \text{cprefix } x \text{ } bs$ **if** $\text{consumed-prefix } f \text{ } bs = \text{Some } x$ **for** x
 using *that the-elem-opt-Some-iff[OF prefixes-at-most-one]* **unfolding** *consumed-prefix-def* **by** *auto*
have $?L = \text{map-option } (\lambda p. (\text{eval-rm } f \text{ } p, \text{cdrop (length } p) \text{ } bs)) \text{ (consumed-prefix } f \text{ } bs)$
 by (*subst wf-random-alt[OF assms]*) *simp*
also have $\dots = ?R$
 using 0 **unfolding** *consumed-bits-def map-option.compositionality comp-def cprefix-def*
 by (*cases consumed-prefix* $f \text{ } bs$) *auto*
finally show *?thesis* **by** *simp*
qed

lemma *consumed-prefix-none-iff*:

assumes *wf-random* f

shows $f \text{ } bs = \text{None} \longleftrightarrow \text{consumed-prefix } f \text{ } bs = \text{None}$
 using *wf-random-alt[OF assms]* **by** (*simp*)

lemma *consumed-bits-inf-iff*:

assumes *wf-random* f

shows $f \text{ bs} = \text{None} \longleftrightarrow \text{consumed-bits } f \text{ bs} = \text{None}$
 using $\text{wf-random-alt2}[OF \text{ assms}]$ by (simp)

lemma *consumed-bits-enat-iff*:

$\text{consumed-bits } f \text{ bs} = \text{Some } n \longleftrightarrow \text{ctake } n \text{ bs} \in \text{ptree-rm } f \text{ (is ?L = ?R)}$

proof

assume $\text{consumed-bits } f \text{ bs} = \text{Some } n$

then obtain p where $\text{the-elem-opt } \{p \in \text{ptree-rm } f. \text{cprefix } p \text{ bs}\} = \text{Some } p$ and $0: \text{length } p = n$

unfolding consumed-bits-def $\text{consumed-prefix-def}$ by $(\text{auto split:option.split-asm})$

hence $p \in \text{ptree-rm } f \text{ cprefix } p \text{ bs}$

unfolding $\text{the-elem-opt-Some-iff}[OF \text{ prefixes-at-most-one}]$ by auto

thus $\text{ctake } n \text{ bs} \in \text{ptree-rm } f$

using 0 unfolding cprefix-def by auto

next

assume $\text{ctake } n \text{ bs} \in \text{ptree-rm } f$

hence $\text{ctake } n \text{ bs} \in \{p \in \text{ptree-rm } f. \text{cprefix } p \text{ bs}\}$

unfolding cprefix-def by auto

hence $\{p \in \text{ptree-rm } f. \text{cprefix } p \text{ bs}\} = \{\text{ctake } n \text{ bs}\}$

using $\text{prefixes-singleton}$ by auto

thus $\text{consumed-bits } f \text{ bs} = \text{Some } n$

unfolding consumed-bits-def $\text{consumed-prefix-def}$ by simp

qed

lemma *consumed-bits-measurable*: $\text{consumed-bits } f \in \mathcal{B} \rightarrow_M \mathcal{D}$

proof –

have $0: \text{consumed-bits } f - \{x\} \cap \text{space } \mathcal{B} \in \text{sets } \mathcal{B} \text{ (is ?L } \in -)$

if $x\text{-ne-inf}: x \neq \text{None}$ for x

proof –

obtain n where $x\text{-def}: x = \text{Some } n$

using $x\text{-ne-inf}$ that by auto

have $?L = \{\text{bs}. \exists z. \text{consumed-prefix } f \text{ bs} = \text{Some } z \wedge \text{length } z = n\}$

unfolding consumed-bits-def vimage-def space-coin-space $x\text{-def}$ by simp

also have $\dots = \{\text{bs}. \exists p. \{p \in \text{ptree-rm } f. \text{cprefix } p \text{ bs}\} = \{p\} \wedge \text{length } p = n\}$

unfolding $\text{consumed-prefix-def}$ $x\text{-def}$ $\text{the-elem-opt-Some-iff}[OF \text{ prefixes-at-most-one}]$ by simp

also have $\dots = \{\text{bs}. \exists p. \text{cprefix } p \text{ bs} \wedge \text{length } p = n \wedge p \in \text{ptree-rm } f\}$

using $\text{prefixes-singleton}$ by $(\text{intro Collect-cong ex-cong1}) \text{ auto}$

also have $\dots = \{\text{bs}. \text{ctake } n \text{ bs} \in \text{ptree-rm } f\}$

unfolding cprefix-def by $(\text{intro Collect-cong}) (\text{metis length-ctake})$

also have $\dots \in \text{sets } \mathcal{B}$

by $(\text{intro measurable-sets-coin-space}[OF \text{ ctake-measurable}]) \text{ simp}$

finally show $?thesis$

by simp

qed

thus $?thesis$

by $(\text{intro measurable-sigma-sets-with-exception}[\text{where } d=\text{None}])$

qed

lemma *R-sets*:

assumes $\text{wf:wf-random } f$

shows $\{\text{bs}. f \text{ bs} = \text{None}\} \in \text{sets } \mathcal{B} \ \{\text{bs}. f \text{ bs} \neq \text{None}\} \in \text{sets } \mathcal{B}$

proof –

show $0: \{\text{bs}. f \text{ bs} = \text{None}\} \in \text{sets } \mathcal{B}$

unfolding $\text{consumed-bits-inf-iff}[OF \text{ wf}]$

by $(\text{intro measurable-sets-coin-space}[OF \text{ consumed-bits-measurable}]) \text{ simp}$

have $\{\text{bs}. f \text{ bs} \neq \text{None}\} = \text{space } \mathcal{B} - \{\text{bs}. f \text{ bs} = \text{None}\}$

unfolding *space-coin-space* **by** (*simp add:set-eq-iff del:not-None-eq*)
also have $\dots \in \text{sets } \mathcal{B}$
by (*intro sets.compl-sets 0*)
finally show $\{bs. f \text{ bs} \neq \text{None}\} \in \text{sets } \mathcal{B}$
by *simp*
qed

lemma *countable-range*:
assumes *wf:wf-random f*
shows *countable (range-rm f)*
proof –
have *countable (eval-rm f ‘ UNIV)*
by (*intro countable-image simp*)
moreover have *range-rm f \subseteq eval-rm f ‘ UNIV*
unfolding *range-rm-alt[OF wf]* **by** *auto*
ultimately show *?thesis* **using** *countable-subset* **by** *blast*
qed

lemma *consumed-prefix-continuous*:
continuous-map euclidean option-ud (consumed-prefix f)
proof (*intro contionuos-into-option-udI*)
fix *x :: bool list*

have *open ((consumed-prefix f) – ‘ {Some x}) (is open ?T)*
proof (*cases x \in ptree-rm f*)
case *True*
hence $0: ?T = \{bs. \text{cprefix } x \text{ bs}\}$
unfolding *vimage-def comp-def* **by** (*simp add:consumed-prefix-some-iff*)
show *?thesis*
unfolding *0* **by** (*intro coin-steam-open*)
next
case *False*
hence $?T = \{\}$
unfolding *vimage-def comp-def* **by** (*simp add:consumed-prefix-some-iff*)
thus *?thesis*
by *simp*
qed
thus *openin euclidean ((consumed-prefix f) – ‘ {Some x} \cap topspace euclidean)*
by *simp*
qed

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 \circ f)*
proof –
have $0: \text{map-option fst} \circ (\lambda bs. f \text{ bs}) =$
 $\text{map-option (eval-rm f)} \circ (\text{consumed-prefix } f)$
by (*subst wf-random-alt[OF wf]*) (*simp add:map-option.compositionality comp-def*)

show *?thesis* **unfolding** *0*
by (*intro continuous-map-compose[OF consumed-prefix-continuous] map-option-continuous*)
qed

lemma *none-measure-subprob-algebra*:
return \mathcal{D} None \in space (subprob-algebra \mathcal{D})
by (*metis measure-subprob return-pmf.rep-eq*)

```

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

lemma the-f-measurable: the ∘ f ∈ R →M  $\mathcal{D}$  ⊗M  $\mathcal{B}$ 
proof -
  define h where h = the ∘ consumed-bits f
  define g where g bs = (ctake (h bs) bs, cdrop (h bs) bs) for bs

  have consumed-bits f bs ≠ None if bs ∈ space R for bs
    using that consumed-bits-inf-iff[OF wf] unfolding R-def space-restrict-space space-coin-space
    by (simp del: not-infinity-eq not-None-eq)

  hence 0: the (f bs) = map-prod (eval-rm f) id (g bs) if bs ∈ space R for bs
    unfolding g-def h-def using that
    by (subst wf-random-alt2[OF wf]) (cases consumed-bits f bs, auto simp del: not-None-eq)

  have 1: h ∈ R →M  $\mathcal{D}$ 
    unfolding R-def h-def
    by (intro measurable-restrict-space1 measurable-comp[OF consumed-bits-measurable]) simp

  have ctake k ∈ R →M  $\mathcal{D}$  for k
    unfolding R-def by (intro measurable-restrict-space1 ctake-measurable)
  moreover have cdrop k ∈ R →M  $\mathcal{B}$  for k
    unfolding R-def by (intro measurable-restrict-space1 cdrop-measurable)
  ultimately have g ∈ R →M  $\mathcal{D}$  ⊗M  $\mathcal{B}$ 
    unfolding g-def
    by (intro measurable-Pair measurable-Pair-compose-split[OF - 1 measurable-id]) simp-all
  hence (map-prod (eval-rm f) id ∘ g) ∈ R →M  $\mathcal{D}$  ⊗M  $\mathcal{B}$ 
    by (intro measurable-comp[where N= $\mathcal{D}$  ⊗M  $\mathcal{B}$ ] map-prod-measurable) auto
  moreover have (the ∘ f) ∈ R →M  $\mathcal{D}$  ⊗M  $\mathcal{B}$  ↔ (map-prod (eval-rm f) id ∘ g) ∈ R →M  $\mathcal{D}$ 
    ⊗M  $\mathcal{B}$ 
    using 0 by (intro measurable-cong) (simp add: comp-def)
  ultimately show ?thesis
    by auto
qed

lemma distr-rai-measurable: map-option fst ∘ f ∈  $\mathcal{B}$  →M  $\mathcal{D}$ 
proof -
  have 0: countable {{bs. f bs ≠ None}, {bs. f bs = None}}
    by simp

  have 1: Ω ∈ sets  $\mathcal{B}$  ∧ map-option fst ∘ f ∈ restrict-space  $\mathcal{B}$  Ω →M  $\mathcal{D}$ 
    if Ω ∈ {{bs. f bs ≠ None}, {bs. f bs = None}} for Ω
  proof (cases Ω = {bs. f bs ≠ None})
    case True
    have Some ∘ fst ∘ (the ∘ f) ∈ R →M  $\mathcal{D}$ 
      by (intro measurable-comp[OF the-f-measurable]) auto
    hence map-option fst ∘ f ∈ R →M  $\mathcal{D}$ 
      unfolding R-def by (subst measurable-cong[where g=Some ∘ fst ∘ (the ∘ f)])
      (auto simp add: space-restrict-space space-coin-space)
    thus Ω ∈ sets  $\mathcal{B}$  ∧ map-option fst ∘ f ∈ restrict-space  $\mathcal{B}$  Ω →M  $\mathcal{D}$ 
      unfolding R-def True using R-sets[OF wf] by auto
  next

```

case *False*
hence $2:\Omega = \{bs. f\ bs = None\}$
using *that by simp*

have *map-option fst* $\circ f \in \text{restrict-space } \mathcal{B} \{bs. f\ bs = None\} \rightarrow_M \mathcal{D}$
by (*subst measurable-cong[where g= λ -. None]*)
(simp-all add:space-restrict-space)

thus $\Omega \in \text{sets } \mathcal{B} \wedge \text{map-option fst} \circ f \in \text{restrict-space } \mathcal{B} \Omega \rightarrow_M \mathcal{D}$
unfolding 2 **using** *R-sets[OF wf]* **by** *auto*

qed

have $3: \text{space } \mathcal{B} \subseteq \bigcup \{\{bs. f\ bs \neq None\}, \{bs. f\ bs = None\}\}$
unfolding *space-coin-space* **by** *auto*

show *?thesis*
by (*rule measurable-piecewise-restrict[OF 0]*) (*use 1 3 space-coin-space in <auto>*)

qed

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

proof –

have *prob-space (distr-rai f)*
unfolding *distr-rai-def* **using** *distr-rai-measurable*
by (*intro coin-space.prob-space-distr*) *auto*

moreover have *sets (distr-rai f) =* \mathcal{D}
unfolding *distr-rai-def* **by** *simp*

ultimately show *?thesis*
unfolding *space-subprob-algebra* **using** *prob-space-imp-subprob-space*
by *auto*

qed

lemma *fst-the-f-measurable:* *fst* \circ *the* \circ *f* $\in R \rightarrow_M \mathcal{D}$

proof –

have *fst* \circ (*the* \circ *f*) $\in R \rightarrow_M \mathcal{D}$
by (*intro measurable-comp[OF the-f-measurable]*) *simp*

thus *?thesis* **by** (*simp add:comp-def*)

qed

lemma *prob-space-distr-rai:*
prob-space (distr-rai f)
unfolding *distr-rai-def* **by** (*intro coin-space.prob-space-distr distr-rai-measurable*)

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 ($\mathcal{D} \otimes_M \mathcal{B}$) (*the* \circ *f*) = *distr R* \mathcal{D} (*fst* \circ *the* \circ *f*) $\otimes_M \mathcal{B}$

proof –

define *C* **where** *C k* = *consumed-bits f* – ‘ $\{Some\ k\}$ ’ **for** *k*

have $2: (\exists k. x \in C\ k) \longleftrightarrow f\ x \neq None$ **for** *x*
using *consumed-bits-inf-iff[OF wf]* **unfolding** *C-def*
by *auto*

hence $5: C\ k \subseteq \text{space } R$ **for** *k*
unfolding *R-def* *space-restrict-space* *space-coin-space*
by *auto*

have 1: $\{bs. f \ bs \neq \text{None}\} \cap \text{space } \mathcal{B} \in \text{sets } \mathcal{B}$
using $R\text{-sets}[OF \ wf]$ **by** simp

have 6: $C \ k \in \text{sets } \mathcal{B}$ **for** k
unfolding $C\text{-def } \text{vimage-def}$
by $(\text{intro measurable-sets-coin-space}[OF \ \text{consumed-bits-measurable}]) \ \text{simp}$

have 8: $x \in C \ k \iff \text{ctake } k \ x \in \text{ptree-rm } f$ **for** $x \ k$
unfolding $C\text{-def}$ **using** $\text{consumed-bits-enat-iff}$ **by** auto

have 7: $\text{the } (f \ (\text{cshift } (\text{ctake } k \ x) \ y)) = (\text{fst } (\text{the } (f \ x)), \ y)$ **if** $x \in C \ k$ **for** $x \ y \ k$
proof –
have $\text{cshift } (\text{ctake } k \ x) \ y \in C \ k$
using $\text{that } 8$ **by** simp
hence $\text{the } (f \ (\text{cshift } (\text{ctake } k \ x) \ y)) = (\text{eval-rm } f \ (\text{ctake } k \ x), \ y)$
using $\text{wf-random-alt2}[OF \ wf]$ **unfolding** $C\text{-def}$ **by** simp
also have $\dots = (\text{fst } (\text{the } (f \ x)), \ y)$
using $\text{that } \text{wf-random-alt2}[OF \ wf]$ **unfolding** $C\text{-def}$ **by** simp
finally show $?thesis$ **by** simp
qed

have $C\text{-disj}$: $\text{disjoint-family } C$
unfolding $\text{disjoint-family-on-def } C\text{-def}$ **by** auto

have 0:
 $\text{emeasure } (\text{distr } R \ (\mathcal{D} \otimes_M \mathcal{B}) \ (\text{the } \circ f)) \ (A \times B) =$
 $\text{emeasure } (\text{distr } R \ \mathcal{D} \ (\text{fst } \circ \text{the } \circ f)) \ A * \text{emeasure } \mathcal{B} \ B$
(is $?L = ?R$) **if** $A \in \text{sets } \mathcal{D} \ B \in \text{sets } \mathcal{B}$ **for** $A \ B$
proof –
have $?3$: $\{bs. \text{fst } (\text{the } (f \ bs)) \in A \wedge bs \in C \ k\} \in \text{sets } \mathcal{B}$ **(is** $?L1 \in -$) **for** k
proof –
have $?L1 = (\text{fst } \circ \text{the } \circ f) \text{-} \text{' } A \cap \text{space } (\text{restrict-space } R \ (C \ k))$
using 5 **unfolding** $\text{vimage-def } \text{space-restrict-space } R\text{-def } \text{space-coin-space}$ **by** auto
also have $\dots \in \text{sets } (\text{restrict-space } R \ (C \ k))$
by $(\text{intro measurable-sets}[OF \ \text{that}(1)] \ \text{measurable-restrict-space1 } \text{fst-the-f-measurable})$
also have $\dots = \text{sets } (\text{restrict-space } \mathcal{B} \ (C \ k))$
using 5 **unfolding** $R\text{-def } \text{sets-restrict-restrict-space } \text{space-restrict-space } \text{space-coin-space}$
by $(\text{intro arg-cong2}[\text{where } f=\text{restrict-space}] \ \text{arg-cong}[\text{where } f=\text{sets}] \ \text{refl}) \ \text{auto}$
finally have $?L1 \in \text{sets } (\text{restrict-space } \mathcal{B} \ (C \ k))$
by simp
thus $?L1 \in \text{sets } \mathcal{B}$
using 6 $\text{space-coin-space } \text{sets-restrict-space-iff}[\text{where } M=\mathcal{B} \ \text{and } \Omega=C \ k]$ **by** auto
qed

have 4: $\{bs. \text{the } (f \ bs) \in A \times B \wedge bs \in C \ k\} \in \text{sets } \mathcal{B}$ **(is** $?L1 \in -$) **for** k
proof –
have $?L1 = (\text{the } \circ f) \text{-} \text{' } (A \times B) \cap \text{space } (\text{restrict-space } R \ (C \ k))$
using 5 **unfolding** $\text{vimage-def } \text{space-restrict-space } R\text{-def } \text{space-coin-space}$ **by** auto
also have $\dots \in \text{sets } (\text{restrict-space } R \ (C \ k))$
using $\text{that } \text{by } (\text{intro measurable-sets}[\text{where } A=\mathcal{D} \otimes_M \mathcal{B}] \ \text{measurable-restrict-space1 } \text{the-f-measurable}) \ \text{auto}$
also have $\dots = \text{sets } (\text{restrict-space } \mathcal{B} \ (C \ k))$
using 5 **unfolding** $R\text{-def } \text{sets-restrict-restrict-space } \text{space-restrict-space } \text{space-coin-space}$
by $(\text{intro arg-cong2}[\text{where } f=\text{restrict-space}] \ \text{arg-cong}[\text{where } f=\text{sets}] \ \text{refl}) \ \text{auto}$
finally have $?L1 \in \text{sets } (\text{restrict-space } \mathcal{B} \ (C \ k))$
by simp
thus $?L1 \in \text{sets } \mathcal{B}$
using 6 $\text{space-coin-space } \text{sets-restrict-space-iff}[\text{where } M=\mathcal{B} \ \text{and } \Omega=C \ k]$ **by** auto

qed

have ?L = *emeasure* R ((*the* ∘ *f*) - ' (A × B) ∩ *space* R)
using *that the-f-measurable* **by** (*intro* *emeasure-distr*) *auto*
also have ... = *emeasure* R {*x*. *the* (*f* *x*) ∈ A × B ∧ *f* *x* ≠ None}
unfolding *vimage-def* *R-def* *Int-def*
by (*simp* *add:space-restrict-space* *space-coin-space*)
also have ... = *emeasure* B {*x*. *the* (*f* *x*) ∈ A × B ∧ (∃ *k*. *x* ∈ C *k*)}
unfolding *R-def* 2 **using** 1 **by** (*intro* *emeasure-restrict-space*) *auto*
also have ... = *emeasure* B (∪ *k*. {*x*. *the* (*f* *x*) ∈ A × B ∧ *x* ∈ C *k*})
by (*intro* *arg-cong2*[**where** *f=emeasure*]) *auto*
also have ... = (∑ *k*. *emeasure* B {*x*. *the* (*f* *x*) ∈ A × B ∧ *x* ∈ C *k*})
using 4 *C-disj*
by (*intro* *suminf-emeasure*[*symmetric*] *subsetI*) (*auto* *simp:disjoint-family-on-def*)
also have ... = (∑ *k*. *emeasure* (*distr* (B ⊗_M B) B (λ(*x*,*y*). (*cshift* (*ctake* *k* *x*) *y*)))
{*x*. *the* (*f* *x*) ∈ A × B ∧ *x* ∈ C *k*})
by (*intro* *suminf-cong* *arg-cong2*[**where** *f=emeasure*] *branch-coin-space*(2)[*symmetric*] *refl*)
also have ... = (∑ *k*. *emeasure* (B ⊗_M B)
{*x*. *the* (*f* (*cshift* (*ctake* *k* (*fst* *x*)) (*snd* *x*))) ∈ A × B ∧ (*cshift* (*ctake* *k* (*fst* *x*)) (*snd* *x*)) ∈ C
k})
using *branch-coin-space*(1) 4 **by** (*subst* *emeasure-distr*)
(*simp-all* *add:case-prod-beta* *Int-def* *space-pair-measure* *space-coin-space*)
also have ... = (∑ *k*. *emeasure* (B ⊗_M B)
{*x*. *the* (*f* (*cshift* (*ctake* *k* (*fst* *x*)) (*snd* *x*))) ∈ A × B ∧ *fst* *x* ∈ C *k*})
using 8 **by** (*intro* *suminf-cong* *arg-cong2*[**where** *f=emeasure*] *refl* *Collect-cong*) *auto*
also have ... = (∑ *k*. *emeasure* (B ⊗_M B) ({*x*. *fst* (*the* (*f* *x*)) ∈ A ∧ *x* ∈ C *k*} × B))
using 7 **by** (*intro* *suminf-cong* *arg-cong2*[**where** *f=emeasure*] *refl*)
(*auto* *simp* *add:mem-Times-iff* *set-eq-iff*)
also have ... = (∑ *k*. *emeasure* B {*x*. *fst* (*the* (*f* *x*)) ∈ A ∧ *x* ∈ C *k*} * *emeasure* B B)
using 3 *that*(2)
by (*intro* *suminf-cong* *coin-space.emeasure-pair-measure-Times*) *auto*
also have ... = (∑ *k*. *emeasure* B {*x*. *fst* (*the* (*f* *x*)) ∈ A ∧ *x* ∈ C *k*}) * *emeasure* B B
by *simp*
also have ... = *emeasure* B (∪ *k*. {*x*. *fst* (*the* (*f* *x*)) ∈ A ∧ *x* ∈ C *k*}) * *emeasure* B B
using 3 *C-disj*
by (*intro* *arg-cong2*[**where** *f=(*)*] *suminf-emeasure* *refl* *image-subsetI*)
(*auto* *simp* *add:disjoint-family-on-def*)
also have ... = *emeasure* B {*x*. *fst* (*the* (*f* *x*)) ∈ A ∧ (∃ *k*. *x* ∈ C *k*)} * *emeasure* B B
by (*intro* *arg-cong2*[**where** *f=emeasure*] *arg-cong2*[**where** *f=(*)*]) *auto*
also have ... = *emeasure* R {*x*. *fst* (*the* (*f* *x*)) ∈ A ∧ *f* *x* ≠ None} * *emeasure* B B
unfolding *R-def* 2 **using** 1
by (*intro* *arg-cong2*[**where** *f=(*)*] *emeasure-restrict-space*[*symmetric*] *subsetI*) *simp-all*
also have ... = *emeasure* R ((*fst* ∘ *the* ∘ *f*) - ' A ∩ *space* R) * *emeasure* B B
unfolding *vimage-def* *R-def* *Int-def* **by** (*simp* *add:space-restrict-space* *space-coin-space*)
also have ... = ?R
using *that*
by (*intro* *arg-cong2*[**where** *f=(*)*] *emeasure-distr*[*symmetric*] *fst-the-f-measurable*) *auto*
finally show ?*thesis* **by** *simp*

qed

have *finite-measure* R
using 1 **unfolding** *R-def* *space-coin-space*
by (*intro* *finite-measure-restrict-space*) *simp-all*
hence *finite-measure* (*distr* R D (*fst* ∘ *the* ∘ *f*))
by (*intro* *finite-measure.finite-measure-distr* *fst-the-f-measurable*)
hence 1:*sigma-finite-measure* (*distr* R D (*fst* ∘ *the* ∘ *f*))
unfolding *finite-measure-def* **by** *auto*

```

have 2:sigma-finite-measure  $\mathcal{B}$ 
  using prob-space-imp-sigma-finite[OF coin-space.prob-space-axioms] by simp

show ?thesis
  using 0 by (intro pair-measure-eqI[symmetric] 1 2) (simp-all add:sets-pair-measure)
qed

end

lemma distr-rai-bind:
  assumes wf-m: wf-random m
  assumes wf-f:  $\bigwedge x. x \in \text{range-rm } m \implies \text{wf-random } (f x)$ 
  shows distr-rai (m  $\ggg$  f) = distr-rai m  $\ggg$ 
    ( $\lambda x. \text{if } x \in \text{Some } \text{'range-rm } m \text{ then } \text{distr-rai } (f \text{ (the } x)) \text{ else return } \mathcal{D} \text{ None}$ )
    (is ?L = ?RHS)
proof (rule measure-eqI)
  have sets ?L = UNIV
    unfolding distr-rai-def by simp
  also have ... = sets ?RHS
    unfolding distr-rai-def by (subst sets-bind[where N= $\mathcal{D}$ ])
    (simp-all add:option.case-distrib option.case-eq-if)
  finally show sets ?L = sets ?RHS by simp
next
  let ?m = distr-rai
  let ?H = count-space (range-rm m)
  let ?R = restrict-space  $\mathcal{B}$  {bs. m bs  $\neq$  None}

  fix A assume A  $\in$  sets (distr-rai (m  $\ggg$  f))
  define N where N = {x. m x  $\neq$  None}

  have N-meas: N  $\in$  sets coin-space
    unfolding N-def using R-sets[OF wf-m] by simp

  hence N-meas':  $-N \in$  sets coin-space
    unfolding Compl-eq-Diff-UNIV using space-coin-space by (metis sets.compl-sets)

  have wf-bind: wf-random (m  $\ggg$  f)
    using wf-bind[OF assms] by auto

  have 0: (map-option fst  $\circ$  (m  $\ggg$  f))  $\in$  coin-space  $\rightarrow_M \mathcal{D}$ 
    using distr-rai-measurable[OF wf-bind] by auto
  have 1: (map-option fst  $\circ$  (m  $\ggg$  f)) - ' A  $\in$  sets  $\mathcal{B}$ 
    unfolding vimage-def by (intro measurable-sets-coin-space[OF 0]) simp

  have {(v, bs). map-option fst (f v bs)  $\in$  A  $\wedge$  v  $\in$  range-rm m} =
    (map-option fst  $\circ$  case-prod f) - ' A  $\cap$  space (?H  $\otimes_M \mathcal{B}$ )
    unfolding vimage-def space-pair-measure space-coin-space by auto
  also have ...  $\in$  sets (?H  $\otimes_M \mathcal{B}$ )
    using distr-rai-measurable[OF wf-f]
    by (intro measurable-sets[where A= $\mathcal{D}$ ] measurable-pair-measure-countable1 countable-range
wf-m)
    (simp-all add:comp-def)
  also have ... = sets (restrict-space  $\mathcal{D}$  (range-rm m)  $\otimes_M \mathcal{B}$ )
    unfolding restrict-count-space inf-top-right by simp
  also have ... = sets (restrict-space ( $\mathcal{D} \otimes_M \mathcal{B}$ ) (range-rm m  $\times$  space coin-space))
    by (subst coin-space.restrict-space-pair-lift) auto
  finally have {(v, bs). map-option fst (f v bs)  $\in$  A  $\wedge$  v  $\in$  range-rm m}  $\in$ 
    sets (restrict-space ( $\mathcal{D} \otimes_M \mathcal{B}$ ) (range-rm m  $\times$  UNIV))

```

unfolding *space-coin-space* **by** *simp*
moreover have *range-rm m × space coin-space ∈ sets (D ⊗_M B)*
by (*intro pair-measureI sets.top*) *auto*
ultimately have 2: $\{(v, bs). \text{map-option fst } (f \ v \ bs) \in A \wedge v \in \text{range-rm } m\} \in \text{sets } (\mathcal{D} \otimes_M \mathcal{B})$
by (*subst (asm) sets-restrict-space-iff*) (*auto simp: space-coin-space*)

have *space-R: space ?R = {x. m x ≠ None}*
by (*simp add:space-restrict-space space-coin-space*)

have 3: *distr-rai (f (the x)) ∈ space (subprob-algebra D)*
if *x ∈ Some ‘range-rm m for x*
using *distr-rai-subprob-space[OF wf-f]* **that** **by** *fastforce*

have ($\lambda x. \text{emeasure } (\text{distr-rai } (f \ (\text{fst } (\text{the } (m \ x)))))) \ A \ * \ \text{indicator } N \ x =$
 $(\lambda x. \text{emeasure } (\text{if } m \ x \neq \text{None} \ \text{then } \text{distr-rai } (f \ (\text{fst } (\text{the } (m \ x)))) \ \text{else } \text{null-measure } \mathcal{D}) \ A)$
unfolding *N-def* **by** (*intro ext*) *simp*
also have ... = $(\lambda v. \text{emeasure } (\text{if } v \in \text{Some } \text{‘range-rm } m \ \text{then } ?m \ (f \ (\text{the } v)) \ \text{else } \text{null-measure } \mathcal{D})) \ A$
A)
 $\circ (\text{map-option fst} \circ m)$
unfolding *comp-def* **by** (*intro ext arg-cong2[where f=emeasure] refl if-cong*)
(auto intro:in-range-rmI simp add:vimage-def image-iff)
also have ... $\in \text{borel-measurable coin-space}$
using 3 **by** (*intro distr-rai-measurable[OF wf-m] measurable-comp[where N=D]*)
measurable-emeasure-kernel[where N=D] simp-all
finally have 4: $(\lambda x. \text{emeasure } (\text{distr-rai } (f \ (\text{fst } (\text{the } (m \ x)))))) \ A \ * \ \text{indicator } N \ x$
 $\in \text{coin-space} \rightarrow_M \text{borel}$ **by** *simp*

let $?N = \text{emeasure } \mathcal{B} \ \{bs. bs \notin N \wedge \text{None} \in A\}$

have *emeasure ?L A = emeasure B ((map-option fst ◦ (m ≫ f)) - ‘A)*
unfolding *distr-rai-def* **using** 0 **by** (*subst emeasure-distr*) (*simp-all add:space-coin-space*)
also have ... =
 $\text{emeasure } \mathcal{B} \ ((\text{map-option fst} \circ (m \gg f)) - ‘A \cap -N) + \text{emeasure } \mathcal{B} \ ((\text{map-option fst} \circ (m \gg f)) - ‘A \cap N)$
N)
using *N-meas N-meas’ 1*
by (*subst emeasure-Un[symmetric]*) (*simp-all add:Int-Un-distrib[symmetric]*)
also have ... =
 $\text{emeasure } \mathcal{B} \ ((\text{map-option fst} \circ (m \gg f)) - ‘A \cap -N) + \text{emeasure } ?R \ ((\text{map-option fst} \circ (m \gg f)) - ‘A \cap N)$
N)
using *N-meas* **unfolding** *N-def*
by (*intro arg-cong2[where f=(+)] refl emeasure-restrict-space[symmetric] simp-all*)
also have ... = $?N + \text{emeasure } ?R \ ((\text{the} \circ m) - ‘$
 $\{(v, bs). \text{map-option fst } (f \ v \ bs) \in A \wedge v \in \text{range-rm } m\} \cap \text{space } ?R)$
unfolding *bind-rai-def N-def space-R apfst-def*
by (*intro arg-cong2[where f=(+)] arg-cong2[where f=emeasure]*)
(simp-all add: set-eq-iff in-range-rmI split:option.split bind-splits)
also have ... = $?N + \text{emeasure } (\text{distr } ?R \ (\mathcal{D} \otimes_M \mathcal{B}) \ (\text{the} \circ m))$
 $\{(v, bs). \text{map-option fst } (f \ v \ bs) \in A \wedge v \in \text{range-rm } m\}$
using 2 **by** (*intro arg-cong2[where f=(+)] emeasure-distr[symmetric]*)
the-f-measurable map-prod-measurable wf-m) *simp-all*
also have ... = $?N + \text{emeasure } (\text{distr } ?R \ \mathcal{D} \ (\text{fst} \circ \text{the} \circ m) \otimes_M \mathcal{B})$
 $\{(v, bs). \text{map-option fst } (f \ v \ bs) \in A \wedge v \in \text{range-rm } m\}$
unfolding *N-def remainder-indep[OF wf-m]* **by** *simp*
also have ... = $?N + \int^+ v. \text{emeasure } \mathcal{B}$
 $\{bs. \text{map-option fst } (f \ v \ bs) \in A \wedge v \in \text{range-rm } m\} \ \partial \text{distr } ?R \ \mathcal{D} \ (\text{fst} \circ (\text{the} \circ m))$
using 2 **by** (*subst coin-space.emeasure-pair-measure-alt*) (*simp-all add:vimage-def comp-assoc*)
also have ... = $?N + \int^+ x. \text{emeasure } \mathcal{B}$
 $\{bs. \text{map-option fst } (f \ ((\text{fst} \circ (\text{the} \circ m)) \ x) \ bs) \in A \wedge (\text{fst} \circ (\text{the} \circ m)) \ x \in \text{range-rm } m\} \ \partial ?R$

using *the-f-measurable*[*OF wf-m*]
by (*intro arg-cong2*[**where** $f=(+)$] *refl nn-integral-distr*) *simp-all*
also have $\dots = ?N + (\int^+ x \in \{bs. m \ bs \neq \text{None}\}, \text{emeasure } \mathcal{B}$
 $\{bs. \text{map-option fst } (f \ (fst \ (the \ (m \ x)))) \ bs \in A \wedge \text{fst } (the \ (m \ x)) \in \text{range-rm } m\} \ \partial \mathcal{B})$
using *N-meas unfolding N-def using nn-integral-restrict-space*
by (*subst nn-integral-restrict-space*) *simp-all*
also have $\dots = ?N + (\int^+ x \in \{bs. m \ bs \neq \text{None}\},$
 $\text{emeasure } \mathcal{B} \ ((\text{map-option fst} \circ f \ (fst \ (the \ (m \ x)))) - 'A \cap \text{space } \mathcal{B}) \ \partial \mathcal{B})$
by (*intro arg-cong2*[**where** $f=(+)$] *set-nn-integral-cong refl arg-cong2*[**where** $f=\text{emeasure}$])
 $(\text{auto intro:in-range-rmI simp:space-coin-space})$
also have $\dots = ?N + (\int^+ x \in N. \text{emeasure } (\text{distr-rai}(f(\text{fst}(the(m \ x)))))) \ A \ \partial \mathcal{B})$
unfolding *distr-rai-def N-def*
by (*intro arg-cong2*[**where** $f=(+)$] *set-nn-integral-cong refl emeasure-distr*[*symmetric*]
 $\text{distr-rai-measurable}[*OF wf-f*]) $(\text{auto intro:in-range-rmI})$
also have $\dots = (\int^+ x. (\text{indicator } \{bs. bs \notin N \wedge \text{None} \in A\}) \ x \ \partial \mathcal{B}) +$
 $(\int^+ x \in N. \text{emeasure } (\text{distr-rai}(f(\text{fst}(the(m \ x)))))) \ A \ \partial \mathcal{B})$
using *N-meas N-meas'*
by (*intro arg-cong2*[**where** $f=(+)$] *nn-integral-indicator*[*symmetric*] *refl*)
 $(\text{cases } \text{None} \in A; \text{auto simp:Collect-neg-eq})$
also have $\dots = \int^+ x. \text{indicator } \{bs. bs \notin N \wedge \text{None} \in A\} \ x +$
 $\text{emeasure } (\text{distr-rai } (f \ (fst \ (the \ (m \ x)))))) \ A * \text{indicator } N \ x \ \partial \mathcal{B}$
using *N-meas' N-meas by (intro nn-integral-add*[*symmetric*] *4*) *simp*
also have $\dots = \int^+ x. \text{indicator } (-N) \ x * \text{indicator } A \ \text{None} +$
 $\text{indicator } N \ x * \text{emeasure } (\text{distr-rai } (f \ (fst \ (the \ (m \ x)))))) \ A \ \partial \mathcal{B}$
unfolding *N-def by (intro arg-cong2*[**where** $f=\text{nn-integral}$] *ext refl arg-cong2*[**where** $f=(+)$])
 $(\text{simp-all split:split-indicator})$
also have $\dots =$
 $\int^+ x. \text{emeasure } (\text{case } m \ x \ \text{of } \text{None} \Rightarrow \text{return } \mathcal{D} \ \text{None} \mid \text{Some } x \Rightarrow \text{distr-rai } (f \ (fst \ x))) \ A \ \partial \mathcal{B}$
unfolding *N-def by (intro arg-cong2*[**where** $f=\text{nn-integral}$] *ext*)
 $(\text{auto split:split-indicator option.split})$
also have $\dots = \int^+ x. \text{emeasure } (\text{if } (\text{map-option fst} \circ m) \ x \in \text{Some } ' \ \text{range-rm } m$
 $\text{then } \text{distr-rai } (f \ (the \ ((\text{map-option fst} \circ m) \ x)))$
 $\text{else } \text{return } \mathcal{D} \ \text{None}) \ A \ \partial \mathcal{B}$
by (*intro arg-cong2*[**where** $f=\text{nn-integral}$] *arg-cong2*[**where** $f=\text{emeasure}$] *refl ext*)
 $(\text{auto simp add: in-range-rmI vimage-def split:option.splits})$
also have $\dots =$
 $\int^+ x. \text{emeasure } (\text{if } x \in \text{Some } ' \ \text{range-rm } m \ \text{then } ?m \ (f \ (the \ x)) \ \text{else } \text{return } \mathcal{D} \ \text{None}) \ A \ \partial ?m \ m$
unfolding *distr-rai-def using distr-rai-measurable*[*OF wf-m*]
by (*intro nn-integral-distr*[*symmetric*]) $(\text{simp-all add:comp-def})$
also have $\dots = \text{emeasure } ?RHS \ A$
using *3 none-measure-subprob-algebra*
by (*intro emeasure-bind*[*symmetric, where N=D*]) $(\text{auto simp add:distr-rai-def Pi-def})$
finally show $\text{emeasure } ?L \ A = \text{emeasure } ?RHS \ A$
by *simp*
qed$

lemma *return-discrete*: $\text{return } \mathcal{D} \ x = \text{return-pmf } x$
by (*intro measure-eqI*) *auto*

lemma *distr-rai-return*: $\text{distr-rai } (\text{return-rai } x) = \text{return } \mathcal{D} \ (\text{Some } x)$
unfolding *return-rai-def distr-rai-def by (simp add:comp-def)*

lemma *distr-rai-return'*: $\text{distr-rai } (\text{return-rai } x) = \text{return-spmf } x$
unfolding *distr-rai-return return-discrete by auto*

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

proof –

have $?L = \text{distr } \mathcal{B} \ \mathcal{D} \ (\lambda x. \text{Some } (\text{chd } x))$

```

  unfolding coin-rai-def distr-rai-def by (simp add:comp-def)
also have ... = distr (distr  $\mathcal{B}$   $\mathcal{D}$  chd)  $\mathcal{D}$  Some
  by (subst distr-distr) (auto simp add:comp-def chd-measurable)
also have ... = map-pmf Some (pmf-of-set UNIV)
  unfolding distr-shd map-pmf-rep-eq by simp
also have ... = spmf-of-pmf (pmf-of-set UNIV)
  by (simp add:spmf-of-pmf-def)
also have ... = coin-spmf
  by auto
finally show ?thesis by simp
qed

```

```

definition ord-rai :: 'a random-alg-int  $\Rightarrow$  'a random-alg-int  $\Rightarrow$  bool
  where ord-rai = fun-ord (flat-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
  unfolding ord-rai-def lub-rai-def
  by (intro partial-function-lift flat-interpretation)

```

```

interpretation random-alg-int-pd: partial-function-definitions ord-rai lub-rai
  by (rule random-alg-int-pd-fact)

```

```

lemma wf-lub-helper:
  assumes ord-rai f g
  assumes wf-on-prefix f p r
  shows wf-on-prefix g p r
proof -
  have g (cshift p cs) = Some (r, cs) for cs
  proof -
  have f (cshift p cs) = Some (r,cs)
    using asms(2) unfolding wf-on-prefix-def by auto
  moreover have flat-ord None (f (cshift p cs)) (g (cshift p cs))
    using asms(1) unfolding ord-rai-def fun-ord-def by simp
  ultimately show ?thesis
    unfolding flat-ord-def by auto
  qed
thus ?thesis
  unfolding wf-on-prefix-def by auto
qed

```

```

lemma wf-lub:
  assumes Complete-Partial-Order.chain ord-rai R
  assumes  $\bigwedge r. r \in R \implies$  wf-random r
  shows wf-random (lub-rai R)
proof (rule wf-randomI)
  fix bs
  assume a:lub-rai R bs  $\neq$  None
  define S where S = (( $\lambda x. x$  bs) ' R)
  have 0:lub-rai R bs = flat-lub None S
    unfolding S-def lub-rai-def fun-lub-def
    by (intro arg-cong2[where f=flat-lub]) auto

  have lub-rai R bs = None if S  $\subseteq$  {None}
    using that unfolding 0 flat-lub-def by auto

```

hence $\neg (S \subseteq \{None\})$
 using a by *auto*
 then obtain r where $1:r \in R$ and $2: r \text{ bs} \neq None$
 unfolding $S\text{-def}$ by *blast*
 then obtain $p \ y$ where $3:\text{cprefix } p \text{ bs}$ and $4:\text{wf-on-prefix } r \ p \ y$
 using $\text{assms}(2)[OF \ 1] \ 2$ unfolding wf-random-def by $(\text{auto } \text{split}:\text{option.split-asm})$
 have $\text{wf-on-prefix } (\text{lub-rai } R) \ p \ y$
 by $(\text{intro } \text{wf-lub-helper}[OF \ - \ 4] \ \text{random-alg-int-pd.lub-upper } 1 \ \text{assms}(1))$
 thus $\exists p \ r. \ \text{cprefix } p \ \text{bs} \wedge \ \text{wf-on-prefix } (\text{lub-rai } R) \ p \ r$
 using 3 by *auto*
 qed

lemma *ord-rai-mono*:
 assumes $\text{ord-rai } f \ g$
 assumes $\neg (P \ None)$
 assumes $P \ (f \ \text{bs})$
 shows $P \ (g \ \text{bs})$
 using assms unfolding $\text{ord-rai-def } \text{fun-ord-def } \text{flat-ord-def}$ by *metis*

lemma *lub-rai-empty*:
 $\text{lub-rai } \{\} = \text{Map.empty}$
 unfolding $\text{lub-rai-def } \text{fun-lub-def } \text{flat-lub-def}$ by *simp*

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

proof –
 have $\text{wf-lub}: \text{wf-random } (\text{lub-rai } F)$
 by $(\text{intro } \text{wf-lub } \text{assms})$

have $4: \text{ord-rai } f \ (\text{lub-rai } F)$ if $f \in F$ for f
 using $\text{that } \text{random-alg-int-pd.lub-upper}[OF \ \text{assms}(2)]$ by *simp*

have $0: \text{map-option fst } (\text{lub-rai } F \ \text{bs}) \in A \iff (\exists f \in F. \ \text{map-option fst } (f \ \text{bs}) \in A)$ for bs
proof

assume $\exists f \in F. \ \text{map-option fst } (f \ \text{bs}) \in A$
 then obtain f where $3: \text{map-option fst } (f \ \text{bs}) \in A$ and $5: f \in F$
 by *auto*
 show $\text{map-option fst } (\text{lub-rai } F \ \text{bs}) \in A$
 by $(\text{rule } \text{ord-rai-mono}[OF \ 4[OF \ 5]])$ (use 3 $\text{assms}(4)$ in *auto*)

next
 assume $\text{map-option fst } (\text{lub-rai } F \ \text{bs}) \in A$
 then obtain y where $6: \text{lub-rai } F \ \text{bs} = \text{Some } y$ and $\text{Some } (fst \ y) \in A$
 using $\text{assms}(4)$ by $(\text{cases } \text{lub-rai } F \ \text{bs})$ *auto*
 hence $f \ \text{bs} = None \vee f \ \text{bs} = \text{Some } y$ if $f \in F$ for f
 using $4[OF \ \text{that}]$ unfolding $\text{ord-rai-def } \text{fun-ord-def } \text{flat-ord-def}$ by *auto*
 moreover have $\text{lub-rai } F \ \text{bs} = None$ if $\bigwedge f. f \in F \implies f \ \text{bs} = None$
 using that unfolding $\text{lub-rai-def } \text{flat-lub-def } \text{fun-lub-def}$ by *auto*
 ultimately obtain f where $f \ \text{bs} = \text{Some } y \ f \in F$
 using $6(1)$ by *auto*
 thus $\exists f \in F. \ \text{map-option fst } (f \ \text{bs}) \in A$
 using $6(2)$ by *force*

qed

have $1: \text{Complete-Partial-Order.chain } (\subseteq) \ ((\lambda f. \ \{\text{bs}. \ \text{map-option fst } (f \ \text{bs}) \in A\}) \text{ ` } F)$

using *assms*(4) by (intro *chain-imageI*[*OF assms*(2)] *Collect-mono impI*) (auto intro:*ord-rai-mono*)

have 2: open {bs. map-option fst (f bs) ∈ A} (is open ?T) if f ∈ F for f
proof –

have wf-f': wf-random f
by (intro *assms* that)

have 4: ?T = {bs ∈ *topspace euclidean*. (map-option fst ∘ f) bs ∈ A}
by *simp*

have *openin option-ud* A

using *assms*(4) **unfolding** *openin-option-ud* by *simp*

hence *openin euclidean* ?T

unfolding 4 by (intro *openin-continuous-map-preimage*[*OF f-continuous*] wf-f')

thus ?thesis

using *open-openin* by *simp*

qed

have 3: {bs. map-option fst (f bs) ∈ A} ∈ sets B (is ?L1 ∈ -) if wf-random f for f

using *distr-rai-measurable*[*OF that*]

by (intro *measurable-sets-coin-space*[**where** P=λx. x ∈ A **and** A=D]) (auto *simp:comp-def*)

have ?L = *emeasure* B ((map-option fst ∘ lub-rai F) - ' A ∩ space B)

unfolding *distr-rai-def* by (intro *emeasure-distr distr-rai-measurable*[*OF wf-lub*]) auto

also have ... = *emeasure* B {x. map-option fst (lub-rai F x) ∈ A}

unfolding *space-coin-space* by (*simp add:vimage-def*)

also have ... = *emeasure* B (∪ f ∈ F. {bs. map-option fst (f bs) ∈ A})

unfolding 0 by (intro *arg-cong2*[**where** f=*emeasure*]) auto

also have ... = *Sup* (*emeasure* B ' (λf. {bs. map-option fst (f bs) ∈ A}) ' F)

using 2 by (intro *tau-additivity*[*OF coin-space-is-borel-measure*] *chain-imp-union-stable* 1)

auto

also have ... = (*SUP* f ∈ F. (*emeasure* B {bs. map-option fst (f bs) ∈ A}))

unfolding *image-image* by *simp*

also have ... = (*SUP* f ∈ F. *emeasure* B ((map-option fst ∘ f) - ' A ∩ space B))

by (*simp add:image-image space-coin-space vimage-def*)

also have ... = ?R

unfolding *distr-rai-def* using *distr-rai-measurable*[*OF wf-f*]

by (intro *arg-cong*[**where** f=(*Sup*)] *image-cong ext emeasure-distr*[*symmetric*]) auto

finally show ?thesis

by *simp*

qed

lemma *distr-rai-ord-rai-mono*:

assumes wf-random f wf-random g *ord-rai* f g

assumes None ∉ A

shows *emeasure* (*distr-rai* f) A ≤ *emeasure* (*distr-rai* g) A (is ?L ≤ ?R)

proof –

have 0: *Complete-Partial-Order.chain ord-rai* {f,g}

using *assms*(3) **unfolding** *Complete-Partial-Order.chain-def*

using *random-alg-int-pd.leq-refl* by auto

have *ord-rai* (lub-rai {f,g}) g

using *assms*(3) *random-alg-int-pd.leq-refl*

by (intro *random-alg-int-pd.lub-least* 0) auto

moreover have *ord-rai* g (lub-rai {f,g})

by (intro *random-alg-int-pd.lub-upper* 0) *simp*

ultimately have 1: g = lub-rai {f,g}

by (intro *random-alg-int-pd.leq-antisym*) auto

have *emeasure* (*distr-rai* f) A ≤ (*SUP* x ∈ {f,g}. *emeasure* (*distr-rai* x) A)

using *prob-space-distr-rai assms*(1,2) *prob-space.measure-le-1*

```

  by (intro cSup-upper bdd-aboveI[where M=1]) auto
also have ... = emeasure (distr-rai (lub-rai {f,g})) A
  using assms by (intro distr-rai-lub[symmetric] 0) auto
also have ... = emeasure (distr-rai g) A
  using 1 by auto
finally show ?thesis
  by simp
qed

```

lemma *distr-rai-None*: $\text{distr-rai } (\lambda\cdot. \text{None}) = \text{measure-pmf } (\text{return-pmf } (\text{None} :: 'a \text{ option}))$

proof –

```

  have emeasure (distr-rai Map.empty) A = emeasure (measure-pmf (return-pmf None)) A
  for A :: 'a option set
  using coin-space.emeasure-space-1 unfolding distr-rai-def
  by (subst emeasure-distr) simp-all
  thus ?thesis
  by (intro measure-eqI) (simp-all add:distr-rai-def)

```

qed

lemma *bind-rai-mono*:

```

  assumes ord-rai f1 f2  $\wedge$  y. ord-rai (g1 y) (g2 y)
  shows ord-rai (bind-rai f1 g1) (bind-rai f2 g2)

```

proof –

```

  have flat-ord None (bind-rai f1 g1 bs) (bind-rai f2 g2 bs) for bs

```

proof (cases $f1 \ggg g1$) bs)

case None

then show ?thesis by (simp add:flat-ord-def)

next

case (Some a)

then obtain $y \text{ bs}'$ where 0: $f1 \text{ bs} = \text{Some } (y, \text{bs}')$ and 1: $g1 \text{ y } \text{bs}' \neq \text{None}$ and $f1 \text{ bs} \neq \text{None}$

by (cases $f1 \text{ bs}$, auto simp:bind-rai-def)

hence $f2 \text{ bs} = f1 \text{ bs}$

using assms(1) unfolding ord-rai-def fun-ord-def flat-ord-def by metis

hence $f2 \text{ bs} = \text{Some } (y, \text{bs}')$

using 0 by auto

moreover have $g1 \text{ y } \text{bs}' = g2 \text{ y } \text{bs}'$

using assms(2) 1 unfolding ord-rai-def fun-ord-def flat-ord-def by metis

ultimately have $(f1 \ggg g1) \text{ bs} = (f2 \ggg g2) \text{ bs}$

unfolding bind-rai-def 0 by auto

thus ?thesis unfolding flat-ord-def by auto

qed

thus ?thesis

unfolding ord-rai-def fun-ord-def by simp

qed

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 the-

orems, 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 -
  have  $(\int x. pmf\ q\ x - pmf\ p\ x\ \partial count\ space\ UNIV) = 0$ 
    by (simp-all add:integrable-pmf integral-pmf)
  moreover have integrable (count-space UNIV)  $(\lambda x. pmf\ q\ x - pmf\ p\ x)$ 
    by (simp add:integrable-pmf)
  moreover have AE x in count-space UNIV.  $0 \leq pmf\ q\ x - pmf\ p\ x$ 
    using assms unfolding AE-count-space by auto
  ultimately have AE x in count-space UNIV.  $pmf\ q\ x - pmf\ p\ x = 0$ 
    using integral-nonneg-eq-0-iff-AE by blast
  hence  $\bigwedge x. pmf\ p\ x = pmf\ q\ x$  unfolding AE-count-space by simp
  thus ?thesis by (intro pmf-eqI) auto
qed
```

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

```
lemma eq-iff-ord-spmf:
  assumes weight-spmf p  $\geq$  weight-spmf q
  assumes ord-spmf (=) p q
  shows  $p = q$ 
proof -
  have  $\bigwedge x. spmf\ p\ x \leq spmf\ q\ x$ 
    using ord-spmf-eq-leD[OF assms(2)] by simp
  moreover have pmf p None  $\leq$  pmf q None
    using assms(1) unfolding pmf-None-eq-weight-spmf by auto
  ultimately have pmf p x  $\leq$  pmf q x for x by (cases x) auto
  thus ?thesis using pmf-eq-iff-le by auto
qed
```

```
lemma wf-empty: wf-random  $(\lambda-. None)$ 
  unfolding wf-random-def by auto
```

```
typedef 'a random-alg = {(r :: 'a random-alg-int). wf-random r}
  using wf-empty by (intro exI[where x= $\lambda-. None$ ]) auto
```

```
setup-lifting type-definition-random-alg
```

```
lift-definition return-ra :: 'a  $\Rightarrow$  'a random-alg is return-rai
  by (rule wf-return)
```

```
lift-definition coin-ra :: bool random-alg is coin-rai
  by (rule wf-coin)
```

```
lift-definition bind-ra :: 'a random-alg  $\Rightarrow$  ('a  $\Rightarrow$  'b random-alg)  $\Rightarrow$  'b random-alg is bind-rai
  by (rule wf-bind)
```

```
adhoc-overloading Monad-Syntax.bind  $\equiv$  bind-ra
```

Monad laws:

lemma *return-bind-ra*:

bind-ra (return-ra x) g = g x
by (*rule return-bind-rai[transferred]*)

lemma *bind-ra-assoc*:

bind-ra (bind-ra f g) h = bind-ra f ($\lambda x. bind-ra (g x) h$)
by (*rule bind-rai-assoc[transferred]*)

lemma *bind-return-ra*:

bind-ra m return-ra = m
by (*rule bind-return-rai[transferred]*)

lift-definition *lub-ra* :: 'a random-alg set \Rightarrow 'a random-alg **is**

($\lambda F. \text{if Complete-Partial-Order.chain ord-rai } F \text{ then lub-rai } F \text{ else } (\lambda x. \text{None}))$)
using *wf-lub wf-empty* **by** *auto*

lift-definition *ord-ra* :: 'a random-alg \Rightarrow 'a random-alg \Rightarrow bool **is** *ord-rai* .

lift-definition *run-ra* :: 'a random-alg \Rightarrow coin-stream \Rightarrow 'a option **is**

($\lambda f s. \text{map-option fst } (f s)$) .

context

begin

interpretation *pmf-as-measure* .

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} \neq 0 (is ?C)

proof –

show *prob-space (distr-rai f)*
using *prob-space-distr-rai[OF assms]* **by** *simp*
then interpret *p: prob-space distr-rai f*
by *auto*
show *?B*
unfolding *distr-rai-def* **by** *simp*

have *AE bs in \mathcal{B} . map-option fst (f bs) \in Some ' range-rm f \cup {None}*

unfolding *range-rm-def*
by (*intro AE-I2*) (*auto simp:image-iff split:option.split*)

hence *AE x in distr-rai f. x \in Some ' range-rm f \cup {None}*

unfolding *distr-rai-def* **using** *distr-rai-measurable[OF assms]*
by (*subst AE-distr-iff*) *auto*

moreover have *countable (Some ' range-rm f \cup {None})*

using *countable-range[OF assms]* **by** *simp*

moreover have *p.events = UNIV*

unfolding *distr-rai-def* **by** *simp*

ultimately show *?C*

by (*intro iffD2[OF p.AE-support-countable] exI[where x= Some ' range-rm f \cup {None}]*) *auto*

qed

lift-definition *spmf-of-ra* :: 'a random-alg \Rightarrow 'a *spmf* **is** *distr-rai*

using *distr-rai-is-pmf* **by** *metis*

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 –

show *prob-space (used-bits-distr f)*

unfolding *used-bits-distr-def*

by (*intro coin-space.prob-space-distr consumed-bits-measurable*)

then interpret *p: prob-space used-bits-distr f*

by *auto*

show *?B*

unfolding *used-bits-distr-def* **by** *simp*

have *p.events = UNIV*

unfolding *used-bits-distr-def* **by** *simp*

thus *?C*

by (*intro iffD2[OF p.AE-support-countable] exI[where x= UNIV]*) *auto*

qed

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

using *used-bits-distr-is-pmf* **by** *auto*

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)

using *Rep-random-alg* **by** *auto*

lemma *set-pmf-spmf-of-ra*:

set-pmf (spmf-of-ra f) ⊆ Some ‘ range-rm (Rep-random-alg f) ∪ {None}

proof

let *?f = Rep-random-alg f*

fix *x* **assume** *x ∈ set-pmf (spmf-of-ra f)*

hence *pmf (spmf-of-ra f) x > 0*

using *pmf-positive* **by** *metis*

hence *measure (distr-rai ?f) {x} > 0*

by (*subst spmf-of-ra.rep-eq[symmetric]*) (*simp add: pmf.rep-eq*)

hence *0 < measure B {ω. map-option fst (?f ω) = x}*

using *distr-rai-measurable[OF wf-rep-rand-alg]* **unfolding** *distr-rai-def*

by (*subst (asm) measure-distr*) (*simp-all add: vimage-def space-coin-space*)

moreover **have** *{ω. map-option fst (?f ω) = x} = {}* **if** *x ∉ range (map-option fst ∘ ?f)*

using *that* **by** (*auto simp:set-eq-iff image-iff*)

hence *measure B {ω. map-option fst (?f ω) = x} = 0* **if** *x ∉ range (map-option fst ∘ ?f)*

using *that* **by** *simp*

ultimately **have** *x ∈ range (map-option fst ∘ ?f)*

by *auto*

thus *x ∈ Some ‘ range-rm (Rep-random-alg f) ∪ {None}*

unfolding *range-rm-def* **by** (*cases x*) *auto*

qed

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

proof –

have *measure-pmf (spmf-of-ra (return-ra x)) = measure-pmf (return-spmf x)*

unfolding *spmf-of-ra.rep-eq distr-rai-return'[symmetric]*
by (*simp add: return-ra.rep-eq*)
thus *?thesis*
using *measure-pmf-inject* **by** *blast*
qed

lemma *spmf-of-ra-coin: spmf-of-ra coin-ra = coin-spmf*

proof –
have *measure-pmf (spmf-of-ra coin-ra) = measure-pmf coin-spmf*
unfolding *spmf-of-ra.rep-eq distr-rai-coin[symmetric]*
by (*simp add: coin-ra.rep-eq*)
thus *?thesis*
using *measure-pmf-inject* **by** *blast*
qed

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 –

let *?f = Rep-random-alg f*
let *?g = λx. Rep-random-alg (g x)*

have *0: x ∈ Some 'range-rm ?f ∨ x = None* **if** *x ∈ set-pmf (spmf-of-ra f)* **for** *x*
using *that set-pmf-spmf-of-ra* **by** *auto*

have *measure-pmf ?L = distr-rai (?f ≫ ?g)*

unfolding *spmf-of-ra.rep-eq bind-ra.rep-eq* **by** (*simp add:comp-def*)

also have *... = distr-rai ?f ≫*

(λx. if x ∈ Some 'range-rm ?f then distr-rai (?g (the x)) else return D None)

by (*intro distr-rai-bind wf-rep-rand-alg*)

also have *... = measure-pmf (spmf-of-ra f) ≫*

(λx. measure-pmf (if x ∈ Some 'range-rm ?f then spmf-of-ra (g (the x)) else return-pmf None))

by (*intro arg-cong2[where f=bind] ext*) (*auto simp:spmf-of-ra.rep-eq return-discrete*)

also have *... = measure-pmf (spmf-of-ra f) ≫*

(λx. if x ∈ Some 'range-rm ?f then spmf-of-ra (g (the x)) else return-pmf None)

unfolding *bind-pmf.rep-eq* **by** (*simp add:comp-def id-def*)

also have *... = measure-pmf ?R*

using *0* **unfolding** *bind-spmf-def*

by (*intro arg-cong[where f=measure-pmf] bind-pmf-cong refl*) (*auto split:option.split*)

finally have *measure-pmf ?L = measure-pmf ?R* **by** *simp*

thus *?thesis*

using *measure-pmf-inject* **by** *blast*

qed

lemma *spmf-of-ra-mono:*

assumes *ord-ra f g*

shows *ord-spmf (=) (spmf-of-ra f) (spmf-of-ra g)*

proof –

have *ord-rai (Rep-random-alg f) (Rep-random-alg g)*

using *assms* **unfolding** *ord-ra.rep-eq* **by** *simp*

hence *ennreal (spmf (spmf-of-ra f) x) ≤ ennreal (spmf (spmf-of-ra g) x)* **for** *x*

unfolding *emeasure-pmf-single[symmetric] spmf-of-ra.rep-eq*

by (*intro distr-rai-ord-rai-mono wf-rep-rand-alg*) *auto*

hence *spmf (spmf-of-ra f) x ≤ spmf (spmf-of-ra g) x* **for** *x*

by *simp*

thus *?thesis*

by (*intro ord-pmf-increaseI*) *auto*

qed

lemma *spmf-of-ra-lub-ra-empty*:

spmf-of-ra (lub-ra {}) = return-pmf None (is ?L = ?R)

proof –

have *measure-pmf* ?L = *distr-rai* (lub-rai {})

unfolding *spmf-of-ra.rep-eq* lub-ra.rep-eq *Complete-Partial-Order.chain-def* by auto

also have ... = *distr-rai* (λ-. None)

unfolding *lub-rai-def* *fun-lub-def* *flat-lub-def* by auto

also have ... = *measure-pmf* ?R

unfolding *distr-rai-None* by simp

finally have *measure-pmf* ?L = *measure-pmf* ?R

by simp

thus ?thesis

using *measure-pmf-inject* by auto

qed

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 (*cases* *A* ≠ {})

case *True*

have 0: *Complete-Partial-Order.chain* ord-rai (*Rep-random-alg* ‘ *A*)

using *assms* unfolding ord-ra.rep-eq *Complete-Partial-Order.chain-def* by auto

have 1: *Complete-Partial-Order.chain* (ord-spmf (=)) (*spmf-of-ra* ‘ *A*)

using *spmf-of-ra-mono* by (intro *chain-imageI*[*OF* *assms*]) auto

show ?thesis

proof (*rule* *spmf-eqI*)

fix *x* :: 'a

have *ennreal* (*spmf* ?L *x*) = *emeasure* (*distr-rai* (lub-rai (*Rep-random-alg* ‘ *A*))) {*Some* *x*}

using 0 unfolding *emeasure-pmf-single*[*symmetric*] *spmf-of-ra.rep-eq* lub-ra.rep-eq by simp

also have ... = (*SUP* *f* ∈ *Rep-random-alg* ‘ *A*. *emeasure* (*distr-rai* *f*) {*Some* *x*})

using *True* *wf-rep-rand-alg* by (intro *distr-rai-lub* 0) auto

also have ... = (*SUP* *p* ∈ *A*. *ennreal* (*spmf* (*spmf-of-ra* *p*) *x*))

unfolding *emeasure-pmf-single*[*symmetric*] *spmf-of-ra.rep-eq* by (*simp* *add:image-image*)

also have ... = (*SUP* *p* ∈ *spmf-of-ra* ‘ *A*. *ennreal* (*spmf* *p* *x*))

by (*simp* *add:image-image*)

also have ... = *ennreal* (*spmf* ?R *x*)

using *True* by (intro *ennreal-spmf-lub-spmf*[*symmetric*] 1) auto

finally have *ennreal* (*spmf* ?L *x*) = *ennreal* (*spmf* ?R *x*)

by simp

thus *spmf* ?L *x* = *spmf* ?R *x*

by simp

qed

next

case *False*

thus ?thesis using *spmf-of-ra-lub-ra-empty* by simp

qed

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 –

have *Complete-Partial-Order.chain* ord-rai (*Rep-random-alg* ‘ *F*)

using *assms* unfolding ord-ra.rep-eq *Complete-Partial-Order.chain-def* by auto

thus ?thesis

unfolding *lub-ra.rep-eq* by simp

qed

lemma *partial-function-image-improved*:

fixes *ord*

assumes $\bigwedge A. \text{Complete-Partial-Order.chain } ord (f \text{ ' } A) \implies l1 (f \text{ ' } A) = f (l2 A)$

assumes *partial-function-definitions ord l1*

assumes *inj f*

shows *partial-function-definitions (img-ord f ord) l2*

proof –

interpret *pd: partial-function-definitions ord l1*

using *assms(2)* by *auto*

have *img-ord f ord x x* for *x*

unfolding *img-ord-def* using *pd.leq-refl* by *simp*

moreover have *img-ord f ord x z* if *img-ord f ord x y* *img-ord f ord y z* for *x y z*

using *that pd.leq-trans* unfolding *img-ord-def* by *blast*

moreover have *x = y* if *img-ord f ord x y* *img-ord f ord y x* for *x y*

proof –

have *f x = f y*

using *that pd.leq-antisym* unfolding *img-ord-def* by *blast*

thus *?thesis*

using *inj-onD[OF assms(3)]* by *simp*

qed

moreover have *img-ord f ord x (l2 A)*

if $x \in A$ *Complete-Partial-Order.chain (img-ord f ord) A* for $x \in A$

proof –

have $0: \text{Complete-Partial-Order.chain } ord (f \text{ ' } A)$

using *that(2)* unfolding *chain-def img-ord-def* by *auto*

have *ord (f x) (l1 (f ' A))*

using *that* by (*intro pd.lub-upper[OF 0]*) *auto*

thus *?thesis*

unfolding *img-ord-def assms(1)[OF 0]* by *auto*

qed

moreover have *img-ord f ord (l2 A) z*

if *Complete-Partial-Order.chain (img-ord f ord) A* $(\forall x. x \in A \longrightarrow \text{img-ord f ord } x z)$

for $z \in A$

proof –

have $0: \text{Complete-Partial-Order.chain } ord (f \text{ ' } A)$

using *that(1)* unfolding *chain-def img-ord-def* by *auto*

have *ord (l1 (f ' A)) (f z)*

using *that(2)* by (*intro pd.lub-least[OF 0]*) (*auto simp:img-ord-def*)

thus *?thesis*

unfolding *img-ord-def assms(1)[OF 0]* by *auto*

qed

ultimately show *?thesis*

unfolding *partial-function-definitions-def* by *blast*

qed

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

proof –

have $0: \text{inj Rep-random-alg}$

using *Rep-random-alg-inject* unfolding *inj-on-def* by *auto*

have $1: \text{partial-function-definitions ord-rai lub-rai}$

using *random-alg-int-pd-fact* by *simp*

have $2: \text{ord-ra} = \text{img-ord Rep-random-alg ord-rai}$

unfolding *ord-ra.rep-eq img-ord-def* by *auto*

show *?thesis*

unfolding 2 by (intro partial-function-image-improved[OF - 1 0]) (auto simp: lub-ra.rep-eq)
qed

interpretation random-alg-pf: partial-function-definitions ord-ra lub-ra
using random-alg-pfd by auto

abbreviation mono-ra \equiv 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)
using assms **unfolding** ord-ra.rep-eq bind-ra.rep-eq
by (intro bind-rai-mono) auto

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))
using assms by (intro monotoneI bind-mono-aux-ra) (auto simp: monotone-def)

definition map-ra :: ('a \Rightarrow 'b) \Rightarrow 'a random-alg \Rightarrow 'b random-alg
where map-ra f p = p \ggg (λx . return-ra (f x))

lemma spmf-of-ra-map: spmf-of-ra (map-ra f p) = map-spmf f (spmof-of-ra p)
unfolding map-ra-def map-spmf-conv-bind-spmf spmf-of-ra-bind spmf-of-ra-return by simp

lemmas spmf-of-ra-simps =

spmof-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))
using assms **unfolding** map-ra-def by (intro bind-mono-ra) auto

definition rel-spmf-of-ra :: 'a spmf \Rightarrow 'a random-alg \Rightarrow bool **where**
rel-spmf-of-ra q p \longleftrightarrow 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 (rule ccpo.admissibleI)

fix Y

assume chain: Complete-Partial-Order.chain ?ord Y

and Y: Y \neq {}

and R: $\forall (p, q) \in Y$. rel-spmf-of-ra p q

from R have R: $\bigwedge p q$. (p, q) \in Y \implies rel-spmf-of-ra p q by auto

have chain1: Complete-Partial-Order.chain (ord-spmf (=)) (fst ' Y)

and chain2: Complete-Partial-Order.chain (ord-ra) (snd ' Y)

using chain by(rule chain-imageI; clarsimp)+

from Y have Y1: fst ' Y \neq {} and Y2: snd ' Y \neq {} by auto

have lub-spmf (fst ' Y) = lub-spmf (spmof-of-ra ' snd ' Y)

unfolding image-image using R

by (intro arg-cong[of - - lub-spmf] image-cong) (auto simp: rel-spmf-of-ra-def)

also have ... = spmf-of-ra (lub-ra (snd ' Y))

by (intro spmf-of-ra-lub-ra[symmetric] chain2)

finally have rel-spmf-of-ra (lub-spmf (fst ' Y)) (lub-ra (snd ' Y))

unfolding rel-spmf-of-ra-def .

then show ?P (?lub Y)

by (simp add: prod-lub-def)
qed

lemma *admissible-rel-spmf-of-ra-cont* [cont-intro]:
 fixes *ord*
 shows \llbracket *mcont lub ord lub-spmf (ord-spmf (=)) f*; *mcont lub ord lub-ra ord-ra g* \rrbracket
 \implies *ccpo.admissible lub ord* ($\lambda x.$ *rel-spmf-of-ra (f x) (g x)*)
 by (rule *admissible-subst*[OF *admissible-rel-spmf-of-ra*, **where** $f=\lambda x.$ (*f x, g x*), *simplified*])
 (rule *mcont-Pair*)

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*
 unfolding *mcont-def monotone-def cont-def*
 by (auto simp: *spmf-of-ra-mono spmf-of-ra-lub-ra*)

context
 includes *lifting-syntax*
begin

lemma *fixp-ra-parametric*[*transfer-rule*]:
 assumes $f: \bigwedge x. \text{mono-spmf } (\lambda f. F f x)$
 and $g: \bigwedge x. \text{mono-ra } (\lambda f. G f x)$
 and *param*: $((A \text{====>} \text{rel-spmf-of-ra}) \text{====>} A \text{====>} \text{rel-spmf-of-ra}) F G$
 shows $(A \text{====>} \text{rel-spmf-of-ra}) (\text{spmf.fixp-fun } F) (\text{random-alg-pf.fixp-fun } G)$
 using *f g*

proof(rule *parallel-fixp-induct-1-1*[OF
partial-function-definitions-spmf random-alg-pfd - - reflexive reflexive,
where $P=(A \text{====>} \text{rel-spmf-of-ra})$])
 show *ccpo.admissible (prod-lub (fun-lub lub-spmf) (fun-lub lub-ra))*
 (*rel-prod (fun-ord (ord-spmf (=))) (fun-ord ord-ra)*)
 ($\lambda x. (A \text{====>} \text{rel-spmf-of-ra}) (\text{fst } x) (\text{snd } x)$)
 unfolding *rel-fun-def*
 by(rule *admissible-all admissible-imp cont-intro*)+
 show $(A \text{====>} \text{rel-spmf-of-ra}) (\lambda-. \text{lub-spmf } \{\}) (\lambda-. \text{lub-ra } \{\})$
 by (auto simp: *rel-fun-def rel-spmf-of-ra-def spmf-of-ra-lub-ra-empty*)
 show $(A \text{====>} \text{rel-spmf-of-ra}) (F f) (G g)$ **if** $(A \text{====>} \text{rel-spmf-of-ra}) f g$ **for** *f g*
 using *that* by(rule *rel-funD*[OF *param*])
 qed

lemma *return-ra-transfer*[*transfer-rule*]: $((=) \text{====>} \text{rel-spmf-of-ra}) \text{return-spmf return-ra}$
 unfolding *rel-fun-def rel-spmf-of-ra-def spmf-of-ra-return* **by** *simp*

lemma *bind-ra-transfer*[*transfer-rule*]:
 $(\text{rel-spmf-of-ra} \text{====>} ((=) \text{====>} \text{rel-spmf-of-ra}) \text{====>} \text{rel-spmf-of-ra}) \text{bind-spmf bind-ra}$
 unfolding *rel-fun-def rel-spmf-of-ra-def spmf-of-ra-bind* **by** *simp presburger*

lemma *coin-ra-transfer*[*transfer-rule*]:
rel-spmf-of-ra coin-spmf coin-ra
 unfolding *rel-fun-def rel-spmf-of-ra-def spmf-of-ra-coin* **by** *simp*

lemma *map-ra-transfer*[*transfer-rule*]:
 $((=) \text{====>} \text{rel-spmf-of-ra} \text{====>} \text{rel-spmf-of-ra}) \text{map-spmf map-ra}$
 unfolding *rel-fun-def rel-spmf-of-ra-def spmf-of-ra-map* **by** *simp*

end

declare \llbracket *function-internals* \rrbracket

declaration \langle Partial-Function.init random-alg **term** \langle random-alg-pf.fixp-fun \rangle
term \langle random-alg-pf.mono-body \rangle
 $\@$ {thm random-alg-pf.fixp-rule-uc} $\@$ {thm random-alg-pf.fixp-induct-uc}
 NONE \rangle

5.1 Almost surely terminating randomized algorithms

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

definition *pmf-of-ra* :: 'a random-alg \Rightarrow '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
by (simp add:map-pmf-comp spmf-of-pmf-def)

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 –

have 0:spmf-of-ra (coin-ra) = spmf-of-pmf (pmf-of-set UNIV)
unfolding spmf-of-ra-coin spmf-of-set-def **by** simp
thus ?thesis
unfolding 0 pmf-of-ra-def pmf-of-spmf coin-pmf-def **by** simp
 qed

lemma *pmf-of-ra-return*: pmf-of-ra (return-ra x) = return-pmf x
unfolding pmf-of-ra-def spmf-of-ra-return **by** simp

lemma *pmf-of-ra-bind*:

assumes *terminates-almost-surely* f
shows pmf-of-ra (f \ggg g) = pmf-of-ra f \ggg (λ x. pmf-of-ra (g x)) (is ?L = ?R)

proof –

have 0:x \neq None **if** x \in set-pmf (spmf-of-ra f) **for** x
using assms that **unfolding** *terminates-almost-surely-def*
by (meson lossless-iff-set-pmf-None)

have ?L = spmf-of-ra f \ggg (λ x. map-pmf the (case-option (return-pmf None) (spmf-of-ra \circ g) x))

unfolding pmf-of-ra-def spmf-of-ra-bind bind-spmf-def map-bind-pmf comp-def **by** simp
also have ... = spmf-of-ra f \ggg
 (λ x. (case x of None \Rightarrow return-pmf (the None) | Some x \Rightarrow pmf-of-ra (g x)))
unfolding map-pmf-def comp-def pmf-of-ra-def map-pmf-def
by (intro arg-cong2[**where** f=bind-pmf] refl ext) (simp add:bind-return-pmf split:option.split)
also have ... = spmf-of-ra f \ggg (λ x. pmf-of-ra (g (the x)))
using 0 **by** (intro bind-pmf-cong refl) (auto split:option.split)
also have ... = ?R
unfolding pmf-of-ra-def map-pmf-def **by** (simp add:bind-assoc-pmf bind-return-pmf)
finally show ?thesis
by simp
 qed

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)
unfolding map-ra-def pmf-of-ra-bind[OF assms] pmf-of-ra-return map-pmf-def **by** simp

lemma *terminates-almost-surely-return*:

terminates-almost-surely (return-ra x)
unfolding *terminates-almost-surely-def* *spmf-of-ra-return* **by** *simp*

lemma *terminates-almost-surely-coin*:
terminates-almost-surely coin-ra
unfolding *terminates-almost-surely-def* *spmf-of-ra-coin* **by** *simp*

lemma *terminates-almost-surely-bind*:
assumes *terminates-almost-surely* f
assumes $\bigwedge x. x \in \text{set-pmf} (\text{pmf-of-ra } f) \implies \text{terminates-almost-surely} (g \ x)$
shows *terminates-almost-surely* (f \ggg g)

proof –

have 0: None \notin set-pmf (spmf-of-ra f)
using *assms(1)* *lossless-iff-set-pmf-None* **unfolding** *terminates-almost-surely-def*
by *blast*

hence Some x \in set-pmf (spmf-of-ra f) \iff x \in the ‘ set-pmf (spmf-of-ra f) **for** x
by (*metis image-iff option.collapse option.sel*)

hence set-spmf (spmf-of-ra f) = set-pmf (pmf-of-ra f)
unfolding *pmf-of-ra-def* *set-map-pmf* **by** (*simp add:set-eq-iff set-spmf-def*)

thus ?thesis

using *assms(1,2)* **unfolding** *terminates-almost-surely-def* *spmf-of-ra-bind* *lossless-bind-spmf*
by *auto*

qed

lemma *terminates-almost-surely-map*:
assumes *terminates-almost-surely* p
shows *terminates-almost-surely* (map-ra f p)
unfolding *map-ra-def*
by (*intro assms terminates-almost-surely-bind terminates-almost-surely-return*)

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* \Rightarrow ('a \times nat) random-*alg-int*

where *track-random-bits* f bs =

do {
 (r, bs $^\wedge$) \leftarrow 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*
using *assms unfolding track-random-bits-def by (cases f bs, auto)*

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

proof (*cases consumed-prefix f bs*)
case *None*
hence *f bs = None*
by (*subst wf-random-alt[OF assms(1)] simp*)
then show *?thesis*
unfolding *track-random-bits-def None by simp*

next
case (*Some a*)
hence *f bs = Some (eval-rm f a, cdrop (length a) bs)*
by (*subst wf-random-alt[OF assms(1)] simp*)
then show *?thesis*
unfolding *track-random-bits-def Some consumed-bits-def by simp*
qed

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

proof (*rule ext*)
fix *bs :: coin-stream*
have *wf-on-prefix coin-rai [chd bs] (chd bs)*
unfolding *wf-on-prefix-def coin-rai-def by simp*
moreover have *cprefix [chd bs] bs*
unfolding *cprefix-def by simp*
ultimately have $\{p \in \text{ptree-rm } (\text{coin-rai}). \text{cprefix } p \text{ bs}\} = \{[\text{chd } \text{bs}]\}$
by (*intro prefixes-singleton (auto simp:ptree-rm-def)*)
hence *consumed-prefix (coin-rai) bs = Some [chd bs]*
unfolding *consumed-prefix-def by simp*
hence *consumed-bits (coin-rai) bs = Some 1*
unfolding *consumed-bits-def by simp*
thus *?L bs = ?R bs*
unfolding *track-random-bits-def bind-rai-def*
by (*simp add:coin-rai-def return-rai-def*)
qed

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

proof (*rule ext*)
fix *bs :: coin-stream*
have *wf-on-prefix (return-rai x) [] x*
unfolding *wf-on-prefix-def return-rai-def by simp*
moreover have *cprefix [] bs*
unfolding *cprefix-def by simp*
ultimately have $\{p \in \text{ptree-rm } (\text{return-rai } x). \text{cprefix } p \text{ bs}\} = \{\}\}$
by (*intro prefixes-singleton (auto simp:ptree-rm-def)*)
hence *consumed-prefix (return-rai x) bs = Some []*
unfolding *consumed-prefix-def by simp*
hence *consumed-bits (return-rai x) bs = Some 0*

unfolding *consumed-bits-def* **by** *simp*
thus $?L\ bs = ?R\ bs$
unfolding *track-random-bits-def* **by** (*simp add:return-rai-def*)
qed

lemma *consumed-prefix-imp-wf*:
assumes *consumed-prefix* $m\ bs = \text{Some } p$
shows *wf-on-prefix* $m\ p\ (\text{eval-rm } m\ p)$
proof –
have $p \in \text{ptree-rm } m$
using *assms* **unfolding** *consumed-prefix-def the-elem-opt-Some-iff[OF prefixes-at-most-one]*
by *blast*
then obtain r **where** *wf-on-prefix* $m\ p\ r$
unfolding *ptree-rm-def* **by** *auto*
thus *?thesis*
unfolding *wf-on-prefix-def eval-rm-def* **by** *simp*
qed

lemma *consumed-prefix-imp-prefix*:
assumes *consumed-prefix* $m\ bs = \text{Some } p$
shows *cprefix* $p\ bs$
using *assms* **unfolding** *consumed-prefix-def the-elem-opt-Some-iff[OF prefixes-at-most-one]* **by**
blast

lemma *consumed-prefix-bindI*:
assumes *consumed-prefix* $m\ bs = \text{Some } p$
assumes *consumed-prefix* $(f\ (\text{eval-rm } m\ p))\ (\text{cdrop } (\text{length } p)\ bs) = \text{Some } q$
shows *consumed-prefix* $(m \gg f)\ bs = \text{Some } (p@q)$
proof –
define r **where** $r = \text{eval-rm } m\ p$

have 0 : *wf-on-prefix* $m\ p\ r$
unfolding *r-def* **using** *consumed-prefix-imp-wf[OF assms(1)]* **by** *simp*

have *consumed-prefix* $(f\ r)\ (\text{cdrop } (\text{length } p)\ bs) = \text{Some } q$
using *assms(2)* **unfolding** *r-def* **by** *simp*
hence 1 : *wf-on-prefix* $(f\ r)\ q\ (\text{eval-rm } (f\ r)\ q)$
using *consumed-prefix-imp-wf* **by** *auto*
have *wf-on-prefix* $(m \gg f)\ (p@q)\ (\text{eval-rm } (f\ r)\ q)$
by (*intro wf-on-prefix-bindI[where r=r] 0 1*)
hence $p@q \in \text{ptree-rm } (m \gg f)$
unfolding *ptree-rm-def* **by** *auto*
moreover have *cprefix* $p\ bs\ \text{cprefix } q\ (\text{cdrop } (\text{length } p)\ bs)$
using *consumed-prefix-imp-prefix assms* **by** *auto*
hence *cprefix* $(p@q)\ bs$
unfolding *cprefix-def* **by** (*metis length-append ctake-add*)
ultimately have $\{p \in \text{ptree-rm } (m \gg f).\ \text{cprefix } p\ bs\} = \{p@q\}$
by (*intro prefixes-singleton*) *auto*
thus *?thesis*
unfolding *consumed-prefix-def* **by** *simp*
qed

lemma *track-rb-bind*:
assumes *wf-random* m
assumes $\bigwedge x. x \in \text{range-rm } m \implies \text{wf-random } (f\ x)$
shows *track-random-bits* $(m \gg f) = \text{track-random-bits } m \gg$
 $(\lambda(r,n). \text{track-random-bits } (f\ r) \gg (\lambda(r',m). \text{return-rai } (r',n+m)))$ (**is** $?L = ?R$)
proof (*rule ext*)

```

fix bs :: coin-stream
have wf-bind: wf-random (m ≫≧ f)
  by (intro wf-bind assms)

consider (a) m bs = None | (b) m bs ≠ None ∧ (m ≫≧ f) bs = None | (c) (m ≫≧ f) bs ≠
None
  by blast
then show ?L bs = ?R bs
proof (cases)
  case a
  thus ?thesis
    unfolding track-random-bits-def bind-rai-def a by simp
next
  case b
  then obtain r bs' where 0:m bs = Some (r,bs') by auto
  have 1:(f r) bs' = None using b unfolding bind-rai-def 0 by simp
  then show ?thesis unfolding track-random-bits-def bind-rai-def 0 by simp
next
  case c
  have (m ≫≧ f) bs = None if m bs = None
    using that unfolding bind-rai-def by simp
  hence m bs ≠ None using c by blast
  then obtain p where 0:
    m bs = Some (eval-rm m p, cdrop (length p) bs) consumed-prefix m bs = Some p
    using wf-random-alt[OF assms(1)] by auto
  define bs' where bs' = cdrop (length p) bs
  define r where r = eval-rm m p
  have 1: m bs = Some (r, bs') unfolding 0 r-def bs'-def by simp
  hence r ∈ range-rm m using 1 in-range-rmI by metis
  hence wf: wf-random (f r) by (intro assms(2))
  have f r bs' ≠ None using c 1 unfolding bind-rai-def by force
  then obtain q where 2:
    f r bs' = Some (eval-rm (f r) q, cdrop (length q) bs') consumed-prefix (f r) bs' = Some q
    using wf-random-alt[OF wf] by auto

  hence 3:consumed-prefix (m ≫≧ f) bs = Some (p@q)
    unfolding r-def bs'-def by (intro consumed-prefix-bindI 0) auto

  have track-random-bits m bs = Some ((r, length p), bs')
    unfolding track-random-bits-alt[OF assms(1)] bind-rai-def 0 bs'-def r-def by simp
  moreover have track-random-bits (f r) bs' =
    Some ((eval-rm (f r) q, length q), cdrop (length q) bs')
    unfolding track-random-bits-alt[OF wf] 2 by simp
  moreover have wf-on-prefix m p r
    unfolding r-def by (intro consumed-prefix-imp-wf[OF 0(2)])
  hence eval-rm (f r) q = eval-rm (m ≫≧ f) (p@q)
    unfolding eval-rm-def bind-rai-def wf-on-prefix-def by simp
  ultimately have
    ?R bs = Some ((eval-rm (m ≫≧ f) (p@q), length p+length q), cdrop (length p+length q) bs)
    unfolding bind-rai-def return-rai-def bs'-def by simp
  also have ... = ?L bs
    unfolding track-random-bits-alt[OF wf-bind] 3 by simp
  finally show ?thesis by simp
qed
qed

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 –
  have track-random-bits f bs = track-random-bits g bs
    if track-random-bits f bs ≠ None for bs
  proof –
    have f bs ≠ None using that track-random-bits-Some-iff by simp
    then obtain r bs' where f bs = Some (r, bs') by auto
    then obtain p where 0:wf-on-prefix f p r and 2:cprefix p bs
      using assms(1) unfolding wf-random-def by (auto split:option.split-asm)

  have 1: wf-on-prefix g p r
    using wf-lub-helper[OF assms(3)] 0 by blast

  have track-random-bits h bs = Some ((r, length p), cdrop (length p) bs)
    if wf-on-prefix h p r wf-random h for h
  proof –
    have p ∈ ptree-rm h
      using that unfolding ptree-rm-def by auto
    hence {p ∈ ptree-rm h. cprefix p bs} = {p}
      using 2 by (intro prefixes-singleton) auto
    hence consumed-prefix h bs = Some p
      unfolding consumed-prefix-def by simp
    moreover have eval-rm h p = r
      using that(1) unfolding wf-on-prefix-def eval-rm-def by simp
    ultimately show ?thesis
      unfolding track-random-bits-alt[OF that(2)] by simp
  qed

  thus ?thesis
    using 0 1 assms(1,2) by simp
  qed
  thus ?thesis
    unfolding ord-rai-def fun-ord-def flat-ord-def by blast
  qed

lemma wf-track-random-bits:
  assumes wf-random f
  shows wf-random (track-random-bits f)
proof (rule wf-randomI)
  fix bs
  assume track-random-bits f bs ≠ None
  hence f bs ≠ None using track-random-bits-Some-iff by blast
  then obtain r bs' where f bs = Some (r, bs')
    by auto
  then obtain p where 0:wf-on-prefix f p r cprefix p bs
    using assms unfolding wf-random-def by (auto split:option.split-asm)
  hence {q ∈ ptree-rm f. cprefix q (cshift p cs)} = {p} for cs
    by (intro prefixes-singleton) (auto simp:cprefix-def ptree-rm-def)
  hence consumed-prefix f (cshift p cs) = Some p for cs
    unfolding consumed-prefix-def by simp
  hence wf-on-prefix (track-random-bits f) p (r, length p)
    using 0 unfolding track-random-bits-def wf-on-prefix-def consumed-bits-def by simp
  thus  $\exists p r. cprefix p bs \wedge wf-on-prefix (track-random-bits f) p r$ 
    using 0 by auto
  qed

lemma track-random-bits-lub-rai:

```

assumes *Complete-Partial-Order.chain ord-rai A*
assumes $\bigwedge r. r \in A \implies wf\text{-random } r$
shows $track\text{-random-bits } (lub\text{-rai } A) = lub\text{-rai } (track\text{-random-bits } 'A)$ **(is ?L = ?R)**
proof –
have $0: Complete\text{-Partial-Order.chain ord-rai } (track\text{-random-bits } 'A)$
by $(intro\ chain\ imageI [OF\ assms(1)]\ track\text{-random-bits-mono}\ assms(2))$

have $?L\ bs = ?R\ bs$ **if** $?L\ bs \neq None$ **for** bs
proof –
have $1: lub\text{-rai } A\ bs \neq None$ **using** *that track-random-bits-Some-iff* **by** *simp*
have $lub\text{-rai } A\ bs = None$ **if** $\bigwedge f. f \in A \implies f\ bs = None$
using *that unfolding lub-rai-def fun-lub-def flat-lub-def* **by** *auto*
then obtain f **where** $f\text{-in-}A: f \in A$ **and** $f\ bs \neq None$
using 1 **by** *blast*
hence *consumed-prefix* $f\ bs \neq None$
using *consumed-prefix-none-iff* $[OF\ assms(2)] [OF\ f\text{-in-}A]$ **by** *simp*
hence $2: track\text{-random-bits } f\ bs \neq None$
unfolding *track-random-bits-alt* $[OF\ assms(2)] [OF\ f\text{-in-}A]$ **by** *simp*
have $ord\text{-rai } (track\text{-random-bits } f)\ (track\text{-random-bits } (lub\text{-rai } A))$
by $(intro\ track\text{-random-bits-mono}\ wf\text{-lub} [OF\ assms(1)]\ assms(2)\ random\text{-alg-int-pd.lub-upper} [OF\ assms(1)]\ f\text{-in-}A)$
hence $track\text{-random-bits } (lub\text{-rai } A)\ bs = track\text{-random-bits } f\ bs$
using 2 **unfolding** *ord-rai-def fun-ord-def flat-ord-def* **by** *metis*

moreover have $ord\text{-rai } (track\text{-random-bits } f)\ (lub\text{-rai } (track\text{-random-bits } 'A))$
using $f\text{-in-}A$ **by** $(intro\ random\text{-alg-int-pd.lub-upper} [OF\ 0])\ auto$
hence $lub\text{-rai } (track\text{-random-bits } 'A)\ bs = track\text{-random-bits } f\ bs$
using 2 **unfolding** *ord-rai-def fun-ord-def flat-ord-def* **by** *metis*
ultimately show *?thesis* **by** *simp*
qed
hence *flat-ord* $None\ (?L\ bs)\ (?R\ bs)$ **for** bs
unfolding *flat-ord-def* **by** *blast*
hence $ord\text{-rai } ?L\ ?R$
unfolding *ord-rai-def fun-ord-def* **by** *simp*
moreover have $ord\text{-rai } (track\text{-random-bits } f)\ (track\text{-random-bits } (lub\text{-rai } A))$ **if** $f \in A$ **for** f
using *that assms(2) wf-lub* $[OF\ assms(1,2)]$
by $(intro\ track\text{-random-bits-mono}\ random\text{-alg-int-pd.lub-upper} [OF\ assms(1)])$
hence $ord\text{-rai } ?R\ ?L$
by $(intro\ random\text{-alg-int-pd.lub-least } 0)\ auto$
ultimately show *?thesis*
using *random-alg-int-pd.leq-antisym* **by** *auto*
qed

lemma *untrack-random-bits*:
assumes *wf-random f*
shows $track\text{-random-bits } f \gg (\lambda x. return\text{-rai } (fst\ x)) = f$ **(is ?L = ?R)**
proof –
have $?L\ bs = ?R\ bs$ **for** bs
unfolding *track-random-bits-alt* $[OF\ assms]$ *bind-rai-def return-rai-def*
by $(subst\ wf\text{-random-alt} [OF\ assms])\ (cases\ consumed\text{-prefix } f\ bs,\ auto)$
thus *?thesis*
by *auto*
qed

lift-definition *track-coin-use* $:: 'a\ random\text{-alg} \Rightarrow ('a \times nat)\ random\text{-alg}$
is *track-random-bits*
by $(rule\ wf\text{-track-random-bits})$

definition *bind-tra* ::
 ('a × nat) random-alg ⇒ ('a ⇒ ('b × nat) random-alg) ⇒ ('b × nat) random-alg
where *bind-tra* m f = do {
 (r,k) ← m;
 (s,l) ← (f r);
 return-ra (s, k+l)
 }

definition *coin-tra* :: (bool × nat) random-alg
where *coin-tra* = do {
 b ← coin-ra;
 return-ra (b,1)
 }

definition *return-tra* :: 'a ⇒ ('a × 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
unfolding *bind-tra-def* *return-tra-def*
by (*simp add:bind-return-ra return-bind-ra*)

lemma *bind-tra-assoc*:
bind-tra (bind-tra f g) h = bind-tra f (λx. bind-tra (g x) h)
unfolding *bind-tra-def*
by (*simp add:bind-return-ra return-bind-ra bind-ra-assoc case-prod-beta' algebra-simps*)

lemma *bind-return-tra*:
bind-tra m return-tra = m
unfolding *bind-tra-def* *return-tra-def*
by (*simp add:bind-return-ra return-bind-ra*)

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 –

have *Rep-random-alg* ?L = *Rep-random-alg* ?R
unfolding *track-coin-use.rep-eq* *bind-ra.rep-eq* *bind-tra-def*
by (*subst track-rb-bind*) (*simp-all add:wf-rep-rand-alg comp-def case-prod-beta'*
track-coin-use.rep-eq *bind-ra.rep-eq* *return-ra.rep-eq*)
thus *?thesis*
using *Rep-random-alg-inject* **by** *auto*

qed

lemma *track-coin-use-coin*: track-coin-use coin-ra = coin-tra (is ?L = ?R)
unfolding *coin-tra-def* **using** *track-rb-coin[transferred]* **by** *metis*

lemma *track-coin-use-return*: track-coin-use (return-ra x) = return-tra x (is ?L = ?R)
unfolding *return-tra-def* **using** *track-rb-return[transferred]* **by** *metis*

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 –

have 0: *Complete-Partial-Order.chain ord-rai (Rep-random-alg ‘ A)*
using *assms unfolding ord-ra.rep-eq Complete-Partial-Order.chain-def* **by** *auto*

have 2: *(Rep-random-alg ‘ track-coin-use ‘ A) = track-random-bits ‘ Rep-random-alg ‘ A*
using *track-coin-use.rep-eq unfolding image-image* **by** *auto*

have 1: *Complete-Partial-Order.chain ord-rai (Rep-random-alg ‘ track-coin-use ‘ A)*
using *wf-rep-rand-alg unfolding 2* **by** *(intro chain-imageI[OF 0] track-random-bits-mono)*
auto

have *Rep-random-alg ?L = track-random-bits (lub-rai (Rep-random-alg ‘ A))*

using 0 **unfolding** *track-coin-use.rep-eq lub-ra.rep-eq* **by** *simp*

also have ... = *lub-rai (track-random-bits ‘ Rep-random-alg ‘ A)*

using *wf-rep-rand-alg* **by** *(intro track-random-bits-lub-rai 0) auto*

also have ... = *Rep-random-alg ?R*

using 1 **unfolding** *lub-ra.rep-eq 2* **by** *simp*

finally have *Rep-random-alg ?L = Rep-random-alg ?R*

by *simp*

thus *?thesis*

using *Rep-random-alg-inject* **by** *auto*

qed

lemma *track-coin-use-mono*:

assumes *ord-ra f g*

shows *ord-ra (track-coin-use f) (track-coin-use g)*

using *assms by transfer (rule track-random-bits-mono)*

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)*

using *assms unfolding bind-tra-def ord-ra.rep-eq bind-ra.rep-eq*

by *(auto intro!:bind-rai-mono random-alg-int-pd.leq-refl*

simp:comp-def bind-ra.rep-eq case-prod-beta' return-ra.rep-eq)

lemma *bind-tra-mono [partial-function-mono]*:

assumes *mono-ra B and $\bigwedge y. mono-ra (C y)$*

shows *mono-ra ($\lambda f. bind-tra (B f) (\lambda y. C y f)$)*

using *assms by (intro monotoneI bind-mono-tra-aux) (auto simp:monotone-def)*

lemma *track-coin-use-empty*:

track-coin-use (lub-ra {}) = (lub-ra {}) **(is ?L = ?R)**

proof –

have *?L = lub-ra (track-coin-use ‘ {})*

by *(intro track-coin-use-lub) (simp add:Complete-Partial-Order.chain-def)*

also have ... = *?R* **by** *simp*

finally show *?thesis* **by** *simp*

qed

lemma *untrack-coin-use*:

map-ra fst (track-coin-use f) = f **(is ?L = ?R)**

proof –

have *Rep-random-alg ?L = Rep-random-alg ?R*

unfolding *map-ra-def bind-ra.rep-eq track-coin-use.rep-eq comp-def return-ra.rep-eq*

by *(auto intro!:untrack-random-bits simp:wf-rep-rand-alg)*

thus *?thesis*

using *Rep-random-alg-inject* **by** *auto*

qed

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 (rule *ccpo.admissibleI*)

fix Y

assume *chain*: Complete-Partial-Order.chain ?ord Y

and Y: Y ≠ {}

and R: ∀ (p, q) ∈ Y. *rel-track-coin-use* p q

from R have R: ∧p q. (p, q) ∈ Y ⇒ *rel-track-coin-use* p q **by** auto

have *chain1*: Complete-Partial-Order.chain (ord-ra) (fst ' Y)

and *chain2*: Complete-Partial-Order.chain (ord-ra) (snd ' Y)

using *chain* **by**(rule *chain-imageI*; *clarsimp*)+

from Y have Y1: fst ' Y ≠ {} and Y2: snd ' Y ≠ {} **by** auto

have lub-ra (fst ' Y) = lub-ra (*track-coin-use* ' snd ' Y)

unfolding *image-image* using R

by (intro *arg-cong*[of - - lub-ra] *image-cong*) (auto *simp*: *rel-track-coin-use-def*)

also have ... = *track-coin-use* (lub-ra (snd ' Y))

by (intro *track-coin-use-lub*[*symmetric*] *chain2*)

finally have *rel-track-coin-use* (lub-ra (fst ' Y)) (lub-ra (snd ' Y))

unfolding *rel-track-coin-use-def* .

then show ?P (?lub Y)

by (*simp add*: *prod-lub-def*)

qed

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*]

⇒ *ccpo.admissible* lub ord (λx. *rel-track-coin-use* (f x) (g x))

by (rule *admissible-subst*[OF *admissible-rel-track-coin-use*, **where** f=λx. (f x, g x), *simplified*])
(rule *mcont-Pair*)

lemma *mcont-track-coin-use*:

mcont lub-ra ord-ra lub-ra ord-ra track-coin-use

unfolding *mcont-def* *monotone-def* *cont-def*

by (auto *simp*: *track-coin-use-mono* *track-coin-use-lub*)

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: ∧x. *mono-ra* (λf. F f x)

and g: ∧x. *mono-ra* (λf. 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)

using f g

proof(rule *parallel-fixp-induct-1-1*[OF

random-alg-pfd *random-alg-pfd* - - *reflexive* *reflexive*,

where P=(A ==> *rel-track-coin-use*)]

show *ccpo.admissible* (prod-lub (fun-lub lub-ra) (fun-lub lub-ra))

(rel-prod (fun-ord ord-ra) (fun-ord ord-ra))

(λx. (A ==> *rel-track-coin-use*) (fst x) (snd x))

```

unfolding rel-fun-def
  by(rule admissible-all admissible-imp cont-intro) +
have 0:track-coin-use (lub-ra {}) = lub-ra {}
  using track-coin-use-lub [where A={} ]
  by (simp add:Complete-Partial-Order.chain-def)
show (A ==> rel-track-coin-use) ( $\lambda\cdot$ . lub-ra {}) ( $\lambda\cdot$ . lub-ra {})
  by (auto simp: rel-fun-def rel-track-coin-use-def 0)
show (A ==> rel-track-coin-use) (F f) (G g) if (A ==> rel-track-coin-use) f g for f g
  using that by(rule rel-funD[OF param])
qed

```

```

lemma return-ra-transfer[transfer-rule]: ((=) ==> rel-track-coin-use) return-tra return-ra
  unfolding rel-fun-def rel-track-coin-use-def track-coin-use-return by simp

```

```

lemma bind-ra-transfer[transfer-rule]:
  (rel-track-coin-use ==> ((=) ==> rel-track-coin-use) ==> rel-track-coin-use) bind-tra
bind-ra
  unfolding rel-fun-def rel-track-coin-use-def track-coin-use-bind by simp presburger

```

```

lemma coin-ra-transfer[transfer-rule]:
  rel-track-coin-use coin-tra coin-ra
  unfolding rel-fun-def rel-track-coin-use-def track-coin-use-coin by simp

```

end

end

7 Tracking SPMFs

This section introduces tracking SPMFs — this is a resource monad on top of SPMFs, we also introduce the Scott-continuous 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 (λ(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
  unfolding bind-tspmf-def return-tspmf-def map-spmf-conv-bind-spmf
  by (simp add:apsnd-def map-prod-def)

```

lemma *bind-tspmf-assoc*:
 $bind\text{-}tspmf\ (bind\text{-}tspmf\ f\ g)\ h = bind\text{-}tspmf\ f\ (\lambda x.\ bind\text{-}tspmf\ (g\ x)\ h)$
unfolding *bind-tspmf-def*
by (*simp add: case-prod-beta' algebra-simps map-spmf-conv-bind-spmf apsnd-def map-prod-def*)

lemma *bind-return-tspmf*:
 $bind\text{-}tspmf\ m\ return\text{-}tspmf = m$
unfolding *bind-tspmf-def return-tspmf-def map-spmf-conv-bind-spmf apsnd-def*
by (*simp add: case-prod-beta'*)

lemma *bind-mono-tspmf-aux*:
assumes *ord-spmf (=) f1 f2 \wedge y. ord-spmf (=) (g1 y) (g2 y)*
shows *ord-spmf (=) (bind-tspmf f1 g1) (bind-tspmf f2 g2)*
using *assms unfolding bind-tspmf-def map-spmf-conv-bind-spmf*
by (*auto intro!: bind-spmf-mono' simp add: case-prod-beta'*)

lemma *bind-mono-tspmf [partial-function-mono]*:
assumes *mono-spmf B and \wedge y. mono-spmf (C y)*
shows *mono-spmf (λ f. bind-tspmf (B f) (λ y. C y f))*
using *assms by (intro monotoneI bind-mono-tspmf-aux) (auto simp: monotone-def)*

definition *ord-tspmf* :: '*a* *tspmf* \Rightarrow '*a* *tspmf* \Rightarrow *bool* **where**
 $ord\text{-}tspmf = ord\text{-}spmf\ (\lambda x\ y.\ fst\ x = fst\ y \wedge snd\ x \geq snd\ y)$

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

definition *coin-usage-of-tspmf* :: '*a* *tspmf* \Rightarrow *enat* *pmf*
where *coin-usage-of-tspmf* = *map-pmf* ($\lambda x.\ case\ x\ of\ None \Rightarrow \infty \mid Some\ y \Rightarrow enat\ (snd\ y)$)

definition *expected-coin-usage-of-tspmf* :: '*a* *tspmf* \Rightarrow *ennreal*
where
 $expected\text{-}coin\text{-}usage\text{-}of\text{-}tspmf\ p = (\int^+ x.\ x\ \partial(\text{map-pmf}\ ennreal\text{-}of\text{-}enat\ (\text{coin-usage-of-tspmf}\ p)))$

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

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

lemma *coin-usage-of-tspmf-alt-def*:
 $coin\text{-}usage\text{-}of\text{-}tspmf\ p = \text{map-pmf}\ (\lambda x.\ case\ x\ of\ None \Rightarrow \infty \mid Some\ y \Rightarrow enat\ y)\ (\text{map-spmf}\ snd\ p)$
unfolding *coin-usage-of-tspmf-def map-pmf-comp map-option-case*
by (*metis enat-def infinity-enat-def option.case-eq-if option.sel*)

lemma *coin-usage-of-tspmf-bind-return*:
 $coin\text{-}usage\text{-}of\text{-}tspmf\ (bind\text{-}tspmf\ f\ (\lambda x.\ return\text{-}tspmf\ (g\ x))) = (coin\text{-}usage\text{-}of\text{-}tspmf\ f)$
unfolding *bind-tspmf-def return-tspmf-def coin-usage-of-tspmf-alt-def map-spmf-bind-spmf*
by (*simp add: comp-def case-prod-beta map-spmf-conv-bind-spmf*)

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

define p' **where** $p' = \text{map-spmf } \text{snd } p$
define q' **where** $q' = \text{map-spmf } \text{snd } q$
have $0:\text{ord-spmf } (\geq) p' q'$
using $\text{assms}(1)$ ord-spmf-mono **unfolding** $p'\text{-def } q'\text{-def } \text{ord-tspmfm-def } \text{ord-spmfm-map-spmf12}$
by fastforce

have $\text{cp}:\text{coin-usage-of-tspmfm } p = \text{map-pmf } (\text{case-option } \infty \text{ enat}) p'$
unfolding $\text{coin-usage-of-tspmfm-alt-def } p'\text{-def}$ **by** simp
have $\text{cq}:\text{coin-usage-of-tspmfm } q = \text{map-pmf } (\text{case-option } \infty \text{ enat}) q'$
unfolding $\text{coin-usage-of-tspmfm-alt-def } q'\text{-def}$ **by** simp

have $0:\text{rel-pmf } (\geq) (\text{coin-usage-of-tspmfm } p) (\text{coin-usage-of-tspmfm } q)$
unfolding $\text{cp } \text{cq } \text{map-pmf-def}$ **by** $(\text{intro } \text{rel-pmf-bindI}[OF\ 0]) (\text{auto } \text{split}:\text{option.split})$
show $?thesis$
unfolding atMost-def **by** $(\text{intro } \text{measure-Ici}[OF\ 0] \text{transp-on-ge}) (\text{simp } \text{add}:\text{reflp-def})$
qed

lemma $\text{coin-usage-of-tspmfm-mono-rev}$:

assumes $\text{ord-tspmfm } p\ q$
shows $\text{measure } (\text{coin-usage-of-tspmfm } q) \{x. x > k\} \leq \text{measure } (\text{coin-usage-of-tspmfm } p) \{x. x > k\}$
(is $?L \leq ?R$)

proof –

have $0:\text{UNIV} - \{x. x > k\} = \{..k\}$
by $(\text{auto } \text{simp } \text{add}:\text{set-diff-eq } \text{set-eq-iff})$
have $1 - ?R \leq 1 - ?L$
using $\text{coin-usage-of-tspmfm-mono}[OF\ \text{assms}]$
by $(\text{subst } (1\ 2) \text{measure-pmf.prob-compl}[\text{symmetric}]) (\text{auto } \text{simp}:0)$
thus $?thesis$
by simp

qed

lemma $\text{expected-coin-usage-of-tspmfm}$:

$\text{expected-coin-usage-of-tspmfm } p = (\sum k. \text{ennreal } (\text{measure } (\text{coin-usage-of-tspmfm } p) \{x. x > \text{enat } k\}))$ **(is** $?L = ?R$)

proof –

have $?L = \text{integral}^N (\text{measure-pmf } (\text{coin-usage-of-tspmfm } p)) \text{ennreal-of-enat}$
unfolding $\text{expected-coin-usage-of-tspmfm-def}$ **by** simp
also have $\dots = (\sum k. \text{emeasure } (\text{measure-pmf } (\text{coin-usage-of-tspmfm } p)) \{x. \text{enat } k < x\})$
by $(\text{subst } \text{nn-integral-enat-function}) \text{auto}$
also have $\dots = ?R$
by $(\text{subst } \text{measure-pmf.emeasure-eq-measure}) \text{simp}$
finally show $?thesis$
by simp

qed

lemma ord-tspmfm-min : $\text{ord-tspmfm } (\text{return-pmf } \text{None})\ p$

unfolding ord-tspmfm-def **by** $(\text{simp } \text{add}:\text{ord-spmfm-reflI})$

lemma ord-tspmfm-refl : $\text{ord-tspmfm } p\ p$

unfolding ord-tspmfm-def **by** $(\text{simp } \text{add}:\text{ord-spmfm-reflI})$

lemma $\text{ord-tspmfm-trans}[\text{trans}]$:

assumes $\text{ord-tspmfm } p\ q\ \text{ord-tspmfm } q\ r$

shows $\text{ord-tspmfm } p\ r$

proof –

have $0:\text{transp } (\text{ord-tspmfm})$
unfolding ord-tspmfm-def

by (intro transp-rel-pmf transp-ord-option) (auto simp:transp-def)
 thus ?thesis
 using assms transpD[OF 0] by auto
 qed

lemma ord-tspmf-map-spmf:
 assumes $\bigwedge x. x \leq f x$
 shows ord-tspmf (map-spmf (apsnd f) p) p
 using assms **unfolding** ord-tspmf-def ord-spmf-map-spmf1
 by (intro ord-spmf-reflI) auto

lemma ord-tspmf-bind-pmf:
 assumes $\bigwedge x. \text{ord-tspmf } (f x) (g x)$
 shows ord-tspmf (bind-pmf p f) (bind-pmf p g)
 using assms **unfolding** ord-tspmf-def
 by (intro rel-pmf-bindI[where R=(=)]) (auto simp add: pmf.rel-refl)

lemma ord-tspmf-bind-tspmf:
 assumes $\bigwedge x. \text{ord-tspmf } (f x) (g x)$
 shows ord-tspmf (bind-tspmf p f) (bind-tspmf p g)
 using assms **unfolding** bind-tspmf-def ord-tspmf-def
 by (intro ord-spmf-bind-reflI) (simp add:case-prod-beta ord-spmf-map-spmf12)

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
unfolding use-coins-def spmf.map-comp
 by (simp add:comp-def apsnd-def map-prod-def case-prod-beta' algebra-simps)

lemma coin-tspmf-split:
 fixes f :: bool \Rightarrow 'b tspmf
 shows (coin-tspmf \ggg f) = use-coins 1 (coin-spmf \ggg f)
unfolding coin-tspmf-def use-coins-def map-spmf-conv-bind-spmf pair-spmf-alt-def bind-tspmf-def
 by (simp)

lemma ord-tspmf-use-coins:
 ord-tspmf (use-coins k p) p
unfolding use-coins-def by (intro ord-tspmf-map-spmf) auto

lemma ord-tspmf-use-coins-2:
 assumes ord-tspmf p q
 shows ord-tspmf (use-coins k p) (use-coins k q)
 using assms **unfolding** use-coins-def ord-tspmf-def ord-spmf-map-spmf12 by auto

lemma result-mono:
 assumes ord-tspmf p q
 shows ord-spmf (=) (result p) (result q)
 using assms ord-spmf-mono **unfolding** result-def ord-tspmf-def ord-spmf-map-spmf12 by force

lemma result-bind:
 result (bind-tspmf f g) = result f \ggg ($\lambda x. \text{result } (g x)$)
unfolding bind-tspmf-def result-def map-spmf-conv-bind-spmf by (simp add:case-prod-beta')

lemma result-return:
 result (return-tspmf x) = return-spmf x
unfolding return-tspmf-def result-def map-spmf-conv-bind-spmf by (simp add:case-prod-beta')

lemma *result-coin*:

result (coin-tspmf) = coin-spmf

unfolding *coin-tspmf-def result-def pair-spmf-alt-def map-spmf-conv-bind-spmf* **by** (*simp add: case-prod-beta'*)

definition *tspmf-of-ra* :: 'a random-alg \Rightarrow 'a tspmf **where**

tspmf-of-ra = spmf-of-ra \circ track-coin-use

lemma *tspmf-of-ra-coin*: *tspmf-of-ra coin-ra = coin-tspmf*

unfolding *tspmf-of-ra-def comp-def track-coin-use-coin coin-tra-def coin-tspmf-def*

spmf-of-ra-bind spmf-of-ra-coin spmf-of-ra-return pair-spmf-alt-def

by *simp*

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

unfolding *tspmf-of-ra-def comp-def track-coin-use-return return-tra-def return-tspmf-def*

spmf-of-ra-return **by** *simp*

lemma *tspmf-of-ra-bind*:

tspmf-of-ra (bind-ra m f) = bind-tspmf (tspmf-of-ra m) ($\lambda x. \text{tspmf-of-ra } (f x)$)

unfolding *tspmf-of-ra-def comp-def track-coin-use-bind bind-tra-def bind-tspmf-def*

map-spmf-conv-bind-spmf

by (*simp add: case-prod-beta' spmf-of-ra-bind spmf-of-ra-return apsnd-def map-prod-def*)

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)*

unfolding *tspmf-of-ra-def comp-def*

by (*intro spmf-of-ra-mono track-coin-use-mono assms*)

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 –

have *0: Complete-Partial-Order.chain ord-ra (track-coin-use ' A)*

by (*intro chain-imageI[OF assms] track-coin-use-mono*)

have *?L = spmf-of-ra (lub-ra (track-coin-use ' A))*

unfolding *tspmf-of-ra-def comp-def*

by (*intro arg-cong[where f=spmf-of-ra] track-coin-use-lub assms*)

also have *... = lub-spmf (spmf-of-ra ' track-coin-use ' A)*

by (*intro spmf-of-ra-lub-ra 0*)

also have *... = ?R*

unfolding *image-image tspmf-of-ra-def* **by** *simp*

finally show *?thesis* **by** *simp*

qed

definition *rel-tspmf-of-ra* :: 'a tspmf \Rightarrow 'a random-alg \Rightarrow bool **where**

rel-tspmf-of-ra q p \longleftrightarrow 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 (*rule ccpo.admissibleI*)

fix *Y*

assume *chain: Complete-Partial-Order.chain ?ord Y*

and *Y: Y \neq {}*

and $R: \forall (p, q) \in Y. \text{rel-tspmf-of-ra } p \ q$
from R **have** $R: \bigwedge p \ q. (p, q) \in Y \implies \text{rel-tspmf-of-ra } p \ q$ **by** *auto*
have *chain1*: *Complete-Partial-Order.chain* (*ord-spmf* (=)) (*fst* ' Y)
and *chain2*: *Complete-Partial-Order.chain* *ord-ra* (*snd* ' Y)
using *chain* **by**(*rule chain-imageI*; *clarsimp*)
from Y **have** $Y1: \text{fst ' } Y \neq \{\}$ **and** $Y2: \text{snd ' } Y \neq \{\}$ **by** *auto*

have $\text{lub-spmf } (\text{fst ' } Y) = \text{lub-spmf } (\text{tspmf-of-ra ' } \text{snd ' } Y)$
unfolding *image-image* **using** R
by (*intro arg-cong*[*of - - lub-spmf*] *image-cong*) (*auto simp: rel-tspmf-of-ra-def*)
also have $\dots = \text{tspmf-of-ra } (\text{lub-ra } (\text{snd ' } Y))$
by (*intro tspmf-of-ra-lub*[*symmetric*] *chain2*)
finally have $\text{rel-tspmf-of-ra } (\text{lub-spmf } (\text{fst ' } Y)) (\text{lub-ra } (\text{snd ' } Y))$
unfolding *rel-tspmf-of-ra-def* .
then show $?P$ ($?lub \ Y$)
by (*simp add: prod-lub-def*)
qed

lemma *admissible-rel-tspmf-of-ra-cont* [*cont-intro*]:
fixes *ord*
shows $\llbracket \text{mcont } \text{lub } \text{ord } \text{lub-spmf } (\text{ord-spmf } (=)) \ f; \text{mcont } \text{lub } \text{ord } \text{lub-ra } \text{ord-ra } \ g \rrbracket$
 $\implies \text{ccpo.admissible } \text{lub } \text{ord } (\lambda x. \text{rel-tspmf-of-ra } (f \ x) (g \ x))$
by (*rule admissible-subst*[*OF admissible-rel-tspmf-of-ra*, **where** $f = \lambda x. (f \ x, g \ x)$, *simplified*])
(*rule mcont-Pair*)

lemma *mcont-tspmf-of-ra*:
 $\text{mcont } \text{lub-ra } \text{ord-ra } \text{lub-spmf } (\text{ord-spmf } (=)) \ \text{tspmf-of-ra}$
unfolding *mcont-def* *monotone-def* *cont-def*
by (*auto simp: tspmf-of-ra-mono tspmf-of-ra-lub*)

lemmas $\text{mcont2mcont-tspmf-of-ra} = \text{mcont-tspmf-of-ra}$ [*THEN* *spmf.mcont2mcont*]

context includes *lifting-syntax*
begin

lemma *fixp-rel-tspmf-of-ra-parametric*[*transfer-rule*]:
assumes $f: \bigwedge x. \text{mono-spmf } (\lambda f. F \ f \ x)$
and $g: \bigwedge x. \text{mono-ra } (\lambda f. G \ f \ x)$
and *param*: $((A \implies \text{rel-tspmf-of-ra}) \implies A \implies \text{rel-tspmf-of-ra}) \ F \ G$
shows $(A \implies \text{rel-tspmf-of-ra}) (\text{spmf.fixp-fun } F) (\text{random-alg-pf.fixp-fun } G)$
using $f \ g$

proof(*rule parallel-fixp-induct-1-1*[*OF*
partial-function-definitions-spmf *random-alg-pfd - - reflexive reflexive*,
where $P = (A \implies \text{rel-tspmf-of-ra})$])
show $\text{ccpo.admissible } (\text{prod-lub } (\text{fun-lub } \text{lub-spmf}) (\text{fun-lub } \text{lub-ra}))$
 $(\text{rel-prod } (\text{fun-ord } (\text{ord-spmf } (=))) (\text{fun-ord } \text{ord-ra}))$
 $(\lambda x. (A \implies \text{rel-tspmf-of-ra}) (\text{fst } x) (\text{snd } x))$
unfolding *rel-fun-def*
by(*rule admissible-all admissible-imp cont-intro*)
have $0: \text{tspmf-of-ra } (\text{lub-ra } \{\}) = \text{return-pmf } \text{None}$
using *tspmf-of-ra-lub*[**where** $A = \{\}$]
by (*simp add: Complete-Partial-Order.chain-def*)
show $(A \implies \text{rel-tspmf-of-ra}) (\lambda-. \text{lub-spmf } \{\}) (\lambda-. \text{lub-ra } \{\})$
by (*auto simp: rel-fun-def rel-tspmf-of-ra-def 0*)
show $(A \implies \text{rel-tspmf-of-ra}) (F \ f) (G \ g)$ **if** $(A \implies \text{rel-tspmf-of-ra}) \ f \ g$ **for** $f \ g$
using *that* **by**(*rule rel-funD*[*OF param*])
qed

lemma *return-ra-transfer*[*transfer-rule*]: $((=) \implies \text{rel-tspmf-of-ra})$ *return-tspmf return-ra*
unfolding *rel-fun-def rel-tspmf-of-ra-def tspmf-of-ra-return* **by** *simp*

lemma *bind-ra-transfer*[*transfer-rule*]:
 $(\text{rel-tspmf-of-ra} \implies ((=) \implies \text{rel-tspmf-of-ra}) \implies \text{rel-tspmf-of-ra})$ *bind-tspmf bind-ra*
unfolding *rel-fun-def rel-tspmf-of-ra-def tspmf-of-ra-bind* **by** *simp presburger*

lemma *coin-ra-transfer*[*transfer-rule*]:
rel-tspmf-of-ra coin-tspmf coin-ra
unfolding *rel-fun-def rel-tspmf-of-ra-def tspmf-of-ra-coin* **by** *simp*

end

lemma *spmf-of-tspmf*:
result (tspmf-of-ra f) = spmf-of-ra f
unfolding *tspmf-of-ra-def result-def*
by (*simp add: untrack-coin-use spmf-of-ra-map[symmetric]*)

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

proof –

let *?p = Rep-random-alg p*

have *measure-pmf (map-spmf snd (tspmf-of-ra p)) =*
distr (distr-rai (track-random-bits ?p)) \mathcal{D} (map-option snd)
unfolding *tspmf-of-ra-def map-pmf-rep-eq spmf-of-ra.rep-eq comp-def track-coin-use.rep-eq*
by *simp*

also have $\dots = \text{distr } \mathcal{B} \mathcal{D} (\text{map-option snd} \circ (\text{map-option fst} \circ \text{track-random-bits } ?p))$
unfolding *distr-rai-def*
by (*intro distr-distr distr-rai-measurable wf-track-random-bits wf-rep-rand-alg*) *simp*

also have $\dots = \text{distr } \mathcal{B} \mathcal{D} (\lambda x. ?p x \gg (\lambda xa. \text{consumed-bits } ?p x))$
unfolding *track-random-bits-def* **by** (*simp add:comp-def map-option-bind case-prod-beta*)

also have $\dots = \text{distr } \mathcal{B} \mathcal{D} (\lambda x. \text{consumed-bits } ?p x)$
by (*intro arg-cong[where f=distr $\mathcal{B} \mathcal{D}$] ext*)
(auto simp:consumed-bits-inf-iff[OF wf-rep-rand-alg] split:bind-split)

also have $\dots = \text{measure-pmf (coin-usage-of-ra-aux p)}$
unfolding *coin-usage-of-ra-aux.rep-eq used-bits-distr-def* **by** *simp*

finally have *measure-pmf (map-spmf snd (tspmf-of-ra p)) = measure-pmf (coin-usage-of-ra-aux p)*
by *simp*

hence $0:\text{map-spmf snd (tspmf-of-ra p)} = \text{coin-usage-of-ra-aux p}$
using *measure-pmf-inject* **by** *auto*

show *?thesis*
unfolding *coin-usage-of-tspmf-def 0[symmetric] coin-usage-of-ra-def map-pmf-comp*
by (*intro map-pmf-cong*) (*auto split:option.split*)

qed

lemma *expected-coin-usage-of-tspmf-correct*:
expected-coin-usage-of-tspmf (tspmf-of-ra p) = expected-coin-usage-of-ra p
unfolding *expected-coin-usage-of-tspmf-def coin-usage-of-tspmf-correct*
expected-coin-usage-of-ra-def **by** *simp*

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  $\implies$   $\lfloor x \rfloor \leq \lceil y \rceil - 1$ 
by linarith

```

lemma *combine-spmf-set-coin-spmf*:

```

fixes f :: nat  $\Rightarrow$  'a spmf
fixes k :: nat
shows pmf-of-set  $\{..<2^k\} \gg (\lambda l. \text{coin-spmf} \gg (\lambda b. f (2*l + \text{of-bool } b))) =$ 
  pmf-of-set  $\{..<2^{k+1}\} \gg f$  (is ?L = ?R)

```

proof –

```

let ?f =  $(\lambda(l::\text{nat}, b). 2*l + \text{of-bool } b)$ 
let ?coin = pmf-of-set (UNIV :: bool set)

```

```

have [simp]:  $\{..<(2::\text{nat})^k\} \neq \{\}$ 
by (simp add: lessThan-empty-iff)

```

```

have bij:bij-betw ?f ( $\{..<2^k\} \times \text{UNIV}$ )  $\{..<2^{k+1}\}$ 
by (intro bij-betwI[where g= $(\lambda x. (x \text{ div } 2, \text{ odd } x))$ ]] auto)

```

```

have pmf (pair-pmf (pmf-of-set  $\{..<2^k\}$ ) ?coin) x =
  pmf (pmf-of-set ( $\{..<2^k\} \times \text{UNIV}$ )) x for x :: nat  $\times$  bool
by (cases x) (simp add:pmf-pair indicator-def)

```

```

hence 0:pair-pmf (pmf-of-set  $\{..<(2::\text{nat})^k\}$ ) ?coin = pmf-of-set ( $\{..<2^k\} \times \text{UNIV}$ )
by (intro pmf-eqI simp)

```

```

have map-pmf ?f (pmf-of-set ( $\{..<2^k\} \times \text{UNIV}$ )) = pmf-of-set (?f ' ( $\{..<2^k\} \times \text{UNIV}$ ))
using bij-betw-imp-inj-on[OF bij] by (intro map-pmf-of-set-inj auto)

```

```

also have ... = pmf-of-set  $\{..<2^{k+1}\}$ 
by (intro arg-cong[where f=pmf-of-set] bij-betw-imp-surj-on[OF bij])

```

```

finally have 1:map-pmf ?f (pmf-of-set ( $\{..<2^k\} \times \text{UNIV}$ )) = pmf-of-set  $\{..<2^{k+1}\}$ 
by simp

```

```

have ?L = pmf-of-set  $\{..<2^k\} \gg (\lambda l. ?coin \gg (\lambda b. f (2*l + \text{of-bool } b)))$ 
unfolding spmf-of-set-def bind-spmf-def spmf-of-pmf-def by (simp add:bind-map-pmf)
also have ... = pair-pmf (pmf-of-set  $\{..<2^k\}$ ) ?coin  $\gg (\lambda(l,b). f (2*l + \text{of-bool } b))$ 
unfolding pair-pmf-def by (simp add:bind-assoc-pmf bind-return-pmf)
also have ... = map-pmf  $(\lambda(l,b). 2 * l + \text{of-bool } b)$  (pmf-of-set ( $\{..<2^k\} \times \text{UNIV}$ ))  $\gg f$ 
unfolding 0 bind-map-pmf by (simp add:comp-def case-prod-beta')
also have ... = ?R

```

```

unfolding 1 by simp
finally show ?thesis by simp

```

qed

lemma *count-ints-in-range*:

```

real (card  $\{x. \text{of-int } x \in \{u..v\}\}) \geq v - u - 1$  (is ?L  $\geq$  ?R)

```

proof (*cases u \leq v*)

case True

```

have 0:of-int x  $\in \{u..v\} \iff x \in \{\lceil u \rceil .. \lfloor v \rfloor\}$  for x by simp linarith

```

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

have $v - u - 1 \leq \lfloor v \rfloor - \lceil u \rceil + 1$ **using** *True* **by** *linarith*
also have $\dots = \text{real} (\text{nat} (\lfloor v \rfloor - \lceil u \rceil + 1))$ **using** *True* **by** (*intro of-nat-nat[symmetric]*) *linarith*
also have $\dots = \text{card} \{\lceil u \rceil.. \lfloor v \rfloor\}$ **by** *simp*
also have $\dots = ?L$
unfolding *0* **by** (*intro arg-cong[where f=real]* *arg-cong[where f=card]*) *auto*
finally show *?thesis* **by** *simp*

next

case *False*
hence $v - u - 1 \leq 0$ **by** *simp*
thus *?thesis* **by** *simp*

qed

partial-function (*random-alg*) *dice-roll-step-ra* :: *real* \Rightarrow *real* \Rightarrow *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* ($l + (h/2) * \text{of-bool } b$) ($h/2$) }
)

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

partial-function (*spmf*) *drs-tspmf* :: *real* \Rightarrow *real* \Rightarrow *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* ($l + (h/2) * \text{of-bool } b$) ($h/2$) }
)

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

lemma *drs-tspmf*: *drs-tspmf l u* = *tspmf-of-ra (dice-roll-step-ra l u)*

proof –

include *lifting-syntax*
have ($(=) \implies (=) \implies \text{rel-tspmf-of-ra}$) *drs-tspmf dice-roll-step-ra*
 unfolding *drs-tspmf-def dice-roll-step-ra-def*
 apply (*rule rel-funD[OF curry-transfer]*)
 apply (*rule fixp-rel-tspmf-of-ra-parametric[OF drs-tspmf.mono dice-roll-step-ra.mono]*)
 by *transfer-prover*
thus *?thesis*
 unfolding *rel-fun-def rel-tspmf-of-ra-def* **by** *auto*

qed

lemma *dice-roll-ra-tspmf*: *tspmf-of-ra (dice-roll-ra n)* = *dice-roll-tspmf n*
unfolding *dice-roll-ra-def dice-roll-tspmf-def map-ra-def tspmf-of-ra-bind tspmf-of-ra-return*
 drs-tspmf **by** *simp*

lemma *dice-roll-step-tspmf-lb*:

assumes $h > 0$
shows *coin-tspmf* \gg ($\lambda b. \text{drs-tspmf } (l + (h/2) * \text{of-bool } b) (h/2)$) \leq_R *drs-tspmf l h*

proof (*cases* $\lfloor l \rfloor = \lceil l+h \rceil - 1$)

case *True*
hence $2:\text{drs-tspmf } l h = \text{return-tspmf } \lfloor l \rfloor$
 by (*subst drs-tspmf.simps*) *simp*

have $0: \lfloor l + h / 2 * \text{of-bool } b \rfloor = \lfloor l \rfloor$ **for** *b*

proof –

have $\lfloor l + h / 2 * \text{of-bool } b \rfloor \leq \lfloor l + h / 2 \rfloor$
using *assms* **by** (*intro floor-mono add-mono*) *auto*
also have $\dots \leq \lfloor l + h \rfloor - 1$
using *assms* **by** (*intro floor-le-ceil-minus-one add-strict-left-mono*) *auto*
also have $\dots = \lfloor l \rfloor$ **using** *True* **by** *simp*
finally have $\lfloor l + h / 2 * \text{of-bool } b \rfloor \leq \lfloor l \rfloor$ **by** *simp*
moreover have $\lfloor l \rfloor \leq \lfloor l + h / 2 * \text{of-bool } b \rfloor$
using *assms* **by** (*intro floor-mono*) *auto*
ultimately show *?thesis* **by** *simp*
qed

have 1: $\lfloor l + h / 2 * \text{of-bool } b + h / 2 \rfloor - 1 = \lfloor l \rfloor$ **for** *b*
proof –
have $\lfloor l + h / 2 * \text{of-bool } b + h / 2 \rfloor - 1 \leq \lfloor l + h \rfloor - 1$
using *assms* **by** (*intro diff-mono ceiling-mono*) *auto*
also have $\dots = \lfloor l \rfloor$ **using** *True* **by** *simp*
finally have $\lfloor l + h / 2 * \text{of-bool } b + h / 2 \rfloor - 1 \leq \lfloor l \rfloor$ **by** *simp*
moreover have $\lfloor l \rfloor \leq \lfloor l + h / 2 * \text{of-bool } b + h / 2 \rfloor - 1$
using *assms* **by** (*intro floor-le-ceil-minus-one*) *auto*
ultimately show *?thesis* **by** *simp*
qed

have 3: *drs-tspmf* ($l + (h/2) * \text{of-bool } b$) ($h/2$) = *return-tspmf* $\lfloor l \rfloor$ **for** *b*
using *0 1* **by** (*subst drs-tspmf.simps*) *simp*

show *?thesis*
unfolding *2 3 bind-tspmf-def coin-tspmf-def pair-spmf-alt-def*
by (*simp add:bind-spmf-const ord-tspmf-map-spmf*)
next
case *False*
thus *?thesis*
by (*subst drs-tspmf.simps*) (*auto intro:ord-tspmf-refl*)
qed

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

lemma *dice-roll-step-tspmf-expand:*

assumes $h > 0$
shows $\text{coins } k \ggg (\lambda l. \text{use-coins } k (\text{drs-tspmf } (\text{real } l * h) h)) \leq_R \text{drs-tspmf } 0 (h * 2^k)$
using *assms*

proof (*induction k arbitrary:h*)

case *0*
have $\{..<\text{Suc } 0\} = \{0\}$ **by** *auto*
then show *?case*
by (*auto intro:ord-tspmf-use-coins simp:pmf-of-set-singleton bind-return-pmf*)

next

case (*Suc k*)
have ($\text{coins } (k+1) \ggg (\lambda l. \text{use-coins } (k+1) (\text{drs-tspmf } (\text{real } l * h) h))$) =
 $\text{coins } k \ggg (\lambda l. \text{coin-spmf } \ggg (\lambda b. \text{use-coins } (k+1) (\text{drs-tspmf } (\text{real } (2 * l + \text{of-bool } b) * h) h)))$
by (*intro combine-spmf-set-coin-spmf[symmetric]*)
also have $\dots = \text{coins } k \ggg (\lambda l. \text{use-coins } (k+1) (\text{coin-spmf } \ggg$
 $(\lambda b. \text{drs-tspmf } (\text{real } l * (2 * h) + h * \text{of-bool } b) h)))$
unfolding *use-coins-def map-spmf-conv-bind-spmf* **by** (*simp add:algebra-simps*)
also have $\dots = \text{coins } k \ggg (\lambda l. \text{use-coins } k (\text{coin-tspmf } \ggg$
 $(\lambda b. \text{drs-tspmf } (\text{real } l * (2 * h) + h * \text{of-bool } b) h)))$
unfolding *coin-tspmf-split use-coins-add* **by** *simp*
also have $\dots = \text{coins } k \ggg (\lambda l. \text{use-coins } k (\text{coin-tspmf } \ggg$
 $(\lambda b. \text{drs-tspmf } (\text{real } l * (2 * h) + ((2 * h) / 2) * \text{of-bool } b) ((2 * h) / 2))))$

using $Suc(2)$ **by** *simp*
also have $\dots \leq_R \text{coins } k \gg (\lambda l. \text{use-coins } k (\text{drs-tspmf } (\text{real } l * (2 * h)) (2 * h)))$
using $Suc(2)$ **by** (*intro ord-tspmf-bind-pmf ord-tspmf-use-coins-2 dice-roll-step-tspmf-lb*) *simp*
also have $\dots \leq_R \text{drs-tspmf } 0 ((2 * h) * 2^k)$
using $Suc(2)$ **by** (*intro Suc(1)*) *auto*
also have $\dots = \text{drs-tspmf } 0 (h * 2^{k+1})$
unfolding *power-add* **by** (*simp add:algebra-simps*)
finally show *?case*
by *simp*
qed

lemma *dice-roll-step-tspmf-approx*:

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

proof –

have $?L = \text{coins } k \gg$
 $(\lambda l. \text{use-coins } k (\text{if } \lfloor \text{real } l * h \rfloor = \lceil (l+1) * h \rceil - 1 \text{ then } \text{return-tspmf } \lfloor l * h \rfloor \text{ else } \text{return-pmf } \text{None}))$
unfolding *f-def return-tspmf-def use-coins-def map-pmf-def*
by (*simp add:if-distribR if-distrib bind-return-pmf algebra-simps cong:if-cong*)
also have $\dots \leq_R \text{coins } k \gg (\lambda l. \text{use-coins } k (\text{drs-tspmf } (\text{real } l * h) h))$
by (*subst drs-tspmf.simps, intro ord-tspmf-bind-pmf ord-tspmf-use-coins-2*)
 $(\text{simp add:ord-tspmf-min ord-tspmf-refl algebra-simps})$
also have $\dots \leq_R \text{drs-tspmf } 0 (h * 2^k)$
by (*intro dice-roll-step-tspmf-expand assms*)
finally show *?thesis* **by** *simp*

qed

lemma *dice-roll-step-spmf-approx*:

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

proof –

have $0: ?L = \text{result } (\text{map-pmf } (\lambda l. \text{if } \lfloor l * h \rfloor = \lceil (l+1) * h \rceil - 1 \text{ then } \text{Some } (\lfloor l * h \rfloor, k) \text{ else } \text{None}) (\text{coins } k))$

unfolding *result-def map-pmf-comp f-def* **by** (*intro map-pmf-cong refl*) *auto*

show *?thesis*

unfolding 0 **using** *assms result-mono[OF dice-roll-step-tspmf-approx]* **by** *simp*

qed

lemma *spmf-dice-roll-step-lb*:

assumes $j < n$
shows $\text{spmf } (\text{result } (\text{drs-tspmf } 0 (\text{of-nat } n))) (\text{of-nat } j) \geq 1/n$ (**is** $?L \geq ?R$)

proof (*rule ccontr*)

assume $\neg(\text{spmf } (\text{result } (\text{drs-tspmf } 0 (\text{of-nat } n))) (\text{of-nat } j) \geq 1/n)$

hence $a: ?L < 1/n$ **by** *simp*

define $k :: \text{nat}$ **where** $k = \text{nat } \lfloor 2 - \log 2 (1/n - ?L) \rfloor$

define h **where** $h = \text{real } n / 2^k$

define f **where** $f l = (\text{if } \lfloor l * h \rfloor = \lceil (l+1) * h \rceil - 1 \text{ then } \text{Some } \lfloor l * h \rfloor \text{ else } \text{None})$ **for** $l :: \text{nat}$

have *h-gt-0*: $h > 0$ **using** *assms* **unfolding** *h-def* **by** *auto*

have *n-gt-0*: $\text{real } n > 0$ **using** *assms* **by** *simp*

have $0: x < 2^k$ **if** $\text{real } x \leq (\text{real } j + 1) * 2^k / n - 1$ **for** x

proof –
 have $\text{real } x \leq (\text{real } j+1) * 2^{\wedge}k/n - 1$ **using** *that by simp*
 also have $\dots \leq \text{real } n * 2^{\wedge}k/n - 1$
 using *assms by (intro diff-mono divide-right-mono mult-right-mono) auto*
 also have $\dots \leq 2^{\wedge}k - 1$ **by** *simp*
 finally have $\text{real } x \leq 2^{\wedge}k - 1$ **by** *simp*
 thus *?thesis* **using** *nat-less-real-le* **by** *auto*
qed
 have 2: $\text{int } \{x. P (\text{real } x)\} = \{x. P (\text{real-of-int } x)\}$ **if** $\bigwedge x. P x \implies x \geq 0$ **for** P
proof –
 have *bij-betw* $\text{int } \{x. P (\text{real } x)\} \{x. P (\text{real-of-int } x)\}$
 using *that by (intro bij-betwI[where g=nat]) force+*
 thus *?thesis*
 using *bij-betw-imp-surj-on* **by** *auto*
qed
 have 1: $\text{real } j * 2^{\wedge}k/n \geq 0$ **by** *auto*

 have 3: $\lfloor \text{real } l * h \rfloor \leq \lceil \text{real } (l+1) * h \rceil - 1$ **for** l
 using *h-gt-0* **by** *(intro floor-le-ceil-minus-one) force*

 have 2 = $(1/n - ?L) * 2^{\text{powr } (1 - \log 2 (1/n - ?L))}$
 using *a n-gt-0 unfolding powr-diff by (subst powr-log-cancel) (auto simp: divide-simps)*
 also have $\dots < (1/n - ?L) * 2^{\text{powr } \lfloor 2 - \log 2 (1/n - ?L) \rfloor}$
 using *a by (intro mult-strict-left-mono powr-less-mono) linarith+*
 also have $\dots \leq (1/n - ?L) * 2^{\text{powr } \text{real } k}$
 using *a unfolding k-def by (intro mult-left-mono powr-mono) auto*
 also have $\dots = (1/n - ?L) * 2^{\wedge}k$ **by** *(subst powr-realpow) auto*
 finally have $2 < (1/n - ?L) * 2^{\wedge}k$ **by** *simp*
 hence $?L < 1/n - 2/2^{\wedge}k$ **by** *(simp add: field-simps)*
 also have $\dots = (((\text{real } j+1) * 2^{\wedge}k/n - 1) - (\text{real } j * 2^{\wedge}k/n - 1)) / 2^{\wedge}k$
 using *n-gt-0* **by** *(simp add: field-simps)*
 also have $\dots \leq \text{real } (\text{card } \{x. \text{of-int } x \in \{\text{real } j * 2^{\wedge}k/n..(\text{real } j+1) * 2^{\wedge}k/n - 1\}\}) / 2^{\wedge}k$
 by *(intro divide-right-mono count-ints-in-range) auto*
 also have $\dots = \text{real } (\text{card } (\text{int } \{x. \text{real } x \in \{\text{real } j * 2^{\wedge}k/n..(\text{real } j+1) * 2^{\wedge}k/n - 1\}\})) / 2^{\wedge}k$
 using *order-trans[OF 1]* **by** *(subst 2) auto*
 also have $\dots = \text{real } (\text{card } \{x. \text{real } x \in \{\text{real } j * 2^{\wedge}k/n..(\text{real } j+1) * 2^{\wedge}k/n - 1\}\}) / 2^{\wedge}k$
 by *(subst card-image) auto*
 also have $\dots = \text{real } (\text{card } \{x. x < 2^{\wedge}k \wedge \text{real } x \in \{\text{real } j * 2^{\wedge}k/n..(\text{real } j+1) * 2^{\wedge}k/n - 1\}\}) / 2^{\wedge}k$
 using *0* **by** *(intro arg-cong[where f=λx. real (card x)/2^k]) auto*
 also have $\dots = \text{real } (\text{card } \{l. l < 2^{\wedge}k \wedge \text{real } j \leq \text{real } l * h \wedge (1 + \text{real } l) * h \leq \text{real } j+1\}) / 2^{\wedge}k$
 using *assms unfolding h-def*
 by *(intro arg-cong[where f=λx. real (card x)/2^k] Collect-cong) (auto simp: field-simps)*
 also have $\dots = \text{measure } (\text{coins } k) \{l. \text{real } j \leq \text{real } l * h \wedge \text{real } (l+1) * h \leq \text{real } j+1\}$
 by *(subst measure-pmf-of-set) (simp-all add: lessThan-empty-iff Int-def)*
 also have $\dots = \text{measure } (\text{coins } k) \{l. \text{int } j \leq \lfloor \text{real } l * h \rfloor \wedge \lceil \text{real } (l+1) * h \rceil - 1 \leq \text{int } j\}$
 by *(intro arg-cong2[where f=measure] refl Collect-cong) linarith*
 also have $\dots = \text{measure } (\text{coins } k) \{l. \text{int } j = \lfloor \text{real } l * h \rfloor \wedge \text{int } j = \lceil \text{real } (l+1) * h \rceil - 1\}$
 using *3 order.trans order-antisym*
 by *(intro arg-cong2[where f=measure] refl Collect-cong iffI, blast, simp)*
 also have $\dots = \text{spmf } (\text{map-pmf } f (\text{coins } k)) j$
 unfolding *pmf-map f-def vimage-def*
 by *(intro arg-cong2[where f=measure] refl Collect-cong) auto*
 also have $\dots \leq \text{spmf } (\text{result } (\text{drs-tspmf } 0 (h * 2^{\wedge}k))) j$
 unfolding *f-def* **by** *(intro ord-spmf-eq-leD dice-roll-step-spmf-approx h-gt-0)*
 also have $\dots = ?L$
 unfolding *h-def* **by** *simp*
 finally have $?L < ?L$ **by** *simp*
 thus *False* **by** *simp*

qed

lemma *dice-roll-correct-aux*:

assumes $n > 0$

shows $\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n)) = \text{spmf-of-set } \{0..<n\}$

proof –

have $\text{weight-spmf } (\text{spmf-of-set } \{0..<int\ } n) \geq \text{weight-spmf } (\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n)))$

using *assms unfolding weight-spmf-of-set*

by (*simp add:lessThan-empty-iff weight-spmf-le-1*)

moreover have $\text{spmf } (\text{spmf-of-set } \{0..<int\ } n) \ x \leq \text{spmf } (\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n))) \ x$

for x

proof (*cases* $x < n \wedge x \geq 0$)

case *True*

hence $\text{spmf } (\text{spmf-of-set } \{0..<int\ } n) \ x = 1/n$

unfolding *spmf-of-set* by *auto*

also have $\dots \leq \text{spmf } (\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n))) \ (\text{of-nat } (nat \ x))$

using *True* by (*intro spmf-dice-roll-step-lb*) *auto*

also have $\dots = \text{spmf } (\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n))) \ x$

using *True* by (*subst of-nat-nat*) *auto*

finally show *?thesis* by *simp*

next

case *False*

hence $\text{spmf } (\text{spmf-of-set } \{0..<int\ } n) \ x = 0$

unfolding *spmf-of-set* by *auto*

then show *?thesis* by *simp*

qed

hence $\text{ord-spmf } (=) \ (\text{spmf-of-set } \{0..<int\ } n) \ (\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n)))$

by (*intro ord-spmf-increaseI refl*) *auto*

ultimately show *?thesis*

by (*intro eq-iff-ord-spmf[symmetric]*) *auto*

qed

theorem *dice-roll-correct*:

assumes $n > 0$

shows

$\text{result } (\text{dice-roll-tspmf } n) = \text{spmf-of-set } \{..<n\}$ (is $?L = ?R$)

$\text{spmf-of-ra } (\text{dice-roll-ra } n) = \text{spmf-of-set } \{..<n\}$

proof –

have $\text{bij:bij-betw } nat \ \{0..<int\ } n \ \{..<n\}$

by (*intro bij-betwI[where g=int]*) *auto*

have $?L = \text{map-spmf } nat \ (\text{spmf-of-set } \{0..<int\ } n)$

unfolding *dice-roll-tspmf-def dice-roll-correct-aux[OF assms]* *result-bind result-return*
map-spmf-conv-bind-spmf by *simp*

also have $\dots = \text{spmf-of-set } (nat \ ‘ \{0..<int\ } n)$

by (*intro map-spmf-of-set-inj-on inj-onI*) *auto*

also have $\dots = ?R$

using *bij-betw-imp-surj-on[OF bij]* by (*intro arg-cong[where f=spmf-of-set]*) *auto*

finally show $?L = ?R$ by *simp*

thus $\text{spmf-of-ra } (\text{dice-roll-ra } n) = ?R$

using *spmf-of-tspmf dice-roll-ra-tspmf* by *metis*

qed

lemma *dice-roll-consumption-bound*:

assumes $n > 0$

shows $\text{measure } (\text{coin-usage-of-tspmf } (\text{dice-roll-tspmf } n)) \ \{x. \ x > \text{enat } k \} \leq \text{real } n/2^k$

(is $?L \leq ?R$)

proof –

define h **where** $h = \text{real } n / 2^k$
define f **where** $f l = (\text{if } \lfloor l * h \rfloor = \lceil (l+1) * h \rceil - 1 \text{ then } \text{Some } (\lfloor l * h \rfloor, k) \text{ else } \text{None})$ **for** $l :: \text{nat}$

have $h\text{-gt-0}$: $h > 0$
using assms **unfolding** $h\text{-def}$
by $(\text{intro } \text{divide-pos-pos}) \text{ auto}$
have 0 : $\text{real } n = h * 2^k$
unfolding $h\text{-def}$ **by** simp

have 1 : $\lfloor \text{real } l * h \rfloor < \lfloor (\text{real } l + 1) * h \rfloor$ **if** $\lfloor \text{real } l * h \rfloor \neq \lceil (\text{real } l + 1) * h \rceil - 1$ **for** l
proof –
have $\lfloor \text{real } l * h \rfloor \leq \lceil (\text{real } l + 1) * h \rceil - 1$
using $h\text{-gt-0}$ **by** $(\text{intro } \text{floor-le-ceil-minus-one}) \text{ force}$
hence $\lfloor \text{real } l * h \rfloor < \lceil (\text{real } l + 1) * h \rceil - 1$
using that **by** simp
also have $\dots \leq \lfloor (\text{real } l + 1) * h \rfloor$
by linarith
finally show $?thesis$ **by** simp
qed

have $?L = \text{measure } (\text{coin-usage-of-tspmf } (\text{drs-tspmf } 0 \ n)) \{x. x > \text{enat } k\}$
unfolding $\text{dice-roll-tspmf-def}$ $\text{coin-usage-of-tspmf-bind-return}$ **by** simp
also have $\dots \leq \text{measure } (\text{coin-usage-of-tspmf } (\text{map-pmf } f \ (\text{coins } k))) \{x. x > \text{enat } k\}$
unfolding $f\text{-def } 0$
by $(\text{intro } \text{coin-usage-of-tspmf-mono-rev } \text{dice-roll-step-tspmf-approx } h\text{-gt-0})$
also have $\dots = \text{measure } (\text{coins } k) \{l. \lfloor \text{real } l * h \rfloor \neq \lceil (\text{real } l + 1) * h \rceil - 1\}$
unfolding $\text{coin-usage-of-tspmf-def}$ map-pmf-comp
by $(\text{simp } \text{add:vimage-def } f\text{-def } \text{algebra-simps } \text{split:option.split})$
also have $\dots \leq \text{measure } (\text{coins } k) \{l. \lfloor \text{real } l * h \rfloor < \lfloor (\text{real } l + 1) * h \rfloor\}$
using 1 **by** $(\text{intro } \text{measure-pmf.finite-measure-mono } \text{subsetI}) \ (\text{simp-all})$
also have $\dots = (\int l. \text{indicator } \{l. \lfloor \text{real } l * h \rfloor < \lfloor (\text{real } l + 1) * h \rfloor\} \ l \ \partial(\text{coins } k))$
by simp
also have $\dots = (\sum j < 2^k. \text{indicat-real } \{l. \lfloor \text{real } l * h \rfloor < \lfloor (\text{real } l + 1) * h \rfloor\} \ j * \text{pmf } (\text{coins } k) \ j)$
by $(\text{intro } \text{integral-measure-pmf-real}[\text{where } A = \{.. < 2^k\}]) \ (\text{simp-all } \text{add:lessThan-empty-iff})$
also have $\dots = (\sum l < 2^k. \text{of-bool } (\lfloor \text{real } l * h \rfloor < \lfloor (\text{real } l + 1) * h \rfloor)) / 2^k$
by $(\text{simp } \text{add:lessThan-empty-iff } \text{indicator-def } \text{flip:sum-divide-distrib})$
also have $\dots \leq (\sum l < 2^k. \text{of-int } \lfloor \text{real } (\text{Suc } l) * h \rfloor - \text{of-int } \lfloor \text{real } l * h \rfloor) / 2^k$
using $h\text{-gt-0}$ int-less-real-le
by $(\text{intro } \text{divide-right-mono } \text{sum-mono}) \ (\text{auto } \text{intro:floor-mono } \text{simp:algebra-simps})$
also have $\dots = \text{of-int } \lfloor 2^k * h \rfloor / 2^k$
by $(\text{subst } \text{sum-lessThan-telescope}) \ \text{simp}$
also have $\dots = \text{real } n / 2^k$
unfolding $h\text{-def}$ **by** simp
finally show $?thesis$
by simp
qed

lemma $\text{dice-roll-expected-consumption-aux}$:

assumes $n > (0 :: \text{nat})$
shows $\text{expected-coin-usage-of-tspmf } (\text{dice-roll-tspmf } n) \leq \log 2 \ n + 2$ **(is** $?L \leq ?R$ **)**

proof –

define $k0$ **where** $k0 = \text{nat } \lceil \log 2 \ n \rceil$
define δ **where** $\delta = \log 2 \ n - \lceil \log 2 \ n \rceil$

have 0 : $\text{ennreal } (\text{measure } (\text{coin-usage-of-tspmf } (\text{dice-roll-tspmf } n)) \{x. x > \text{enat } k\}) \leq$
 $\text{ennreal } (\text{min } (\text{real } n / 2^k) \ 1)$ **(is** $?L1 \leq ?R1$ **)** **for** k
by $(\text{intro } \text{iffD2}[\text{OF } \text{ennreal-le-iff}] \ \text{min.boundedI } \text{dice-roll-consumption-bound}[\text{OF } \text{assms}]) \ \text{auto}$

have $1[simp]: (2::ennreal)^\wedge k < Orderings.top$ **for** $k::nat$
using *ennreal-numeral-less-top power-less-top-ennreal* **by** *blast*

have $(\sum k. ennreal ((1/2)^\wedge k)) = ennreal (\sum k. ((1/2)^\wedge k))$
by *(intro suminf-ennreal2) auto*
also have $\dots = ennreal 2$
by *(subst suminf-geometric) simp-all*
finally have $2:(\sum k. ennreal ((1/2)^\wedge k)) = ennreal 2$
by *simp*

have $real\ n \geq 1$
using *assms* **by** *simp*
hence $3: \log 2 (real\ n) \geq 0$
by *simp*

have $real-of-int \lceil \log 2 (real\ n) \rceil \leq 1 + \log 2 (real\ n)$
by *linarith*
hence $4: \delta+1 \in \{0..1\}$
unfolding $\delta-def$ **by** *(auto simp:algebra-simps)*

have *twop-conv: convex-on UNIV* $(\lambda x. 2\ powr (x::real))$
using *convex-on-exp* **where** $l=ln\ 2$ **unfolding** *powr-def*
by *(auto simp:algebra-simps)*
have $5: 2\ powr\ x \leq 1 + x$ **if** $x \in \{0..1\}$ **for** $x::real$
using *that convex-onD[OF twop-conv, where x=0 and y=1 and t=x]*
by *(simp add:algebra-simps)*

have $?L = (\sum k. ennreal (measure (coin-usage-of-tspmf (dice-roll-tspmf\ n)) \{x. x > enat\ k\}))$
unfolding *expected-coin-usage-of-tspmf* **by** *simp*
also have $\dots \leq (\sum k. ennreal (min (real\ n/2^\wedge k) 1))$
by *(intro suminf-le summableI 0)*
also have $\dots = (\sum k. ennreal (min (real\ n/2^\wedge(k+k0)) 1)) + (\sum k < k0. ennreal (min (real\ n/2^\wedge k)$
 $1))$
by *(intro suminf-offset summableI)*
also have $\dots \leq (\sum k. ennreal (real\ n/2^\wedge(k+k0))) + (\sum k < k0. 1)$
by *(intro add-mono suminf-le summableI sum-mono iffD2[OF ennreal-le-iff]) auto*
also have $\dots = (\sum k. ennreal (real\ n / 2^\wedge k0) * ennreal ((1/2)^\wedge k)) + of-nat\ k0$
by *(intro suminf-cong arg-cong2[where f=(+)]*
(simp-all add: ennreal-mult[symmetric] power-add divide-simps)
also have $\dots = ennreal (real\ n / 2^\wedge k0) * (\sum k. ennreal ((1/2)^\wedge k)) + ennreal (real\ k0)$
unfolding *ennreal-of-nat-eq-real-of-nat* **by** *simp*
also have $\dots = ennreal (real\ n / 2^\wedge k0 * 2 + real\ k0)$
unfolding 2 **by** *(subst ennreal-mult[symmetric]) simp-all*
also have $\dots = ennreal (real\ n / 2\ powr\ k0 * 2 + real\ k0)$
by *(subst powr-realpow) auto*
also have $\dots = ennreal (real\ n / 2\ powr \lceil \log 2\ n \rceil * 2 + real\ k0)$
using 3 **unfolding** $k0-def$ **by** *(subst of-nat-nat) auto*
also have $\dots = ennreal (real\ n / 2\ powr (\log 2\ n - \delta) * 2 + real\ k0)$
unfolding $\delta-def$ **by** *simp*
also have $\dots = ennreal (2\ powr\ \delta * 2\ powr\ 1 + real\ k0)$
using *assms* **unfolding** *powr-diff* **by** *(subst powr-log-cancel) auto*
also have $\dots \leq ennreal (1+(\delta+1) + real\ k0)$
using 4 **unfolding** *powr-add[symmetric]*
by *(intro iffD2[OF ennreal-le-iff] add-mono 5) auto*
also have $\dots = ?R$
using 3 **unfolding** $\delta-def\ k0-def$ **by** *(subst of-nat-nat) auto*
finally show *?thesis*
by *simp*

qed

theorem *dice-roll-coin-usage*:

assumes $n > (0::nat)$

shows $expected\text{-}coin\text{-}usage\text{-}of\text{-}ra (dice\text{-}roll\text{-}ra\ n) \leq \log 2\ n + 2$ (**is** $?L \leq ?R$)

proof –

have $?L = expected\text{-}coin\text{-}usage\text{-}of\text{-}tspmf (tspmf\text{-}of\text{-}ra (dice\text{-}roll\text{-}ra\ n))$

unfolding $expected\text{-}coin\text{-}usage\text{-}of\text{-}tspmf\text{-}correct[symmetric]$ **by** *simp*

also have $\dots = expected\text{-}coin\text{-}usage\text{-}of\text{-}tspmf (dice\text{-}roll\text{-}tspmf\ n)$

unfolding $dice\text{-}roll\text{-}ra\text{-}tspmf$ **by** *simp*

also have $\dots \leq ?R$

by (*intro dice-roll-expected-consumption-aux assms*)

finally show *?thesis*

by *simp*

qed

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 $pcg\text{-}mult = 6364136223846793005$

definition *pcg-increment* :: $64\ word$

where $pcg\text{-}increment = 1442695040888963407$

fun *pcg-rotr* :: $32\ word \Rightarrow nat \Rightarrow 32\ word$

where $pcg\text{-}rotr\ x\ r = Bit\text{-}Operations.or (drop\text{-}bit\ r\ x) (push\text{-}bit\ (32-r)\ x)$

fun *pcg-step* :: $64\ word \Rightarrow 64\ word$

where $pcg\text{-}step\ state = state * pcg\text{-}mult + pcg\text{-}increment$

Based on [9, Section 6.3.1]:

fun *pcg-get* :: $64\ word \Rightarrow 32\ word$

where $pcg\text{-}get\ state =$

$(let\ count = unsigned (drop\text{-}bit\ 59\ state);$

$x = xor (drop\text{-}bit\ 18\ state)\ state$

$in\ pcg\text{-}rotr (ucast (drop\text{-}bit\ 27\ x))\ count)$

fun *pcg-init* :: $64\ word \Rightarrow 64\ word$

where $pcg\text{-}init\ seed = pcg\text{-}step (seed + pcg\text{-}increment)$

definition *to-bits* :: $32\ word \Rightarrow bool\ list$

where $to\text{-}bits\ x = map (\lambda k. bit\ x\ k) [0..<32]$

```

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)

by (*induction* n) (*auto intro:terminates-almost-surely-intros*)

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

using *binary-dice-roll-terminates*

by (*induction* n) (*simp-all add:terminates-almost-surely-intros pmf-of-ra-simps*)

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

```

```

have ((=) ==> rel-spmf-of-ra) binary-geometric-spmf binary-geometric

```

```

  unfolding binary-geometric-def binary-geometric-spmf-def

```

```

  apply (rule fix-ra-parametric[OF binary-geometric-spmf.mono binary-geometric.mono])

```

```

  by transfer-prover

```

```

thus ?thesis

```

```

  unfolding rel-fun-def rel-spmf-of-ra-def by auto

```

qed

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 ≥ 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 (*induction arbitrary:x rule:bernoulli.fixp-induct*)

```

case 1

```

```

  thus ?case by simp

```

```

next

```

```

case 2

```

```

  thus ?case by simp

```

```

next

```

```

case (3 p)

```

```

  thus ?case by (subst bernoulli-ra.simps)

```

```

    (auto intro:ord-spmf-bind-refl simp:spmf-of-ra-simps)

```

qed

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

```
lemma bernoulli-ra-correct: bernoulli  $x = \text{spmf-of-ra } (\text{bernoulli-ra } x)$   
using lossless-bernoulli weight-spmf-le-1 unfolding lossless-spmf-def  
by (intro eq-iff-ord-spmf[OF - bernoulli-ra-correct-aux]) auto
```

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  $p = \text{pmf-of-ra } (\text{bernoulli-ra } p)$   
unfolding terminates-almost-surely-def pmf-of-ra-def bernoulli-ra-correct[symmetric]  
by (simp-all add: bernoulli-eq-bernoulli-pmf pmf-of-spmf)
```

context

```
includes lifting-syntax  
begin
```

```
lemma bernoulli-ra-transfer [transfer-rule]:  
((=) ==> rel-spmf-of-ra) bernoulli bernoulli-ra  
unfolding rel-fun-def rel-spmf-of-ra-def bernoulli-ra-correct by simp
```

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  $\Rightarrow$  nat random-alg where  
geometric-ra  $p = \text{do } \{$   
   $b \leftarrow \text{bernoulli-ra } p;$   
   $\text{if } b \text{ then return-ra } 0 \text{ else map-ra } ((+) 1) (\text{geometric-ra } p)$   
   $\}$   
declare geometric-ra.simps[code]
```

```
lemma geometric-ra-correct: spmf-of-ra (geometric-ra  $x$ ) = geometric-spmf  $x$ 
```

proof –

```
include lifting-syntax  
have ((=) ==> rel-spmf-of-ra) geometric-spmf geometric-ra  
  unfolding geometric-ra-def geometric-spmf-def  
  apply (rule fix-ra-parametric[OF geometric-spmf.mono geometric-ra.mono])  
  by transfer-prover  
thus ?thesis  
  unfolding rel-fun-def rel-spmf-of-ra-def by auto
```

qed

Replication of a distribution

```
fun replicate-ra :: nat  $\Rightarrow$  'a random-alg  $\Rightarrow$  'a list random-alg  
where  
   $\text{replicate-ra } 0 f = \text{return-ra } []$  |  
   $\text{replicate-ra } (\text{Suc } n) f = \text{do } \{ xh \leftarrow f; xt \leftarrow \text{replicate-ra } n f; \text{return-ra } (xh\#xt) \}$ 
```

```
fun replicate-spmf :: nat  $\Rightarrow$  'a spmf  $\Rightarrow$  'a list spmf  
where  
   $\text{replicate-spmf } 0 f = \text{return-spmf } []$  |  
   $\text{replicate-spmf } (\text{Suc } n) f = \text{do } \{ xh \leftarrow f; xt \leftarrow \text{replicate-spmf } n f; \text{return-spmf } (xh\#xt) \}$ 
```

```
lemma replicate-ra-correct: spmf-of-ra (replicate-ra  $n f$ ) = replicate-spmf  $n (\text{spmf-of-ra } f)$ 
```

by (induction n) (auto simp : spmf-of-ra-simps)

lemma replicate-spmf-of-pmf: replicate-spmf n (spmof-of-pmf f) = spmf-of-pmf (replicate-pmf n f)
by (induction n) (simp-all add: spmf-of-pmf-bind)

Binomial distribution

definition binomial-ra :: nat \Rightarrow real \Rightarrow nat random-alg
where binomial-ra n p = map-ra (length \circ 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 –

have spmf-of-ra (replicate-ra n (bernoulli-ra p)) = spmf-of-pmf (replicate-pmf n (bernoulli-pmf p))

unfolding replicate-ra-correct bernoulli-ra-correct[symmetric] bernoulli-eq-bernoulli-pmf

by (simp add: replicate-spmf-of-pmf)

thus ?thesis

unfolding binomial-pmf-altdef[OF assms] binomial-ra-def

by (simp flip: map-spmf-of-pmf add: spmf-of-ra-map)

qed

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.