

# Executable Randomized Algorithms

Emin Karayel and Manuel Eberl

April 18, 2024

## 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b><math>\tau</math>-Additivity</b>	<b>2</b>
<b>3</b>	<b>Coin Flip Space</b>	<b>5</b>
<b>4</b>	<b>Randomized Algorithms (Internal Representation)</b>	<b>23</b>
<b>5</b>	<b>Randomized Algorithms</b>	<b>43</b>
5.1	Almost surely terminating randomized algorithms . . . . .	51
<b>6</b>	<b>Tracking Randomized Algorithms</b>	<b>52</b>
<b>7</b>	<b>Tracking SPMFs</b>	<b>61</b>
<b>8</b>	<b>Dice Roll</b>	<b>68</b>
<b>9</b>	<b>A Pseudo-random Number Generator</b>	<b>76</b>
<b>10</b>	<b>Basic Randomized Algorithms</b>	<b>77</b>

## 1 Introduction

In Isabelle, randomized algorithms are usually represented using probability mass functions (PMFs). (These are distributions on the discrete  $\sigma$ -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.

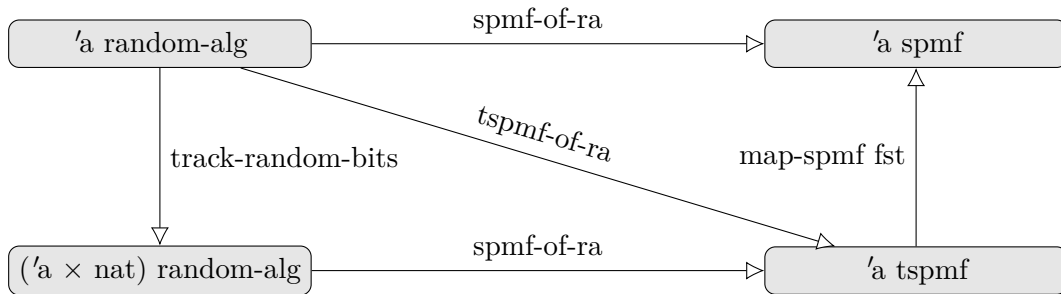


Figure 1: Scott-continuous monad morphisms verified in this work.

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 $\tau$ -Additivity

```
theory Tau-Additivity
  imports HOL-Analysis.Regularity
begin
```

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

the sets is equal to the measure of the union of the family:

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

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

It is possible to generalize that to arbitrary chains<sup>1</sup> of open sets for some measures without the restriction of countability, such measures are called  $\tau$ -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*

---

<sup>1</sup>More generally families closed under pairwise unions.

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

have  $?L = (SUP K \in \{K. K \subseteq \bigcup A \wedge compact\ K\}, \text{emeasure } M\ K)$   
 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*  $x \subseteq y$ )  
 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  $M :: 'a :: \{second-countable-topology, complete-space\}$  *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  $\text{emeasure } M\ (\bigcup A) = (SUP a \in A. \text{emeasure } M\ a)$  (**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  $\in A. emeasure M ((\lambda x. x) - ' a \cap space M)$ )
  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  $x = smap id x$  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  $\sigma$ -algebra:

```

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

```

```

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

```

```

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

```

```

unbundle discrete-sigma-algebra-notation

```

```

lemma map-prod-measurable[measurable]:

```

**assumes**  $f \in M \rightarrow_M M'$   
**assumes**  $g \in N \rightarrow_M N'$   
**shows**  $\text{map-prod } f g \in M \otimes_M N \rightarrow_M M' \otimes_M N'$   
**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-def* **by** (*subst sigma-sets-singletons*) *auto*

**have**  $1$ :  $f -' \{x\} \cap \text{space } M \in \text{sets } M$  **for**  $x$

**proof** (*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** (*auto simp add:set-eq-iff*)

**also have**  $\dots \in \text{sets } M$

**using** *assms*

**by** (*intro sets.compl-sets sets.countable-UN*) *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-def* **by** *auto*

**thus** *?thesis*

**by** (*intro measurable-sigma-sets[OF 0]*) *simp-all*

**qed**

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

**by** (*intro measure-eqI*) (*auto simp add:sets-restrict-space*)

**lemma** (*in prob-space*) *distr-stream-space-snth* [*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** (*rule distr-cong*) (*use assms in auto*)

**also have**  $\dots = \text{distr } (Pi_M UNIV (\lambda i. M)) M (\lambda f. f n)$

**by** (*subst stream-space-eq-distr, subst distr-distr*) (*auto simp: to-stream-def o-def*)

**also have**  $\dots = M$

**by** (*intro distr-PiM-component prob-space-axioms*) *auto*

**finally show** *?thesis* .

**qed**

**lemma** (*in prob-space*) *distr-stream-space-shd* [*simp*]:

**assumes**  $\text{sets } M = \text{sets } N$

**shows**  $\text{distr } (\text{stream-space } M) N \text{shd} = M$

**using** *distr-stream-space-snth[OF assms, of 0]* **by** (*simp del: distr-stream-space-snth*)

**lemma** *shift-measurable*:

**assumes**  $set\ x \subseteq space\ M$   
**shows**  $(\lambda bs. x @- bs) \in stream-space\ M \rightarrow_M stream-space\ M$   
**proof** –  
**have**  $(\lambda bs. (x @- bs) !! n) \in (stream-space\ M) \rightarrow_M M$  **for**  $n$   
**proof** (*cases*  $n < length\ x$ )  
  **case** *True*  
  **have**  $(\lambda bs. (x @- bs) !! n) = (\lambda bs. x ! n)$   
  **using** *True* **by** *simp*  
  **also have**  $\dots \in stream-space\ M \rightarrow_M M$   
  **using** *assms True* **by** (*intro measurable-const*) *auto*  
  **finally show** *?thesis* **by** *simp*  
**next**  
  **case** *False*  
  **have**  $(\lambda bs. (x @- bs) !! n) = (\lambda bs. bs !! (n - length\ x))$   
  **using** *False* **by** *simp*  
  **also have**  $\dots \in (stream-space\ M) \rightarrow_M M$   
  **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**  $A' \in sets\ A$   
**shows**  $restrict-space\ A\ A' \otimes_M M = restrict-space\ (A \otimes_M M)\ (A' \times space\ M)$  (**is**  $?L = ?R$ )

**proof** –

**let**  $?X = ((\cap)\ (A' \times space\ M)) \text{ ‘ } \{a \times b \mid a \in sets\ A \wedge b \in sets\ M\}$   
**have**  $0: A' \subseteq space\ A$   
  **using** *assms sets.sets-into-space* **by** *blast*

**have**  $?X \subseteq \{a \times b \mid a \in sets\ (restrict-space\ A\ A') \wedge b \in sets\ M\}$

**proof** (*rule image-subsetI*)

**fix**  $x$  **assume**  $x \in \{a \times b \mid a \in sets\ A \wedge b \in sets\ M\}$   
**then obtain**  $u\ v$  **where** *uv-def*:  $x = u \times v$   $u \in sets\ A$   $v \in sets\ M$   
  **by** *auto*  
**have**  $1: u \cap A' \in sets\ (restrict-space\ A\ A')$   
  **using** *uv-def(2)* **unfolding** *sets-restrict-space* **by** *auto*  
**have**  $v \subseteq space\ M$   
  **using** *uv-def(3)* *sets.sets-into-space* **by** *auto*  
**hence**  $A' \times space\ M \cap x = (u \cap A') \times v$   
  **unfolding** *uv-def(1)* **by** *auto*  
**also have**  $\dots \in \{a \times b \mid a \in sets\ (restrict-space\ A\ A') \wedge b \in sets\ M\}$   
  **using**  $1$  *uv-def(3)* **by** *auto*

**finally show**  $A' \times space\ M \cap x \in \{a \times b \mid a \in sets\ (restrict-space\ A\ A') \wedge b \in sets\ M\}$   
  **by** *simp*

**qed**

**moreover have**  $\{a \times b \mid a \in sets\ (restrict-space\ A\ A') \wedge b \in sets\ M\} \subseteq ?X$

**proof** (*rule subsetI*)

**fix**  $x$  **assume**  $x \in \{a \times b \mid a \in sets\ (restrict-space\ A\ A') \wedge b \in sets\ M\}$   
**then obtain**  $u\ v$  **where** *uv-def*:  $x = u \times v$   $u \in sets\ (restrict-space\ A\ A')$   $v \in sets\ M$   
  **by** *auto*

**have**  $x = (A' \times space\ M) \cap x$

**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**  $u \in sets\ A$  **using** *uv-def(2)* *assms* **unfolding** *sets-restrict-space* **by** *blast*

**hence**  $x \in \{a \times b \mid a \in \text{sets } A \wedge b \in \text{sets } M\}$   
**unfolding** *wv-def(1)* **using** *wv-def(3)* **by** *auto*  
**ultimately show**  $x \in ?X$   
**by** *simp*  
**qed**  
**ultimately have**  $?2: ?X = \{a \times b \mid a \in \text{sets } (\text{restrict-space } A \ A') \wedge b \in \text{sets } M\}$  **by** *simp*

**have** *sets ?R = sigma-sets (A' × space M) ((∩) (A' × space M) ‘ {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**  $\dots = \text{sets } (\text{restrict-space } A \ A' \otimes_M M)$   
**unfolding** *sets-pair-measure space-restrict-space Int-absorb2[OF 0]* *sets-restrict-space 2*  
**by** *auto*  
**finally have**  $?3: \text{sets } (\text{restrict-space } (A \otimes_M M) (A' \times \text{space } M)) = \text{sets } (\text{restrict-space } A \ A' \otimes_M M)$   
**by** *simp*

**have**  $?4: \text{emeasure } (\text{restrict-space } A \ A' \otimes_M M) S = \text{emeasure } (\text{restrict-space } (A \otimes_M M) (A' \times \text{space } M)) S$   
**(is ?L1 = ?R1) if**  $?5: S \in \text{sets } (\text{restrict-space } A \ A' \otimes_M M)$  **for**  $S$   
**proof** –  
**have** *Pair x - ‘ S = {}* **if**  $x \notin A'$   $x \in \text{space } A$  **for**  $x$   
**using** *that 5 by (auto simp add: ?3[symmetric] sets-restrict-space)*  
**hence**  $?5: \text{emeasure } M (\text{Pair } x \ - \ ' S) = 0$  **if**  $x \notin A'$   $x \in \text{space } A$  **for**  $x$   
**using** *that by auto*  
**have**  $?L1 = (\int^+ x. \text{emeasure } M (\text{Pair } x \ - \ ' S) \partial \text{restrict-space } A \ A')$   
**by** *(intro emeasure-pair-measure-alt[OF that])*  
**also have**  $\dots = (\int^+ x \in A'. \text{emeasure } M (\text{Pair } x \ - \ ' S) \partial A)$   
**using** *assms by (intro nn-integral-restrict-space) auto*  
**also have**  $\dots = (\int^+ x. \text{emeasure } M (\text{Pair } x \ - \ ' S) \partial A)$   
**using**  $?5$  **by** *(intro nn-integral-cong) force*  
**also have**  $\dots = \text{emeasure } (A \otimes_M M) S$   
**using** *that assms by (intro emeasure-pair-measure-alt[symmetric])*  
*(auto simp add: ?3[symmetric] sets-restrict-space)*  
**also have**  $\dots = ?R1$   
**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*:  
 $\text{to-stream } (\text{comb-seq } n \ x \ y) = \text{stake } n \ (\text{to-stream } x) @- \text{to-stream } y$   
**unfolding** *comb-seq-def to-stream-def*  
**by** *(intro stream-eq-iff) simp*

**lemma** *to-stream-snth*:  $\text{to-stream } (!! \ x) = x$   
**by** *(intro ext stream-eq-iff) (simp add: to-stream-def)*

**lemma** *snth-to-stream*:  $\text{snth } (\text{to-stream } x) = x$   
**by** *(intro ext) (simp add: to-stream-def)*

**lemma** *(in prob-space) branch-stream-space*:  
 $(\lambda(x, y). \text{stake } n \ x @- y) \in \text{stream-space } M \otimes_M \text{stream-space } M \rightarrow_M \text{stream-space } M$



$distr (stream\text{-}space\ M \otimes_M stream\text{-}space\ M) (stream\text{-}space\ M) (\lambda(x,y). stake\ n\ x @- y)$   
 $= stream\text{-}space\ M (is\ ?L = ?R)$

**proof** –

**let**  $?T = stream\text{-}space\ M$   
**let**  $?S = PiM\ UNIV\ (\lambda\cdot.\ M)$

**interpret**  $S$ : *sequence-space*  $M$   
**by** *standard*

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

**have**  $?L = distr (distr\ ?S\ ?T\ to\text{-}stream \otimes_M distr\ ?S\ ?T\ to\text{-}stream) ?T (\lambda(x,y). stake\ n\ x @- y)$   
**by** (*subst* (1 2) *stream-space-eq-distr*) *simp*  
**also have**  $\dots = distr (distr\ (?S \otimes_M ?S) (?T \otimes_M ?T) (\lambda(x, y). (to\text{-}stream\ x, to\text{-}stream\ y)))$   
 $?T (\lambda(x, y). stake\ n\ x @- y)$   
**using** *prob-space-imp-sigma-finite*[*OF prob-space-stream-space*]  
**by** (*intro arg-cong2*[**where**  $f=(\lambda x y. distr\ x\ ?T\ y)$ ] *pair-measure-distr*)  
(*simp-all flip:stream-space-eq-distr*)  
**also have**  $\dots = distr (?S \otimes_M ?S) ?T ((\lambda(x, y). stake\ n\ x @- y) \circ (\lambda(x, y). (to\text{-}stream\ x, to\text{-}stream\ y)))$   
**by** (*intro distr-distr 0*) (*simp add: measurable-pair-iff*)  
**also have**  $\dots = distr (?S \otimes_M ?S) ?T ((\lambda(x, y). stake\ n\ (to\text{-}stream\ x) @- to\text{-}stream\ y))$   
**by** (*simp add:comp-def case-prod-beta'*)  
**also have**  $\dots = distr (?S \otimes_M ?S) ?T (to\text{-}stream \circ (\lambda(x, y). comb\text{-}seq\ n\ x\ y))$   
**using** *to-stream-comb-seq-eq*[*symmetric*]  
**by** (*intro arg-cong2*[**where**  $f=(\lambda x y. distr\ x\ ?T\ y)$ ] *ext*) *auto*  
**also have**  $\dots = distr (distr\ (?S \otimes_M ?S) ?S (\lambda(x, y). comb\text{-}seq\ n\ x\ y)) ?T\ to\text{-}stream$   
**by** (*intro distr-distr*[*symmetric*] *measurable-comb-seq*) *simp*  
**also have**  $\dots = distr\ ?S\ ?T\ to\text{-}stream$   
**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**  $R=(\lambda x y. x = to\text{-}coins\ (from\text{-}coins\ y))$ ]) *simp-all*

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

**lemma** *bij-to-coins*: *bij to-coins*  
**by** (*intro bij-betwI*[**where**  $g=from\text{-}coins$ ] *to-from-coins from-to-coins*) *auto*

**lemma** *bij-from-coins*: *bij from-coins*  
**by** (*intro* *bij-betuI*[**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* = *to-coins* (*sdrop* *n* (*from-coins* *x*))

**definition** *rel-coins* **where** *rel-coins* *x y* = (*to-coins* *x* = *y*)

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

**definition** *cconst* **where** *cconst* *x* = *to-coins* (*sconst* *x*)

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

**lemma** *bi-unique-rel-coins* [*transfer-rule*]: *bi-unique rel-coins*  
**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*]: ((=)  $\implies$  *rel-coins*  $\implies$  *rel-coins*) *shift* *cshift*  
**unfolding** *rel-coins-def* *cshift-def* *rel-fun-def* **by** (*auto* *simp:from-to-coins*)

**lemma** *ctake-transfer* [*transfer-rule*]: ((=)  $\implies$  *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*]: ((=)  $\implies$  *rel-coins*  $\implies$  *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*  $\implies$  *rel-coins*) *stl* *ctl*  
**unfolding** *rel-coins-def* *rel-fun-def* **by** (*auto* *simp:from-to-coins*)

**lemma** *cconst-transfer* [*transfer-rule*]: ((=)  $\implies$  *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*:  $\text{cshift } (\text{ctake } n \ s) \ (\text{cdrop } n \ s) = s$   
**by transfer** (rule *stake-sdrop*)

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

**lemma** *cshift-empty[simp]*:  $\text{cshift } [] \ xs = xs$   
**by transfer** *simp*

**lemma** *ctake-null[simp]*:  $\text{ctake } 0 \ xs = []$   
**by transfer** *simp*

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

**lemma** *cdrop-null[simp]*:  $\text{cdrop } 0 \ s = s$   
**by transfer** *simp*

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

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

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

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

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

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

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

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

**proof** –  
**have**  $?L \longleftrightarrow \text{ctake } (\text{length } x) \ y = x$   
**unfolding** *cprefix-def* **by** *simp*  
**also have**  $\dots \longleftrightarrow (\forall i < \text{length } x. (\text{ctake } (\text{length } x) \ y) \ ! \ i = x \ ! \ i)$   
**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**  $\bigwedge cs. \text{cshift } h \ cs = cs$   
**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 (¬ hh)))
    by simp
  also have ... = ctake 1 (cconst (¬ hh))
    using assms unfolding Cons by simp
  also have ... = [¬ hh] by simp
  finally show ?thesis by simp
qed

```

Stream version of *prefix-length-prefix*:

```

lemma cprefix-length-prefix:
  assumes length x ≤ 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 ¬(x || y)
  using assms cprefix-length-prefix
  by (cases length x ≤ length y) auto

```

```

lemma ctake-shift:
  ctake m (cshift xs ys) = (if m ≤ 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 ≤ length xs ⇒ ctake m (cshift xs ys) = take m xs
  and ctake-shift-big [simp]:
    m ≥ length xs ⇒ 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 ≤ 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*]:  
 $m \leq \text{length } xs \implies \text{cdrop } m \text{ (cshift } xs \text{ } ys) = \text{cshift (drop } m \text{ } xs) \text{ } ys$   
**and** *cdrop-shift-big* [*simp*]:  
 $m \geq \text{length } xs \implies \text{cdrop } m \text{ (cshift } xs \text{ } ys) = \text{cdrop (} m - \text{length } xs) \text{ } ys$   
**by** (*subst cdrop-shift; simp*)<sup>+</sup>

Infrastructure for building coin streams:

**primcorec** *cmap-iterate* :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  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**  $R = (\lambda xs \text{ } ys. (\exists x. xs = \text{cmap-iterate } m \text{ } f \text{ } x \wedge ys = \text{to-coins (smap } m \text{ (siterate } f \text{ } x))))$ ])

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

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

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

**next**

**fix** xs ys

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

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

**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**

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

**by** *auto*

**qed**

**definition** *build-coin-gen* :: ('a  $\Rightarrow$  bool list)  $\Rightarrow$  ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  coin-stream

**where**

*build-coin-gen* m f s = *cmap-iterate* (hd  $\circ$  fst)

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

**lemma** *build-coin-gen-aux*:

**fixes** f :: 'a  $\Rightarrow$  'b stream

**assumes**  $\bigwedge x. (\exists n \text{ } y. n \neq [] \wedge f \text{ } x = n @- f \text{ } y \wedge g \text{ } x = n @- g \text{ } y)$

**shows** f x = g x

**proof** (*rule stream.coinduct[where R=( $\lambda xs \text{ } ys. (\exists x \text{ } n. xs = n @- (f \text{ } x) \wedge ys = n @- (g \text{ } x))$ )*])

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

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

**next**

**fix** xs ys :: 'b stream

**assume**  $\exists x \text{ } n. xs = n @- (f \text{ } x) \wedge ys = n @- (g \text{ } x)$

**hence**  $\exists x \text{ } n. n \neq [] \wedge xs = n @- (f \text{ } x) \wedge ys = n @- (g \text{ } x)$

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

**then obtain** x n **where**  $0:xs = n @- (f \text{ } x) \wedge ys = n @- (g \text{ } x) \wedge n \neq []$

**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**  $shd\ xs = shd\ ys \wedge (\exists x\ n.\ stl\ xs = n@-\ (f\ x) \wedge stl\ ys = n@-\ (g\ x))$   
**by** *auto*  
**qed**

**lemma** *build-coin-gen*:

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

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

**proof**  $-$

**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-space* = *embed-measure* (*stream-space* (*measure-pmf* (*pmf-of-set* *UNIV*))) *to-coins*

**bundle** *coin-space-notation*

**begin**

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

**end**

**bundle** *no-coin-space-notation*

**begin**

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

**end**

**unbundle** *coin-space-notation*

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

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

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

**lemma** *from-coins-measurable*:  $\text{from-coins} \in \mathcal{B} \rightarrow_M (\text{stream-space } (\text{pmf-of-set } \text{UNIV}))$   
**unfolding** *coin-space-def* **by** (*intro measurable-embed-measure1*) (*simp add:from-to-coins*)

**lemma** *to-coins-measurable*:  $\text{to-coins} \in (\text{stream-space } (\text{pmf-of-set } \text{UNIV})) \rightarrow_M \mathcal{B}$   
**unfolding** *coin-space-def*  
**by** (*intro measurable-embed-measure2 bij-is-inj*[*OF bij-to-coins*])

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

**proof** –

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

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

**proof** –

**let**  $?S = \text{stream-space } (\text{pmf-of-set } \text{UNIV})$   
**have**  $?R = \text{distr } (\text{distr } ?S \ \mathcal{B} \ \text{to-coins}) \ ?S \ \text{from-coins}$   
**using** *B-t-eq-distr* **by** *simp*  
**also have**  $\dots = \text{distr } ?S \ ?S \ (\text{from-coins} \circ \text{to-coins})$   
**by** (*intro distr-distr to-coins-measurable from-coins-measurable*)  
**also have**  $\dots = \text{distr } ?S \ ?S \ \text{id}$   
**unfolding** *id-def comp-def from-to-coins* **by** *simp*  
**also have**  $\dots = ?L$   
**unfolding** *id-def* **by** *simp*  
**finally show** *?thesis*  
**by** *simp*

**qed**

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

**proof** –

**let**  $?S = \text{stream-space } (\text{pmf-of-set } (\text{UNIV}::\text{bool set}))$   
**have**  $1 = \text{emeasure } ?S (\text{space } ?S)$   
**by** (*intro prob-space.emeasure-space-1*[*symmetric*] *prob-space.prob-space-stream-space prob-space-measure-pmf*)  
**also have**  $\dots = \text{emeasure } \mathcal{B} (\text{from-coins } -' (\text{space } (\text{stream-space } (\text{pmf-of-set } \text{UNIV}))) \cap \text{space } \mathcal{B})$   
**by** (*subst B-eq-distr*) (*intro emeasure-distr from-coins-measurable sets.top*)  
**also have**  $\dots = \text{emeasure } \mathcal{B} (\text{space } \mathcal{B})$   
**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*:  $\text{distr } \mathcal{B} \ \mathcal{D} \ \text{chd} = \text{pmf-of-set } \text{UNIV}$  **(is ?L = ?R)**

**proof** –

**have**  $?L = \text{distr } (\text{stream-space } (\text{measure-pmf } (\text{pmf-of-set } \text{UNIV}))) \ \mathcal{D} \ (\text{chd} \circ \text{to-coins})$   
**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-coins*  $\circ$  *sdrop*  $x$   $\circ$  *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** *cdrop-def* **by** (*simp add:comp-def*)  
**qed**

**lemma** *ctake-measurable*:  $ctake\ k \in \mathcal{B} \rightarrow_M \mathcal{D}$   
**proof** –  
**have** *stake*  $k$   $\circ$  *from-coins*  $\in \mathcal{B} \rightarrow_M \mathcal{D}$   
**by** (*intro measurable-comp[OF from-coins-measurable]*) *simp*  
**thus** *?thesis*  
**unfolding** *ctake-def* **by** (*simp add: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**  $(\lambda(x, y). cshift (ctake\ n\ x)\ y) = to-coins \circ (?f \circ ?g)$   
**by** (*simp add:comp-def cshift-def ctake-def case-prod-beta'*)  
**also have** ...  $\in \mathcal{B} \otimes_M \mathcal{B} \rightarrow_M \mathcal{B}$   
**by** (*intro measurable-comp[OF - to-coins-measurable] measurable-comp[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**  $distr\ (\mathcal{B} \otimes_M \mathcal{B})\ (?M \otimes_M ?M)\ ?g = (distr\ \mathcal{B}\ ?M\ from-coins \otimes_M distr\ \mathcal{B}\ ?M\ from-coins)$   
**unfolding** *map-prod-def* **using** *prob-space-measure-pmf*  
**by** (*intro pair-measure-distr[symmetric] from-coins-measurable*) (*auto intro!*  
*prob-space-imp-sigma-finite prob-space.prob-space-stream-space simp:B-eg-distr[symmetric]*)  
**also have** ... =  $?M \otimes_M ?M$   
**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  $?L = \text{distr } (\mathcal{B} \otimes_M \mathcal{B}) \mathcal{B} (to\text{-coins} \circ ?f \circ ?g)$   
   **unfolding** *cshift-def ctake-def* **by** (*simp add:comp-def map-prod-def case-prod-beta*)  
 also have  $\dots = \text{distr } (\text{distr } (\mathcal{B} \otimes_M \mathcal{B}) (?M \otimes_M ?M) ?g) \mathcal{B} (to\text{-coins} \circ ?f)$   
   **by** (*intro distr-distr[symmetric] map-prod-measurable from-coins-measurable*  
   *measurable-comp[OF - to-coins-measurable] prob-space-measure-pmf*) *simp*  
 also have  $\dots = \text{distr } (?M \otimes_M ?M) \mathcal{B} (to\text{-coins} \circ ?f)$   
   **unfolding** 0 **by** *simp*  
 also have  $\dots = \text{distr } (\text{distr } (?M \otimes_M ?M) ?M ?f) \mathcal{B} to\text{-coins}$   
   **by** (*intro distr-distr[symmetric] to-coins-measurable*) *simp*  
 also have  $\dots = \text{distr } ?M \mathcal{B} to\text{-coins}$   
   **by** (*subst prob-space.branch-stream-space(2)*) (*auto intro:prob-space-measure-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**  $\forall S \in K.$  *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* =  $(\lambda x. \text{Collect } (\text{cprefix } x)) \text{ ' UNIV}$

**lemma** *image-collect-eq*:  $f \text{ ' } \{x. A (f x)\} = \{x. A x\} \cap \text{range } f$

**by** *auto*

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

**proof** –

**have** *bij-betw*  $(\lambda x. (!!)) (\text{smap } \text{discrete } x) \text{ UNIV UNIV}$

**by** (*intro* *bij-betwI* [**where**  $g = \text{smap } \text{of-discrete} \circ \text{to-stream}$ ]) (*simp-all* *add:to-stream-snth* *snth-to-stream* *stream.map-comp* *comp-def of-discrete-inverse* *discrete-inverse* *stream.map-ident*)

**hence**  $\exists \text{range } (\lambda x. (!!)) (\text{smap } \text{discrete } x) = \text{UNIV}$

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

**unfolding**  $2$  **by** *simp*

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

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

**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*  $U\text{-eq}$  *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*  $(\text{from-coins-t ' } U)$  *from-coins-t*  $x \in \text{from-coins-t ' } U$

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

**then obtain**  $B$  **where**  $B: B \in K$  *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 = PiE UNIV X and fin-X: finite {i. X i ≠ UNIV}
    using K-cylinder B(1) by auto
  define Z where Z i = (X i ≠ UNIV) for i
  define n where n = (if {i. X i ≠ UNIV} ≠ {} then Suc (Max {i. X i ≠ UNIV}) 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 = UNIV if i ≥ n for i
    using that leD unfolding Z-def by auto

  define R where R = {xs. cprefix (ctake n x) xs}
  have {discrete (ctake n x ! i)} ⊆ X i if i < n for i
  proof -
    have {discrete (ctake n x ! i)} = {discrete (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 ... ⊆ X i
      using B(2) unfolding X PiE-def Pi-def by auto
    finally show ?thesis
      by simp
  qed
  hence from-coins-t ' R ⊆ PiE UNIV X
    using X-univ unfolding R-def from-coins-cprefix
    by (intro PiE-mono) auto
  moreover have ... ⊆ from-coins-t ' U
    using B(3) X by simp
  ultimately have from-coins-t ' R ⊆ from-coins-t ' U
    by simp
  hence R ⊆ U
    using bij-is-inj[OF bij-from-coins-t]
    by (simp add: inj-image-eq-iff subset-image-iff)
  moreover have R ∈ coin-stream-basis x ∈ 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-steam-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 ∃(B :: coin-stream set set). countable B ∧ 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 ⇒ coin-stream ⇒ real
  where dist-coin-stream x y = dist (from-coins-t x) (from-coins-t y)

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

```

```

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

lemma in-from-coins-iff:  $x \in \text{from-coins-}t \text{ ' } U \iff (\text{to-coins-}t \ x \in U)$ 
  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  $U = (\forall x \in U. \forall_F (x', y) \text{ in } \text{uniformity}. x' = x \implies y \in U)$  for  $U :: \text{coin-stream set}$ 
  proof -
    have open  $U \iff \text{open} (\text{from-coins-}t \text{ ' } U)$ 
      unfolding open-coin-stream-def by simp
    also have ...  $\iff (\forall x \in U. \exists e > 0. \forall y. \text{dist} (\text{from-coins-}t \ x) \ y < e \implies y \in \text{from-coins-}t \text{ ' } U)$ 
      unfolding fun-open-ball-aux by auto
    also have ...  $\iff (\forall x \in U. \exists e > 0. \forall y \in \text{to-coins-}t \text{ ' } UNIV. \text{dist} \ x \ y < e \implies y \in U)$ 
      unfolding dist-coin-stream-def by (intro ball-cong refl ex-cong)
      (simp add: from-to-coins-t in-from-coins-iff)
    also have ...  $\iff (\forall x \in U. \exists e > 0. \forall y. \text{dist} \ x \ y < e \implies y \in U)$ 
      using bij-is-surj[OF bij-to-coins-t] by simp
    finally have open  $U = (\forall x \in U. \exists e > 0. \forall y. \text{dist} \ x \ y < e \implies y \in U)$ 
      by simp
    thus ?thesis
      unfolding eventually-uniformity-metric by simp
  qed
  show  $(\text{dist} \ x \ y = 0) = (x = y)$  for  $x \ y :: \text{coin-stream}$ 
    unfolding dist-coin-stream-def by (metis dist-eq-0-iff to-from-coins-t)
  show  $\text{dist} \ x \ y \leq \text{dist} \ x \ z + \text{dist} \ y \ z$  for  $x \ y \ z :: \text{coin-stream}$ 
    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  $X :: \text{nat} \Rightarrow \text{coin-stream}$ 
  proof -
    have Cauchy  $(\text{from-coins-}t \circ X)$ 
      using uniformly-continuous-imp-Cauchy-continuous[unfolded Cauchy-continuous-on-def]
      from-coins-t-u-continuous that by auto
    hence convergent  $(\text{from-coins-}t \circ X)$ 
      by (rule Cauchy-convergent)
    then obtain  $x$  where  $(\text{from-coins-}t \circ X) \longrightarrow x$ 
      unfolding convergent-def by auto
    moreover have isCont to-coins-t  $x$ 
      using to-coins-t-continuous continuous-on-eq-continuous-within by blast
  qed

```

**ultimately have**  $(to\text{-}coins\text{-}t \circ from\text{-}coins\text{-}t \circ X) \longrightarrow to\text{-}coins\text{-}t\ x$   
**using** *isCont-tendsto-compose* **by**  $(auto\ simp\ add:comp\text{-}def)$   
**thus convergent**  $X$   
**unfolding** *convergent-def comp-def to-from-coins-t* **by** *auto*  
**qed**  
**qed**

**lemma** *at-least-borelI*:

**assumes** *topological-basis*  $K$   
**assumes** *countable*  $K$   
**assumes**  $K \subseteq sets\ M$   
**assumes** *open*  $U$   
**shows**  $U \in sets\ M$

**proof** –

**obtain**  $K'$  **where**  $K'\text{-range}: K' \subseteq K$  **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**  $K'\text{-range}$  *assms(2,3)* *countable-subset*  
**by**  $(intro\ sets.countable\text{-}Union)$  *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**  $\{xs.\ P\ (f\ xs)\} = f\ \text{--}'\ Collect\ P \cap space\ \mathcal{B}$   
**unfolding** *vimage-def space-coin-space* **by** *simp*  
**also have**  $\dots \in sets\ \mathcal{B}$   
**by**  $(intro\ measurable\text{-}sets[OF\ assms(1,2)])$   
**finally show** *?thesis* **by** *simp*

**qed**

**lemma** *coin-space-is-borel-measure*:

**assumes** *open*  $U$   
**shows**  $U \in sets\ \mathcal{B}$

**proof** –

**have**  $0:countable\ coin\text{-}stream\text{-}basis$   
**unfolding** *coin-stream-basis-def* **by** *simp*

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

**have**  $\{xs.\ cprefix\ x\ xs\} \in sets\ \mathcal{B}$  **for**  $x$

**proof**  $(cases\ x \neq [])$

**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**  $(intro\ sets.countable\text{-}INT\ image\text{-}subsetI)\ auto$

**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**  $\dots \in \text{sets } \mathcal{B}$   
**by** *simp*  
**finally show** *?thesis* **by** *simp*  
**qed**  
**hence**  $1:\text{coin-stream-basis} \subseteq \text{sets } \mathcal{B}$   
**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* ( $\lambda S. S = \text{UNIV} \vee \text{None} \notin S$ )

**lemma** *option-ud-topology*: *istopology* ( $\lambda S. S = \text{UNIV} \vee \text{None} \notin S$ ) (**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*  $S \longleftrightarrow (S = \text{UNIV} \vee \text{None} \notin S)$   
**unfolding** *option-ud-def* **by** (*subst topology-inverse'[OF option-ud-topology]*) *auto*

**lemma** *topspace-option-ud*: *topspace* *option-ud* = *UNIV*

**proof** –

**have**  $\text{UNIV} \subseteq \text{topspace } \text{option-ud}$  **by** (*intro openin-subset*) (*simp add:openin-option-ud*)  
**thus** *?thesis* **by** *auto*

**qed**

**lemma** *contionuos-into-option-udI*:

**assumes**  $\bigwedge x. \text{openin } X (f - \{ \text{Some } x \} \cap \text{topspace } X)$   
**shows** *continuous-map*  $X$  *option-ud*  $f$

**proof** –

**have** *openin*  $X \{x \in \text{topspace } X. f x \in U\}$  **if** *openin* *option-ud*  $U$  **for**  $U$

**proof** (*cases*  $U = \text{UNIV}$ )

**case** *True*

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

**next**

**case** *False*

**define**  $V$  **where**  $V = \text{the } 'U$

**have**  $\text{None} \notin U$

**using** *that False* **unfolding** *openin-option-ud* **by** *simp*

**hence**  $\text{Some } 'V = \text{id } 'U$

**unfolding** *V-def image-image id-def*

**by** (*intro image-cong refl*) (*metis option.exhaust-sel*)

**hence**  $U = \text{Some } 'V$  **by** *simp*

**hence**  $\{x \in \text{topspace } X. f x \in U\} = (\bigcup v \in V. f - \{ \text{Some } v \} \cap \text{topspace } X)$  **by** *auto*

**moreover have** *openin*  $X (\bigcup v \in V. f - \{ \text{Some } v \} \cap \text{topspace } X)$

**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**  $P$  {*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 ⇒ 'a *random-alg-int*  
**where** *return-rai* x bs = *Some* (x, bs)

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

**definition** *bind-rai* :: 'a *random-alg-int* ⇒ ('a ⇒ 'b *random-alg-int*) ⇒ 'b *random-alg-int*  
**where** *bind-rai* m f bs =  
do {  
(r, bs') ← m bs;  
f r bs'  
}

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

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

**definition** *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]<sup>2</sup>. 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

---

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



**unfolding** *bind-rai-def return-rai-def* **by** *simp*

**lemma** *bind-rai-assoc*:  $\text{bind-rai } (\text{bind-rai } f \ g) \ h = \text{bind-rai } f \ (\lambda x. \text{bind-rai } (g \ x) \ h)$   
**unfolding** *bind-rai-def* **by** (*simp add:case-prod-beta'*)

**lemma** *bind-return-rai*:  $\text{bind-rai } m \ \text{return-rai} = m$   
**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* =  $(\forall \text{cs}. f \ (\text{cshift } p \ \text{cs}) = \text{Some } (r, \text{cs}))$

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

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

**lemma** *in-range-rmI*:  
  **assumes**  $r \ \text{bs} = \text{Some } (y, n)$   
  **shows**  $y \in \text{range-rm } r$   
**proof** -  
  **have**  $\text{Some } (y, n) \in \text{range } r$   
  **using** *assms[symmetric]* **by** *auto*  
  **thus** *?thesis*  
  **unfolding** *range-rm-def* **using** *fun.set-map* **by** *force*  
**qed**

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

**lemma** *wf-randomI*:  
  **assumes**  $\bigwedge \text{bs}. f \ \text{bs} \neq \text{None} \implies (\exists p \ r. \text{cprefix } p \ \text{bs} \wedge \text{wf-on-prefix } f \ p \ r)$   
  **shows** *wf-random* *f*  
**proof** -  
  **have**  $\exists p. \text{cprefix } p \ \text{bs} \wedge \text{wf-on-prefix } f \ p \ r$  **if**  $0: f \ \text{bs} = \text{Some } (r, \text{bs}')$  **for** *bs r bs'*  
  **proof** -  
    **obtain** *p r'* **where**  $1: \text{cprefix } p \ \text{bs}$  **and**  $2: \text{wf-on-prefix } f \ p \ r'$   
    **using** *assms 0* **by** *force*  
    **have**  $f \ \text{bs} = f \ (\text{cshift } p \ (\text{cdrop } (\text{length } p) \ \text{bs}))$   
    **using**  $1$  **unfolding** *cprefix-def* **by** (*metis* *ctake-cdrop*)  
    **also have**  $\dots = \text{Some } (r', \text{cdrop } (\text{length } p) \ \text{bs})$   
    **using**  $2$  **unfolding** *wf-on-prefix-def* **by** *auto*  
    **finally have**  $f \ \text{bs} = \text{Some } (r', \text{cdrop } (\text{length } p) \ \text{bs})$   
    **by** *simp*  
    **hence**  $r = r'$  **using**  $0$  **by** *simp*  
    **thus** *?thesis* **using**  $1 \ 2$  **by** *auto*  
  **qed**  
  **thus** *?thesis*  
  **unfolding** *wf-random-def* **by** (*auto split:option.split*)  
**qed**

**lemma** *wf-on-prefix-bindI*:  
  **assumes** *wf-on-prefix* *m p r*  
  **assumes** *wf-on-prefix* (*f r*) *q s*  
  **shows** *wf-on-prefix* ( $m \gg f$ ) ( $p @ q$ ) *s*

**proof** –  
**have**  $(m \ggg f) (cshift (p@q) cs) = Some (s, cs)$  **for**  $cs$   
**proof** –  
**have**  $(m \ggg f) (cshift (p@q) cs) = (m \ggg f) (cshift p (cshift q cs))$   
**by** *simp*  
**also have**  $\dots = (f r) (cshift q cs)$   
**using** *assms unfolding wf-on-prefix-def bind-rai-def* **by** *simp*  
**also have**  $\dots = Some (s, cs)$   
**using** *assms unfolding wf-on-prefix-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**  $\bigwedge x. x \in range\text{-}rm\ m \implies wf\text{-}random (f\ x)$   
**shows** *wf-random (m >>> f)*

**proof** (*rule wf-randomI*)

**fix**  $bs$   
**assume**  $(m \ggg f) bs \neq None$   
**then obtain**  $x\ bs'\ y\ bs''$  **where**  $1: m\ bs = Some (x, bs')$  **and**  $2: f\ x\ bs' = Some (y, bs'')$   
**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**  $5: wf\text{-}on\text{-}prefix\ m\ p\ x$  **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**  $m\ bs = Some (x, cdrop (length\ p) bs)$   
**using**  $5$  **unfolding** *wf-on-prefix-def* **by** *metis*  
**hence**  $bs' = cdrop (length\ p) bs$   
**using**  $1$  **by** *auto*  
**hence**  $6: bs = cshift\ p\ bs'$   
**using**  $4$  **by** *auto*

**obtain**  $q$  **where**  $7: wf\text{-}on\text{-}prefix (f\ x) q\ y$  **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 >>> f) (p@q) y*  
**by** (*intro wf-on-prefix-bindI[OF 5] 7*)  
**ultimately show**  $\exists p\ r. cprefix\ p\ bs \wedge wf\text{-}on\text{-}prefix (m \ggg f) p\ r$   
**by** *auto*

**qed**

**lemma** *wf-return*:

*wf-random (return-rai x)*

**proof** (*rule wf-randomI*)

**fix**  $bs$  **assume** *return-rai x bs <math>\neq None</math>*  
**have** *wf-on-prefix (return-rai x) [] x*  
**unfolding** *wf-on-prefix-def return-rai-def* **by** *auto*  
**moreover have** *cprefix [] bs*  
**unfolding** *cprefix-def* **by** *auto*  
**ultimately show**  $\exists p\ r. cprefix\ p\ bs \wedge wf\text{-}on\text{-}prefix (return\text{-}rai\ x) p\ r$

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

by auto

qed

definition *ptree-rm* :: 'a *random-alg-int*  $\Rightarrow$  *bool list set*

where *ptree-rm* *f* = {*p*.  $\exists r. \text{wf-on-prefix } f \text{ } p \text{ } r$ }

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**  $x = \text{cshift } h \ x$   
**by simp**  
**qed**  
**hence**  $h = []$   
**using** *empty-if-shift-idem* **by simp**  
**thus ?thesis using 0 by simp**  
**qed**

**lemma** *prefixes-parallel*:  
**assumes**  $p \in \text{ptree-rm } f$   
**assumes**  $q \in \text{ptree-rm } f$   
**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*  $\{p \in \text{ptree-rm } f. \text{cprefix } p \text{ } bs\}$   
**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**  $f \text{ } bs = \text{map-option } (\lambda p. (\text{eval-rm } f \ p, \text{cdrop } (\text{length } p) \text{ } bs)) (\text{consumed-prefix } f \text{ } bs)$

**proof** (*cases*  $f \text{ } bs$ )  
**case** *None*  
**have** *False* **if**  $p$ -*in*:  $p \in \text{ptree-rm } f$  **and**  $p$ -*pref*:  $\text{cprefix } p \text{ } bs$  **for**  $p$   
**proof** –  
**obtain**  $r$  **where** *wf*: *wf-on-prefix*  $f \ p \ r$  **using** *that*  $p$ -*in* **unfolding** *ptree-rm-def* **by auto**  
**have**  $bs = \text{cshift } p \ (\text{cdrop } (\text{length } p) \text{ } bs)$   
**using**  $p$ -*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**  $\exists p. \text{cprefix } p \text{ } bs \wedge \text{wf-on-prefix } f \ p \ r$   
**using** *assms(1)* **unfolding** *wf-random-def* **by** (*auto* *split:option.split-asm*)

**then obtain**  $p$  **where**  $sp: cprefix\ p\ bs$  **and**  $wf: wf\text{-on-prefix}\ f\ p\ r$   
**by** *auto*  
**hence**  $p \in \{p \in ptree\text{-rm}\ f.\ cprefix\ p\ bs\}$   
**unfolding** *ptree-rm-def* **by** *auto*  
**hence**  $0:\{p \in ptree\text{-rm}\ f.\ cprefix\ p\ bs\} = \{p\}$   
**using** *prefixes-singleton* **by** *auto*  
**show** *?thesis* **unfolding**  $0\ wf\text{-on-prefix}D[OF\ wf\ sp]$  *consumed-prefix-def* **by** *simp*  
**qed**

**lemma** *range-rm-alt*:

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

**proof** –

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

**qed**

**lemma** *consumed-prefix-some-iff*:

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

**proof** –

**have**  $p \in ptree\text{-rm}\ f \implies cprefix\ p\ bs \implies x \in ptree\text{-rm}\ f \implies cprefix\ x\ bs \implies x = p$  **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**

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

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

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

**lemma** *wf-random-alt2*:

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

**proof** –

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

**using**  $0$  **unfolding** *consumed-bits-def map-option.compositionality comp-def cprefix-def*  
**by** (*cases consumed-prefix f bs*) *auto*  
**finally show** *?thesis* **by** *simp*  
**qed**

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

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

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

**proof**

**assume**  $\text{consumed-bits } f\ bs = \text{Some } n$   
**then obtain**  $p$  **where**  $\text{the-elem-opt } \{p \in \text{ptree-rm } f. \text{cprefix } p\ bs\} = \text{Some } p$  **and**  $0: \text{length } p = n$   
**unfolding** *consumed-bits-def consumed-prefix-def* **by** (*auto split:option.split-asm*)  
**hence**  $p \in \text{ptree-rm } f\ \text{cprefix } p\ bs$   
**unfolding** *the-elem-opt-Some-iff[OF prefixes-at-most-one]* **by** *auto*  
**thus**  $\text{ctake } n\ bs \in \text{ptree-rm } f$   
**using**  $0$  **unfolding** *cprefix-def* **by** *auto*

**next**

**assume**  $\text{ctake } n\ bs \in \text{ptree-rm } f$   
**hence**  $\text{ctake } n\ bs \in \{p \in \text{ptree-rm } f. \text{cprefix } p\ bs\}$   
**unfolding** *cprefix-def* **by** *auto*  
**hence**  $\{p \in \text{ptree-rm } f. \text{cprefix } p\ bs\} = \{\text{ctake } n\ bs\}$   
**using** *prefixes-singleton* **by** *auto*  
**thus**  $\text{consumed-bits } f\ bs = \text{Some } n$   
**unfolding** *consumed-bits-def consumed-prefix-def* **by** *simp*

**qed**

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

**proof** –

**have**  $0: \text{consumed-bits } f \text{ -- } \{x\} \cap \text{space } \mathcal{B} \in \text{sets } \mathcal{B}$  (**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**  $?L = \{bs. \exists z. \text{consumed-prefix } f\ bs = \text{Some } z \wedge \text{length } z = n\}$   
**unfolding** *consumed-bits-def vimage-def space-coin-space x-def* **by** *simp*  
**also have**  $\dots = \{bs. \exists p. \{p \in \text{ptree-rm } f. \text{cprefix } p\ bs\} = \{p\} \wedge \text{length } p = n\}$   
**unfolding** *consumed-prefix-def x-def the-elem-opt-Some-iff[OF prefixes-at-most-one]* **by** *simp*  
**also have**  $\dots = \{bs. \exists p. \text{cprefix } p\ bs \wedge \text{length } p = n \wedge p \in \text{ptree-rm } f\}$   
**using** *prefixes-singleton* **by** (*intro Collect-cong ex-cong1*) *auto*  
**also have**  $\dots = \{bs. \text{ctake } n\ bs \in \text{ptree-rm } f\}$   
**unfolding** *cprefix-def* **by** (*intro Collect-cong*) (*metis length-ctake*)  
**also have**  $\dots \in \text{sets } \mathcal{B}$   
**by** (*intro measurable-sets-coin-space[OF ctake-measurable]*) *simp*  
**finally show** *?thesis*  
**by** *simp*

**qed**

**thus** *?thesis*  
**by** (*intro measurable-sigma-sets-with-exception*[**where**  $d=$ *None*])  
**qed**

**lemma** *R-sets*:

**assumes** *wf:wf-random f*  
**shows**  $\{bs. f\ bs = \text{None}\} \in \text{sets } \mathcal{B}$   $\{bs. f\ bs \neq \text{None}\} \in \text{sets } \mathcal{B}$   
**proof** –  
**show**  $0: \{bs. f\ bs = \text{None}\} \in \text{sets } \mathcal{B}$   
**unfolding** *consumed-bits-inf-iff*[*OF wf*]  
**by** (*intro measurable-sets-coin-space*[*OF consumed-bits-measurable*]) *simp*  
**have**  $\{bs. f\ bs \neq \text{None}\} = \text{space } \mathcal{B} - \{bs. f\ bs = \text{None}\}$   
**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**  $\{bs. f\ bs \neq \text{None}\} \in \text{sets } \mathcal{B}$   
**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**  $\text{range-rm } f \subseteq \text{eval-rm } f \text{ ‘ UNIV}$   
**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**  $x :: \text{bool list}$   
  
**have** *open ((consumed-prefix f) – ‘ {Some x}) (is open ?T)*  
**proof** (*cases x ∈ ptree-rm f*)  
**case** *True*  
**hence**  $0: ?T = \{bs. \text{cprefix } x\ bs\}$   
**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} ∩ 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 ∘ f)*

**proof** –

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

**show** *?thesis* **unfolding**  $0$

**by** ( $\text{intro continuous-map-compose[OF consumed-prefix-continuous] map-option-continuous}$ )  
**qed**

**lemma** *none-measure-subprob-algebra*:

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

**context**

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

**begin**

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

**proof** –

**define**  $h$  **where**  $h = \text{the} \circ \text{consumed-bits } f$   
**define**  $g$  **where**  $g bs = (\text{ctake } (h bs) bs, \text{cdrop } (h bs) bs)$  **for**  $bs$

**have**  $\text{consumed-bits } f bs \neq \text{None}$  **if**  $bs \in \text{space } R$  **for**  $bs$   
**using**  $\text{that consumed-bits-inf-iff[OF wf]}$  **unfolding**  $R\text{-def space-restrict-space space-coin-space}$   
**by** ( $\text{simp del:not-infinity-eq not-None-eq}$ )

**hence**  $0$ :  $\text{the } (f bs) = \text{map-prod (eval-rm } f) \text{ id } (g bs)$  **if**  $bs \in \text{space } R$  **for**  $bs$

**unfolding**  $g\text{-def } h\text{-def}$  **using**  $\text{that}$   
**by** ( $\text{subst wf-random-alt2[OF wf]}$ ) ( $\text{cases consumed-bits } f bs, \text{auto simp del: not-None-eq}$ )

**have**  $1$ :  $h \in R \rightarrow_M \mathcal{D}$

**unfolding**  $R\text{-def } h\text{-def}$   
**by** ( $\text{intro measurable-restrict-space1 measurable-comp[OF consumed-bits-measurable]}$ )  $\text{simp}$

**have**  $\text{ctake } k \in R \rightarrow_M \mathcal{D}$  **for**  $k$

**unfolding**  $R\text{-def}$  **by** ( $\text{intro measurable-restrict-space1 ctake-measurable}$ )

**moreover** **have**  $\text{cdrop } k \in R \rightarrow_M \mathcal{B}$  **for**  $k$

**unfolding**  $R\text{-def}$  **by** ( $\text{intro measurable-restrict-space1 cdrop-measurable}$ )

**ultimately** **have**  $g \in R \rightarrow_M \mathcal{D} \otimes_M \mathcal{B}$

**unfolding**  $g\text{-def}$

**by** ( $\text{intro measurable-Pair measurable-Pair-compose-split[OF - 1 measurable-id]}$ )  $\text{simp-all}$

**hence**  $(\text{map-prod (eval-rm } f) \text{ id} \circ g) \in R \rightarrow_M \mathcal{D} \otimes_M \mathcal{B}$

**by** ( $\text{intro measurable-comp[where } N=\mathcal{D} \otimes_M \mathcal{B}] \text{map-prod-measurable}$ )  $\text{auto}$

**moreover** **have**  $(\text{the} \circ f) \in R \rightarrow_M \mathcal{D} \otimes_M \mathcal{B} \longleftrightarrow (\text{map-prod (eval-rm } f) \text{ id} \circ g) \in R \rightarrow_M \mathcal{D} \otimes_M \mathcal{B}$

**using**  $0$  **by** ( $\text{intro measurable-cong}$ ) ( $\text{simp add:comp-def}$ )

**ultimately** **show** *?thesis*

**by**  $\text{auto}$

**qed**

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

**proof** –

**have**  $0$ :  $\text{countable } \{\{bs. f bs \neq \text{None}\}, \{bs. f bs = \text{None}\}\}$   
**by**  $\text{simp}$



**have** 1:  $\Omega \in \text{sets } \mathcal{B} \wedge \text{map-option } \text{fst} \circ f \in \text{restrict-space } \mathcal{B} \Omega \rightarrow_M \mathcal{D}$   
**if**  $\Omega \in \{\{bs. f \text{ bs} \neq \text{None}\}, \{bs. f \text{ bs} = \text{None}\}\}$  **for**  $\Omega$   
**proof** (cases  $\Omega = \{bs. f \text{ bs} \neq \text{None}\}$ )  
**case** *True*  
**have**  $\text{Some} \circ \text{fst} \circ (\text{the} \circ f) \in R \rightarrow_M \mathcal{D}$   
**by** (intro measurable-comp[OF the-f-measurable]) auto  
**hence**  $\text{map-option } \text{fst} \circ f \in R \rightarrow_M \mathcal{D}$   
**unfolding** *R-def* **by** (subst measurable-cong[where  $g = \text{Some} \circ \text{fst} \circ (\text{the} \circ f)$ ])  
(auto simp add: space-restrict-space space-coin-space)  
**thus**  $\Omega \in \text{sets } \mathcal{B} \wedge \text{map-option } \text{fst} \circ f \in \text{restrict-space } \mathcal{B} \Omega \rightarrow_M \mathcal{D}$   
**unfolding** *R-def True* **using** *R-sets[OF wf]* **by** auto  
**next**  
**case** *False*  
**hence**  $2:\Omega = \{bs. f \text{ bs} = \text{None}\}$   
**using** *that* **by** simp  
  
**have**  $\text{map-option } \text{fst} \circ f \in \text{restrict-space } \mathcal{B} \{bs. f \text{ bs} = \text{None}\} \rightarrow_M \mathcal{D}$   
**by** (subst measurable-cong[where  $g = \lambda-. \text{None}$ ])  
(simp-all add:space-restrict-space)  
  
**thus**  $\Omega \in \text{sets } \mathcal{B} \wedge \text{map-option } \text{fst} \circ f \in \text{restrict-space } \mathcal{B} \Omega \rightarrow_M \mathcal{D}$   
**unfolding** 2 **using** *R-sets[OF wf]* **by** auto  
**qed**

**have** 3:  $\text{space } \mathcal{B} \subseteq \bigcup \{\{bs. f \text{ bs} \neq \text{None}\}, \{bs. f \text{ bs} = \text{None}\}\}$   
**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*  $f \in \text{space } (\text{subprob-algebra } \mathcal{D})$   
**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*:  $\text{fst} \circ \text{the} \circ f \in R \rightarrow_M \mathcal{D}$   
**proof** –  
**have**  $\text{fst} \circ (\text{the} \circ f) \in R \rightarrow_M \mathcal{D}$   
**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**  $C k = consumed\text{-}bits\ f \text{ -- } \{Some\ k\}$  **for**  $k$

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

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

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

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

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

**have** 7:  $the\ (f\ (cshift\ (ctake\ k\ x)\ y)) = (fst\ (the\ (f\ x)),\ y)$  **if**  $x \in C k$  **for**  $x\ y\ k$

**proof** –

**have**  $cshift\ (ctake\ k\ x)\ y \in C k$   
**using**  $that\ 8$  **by**  $simp$

**hence**  $the\ (f\ (cshift\ (ctake\ k\ x)\ y)) = (eval\text{-}rm\ f\ (ctake\ k\ x),\ y)$   
**using**  $wf\text{-}random\text{-}alt2[OF\ wf]$  **unfolding**  $C\text{-}def$  **by**  $simp$

**also have**  $\dots = (fst\ (the\ (f\ x)),\ y)$   
**using**  $that\ wf\text{-}random\text{-}alt2[OF\ wf]$  **unfolding**  $C\text{-}def$  **by**  $simp$

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

**qed**

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

**have** 0:

$emeasure\ (distr\ R\ (\mathcal{D} \otimes_M \mathcal{B})\ (the \circ f))\ (A \times B) =$   
 $emeasure\ (distr\ R\ \mathcal{D}\ (fst \circ the \circ f))\ A * emeasure\ \mathcal{B}\ B$   
**(is ?L = ?R)** **if**  $A \in sets\ \mathcal{D}\ B \in sets\ \mathcal{B}$  **for**  $A\ B$

**proof** –

**have** 3:  $\{bs. fst\ (the\ (f\ bs)) \in A \wedge bs \in C k\} \in sets\ \mathcal{B}$  **(is ?L1 ∈ -)** **for**  $k$

**proof** –

**have**  $?L1 = (fst \circ the \circ f) \text{ -- } A \cap space\ (restrict\text{-}space\ R\ (C\ k))$

**using** 5 **unfolding**  $vimage\text{-}def\ space\text{-}restrict\text{-}space\ R\text{-}def\ space\text{-}coin\text{-}space$  **by**  $auto$

**also have**  $\dots \in sets\ (restrict\text{-}space\ R\ (C\ k))$

**by**  $(intro\ measurable\text{-}sets[OF\ \text{-}\ that(1)]\ measurable\text{-}restrict\text{-}space1\ fst\text{-}the\text{-}f\text{-}measurable)$

**also have**  $\dots = sets\ (restrict\text{-}space\ \mathcal{B}\ (C\ k))$

**using** 5 **unfolding**  $R\text{-}def\ sets\text{-}restrict\text{-}restrict\text{-}space\ space\text{-}restrict\text{-}space\ space\text{-}coin\text{-}space$

**by**  $(intro\ arg\text{-}cong2[\mathbf{where}\ f=restrict\text{-}space]\ arg\text{-}cong[\mathbf{where}\ f=sets]\ refl)\ auto$

**finally have**  $?L1 \in sets\ (restrict\text{-}space\ \mathcal{B}\ (C\ k))$

**by**  $simp$

**thus**  $?L1 \in sets\ \mathcal{B}$

**using** 6  $space\text{-}coin\text{-}space\ sets\text{-}restrict\text{-}space\text{-}iff[\mathbf{where}\ M=\mathcal{B}\ \mathbf{and}\ \Omega=C\ k]$  **by**  $auto$

**qed**

**have** 4:  $\{bs. the\ (f\ bs) \in A \times B \wedge bs \in C k\} \in sets\ \mathcal{B}$  **(is ?L1 ∈ -)** **for**  $k$

**proof** –

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

**have** ?L = emeasure R ((the  $\circ$  f) - ' (A  $\times$  B)  $\cap$  space R)  
**using** that the-f-measurable **by** (intro emeasure-distr) auto  
**also have** ... = emeasure R {x. the (f x)  $\in$  A  $\times$  B  $\wedge$  f x  $\neq$  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)  $\in$  A  $\times$  B  $\wedge$  ( $\exists$  k. x  $\in$  C k)}  
**unfolding** R-def 2 **using** 1 **by** (intro emeasure-restrict-space) auto  
**also have** ... = emeasure B ( $\bigcup$  k. {x. the (f x)  $\in$  A  $\times$  B  $\wedge$  x  $\in$  C k})  
**by** (intro arg-cong2[**where** f=emeasure]) auto  
**also have** ... = ( $\sum$  k. emeasure B {x. the (f x)  $\in$  A  $\times$  B  $\wedge$  x  $\in$  C k})  
**using** 4 C-disj  
**by** (intro suminf-emeasure[symmetric] subsetI) (auto simp:disjoint-family-on-def)  
**also have** ... = ( $\sum$  k. emeasure (distr (B  $\otimes_M$  B) B ( $\lambda(x,y). (cshift (ctake k x) y))$ )  
{x. the (f x)  $\in$  A  $\times$  B  $\wedge$  x  $\in$  C k})  
**by** (intro suminf-cong arg-cong2[**where** f=emeasure] branch-coin-space(2)[symmetric] refl)  
**also have** ... = ( $\sum$  k. emeasure (B  $\otimes_M$  B)  
{x. the (f (cshift (ctake k (fst x)) (snd x)))  $\in$  A  $\times$  B  $\wedge$  (cshift (ctake k (fst x)) (snd x))  $\in$  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** ... = ( $\sum$  k. emeasure (B  $\otimes_M$  B)  
{x. the (f (cshift (ctake k (fst x)) (snd x)))  $\in$  A  $\times$  B  $\wedge$  fst x  $\in$  C k})  
**using** 8 **by** (intro suminf-cong arg-cong2[**where** f=emeasure] refl Collect-cong) auto  
**also have** ... = ( $\sum$  k. emeasure (B  $\otimes_M$  B) ({x. fst (the (f x))  $\in$  A  $\wedge$  x  $\in$  C k}  $\times$  B))  
**using** 7 **by** (intro suminf-cong arg-cong2[**where** f=emeasure] refl)  
(auto simp add:mem-Times-iff set-eq-iff)  
**also have** ... = ( $\sum$  k. emeasure B {x. fst (the (f x))  $\in$  A  $\wedge$  x  $\in$  C k} \* emeasure B B)  
**using** 3 that(2)  
**by** (intro suminf-cong coin-space.emeasure-pair-measure-Times) auto  
**also have** ... = ( $\sum$  k. emeasure B {x. fst (the (f x))  $\in$  A  $\wedge$  x  $\in$  C k}) \* emeasure B B  
**by** simp  
**also have** ... = emeasure B ( $\bigcup$  k. {x. fst (the (f x))  $\in$  A  $\wedge$  x  $\in$  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))  $\in$  A  $\wedge$  ( $\exists$  k. x  $\in$  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))  $\in$  A  $\wedge$  f x  $\neq$  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  $\circ$  the  $\circ$  f) - ' A  $\cap$  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  $\mathcal{D}$  (fst  $\circ$  the  $\circ$  f))  
 by (intro finite-measure.finite-measure-distr fst-the-f-measurable)  
 hence 1:sigma-finite-measure (distr R  $\mathcal{D}$  (fst  $\circ$  the  $\circ$  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**  $\dots \in \text{sets } (?H \otimes_M \mathcal{B})$   
**using** *distr-rai-measurable[OF wf-f]*  
**by** (*intro measurable-sets[where A=D] measurable-pair-measure-countable1 countable-range wf-m*)  
*(simp-all add:comp-def)*  
**also have**  $\dots = \text{sets } (\text{restrict-space } \mathcal{D} (\text{range-rm } m) \otimes_M \mathcal{B})$   
**unfolding** *restrict-count-space inf-top-right* **by** *simp*  
**also have**  $\dots = \text{sets } (\text{restrict-space } (\mathcal{D} \otimes_M \mathcal{B}) (\text{range-rm } m \times \text{space coin-space}))$   
**by** (*subst coin-space.restrict-space-pair-lift*) *auto*  
**finally have**  $\{(v, bs). \text{map-option fst } (f v bs) \in A \wedge v \in \text{range-rm } m\} \in$   
 $\text{sets } (\text{restrict-space } (\mathcal{D} \otimes_M \mathcal{B}) (\text{range-rm } m \times \text{UNIV}))$   
**unfolding** *space-coin-space* **by** *simp*  
**moreover have**  $\text{range-rm } m \times \text{space coin-space} \in \text{sets } (\mathcal{D} \otimes_M \mathcal{B})$   
**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**  $3: \text{distr-rai } (f (\text{the } x)) \in \text{space } (\text{subprob-algebra } \mathcal{D})$   
**if**  $x \in \text{Some } \text{'range-rm } m \text{ for } x$   
**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**  $\dots = (\lambda v. \text{emeasure } (\text{if } v \in \text{Some } \text{'range-rm } m \text{ then } ?m (f (\text{the } v)) \text{ else } \text{null-measure } \mathcal{D}))$   
 $A)$   
 $\circ (\text{map-option fst } \circ m)$   
**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**  $\dots \in \text{borel-measurable coin-space}$   
**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**  $\text{emeasure } ?L A = \text{emeasure } \mathcal{B} ((\text{map-option fst } \circ (m \gg= f)) - 'A)$   
**unfolding** *distr-rai-def* **using**  $0$  **by** (*subst emeasure-distr*) (*simp-all add:space-coin-space*)  
**also have**  $\dots =$   
 $\text{emeasure } \mathcal{B} ((\text{map-option fst} \circ (m \gg= f)) - 'A \cap -N) + \text{emeasure } \mathcal{B} ((\text{map-option fst} \circ (m \gg= f)) - 'A$   
 $\cap N)$   
**using** *N-meas N-meas' 1*  
**by** (*subst emeasure-Un'[symmetric]*) (*simp-all add:Int-Un-distrib[symmetric]*)  
**also have**  $\dots =$   
 $\text{emeasure } \mathcal{B} ((\text{map-option fst} \circ (m \gg= f)) - 'A \cap -N) + \text{emeasure } ?R ((\text{map-option fst} \circ (m \gg= f)) - 'A \cap$   
 $N)$   
**using** *N-meas* **unfolding** *N-def*  
**by** (*intro arg-cong2[where f=(+)] refl emeasure-restrict-space[symmetric]*) *simp-all*  
**also have**  $\dots = ?N + \text{emeasure } ?R ((\text{the } \circ m) - \text{'$   
 $\{(v, bs). \text{map-option fst } (f v bs) \in A \wedge v \in \text{range-rm } m\} \cap \text{space } ?R)$   
**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 + emeasure (distr ?R ( $\mathcal{D} \otimes_M \mathcal{B}$ ) (the  $\circ$  m))  
 {(v,bs). map-option fst (f v bs)  $\in$  A  $\wedge$  v  $\in$  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 + emeasure (distr ?R  $\mathcal{D}$  (fst  $\circ$  the  $\circ$  m)  $\otimes_M$   $\mathcal{B}$ )  
 {(v,bs). map-option fst (f v bs)  $\in$  A  $\wedge$  v  $\in$  range-rm m}  
**unfolding** N-def remainder-indep[OF wf-m] **by** simp  
**also have** ... = ?N +  $\int^+$  v. emeasure  $\mathcal{B}$   
 {bs. map-option fst (f v bs)  $\in$  A  $\wedge$  v  $\in$  range-rm m}  $\partial$ distr ?R  $\mathcal{D}$  (fst  $\circ$  (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. emeasure  $\mathcal{B}$   
 {bs. map-option fst (f ((fst  $\circ$  (the  $\circ$  m)) x) bs)  $\in$  A  $\wedge$  (fst  $\circ$  (the  $\circ$  m)) x  $\in$  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** ... = ?N + ( $\int^+$  x  $\in$  {bs. m bs  $\neq$  None}. emeasure  $\mathcal{B}$   
 {bs. map-option fst (f (fst (the (m x))) bs)  $\in$  A  $\wedge$  fst (the (m x))  $\in$  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** ... = ?N + ( $\int^+$  x  $\in$  {bs. m bs  $\neq$  None}.  
 emeasure  $\mathcal{B}$  ((map-option fst  $\circ$  f (fst (the (m x)))) - ' A  $\cap$  space  $\mathcal{B}$ )  $\partial$  $\mathcal{B}$ )  
**by** (intro arg-cong2[**where** f=(+)] set-nn-integral-cong refl arg-cong2[**where** f=emeasure])  
 (auto intro:in-range-rmI simp:space-coin-space)  
**also have** ... = ?N + ( $\int^+$  x  $\in$  N. emeasure (distr-rai(f(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]  
 distr-rai-measurable[OF wf-f]) (auto intro:in-range-rmI)  
**also have** ... = ( $\int^+$  x. (indicator {bs. bs  $\notin$  N  $\wedge$  None  $\in$  A}) x  $\partial$  $\mathcal{B}$ ) +  
 ( $\int^+$  x  $\in$  N. emeasure (distr-rai(f(fst(the(m x)))))) A  $\partial$  $\mathcal{B}$ )  
**using** N-meas N-meas'  
**by** (intro arg-cong2[**where** f=(+)] nn-integral-indicator[symmetric] refl)  
 (cases None  $\in$  A; auto simp:Collect-neg-eq)  
**also have** ... =  $\int^+$  x. indicator {bs. bs  $\notin$  N  $\wedge$  None  $\in$  A} x +  
 emeasure (distr-rai (f (fst (the (m x)))))) A \* indicator N x  $\partial$  $\mathcal{B}$   
**using** N-meas' N-meas **by** (intro nn-integral-add[symmetric] 4) simp  
**also have** ... =  $\int^+$  x. indicator (-N) x \* indicator A None +  
 indicator N x \* emeasure (distr-rai (f (fst (the (m x)))))) A  $\partial$  $\mathcal{B}$   
**unfolding** N-def **by** (intro arg-cong2[**where** f=nn-integral] ext refl arg-cong2[**where** f=(+)]  
 (simp-all split:split-indicator))  
**also have** ... =  
 $\int^+$  x. emeasure (case m x of None  $\Rightarrow$  return  $\mathcal{D}$  None | Some x  $\Rightarrow$  distr-rai (f (fst x))) A  $\partial$  $\mathcal{B}$   
**unfolding** N-def **by** (intro arg-cong2[**where** f=nn-integral] ext)  
 (auto split:split-indicator option.split)  
**also have** ... =  $\int^+$  x. emeasure (if (map-option fst  $\circ$  m) x  $\in$  Some ' range-rm m  
 then distr-rai (f (the ((map-option fst  $\circ$  m) x)))  
 else return  $\mathcal{D}$  None) A  $\partial$  $\mathcal{B}$ )  
**by** (intro arg-cong2[**where** f=nn-integral] arg-cong2[**where** f=emeasure] refl ext)  
 (auto simp add: in-range-rmI vimage-def split:option.splits)  
**also have** ... =  
 $\int^+$  x. emeasure (if x  $\in$  Some ' range-rm m then ?m (f (the x)) else return  $\mathcal{D}$  None) A  $\partial$ ?m m  
**unfolding** distr-rai-def **using** distr-rai-measurable[OF wf-m]  
**by** (intro nn-integral-distr[symmetric]) (simp-all add:comp-def)  
**also have** ... = emeasure ?RHS A  
**using** 3 none-measure-subprob-algebra  
**by** (intro emeasure-bind[symmetric, **where** N= $\mathcal{D}$ ]) (auto simp add:distr-rai-def Pi-def)  
**finally show** emeasure ?L A = 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*:  $\text{distr-rai } \text{coin-rai} = \text{coin-spmf}$  (**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**  $\dots = \text{distr } (\text{distr } \mathcal{B} \mathcal{D} \text{chd}) \mathcal{D} \text{Some}$   
**by** (*subst distr-distr*) (*auto simp add:comp-def chd-measurable*)  
**also have**  $\dots = \text{map-pmf } \text{Some } (\text{pmf-of-set } \text{UNIV})$   
**unfolding** *distr-shd map-pmf-rep-eq* **by** *simp*  
**also have**  $\dots = \text{spmf-of-pmf } (\text{pmf-of-set } \text{UNIV})$   
**by** (*simp add:spmf-of-pmf-def*)  
**also have**  $\dots = \text{coin-spmf}$   
**by** *auto*

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

**qed**

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

**definition** *lub-rai* ::  $'a \text{ random-alg-int set} \Rightarrow 'a \text{ random-alg-int}$   
**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**  $g (\text{cshift } p \text{ cs}) = \text{Some } (r, \text{cs})$  **for** *cs*

**proof** –

**have**  $f (\text{cshift } p \text{ cs}) = \text{Some } (r, \text{cs})$   
**using** *assms(2)* **unfolding** *wf-on-prefix-def* **by** *auto*  
**moreover have**  $\text{flat-ord } \text{None } (f (\text{cshift } p \text{ cs})) (g (\text{cshift } p \text{ cs}))$   
**using** *assms(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**  $\bigwedge r. r \in R \implies \text{wf-random } r$   
**shows**  $\text{wf-random } (\text{lub-rai } R)$   
**proof** (*rule wf-randomI*)  
**fix**  $bs$   
**assume**  $a:\text{lub-rai } R \text{ } bs \neq \text{None}$   
**define**  $S$  **where**  $S = ((\lambda x. x \text{ } bs) \text{ } R)$   
**have**  $0:\text{lub-rai } R \text{ } bs = \text{flat-lub } \text{None } S$   
**unfolding**  $S\text{-def } \text{lub-rai-def } \text{fun-lub-def}$   
**by** (*intro arg-cong2[where f=flat-lub]*) *auto*  
  
**have**  $\text{lub-rai } R \text{ } bs = \text{None}$  **if**  $S \subseteq \{\text{None}\}$   
**using** *that* **unfolding**  $0 \text{ flat-lub-def}$  **by** *auto*  
**hence**  $\neg (S \subseteq \{\text{None}\})$   
**using**  $a$  **by** *auto*  
**then obtain**  $r$  **where**  $1:r \in R$  **and**  $2: r \text{ } bs \neq \text{None}$   
**unfolding**  $S\text{-def}$  **by** *blast*  
**then obtain**  $p \ y$  **where**  $3:\text{cprefix } p \text{ } bs$  **and**  $4:\text{wf-on-prefix } r \text{ } p \ y$   
**using**  $\text{assms}(2)[\text{OF } 1]$   $2$  **unfolding**  $\text{wf-random-def}$  **by** (*auto split.option.split-asm*)  
**have**  $\text{wf-on-prefix } (\text{lub-rai } R) \text{ } p \ y$   
**by** (*intro wf-lub-helper[OF - 4] random-alg-int-pd.lub-upper 1 assms(1)*)  
**thus**  $\exists p \ r. \text{cprefix } p \text{ } bs \wedge \text{wf-on-prefix } (\text{lub-rai } R) \text{ } p \ r$   
**using**  $3$  **by** *auto*  
**qed**

**lemma** *ord-rai-mono*:  
**assumes**  $\text{ord-rai } f \ g$   
**assumes**  $\neg (P \ \text{None})$   
**assumes**  $P (f \ bs)$   
**shows**  $P (g \ bs)$   
**using**  $\text{assms}$  **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**  $\text{None} \notin A$   
**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** (*intro wf-lub assms*)

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

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

**proof**

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

**next**

**assume**  $\text{map-option fst } (\text{lub-rai } F \text{ } bs) \in A$   
**then obtain**  $y$  **where**  $6:\text{lub-rai } F \text{ } bs = \text{Some } y \ \text{Some } (\text{fst } y) \in A$



using *assms(4)* by (*cases lub-rai F bs*) *auto*  
 hence  $f bs = None \vee f bs = Some y$  if  $f \in F$  for  $f$   
 using  $\_4[OF\ that]$  **unfolding** *ord-rai-def fun-ord-def flat-ord-def* by *auto*  
 moreover have  $lub-rai\ F\ bs = None$  if  $\bigwedge f. f \in F \implies f bs = None$   
 using *that* **unfolding** *lub-rai-def flat-lub-def fun-lub-def* by *auto*  
 ultimately obtain  $f$  where  $f bs = Some y$   $f \in F$   
 using *6(1)* by *auto*  
 thus  $\exists f \in F. map-option\ fst\ (f\ bs) \in A$   
 using *6(2)* by *force*  
 qed

**have 1:** *Complete-Partial-Order.chain* ( $\subseteq$ ) ( $(\lambda f. \{bs. map-option\ fst\ (f\ bs) \in A\}) \text{ ' } F$ )  
 using *assms(4)* by (*intro chain-imageI[OF assms(2)] Collect-mono impI*) (*auto intro:ord-rai-mono*)

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

have  $wf-f'$ : *wf-random*  $f$   
 by (*intro assms that*)  
 have  $\_4: ?T = \{bs \in tospace\ euclidean. (map-option\ fst \circ f)\ bs \in A\}$   
 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:**  $\{bs. map-option\ fst\ (f\ bs) \in A\} \in sets\ \mathcal{B}$  (**is**  $?L1 \in -$ ) if *wf-random*  $f$  for  $f$   
 using *distr-rai-measurable[OF that]*  
 by (*intro measurable-sets-coin-space[where P= $\lambda x. x \in A$  and  $A=D$ ]*) (*auto simp:comp-def*)

**have**  $?L = emeasure\ \mathcal{B}\ ((map-option\ fst \circ lub-rai\ F) - \text{' } A \cap space\ \mathcal{B})$   
**unfolding** *distr-rai-def* by (*intro emeasure-distr distr-rai-measurable[OF wf-lub]*) *auto*  
**also have**  $\dots = emeasure\ \mathcal{B}\ \{x. map-option\ fst\ (lub-rai\ F\ x) \in A\}$   
**unfolding** *space-coin-space* by (*simp add:vimage-def*)  
**also have**  $\dots = emeasure\ \mathcal{B}\ (\bigcup f \in F. \{bs. map-option\ fst\ (f\ bs) \in A\})$   
**unfolding**  $0$  by (*intro arg-cong2[where f=*emeasure*]*) *auto*  
**also have**  $\dots = Sup\ (emeasure\ \mathcal{B}\ \text{' } (\lambda f. \{bs. map-option\ fst\ (f\ bs) \in A\}) \text{' } F)$   
 using  $2$  by (*intro tau-additivity[OF coin-space-is-borel-measure]* *chain-imp-union-stable 1*)

*auto*

**also have**  $\dots = (SUP\ f \in F. (emeasure\ \mathcal{B}\ \{bs. map-option\ fst\ (f\ bs) \in A\}))$   
**unfolding** *image-image* by *simp*  
**also have**  $\dots = (SUP\ f \in F. emeasure\ \mathcal{B}\ ((map-option\ fst \circ f) - \text{' } A \cap space\ \mathcal{B}))$   
 by (*simp add:image-image space-coin-space vimage-def*)  
**also have**  $\dots = ?R$   
**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**  $None \notin A$   
**shows**  $emeasure\ (distr-rai\ f)\ A \leq emeasure\ (distr-rai\ g)\ A$  (**is**  $?L \leq ?R$ )

**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: distr-rai (λ-. None) = measure-pmf (return-pmf (None :: 'a 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 ∧ 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 ≧= g1) bs)
        case None
        then show ?thesis by (simp add:flat-ord-def)
    next
        case (Some a)
        then obtain y bs' where 0: f1 bs = Some (y,bs') and 1:g1 y bs' ≠ None and f1 bs ≠ None
            by (cases f1 bs, auto simp:bind-rai-def)
        hence f2 bs = f1 bs
            using assms(1) unfolding ord-rai-def fun-ord-def flat-ord-def by metis
        hence f2 bs = Some (y,bs')
            using 0 by auto
        moreover have g1 y bs' = g2 y bs'
            using assms(2) 1 unfolding ord-rai-def fun-ord-def flat-ord-def by metis
        ultimately have (f1 ≧= g1) bs = (f2 ≧= g2) 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- $\text{alg}$*  monad, that can be used to represent executable randomized algorithms. It is a type-definition based on the internal representation from Section 4 with the wellformedness restriction.

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

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

**theory** *Randomized-Algorithm*

**imports**

*Randomized-Algorithm-Internal*

**begin**

A stronger variant of *pmf-eqI*.

**lemma** *pmf-eq-iff-le*:

**fixes**  $p\ q :: 'a\ \text{pmf}$

**assumes**  $\bigwedge x. \text{pmf } p\ x \leq \text{pmf } q\ x$

**shows**  $p = q$

**proof** –

**have**  $(\int x. \text{pmf } q\ x - \text{pmf } p\ x\ \partial \text{count-space UNIV}) = 0$

**by** (*simp-all add:integrable-pmf integral-pmf*)

**moreover have** *integrable (count-space UNIV) ( $\lambda x. \text{pmf } q\ x - \text{pmf } p\ x$ )*

**by** (*simp add:integrable-pmf*)

**moreover have** *AE x in count-space UNIV.  $0 \leq \text{pmf } q\ x - \text{pmf } p\ x$*

**using** *assms unfolding AE-count-space by auto*

**ultimately have** *AE x in count-space UNIV.  $\text{pmf } q\ x - \text{pmf } p\ x = 0$*

**using** *integral-nonneg-eq-0-iff-AE by blast*

**hence**  $\bigwedge x. \text{pmf } p\ x = \text{pmf } q\ x$  **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  $p \geq \text{weight-spmf } q$*

**assumes** *ord-spmf (=)  $p\ q$*

**shows**  $p = q$

**proof** –

**have**  $\bigwedge x. \text{spmf } p\ x \leq \text{spmf } q\ x$

**using** *ord-spmf-eq-leD[OF assms(2)] by simp*

**moreover have** *pmf  $p\ \text{None} \leq \text{pmf } q\ \text{None}$*

**using** *assms(1) unfolding pmf-None-eq-weight-spmf by auto*

**ultimately have** *pmf  $p\ x \leq \text{pmf } q\ x$  for  $x$  by (cases  $x$ ) auto*

**thus** *?thesis using pmf-eq-iff-le by auto*

**qed**

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

**unfolding** *wf-random-def by auto*

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

**using** *wf-empty by (intro exI[where  $x = \lambda-. \text{None}$ ]) auto*

**setup-lifting** *type-definition-random- $\text{alg}$*

**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* *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**  
( $\lambda F.$  if *Complete-Partial-Order.chain ord-rai* F then *lub-rai* F else ( $\lambda x.$  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**  
( $\lambda f s.$  *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**  $x = \text{Some } 'range-rm\ f \cup \{None\}$ ]]) *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*  $f$ )  $\{x\} \neq 0$  (**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*  $\Rightarrow$  *nat* *spmf* **is** *used-bits-distr*  
**using** *used-bits-distr-is-pmf* **by** *auto*

**definition** *coin-usage-of-ra*

**where** *coin-usage-of-ra*  $p = \text{map-pmf } (case-option \ \infty \ \text{enat}) \ (coin-usage-of-ra-aux\ p)$

**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*  $f$ )  $\subseteq \text{Some } 'range-rm \ (Rep-random-alg\ f) \cup \{None\}$

**proof**

**let**  $?f = \text{Rep-random-alg } f$

**fix**  $x$  **assume**  $x \in \text{set-pmf } (spmf-of-ra\ f)$

**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**  $0 < \text{measure } \mathcal{B} \ \{\omega. \text{map-option } \text{fst } (?f\ \omega) = x\}$

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

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

**moreover have**  $\{\omega. \text{map-option } \text{fst } (?f\ \omega) = x\} = \{\}$  **if**  $x \notin \text{range } (\text{map-option } \text{fst} \circ ?f)$

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

hence  $\text{measure } \mathcal{B} \{ \omega. \text{map-option fst } (?f \ \omega) = x \} = 0$  if  $x \notin \text{range } (\text{map-option fst } \circ ?f)$   
 using that by *simp*  
 ultimately have  $x \in \text{range } (\text{map-option fst } \circ ?f)$   
 by *auto*  
 thus  $x \in \text{Some } \text{'range-rm } (\text{Rep-random-alg } f) \cup \{ \text{None} \}$   
 unfolding *range-rm-def* by *(cases x) auto*  
 qed

lemma *spmf-of-ra-return*:  $\text{spmf-of-ra } (\text{return-ra } x) = \text{return-spmf } x$

proof –

have  $\text{measure-pmf } (\text{spmf-of-ra } (\text{return-ra } x)) = \text{measure-pmf } (\text{return-spmf } x)$   
 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*:  $\text{spmf-of-ra } \text{coin-ra} = \text{coin-spmf}$

proof –

have  $\text{measure-pmf } (\text{spmf-of-ra } \text{coin-ra}) = \text{measure-pmf } \text{coin-spmf}$   
 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*:

$\text{spmf-of-ra } (\text{bind-ra } f \ g) = \text{bind-spmf } (\text{spmf-of-ra } f) (\lambda x. \text{spmf-of-ra } (g \ x))$  (is  $?L = ?R$ )

proof –

let  $?f = \text{Rep-random-alg } f$   
 let  $?g = \lambda x. \text{Rep-random-alg } (g \ x)$

have 0:  $x \in \text{Some } \text{'range-rm } ?f \vee x = \text{None}$  if  $x \in \text{set-pmf } (\text{spmf-of-ra } f)$  for  $x$   
 using that *set-pmf-spmf-of-ra* by *auto*

have  $\text{measure-pmf } ?L = \text{distr-rai } (?f \ggg ?g)$   
 unfolding *spmf-of-ra.rep-eq bind-ra.rep-eq* by *(simp add: comp-def)*  
 also have  $\dots = \text{distr-rai } ?f \ggg$   
 $(\lambda x. \text{if } x \in \text{Some } \text{'range-rm } ?f \text{ then } \text{distr-rai } (?g \ (\text{the } x)) \text{ else } \text{return } \mathcal{D} \ \text{None})$   
 by *(intro distr-rai-bind wf-rep-rand-alg)*  
 also have  $\dots = \text{measure-pmf } (\text{spmf-of-ra } f) \ggg$   
 $(\lambda x. \text{measure-pmf } (\text{if } x \in \text{Some } \text{'range-rm } ?f \text{ then } \text{spmf-of-ra } (g \ (\text{the } x)) \text{ else } \text{return-pmf } \text{None}))$   
 by *(intro arg-cong2 [where f=bind] ext) (auto simp: spmf-of-ra.rep-eq return-discrete)*  
 also have  $\dots = \text{measure-pmf } (\text{spmf-of-ra } f) \ggg$   
 $(\lambda x. \text{if } x \in \text{Some } \text{'range-rm } ?f \text{ then } \text{spmf-of-ra } (g \ (\text{the } x)) \text{ else } \text{return-pmf } \text{None})$   
 unfolding *bind-pmf.rep-eq* by *(simp add: comp-def id-def)*  
 also have  $\dots = \text{measure-pmf } ?R$   
 using 0 unfolding *bind-spmf-def*  
 by *(intro arg-cong [where f=measure-pmf] bind-pmf-cong refl) (auto split: option.split)*  
 finally have  $\text{measure-pmf } ?L = \text{measure-pmf } ?R$  by *simp*  
 thus *?thesis*  
 using *measure-pmf-inject* by *blast*

qed

lemma *spmf-of-ra-mono*:

assumes *ord-ra f g*  
 shows  $\text{ord-spmf } (=) (\text{spmf-of-ra } f) (\text{spmf-of-ra } g)$

proof –

**have**  $\text{ord-rai } (\text{Rep-random-alg } f) (\text{Rep-random-alg } g)$   
**using** *assms* **unfolding**  $\text{ord-ra.rep-eq}$  **by** *simp*  
**hence**  $\text{ennreal } (\text{spmf } (\text{spmf-of-ra } f) x) \leq \text{ennreal } (\text{spmf } (\text{spmf-of-ra } g) x)$  **for**  $x$   
**unfolding**  $\text{emeasure-pmf-single[symmetric]}$   $\text{spmf-of-ra.rep-eq}$   
**by** (*intro distr-rai-ord-rai-mono wf-rep-rand-alg*) *auto*  
**hence**  $\text{spmf } (\text{spmf-of-ra } f) x \leq \text{spmf } (\text{spmf-of-ra } g) x$  **for**  $x$   
**by** *simp*  
**thus** *?thesis*  
**by** (*intro ord-pmf-increaseI*) *auto*  
**qed**

**lemma** *spmf-of-ra-lub-ra-empty*:  
 $\text{spmf-of-ra } (\text{lub-ra } \{\}) = \text{return-pmf } \text{None}$  (**is**  $?L = ?R$ )

**proof** –  
**have**  $\text{measure-pmf } ?L = \text{distr-rai } (\text{lub-rai } \{\})$   
**unfolding**  $\text{spmf-of-ra.rep-eq lub-ra.rep-eq Complete-Partial-Order.chain-def}$  **by** *auto*  
**also have**  $\dots = \text{distr-rai } (\lambda-. \text{None})$   
**unfolding**  $\text{lub-rai-def fun-lub-def flat-lub-def}$  **by** *auto*  
**also have**  $\dots = \text{measure-pmf } ?R$   
**unfolding**  $\text{distr-rai-None}$  **by** *simp*  
**finally have**  $\text{measure-pmf } ?L = \text{measure-pmf } ?R$   
**by** *simp*  
**thus** *?thesis*  
**using** *measure-pmf-inject* **by** *auto*  
**qed**

**lemma** *spmf-of-ra-lub-ra*:  
**fixes**  $A :: 'a \text{ random-alg set}$   
**assumes**  $\text{Complete-Partial-Order.chain ord-ra } A$   
**shows**  $\text{spmf-of-ra } (\text{lub-ra } A) = \text{lub-spmf } (\text{spmf-of-ra } `A)$  (**is**  $?L = ?R$ )

**proof** (*cases*  $A \neq \{\}$ )  
**case** *True*  
**have**  $0: \text{Complete-Partial-Order.chain ord-rai } (\text{Rep-random-alg } `A)$   
**using** *assms* **unfolding**  $\text{ord-ra.rep-eq Complete-Partial-Order.chain-def}$  **by** *auto*  
**have**  $1: \text{Complete-Partial-Order.chain } (\text{ord-spmf } (=)) (\text{spmf-of-ra } `A)$   
**using**  $\text{spmf-of-ra-mono}$  **by** (*intro chain-imageI[OF assms]*) *auto*

**show** *?thesis*

**proof** (*rule*  $\text{spmf-eqI}$ )  
**fix**  $x :: 'a$   
**have**  $\text{ennreal } (\text{spmf } ?L x) = \text{emeasure } (\text{distr-rai } (\text{lub-rai } (\text{Rep-random-alg } `A))) \{\text{Some } x\}$   
**using**  $0$  **unfolding**  $\text{emeasure-pmf-single[symmetric]}$   $\text{spmf-of-ra.rep-eq lub-ra.rep-eq}$  **by** *simp*  
**also have**  $\dots = (\text{SUP } f \in \text{Rep-random-alg } `A. \text{emeasure } (\text{distr-rai } f) \{\text{Some } x\})$   
**using** *True wf-rep-rand-alg* **by** (*intro distr-rai-lub 0*) *auto*  
**also have**  $\dots = (\text{SUP } p \in A. \text{ennreal } (\text{spmf } (\text{spmf-of-ra } p) x))$   
**unfolding**  $\text{emeasure-pmf-single[symmetric]}$   $\text{spmf-of-ra.rep-eq}$  **by** (*simp add:image-image*)  
**also have**  $\dots = (\text{SUP } p \in \text{spmf-of-ra } `A. \text{ennreal } (\text{spmf } p x))$   
**by** (*simp add:image-image*)  
**also have**  $\dots = \text{ennreal } (\text{spmf } ?R x)$   
**using** *True* **by** (*intro ennreal-spmf-lub-spmf[symmetric] 1*) *auto*  
**finally have**  $\text{ennreal } (\text{spmf } ?L x) = \text{ennreal } (\text{spmf } ?R x)$   
**by** *simp*  
**thus**  $\text{spmf } ?L x = \text{spmf } ?R x$   
**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  $\bigwedge A. \text{Complete-Partial-Order.chain } ord (f ' A) \implies l1 (f ' A) = f (l2 A)$   
assumes *partial-function-definitions ord l1*  
assumes *inj f*  
shows *partial-function-definitions (img-ord f ord) l2*

proof –

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 A*

proof –

have  $0: \text{Complete-Partial-Order.chain } ord (f ' A)$   
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* ( $\forall x. x \in A \longrightarrow \text{img-ord } f \text{ ord } x z$ )  
for *z A*

proof –

have  $0: \text{Complete-Partial-Order.chain } ord (f ' A)$   
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**  $0$ : *inj Rep-random-alg*  
**using** *Rep-random-alg-inject unfolding inj-on-def* **by** *auto*

**have**  $1$ : *partial-function-definitions ord-rai lub-rai*  
**using** *random-alg-int-pd-fact* **by** *simp*

**have**  $2$ : *ord-ra = 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 ( $\lambda f.$  bind-ra (B f) ( $\lambda y.$  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*  $\ggg (\lambda x.$  *return-ra (f x)*)

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

*spmf-of-ra-return spmf-of-ra-bind spmf-of-ra-coin spmf-of-ra-map*

**lemma** *map-mono-ra [partial-function-mono]*:

**assumes** *mono-ra B*  
**shows** *mono-ra ( $\lambda f.$  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* (*spmf-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** [*mcont lub ord lub-spmf* (*ord-spmf* (=)) *f*; *mcont lub ord lub-ra ord-ra g* ]  
 $\implies$  *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*  $\implies$  *rel-spmf-of-ra*)  $\implies$  *A*  $\implies$  *rel-spmf-of-ra*) *F* *G*  
**shows** (*A*  $\implies$  *rel-spmf-of-ra*) (*spmf.fixp-fun* *F*) (*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**  $P=(A \implies \text{rel-spmf-of-ra})$ ])  
**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*  $\implies$  *rel-spmf-of-ra*) (*fst* *x*) (*snd* *x*))  
**unfolding** *rel-fun-def*  
**by**(*rule admissible-all admissible-imp cont-intro*)  
**show** (*A*  $\implies$  *rel-spmf-of-ra*) ( $\lambda.$  *lub-spmf* {}) ( $\lambda.$  *lub-ra* {})  
**by** (*auto simp: rel-fun-def rel-spmf-of-ra-def spmf-of-ra-lub-ra-empty*)  
**show** (*A*  $\implies$  *rel-spmf-of-ra*) (*F* *f*) (*G* *g*) **if** (*A*  $\implies$  *rel-spmf-of-ra*) *f g* **for** *f g*  
**using** *that* **by**(*rule rel-funD*[*OF param*])  
**qed**

**lemma** *return-ra-transfer*[*transfer-rule*]: ((=)  $\implies$  *rel-spmf-of-ra*) *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*]:  
(*rel-spmf-of-ra*  $\implies$  ((=)  $\implies$  *rel-spmf-of-ra*)  $\implies$  *rel-spmf-of-ra*) *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]*:  
 $((=) \implies \text{rel-spmf-of-ra} \implies \text{rel-spmf-of-ra}) \text{ map-spmf map-ra}$   
**unfolding** *rel-fun-def rel-spmf-of-ra-def spmf-of-ra-map* **by** *simp*

**end**

**declare** *[[function-internals]]*

**declaration**  $\langle \text{Partial-Function.init random-alg term} \langle \text{random-alg-pf.fixp-fun} \rangle$   
 $\text{term} \langle \text{random-alg-pf.mono-body} \rangle$   
 $\text{@}\{ \text{thm random-alg-pf.fixp-rule-uc} \} \text{@}\{ \text{thm random-alg-pf.fixp-induct-uc} \}$   
 $\text{NONE} \rangle$

## 5.1 Almost surely terminating randomized algorithms

**definition** *terminates-almost-surely* ::  $'a \text{ random-alg} \Rightarrow \text{bool}$   
**where** *terminates-almost-surely*  $f \longleftrightarrow \text{lossless-spmf} (\text{spmof-of-ra } f)$

**definition** *pmf-of-ra* ::  $'a \text{ random-alg} \Rightarrow 'a \text{ pmf}$  **where**  
*pmf-of-ra*  $p = \text{map-pmf the} (\text{spmof-of-ra } p)$

**lemma** *pmf-of-spmf*:  $\text{map-pmf the} (\text{spmof-of-pmf } x) = x$   
**by** (*simp add:map-pmf-comp spmf-of-pmf-def*)

**definition** *coin-pmf* ::  $\text{bool pmf}$  **where** *coin-pmf* = *pmf-of-set UNIV*

**lemma** *pmf-of-ra-coin*:  $\text{pmf-of-ra} (\text{coin-ra}) = \text{coin-pmf}$  (**is**  $?L = ?R$ )

**proof** –

**have**  $0: \text{spmof-of-ra} (\text{coin-ra}) = \text{spmof-of-pmf} (\text{pmf-of-set UNIV})$

**unfolding** *spmof-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*:  $\text{pmf-of-ra} (\text{return-ra } x) = \text{return-pmf } x$

**unfolding** *pmf-of-ra-def spmf-of-ra-return* **by** *simp*

**lemma** *pmf-of-ra-bind*:

**assumes** *terminates-almost-surely*  $f$

**shows**  $\text{pmf-of-ra} (f \ggg g) = \text{pmf-of-ra } f \ggg (\lambda x. \text{pmf-of-ra} (g x))$  (**is**  $?L = ?R$ )

**proof** –

**have**  $0: x \neq \text{None}$  **if**  $x \in \text{set-pmf} (\text{spmof-of-ra } f)$  **for**  $x$

**using** *assms that* **unfolding** *terminates-almost-surely-def*

**by** (*meson lossless-iff-set-pmf-None*)

**have**  $?L = \text{spmof-of-ra } f \ggg (\lambda x. \text{map-pmf the} (\text{case-option} (\text{return-pmf None}) (\text{spmof-of-ra} \circ g))$   
 $x))$

**unfolding** *pmf-of-ra-def spmf-of-ra-bind bind-spmf-def map-bind-pmf comp-def* **by** *simp*

**also have**  $\dots = \text{spmof-of-ra } f \ggg$

$(\lambda x. (\text{case } x \text{ of None} \Rightarrow \text{return-pmf} (\text{the None}) \mid \text{Some } x \Rightarrow \text{pmf-of-ra} (g x)))$

**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**  $\dots = \text{spmof-of-ra } f \ggg (\lambda x. \text{pmf-of-ra} (g (\text{the } x)))$

**using**  $0$  **by** (*intro bind-pmf-cong refl*) (*auto split:option.split*)

**also have**  $\dots = ?R$

**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 (spmof-of-ra f)*  
**using** *assms(1) lossless-iff-set-pmf-None* **unfolding** *terminates-almost-surely-def*  
**by** *blast*  
**hence** *Some x  $\in$  set-pmf (spmof-of-ra f)  $\iff$  x  $\in$  the ‘ set-pmf (spmof-of-ra f) for x*  
**by** (*metis image-iff option.collapse option.sel*)  
**hence** *set-spmf (spmof-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')  $\leftarrow$  *f* *bs*;  
n  $\leftarrow$  *consumed-bits* *f* *bs*;  
Some ((r,n),bs')  
}

**lemma** *track-random-bits-Some-iff*:

**assumes** *track-random-bits* *f* *bs*  $\neq$  None

**shows** *f* *bs*  $\neq$  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* ( $\lambda p. ((\text{eval-rm } f \ p, \text{length } p), \text{cdrop } (\text{length } p) \text{ } bs))$  (*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*  $\ggg$  ( $\lambda b. \text{return-rai } (b,1)$ ) (**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**  $\{p \in \text{ptree-rm } (\text{coin-rai}). \text{cprefix } p \text{ } bs\} = \{[\text{chd } bs]\}$

**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**  $bs :: \text{coin-stream}$   
**have**  $wf\text{-on-prefix (return-rai } x) [] x$   
**unfolding**  $wf\text{-on-prefix-def return-rai-def}$  **by** *simp*  
**moreover have**  $c\text{prefix } [] bs$   
**unfolding**  $c\text{prefix-def}$  **by** *simp*  
**ultimately have**  $\{p \in \text{ptree-rm (return-rai } x). c\text{prefix } p bs\} = \{[]\}$   
**by** (*intro prefixes-singleton*) (*auto simp:ptree-rm-def*)  
**hence**  $\text{consumed-prefix (return-rai } x) bs = \text{Some } []$   
**unfolding**  $\text{consumed-prefix-def}$  **by** *simp*  
**hence**  $\text{consumed-bits (return-rai } x) bs = \text{Some } 0$   
**unfolding**  $\text{consumed-bits-def}$  **by** *simp*  
**thus**  $?L bs = ?R bs$   
**unfolding**  $\text{track-random-bits-def}$  **by** (*simp add:return-rai-def*)  
**qed**

**lemma** *consumed-prefix-imp-wf*:  
**assumes**  $\text{consumed-prefix } m bs = \text{Some } p$   
**shows**  $wf\text{-on-prefix } m p (\text{eval-rm } m p)$   
**proof** –  
**have**  $p \in \text{ptree-rm } m$   
**using** *assms* **unfolding**  $\text{consumed-prefix-def the-elem-opt-Some-iff}[OF \text{ prefixes-at-most-one}]$   
**by** *blast*  
**then obtain**  $r$  **where**  $wf\text{-on-prefix } m p r$   
**unfolding**  $\text{ptree-rm-def}$  **by** *auto*  
**thus** *?thesis*  
**unfolding**  $wf\text{-on-prefix-def eval-rm-def}$  **by** *simp*  
**qed**

**lemma** *consumed-prefix-imp-prefix*:  
**assumes**  $\text{consumed-prefix } m bs = \text{Some } p$   
**shows**  $c\text{prefix } p bs$   
**using** *assms* **unfolding**  $\text{consumed-prefix-def the-elem-opt-Some-iff}[OF \text{ prefixes-at-most-one}]$  **by**  
*blast*

**lemma** *consumed-prefix-bindI*:  
**assumes**  $\text{consumed-prefix } m bs = \text{Some } p$   
**assumes**  $\text{consumed-prefix (f (eval-rm } m p)) (\text{cdrop (length } p) bs) = \text{Some } q$   
**shows**  $\text{consumed-prefix (m } \gg f) bs = \text{Some (p@q)}$

**proof** –  
**define**  $r$  **where**  $r = \text{eval-rm } m p$   
  
**have**  $0: wf\text{-on-prefix } m p r$   
**unfolding**  $r\text{-def}$  **using**  $\text{consumed-prefix-imp-wf}[OF \text{ assms}(1)]$  **by** *simp*  
  
**have**  $\text{consumed-prefix (f } r) (\text{cdrop (length } p) bs) = \text{Some } q$   
**using** *assms*(2) **unfolding**  $r\text{-def}$  **by** *simp*  
**hence**  $1: wf\text{-on-prefix (f } r) q (\text{eval-rm (f } r) q)$   
**using**  $\text{consumed-prefix-imp-wf}$  **by** *auto*  
**have**  $wf\text{-on-prefix (m } \gg f) (p@q) (\text{eval-rm (f } r) q)$   
**by** (*intro wf-on-prefix-bindI[where r=r] 0 1*)  
**hence**  $p@q \in \text{ptree-rm (m } \gg f)$   
**unfolding**  $\text{ptree-rm-def}$  **by** *auto*  
**moreover have**  $c\text{prefix } p bs c\text{prefix } q (\text{cdrop (length } p) bs)$   
**using**  $\text{consumed-prefix-imp-prefix assms}$  **by** *auto*  
**hence**  $c\text{prefix (p@q) } bs$   
**unfolding**  $c\text{prefix-def}$  **by** (*metis length-append ctake-add*)  
**ultimately have**  $\{p \in \text{ptree-rm (m } \gg f). c\text{prefix } p bs\} = \{p@q\}$

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  $\text{track-random-bits } (m \ggg f) = \text{track-random-bits } m \ggg$   
 $(\lambda(r,n). \text{track-random-bits } (f r) \ggg (\lambda(r',m). \text{return-rai } (r',n+m)))$  (is ?L = ?R)

proof (rule ext)

fix bs :: coin-stream  
 have wf-bind: wf-random (m  $\ggg$  f)  
 by (intro wf-bind assms)

consider (a) m bs = None | (b) m bs  $\neq$  None  $\wedge$  (m  $\ggg$  f) bs = None | (c) (m  $\ggg$  f) bs  $\neq$  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  $\ggg$  f) bs = None if m bs = None  
 using that unfolding bind-rai-def by simp  
 hence m bs  $\neq$  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  $\in$  range-rm m using 1 in-range-rmI by metis  
 hence wf: wf-random (f r) by (intro assms(2))  
 have f r bs'  $\neq$  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  $\ggg$  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  $\ggg$  f) (p@q)  
 unfolding eval-rm-def bind-rai-def wf-on-prefix-def by simp

**ultimately have**  
 $?R \text{ } bs = \text{Some } ((\text{eval-rm } (m \gg f) (p@q), \text{length } p + \text{length } q), \text{cdrop } (\text{length } p + \text{length } q) \text{ } bs)$   
**unfolding** *bind-rai-def return-rai-def bs'-def* **by** *simp*  
**also have**  $\dots = ?L \text{ } bs$   
**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  $\neq$  None* **for** *bs*  
**proof** –  
**have** *f bs  $\neq$  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  $\in$  ptree-rm h*  
**using** *that* **unfolding** *ptree-rm-def* **by** *auto*  
**hence**  $\{p \in \text{ptree-rm } h. \text{cprefix } p \text{ } bs\} = \{p\}$   
**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  $\neq$  None*  
**hence** *f bs  $\neq$  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**  $\{q \in \text{ptree-rm } f. \text{cprefix } q \text{ (cshift } p \text{ cs)}\} = \{p\}$  **for**  $cs$   
**by** *(intro prefixes-singleton)* *(auto simp:cprefix-def ptree-rm-def)*  
**hence**  $\text{consumed-prefix } f \text{ (cshift } p \text{ cs)} = \text{Some } p$  **for**  $cs$   
**unfolding** *consumed-prefix-def* **by** *simp*  
**hence**  $\text{wf-on-prefix (track-random-bits } f) p (r, \text{length } p)$   
**using**  $0$  **unfolding** *track-random-bits-def wf-on-prefix-def consumed-bits-def* **by** *simp*  
**thus**  $\exists p r. \text{cprefix } p \text{ bs} \wedge \text{wf-on-prefix (track-random-bits } f) p r$   
**using**  $0$  **by** *auto*  
**qed**

**lemma** *track-random-bits-lub-rai:*

**assumes** *Complete-Partial-Order.chain ord-rai A*  
**assumes**  $\bigwedge r. r \in A \implies \text{wf-random } r$   
**shows**  $\text{track-random-bits (lub-rai } A) = \text{lub-rai (track-random-bits ' } A)$  **(is ?L = ?R)**

**proof** –

**have**  $0: \text{Complete-Partial-Order.chain ord-rai (track-random-bits ' } A)$   
**by** *(intro chain-imageI[OF assms(1)] track-random-bits-mono assms(2))*

**have**  $?L \text{ bs} = ?R \text{ bs}$  **if**  $?L \text{ bs} \neq \text{None}$  **for**  $bs$

**proof** –

**have**  $1: \text{lub-rai } A \text{ bs} \neq \text{None}$  **using** *that track-random-bits-Some-iff* **by** *simp*

**have**  $\text{lub-rai } A \text{ bs} = \text{None}$  **if**  $\bigwedge f. f \in A \implies f \text{ bs} = \text{None}$

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

**using**  $1$  **by** *blast*

**hence**  $\text{consumed-prefix } f \text{ bs} \neq \text{None}$

**using** *consumed-prefix-none-iff[OF assms(2)][OF f-in-A]* **by** *simp*

**hence**  $2: \text{track-random-bits } f \text{ bs} \neq \text{None}$

**unfolding** *track-random-bits-alt[OF assms(2)][OF f-in-A]* **by** *simp*

**have**  $\text{ord-rai (track-random-bits } f) (\text{track-random-bits (lub-rai } A))$

**by** *(intro track-random-bits-mono wf-lub[OF assms(1)] assms(2)*

*random-alg-int-pd.lub-upper[OF assms(1)] f-in-A)*

**hence**  $\text{track-random-bits (lub-rai } A) \text{ bs} = \text{track-random-bits } f \text{ bs}$

**using**  $2$  **unfolding** *ord-rai-def fun-ord-def flat-ord-def* **by** *metis*

**moreover have**  $\text{ord-rai (track-random-bits } f) (\text{lub-rai (track-random-bits ' } A))$

**using**  $f \text{-in-} A$  **by** *(intro random-alg-int-pd.lub-upper[OF 0]) auto*

**hence**  $\text{lub-rai (track-random-bits ' } A) \text{ bs} = \text{track-random-bits } f \text{ bs}$

**using**  $2$  **unfolding** *ord-rai-def fun-ord-def flat-ord-def* **by** *metis*

**ultimately show** *?thesis* **by** *simp*

**qed**

**hence**  $\text{flat-ord None (?L } \text{bs}) (?R \text{ bs})$  **for**  $bs$

**unfolding** *flat-ord-def* **by** *blast*

**hence**  $\text{ord-rai ?L ?R}$

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

**moreover have**  $\text{ord-rai (track-random-bits } f) (\text{track-random-bits (lub-rai } A))$  **if**  $f \in A$  **for**  $f$

**using** *that assms(2) wf-lub[OF assms(1,2)]*

**by** *(intro track-random-bits-mono random-alg-int-pd.lub-upper[OF assms(1)])*

**hence**  $\text{ord-rai ?R ?L}$

**by** *(intro random-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**  $\text{wf-random } f$

**shows**  $\text{track-random-bits } f \ggg (\lambda x. \text{return-rai (fst } x)) = f$  **(is ?L = ?R)**

**proof** –

**have**  $?L \text{ bs} = ?R \text{ bs}$  **for**  $\text{bs}$   
**unfolding** *track-random-bits-alt*[*OF assms*] *bind-rai-def return-rai-def*  
**by** (*subst wf-random-alt*[*OF assms*]) (*cases consumed-prefix f bs, auto*)  
**thus** *?thesis*  
**by** *auto*  
**qed**

**lift-definition** *track-coin-use* ::  $'a \text{ random-alg} \Rightarrow ('a \times \text{nat}) \text{ random-alg}$   
**is** *track-random-bits*  
**by** (*rule wf-track-random-bits*)

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

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

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

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

Monad laws:

**lemma** *return-bind-tra*:  
 $\text{bind-tra } (\text{return-tra } x) g = g \ x$   
**unfolding** *bind-tra-def return-tra-def*  
**by** (*simp add:bind-return-ra return-bind-ra*)

**lemma** *bind-tra-assoc*:  
 $\text{bind-tra } (\text{bind-tra } f g) h = \text{bind-tra } f (\lambda x. \text{bind-tra } (g \ x) h)$   
**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*:  
 $\text{bind-tra } m \text{return-tra} = m$   
**unfolding** *bind-tra-def return-tra-def*  
**by** (*simp add:bind-return-ra return-bind-ra*)

**lemma** *track-coin-use-bind*:  
**fixes**  $m :: 'a \text{ random-alg}$   
**fixes**  $f :: 'a \Rightarrow 'b \text{ random-alg}$   
**shows**  $\text{track-coin-use } (m \ggg f) = \text{track-coin-use } m \ggg (\lambda r. \text{track-coin-use } (f \ r))$   
**(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*  $\Rightarrow$  *'a random-alg*  $\Rightarrow$  *bool* **where**

*rel-track-coin-use q p*  $\longleftrightarrow$  *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  $\neq$  {}*

**and** *R:  $\forall (p, q) \in Y. rel-track-coin-use p q$*

**from** *R* **have** *R:  $\bigwedge p q. (p, q) \in Y \implies rel-track-coin-use p q$*  **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  $\neq$  {}* **and** *Y2: snd ' Y  $\neq$  {}* **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**  $\llbracket mcont\ lub\ ord\ lub-ra\ ord-ra\ f; mcont\ lub\ ord\ lub-ra\ ord-ra\ g \rrbracket$

$\implies ccpo.admissible\ lub\ ord\ (\lambda x. rel-track-coin-use\ (f\ x)\ (g\ x))$

**by** (*rule admissible-subst[OF admissible-rel-track-coin-use, where f= $\lambda x. (f\ x, g\ x)$ , 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  $f: \bigwedge x. \text{mono-ra } (\lambda f. F f x)$ 
and  $g: \bigwedge x. \text{mono-ra } (\lambda f. G f x)$ 
and param:  $((A \text{====> rel-track-coin-use}) \text{====> } A \text{====> rel-track-coin-use}) F G$ 
shows  $(A \text{====> rel-track-coin-use}) (\text{random-alg-pf.fixp-fun } F) (\text{random-alg-pf.fixp-fun } G)$ 
using  $f g$ 
proof(rule parallel-fixp-induct-1-1[OF
  random-alg-pfd random-alg-pfd - - reflexive reflexive,
  where  $P=(A \text{====> rel-track-coin-use})$ ])
show  $\text{ccpo.admissible } (\text{prod-lub } (\text{fun-lub lub-ra}) (\text{fun-lub lub-ra}))$ 
   $(\text{rel-prod } (\text{fun-ord ord-ra}) (\text{fun-ord ord-ra}))$ 
   $(\lambda x. (A \text{====> rel-track-coin-use}) (\text{fst } x) (\text{snd } x))$ 
unfolding rel-fun-def
by(rule admissible-all admissible-imp cont-intro)+
have  $0:\text{track-coin-use } (\text{lub-ra } \{\}) = \text{lub-ra } \{\}$ 
using track-coin-use-lub[where  $A=\{\}$ ]
by (simp add: Complete-Partial-Order.chain-def)
show  $(A \text{====> rel-track-coin-use}) (\lambda-. \text{lub-ra } \{\}) (\lambda-. \text{lub-ra } \{\})$ 
by (auto simp: rel-fun-def rel-track-coin-use-def 0)
show  $(A \text{====> rel-track-coin-use}) (F f) (G g)$  if  $(A \text{====> rel-track-coin-use}) f g$  for  $f g$ 
using that by(rule rel-funD[OF param])
qed

```

```

lemma return-ra-transfer[transfer-rule]:  $((=) \text{====> rel-track-coin-use}) \text{return-tra return-ra}$ 
unfolding rel-fun-def rel-track-coin-use-def track-coin-use-return by simp

```

```

lemma bind-ra-transfer[transfer-rule]:
   $(\text{rel-track-coin-use} \text{====> } ((=) \text{====> rel-track-coin-use}) \text{====> rel-track-coin-use}) \text{bind-tra}$ 
  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  $'a \text{tspmf} = ('a \times \text{nat}) \text{spmf}$ 

```

```

definition return-tspmf  $:: 'a \Rightarrow 'a \text{tspmf}$  where
  return-tspmf  $x = \text{return-spmf } (x,0)$ 

```

```

definition coin-tspmf  $:: \text{bool} \text{tspmf}$  where
  coin-tspmf  $= \text{pair-spmf coin-spmf } (\text{return-spmf } 1)$ 

```

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

**ad hoc overloading**  $Monad\text{-}Syntax.bind\ bind\text{-}tspmf$

Monad laws:

**lemma**  $return\text{-}bind\text{-}tspmf$ :

$bind\text{-}tspmf\ (return\text{-}tspmf\ x)\ g = g\ x$

**unfolding**  $bind\text{-}tspmf\text{-}def\ return\text{-}tspmf\text{-}def\ map\text{-}spmf\text{-}conv\text{-}bind\text{-}spmf$   
**by** ( $simp\ add:apsnd\text{-}def\ map\text{-}prod\text{-}def$ )

**lemma**  $bind\text{-}tspmf\text{-}assoc$ :

$bind\text{-}tspmf\ (bind\text{-}tspmf\ f\ g)\ h = bind\text{-}tspmf\ f\ (\lambda x. bind\text{-}tspmf\ (g\ x)\ h)$

**unfolding**  $bind\text{-}tspmf\text{-}def$   
**by** ( $simp\ add:case\text{-}prod\text{-}beta'\ algebra\text{-}simps\ map\text{-}spmf\text{-}conv\text{-}bind\text{-}spmf\ apsnd\text{-}def\ map\text{-}prod\text{-}def$ )

**lemma**  $bind\text{-}return\text{-}tspmf$ :

$bind\text{-}tspmf\ m\ return\text{-}tspmf = m$

**unfolding**  $bind\text{-}tspmf\text{-}def\ return\text{-}tspmf\text{-}def\ map\text{-}spmf\text{-}conv\text{-}bind\text{-}spmf\ apsnd\text{-}def$   
**by** ( $simp\ add:case\text{-}prod\text{-}beta'$ )

**lemma**  $bind\text{-}mono\text{-}tspmf\text{-}aux$ :

**assumes**  $ord\text{-}spmf\ (=)\ f1\ f2 \wedge y. ord\text{-}spmf\ (=)\ (g1\ y)\ (g2\ y)$

**shows**  $ord\text{-}spmf\ (=)\ (bind\text{-}tspmf\ f1\ g1)\ (bind\text{-}tspmf\ f2\ g2)$

**using**  $assms$  **unfolding**  $bind\text{-}tspmf\text{-}def\ map\text{-}spmf\text{-}conv\text{-}bind\text{-}spmf$   
**by** ( $auto\ intro!: bind\text{-}spmf\text{-}mono'\ simp\ add:case\text{-}prod\text{-}beta'$ )

**lemma**  $bind\text{-}mono\text{-}tspmf\ [partial\text{-}function\text{-}mono]$ :

**assumes**  $mono\text{-}spmf\ B$  **and**  $\wedge y. mono\text{-}spmf\ (C\ y)$

**shows**  $mono\text{-}spmf\ (\lambda f. bind\text{-}tspmf\ (B\ f)\ (\lambda y. C\ y\ f))$

**using**  $assms$  **by** ( $intro\ monotoneI\ bind\text{-}mono\text{-}tspmf\text{-}aux$ ) ( $auto\ simp:monotone\text{-}def$ )

**definition**  $ord\text{-}tspmf :: 'a\ tspmf \Rightarrow 'a\ tspmf \Rightarrow bool$  **where**

$ord\text{-}tspmf = ord\text{-}spmf\ (\lambda x\ y. fst\ x = fst\ y \wedge snd\ x \geq snd\ y)$

**bundle**  $ord\text{-}tspmf\text{-}notation$

**begin**

**notation**  $ord\text{-}tspmf\ ((-/ \leq_R -) [51, 51] 50)$

**end**

**bundle**  $no\text{-}ord\text{-}tspmf\text{-}notation$

**begin**

**no-notation**  $ord\text{-}tspmf\ ((-/ \leq_R -) [51, 51] 50)$

**end**

**unbundle**  $ord\text{-}tspmf\text{-}notation$

**definition**  $coin\text{-}usage\text{-}of\text{-}tspmf :: 'a\ tspmf \Rightarrow enat\ pmf$

**where**  $coin\text{-}usage\text{-}of\text{-}tspmf = map\text{-}pmf\ (\lambda x. case\ x\ of\ None \Rightarrow \infty \mid Some\ y \Rightarrow enat\ (snd\ y))$

**definition**  $expected\text{-}coin\text{-}usage\text{-}of\text{-}tspmf :: 'a\ tspmf \Rightarrow ennreal$

**where**

$expected\text{-}coin\text{-}usage\text{-}of\text{-}tspmf\ p = (\int^+ x. x\ \partial(map\text{-}pmf\ ennreal\text{-}of\text{-}enat\ (coin\text{-}usage\text{-}of\text{-}tspmf\ p)))$

**definition**  $expected\text{-}coin\text{-}usage\text{-}of\text{-}ra$  **where**

$expected\text{-}coin\text{-}usage\text{-}of\text{-}ra\ p = \int^+ x. x\ \partial(map\text{-}pmf\ ennreal\text{-}of\text{-}enat\ (coin\text{-}usage\text{-}of\text{-}ra\ p))$

**definition** *result* :: 'a tpmf  $\Rightarrow$  'a spmf  
**where** *result* = *map-spmf fst*

**lemma** *coin-usage-of-tpmf-alt-def*:

*coin-usage-of-tpmf p* = *map-pmf* ( $\lambda x. \text{case } x \text{ of None } \Rightarrow \infty \mid \text{Some } y \Rightarrow \text{enat } y$ ) (*map-spmf snd p*)

**unfolding** *coin-usage-of-tpmf-def map-pmf-comp map-option-case*  
**by** (*metis enat-def infinity-enat-def option.case-eq-if option.sel*)

**lemma** *coin-usage-of-tpmf-bind-return*:

*coin-usage-of-tpmf* (*bind-tpmf f* ( $\lambda x. \text{return-tpmf } (g \ x)$ )) = (*coin-usage-of-tpmf f*)

**unfolding** *bind-tpmf-def return-tpmf-def coin-usage-of-tpmf-alt-def map-spmf-bind-spmf*  
**by** (*simp add:comp-def case-prod-beta map-spmf-conv-bind-spmf*)

**lemma** *coin-usage-of-tpmf-mono*:

**assumes** *ord-tpmf p q*

**shows** *measure* (*coin-usage-of-tpmf p*)  $\{..k\} \leq \text{measure}$  (*coin-usage-of-tpmf q*)  $\{..k\}$

**proof** –

**define** *p'* **where** *p'* = *map-spmf snd p*

**define** *q'* **where** *q'* = *map-spmf snd q*

**have** *0:ord-spmf* ( $\geq$ ) *p' q'*

**using** *assms(1) ord-spmf-mono unfolding p'-def q'-def ord-tpmf-def ord-spmf-map-spmf12*

**by** *fastforce*

**have** *cp:coin-usage-of-tpmf p* = *map-pmf* (*case-option*  $\infty$  *enat*) *p'*

**unfolding** *coin-usage-of-tpmf-alt-def p'-def* **by** *simp*

**have** *cq:coin-usage-of-tpmf q* = *map-pmf* (*case-option*  $\infty$  *enat*) *q'*

**unfolding** *coin-usage-of-tpmf-alt-def q'-def* **by** *simp*

**have** *0:rel-pmf* ( $\geq$ ) (*coin-usage-of-tpmf p*) (*coin-usage-of-tpmf q*)

**unfolding** *cp cq map-pmf-def* **by** (*intro rel-pmf-bindI[OF 0]*) (*auto split:option.split*)

**show** *?thesis*

**unfolding** *atMost-def* **by** (*intro measure-Ici[OF 0] transp-on-ge*) (*simp add:reflp-def*)

**qed**

**lemma** *coin-usage-of-tpmf-mono-rev*:

**assumes** *ord-tpmf p q*

**shows** *measure* (*coin-usage-of-tpmf q*)  $\{x. x > k\} \leq \text{measure}$  (*coin-usage-of-tpmf p*)  $\{x. x > k\}$

(*is ?L*  $\leq$  *?R*)

**proof** –

**have** *0:UNIV* –  $\{x. x > k\} = \{..k\}$

**by** (*auto simp add:set-diff-eq set-eq-iff*)

**have**  $1 - ?R \leq 1 - ?L$

**using** *coin-usage-of-tpmf-mono[OF assms]*

**by** (*subst (1 2) measure-pmf.prob-compl[symmetric]*) (*auto simp:0*)

**thus** *?thesis*

**by** *simp*

**qed**

**lemma** *expected-coin-usage-of-tpmf*:

*expected-coin-usage-of-tpmf p* = ( $\sum k. \text{ennreal}$  (*measure* (*coin-usage-of-tpmf p*)  $\{x. x > \text{enat } k\}$ )) (*is ?L* = *?R*)

**proof** –

**have** *?L* = *integral<sup>N</sup>* (*measure-pmf* (*coin-usage-of-tpmf p*)) *ennreal-of-enat*

**unfolding** *expected-coin-usage-of-tpmf-def* **by** *simp*

**also have** ... = ( $\sum k. \text{emeasure}$  (*measure-pmf* (*coin-usage-of-tpmf p*))  $\{x. \text{enat } k < x\}$ )

**by** (*subst nn-integral-enat-function*) *auto*

also have ... = ?R  
 by (subst measure-pmf.emeasure-eq-measure) simp  
 finally show ?thesis  
 by simp  
 qed

lemma ord-tspmf-min: ord-tspmf (return-pmf None) p  
 unfolding ord-tspmf-def by (simp add: ord-spmf-reflI)

lemma ord-tspmf-refl: ord-tspmf p p  
 unfolding ord-tspmf-def by (simp add: ord-spmf-reflI)

lemma ord-tspmf-trans[trans]:  
 assumes ord-tspmf p q ord-tspmf q r  
 shows ord-tspmf p r

proof –  
 have 0:transp (ord-tspmf)  
 unfolding ord-tspmf-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  $\gg$  ( $\lambda x.$  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) ( $\lambda x.$  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: fst ' Y  $\neq$  {} **and** Y2: snd ' Y  $\neq$  {} **by** auto

**have** lub-spmf (fst ' Y) = lub-spmf (tspmf-of-ra ' 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** ... = tspmf-of-ra (lub-ra (snd ' Y))  
 by (intro tspmf-of-ra-lub[symmetric] chain2)  
**finally have** rel-tspmf-of-ra (lub-spmf (fst ' Y)) (lub-ra (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**  $\llbracket \text{mcont lub ord lub-spmf (ord-spmf (=)) } f; \text{mcont lub ord lub-ra ord-ra } g \rrbracket$   
 $\implies \text{ccpo.admissible lub ord } (\lambda x. \text{rel-tspmf-of-ra } (f \ x) \ (g \ x))$   
**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:

mcont lub-ra ord-ra lub-spmf (ord-spmf (=)) tspmf-of-ra  
 unfolding mcont-def monotone-def cont-def  
 by (auto simp: tspmf-of-ra-mono tspmf-of-ra-lub)

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

**context includes** lifting-syntax

**begin**

**lemma** fixp-rel-tspmf-of-ra-parametric[transfer-rule]:

**assumes** f:  $\bigwedge x. \text{mono-spmf } (\lambda f. F \ f \ x)$   
**and** g:  $\bigwedge x. \text{mono-ra } (\lambda f. G \ f \ x)$   
**and** param: ((A  $\implies$  rel-tspmf-of-ra)  $\implies$  A  $\implies$  rel-tspmf-of-ra) F G  
**shows** (A  $\implies$  rel-tspmf-of-ra) (spmf.fixp-fun F) (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 ==> rel-tspmf-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))
  (λx. (A ==> rel-tspmf-of-ra) (fst x) (snd x))
  unfolding rel-fun-def
  by(rule admissible-all admissible-imp cont-intro)+
have 0:tspmf-of-ra (lub-ra {}) = return-pmf None
  using tspmf-of-ra-lub[where A={}]
  by (simp add: Complete-Partial-Order.chain-def)
show (A ==> rel-tspmf-of-ra) (λ-. lub-spmf {}) (λ-. lub-ra {})
  by (auto simp: rel-fun-def rel-tspmf-of-ra-def 0)
show (A ==> rel-tspmf-of-ra) (F f) (G g) if (A ==> rel-tspmf-of-ra) f g for f g
  using that by(rule rel-funD[OF param])
qed

lemma return-ra-transfer[transfer-rule]: ((=) ==> 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]:
  (rel-tspmf-of-ra ==> ((=) ==> rel-tspmf-of-ra) ==> 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 ... = distr  $\mathcal{B}$   $\mathcal{D}$  (map-option snd  $\circ$  (map-option fst  $\circ$  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 ... = distr  $\mathcal{B}$   $\mathcal{D}$  (λx. ?p x  $\gg$  (λxa. consumed-bits ?p x))
    unfolding track-random-bits-def by (simp add: comp-def map-option-bind case-prod-beta)
  also have ... = distr  $\mathcal{B}$   $\mathcal{D}$  (λx. 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 ... = 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**  $0:\text{map-spmf snd (tspmfmf-of-ra p) = coin-usage-of-ra-aux p}$   
**using** *measure-pmf-inject* **by** *auto*  
**show** *?thesis*  
**unfolding** *coin-usage-of-tspmfmf-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-tspmfmf-correct*:  
 $\text{expected-coin-usage-of-tspmfmf (tspmfmf-of-ra p) = expected-coin-usage-of-ra p}$   
**unfolding** *expected-coin-usage-of-tspmfmf-def coin-usage-of-tspmfmf-correct*  
 $\text{expected-coin-usage-of-ra-def}$  **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].<sup>3</sup>

**lemma** *floor-le-ceil-minus-one*:  
**fixes**  $x y :: \text{real}$   
**shows**  $x < y \implies \lfloor x \rfloor \leq \lceil y \rceil - 1$   
**by** *linarith*

**lemma** *combine-spmfmf-set-coin-spmfmf*:  
**fixes**  $f :: \text{nat} \Rightarrow 'a \text{ spmf}$   
**fixes**  $k :: \text{nat}$   
**shows**  $\text{pmfmf-of-set } \{..<2^k\} \ggg (\lambda l. \text{coin-spmfmf } \ggg (\lambda b. f (2 * l + \text{of-bool } b))) =$   
 $\text{pmfmf-of-set } \{..<2^{k+1}\} \ggg f$  (**is**  $?L = ?R$ )

**proof** –

**let**  $?f = (\lambda (l::\text{nat}, b). 2 * l + \text{of-bool } b)$   
**let**  $?coin = \text{pmfmf-of-set (UNIV :: bool set)}$

**have**  $[\text{simp}]: \{..<(2::\text{nat})^k\} \neq \{\}$   
**by** (*simp add: lessThan-empty-iff*)

**have**  $\text{bij:bij-betw } ?f (\{..<2^k\} \times \text{UNIV}) \{..<2^{k+1}\}$   
**by** (*intro bij-betwI[where g=( $\lambda x. (x \text{ div } 2, \text{ odd } x)$ )]*) *auto*

**have**  $\text{pmfmf (pair-pmfmf (pmfmf-of-set } \{..<2^k\}) ?coin) x =}$   
 $\text{pmfmf (pmfmf-of-set } (\{..<2^k\} \times \text{UNIV})) x$  **for**  $x :: \text{nat} \times \text{bool}$   
**by** (*cases x*) (*simp add:pmfmf-pair indicator-def*)  
**hence**  $0:\text{pair-pmfmf (pmfmf-of-set } \{..<(2::\text{nat})^k\}) ?coin = \text{pmfmf-of-set } (\{..<2^k\} \times \text{UNIV})$   
**by** (*intro pmfmf-eqI*) *simp*

**have**  $\text{map-pmfmf } ?f (\text{pmfmf-of-set } (\{..<2^k\} \times \text{UNIV})) = \text{pmfmf-of-set } (?f ' (\{..<2^k\} \times \text{UNIV}))$   
**using** *bij-betw-imp-inj-on[OF bij]* **by** (*intro map-pmfmf-of-set-inj*) *auto*  
**also have**  $\dots = \text{pmfmf-of-set } \{..<2^{k+1}\}$   
**by** (*intro arg-cong[where f=pmfmf-of-set]*) *bij-betw-imp-surj-on[OF bij]*  
**finally have**  $1:\text{map-pmfmf } ?f (\text{pmfmf-of-set } (\{..<2^k\} \times \text{UNIV})) = \text{pmfmf-of-set } \{..<2^{k+1}\}$   
**by** *simp*

---

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

**have**  $?L = \text{pmf-of-set } \{..<2^k\} \gg= (\lambda l. ?\text{coin} \gg= (\lambda b. f (2 * l + \text{of-bool } b)))$   
**unfolding**  $\text{spmfm-of-set-def bind-spmfm-def spmfm-of-pmf-def}$  **by**  $(\text{simp add:bind-map-pmf})$   
**also have**  $... = \text{pair-pmf } (\text{pmf-of-set } \{..<2^k\}) ?\text{coin} \gg= (\lambda(l,b). f (2 * l + \text{of-bool } b))$   
**unfolding**  $\text{pair-pmf-def}$  **by**  $(\text{simp add:bind-assoc-pmf bind-return-pmf})$   
**also have**  $... = \text{map-pmf } (\lambda(l,b). 2 * l + \text{of-bool } b) (\text{pmf-of-set } (\{..<2^k\} \times \text{UNIV})) \gg= f$   
**unfolding**  $0 \text{ bind-map-pmf}$  **by**  $(\text{simp add:comp-def case-prod-beta}^\wedge)$   
**also have**  $... = ?R$   
**unfolding**  $1$  **by**  $\text{simp}$   
**finally show**  $?thesis$  **by**  $\text{simp}$   
**qed**

**lemma** *count-ints-in-range:*

*real*  $(\text{card } \{x. \text{of-int } x \in \{u..v\}\}) \geq v - u - 1$  (**is**  $?L \geq ?R$ )

**proof** (*cases*  $u \leq v$ )

**case** *True*

**have**  $0 : \text{of-int } x \in \{u..v\} \longleftrightarrow x \in \{[u]..[v]\}$  **for**  $x$  **by**  $\text{simp linarith}$

**have**  $v - u - 1 \leq [v] - [u] + 1$  **using** *True* **by**  $\text{linarith}$

**also have**  $... = \text{real } (\text{nat } ([v] - [u] + 1))$  **using** *True* **by**  $(\text{intro of-nat-nat[symmetric]}) \text{ linarith}$

**also have**  $... = \text{card } \{[u]..[v]\}$  **by**  $\text{simp}$

**also have**  $... = ?L$

**unfolding**  $0$  **by**  $(\text{intro arg-cong[where } f=\text{real}] \text{ arg-cong[where } f=\text{card}]) \text{ auto}$

**finally show**  $?thesis$  **by**  $\text{simp}$

**next**

**case** *False*

**hence**  $v - u - 1 \leq 0$  **by**  $\text{simp}$

**thus**  $?thesis$  **by**  $\text{simp}$

**qed**

**partial-function** (*random-alg*)  $\text{dice-roll-step-ra} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{int random-alg}$

**where**  $\text{dice-roll-step-ra } l \ h = ($

*if*  $[l] = [l+h] - 1$  *then*

*return-ra*  $[l]$

*else*

*do*  $\{ b \leftarrow \text{coin-ra}; \text{dice-roll-step-ra } (l + (h/2) * \text{of-bool } b) (h/2) \}$

$)$

**definition**  $\text{dice-roll-ra } n = \text{map-ra nat } (\text{dice-roll-step-ra } 0 (\text{of-nat } n))$

**partial-function** (*spmfm*)  $\text{drs-tspmfm} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{int tspmfm}$

**where**  $\text{drs-tspmfm } l \ h = ($

*if*  $[l] = [l+h] - 1$  *then*

*return-tspmfm*  $[l]$

*else*

*do*  $\{ b \leftarrow \text{coin-tspmfm}; \text{drs-tspmfm } (l + (h/2) * \text{of-bool } b) (h/2) \}$

$)$

**definition**  $\text{dice-roll-tspmfm } n = \text{drs-tspmfm } 0 (\text{of-nat } n) \gg= (\lambda x. \text{return-tspmfm } (\text{nat } x))$

**lemma**  $\text{drs-tspmfm}: \text{drs-tspmfm } l \ u = \text{tspmfm-of-ra } (\text{dice-roll-step-ra } l \ u)$

**proof** –

**include** *lifting-syntax*

**have**  $((=) ==> (=) ==> \text{rel-tspmfm-of-ra}) \text{ drs-tspmfm dice-roll-step-ra}$

**unfolding**  $\text{drs-tspmfm-def dice-roll-step-ra-def}$

**apply**  $(\text{rule rel-funD[OF curry-transfer]})$

**apply**  $(\text{rule fixp-rel-tspmfm-of-ra-parametric[OF drs-tspmfm.mono dice-roll-step-ra.mono]})$

**by**  $\text{transfer-prover}$

**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**  $\text{coin-tspmf} \gg (\lambda b. \text{drs-tspmf } (l + (h/2) * \text{of-bool } b) (h/2)) \leq_R \text{drs-tspmf } l h$   
**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**  $\dots \leq \lceil l + h \rceil - 1$   
**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**  $1: \lceil l + h / 2 * \text{of-bool } b + h / 2 \rceil - 1 = \lfloor l \rfloor$  **for**  $b$   
**proof** –  
**have**  $\lceil l + h / 2 * \text{of-bool } b + h / 2 \rceil - 1 \leq \lceil l + h \rceil - 1$   
**using** *assms* **by** (*intro diff-mono ceiling-mono*) *auto*  
**also have**  $\dots = \lfloor l \rfloor$  **using** *True* **by** *simp*  
**finally have**  $\lceil l + h / 2 * \text{of-bool } b + h / 2 \rceil - 1 \leq \lfloor l \rfloor$  **by** *simp*  
**moreover have**  $\lfloor l \rfloor \leq \lceil l + h / 2 * \text{of-bool } b + h / 2 \rceil - 1$   
**using** *assms* **by** (*intro floor-le-ceil-minus-one*) *auto*  
**ultimately show** *?thesis* **by** *simp*  
**qed**

**have**  $3:\text{drs-tspmf } (l + (h/2) * \text{of-bool } b) (h/2) = \text{return-tspmf } \lfloor l \rfloor$  **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**  $\text{coins } k \equiv \text{pmf-of-set } \{..<(2::\text{nat})\hat{\sim}k\}$

**lemma** *dice-roll-step-tspmf-expand*:  
**assumes**  $h > 0$   
**shows**  $\text{coins } k \gg (\lambda l. \text{use-coins } k (\text{drs-tspmf } (\text{real } l * h) h)) \leq_R \text{drs-tspmf } 0 (h * 2^{\hat{\sim}k})$   
**using** *assms*

**proof** (*induction k arbitrary:h*)  
**case** 0  
**have**  $\{..<Suc\ 0\} = \{0\}$  **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** (*coins (k+1)  $\gg$  (\lambda l. use-coins (k+1) (drs-tspmf (real l\*h) h)) =*  
*coins k  $\gg$  (\lambda l. coin-spmf  $\gg$  (\lambda b. use-coins (k+1) (drs-tspmf (real (2\*l+ of-bool b) \* h) h))*)  
**by** (*intro combine-spmf-set-coin-spmf[symmetric]*)  
**also have** ... = *coins k  $\gg$  (\lambda l. use-coins (k+1) (coin-spmf  $\gg$*   
*(\lambda b. drs-tspmf (real l\* (2\*h) + h \* of-bool b) h))*)  
**unfolding** *use-coins-def map-spmf-conv-bind-spmf* **by** (*simp add:algebra-simps*)  
**also have** ... = *coins k  $\gg$  (\lambda l. use-coins k (coin-tspmf  $\gg$*   
*(\lambda b. drs-tspmf (real l\* (2\*h) + h \* of-bool b) h))*)  
**unfolding** *coin-tspmf-split use-coins-add* **by** *simp*  
**also have** ... = *coins k  $\gg$  (\lambda l. use-coins k (coin-tspmf  $\gg$*   
*(\lambda b. drs-tspmf (real l\* (2\*h) + ((2\*h)/2) \* of-bool b) ((2\*h)/2)))*)  
**using** *Suc(2)* **by** *simp*  
**also have** ...  $\leq_R$  *coins k  $\gg$  (\lambda l. use-coins k (drs-tspmf (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** ...  $\leq_R$  *drs-tspmf 0 ((2\*h)\*2^k)*  
**using** *Suc(2)* **by** (*intro Suc(1)*) *auto*  
**also have** ... = *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 :: nat*  
**assumes** *h > (0::real)*  
**defines** *f  $\equiv$  (\lambda l. if [l\*h]=[l+1]\*h]-1 then Some ([l\*h],k) else None)*  
**shows** *map-pmf f (coins k)  $\leq_R$  drs-tspmf 0 (h\*2^k)* (**is** ?*L*  $\leq_R$  ?*R*)

**proof** –

**have** ?*L* = *coins k  $\gg$*   
*(\lambda l. use-coins k (if [real l\*h]=[l+1]\*h]-1 then return-tspmf [l\*h] else return-pmf 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** ...  $\leq_R$  *coins k  $\gg$  (\lambda l. use-coins k (drs-tspmf (real l\*h) h))*  
**by** (*subst drs-tspmf.simps, intro ord-tspmf-bind-pmf ord-tspmf-use-coins-2*)  
*(simp add:ord-tspmf-min ord-tspmf-refl algebra-simps)*  
**also have** ...  $\leq_R$  *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 :: nat*  
**assumes** *h > (0::real)*  
**defines** *f  $\equiv$  (\lambda l. if [l\*h]=[l+1]\*h]-1 then Some ([l\*h]) else None)*  
**shows** *ord-spmf (=) (map-pmf f (coins k)) (result (drs-tspmf 0 (h\*2^k)))*  
**(is** *ord-spmf - ?L ?R*)

**proof** –

**have** 0: ?*L* = *result (map-pmf (\lambda l. if [l\*h]=[l+1]\*h]-1 then Some ([l\*h],k) else None) (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  $f l = (\text{if } \lfloor l * h \rfloor = \lfloor (l+1) * h \rfloor - 1 \text{ then Some } \lfloor l * h \rfloor \text{ else None})$  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  $\text{real } x \leq (\text{real } j+1) * 2^k / n - 1$  using *that* by *simp*

also have  $\dots \leq \text{real } n * 2^k / n - 1$

using *assms* by (intro *diff-mono divide-right-mono mult-right-mono*) *auto*

also have  $\dots \leq 2^k - 1$  by *simp*

finally have  $\text{real } x \leq 2^k - 1$  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  $g = \text{nat}$ ]) *force+*

thus *?thesis*

using *bij-betw-imp-surj-on* by *auto*

qed

have 1:  $\text{real } j * 2^k / n \geq 0$  by *auto*

have 3:  $\lfloor \text{real } l * h \rfloor \leq \lfloor \text{real } (l+1) * h \rfloor - 1$  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  $\dots = (1/n - ?L) * 2^k$  by (subst *powr-realpow*) *auto*

finally have  $2 < (1/n - ?L) * 2^k$  by *simp*

hence  $?L < 1/n - 2 / 2^k$  by (*simp add:field-simps*)

also have  $\dots = (((\text{real } j+1) * 2^k / n - 1) - (\text{real } j * 2^k / n) - 1) / 2^k$

using *n-gt-0* by (*simp add:field-simps*)

also have  $\dots \leq \text{real } (\text{card } \{x. \text{of-int } x \in \{\text{real } j * 2^k / n .. (\text{real } j+1) * 2^k / n - 1\}\}) / 2^k$   
 by (intro *divide-right-mono count-ints-in-range*) *auto*

also have  $\dots = \text{real } (\text{card } (\text{int } \{x. \text{real } x \in \{\text{real } j * 2^k / n .. (\text{real } j+1) * 2^k / n - 1\}\})) / 2^k$

using *order-trans[OF 1]* by (subst 2) *auto*

also have  $\dots = \text{real } (\text{card } \{x. \text{real } x \in \{\text{real } j * 2^k / n .. (\text{real } j+1) * 2^k / n - 1\}\}) / 2^k$

by (subst *card-image*) *auto*

also have  $\dots = \text{real } (\text{card } \{x. x < 2^k \wedge \text{real } x \in \{\text{real } j * 2^k / n .. (\text{real } j+1) * 2^k / n - 1\}\}) / 2^k$

using 0 by (intro *arg-cong* [where  $f = \lambda x. \text{real } (\text{card } x) / 2^k$ ]) *auto*

also have  $\dots = \text{real } (\text{card } \{l. l < 2^k \wedge \text{real } j \leq \text{real } l * h \wedge (1 + \text{real } l) * h \leq \text{real } j+1\}) / 2^k$



**using** *assms* **unfolding** *h-def*  
**by** (*intro arg-cong*[**where**  $f = \lambda x. \text{real } (\text{card } x) / 2^k$ ] *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**  $f = \text{measure}$ ] *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**  $f = \text{measure}$ ] *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**  $f = \text{measure}$ ] *refl Collect-cong*) *auto*  
**also have**  $\dots \leq \text{spmf } (\text{result } (\text{drs-tspmf } 0 \text{ (} h * 2^k \text{)})) j$   
**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**  $\text{weight-spmf } (\text{spmf-of-set } \{0..<\text{int } n\}) \geq \text{weight-spmf } (\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n)))$   
**using** *assms* **unfolding** *weight-spmf-of-set*  
**by** (*simp add:lessThan-empty-iff weight-spmf-le-1*)  
**moreover have**  $\text{spmf } (\text{spmf-of-set } \{0..<\text{int } n\}) x \leq \text{spmf } (\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n))) x$   
**for**  $x$   
**proof** (*cases*  $x < n \wedge x \geq 0$ )  
**case** *True*  
**hence**  $\text{spmf } (\text{spmf-of-set } \{0..<\text{int } n\}) x = 1/n$   
**unfolding** *spmf-of-set* **by** *auto*  
**also have**  $\dots \leq \text{spmf } (\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n))) (\text{of-nat } (\text{nat } x))$   
**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**  $\text{spmf } (\text{spmf-of-set } \{0..<\text{int } n\}) x = 0$   
**unfolding** *spmf-of-set* **by** *auto*  
**then show** *?thesis* **by** *simp*  
**qed**  
**hence**  $\text{ord-spmf } (=) (\text{spmf-of-set } \{0..<\text{int } n\}) (\text{result } (\text{drs-tspmf } 0 \text{ (of-nat } n)))$   
**by** (*intro ord-pmf-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** *bij:bij-betw*  $\text{nat } \{0..<\text{int } n\} \{..<n\}$   
**by** (*intro bij-betwI*[**where**  $g = \text{int}$ ]) *auto*

**have**  $?L = \text{map-spmf nat } (\text{spmf-of-set } \{0..<\text{int } n\})$   
**unfolding**  $\text{dice-roll-tspmf-def dice-roll-correct-aux}[OF \text{ assms}] \text{ result-bind result-return}$   
 $\text{map-spmf-conv-bind-spmf}$  **by**  $\text{simp}$   
**also have**  $\dots = \text{spmf-of-set } (\text{nat } \{0..<\text{int } n\})$   
**by**  $(\text{intro map-spmf-of-set-inj-on inj-onI}) \text{ auto}$   
**also have**  $\dots = ?R$   
**using**  $\text{bij-betw-imp-surj-on}[OF \text{ bij}]$  **by**  $(\text{intro arg-cong}[\text{where } f = \text{spmf-of-set}]) \text{ auto}$   
**finally show**  $?L = ?R$  **by**  $\text{simp}$   
**thus**  $\text{spmf-of-ra } (\text{dice-roll-ra } n) = ?R$   
**using**  $\text{spmf-of-tspmf dice-roll-ra-tspmf}$  **by**  $\text{metis}$   
**qed**

**lemma**  $\text{dice-roll-consumption-bound}$ :

**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**  $\text{assms}$  **unfolding**  $h\text{-def}$   
**by**  $(\text{intro divide-pos-pos}) \text{ auto}$   
**have**  $0 : \text{real } n = h * 2^k$   
**unfolding**  $h\text{-def}$  **by**  $\text{simp}$

**have**  $1 : \lfloor \text{real } l * h \rfloor < \lceil (\text{real } l + 1) * h \rceil$  **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**  $(\text{intro floor-le-ceil-minus-one}) \text{ force}$   
**hence**  $\lfloor \text{real } l * h \rfloor < \lceil (\text{real } l + 1) * h \rceil - 1$   
**using**  $\text{that}$  **by**  $\text{simp}$   
**also have**  $\dots \leq \lfloor (\text{real } l + 1) * h \rfloor$   
**by**  $\text{linarith}$   
**finally show**  $?thesis$  **by**  $\text{simp}$

**qed**

**have**  $?L = \text{measure } (\text{coin-usage-of-tspmf } (\text{drs-tspmf } 0\ n)) \{x. x > \text{enat } k\}$   
**unfolding**  $\text{dice-roll-tspmf-def coin-usage-of-tspmf-bind-return}$  **by**  $\text{simp}$   
**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**  $(\text{intro coin-usage-of-tspmf-mono-rev dice-roll-step-tspmf-approx } h\text{-gt-0})$   
**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**  $(\text{simp add:vimage-def f-def algebra-simps split:option.split})$   
**also have**  $\dots \leq \text{measure } (\text{coins } k) \{l. \lfloor \text{real } l * h \rfloor < \lceil (\text{real } l + 1) * h \rceil\}$   
**using**  $1$  **by**  $(\text{intro measure-pmf.finite-measure-mono subsetI}) (\text{simp-all})$   
**also have**  $\dots = (\int l. \text{indicator } \{l. \lfloor \text{real } l * h \rfloor < \lceil (\text{real } l + 1) * h \rceil\} l\ \partial(\text{coins } k))$   
**by**  $\text{simp}$   
**also have**  $\dots = (\sum j < 2^k. \text{indicat-real } \{l. \lfloor \text{real } l * h \rfloor < \lceil (\text{real } l + 1) * h \rceil\} j * \text{pmf } (\text{coins } k) j)$   
**by**  $(\text{intro integral-measure-pmf-real}[\text{where } A = \{.. < 2^k\}]) (\text{simp-all add:lessThan-empty-iff})$   
**also have**  $\dots = (\sum l < 2^k. \text{of-bool } (\lfloor \text{real } l * h \rfloor < \lceil (\text{real } l + 1) * h \rceil)) / 2^k$   
**by**  $(\text{simp add:lessThan-empty-iff indicator-def flip:sum-divide-distrib})$   
**also have**  $\dots \leq (\sum l < 2^k. \text{of-int } [\text{real } (\text{Suc } l) * h] - \text{of-int } [\text{real } l * h]) / 2^k$   
**using**  $h\text{-gt-0 int-less-real-le}$   
**by**  $(\text{intro divide-right-mono sum-mono}) (\text{auto intro:floor-mono simp:algebra-simps})$   
**also have**  $\dots = \text{of-int } [2^k * h] / 2^k$

by (subst sum-lessThan-telescope) simp  
 also have ... = real n / 2<sup>k</sup>  
 unfolding h-def by simp  
 finally show ?thesis  
 by simp  
 qed

lemma dice-roll-expected-consumption-aux:

assumes  $n > (0::nat)$

shows  $expected\text{-}coin\text{-}usage\text{-}of\text{-}tspmf (dice\text{-}roll\text{-}tspmf\ n) \leq \log 2\ n + 2$  (is ?L ≤ ?R)

proof –

define  $k0$  where  $k0 = nat \lceil \log 2\ n \rceil$

define  $\delta$  where  $\delta = \log 2\ n - \lceil \log 2\ n \rceil$

have 0:  $ennreal (measure (coin\text{-}usage\text{-}of\text{-}tspmf (dice\text{-}roll\text{-}tspmf\ n)) \{x. x > enat\ k\}) \leq$   
 $ennreal (min (real\ n/2^k) 1)$  (is ?L1 ≤ ?R1) for  $k$   
 by (intro iffD2[OF ennreal-le-iff] min.boundedI dice-roll-consumption-bound[OF assms]) auto

have 1[simp]:  $(2::ennreal)^k < Orderings.top$  for  $k::nat$   
 using ennreal-numeral-less-top power-less-top-ennreal by blast

have  $(\sum k. ennreal ((1/2)^k)) = ennreal (\sum k. ((1/2)^k))$   
 by (intro suminf-ennreal2) auto

also have ... =  $ennreal\ 2$

by (subst suminf-geometric) simp-all

finally have 2:  $(\sum k. ennreal ((1/2)^k)) = ennreal\ 2$   
 by simp

have  $real\ n \geq 1$

using assms by simp

hence 3:  $\log 2 (real\ n) \geq 0$

by simp

have  $real\text{-}of\text{-}int \lceil \log 2 (real\ n) \rceil \leq 1 + \log 2 (real\ n)$

by linarith

hence 4:  $\delta + 1 \in \{0..1\}$

unfolding  $\delta$ -def by (auto simp: algebra-simps)

have  $twop\text{-}conv: convex\text{-}on\ UNIV (\lambda x. 2\ powr (x::real))$

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 =  $(\sum k. ennreal (measure (coin\text{-}usage\text{-}of\text{-}tspmf (dice\text{-}roll\text{-}tspmf\ n)) \{x. x > enat\ k\}))$   
 unfolding expected-coin-usage-of-tspmf by simp

also have ... ≤  $(\sum k. ennreal (min (real\ n/2^k) 1))$

by (intro suminf-le summableI 0)

also have ... =  $(\sum k. ennreal (min (real\ n/2^{(k+k0)}) 1)) + (\sum k < k0. ennreal (min (real\ n/2^k)$

1))

by (intro suminf-offset summableI)

also have ... ≤  $(\sum k. ennreal (real\ n/2^{(k+k0)})) + (\sum k < k0. 1)$

by (intro add-mono suminf-le summableI sum-mono iffD2[OF ennreal-le-iff]) auto

also have ... =  $(\sum k. ennreal (real\ n / 2^k) * ennreal ((1/2)^k)) + of\text{-}nat\ k0$

by (intro suminf-cong arg-cong2[where  $f=(+)$ ])

(simp-all add: ennreal-mult[symmetric] power-add divide-simps)

also have ... =  $ennreal (real\ n / 2^{k0}) * (\sum k. ennreal ((1/2)^k)) + ennreal (real\ k0)$

```

    unfolding ennreal-of-nat-eq-real-of-nat by simp
  also have ... = ennreal (real n / 2k0 * 2 + real k0)
    unfolding 2 by (subst ennreal-mult[symmetric]) simp-all
  also have ... = ennreal (real n / 2powr k0 * 2 + real k0)
    by (subst powr-realpow) auto
  also have ... = ennreal (real n / 2powr [log 2 n] * 2 + real k0)
    using 3 unfolding k0-def by (subst of-nat-nat) auto
  also have ... = ennreal (real n / 2powr (log 2 n - δ) * 2 + real k0)
    unfolding δ-def by simp
  also have ... = ennreal (2powr δ * 2powr 1 + real k0)
    using assms unfolding powr-diff by (subst powr-log-cancel) auto
  also have ... ≤ ennreal (1+(δ+1) + real k0)
    using 4 unfolding powr-add[symmetric]
    by (intro iffD2[OF ennreal-le-iff] add-mono 5) auto
  also have ... = ?R
    using 3 unfolding δ-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-coin-usage-of-ra (dice-roll-ra n) ≤ log 2 n + 2 (is ?L ≤ ?R)
proof -
  have ?L = expected-coin-usage-of-tspmf (tspmf-of-ra (dice-roll-ra n))
    unfolding expected-coin-usage-of-tspmf-correct[symmetric] by simp
  also have ... = expected-coin-usage-of-tspmf (dice-roll-tspmf n)
    unfolding dice-roll-ra-tspmf by simp
  also have ... ≤ ?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-mult = 6364136223846793005
definition pcg-increment :: 64 word
  where pcg-increment = 1442695040888963407

```

```

fun pcg-rotr :: 32 word ⇒ nat ⇒ 32 word

```

```

where pcg-rotr x r = Bit-Operations.or (drop-bit r x) (push-bit (32-r) x)

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

```

Based on [9, Section 6.3.1]:

```

fun pcg-get :: 64 word ⇒ 32 word
  where pcg-get state =
    (let count = unsigned (drop-bit 59 state);  

     x = xor (drop-bit 18 state) state  

     in pcg-rotr (ucast (drop-bit 27 x)) count)

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

```

```

definition to-bits :: 32 word ⇒ bool list
  where to-bits x = map (λk. 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 fixp-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-reflI 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* = *spmf-of-ra* (*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* = *pmf-of-ra* (*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* = *do* {

*b*  $\leftarrow$  *bernoulli-ra* *p*;

*if* *b* *then* *return-ra* 0 *else* *map-ra* ((+) 1) (*geometric-ra* *p*)

}

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

**lemma** *geometric-ra-correct*: *spmf-of-ra* (*geometric-ra* *x*) = *geometric-spmf* *x*

**proof** –

**include** *lifting-syntax*

**have** ((=) ==> *rel-spmf-of-ra*) *geometric-spmf* *geometric-ra*

**unfolding** *geometric-ra-def geometric-spmf-def*

```

apply (rule fixp-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
  replicate-ra 0 f = return-ra [] |
  replicate-ra (Suc n) f = do { xh  $\leftarrow$  f; xt  $\leftarrow$  replicate-ra n f; return-ra (xh#xt) }

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

```

```

lemma replicate-ra-correct: spmf-of-ra (replicate-ra n f) = replicate-spmf n (spmof-of-ra f)
by (induction n) (auto simp :spmof-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:spmof-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:spmof-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.