Executable Randomized Algorithms

Emin Karayel and Manuel Eberl

June 29, 2023

Abstract

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

Contents

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

1 Introduction

In Isabelle, randomized algorithms are usually represented using probability mass functions (PMFs). (These are distributions on the discrete \( \sigma \)-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 maybe an SPMF (a finite measure space with total measure less than 1). (Internally these are just PMFs over the `option` type, where `None` represents non-termination.) If a randomized algorithm terminates almost surely, the weight of the SPMF will be 1.

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

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

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

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

this is shown in SUP-emasure-incseq.

It is possible to generalize that to arbitrary chains \(^1\) of open sets for some measures without
the restriction of countability, such measures are called \(\tau\)-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 = (\(\forall x\ y. x \in F \land y \in F \rightarrow op\ x\ y \in F\))

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::\(\text{\{second-countable-topology, complete-space\}}\) measure
  assumes sb: \(\text{sets M = sets borel}\)
  assumes \(\text{fin: emeasure M (space M) \neq \infty}\)
  assumes of: \(\land a\ a \in A \implies \text{open a}\)
  assumes ud: \(\text{op-stable} (\cup) A\)
  shows emeasure M (\(\bigcup A\)) = (SUP a \in A. emeasure M a) (is \(?L = \?R\))
  proof (cases \(A \neq \{\}\))
    case True
    have \(\text{open (\bigcup A)}\) using of by auto
    hence \(\bigcup A \in \text{sets borel}\) by simp
    hence \(\text{asets: \(\bigcup A \in \text{sets M}\)}\) using assms(1) by simp
    have \(0:a \in \text{sets borel if } a \in A \text{ for } a\)
      using of that by simp
    have \(1: \bigcup T \in A \text{ if } \text{finite } T\ T \neq \{\} \ T \subseteq A \text{ for } T\)
      using that op-stableD[OF ud] by (induction T rule:finite-ne-induct) auto
    have \(2:\text{emeasure M K \leq } \text{\(}\text{\(?R\)) if } \text{K-def: compact K K \subseteq \bigcup A \text{ for K}\)}
      proof (cases K \(\neq \{\}\))
        case True
        obtain T where T-def: K \(\subseteq \bigcup T T \subseteq A \text{ finite } T\)
          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 \(\leq \text{emeasure M t}\)
          using 0 sb t-in by (intro emeasure-mono) auto
        also have \(\ldots \leq \text{\(}\text{\(?R\))}\)
          using t-in by (intro cSup-upper) auto
        finally show \(?\text{thesis}\)

\(^1\)More generally families closed under pairwise unions.
by simp

next
  case False
  hence $K = \emptyset$ by simp
  thus $\exists \text{thesis}$ by simp
qed

have $L = (\sup K \in \{ K. \ K \subseteq \bigcup A \land \text{compact } K \}. \ \text{emeasure } M K)$
  using usets unfolding sb by (intro inner-regular[OF sb fin]) auto
also have $\ldots \leq R$
  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 $\exists \text{thesis}$ by auto

next
  case False
  thus $\exists \text{thesis}$ 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 $\exists \text{thesis}$ 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 $\exists \text{thesis}$ using that sup.absorb1[OF 0] by simp
    qed
  thus $\exists \text{thesis}$ unfolding op-stable_def by auto
qed

theorem tau-additivity:
  fixes $M :: \{ \text{second-countable-topology}, \text{complete-space} \}$ measure
  assumes sb: $\forall x. \text{open } x \Longrightarrow x \in \text{sets } M$
  assumes fin: $\text{emeasure } M (\text{space } M) \neq \infty$
  assumes of: $\forall a. a \in A \Longrightarrow \text{open } a$
  assumes ud: op-stable $(\cup) A$
  shows $\text{emeasure } M (\bigcup A) = (\sup a \in A. \text{emeasure } M a) \ (\text{is } L = R)$
proof –
  have UNIV $\in$ sets M
    using open-UNIV sb by auto
  hence space-M[simp]:space M = UNIV
    using sets.sets-into-space by blast
  have id-borel: $(\lambda x. x) \in M \rightarrow M$ borel
    using sb by (intro borel-measurable1) auto

  have open $(\bigcup A)$ using of by auto
  hence usets: $(\bigcup A) \in \text{sets borel}$ by simp
define $N$ where $N = \text{distr } M \text{ borel } (\lambda x. x)$
have sets-$N$: sets $N = \text{sets borel}$
  unfolding $N$-def by simp
have fin-$N$: $\text{emeasure } N \text{ (space } N) \neq \infty$
  using fin id-borel unfolding $N$-def
  by (subst emeasure-distr) auto
have $?L = \text{emeasure } N \big( \bigcup A \big)$
  unfolding $N$-def by (subst emeasure-distr[OF id-borel usets]) auto
also have ... = (SUP $a \in A. \text{emeasure } N a$)
  by (intro tau-additivity-aux sets-$N$ of ud fin-$N$) auto
also have ... = \big( \text{SUP } a \in A. \text{emeasure } M \big((\lambda x. x) - a \cap \text{space } M\big)\big)
  unfolding $N$-def using of
  by (intro arg-cong[where $f=\text{Sup}$] 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 $\forall i. x !! i = y !! i$
  shows $x = y$
proof
  have $x = \text{smap id } x$ by (simp add: stream.map-id)
  also have ... = $y$ using assms unfolding smap-alt by auto
  finally show $?thesis$ by simp
qed

Notation for the discrete $\sigma$-algebra:

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

bundle discrete-sigma-algebra-notation
begin
  notation discrete-sigma-algebra ($D$)
end

bundle no-discrete-sigma-algebra-notation
begin
  no-notation discrete-sigma-algebra ($D$)
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 ⇒ 'b :: countable
assumes $\forall x. x \neq d \implies f \cdot \{ x \} \cap \text{space } M \in \text{sets } M$
shows $f \in M \rightarrow_M \text{count-space } UNIV$
proof –
  define $A :: b \text{ set }$ where $A = (\lambda x. \{ x \})$ $\cdot$ $\text{UNIV}$
  have 0: $\text{sets } (\text{count-space } UNIV) = \sigma_{\text{sets } (\text{UNIV} :: \cdot b \text{ set})} A$
    unfolding $A$-def by (subst sigma-sets-singletons) auto
  have 1: $f \cdot \{ x \} \cap \text{space } M \in \text{sets } M$ for $x$
    proof (cases $x = d$)
      case True
      have $f \cdot \{ d \} \cap \text{space } M = \text{space } M - (\bigcup y \in \text{UNIV} - \{ d \}. f \cdot \{ y \} \cap \text{space } M)$
        by (auto simp add: set-eq-iff)
      also have \ldots $\in$ $\text{sets } M$
        using assms by (intro sets.compl-sets sets.countable-UN) auto
    finally show $\text{thesis}$
      using $\text{True}$ by simp
  next
    case False
    then show $\text{thesis}$ using assms by simp
  qed
hence $\forall y. y \in A \implies f \cdot y \cap \text{space } M \in \text{sets } M$
unfolding $A$-def by auto
thus $\text{thesis}$
  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 $\text{distr } (\text{stream-space } M) N (\lambda x. \text{s nth } x s n) = M$
proof –
  have $\text{distr } (\text{stream-space } M) N (\lambda x. \text{s nth } x s n) = \text{distr } (\text{stream-space } M) M (\lambda x. \text{s nth } x s n)$
    by (rule distr-cong) (use assms in auto)
  also have \ldots $= \text{distr } (\Pi_M \text{ UNIV } (\lambda i. M)) M (\lambda x. f n)$
    by (subst stream-space-eq-distr, subst distr-distr) (auto simp: to-stream-def o-def)
  also have \ldots $= M$
    by (intro distr-PiM-component prob-space-axioms) auto
finally show $\text{thesis}$
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 \text{s}. \ x \circlearrowleft \text{s}) \in \text{stream-space $M$} \rightarrow_{\text{M}} \text{stream-space $M$}$
proof –
  have $(\lambda \text{s}. \ x \circlearrowleft \text{s} !! n) \in (\text{stream-space $M$}) \rightarrow_{\text{M}} M$ for $n$
  proof (cases $n < \text{length} \ x$)
  case True
  have $(\lambda \text{s}. \ x \circlearrowleft \text{s} !! n) = (\lambda \text{s}. \ x ! n)$
    using True by simp
  also have ... $\in \text{stream-space $M$} \rightarrow_{\text{M}} M$
    by (intro measurable-const) auto
  finally show $\exists \text{thesis}$ by simp
next
  case False
  have $(\lambda \text{s}. \ x \circlearrowleft \text{s} !! n) = (\lambda \text{s}. \ text{s} !! (n - \text{length} \ x))$
    using False by simp
  also have ... $\in (\text{stream-space $M$}) \rightarrow_{\text{M}} M$
    by (intro measurable-snth)
  finally show $\exists \text{thesis}$ by simp
qed
thus $\exists \text{thesis}$
  by (intro measurable-stream-space2) auto
qed

lemma (in sigma-finite-measure) restrict-space-pair-lift:
  assumes $A' \in \text{sets $A$}$
  shows $\text{restrict-space $A$} A' \times_{\text{M}} M = \text{restrict-space} (A \times_{\text{M}} M) (A' \times \text{space $M$})$ (is $\exists \text{L} = \exists \text{R}$)
proof –
  let $?X = \{(\cap) (A' \times \text{space $M$}) \cdot \{a \times b | a \ b, a \in \text{sets $A$} \land b \in \text{sets $M$}\}\}$
  have $0: A' \subseteq \text{space $A$}$
    using assms sets.sets-into-space by blast
  have $?X \subseteq \{a \times b | a \ b, a \in \text{sets} (\text{restrict-space $A$} A') \land b \in \text{sets $M$}\}$
    proof (rule image-subsetI)
      fix $x \ assume x \in \{a \times b | a \ b, a \in \text{sets $A$} \land b \in \text{sets $M$}\}$
      then obtain $u \ v$ where $uv\text{-def}: x = u \times v \ u \in \text{sets $A$} v \in \text{sets $M$}$
        by auto
      have $1: u \cap A' \in \text{sets} (\text{restrict-space $A$} A')$
        using $uv\text{-def}(2)$ unfolding sets-restrict-space by auto
      have $v \subseteq \text{space $M$}$
        using $uv\text{-def}(3)$ sets.sets-into-space by auto
      hence $A' \times \text{space $M$} \cap x = (u \cap A') \times v$
        unfolding $uv\text{-def}(1)$ by auto
      also have ... $\in \{a \times b | a \ b, a \in \text{sets} (\text{restrict-space $A$} A') \land b \in \text{sets $M$}\}$
        using $1 uv\text{-def}(3)$ by auto
    finally show $A' \times \text{space $M$} \cap x \in \{a \times b | a \ b, a \in \text{sets} (\text{restrict-space $A$} A') \land b \in \text{sets $M$}\}$
      by simp
qed
moreover have $\{a \times b | a \ b, a \in \text{sets} (\text{restrict-space $A$} A') \land b \in \text{sets $M$}\} \subseteq ?X$
  proof (rule subsetI)
    fix $x \ assume x \in \{a \times b | a \ b, a \in \text{sets} (\text{restrict-space $A$} A') \land b \in \text{sets $M$}\}$
    then obtain $u \ v$ where $uv\text{-def}: x = u \times v \ u \in \text{sets} (\text{restrict-space $A$} A') v \in \text{sets $M$}$
      by auto
    have $x = (A' \times \text{space $M$}) \cap x$
      unfolding $uv\text{-def}(1)$ using $uv\text{-def}(2, 3)$ sets.sets-into-space
      by (intro Int-absorb1 [symmetric]) (auto simp add: sets-restrict-space)
moreover have $u \in \text{sets $A$}$ using $uv\text{-def}(2)$ assms unfolding sets-restrict-space by blast
hence \( x \in \{ a \times b \mid a, b \in \text{sets } A \land b \in \text{sets } M \} \)

unfolding \( \text{uv-def(1)} \) using \( \text{uv-def(3)} \) by simp
ultimately show \( x \in \exists X \)
by simp

qed
ultimately have \( 2: \exists X = \{ a \times b \mid a, b \in \text{sets } \{ \text{restrict-space } A A' \} \land b \in \text{sets } M \} \) by simp

have \( \exists R = \text{sigma-sets} (A' \times \text{space } M) (\cap) (A' \times \text{space } M) \cap \{ a \times b \mid a, b \in \text{sets } A \land b \in \text{sets } M \} \)

unfolding \( \text{sets-restrict-space sets-pair-measure using assms sets.sets-into-space} \)
by \( \text{(intro sigma-sets-Int sigma-sets.Basic) auto} \)
also have \( \ldots = \text{sets } (\text{restrict-space } A A' \otimes_M M) \)

unfolding \( \text{sets-pair-measure space-restrict-space Int-absorb2[OF 0] sets-restrict-space 2} \)
by auto

finally have \( \exists: \text{sets } (\text{restrict-space } (A \otimes_M M) (A' \times \text{space } M)) = \text{sets } (\text{restrict-space } A A' \otimes_M M) \)
by simp

have \( 4: \text{emeasure } (\text{restrict-space } A A' \otimes_M M) S = \text{emeasure } (\text{restrict-space } (A \otimes_M M) (A' \times \text{space } M)) \)
(is \( \exists L1 = \exists R1 \) if \( 5: S \in \text{sets } (\text{restrict-space } A A' \otimes_M M) \) for \( S \)

proof -

have \( \exists L1 = (\exists x. \text{emeasure } M (\exists x \in A' \times \text{space } A) \otimes \text{restrict-space } A A') \)
by \( \text{(intro emeasure-pair-measure-alt[OF that])} \)
also have \( \ldots = (\exists x \in A'. \text{emeasure } M (\exists x \in A') \otimes \text{restrict-space } A A') \)
using assms by \( \text{(intro nn-integral-restrict-space) auto} \)
also have \( \ldots = (\exists x. \text{emeasure } M (\exists x \in A') \otimes \text{restrict-space } A A') \)
using \( \exists L1 \) by \( \text{(intro nn-integral-cong) force} \)
also have \( \ldots = \text{emeasure } (A \otimes_M M) S \)
using assms by \( \text{(intro emeasure-restrict-space[symmetric])} \)
(auto simp add:3[symmetric] sets-restrict-space)
also have \( \ldots = \exists R1 \)
using assms that by \( \text{(intro emeasure-restrict-space[symmetric])} \)
(auto simp add:3[symmetric] sets-restrict-space)

finally show \( \exists \)thesis by simp

qed

show \( \exists \)thesis using \( 3, 4 \)
by \( \text{(intro measure-eqv) auto} \)

qed

lemma to-stream-comb-seq-eq:
\text{to-stream } (\text{comb-seq } \text{n } \text{x } \text{y}) = \text{stake } \text{n } (\text{to-stream } \text{x}) @\ldots - (\text{to-stream } \text{y})

unfolding \( \text{comb-seq-def to-stream-def} \)
by \( \text{(intro stream-eq-iff) simp} \)

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

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

lemma \( \text{(in prob-space) branch-stream-space:} \)
(\( \lambda(x, y). \text{stake } \text{n } x @\ldots - y \)) \in \text{stream-space } M \otimes_M \text{stream-space } M \rightarrow_M \text{stream-space } M
distr (stream-space $M \otimes M$ stream-space $M$) (stream-space $M$) $(\lambda(x,y). \text{stake } n \, x@-\, y)$
$= \text{stream-space } M \, (\text{is } ?L = ?R)$

proof

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

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

show 0;$(\lambda(x, y). \text{stake } n \, x@-\, y) \in ?T \otimes M \, ?T \otimes M \, ?T$
by simp

have $?L = \text{distr}$ $(\text{distr } ?S \otimes M \, \text{distr } ?T \otimes M \, \text{distr } ?T \otimes M \, \text{distr } ?T \otimes M \, \text{distr } ?T \otimes M \, \text{distr } ?T) \, (\lambda(x,y). \text{stake } n \, x@-\, y)$
by (subst (1 2) stream-space-eq-distr) simp
also have ... = distr $(\text{distr } ?S \otimes M \, \text{distr } ?S \otimes M \, \text{distr } ?T \otimes M \, \text{distr } ?T \otimes M \, \text{distr } ?T \otimes M \, \text{distr } ?T \otimes M \, \text{distr } ?T \otimes M \, \text{distr } ?T) \, (\lambda(x,y). \text{stake } n \, x@-\, y)$
using prob-space-imp-sigma-finite[OF prob-space-stream-space]
by (intro arg-cong2[where $f=\lambda(x y. \text{distr } x ?T y)]$ pair-measure-distr)
(simp-all flip:stream-space-eq-distr)
also have ... = distr $(?S \otimes M) \, ?T \, ?T \, ((\lambda(x, y). \text{stake } n \, x@-\, y) \circ (\lambda(x,y). \text{(to-stream } x,\text{to-stream } y)))$
by (intro distr-distr 0) (simp add: measurable-pair-iff)
also have ... = distr $(?S \otimes M) \, ?T \, ((\lambda(x,y). \text{stake } n \, \text{(to-stream } x \, @-\, \text{to-stream } y)))$
by (simp add: comp-def case-prod-beta)
also have ... = distr $(?S \otimes M) \, ?T \, \text{to-stream } \circ (\lambda(x,y). \text{comb-seq } n \, x \, y)$
using to-stream-comb-seq-eq[symmetric]
by (intro arg-cong2[where $f=\lambda(x y. \text{distr } x ?T y)]$ ext) auto
also have ... = distr $(?S \otimes M) \, ?S \, \text{to-stream } (\lambda(x,y). \text{comb-seq } n \, x \, y) \, ?T \otimes M \, \text{to-stream}$
by (intro distr-distr[symmetric] measurable-comb-seq) simp
also have ... = distr $?S \otimes M \, \text{to-stream}$
by (subst S.PiM-comb-seq) simp
also have ... = $?R$
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 $\text{bool stream}$. Nevertheless, we introduce it as a separate type to be able to introduce a topology and mark it as a lazy type for code-generation:

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

code-lazy-type coin-stream

primcorec from-coins :: coin-stream $\Rightarrow$ bool stream where
from-coins coins = chd coins ## (from-coins (ctl coins))

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

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

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

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

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

context
  includes lifting-syntax

begin

lemma bi-unique-rel-coins [transfer-rule]: bi-unique rel-coins
  unfolding rel-coins-def using inj-onD [OF bij-is-inj [OF bij-to-coins]]
  by (intro bi-uniqueI left-uniqueI right-uniqueI)

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)

lemma cnth-transfer [transfer-rule]: (rel-coins ===> (=) ===> (=)) snth cnth
  unfolding rel-coins-def cnth-def rel-fun-def by (auto simp: from-to-coins)

lemma cshift-transfer [transfer-rule]: ((=) ===> rel-coins ===> rel-coins) shift cshift
  unfolding rel-coins-def cshift-def rel-fun-def by (auto simp: from-to-coins)

lemma ctake-transfer [transfer-rule]: ((=) ===> rel-coins ===> (=)) stake ctake
  unfolding rel-coins-def ctake-def rel-fun-def by (auto simp: from-to-coins)

lemma cdrop-transfer [transfer-rule]: ((=) ===> rel-coins ===> 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 ===> (=)) shd chd
  unfolding rel-coins-def rel-fun-def by (auto simp: from-to-coins)

lemma ctl-transfer [transfer-rule]: (rel-coins ===> rel-coins) stl ctl
  unfolding rel-coins-def rel-fun-def by (auto simp: from-to-coins)

lemma cconst-transfer [transfer-rule]: ((=) ===> 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 ∀i. cnth x i = cnth y i
  shows x = y
  proof
    have (∀i. cnth x i = cnth y i) → 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 take-nth[simp]: \( m < n \implies \text{take } n \ s \ ! \ m = \text{nth} \ s \ m \)
by transfer (rule stake-nth)

lemma take-cdrop: \( \text{cshift} \ (\text{take } n \ s) \ (\text{cdrop} \ n \ s) = s \)
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]: \( \text{cshift} \ [] \ x s = x s \)
by transfer simp

lemma cshift-null[simp]: \( \text{cshift} \ 0 \ x s = [] \)
by transfer simp

lemma cshift-Suc[simp]: \( \text{cshift} \ (\text{Suc} \ n) \ s = \text{chd} \ s \ # \ \text{take} \ n \ (\text{ctl} \ s) \)
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]: \( \text{chd} \ (\text{cshift} \ x s \ s) = (\text{if } x s = [] \ \text{then} \ \text{chd} \ s \ \text{else} \ \text{hd} \ x s) \)
by transfer simp

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

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

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

lemma take-add[simp]: \( \text{take} \ m \ s \ @ \ \text{take} \ n \ (\text{cdrop} \ m \ s) = \text{take} \ (m + n) \ s \)
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: \( \text{cprefix} \ x \ y \iff (\forall i < \text{length} \ x. \ \text{cnth} \ y \ i = x ! i) \) (is \( ?L \iff ?R \))
proof -
  have \( ?L \iff \text{take} \ (\text{length} \ x) \ y = x \)
    unfolding cprefix-def by simp
  also have ... \iff (\forall i < \text{length} \ x. \ (\text{take} \ (\text{length} \ x) \ y) ! i = x ! i)
    by (simp add: list-eq-iff-nth-eq)
  also have ... \iff ?R
    by (intro all-cong) simp
  finally show \( ?\text{thesis} \) by simp
qed

A non-empty shift is not idempotent:

lemma empty-if-shift-idem:
  assumes \( \forall cs. \text{cshift} \ h \ cs = cs \)
  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
Infrastructure for building coin streams:

**primcorec**  
**cmap-iterate** :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ coin-stream
  
  where
  
  **cmap-iterate** m f s = Coin (m s) (**cmap-iterate** m f (f s))

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

**proof**  
(rule coin-stream.coinduct)

  **where**
  
  **R** = (λxs ys. (∃x. xs = **cmap-iterate** m f x ∧ ys = **to-coins** (**smap** m (**siterate** f x))))

  show (∃x. **cmap-iterate** m f s = **cmap-iterate** m f x ∧

  **to-coins** (**smap** m (**siterate** f s)) = **to-coins** (**smap** m (**siterate** f x))

  by (intro exl[where x=s] refl conj)
ultimately show \( shd\; xs = shd\; ys \land (\exists\; x\; n.\; stl\; xs = n \oplus (f\; x) \land stl\; ys = n \oplus (g\; x)) \)
by auto

qed

lemma build-coin-gen:
assumes \( \forall\; x.\; m\; x \neq [] \)
shows \( build-coin-gen\; m\; f\; s = to-coins\; \left( flat\; \left( smap\; m\; \left( siterate\; f\; s \right) \right) \right) \)

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

have liter: \( smap\; \left( hd \circ\; fst \right)\; \left( siterate\; ?g\; \left( bs,\; x \right) \right) = bs \oplus - \left( smap\; \left( hd \circ\; fst \right)\; \left( siterate\; ?g\; \left( m\; x,\; f\; x \right) \right) \right) \) if \( bs \neq [] \) for \( x\) \( bs\)
using that
proof (induction \( bs \) rule: list-nonempty-induct)
  case \( (\text{single}\; y) \)
  then show \(?case\) by (subst siterate.ctr simp)
next
  case \( (\text{cons}\; y\; ys) \)
  then show \(?case\) by (subst siterate.ctr (simp add: comp-def))
qed

ultimately have \( \exists\; n\; y.\; n \neq [] \land\)
\( smap\; \left( hd \circ\; fst \right)\; \left( siterate\; ?g\; \left( m\; x,\; f\; x \right) \right) = n \oplus - \left( smap\; \left( hd \circ\; fst \right)\; \left( siterate\; ?g\; \left( m\; y,\; f\; y \right) \right) \right) \land\)
\( flat\; \left( smap\; m\; \left( siterate\; f\; x \right) \right) = n \oplus - \left( flat\; \left( smap\; m\; \left( siterate\; f\; y \right) \right) \right) \) for \( x\)
by (intro exI [where \( x = m\; x \)] exI [where \( x = f\; x \)] conjI assms)

hence \( smap\; \left( hd \circ\; fst \right)\; \left( siterate\; ?g\; \left( m\; s',\; f\; s' \right) \right) = flat\; \left( smap\; m\; \left( siterate\; f\; s' \right) \right) \) for \( s'\)
by (rule build-coin-gen-aux [where \( f = (\lambda\; x.\; smap\; \left( hd \circ\; fst \right)\; \left( siterate\; ?g\; \left( m\; x,\; f\; x \right) \right) \) ])
thus \(?thesis\)
unfolding build-coin-gen-def cmap-iterate by simp

qed

Measure space for coin streams:

definition coin-space :: coin-stream measure
where \( coin-space = \text{embed-measure}\; \left( stream-space\; \left( \text{measure-pmf}\; \left( \text{pmf-of-set}\; \text{UNIV} \right) \right) \right)\) to-coins

bundle coin-space-notation
begin
  notation coin-space \( (B) \)
end

bundle no-coin-space-notation
begin
  no-notation coin-space \( (B) \)
end

unbundle coin-space-notation

lemma space-coin-space: \( space\; B = \text{UNIV} \)
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: \( B = \text{distr}\; \left( \text{stream-space}\; \left( \text{pmf-of-set}\; \text{UNIV} \right) \right)\) to-coins
unfolding coin-space-def by (intro embed-measure-eq-distr bij-is-inj[OF bij-to-coins])

lemma from-coins-measurable: from-coins ∈ B → M (stream-space (pmf-of-set UNIV))
unfolding coin-space-def by (intro measurable-embed-measure1) (simp add:from-coins)

lemma to-coins-measurable: to-coins ∈ (stream-space (pmf-of-set UNIV)) → M B
unfolding coin-space-def by (intro measurable-embed-measure2 bij-is-inj[OF bij-to-coins])

lemma chd-measurable: chd ∈ B → M D
proof –
have 0: chd (to-coins x) = shd x for x
  using chd-transfer unfolding rel-fun-def by auto
thus ?thesis
unfolding coin-space-def by (intro measurable-embed-measure1) simp
qed

lemma cnth-measurable: (λxs. cnth xs i) ∈ B → M D
unfolding coin-space-def cnth-def by (intro measurable-embed-measure1) (simp add:from-coins)

lemma B-eq-distr:
stream-space (pmf-of-set UNIV) = distr B (stream-space (pmf-of-set UNIV)) from-coins
(is ?L = ?R)
proof –
let ?S = stream-space (pmf-of-set UNIV)
have ?R = distr (distr ?S B to-coins) ?S from-coins
  using B-t-eq-distr by simp
also have ... = distr ?S ?S (from-coins ∘ to-coins)
  by (intro distr-distr from-coins-measurable from-coins-measurable)
also have ... = distr ?S ?S id
  unfolding id-def comp-def from-to-coins by simp
also have ... = ?L
  unfolding id-def by simp
finally show ?thesis by simp
qed

lemma B-t-finite: emeasure B (space B) = 1
proof –
let ?S = stream-space (pmf-of-set (UNIV::bool set))
have 1 = emeasure ?S (space ?S)
  by (intro prob-space.emeasure-space-1[symmetric] prob-space.prob-space-stream-space
  prob-space-measure-pmf)
also have ... = emeasure B (from-coins −‘ (space (stream-space (pmf-of-set UNIV))) ∩ space
  B)
  by (subst B-eq-distr) (intro emeasure-distr from-coins-measurable sets.top)
also have ... = emeasure B (space B)
  unfolding space-coin-space space-stream-space vinage-def by simp
finally show ?thesis by simp
qed

interpretation coin-space: prob-space coin-space
using B-t-finite by standard

lemma distr-shd: distr B D chd = pmf-of-set UNIV (is ?L = ?R)
proof –
have ?L = distr (stream-space (measure-pmf (pmf-of-set UNIV))) D (chd ∘ to-coins)
  by (subst B-t-eq-distr) (intro distr-distr to-coins-measurable chd-measurable)
also have \( \ldots = \text{distr} \) (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 \( \ldots = ?R \)
using coin-space.distr-stream-space-shd by auto
finally show ?thesis by simp
qed

lemma cshift-measurable: \( \text{cshift} \ x \in \mathcal{B} \rightarrow_{M} \mathcal{B} \)
proof –
have \( \text{(to-coins} \circ \text{cshift} \ x \circ \text{from-coins}) \in \mathcal{B} \rightarrow_{M} \mathcal{B} \)
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: \( \text{cdrop} \ x \in \mathcal{B} \rightarrow_{M} \mathcal{B} \)
proof –
have \( \text{(to-coins} \circ \text{sdrop} \ x \circ \text{from-coins}) \in \mathcal{B} \rightarrow_{M} \mathcal{B} \)
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: \( \text{ctake} \ k \in \mathcal{B} \rightarrow_{M} \mathcal{D} \)
proof –
have \( \text{stake} \ k \circ \text{from-coins} \in \mathcal{B} \rightarrow_{M} \mathcal{D} \)
by (intro measurable-comp[OF from-coins-measurable]) simp
thus ?thesis
unfolding ctake-def by (simp add: comp-def)
qed

lemma branch-coin-space:
\[ (\lambda(x, y). \text{cshift} \ (\text{ctake} \ n \ x) \ y) \in \mathcal{B} \otimes_{M} \mathcal{B} \rightarrow_{M} \mathcal{B} \]
distr \( \mathcal{B} \otimes_{M} \mathcal{B} \) \( \lambda(x,y). \text{cshift} \ (\text{ctake} \ n \ x) \ y \) = \( \mathcal{B} \) \( \text{(is } \mathcal{L} = ?R) \)
proof –
let \( ?M = \text{stream-space} \) (measure-pmf (pmf-of-set UNIV))
let \( ?f = (\lambda(x,y). \text{stake} \ n \ x \ @ y) \)
let \( ?g = \text{map-prod from-coins from-coins} \)
have \( (\lambda(x, y). \text{cshift} \ (\text{ctake} \ n \ x) \ y) = \text{to-coins} \circ (?f \circ ?g) \)
by (simp add: comp-def cshift-def ctake-def case-prod-beta)
also have \( \ldots \in \mathcal{B} \otimes_{M} \mathcal{B} \rightarrow_{M} \mathcal{B} \)
by (intro measurable-comp[OF from-coins-measurable] measurable-comp[OF - to-coins-measurable]
shift-measurable) auto
thus ?thesis
unfolding map-prod-def using prob-space-measure-pmf
finally show \( (\lambda(x, y). \text{cshift} \ (\text{ctake} \ n \ x) \ y) \in \mathcal{B} \otimes_{M} \mathcal{B} \rightarrow_{M} \mathcal{B} \)
by simp

have \( \text{distr} \) \( \mathcal{B} \otimes_{M} \mathcal{B} \) \( \otimes (?M \otimes_{M} ?M) \) \( ?g = \) \( (\text{distr} ?M \) from-coins \otimes_{M} \text{distr} ?M \) from-coins
unfolding map-prod-def using prob-space-measure-pmf
by (intro pair-measure-distr[where N=(?M \otimes_{M} ?M)]) (auto intro:
prob-space-imp-sigma-finite prob-space.prob-space-stream-space simp:B-eq-distr[symmetric])
also have \( \ldots = ?M \otimes_{M} ?M \)
unfolding B-eq-distr[symmetric] by simp
finally have \( 0: \text{distr} \) \( \mathcal{B} \otimes_{M} \mathcal{B} \) \( \otimes (?M \otimes_{M} ?M) \) \( ?g = \) \( (\text{distr} \) \( \mathcal{B} \otimes_{M} ?M) \)
by simp

have \( L = \text{distr} (B \otimes M) B \) (to-coins \( \circ f \circ g \))
unfolding shift-def ctake-def by (simp add:comp-def map-prod-def case-prod-beta)
also have ... = distr (distr (B \otimes M) B (to-coins \( \circ f \))
also have ... = distr (distr (M \otimes M) M \otimes M (to-coins \( \circ f \))
by (intro distr-distr[symmetric] to-coins-measurable) simp
also have ... = \( M \) B to-coins
by (subst prob-space.branch-stream-space(2)) (auto intro:prob-space-measure-pmf)
also have ... = \( M \) B to-coins
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 \cdot 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 = (λx. Collect (cprefix x)) ∪ UNIV

lemma image-collect-eq: f : {x. A (f x)} = {x. A x} ∩ range f
  by auto

lemma coin-stream-basis: topological-basis coin-stream-basis
proof –
  have bij-betw (λx. (!) (smap discrete x)) UNIV UNIV
  by (intro bij-betw[where g=smap of-discrete o 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 3:range (λx. (!) (smap discrete x)) = UNIV
  using bij-is-surj by auto
obtain K :: (nat ⇒ bool discrete) set set where
  K-countable: countable K and K-top-basis: topological-basis K and
  K-cylinder: ∀k∈K. ∃X. (k = PiE UNIV X) ∧ (∀i. open (X i)) ∧ finite {i. X i ≠ UNIV}
using product-topology-countable-basis by auto
have from-coins-cprefix: from-coins-t' {xs. cprefix p xs} = PiE UNIV (λi. if i < length p then {discrete (p ! i)} else UNIV)
  (is ?L = ?R) for p
proof –
  have 2:from-coins ' {xs. cprefix p xs} = {f. ∃i < 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 from-coins-t'{xs. cprefix p xs} = (snth-smap discrete)'{from-coins ' {xs. cprefix p xs}}
  unfolding from-coins-t-def image-image by simp
also have ... = (smap discrete) ' {f. ∃i < length p. f !! i = p ! i}
  unfolding ? by simp
also have ... = (λx. snth (smap discrete x)) '
  {f. ∀i < length p. (smap discrete f) !! i = discrete (p ! i)}
  by (simp add:discrete-inject)
also have ... = {x. ∀i<length p. x i = discrete (p ! i)} ∩ range (λx. (!) (smap discrete x))
  by (intro image-collect-eq)
also have ... = {x. ∀i<length p. x i = discrete (p ! i)}
  unfolding ? by simp
also have ... = PiE UNIV (λi. if i < length p then {discrete (p ! i)} else UNIV)
  unfolding PiE-def Pi-def by auto
finally show ?thesis
  by simp
qed
have open U if 0:U ∈ coin-stream-basis for U
proof –
  obtain p where U-eq:U = {xs. cprefix p xs} using ? unfolding coin-stream-basis-def by auto
  show ?thesis
  unfolding open-coin-stream-def U-eq from-coins-cprefix
  by (intro open-PiE) (auto intro:open-discrete)
qed
moreover have ∃B∈coin-stream-basis. x ∈ B ∧ B ⊆ U if open U x ∈ U for U x
proof –
  have open (from-coins-t ' U) from-coins-t x ∈ from-coins-t ' U
    using that unfolding open-coin-stream-def by auto
  then obtain B where B: B ∈ K from-coins-t x ∈ B B ⊆ from-coins-t ' U
using topological-basis E[OF K-top-basis] by blast
obtain X where X \( B = \Pi E \) UNIV X and fin-X: finite \( \{i. X i \neq \text{UNIV}\} \)
using K-cylinder B(1) by auto
define Z where Z i = (X i \neq \text{UNIV}) for i
define n where n = (if \( \{i. X i \neq \text{UNIV}\} \neq \{\}\) then Suc (Max \( \{i. X i \neq \text{UNIV}\}\)) 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 = \text{UNIV} if \( i \geq n \) for i

proof

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

proof –

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

by simp

also have \( ... = \{\text{from-coins-t x i}\} \)

unfolding from-coins-t-def cnth-def by simp

also have \( \subseteq X i \)

using B(2) unfolding X PiE-def Pi-def by auto

finally show \( ?\text{thesis} \)

by simp

qed

hence from-coins-t \( R \subseteq \Pi E \) UNIV X

using X-univ unfolding R-def from-coins-cprefix

by (intro PiE-mono) auto

moreover have \( ... \subseteq\) from-coins-t \( R \subseteq\) from-coins-t \( U \)

using B(3) X by simp

ultimately have from-coins-t \( R \subseteq\) from-coins-t \( U \)

by simp

hence \( R \subseteq U \)

using bij-is-inj[OF bij-from-coins-t]

by (simp add: inj-image-eq-iff subset-image-iff)

moreover have \( R \in\) coin-stream-basis \( x \in R \)

unfolding R-def coin-stream-basis-def by (auto simp:cprefix-def)

ultimately show \( ?\text{thesis} \)

by auto

qed

ultimately show \( ?\text{thesis} \)

by (intro topological-basisI) auto

qed

lemma coin-stream-open: open \{xs. cprefix x xs\}

by (intro topological-basis-open[OF coin-stream-basis]) (simp add:coin-stream-basis-def)

instance coin-stream :: second-countable-topology

proof

show \( \exists (B :: \text{coin-stream set set}). \text{countable B} \wedge \text{open = generate-topology B} \)

by (intro exI[where x=coin-stream-basis] topological-basis-imp-subbasis conjI

coin-stream-basis) (simp add:coin-stream-basis-def)

qed

instantiation coin-stream :: uniformity-dist

begin
definition dist-coin-stream :: coin-stream \( \Rightarrow\) coin-stream \( \Rightarrow\) real

where dist-coin-stream x y = dist (from-coins-t x) (from-coins-t y)

definition uniformity-coin-stream :: (coin-stream \( \times\) coin-stream) filter

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

lemma \( \text{in-from-coins-iff} \): \( x \in \text{from-coins-t} \implies U \iff (\text{to-coins-t} \ x \in U) \)
  using to-from-coins-t from-to-coins-t by (simp add:image-iff) metis

instantiation \( \text{coin-stream} :: \text{metric-space} \)
begin
instance proof
  show open \( U = (\forall x \in U. \forall F (x', y) \text{ in uniformity. } x' = x \longrightarrow y \in U) \) for \( U :: \text{coin-stream set} \)
    proof
    have open \( U \implies \text{open} (\text{from-coins-t} \ U) \)
      unfolding open-coin-stream-def by simp
    also have \( \ldots \implies (\forall x \in U. \exists e > 0. \forall y. \text{dist} (\text{from-coins-t} x) y < e \longrightarrow y \in \text{from-coins-t} \ U) \)
      unfolding fun-open-ball-aux by auto
    also have \( \ldots \implies (\forall x \in U. \exists e > 0. \forall y \in \text{to-coins-t} \ U. \text{dist} x y < e \longrightarrow y \in U) \)
      unfolding dist-coin-stream-def by (intro ball-cong refl ex-cong)
      (simp add: from-to-coins-t in-from-coins-iff)
    also have \( \ldots \implies (\forall x \in U. \exists e > 0. \forall y. \text{dist} x y < e \longrightarrow y \in U) \)
      using bij-is-surj[OF bij-to-coins-t] by simp
    finally have open \( U = (\forall x \in U. \exists e > 0. \forall y. \text{dist} x y < e \longrightarrow y \in U) \)
      by simp
    thus \( \asthesis \)
      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 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 \( \text{from-coins-t-u-continuous} \): \( \text{uniformly-continuous-on} \ \text{UNIV} \ \text{from-coins-t} \)
  unfolding uniformly-continuous-on-def dist-coin-stream-def by auto

lemma \( \text{to-coins-t-u-continuous} \): \( \text{uniformly-continuous-on} \ \text{UNIV} \ \text{to-coins-t} \)
  unfolding uniformly-continuous-on-def dist-coin-stream-def from-to-coins-t by auto

lemma \( \text{to-coins-t-continuous} \): \( \text{continuous-on} \ \text{UNIV} \ \text{to-coins-t} \)
  using to-coins-t-u-continuous uniformly-continuous-imp-continuous by auto

instance \( \text{coin-stream} :: \text{complete-space} \)
proof
  show convergent \( X \) if \( \text{Cauchy} \ X \) for \( X :: \text{nat} \Rightarrow \text{coin-stream} \)
    proof
    have \( \text{Cauchy} \ (\text{from-coins-t} \circ X) \)
      by (intro uniformly-continuous-imp-Cauchy-continuous[OF from-coins-t-u-continuous that])
    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 \( \text{isCont} \ \text{to-coins-t} \ x \)
      using to-coins-t-continuous continuous-on-eq-continuous-within by blast
  qed
ultimately have \((\text{to-coins} \circ \text{from-coins} \circ X) \longrightarrow \text{to-coins} \circ X\) using isCont-tendsto-compose by (auto simp add: comp-def)

thus convergent X

unfolding convergent-def comp-def to-from-coins by auto

qed

lemma at-least-borelI:

assumes topological-basis K
assumes countable K
assumes \(K \subseteq \text{sets} \ A\)
assumes open U
shows \(U \in \text{sets} \ A\)

proof

obtain \(K'\) where \(K'\)-range: \(K' \subseteq K\) and \(\bigcup K' = U\)

using assms(1,4) unfolding topological-basis-def by blast

hence \(U = \bigcup K'\) by simp

also have ... \(\in \text{sets} \ A\)

using \(K'\)-range assms(2,3) countable-subset

by (intro sets.countable-Union) auto

finally show \(?\text{thesis}\) by simp

qed

lemma measurable-sets-coin-space:

assumes \(f \in \text{measurable} \ B \ A\)
assumes Collect P \(\in \text{sets} \ A\)
shows \(\{xs. \ P(f xs)\} \in \text{sets} \ B\)

proof

have \(\{xs. \ P(f xs)\} = f \circ \text{Collect} \cap \text{space} B\)

unfolding vimage-def space-coin-space by simp

also have ... \(\in \text{sets} \ B\)

by (intro measurable-sets[OF assms(1,2)])

finally show \(?\text{thesis}\) by simp

qed

lemma coin-space-is-borel-measure:

assumes open U
shows \(U \in \text{sets} \ B\)

proof

have 0:countable \(\text{coin-stream-basis}\)

unfolding \(\text{coin-stream-basis-def}\) by simp

have cnth-sets: \(\{xs. \ \text{cnth} xs \ i = v\} \in \text{sets} \ B\) for \(i \ v\)

by (intro measurable-sets-coin-space[OF cnth-measurable]) auto

have \(\{xs. \ \text{cprefix} x xs\} \in \text{sets} \ B\) for \(x\)

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

case True

have \(\{xs. \ \text{cprefix} x xs\} = (\bigcap i < \text{length} x. \{xs. \ \text{cnth} xs \ i = x ! i\})\)

unfolding cprefix-iff by auto

also have ... \(\in \text{sets} \ B\)

using cnth-sets True

by (intro sets.countable-INT image-subsetI) auto

finally show \(?\text{thesis}\) by simp

next

case False

hence \(\{xs. \ \text{cprefix} x xs\} = \text{space} B\)

unfolding cprefix-iff space-coin-space by simp

qed
also have ... ∈ sets B
by simp
finally show ?thesis by simp
qed
hence 1:coin-stream-basis ⊆ sets 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 (λS. S=UNIV ∨ None ∉ S)

lemma option-ud-topology: istopology (λS. S=UNIV ∨ None ∉ S) (is istopology ?T)
proof –
  have ?T (U ∩ V) if ?T U ?T V for U V using that by auto
  moreover have ?T (⋃ K) if ⋀ U. U ∈ K ⇒ ?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 ←→ (S = UNIV ∨ None ∉ S)
unfolding option-ud-def by (subst topology-inverse[OF option-ud-topology]) auto

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

lemma continuous-into-option-udI:
assumes ⋀ x. openin X (f −’ {Some x} ∩ topspace X)
shows continuous-map X option-ud f
proof (cases U = UNIV)
  case True
  then show ?thesis by simp
next
  case False
  define V where V = the ’ U
  have None ∉ U
    using that False unfolding openin-option-ud by simp
  hence Some ’ V = id ’ U
  unfolding V-def image-image id-def
  by (intro image-cong refl) (metis option.exhaust-sel)
  hence U = Some ’ V by simp
  hence {x ∈ topspace X. f x ∈ U} = (∪ v ∈ V. f −’ {Some v} ∩ topspace X) by auto
  moreover have openin X (∪ v ∈ V. f −’ {Some v} ∩ 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)
4 Randomized Algorithms (Internal Representation)

theory Randomized-Algorithm-Internal
  imports
    HOL-Probability, Probability
    Coin-Space
    Tau-Additivity
    Zeta-Function_Zeta-Library
begin

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

This is the inverse of set-option

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

lemma the-elem-opt-empty[simp]: the-elem-opt {} = None
  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 ⇒ bool
  where at-most-one S ←→ (∀ x y. x ∈ S ∧ y ∈ S → 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 ∈ S by auto
  hence S = {x} unfolding at-most-one-def by auto
  thus ?thesis using assms(1) by simp
qed

lemma the-elem-opt-Some-iff[simp]: at-most-one S → the-elem-opt S = Some x ⇔ S = {x}
  by (induction S rule:at-most-one-cases) auto

lemma the-elem-opt-None-iff[simp]: at-most-one S → the-elem-opt S = None ⇔ 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 ⇒ ('a × 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]². It is also closely related to the construction of parser monads in functional languages [6].

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

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

²Although 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: bind-rai (bind-rai f g) h = bind-rai f (λx. bind-rai (g x) h)
unfolding bind-rai-def by (simp add:case-prod-beta’)

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

definition wf-on-prefix :: ’a random-alg-int ⇒ bool list ⇒ ’a ⇒ bool where
wf-on-prefix f p r = (∀ cs. f (cshift p cs) = Some (r,cs))

definition wf-random :: ’a random-alg-int ⇒ bool where
wf-random f ←→ (∀ bs. case f bs of None ⇒ True | Some (r,bs’) ⇒ (∃ p. cprefix p bs ∧ wf-on-prefix f p r))

definition range-rm :: ’a random-alg-int ⇒ ’a set where
range-rm f = Some −’ (range (map-option fst ◦ f))

lemma in-range-rmI:
assumes r bs = Some (y, n)
sows y ∈ range-rm r
proof –
  have Some (y, n) ∈ 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 ⇒ ’a option measure where
distr-rai f = distr BD (map-option fst ◦ f)

lemma wf-randomI:
assumes ⋀ bs. f bs ≠ None ⇒ (∃ r. cprefix p bs ∧ wf-on-prefix f p r)
sows wf-random f
proof –
  have ∃ p. cprefix p bs ∧ wf-on-prefix f p r if 0:f bs = Some (r, bs’) for bs r bs’
  proof –
    obtain p r’ where 1:cprefix p bs and 2:wf-on-prefix f p r’
    using assms 0 by force
    have f bs = f (cshift p (cdrop (length p) bs))
    using 1 unfolding cprefix-def by (metis ctake-cdrop)
    also have ... = Some (r’, cdrop (length p) bs)
    using 2 unfolding wf-on-prefix-def by auto
    finally have f bs = Some (r’, cdrop (length p) bs)
    by simp
    hence r = r’ using 0 by simp
    thus ?thesis using 1 2 by auto
  qed
  thus ?thesis
  unfolding wf-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 ≥ f) (p@q) s
proof –
  have \((m \gg f) (\mathit{cshift} \ (p@q) \ cs) = \text{Some} \ (s, cs)\) for \(cs\)
  proof –
    have \((m \gg f) (\mathit{cshift} \ (p@q) \ cs) = (m \gg f) (\mathit{cshift} \ p \ (\mathit{cshift} \ q \ cs))\)
    by simp
    also have \(= (f \ x) (\mathit{cshift} \ q \ cs)\)
    using assms unfolding \(\mathit{wf-on-prefix-def} \ \mathit{bind-rai-def}\) by simp
    also have \(= \text{Some} \ (s, cs)\)
    finally show \(?\text{thesis}\) by simp
qed
thus \(?\text{thesis}\)
unfolding \(\mathit{wf-on-prefix-def}\) by simp
qed

lemma \(\mathit{wf-bind}:\)
  assumes \(\mathit{wf-random} \ m\)
  assumes \(\forall x. \ x \in \mathit{range-rm} \ m = \Rightarrow \mathit{wf-random} \ (f \ x)\)
  shows \(\mathit{wf-random} \ (m \gg f)\)
  proof (rule \(\mathit{wf-randomI}\))
    fix \(bs\)
    assume \((m \gg f) \ bs \neq None\)
    then obtain \(x \ bs' \ y \ bs''\) where \(1: m \ bs = \text{Some} \ (x, bs')\) and \(2: f \ x \ bs' = \text{Some} \ (y, bs'')\)
    unfolding \(\mathit{bind-rai-def}\) by (cases \(m \ bs\)) auto
    hence \(wf: \mathit{wf-random} \ (f \ x)\)
    by (intro assms \(2\) in-range-rmI) auto
    obtain \(p\) where \(5: \mathit{wf-on-prefix} \ m \ p \ x\) and \(3: \mathit{cprefix} \ p \ bs\)
    using \(wf\) \(2\) unfolding \(\mathit{wf-random-def}\) by (auto split:option.split-asm)
    have \(4: bs = \mathit{cshift} \ p \ (\mathit{cdrop} \ (\mathit{length} \ p) \ bs)\)
    using \(3\) unfolding \(\mathit{cprefix-def}\) by (metis ctake-cdrop)
    hence \(m \ bs = \text{Some} \ (x, \mathit{cdrop} \ (\mathit{length} \ p) \ bs)\)
    using \(4\) by auto
    hence \(6: bs = \mathit{cshift} \ p \ bs'\)
    using \(4\) by auto
    obtain \(q\) where \(7: \mathit{wf-on-prefix} \ (f \ x) \ q \ y\) and \(8: \mathit{cprefix} \ q \ bs'\)
    using \(wf\) \(2\) unfolding \(\mathit{wf-random-def}\) by (auto split:option.split-asm)
    have \(\mathit{cprefix} \ (p@q) \ bs\)
    unfolding \(6\) using \(8\) unfolding \(\mathit{cprefix-def}\) by auto
    moreover have \(\mathit{wf-on-prefix} \ (m \gg f) \ (p@q) \ y\)
    by (intro \(\mathit{wf-on-prefix-bindI}\) [OF \(5\) \(7\)])
    ultimately show \(\exists p \ r. \ \mathit{cprefix} \ p \ bs \wedge \mathit{wf-on-prefix} \ (m \gg f) \ p \ r\)
    by auto
qed

lemma \(\mathit{wf-return}:\)
  \(\mathit{wf-random} \ (\mathit{return-rai} \ x)\)
  proof (rule \(\mathit{wf-randomI}\))
    fix \(bs\) assume \(\mathit{return-rai} \ x \ bs \neq None\)
    have \(\mathit{wf-on-prefix} \ (\mathit{return-rai} \ x) \ [] \ x\)
    unfolding \(\mathit{wf-on-prefix-def} \ \mathit{return-rai-def}\) by auto
    moreover have \(\mathit{cprefix} \ [] \ bs\)
    unfolding \(\mathit{cprefix-def}\) by auto
    ultimately show \(\exists p \ r. \ \mathit{cprefix} \ p \ bs \wedge \mathit{wf-on-prefix} \ (\mathit{return-rai} \ x) \ p \ r\)
by auto

qed

lemma wf-coin:
wf-random (coin-rai)
proof (rule wf-randomI)
  fix bs assume coin-rai bs ≠ 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 ∃ p r. cprefix p bs ∧ wf-on-prefix coin-rai p r
    by auto
qed

definition ptree-rm :: 'a random-alg-int ⇒ bool list set
where ptree-rm f = { p. ∃ r. wf-on-prefix f p r }

definition eval-rm :: 'a random-alg-int ⇒ bool list ⇒ '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 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 ∈ ptree-rm f
assumes q ∈ 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)
    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 \( \theta \) thesis using \( \emptyset \) by simp
qed

lemma prefixes-parallel:
assumes \( p \in \text{ptree-rm} \; f \)
assumes \( q \in \text{ptree-rm} \; f \)
shows \( p = q \lor p \parallel q \)
using prefixes-parallel-helper assms by blast

lemma prefixes-singleton:
assumes \( p \in \{ \, p \in \text{ptree-rm} \; f \; \land \; \text{cprefix} \; p \; bs \, \} \)
shows \( \{ \; p \in \text{ptree-rm} \; f \; \land \; \text{cprefix} \; p \; bs \; \} = \{ p \} \)
proof
have \( q = p \) if \( q \in \text{ptree-rm} \; f \; \text{cprefix} \; q \; bs \) for \( q \)
using same-prefix-not-parallel assms prefixes-parallel that by blast
thus \( \{ p \in \text{ptree-rm} \; f \; \land \; \text{cprefix} \; p \; bs \} \subseteq \{ p \} \)
by (intro subsetI simp)
next
show \( \{ p \} \subseteq \{ p \in \text{ptree-rm} \; f \; \land \; \text{cprefix} \; p \; bs \} \)
using assms by auto
qed

lemma prefixes-at-most-one:
at-most-one \( \{ p \in \text{ptree-rm} \; f \; \land \; \text{cprefix} \; p \; x \} \)
unfolding at-most-one-def using same-prefix-not-parallel prefixes-parallel by blast

definition consumed-prefix f bs = the-elem-opt \( \{ p \in \text{ptree-rm} \; f \; \land \; \text{cprefix} \; p \; bs \} \)

lemma wf-random-alt:
assumes \( f \) random f
shows \( f \; bs = \text{map-option} \; (\lambda p. \, (\text{eval-rm} \; f \; p, \; \text{cdrop} \; (\text{length} \; p) \; bs)) \; (\text{consumed-prefix} \; f \; bs) \)
proof (cases f bs)
  case None
  have False if \( p \)-in: \( p \in \text{ptree-rm} \; f \) and \( p\)-pref: \( \text{cprefix} \; p \; bs \) for \( p \)
  proof
  obtain \( r \) where \( w f: \text{wf-on-prefix} \; f \; p \; r \) using that \( p\)-in unfolding ptree-rm-def by auto
  have \( bs = \text{cshift} \; p \; (\text{cdrop} \; (\text{length} \; p) \; bs) \)
  using p-pref unfolding cprefix-def by (metis ctake-cdrop)
  hence \( f \; bs \neq \text{None} \)
  using w f unfolding wf-on-prefix-def
  by (metis option.simps(3))
  thus False using None by simp
  qed
  hence \( \emptyset: \{ p \in \text{ptree-rm} \; f \; \land \; \text{cprefix} \; p \; bs \} = \{ \} \)
  by auto
  show \( \theta \) thesis unfolding \( \emptyset \) None consumed-prefix-def by simp
next
  case (Some a)
  moreover obtain \( r \; cs \) where \( a = (r, \; cs) \) by (cases a) auto
  ultimately have \( f \; bs = \text{Some} \; (r, \; cs) \) by simp
  hence \( \exists p. \, \text{cprefix} \; p \; bs \land \text{wf-on-prefix} \; f \; p \; r \)
  using assms(1) unfolding wf-random-def by (auto split:option.split_asm)
then obtain $p$ where $sp : \text{cprefix } p \; bs$ and $wf : \text{wf-on-prefix } f \; p \; r$
by auto
hence $p \in \{ p \in \text{ptree-rm } f . \; \text{cprefix } p \; bs \}$
unfolding $\text{ptree-rm-def}$ by auto
hence $0 : \{ p \in \text{ptree-rm } f . \; \text{cprefix } p \; bs \} = \{ p \}$
using $\text{prefixes-singleton}$ by auto
show $\text{thesis}$ unfolding $0 \; \text{wf-on-prefixD} [OF \; \text{wf } \; \text{sp}] \; \text{consumed-prefix-def}$ by simp
qed

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

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

definition $\text{consumed-bits}$ where
$\text{consumed-bits } f \; bs = \text{map-option } \text{length} \; (\text{consumed-prefix } f \; bs)$
definition $\text{used-bits-distr} :: \alpha \text{ random-alg-int } \Rightarrow \text{nat option measure}$
where $\text{used-bits-distr } f = \text{distr } \mathcal{B} \; \mathcal{D} \; (\text{consumed-bits } f)$

lemma $\text{wf-random-alt2}$:
assumes $\text{wf-random } f$
shows $f \; bs = \text{map-option } (\lambda n. \text{eval-rm } f \; (\text{ctake } n \; bs) \; , \; \text{cdrop } n \; bs)) \; (\text{consumed-bits } f \; bs)$
(is $\text{?L} = \; \text{?R}$)
proof –
  have $0 : \text{cprefix } x \; bs$ if $\text{consumed-prefix } f \; bs = \text{Some } x$ for $x$
  using $\text{that } \text{the-elem-opt-Some-iff} [OF \; \text{prefixes-at-most-one}]$ unfolding $\text{consumed-prefix-def}$ by auto
  have $\text{?L} = \text{map-option } (\lambda p. \; (\text{eval-rm } f \; p \; , \; \text{cdrop } (\text{length } p) \; bs)) \; (\text{consumed-prefix } f \; bs)$
  by (subst $\text{wf-random-alt} [OF \; \text{assms}]$) simp
also have $\ldots = \; \text{?R}$

using \( \emptyset \) 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 \( f bs = None \iff consumed-prefix f bs = None \)
using wf-random-alt[OF assms] by (simp)

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

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

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

qed

lemma consumed-bits-measurable: \( consumed-bits f : B \to M \ D \)
proof
have \( \emptyset: consumed-bits f = \neg \{ x \} \cap space B \in sets B \) (is \(?L = \neg\))
if \( x-ne-inf: x \neq None \) for \( x \)
proof
obtain \( n \) where \( x-def: x = Some n \)
using \( x-ne-inf \) that by auto
have \( ?L = \{ bs. \exists z. consumed-prefix f bs = Some z \wedge length z = n \} \)
unfolding consumed-bits-def vimage-def space-coin-space x-def by simp
also have \( \ldots = \{ bs. \exists p. \{ p \in ptree-rm f. cprefix p bs \} = \{ p \} \wedge length p = n \} \)
unfolding consumed-prefix-def x-def the-elem-opt-Some-iff[OF prefixes-at-most-one] by simp
also have \( \ldots = \{ bs. \exists p. cprefix p bs \wedge length p = n \wedge p \in ptree-rm f \} \)
using prefixes-singleton by (intro Collect-cong ex-cong1) auto
also have \( \ldots = \{ bs. ctake n bs \in ptree-rm f \} \)
unfolding cprefix-def by (intro Collect-cong) (metis length-ctake)
also have \( \ldots \in sets B \)
by (intro measurable-sets-coin-space[OF ctake-measurable]) simp
finally show \(?thesis\)
by simp

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

lemma \( R\)-sets:
assumes \( \text{wf} \): \( \text{wf-random \ f} \)
shows \( \{ \text{bs. \ f \ bs = \text{None} \} \in \text{sets \ B} \} \{ \text{bs. \ f \ bs \neq \text{None} \} \in \text{sets \ B} \)
proof
  show 0: \( \{ \text{bs. \ f \ bs = \text{None} \} \in \text{sets \ B} \)
    unfolding consumed-bits-inf-iff[OF \( \text{wf} \)]
    by (intro measurable-sets-coin-space[OF consumed-bits-measurable]) simp
  have \( \{ \text{bs. \ f \ bs \neq \text{None} \} = \text{space \ B} - \{ \text{bs. \ f \ bs = \text{None} \} \}
    unfolding space-coin-space by (simp add:set-eq-iff del:not-None-eq)
  also have ... \( \in \text{sets \ B} \)
    by (intro sets.compl-sets 0)
  finally show \( \{ \text{bs. \ f \ bs \neq \text{None} \} \in \text{sets \ B} \)
    by simp
qed

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

lemma consumed-prefix-continuous:
  continuous-map euclidean option-ud (consumed-prefix \( \ f \))
proof (intro contionuous-into-option-udI)
  fix \( x :: \text{bool list} \)

  have open ((consumed-prefix \( \ f \)) -` `{\( \text{Some \ x} \}) (is open \( \tau \))
proof (cases \( x \in \text{ptree-rm \ f} \))
  case True
  hence \( \tau = \{ \text{bs. \ cprefix \ x \ bs} \)
    unfolding vimage-def comp-def by (simp add:consumed-prefix-some-iff)
  show \( \text{thesis} \)
    unfolding \( \tau \) by (intro coin-steam-open)
next
  case False
  hence \( \tau = \{ \}
    unfolding vimage-def comp-def by (simp add:consumed-prefix-some-iff)
  thus \( \text{thesis} \)
    by simp
qed
thus openin euclidean ((consumed-prefix \( \ f \)) -` `{\( \text{Some \ x} \} \cap \text{topspace \ euclidean} \)
    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 \( \text{wf} : \text{wf-random \ f} \)
shows continuous-map euclidean option-ud (map-option \( \text{fst} \circ \ f \))
proof

have 0: map-option fst ≼ (λbs. f bs)
  by (subt wf-random-alt [OF wf]) (simp add:map-option.compositionality comp-def)

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

lemma none-measure-subprob-algebra:
  return D None ∈ space (subprob-algebra D)
  by (metis measure-subprob return-pmf rep-eq)
context
  fixes f :: 'a random-alg-int
  fixes R
  assumes wf: wf-random f
  defines R ≡ restrict-space B {bs. f bs ≠ None}
begin

lemma the-f-measurable: the ◦ f ∈ R → M D ⊗ M B
proof
  have consumed-bits f bs ≠ None if bs ∈ space R for bs
    unfolding R-def space-restrict-space space-coin-space
    by (simp del:not-infinity-eq not-None-eq)
  hence 0: the (f bs) = map-prod (eval-rm f) id (g bs) if bs ∈ space R for bs
    unfolding g-def using that
    by (intro measurable-restrict-space1 measurable-comp [OF consumed-bits-measurable]) simp
  have 1: h ∈ R → M D
    unfolding R-def h-def
    by (intro measurable-restrict-space1 measurable-comp [OF consumed-bits-measurable]) simp
  have ctake k ∈ R → M D for k
    unfolding R-def by (intro measurable-restrict-space1 ctake-measurable)
  moreover have cdrop k ∈ R → M B for k
    unfolding R-def by (intro measurable-restrict-space1 cdrop-measurable)
  ultimately have g ∈ R → M D ⊗ M B
    unfolding g-def
    by (intro measurable-Pair measurable-Pair-compose-split [OF 1 measurable-id]) simp
  hence (map-prod (eval-rm f) id ◦ g) ∈ R → M D ⊗ M B
    by (intro measurable-comp [where N = D ⊗ M B] map-prod-measurable) auto
  moreover have (the ◦ f) ∈ R → M D ⊗ M B ←→ (map-prod (eval-rm f) id ◦ g) ∈ R → M D ⊗ M B
    using 0 by (intro measurable-cong) (simp add:comp-def)
  ultimately show ?thesis
    by auto
qed

lemma distr-rai-measurable: map-option fst ◦ f ∈ B → M D
proof
  have 0: countable {bs. f bs ≠ None}, {bs. f bs = None}
    by simp
have 1: Ω ∈ sets B ∧ map-option fst ◦ f ∈ restrict-space B Ω → M D
if Ω ∈ {{bs. f bs ≠ None}, {bs. f bs = None}} for Ω
proof (cases Ω = {bs. f bs ≠ None})
  case True
  have Some ◦ fst ◦ (the ◦ f) ∈ R → M D
  by (intro measurable-comp[OF the-f-measurable]) auto
  hence map-option fst ◦ f ∈ restrict-space B Ω → M D
  unfolding R-def by (subst measurable-cong[where g=Some ◦ fst ◦ (the ◦ f)])
  (auto simp add: space-restrict-space space-coin-space)
  thus Ω ∈ sets B ∧ map-option fst ◦ f ∈ restrict-space B Ω → M D
  unfolding R-def True using R-sets[OF wf] by auto
next
  case False
  hence 2: Ω = {bs. f bs = None}
  using that by simp
  have map-option fst ◦ f ∈ restrict-space B{bs. f bs = None} → M D
  by (subst measurable-cong[where g=λ -. None])
  (simp-all add: space-restrict-space)
  thus Ω ∈ sets B ∧ map-option fst ◦ f ∈ restrict-space B Ω → M D
  unfolding 2 using R-sets[OF wf] by auto
qed

have 3: space B ⊆ ∪ { {bs. f bs ≠ None}, {bs. f bs = None} }
  unfolding space-coin-space by auto

show ?thesis
  by (rule measurable-piecewise-restrict[OF 0]) (use 1 3 space-coin-space in ⟨auto⟩)
qed

lemma distr-rai-subprob-space:
distr-rai f ∈ space (subprob-algebra 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) = D
  unfolding distr-rai-def by simp
  ultimately show ?thesis
  unfolding space-subprob-algebra using prob-space-imp-subprob-space
  by auto
  qed

lemma fst-the-f-measurable: fst ◦ the ◦ f ∈ R → M D
proof –
  have fst ◦ (the ◦ f) ∈ R → M 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:

33
\textit{distr} \(R \ (D \otimes_M B)\) (the \(\circ f\)) = \textit{distr} \(R \ D\) (\(\text{fst} \circ \text{the} \circ f\)) \(\otimes_M B\)

**proof**

- define \(C\) where \(C k = \text{consumed-bits} f - ' [\text{Some} k]\) \(\text{for} k\)

have 2: \((\exists k \ k \in C k) \longrightarrow \text{fx} \neq \text{None} \text{ for} \ x\)

- using \(\text{consumed-bits-inf-iff}[\text{OF uwf}]\) unfolding \(C\)-def

by auto

hence 5: \(C k \subseteq \text{space} \ R \text{ for} k\)

- unfolding \(R\)-def \space-restrict-space \space-coin-space

by auto

have 1: \{\text{bs. f bs} \neq \text{None}\} \cap \text{space} \ B \in \text{sets} \ B

- using \(\text{R-sets}[\text{OF uwf}]\) by simp

have 6: \(C k \in \text{sets} \ B \text{ for} k\)

- unfolding \(C\)-def \(\text{vimage-def}\)

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

have 8: \(x \in C k \longrightarrow \text{ctake} k x \in \text{ptree-rm} \text{ f for} x k\)

- unfolding \(C\)-def using \(\text{consumed-bits-enat-iff}\) by auto

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

**proof**

- have \(\text{cshift} \ (\text{ctake} k x) \ y \in C k\)

- using \(8\) by simp

hence the \((f \ (\text{cshift} \ (\text{ctake} k x) \ y))\) = \((\text{eval-rm} \ f \ (\text{ctake} k x), \ y)\)

- using \(\text{wf-random-alt2}[\text{OF uwf}]\) unfolding \(C\)-def by simp

also have \(\ldots = \ (\text{fst} \ (\text{the} \ (f x)), \ y)\)

- using \(\text{that} \ \text{wf-random-alt2}[\text{OF uwf}]\) unfolding \(C\)-def by simp

finally show \(\text{thesis} \ \text{by simp}\)

qed

have \(C\)-disj: disjoint-family \(C\)

- unfolding disjoint-family-on-def \(C\)-def by auto

have 0:

- \( \text{emeasure} \ (\text{distr} \ (D \otimes_M B) \ \text{(the} \circ f)) \ (A \times B) = \)

  - \( \text{emeasure} \ (\text{distr} \ D \ \text{(fst} \circ \text{the} \circ f)) \ A \ast \text{emeasure} \ B \ B\)

(is \(?L = ?L\) if \(A \in \text{sets} \ D \ B \in \text{sets} \ B\) \(\text{for} A B\))

**proof**

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

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

also have \(\ldots \in \text{sets} \ (\text{restrict-space} \ (C k))\)

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

also have \(\ldots = \text{sets} \ (\text{restrict-space} \ B \ (C k))\)

- using 5 unfolding \(\text{R-def} \ \text{sets-restrict-space} \ \text{space-restrict-space} \ \text{space-coin-space}\)

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

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

- by simp

thus \(?L1 \in \text{sets} \ B\)

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

qed

have 4: \(\{\text{bs. the} \ (f \ \text{bs}) \in A \times B \wedge \text{bs} \in C k\} \in \text{sets} \ B \ \text{is} \ ?L1 \in \ldots \ \text{for} k\)

**proof**

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

have \(?L = \text{emeasure R ((the} \circ f) - ' (A \times B) \cap \text{space R})
using that the-f-measurable by (intro emeasure-distr) auto
also have \(\ldots = \text{emeasure B \{x. (f x) \in A \times B \land f x \neq \text{None}\}
unfolding \text{vimage-def R-def Int-def}
by (simp add:space-restrict-space-space-coin-space)
also have \(\ldots = \text{emeasure B \{x. \text{the} (f x) \in A \times B \land (\exists k. \text{x C k})\}
unfolding R-def 2 using 1 by (intro emeasure-restrict-space) auto
also have \(\ldots = \text{emeasure B (} \bigcup \text{k. \{x. \text{the} (f x) \in A \times B \land x \in C k\})
by (intro arg-cong2[where \(f=\text{emeasure}\)] branch-coin-space(2)[symmetric] refl)
also have \(\ldots = (\sum k. \text{emeasure B \{x. \text{the} (f x) \in A \times B \land x \in C k\})
using 4 C-disj
by (intro suminf-emeasure[symmetric] subsetI) (auto simp:disjoint-family-on-def)
also have \(\ldots = (\sum k. \text{emeasure (distr (B \bigotimes_M B) B (x,y) (cshift (ctake k x) y)))}
\{x. \text{the} (f x) \in A \times B \land x \in C k\}
by (intro suminf-cong arg-cong2[where \(f=\text{emeasure}\)] branch-coin-space(2)[symmetric] refl)
also have \(\ldots = (\sum k. \text{emeasure (B \bigotimes_M B)}
\{x. \text{the} (f (cshift (ctake k (fst x)) (snd x))) \in A \times B \land (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 \(\ldots = (\sum k. \text{emeasure (B \bigotimes_M B)}
\{x. \text{the} (f (cshift (ctake k (fst x)) (snd x))) \in A \times B \land f x \in C k\}
using 8 by (intro suminf-cong arg-cong2[where \(f=\text{emeasure}\)] refl Collect-cong) auto
also have \(\ldots = (\sum k. \text{emeasure (B \bigotimes_M B)}
\{x. \text{fst (the} (f x)) \in A \land x \in C k \times B\}
using 7 by (intro suminf-cong arg-cong2[where \(f=\text{emeasure}\)] refl)
(auto simp add:mem-Times-iff set-equ-if)
also have \(\ldots = (\sum k. \text{emeasure B \{x. \text{fst (the} (f x)) \in A \times x \in C k\} \ast \text{emeasure B B)}
using 3 that(2)
by (intro suminf-cong coin-space.emeasure-pair-measure-Times) auto
also have \(\ldots = (\sum k. \text{emeasure B \{x. \text{fst (the} (f x)) \in A \times x \in C k\} \ast \text{emeasure B B}
by simp
also have \(\ldots = \text{emeasure B (} \bigcup \text{k. \{x. \text{fst (the} (f x)) \in A \land x \in C k\}) \ast \text{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 \(\ldots = \text{emeasure B \{x. \text{fst (the} (f x)) \in A \land (\exists k. \text{x C k})\} \ast \text{emeasure B B}
by (intro arg-cong2[where \(f=\text{emeasure}\)] arg-cong2[where \(f=(*))\]) auto
also have \(\ldots = \text{emeasure R \{x. \text{fst (the} (f x)) \in A \land f x \neq \text{None}\} \ast \text{emeasure B B}
unfolding R-def 2 using 1
by (intro arg-cong2[where \(f=(*))\] emeasure-restrict-space[symmetric] subsetI) simp-all
also have \(\ldots = \text{emeasure R ((f x \circ \text{the} \circ f) - ') A \land \text{space R}) \ast \text{emeasure B B}
unfolding vimage-def R-def Int-def by (simp add:space-restrict-space-space-coin-space)
also have \(\ldots = \vdash k\)
using that
by (intro arg-cong2[where \( f = (*) \)] emeasure-distr[ symmetric] fst-the-f-measurable) auto

finally show \( \theta \)thesis by simp

qed

have finite-measure \( R \)
  using 1 unfolding R-def space-coin-space
  by (intro finite-measure-restrict-space) simp-all

hence finite-measure (distr \( R \) \( D \) (fst \( \circ \) the \( f \)))
  by (intro finite-measure.finite-measure-distr fst-the-f-measurable)

hence 1:sigma-finite-measure (distr \( R \) \( D \) (fst \( \circ \) the \( f \)))
  unfolding finite-measure-def by auto

have 2:sigma-finite-measure \( B \)
  using prob-space-imp-sigma-finite[of coin-space. prob-space-axioms]
  by simp

show \( \theta \)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: \( \forall x. x \in \text{range-\( rm \) } m \implies \text{wf-random } (f x) \)
  shows distr-rai \( (m \gg f) \) = distr-rai \( m \gg \)

  \( (\lambda x. \text{if } x \in \text{Some } \text{range-\( rm \) } m \text{ then distr-rai } (f (\text{the } x)) \text{ else return } \text{ \( D \) None}) \)

  (is \( ?L = ?RHS \))

proof (rule measure-eqI)
  have sets \( ?L = \text{UNIV} \)
    unfolding distr-rai-def by simp
  also have ... = sets \( ?RHS \)
    unfolding distr-rai-def by (subst sets-bind[of \( N=D \)])
    (simp-all add: option.case-distrib option.case-eq-if)
  finally show sets \( ?L = ?RHS \) by simp

next
  let \( ?m = \text{distr-rai} \)
  let \( ?H = \text{count-space} \ (\text{range-\( rm \) } m) \)
  let \( ?R = \text{restrict-space} \ \mathcal{B} \ {bs. m bs \neq \text{None}} \)

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

  have N-meas: \( N \in \text{sets } \text{coin-space} \)
    unfolding N-def using R-sets[of \( \text{wf-m} \)] by simp

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

  have wf-bind: wf-random \( (m \gg f) \)
    using wf-bind[of \( \text{Assms} \)] by auto

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

  have \( \{v, bs\} \). \( \text{map-option } \text{fst } (f v bs) \in A \wedge v \in \text{range-\( rm \) } m \}
    = \( \text{map-option } \text{fst } \circ \text{case-prod } f \) \( \rightarrow_A \text{ space } (\mathcal{H} \otimes_M \mathcal{B}) \)

36
unfolding vimage-def space-pair-measure space-coin-space by auto
also have ... ∈ sets (\{H ⊗ M \ 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 ... = sets (restrict-space D (range-rm m) ⊗ M \ B)
unfolding restrict-space inf-top-right by simp
also have ... = sets (restrict-space (D ⊗ M \ B) (range-rm m × space coin-space))
by (subst coin-space.restrict-space-pair-lift) auto
finally have \{v, bs\}. map-option fst (f v bs) ∈ A ∧ v ∈ range-rm m \in sets (restrict-space (D ⊗ M \ B) (range-rm m × UNIV))
unfolding space-coin-space by simp
moreover have range-rm m × space-coin-space ∈ sets (D ⊗ M \ B)
by (intro pair-measure1 sets.top) auto
ultimately have 2: \{v, bs\}. map-option fst (f v bs) ∈ A ∧ v ∈ range-rm m \in sets (D ⊗ M \ B)
by (subst (asm) sets-restrict-space-iff) (auto simp: space-coin-space)

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

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

have \{\lambda x. emeasure (distr-rai (f (fst (the (m x))))) \} A * indicator N x =
\{\lambda x. emeasure (if m x ≠ None then distr-rai (f (fst (the (m x)))) else null-measure D) A\)
unfolding N-def by (intro ext) simp
also have ... = (\lambda v. emeasure (if v∈Some ' range-rm m then ?m (f (the v)) else null-measure D) A)

(\circ (map-option fst o 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 ... ∈ 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
finnally have 4: (\lambda x. emeasure (distr-rai (f (fst (the (m x))))) \} A * indicator N x)
∈ coin-space → M borel by simp

let ?N = emeasure B \{bs, bs ∉ N ∧ None \in A\}

have emeasure ?L A = emeasure B (\{map-option fst o (m ≫ f)\} ' A)
unfolding distr-rai-def using 0 by (subst emeasure-distr) (simp-all add: space-coin-space)
also have ...
emeasure B (\{map-option fst o (m ≫ f)\} ' A ⊗ N) + emeasure B (\{map-option fst o (m ≫ f)\} ' A ⊗ N)
using N-meas N-meas' 1
by (subst emeasure-Un'[symmetric]) (simp-all add: Int-Un-distrib'[symmetric])
also have ...
emeasure B (\{map-option fst o (m ≫ f)\} ' A ⊗ N) + emeasure ?R (\{map-option fst o (m ≫ f)\} ' A ⊗ N)
using N-meas unfolding N-def
by (intro arg-cong2[where f=emeasure]) refl emeasure-restrict-space simp-all
also have ... = \{N + emeasure ?R (the o m) - '
\{v, bs\}. map-option fst (f v bs) ∈ A ∧ v ∈ range-rm m \in space ?R)
unfolding bind-rai-def N-def space-R apfst-def
by (intro arg-cong2[where f=emeasure])
(simp-all add: set-eq-iff in-range-rmI split:option.split bind-splits)
also have \( \forall N + \text{emeasure (distr } \forall R (D \otimes M) (\text{the } o m)) \) 
\( \{ (v,b) . \text{map-option fst (f v bs) } \in A \wedge v \in \text{ range-rm m} \) 
\( \text{using 2 by \{(intro cong2)[where f=\(+\)]\ emeasure-distr[\text{symmetric}] \) } 
the-f-measurable map-prod-measurable \text{wf-m} \text{ simp-all} \) 
also have \( \forall N + \text{emeasure (distr } \forall R D (\text{fst } o \text{the } o m) \otimes M) B \) 
\( \{ (v,b) . \text{map-option fst (f v bs) } \in A \wedge v \in \text{ range-rm m} \) 
\( \text{unfolding N-def remainder-indep[OF \text{wf-m}] by simp} \) 
also have \( \forall N + \int^+ y. \text{emeasure B} \) 
\( \{ bs. \text{map-option fst (f v bs) } \in A \wedge v \in \text{range-rm m} \} \text{\partial \text{distr } \forall R D (\text{fst } o \text{the } o m) \) 
\( \text{using 2 by \{(subst coin-space.emeasure-pair-measure-alt) simp-all add:vimage-def comp-assoc} \) 
also have \( \forall N + \int^+ y. \text{emeasure B} \) 
\( \{ bs. \text{map-option fst (f (\text{fst } \text{the } (m x))) } \in A \wedge \text{fst } \text{the } (m x) ) \in \text{range-rm m} \} \text{\partial B} \) 
\( \text{using N-meas unfolding N-def using } \text{nn-integral-restrict-space} \) 
\( \text{by \{(subst \text{nn-integral-restrict-space) simp-all}} \) 
also have \( \forall N + \int^+ y x. \text{emeasure (distr-rai (f (\text{fst } \text{the } (m x)))) A } \) \( \text{partial} \) 
\( \text{unfolding distr-rai-def N-def by \{(intro cong2)[where f=\(+\)]\ set-nn-integral-cong refl cong2[where f=\text{emeasure}] \) } 
\( \text{(auto intro:in-range-\text{nn-integral})} \) 
also have \( \forall N + \int^+ y x. \text{emeasure (distr-rai (f (\text{fst } \text{the } (m x)))) A } \) \( \text{partial} \) 
\( \text{using N-meas N-meas’} \) 
\( \text{by \{(intro cong2)[where f=\(+\)]\ set-nn-integral-cong refl \text{emeasure-distr[\text{symmetric}] \) } 
\( \text{distr-rai-measurable[OF \text{wf-f}]}) \text{(auto intro:in-range-\text{nn-integral})} \) 
also have \( \forall N + \int^+ y x. \text{indicator \{bs. } x \notin N \wedge \text{None } \in A \} x \) \( \text{partial} \) 
\( \text{using N-meas’ N-meas by \{(intro \text{nn-integral-add[\text{symmetric}]}) simp} \) 
also have \( \forall N + \int^+ y x. \text{indicator } (-N) x \times \text{indicator A None} \) \( \text{partial} \) 
\( \text{indicator N x } \times \text{emeasure (distr-rai (f (\text{fst } \text{the } (m x)))) A } \) \( \text{partial} \) 
\( \text{unfolding N-def by \{(intro cong2)[where f=\text{nn-integral}] ext refl cong2[where f=\(+\)] \) } 
\( \text{(simp-all split:split-indicator}) \) 
also have \( \forall N + \int^+ y x. \text{emeasure (case m x of None } \Rightarrow \text{return } D \text{ None | Some x } \Rightarrow \text{distr-rai (f (\text{fst } x))) A } \) \( \text{partial} \) 
\( \text{unfolding N-def by \{(intro cong2)[where f=\text{nn-integral}] ext \) } 
\( \text{auto split:split-indicator option.split} \) 
also have \( \forall N + \int^+ y x. \text{emeasure (if (\text{map-option fst } o m) x } \in \text{Some ‘ range-rm m then distr-rai (f (\text{the } ((\text{map-option fst } o m) x)))} \) \( \text{partial} \) 
\( \text{then return } D \text{ None} A \) \( \text{partial} \) 
\( \text{by \{(intro cong2)[where f=\text{nn-integral}] arg cong2[where f=\text{emeasure}] refl ext \) } 
\( \text{(auto simp add: in-range-\text{nn-integral vimage-def split:option.splits})} \) 
also have \( \forall N + \int^+ y x. \text{emeasure (if } x \in \text{Some ‘ range-rm m then \?m } \in \text{Some ‘ range-rm m then \?m \in D None A } \) \( \text{partial} \) 
\( \text{unfolding distr-rai-def using distr-rai-measurable[OF \text{wf-m}] \) 
\( \text{by \{(intro \text{nn-integral-distr[\text{symmetric}]]) simp-all add:comp-def} \) 
also have \( \forall N + \text{emeasure } \forall RHS A \) 
\( \text{using 3 none-measure-subprob-algebra \) 
\( \text{by \{(intro emeasure-bind[\text{symmetric}] \text{where N=D}) \text{(auto simp add:distr-rai-def Pi-def} \) 
finally show \( \text{emeasure } \forall L A = \text{emeasure } \forall RHS A \) 
\( \text{by simp} \) 
\text{qed} \)
lemma return-discrete: return D x = return-pmf x
  by (intro measure-eqI) auto

lemma distr-rai-return: distr-rai (return-rai x) = return D (Some x)
  unfolding return-rai-def distr-rai-def by (simp add:comp-def)

lemma distr-rai-return\': distr-rai (return-rai x) = return-spmf x
  unfolding distr-rai-return return-discrete by auto

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

definition ord-rai :: 'a random-alg-int ⇒ 'a random-alg-int ⇒ bool
  where ord-rai = fun-ord (flat-ord None)

definition lub-rai :: 'a random-alg-int set ⇒ 'a random-alg-int
  where lub-rai = fun-lub (flat-lub None)

lemma random-alg-int-pd-fact:
  partial-function-definitions ord-rai lub-rai
  unfolding ord-rai-def lub-rai-def
  by (intro partial-function-lift flat-interpretation)

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

lemma wf-lub-helper:
  assumes ord-rai f g
  assumes wf-on-prefix f p r
  shows wf-on-prefix g p r
proof
  have f (cshift p cs) = Some (r, cs) for cs
    proof
      have f (cshift p cs) = Some (r,cs)
        using assms(2) unfolding wf-on-prefix-def by auto
      moreover have flat-ord None (f (cshift p cs)) (g (cshift p 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 \( r \in R \implies \text{wf-random } r \)
shows \( \text{wf-random (lub-rai } R) \)
proof (rule wf-randomI)
  fix \( bs \)
  assume \( a \colon \text{lub-rai } R \neq \text{None} \)
  define \( S \) where \( S = (\lambda x. x bs) ' R \)
  have \( 0 \colon \text{lub-rai } R \ bs = \text{flat-lub } \text{None } S \)
  unfolding S-def lub-rai-def fun-lub-def 
  by (intro arg-cong2[where \( f = \text{flat-lub} ]] \text{auto})
  have \( \text{lub-rai } R \ bs = \text{None } \) if \( S \subseteq \{ \text{None} \} \)
  using that unfolding 0 flat-lub-def by auto
  hence \( \neg (S \subseteq \{ \text{None} \}) \)
  using a by auto
  then obtain \( r \) where \( 1 \colon r \in R \) and \( 2 \colon r \ bs \neq \text{None} \)
  unfolding S-def by blast
qd

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

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

lemma distr-rai-lub:
  assumes \( F \neq \{ \} \)
  assumes Complete-Partial-Order.chain ord-rai F
  assumes \( \text{wf-f: } \bigwedge f. f \in F \implies \text{wf-random } f \)
  assumes None \( \notin A \)
  shows \( \text{emeasure (distr-rai (lub-rai } F) A = (SUP f \in F. \text{emeasure (distr-rai } f) A) (is } \mathbb{L} = \mathbb{R}) \)
proof
  have \( \text{wf-lub: } \text{wf-random (lub-rai } F) \)
  by (intro wf-lub assms)
  have \( 4 \colon \text{ord-rai } f \ (\text{lub-rai } F) \) if \( f \in F \) for \( f \)
  using that random-alg-int-pd.lub-upper[OF assms(2)] by simp
  have \( 0 \colon \text{map-option } \text{fst (lub-rai } F \ bs) \in A \iff (\exists f \in F. \text{map-option } \text{fst (f } bs) \in A) \) for \( bs \)
  proof
    assume \( \exists f \in F. \text{map-option } \text{fst (f } bs) \in A \)
    then obtain \( f \) where \( 3 \colon \text{map-option } \text{fst (f } bs) \in A \) and \( 5 \colon f \in F \)
    by auto
    show \( \text{map-option } \text{fst (lub-rai } F \ bs) \in A \)
    by (rule ord-rai-mono[OF 4[OF 5]]) (use 3 assms(4) in auto)
  next
    assume \( \text{map-option } \text{fst (lub-rai } F \ bs) \in A \)
    then obtain \( y \) where \( 6 \colon \text{lub-rai } F \ bs = \text{Some } y \) Some \( (f \ y) \in A \)
  qed

40
using assms(4) by (cases lub-rai F bs) auto

hence \( f \) bs = None \lor f bs = Some \( y \) if \( f \in F \) for \( f \)

using \( \{\text{OF that}\} \) unfolding ord-rai-def fun-ord-def flat-ord-def by auto

moreover have lub-rai F bs = None if \( \forall f \in F \Rightarrow f \) bs = None

using that unfolding lub-rai-flat lub-flat-def fun-lub-def by auto

ultimately obtain \( f \) where \( f \) bs = Some \( y \) if \( f \in F \)

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\}) \( \cdot\) 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 topspace 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 B (is \(?\)L1 \( \in\) -) if \( \text{wf-random} f \) for \( f \)

using distr-rai-measurable[OF that]

by (intro measurable-sets-coin-space[where \( P = \lambda x. \) x \( \in\) A and A=A]) (auto simp:comp-def)

have \(?\)L = emeasure B ((map-option fst \( \circ\) lub-rai F) \( \cdot\) (A \( \cap\) space B))

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

also have \( \ldots\) = emeasure B \{x. map-option fst (lub-rai F x) \( \in\) A\}

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

also have \( \ldots\) = emeasure B \{\( \bigcup f \in F. \) \{bs. map-option fst (f bs) \( \in\) A\}\}

unfolding 0 by (intro arg-cong2[where \( f = \text{emeasure}\)] auto

also have \( \ldots\) = Sup (emeasure B \( \cdot\) (\( \lambda f. \) \{bs. map-option fst (f bs) \( \in\) A\}) \( \cdot\) F)

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

auto

also have \( \ldots\) = (SUP f \( \in\) F. (emeasure B \{bs. map-option fst (f bs) \( \in\) A\}))

unfolding image-image by simp

also have \( \ldots\) = (SUP f\( \in\)F. emeasure B ((map-option fst \( \circ\) f) \( \cdot\) (A \( \cap\) space B))

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

also have \( \ldots\) = \(?\)R

unfolding distr-rai-def using distr-rai-measurable[OF wf-f] by (intro arg-cong[where \( f = \text{Sup}\)] image-cong ext emeasure-distr[symmetric]) auto

finally show \(?\)thesis

by simp

Qed

lemma distr-rai-ord-rai-mono:

assumes \( \text{wf-random} f \) \( \text{wf-random} g \) 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 \(1 : g = \{\text{lub-rai} \{f,g\}\}\)
  by (intro random-alg-int-pd.leq-refl) auto

have emeasure (distr-rai f) \(A \leq (\text{SUP} x \in \{f,g\}, \text{emeasure (distr-rai x)}) A\)
  using prob-space-distr-assms(1,2) prob-space.measure-le-1
by (intro cSup-upper bdd-aboveI[where M=1]) auto
also have \(\ldots = \text{emeasure (distr-rai (lub-rai \{f,g\}))} A\)
  using assms by (intro distr-rai-lub[symmetric] 0) auto
also have \(\ldots = \text{emeasure (distr-rai g) A}\)
  using 1 by auto
finally show ?thesis
  by simp
qed

lemma distr-rai-None: distr-rai (\(\lambda\). 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 \(\forall y. \text{ord-rai (g1 y) (g2 y)}\)
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 \(\gg\) g1) bs)
  case None
  then show ?thesis by (simp add:flat-ord-def)
next
  case (Some a)
  then obtain y bs' where 0: \(f1 \leq g1\) y bs' \(\neq None\) and \(f1 bs \neq None\)
    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 \(f1 \gg g1\) bs = \(f2 \gg g2\) 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-alg monad, that can be used to represent executable randomized algorithms. It is a type-definition based on the internal representation from Section 4 with the wellformedness restriction.

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

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

theory Randomized-Algorithm
imports
  Randomized-Algorithm-Internal
begin

A stronger variant of pmf-eqI.

lemma pmf-eq-iff-le:
  fixes p q :: 'a pmf
  assumes \( \forall x. \text{pmf } p \ x \leq \text{pmf } q \ x \)
  shows \( p = q \)
proof
  have \( \{ x. \text{pmf } q \ x - \text{pmf } p \ x \text{ count-space UNIV} \} = 0 \)
  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 \( \forall x. \text{pmf } p \ x = \text{pmf } q \ x \) 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 \( \geq \) weight-spmf q
  assumes ord-spmf (=) p q
  shows \( p = q \)
proof
  have \( \forall x. \text{spmff } p \ x \leq \text{spmff } q \ x \)
  using ord-spmf-eq-leD[OF assms(2)] by simp
  moreover have pmf p None \( \leq \) pmf q None
  using assms(1) unfolding pmf-None-eq-weight-spmf by auto
  ultimately have pmf p x \( \leq \) pmf q x for x by (cases x) auto
  thus ?thesis using pmf-eq-iff-le by auto
qed

lemma wf-empty: wf-random (\x. None)
  unfolding wf-random-def by auto

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

setup-lifting type-definition-random-alg
lift-definition return-ra :: 'a ⇒ '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 ⇒ ('a ⇒ 'b random-alg) ⇒ '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 (λx. 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 ⇒ 'a random-alg is (λF. if Complete-Partial-Order.chain ord-rai F then lub-rai F else (λx. None))
using wf-lub wf-empty by auto

lift-definition ord-ra :: 'a random-alg ⇒ 'a random-alg ⇒ bool is ord-rai.

lift-definition run-ra :: 'a random-alg ⇒ coin-stream ⇒ 'a option is (λf 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} ≠ 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 B. map-option fst (f bs) ∈ Some ' range-rm f ∪ {None}
unfolding range-rm-def
by (intro AE-12) (auto simp:image-iff split:option.split)

hence AE x in distr-rai f. x ∈ Some ' range-rm f ∪ {None}
unfolding distr-rai-def using distr-rai-measurable[OF assms]
by (subst AE-distr-iff) auto
moreover have countable (Some ' range-rm f ∪ {None})
  using countable-range[OF assms] by simp
moreover have p.events = UNIV
  unfolding distr-rai-def by simp
ultimately show ?C
  by (intro iffD2[OF p.AE-support-countable] exI[where x= Some ' range-rm f ∪ {None}]) auto
qed

lift-definition spmf-of-ra :: 'a random-alg ⇒ 'a spmf is distr-ra
  using distr-ra-is-pmf by metis

lemma used-bits-distr-is-pmf:
  assumes wf-random f
  shows AE x in used-bits-distr f. measure (used-bits-distr f) {x} ≠ 0 (is ?C)
proof –
  show prob-space (used-bits-distr f)
    unfolding used-bits-distr-def by simp
  have p.events = UNIV
    unfolding used-bits-distr-def by simp
  thus ?C
    by (intro iffD2[OF p.AE-support-countable] exI[where x= UNIV]) auto
qed

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

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

lemma wf-rep-rand-alg:
  wf-random (Rep-random-alg f)
  using Rep-random-alg by auto

lemma set-pmf-spmf-of-ra:
  set-pmf (spmf-of-ra f) ⊆ Some ' range-rm (Rep-random-alg f) ∪ {None}
proof
  let ?f = Rep-random-alg f
  fix x assume x ∈ set-pmf (spmf-of-ra f)
  hence pmf (spmf-of-ra f) x > 0
    using pmf-positive by metis
  hence measure (distr-rai ?f) {x} > 0
    by (subst spmf-of-ra, rep-eq[symmetric]) (simp add: pmf.rep-eq)
  hence 0 < measure B [ω. map-option fst (if ω) = x]
    by (subst (asm) measure-distr) (simp add: image-def space-coin-space)
  moreover have {ω. map-option fst (if ω) = x} = {} if x /∈ range (map-option fst ∘ ?f)
    using that by (auto simp: set-eq-iff image-iff)

45
hence $\text{measure } B \{ \omega. \text{map-option} \text{fst} (\omega) = x \} = 0$ if $x \notin \text{range} (\text{map-option} \text{fst} \circ \omega)$

using that by simp

ultimately have $x \in \text{range} (\text{map-option} \text{fst} \circ \omega)$

by auto

thus $x \in \text{Some ' range-rm} (\text{Rep-random-alg f}) \cup \{ \text{None} \}$

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

dqed

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 $\text{thesis}$

using measure-pmf-inject by blast

dqed

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 $\text{thesis}$

using measure-pmf-inject by blast

dqed

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 $0: x \in \text{Some ' range-rm} ?f \lor x = \text{None}$ if $x \in \text{set-pmf} (\text{spmf-of-ra} f)$ for $x$

using that set-pmf-spmf-of-ra by auto

have $\text{measure-pmf} ?L = \text{distr-rai} (?f \gg ?g)$

also have $\ldots = \text{distr-rai} ?f \gg$

$(\lambda x. \text{if } x \in \text{Some ' range-rm} ?f \text{ then } \text{distr-rai} (?g (\text{the } x)) \text{ else return } \mathcal{D} \text{ None})$

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

also have $\ldots = \text{measure-pmf} (\text{spmf-of-ra} f) \gg$

$(\lambda x. \text{measure-pmf} (\text{if } x \in \text{Some ' range-rm} ?f \text{ then } \text{spmf-of-ra} (g (\text{the } x)) \text{ else return-pmf None}))$

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

also have $\ldots = \text{measure-pmf} (\text{spmf-of-ra} f) \gg$

$(\lambda x. \text{if } x \in \text{Some ' range-rm} ?f \text{ then } \text{spmf-of-ra} (g (\text{the } x)) \text{ else return-pmf None}))$

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

also have $\ldots = \text{measure-pmf} ?R$

using $0$ unfolding bind-spmf-def

by (intro arg-cong[where $f=\text{measure-pmf}$ bind-spmf-cong refl]) (auto split:option.split)

finally have $\text{measure-pmf} ?L = \text{measure-pmf} ?R$ by simp

thus $\text{thesis}$

using measure-pmf-inject by blast

dqed

lemma spmf-of-ra-mono:

assumes ord-ra $f \ g$

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

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

lemma spmf-of-ra-lub-ra-empty:
  spmf-of-ra (lub-ra {}) = return-pmf None (is ?L = ?R)
proof –
  have measure-pmf ?L = distr-rai (lub-ra {}) using assms unfolding ord-ra
    rep-eq Complete-Partial-Order.chain-def by auto
  also have ... = distr-rai (λ-. None) unfolding lub-ra-def fun-lub-def flat-lub-def by auto
  also have ... = measure-pmf ?R unfolding distr-rai-None by simp
  finally have measure-pmf ?L = measure-pmf ?R by simp
  thus ?thesis using measure-pmf-inject by auto
qed

lemma spmf-of-ra-lub-ra:
  fixes A :: ′a random-alg set
  assumes Complete-Partial-Order.chain ord-ra A
  shows spmf-of-ra (lub-ra A) = lub-spmf (spmfl-ra ′ A) (is ?L = ?R)
proof (cases A ≠ {}) case True
  have 0:Complete-Partial-Order.chain ord-ra (Rep-random-alg ′ A)
    using assms unfolding ord-ra.rep-eq Complete-Partial-Order.chain-def by auto
  have 1:Complete-Partial-Order.chain (ord-spmf (=)) (spmfl-ra ′ A)
    using spmf-of-ra-mono by (intro chain-imageI[OF assms]) auto
  show ?thesis
    proof (rule spmf-eqI)
      fix x :: ′a
      have ennreal (spmf ?L x) = emeasure (distr-rai (lub-rai (Rep-random-alg ′ A))){Some x}
      also have ... = (SUP f ∈ Rep-random-alg ′ A. emeasure (distr-rai f) {Some x})
        using True wf-rep-rand-alg by (intro distr-rai-lub 0) auto
      also have ... = (SUP p ∈ A. ennreal (spmf (spmfl-ra p) x))
        unfolding emeasure-pmf-single[symmetric] spmf-of-ra.rep-eq by (simp add:image-image)
      also have ... = (SUP p ∈ spmf-of-ra ′ A. ennreal (spmf p x))
        by (simp add:image-image)
      also have ... = ennreal (spmf ?R x)
        using True by (intro ennreal-spmf-lub-spmf[symmetric] 1) auto
      finally have ennreal (spmf ?L x) = ennreal (spmf ?R x)
        by simp
      thus spmf ?L x = spmf ?R x
        by simp
    qed
  next
    case False
    thus ?thesis using spmf-of-ra-lub-ra-empty by simp

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 ⋀ A. Complete-Partial-Order.chain ord (f ' A) =⇒ 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 unfolding img-ord-def
  have 0: Complete-Partial-Order.chain ord (f ' A)
    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 simp: img-ord-def)
  thus ?thesis
    unfolding img-ord-def assms[1][OF 0] by auto
qed

moreover have img-ord f ord x (l2 A)
  if x ∈ A Complete-Partial-Order.chain (img-ord f ord) A for x A
proof –
  have 0: Complete-Partial-Order.chain ord (f ' A)
    unfolding chain-def img-ord-def by auto
  have ord (f x) (l1 (f ' A))
    using that 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

moreover have img-ord f ord (l2 A) z
  if Complete-Partial-Order.chain (img-ord f ord) A (∀ x. x ∈ A — img-ord f ord x z)
  for z A
proof –
  have 0: Complete-Partial-Order.chain ord (f ' A)
    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-rai. rep-eq)

qed

interpretation random-alg-pf: partial-function-definitions ord-ra lub-rai
  using random-alg-pfd by auto

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

lemma bind-mono-aux-ra:
  assumes ord-ra f1 f2 \( \forall y. \) 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 \( \forall y. \) mono-ra \((C y)\)
  shows mono-ra \((\lambda f. bind-ra (B f) (Ay. C y f))\)
  using assms by (intro monotoneI bind-mono-aux-ra) (auto simp: monotone-def)

definition map-ra :: \( \alpha \Rightarrow \beta \Rightarrow \alpha \) random-alg \Rightarrow \beta \)
  where map-ra f p = p >> (\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-cone-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 :: \( \alpha \Rightarrow \beta \Rightarrow \alpha \) random-alg \Rightarrow bool \)
  where rel-spmf-of-ra \((q p)\) \iff 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: \( \forall (p, q) \in Y. \Rightarrow \rel-spmf-of-ra p q \) by auto
  have chain1: Complete-Partial-Order.chain \(\ord-spmf (\_)\) \(\f 1 Y\)
    and chain2: Complete-Partial-Order.chain \(\ord-ra\) \(\snd \1 Y\)
using chain by(rule chain-image I; clarsimp)+
from Y have Y1: fst ' Y ≠ {} and Y2: snd ' Y ≠ {} 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 image-cong) (auto simp add: rel-spmf-of-ra-def lub-spmf)
finally have rel-spmf-of-ra (lub-spmf (fst ' Y)) (lub-ra (snd ' Y))
  unfolding rel-spmf-of-ra-def .
then show ?P
  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 ]
  ===> ccpo.admissible lub ord (λx. rel-spmf-of-ra (f x) (g x))
  by (rule admissible-subst[OF admissible-rel-spmf-of-ra, where f=λx. (f x, g x), simplified])
  (rule mcont-Pair)

lemma mcont2mcont-spmf-of-ra [THEN spmf.mcont2mcont, cont-intro]:
  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: λx. mono-spmf (λf. F f x)
  and g: λx. mono-ra (λf. G f x)
  using f g
proof (rule parallel-fixp-induct-1-1[OF partial-function-defnitions-spmf random-alg-pfd - reflexive reflexive, 
  where P=(A ===> 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))
    (λx. (A ===> rel-spmf-of-ra) (fst x) (snd x))
    unfolding rel-fun-def
    by (rule admissible-all admissible-imp cont-intro)+
  show (A ===> rel-spmf-of-ra) (λ-. lub-spmf {}) (λ-. lub-ra {})
    by (auto simp: rel-fun-def rel-spmf-of-ra-lub-ra-empty)
  show (A ===> rel-spmf-of-ra) (F f) (G g) if (A ===> rel-spmf-of-ra) f g for f g
    using that by (rule rel-funD[OF param])
qed

lemma return-ra-transfer[transfer-rule]: (=) ===> rel-spmf-of-ra) return-spmf return-ra
  unfolding rel-fun-def rel-spmf-of-ra-return by simp

lemma bind-ra-transfer[transfer-rule]:
  (rel-spmf-of-ra ===> (=) ===> rel-spmf-of-ra) ===> rel-spmf-of-ra) bind-spmf bind-ra
  unfolding rel-fun-def rel-spmf-of-ra-return 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]:
((=) ===> rel-spmf-of-ra ===> rel-spmf-of-ra) map-spmf map-ra
unfolding rel-fun-def rel-spmf-of-ra-def spmf-of-ra-map by simp
end

declare [[function-internals]]

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

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

lemma pmf-of-spmf: map-pmf the (spmf-of-pmf x) = x
  by (simp add:map-pmf-comp spmf-of-pmf-def)

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

lemma pmf-of-ra-coin: pmf-of-ra (coin-ra) = coin-pmf (is ?L = ?R)
proof –
  have 0:spmf-of-ra (coin-ra) = spmf-of-pmf (pmf-of-set UNIV)
    unfolding spmf-of-ra-coin spmf-of-set-def by simp
  thus ?thesis
    unfolding 0 pmf-of-ra-def pmf-of-spmf coin-pmf-def by simp
qed

lemma pmf-of-ra-return: pmf-of-ra (return-ra x) = return-pmf x
  unfolding pmf-of-ra-def spmf-of-ra-return by simp

lemma pmf-of-ra-bind:
  assumes terminates-almost-surely f
  shows pmf-of-ra (f >>= g) = pmf-of-ra f >>= (λx. pmf-of-ra (g x)) (is ?L = ?R)
proof –
  have 0:x ≠ None if x ∈ set-pmf (spmf-of-ra f) for x
    using assms that unfolding terminates-almost-surely-def
    by (meson lossless-iff-set-pmf-None)

  have ?L = spmf-of-ra f >>= (λx. map-pmf the (case-option (return-pmf None) (spmf-of-ra o g) x))
    unfolding pmf-of-ra-def spmf-of-ra-bind bind-spmf-def map-bind-pmf comp-def by simp
  also have ... = spmf-of-ra f >>=
    (λx. (case x of None ⇒ return-pmf (the None) | Some x ⇒ pmf-of-ra (g x)))
    unfolding map-pmf-def comp-def pmf-of-ra-def map-pmf-def
    by (intro arg-cong2[where f=bind-pmf] refl ext) (simp add:bind-return-pmf split:option.split)
  also have ... = spmf-of-ra f >>= (λx. pmf-of-ra (g (the x)))
    using 0 by (intro bind-pmf-cong refl) (auto split:option.split)
  also have ... = ?R
unfolding pmf-of-ra-def map-pmf-def by (simp add: bind-assoc-pmf bind-return-pmf)

finally show ?thesis
  by simp

qed

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


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 \( \forall x . x \in \text{set-pmf} (\text{pmf-of-ra} f) \implies \text{terminates-almost-surely} (g x) \)
  shows terminates-almost-surely (f >>= g)

proof
  have 0: None \notin \text{set-pmf} (\text{spmf-of-ra} f)
    using assms(1) lossless-iff-set-pmf-None unfolding terminates-almost-surely-def
    by blast

  hence \( \exists x \in \text{set-pmf} (\text{spmf-of-ra} f) \iff x \in \text{the ' set-pmf} (\text{spmf-of-ra} f) \) for x
    by (metis image-iff option.collapse option.sel)

  hence \( \text{set-spmf} (\text{spmf-of-ra} f) = \text{set-pmf} (\text{pmf-of-ra} f) \)

  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 =

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 verifi-
cation of coin-flip usage, the morphism \( \text{tspmf-of-ra} \) introduced in Section 7 is more useful.

theory Tracking-Randomized-Algorithm
imports Randomized-Algorithm
begin

definition track-random-bits :: \('a \text{ random-alg-int} \Rightarrow (\'a \times \text{ nat}) \text{ random-alg-int}\)
where track-random-bits f bs =
  do 
    \((r,bs')\) ← f bs;
    n ← consumed-bits f bs;
    Some ((r,n),bs')

lemma track-random-bits-Some-iff:
  assumes track-random-bits f bs \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 (λp. ((eval-rm f p, length p), cdrop (length p) bs)) (consumed-prefix f bs)
  proof (cases consumed-prefix f bs)
    case None
      hence f bs = None
      by (subst wf-random-alt[OF assms(1)]) simp
    then show \?thesis
      unfolding track-random-bits-def None by simp
    next
    case (Some a)
      hence f bs = Some (eval-rm f a, cdrop (length a) bs)
      by (subst wf-random-alt[OF assms(1)]) simp
    then show \?thesis
      unfolding track-random-bits-def Some consumed-bits-def by simp
  qed

lemma track-rb-coin:
  track-random-bits coin-rai = coin-rai \(\cong\) (λb. return-rai (b,1)) (is \?L = \?R)
  proof (rule ext)
    fix bs :: coin-stream
    have wf-on-prefix coin-rai [chd bs] (chd bs)
      unfolding wf-on-prefix-def coin-rai-def by simp
    moreover have cprefix [chd bs] bs
      unfolding cprefix-def by simp
    ultimately have \{p ∈ ptree-rm (coin-rai). cprefix p bs\} = \{[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 :: coin-stream
  have wf-on-prefix (return-rai x) [] x
    unfolding wf-on-prefix-def return-rai-def by simp
  moreover have cprefix [] bs
    unfolding cprefix-def by simp
  ultimately have \{ p \in ptree-rm (return-rai x), cprefix p bs \} = {[]}
    by (intro prefixes-singleton) (auto simp:ptree-rm-def)
  hence consumed-prefix (return-rai x) bs = Some []
    unfolding consumed-prefix-def by simp
  hence consumed-bits (return-rai x) bs = Some 0
    unfolding consumed-bits-def by simp
  thus ?L bs = ?R bs
    unfolding track-random-bits-def by (simp add:return-rai-def)
  qed

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

lemma consumed-prefix-imp-prefix:
  assumes consumed-prefix m bs = Some p
  shows cprefix p bs
    using assms unfolding consumed-prefix-def the-elem-opt-Some-iff[OF prefixes-at-most-one] by blast

lemma consumed-prefix-bindI:
  assumes consumed-prefix m bs = Some p
  assumes consumed-prefix (f (eval-rm m p)) (cdrop (length p) bs) = Some q
  shows consumed-prefix (m \gg= f) (p@q)
proof
  define r where r = eval-rm m p
  have 0: wf-on-prefix m p r
    unfolding r-def using consumed-prefix-imp-wf[OF assms(1)] by simp
  have consumed-prefix (f r) (cdrop (length p) bs) = Some q
    using assms(2) unfolding r-def by simp
  hence 1: wf-on-prefix (f r) q (eval-rm (f r) q)
    unfolding consumed-prefix-imp-wf by auto
  have wf-on-prefix (m \gg= f) (p@q) (eval-rm (f r) q)
    by (intro wf-on-prefix-bindI[where r=r] 0 1)
  hence p@q \in ptree-rm (m \gg= f)
    unfolding ptree-rm-def by auto
  moreover have cprefix p bs cprefix q (cdrop (length p) bs)
    unfolding cprefix-def by (metis length-append ctake-add)
  ultimately have \{ p \in ptree-rm (m \gg= f), cprefix p bs \} = \{p@q\}
by (intro prefixes-singleton) auto
thus \$thesis
  unfolding consumed-prefix-def by simp
qed

lemma track-rb-bind:
assumes \$wf-random m
assumes \$\lambda x. x \in range-rm m \implies \$wf-random (f x)
shows track-random-bits (m \gg f) = track-random-bits m \gg
(\$\lambda (r,n), \$track-random-bits (f r) \gg (\$\lambda (r',m), \$return-rai (r',n+m))) (is \$L = \$R)
proof (rule ext)
fix bs :: coin-stream
have \$wf-bind: \$wf-random (m \gg f)
  by (intro \$wf-bind assms)
consider (a) m bs = None | (b) m bs \neq None \wedge (m \gg f) bs = None | (c) (m \gg 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 \gg 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-rmf 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 \gg 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 OF 0(2)])
then show \$thesis using 3 unfolding eval-rm-def bind-rai-def wf-on-prefix-def by simp

55
ultimately have

\[ R \, bs = \text{Some} \left( (\text{eval-rm} \, (m \gg f) \, (p@q), \text{length} \, p + \text{length} \, q), \text{cdrop} \, (\text{length} \, p + \text{length} \, q) \, bs) \]  

unfolding bind-rai-def return-rai-def bs'-def by simp

also have \[ ?L \, bs \]
unfolding track-random-bits-alt \[ OF \, wf-bind \] 3 by simp

finally show \[ ?\text{thesis} \] by simp

qed

qed

lemma track-random-bits-mono:

assumes \[ \text{wf-random} \, f \, \text{wf-random} \, g \]

assumes \[ \text{ord-rai} \, f \, g \]

shows \[ \text{ord-rai} \, (\text{track-random-bits} \, f) \, (\text{track-random-bits} \, g) \]

proof –

have \[ \text{track-random-bits} \, f \, bs = \text{track-random-bits} \, g \, bs \]

if \[ \text{track-random-bits} \, f \, bs \neq \text{None} \] for \[ bs \]

proof –

have \[ f \, bs \neq \text{None} \] using that \[ \text{track-random-bits-Some-iff} \] by simp

then obtain \[ r \, bs' \, \text{where} \, f \, bs = \text{Some} \, (r, \, bs') \] by auto

then obtain \[ p \, \text{where} \, 0:{\text{wf-on-prefix}} \, f \, p \, r \, \text{and} \, 2:{\text{cprefix}} \, p \, bs \]

using assms(1) unfolding \[ \text{wf-random-def} \] by (auto split: option.split-asm)

have \[ 1: \text{wf-on-prefix} \, g \, p \, r \]

using \[ \text{wf-lub-helper} \,[OF \, \text{assms}(3)] \] 0 by blast

have \[ \text{track-random-bits} \, h \, bs = \text{Some} \, ((r, \, \text{length} \, p), \text{cdrop} \, (\text{length} \, p) \, bs) \]

if \[ \text{wf-on-prefix} \, h \, p \, r \, \text{wf-random} \, h \, \text{for} \, h \]

proof –

have \[ p \in \text{ptree-rm} \, h \]

using that \[ \text{unfolding} \, \text{ptree-rm-def} \] by auto

hence \[ \{ p \in \text{ptree-rm} \, h, \, \text{cprefix} \, p \, bs \} = \{ p \} \]

using \[ 2 \] by (intro prefixes-singleton) auto

hence \[ \text{consumed-prefix} \, h \, bs = \text{Some} \, p \]

unfolding \[ \text{consumed-prefix-def} \] by simp

moreover have \[ \text{eval-rm} \, h \, p = r \]

using \[ \text{that}(1) \] unfolding \[ \text{wf-on-prefix-def} \, \text{eval-rm-def} \] by simp

ultimately show \[ ?\text{thesis} \]

unfolding \[ \text{track-random-bits-alt} \,[OF \, \text{that}(2)] \] by simp

qed

thus \[ ?\text{thesis} \]

using \[ 0 \, 1 \] \[ \text{assms}(1, 2) \] by simp

qed

thus \[ ?\text{thesis} \]

unfolding \[ \text{ord-rai-def} \, \text{fun-ord-def} \, \text{flat-ord-def} \] by blast

qed

lemma \[ \text{wf-track-random-bits} \]:

assumes \[ \text{wf-random} \, f \]

shows \[ \text{wf-random} \, (\text{track-random-bits} \, f) \]

proof (rule \[ \text{wf-randomI} \])

fix \[ bs \]

assume \[ \text{track-random-bits} \, f \, bs \neq \text{None} \]

hence \[ f \, bs \neq \text{None} \] using \[ \text{track-random-bits-Some-iff} \] by blast

then obtain \[ r \, bs' \, \text{where} \, f \, bs = \text{Some} \, (r, \, bs') \]

by auto

then obtain \[ p \, \text{where} \, 0:{\text{wf-on-prefix}} \, f \, p \, r \, \text{cprefix} \, p \, bs \]

using \[ \text{assms} \] unfolding \[ \text{wf-random-def} \] by (auto split: option.split-asm)
proof

lemma untrack-random-bits:
assumes wf-random f
shows track-random-bits f ≫= (λx. return-rai (fst x)) = f (is ?L = ?R)
proof –

qed

lemma track-random-bits-lub-rai:
assumes Complete-Partial-Order.chain ord-rai A
assumes ∀r. r ∈ A → wf-random r
shows track-random-bits (lub-rai A) = lub-rai (track-random-bits ' A) (is ?L = ?R)
proof –

have 0:Complete-Partial-Order.chain ord-rai (track-random-bits ' A)
by (intro chain-imageI[OF assms(1)])

have ?L bs = ?R bs if ?L bs ≠ None for bs
proof –

have 1:lub-rai A bs ≠ None using that track-random-bits-Some-iff by simp
have lub-rai A bs = None if ∃f. f ∈ A → f bs = None
using that unfolding lub-rai-def fun-lub-def auto
then obtain f where f-in-A: f ∈ A and f bs ≠ None
using f by blast
hence consumed-prefix f bs ≠ None
using consumed-prefix-none-iff[OF assms(2)[OF f-in-A]] by simp
hence 2:track-random-bits bs ≠ None
unfolding track-random-bits-alt[OF assms(2)[OF f-in-A]] by simp
have ord-rai (track-random-bits f) (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 track-random-bits (lub-rai A) bs = track-random-bits f bs
using 2 unfolding ord-rai-def fun-ord-def auto

moreover have ord-rai (track-random-bits f) (lub-rai (track-random-bits ' A))
using f-in-A by (intro random-alg-int-pd.lub-upper[OF assms(1)])

hence lub-rai (track-random-bits ' A) bs = track-random-bits f bs
using 2 unfolding ord-rai-def fun-ord-def auto
ultimately show ?thesis by auto

qed

lemma flat-ord None (is ?L) (is ?R)
proof

unfolding flat-ord-def by blast

hence ord-rai ?L ?R
unfolding ord-rai-def fun-ord-def by simp

moreover have ord-rai (track-random-bits f) (track-random-bits (lub-rai A)) if f ∈ 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 ord-rai ?R ?L
by (intro random-alg-int-pd.lub-least auto)
ultimately show ?thesis by auto

qed

lemma untrack-random-bits:
assumes wf-random f
shows track-random-bits f ≫= (λx. return-rai (fst x)) = f (is ?L = ?R)
proof –
have \( ?L \) \( bs = ?R \) \( bs \) for \( bs \)

unfolding track-random-bits-alt[\( OF \) assms] bind-rai-def return-rai-def
by (subst wf-random-alt[\( OF \) assms]) (cases consumed-prefix \( f \) \( bs \), auto)
thus \(?thesis\)
by auto
qed

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

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


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


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

adhoc-overloading Monad-Syntax.bind bind-tra

Monad laws:

lemma return-bind-tra:
bind-tra (return-tra \( x \)) \( g \) = \( g \) \( x \)
unfolding bind-tra-def return-tra-def
by (simp add:bind-return-ra return-bind-ra)

lemma bind-tra-assoc:
bind-tra (bind-tra \( f \) \( g \)) \( h \) = bind-tra \( f \) (\( \lambda x \). bind-tra \( g \) \( x \)) \( h \)
unfolding bind-tra-def
by (simp add:bind-return-ra return-bind-ra bind-ra-assoc case-prod-beta' algebra-simps)

lemma bind-return-tra:
bind-tra \( m \) return-tra = \( m \)
unfolding bind-tra-def return-tra-def
by (simp add:bind-return-ra return-bind-ra)

lemma track-coin-use-bind:
fixes \( m \) :: \( 'a \) random-alg
fixes \( f \) :: \( 'a \Rightarrow 'b \) random-alg
shows track-coin-use \( m \gg \gg \) \( f \gg \gg ( \lambda r. \) track-coin-use \( f \) \( r ) \gg \gg \)
(is \( ?L = ?R \))
proof –

have \( \text{Rep-random-alg} \ ?L = \text{Rep-random-alg} \ ?R \)
unfolding track-coin-use.rep-eq bind-ra.rep-eq bind-tra-def
by (subst track-rb-bind) (simp-all add:wf-rep-random-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
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-ra (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 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-rai.rep-eq by simp
  also have ... = lub-rai (track-random-bits ' Rep-random-alg ' A)
  using wf-rep-rand-alg unfolding 2 by auto
  also have ... = Rep-random-alg ?R
  using 1 unfolding lub-rai.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 \(\wedge y.\) 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

lemma bind-tra-mono [partial-function-mono]:
  assumes mono-ra B and \(\wedge y.\) mono-ra (C y)
  shows mono-ra (\(\lambda f.\) bind-tra (B f) (\(\lambda y.\) C y f))
  using assms by (intro monotoneI bind-tra-def mono-tra-optional) (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
qed

lemma untrack-coin-use:
map-ra fst (track-coin-use f) = f (is ?L = ?R)
proof
have \(\text{Rep-random-alg} \ ?L = \text{Rep-random-alg} \ ?R\)
  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 \(?\text{thesis}\)
using Rep-random-alg-inject by auto
qed

definition rel-track-coin-use :: \(\text{('a \times \text{nat}) \rightarrow \text{random-alg} \Rightarrow \text{('a random-alg} \Rightarrow \text{bool}\)
where
rel-track-coin-use q p ←→ q = track-coin-use p

lemma admissible-rel-track-coin-use:
ccpo.admissible (prod-lub lub-ra lub-ra) (rel-prod ord-ra ord-ra) (case-prod rel-track-coin-use)
(is ccpo.admissible ?lub ?ord ?P)
proof (rule ccpo.admissibleI)
fix Y
assume chain: Complete-Partial-Order.chain ?ord Y
  and Y: Y ≠ \{}
  and R: \(\forall \ (p, q) \in Y. \ \text{rel-track-coin-use} \ p \ q\)
from R have R: \(\forall \ (p, q) \in Y. \ \text{rel-track-coin-use} \ p \ q\)
  by auto
have chain1: Complete-Partial-Order.chain (ord-ra) (fst ' Y)
  and chain2: Complete-Partial-Order.chain (ord-ra) (snd ' Y)
  using chain by (rule chain-imageI; clarsimp)+
from Y have Y1: fst ' Y ≠ \{}
  and Y2: snd ' Y ≠ \{}
  by auto
have lub-ra (fst ' Y) = lub-ra (track-coin-use (snd ' Y))
  unfolding image-image using R
  by (intro arg-cong[of - - lub-ra] image-cong) (auto simp: rel-track-coin-use-def)
also have \(\ldots\) = 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 \(?P \ (?\text{lub} \ Y)\)
  by (simp add: prod-lub-def)
qed

lemma admissible-rel-track-coin-use-cont [cont-intro]:
fixes ord
shows \[ \text{mcont} \text{ lub} \text{ ord} \text{ lub-ra ord-ra} \ f; \text{mcont} \text{ lub} \text{ ord} \text{ lub-ra ord-ra} \ g \]\ ⇒ \ccpo.admissible lub ord (\l x. \text{rel-track-coin-use} (f x) (g x))
by (rule admissible-subst[OF admissible-rel-track-coin-use, where f=\l x. (f x, g x), simplified])
  (rule mcont-Pair)

lemma mcont-track-coin-use:
\text{mcont} \text{ lub-ra ord-ra ord-ra} \text{ 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 \( f: \forall x. \text{mono-ra} \ (\lambda f. \ F f x) \)
and \( g: \forall x. \text{mono-ra} \ (\lambda f. \ G f x) \)
and \( \text{param}: \ ((A \Rightarrow \text{rel-track-coin-use}) \Rightarrow A \Rightarrow \text{rel-track-coin-use}) \ F G \)
shows \( (A \Rightarrow \text{rel-track-coin-use}) \) \((\text{random-alg-pf}. \text{fixp-fun} \ F) \) \((\text{random-alg-pf}. \text{fixp-fun} \ G) \)
using \( f g \)

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

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

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

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

end

7 Tracking SPMFs

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

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

theory Tracking-SPMF
  imports Tracking-Randomized-Algorithm
begin

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

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

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

end

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

adhoc-overloading Monad-Syntax.bind bind-tspmf

Monad laws:

**lemma** return-bind-tspmf:
bind-tspmf (return-tspmf x) g = g x
**unfolding** bind-tspmf-def return-tspmf-def map-spmf-conv-bind-spmf
**by** (simp add: apsnd-def map-prod-def)

**lemma** bind-tspmf-assoc:
bind-tspmf (bind-tspmf f g) h = bind-tspmf f (λx. bind-tspmf (g x) h)
**unfolding** bind-tspmf-def
**by** (simp add: case-prod-beta' algebra-simps map-spmf-conv-bind-spmf apsnd-def map-prod-def)

**lemma** bind-return-tspmf:
bind-tspmf m return-tspmf = m
**unfolding** bind-tspmf-def return-tspmf-def map-spmf-conv-bind-spmf apsnd-def
**by** (simp add: case-prod-beta')

**lemma** bind-mono-tspmf-aux:
assumes ord-spmf (=) f1 f2 □ y. ord-spmf (=) (g1 y) (g2 y)
shows ord-spmf (=) (bind-tspmf f1 g1) (bind-tspmf f2 g2)
**using** assms
**unfolding** bind-tspmf-def map-spmf-conv-bind-spmf
**by** (auto intro: bind-spmf-mono simp add: case-prod-beta')

**lemma** bind-mono-tspmf [partial-function-mono]:
assumes mono-spmf B and □ y. mono-spmf (C y)
sshows mono-spmf (λf. bind-tspmf (B f) (λy. C y f))
**using** assms
**by** (intro monotoneI bind-mono-tspmf-aux) (auto simp: monotone-def)

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

**bundle** ord-tspmf-notation
begin
  **notation** ord-tspmf ((/- ≤R -) [51 , 51] 50)
end

**bundle** no-ord-tspmf-notation
begin
  **no-notation** ord-tspmf ((/- ≤R -) [51 , 51] 50)
end

**unbundle** ord-tspmf-notation

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

**definition** expected-coin-usage-of-tspmf :: 'a tspmf ⇒ ennreal
where expected-coin-usage-of-tspmf p = (∫ x. x * pmf ennreal-of-enat (coin-usage-of-tspmf p))

**definition** expected-coin-usage-of-ra where
expected-coin-usage-of-ra p = (∫ x. x * pmf ennreal-of-enat (coin-usage-of-ra p))
definition result :: 'a tspmf ⇒ 'a spmf
where result = map-spmf fst

lemma coin-usage-of-tspmf-alt-def:
coin-usage-of-tspmf p = map-pmf (λx. case x of None ⇒ ∞ | Some y ⇒ enat y) (map-spmf snd p)
unfolding coin-usage-of-tspmf-def map-pmf-comp map-option-case
by (metis enat-def infinity-enat-def option.case-eq-if option.sel)

lemma coin-usage-of-tspmf-bind-return:
coin-usage-of-tspmf (bind-tspmf f (λx. return-tspmf (g x))) = (coin-usage-of-tspmf f)
unfolding bind-tspmf-def return-tspmf-def coin-usage-of-tspmf-alt-def map-spmf-bind-spmf
by (simp add:comp-def case-prod-beta map-spmf-conv-bind-spmf)

lemma coin-usage-of-tspmf-mono:
assumes ord-tspmf p q
shows measure (coin-usage-of-tspmf p) {..k} ≤ measure (coin-usage-of-tspmf q) {..k}
proof –
define p' where p' = map-spmf snd p
define q' where q' = map-spmf snd q
have 0:ord-spmf (≥) p' q'
using assms(1) ord-spmf-mono unfolding p'-def q'-def ord-tspmf-def ord-spmf-map-spmf
by fastforce
have cp:coin-usage-of-tspmf p = map-pmf (case-option ∞ enat) p'
unfolding coin-usage-of-tspmf-alt-def p'-def by simp
have cq:coin-usage-of-tspmf q = map-pmf (case-option ∞ enat) q'
unfolding coin-usage-of-tspmf-alt-def q'-def by simp
have 0:rel-pmf (≥) (coin-usage-of-tspmf p) (coin-usage-of-tspmf 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-ge) (simp add:reflp-def)
qed

lemma coin-usage-of-tspmf-mono-rev:
assumes ord-tspmf p q
shows measure (coin-usage-of-tspmf q) {x. x > k} ≤ measure (coin-usage-of-tspmf p) {x. x > k}
(is ?L ≤ ?R)
proof –
have 0:UNIV - {x. x > k} = {..k}
by (auto simp add:set-diff-eq set-eq-iff)
have 1 - ?R ≤ 1 - ?L
using coin-usage-of-tspmf-mono[OF assms]
by (subst (1 2) measure-pmf.prob-comp[symmetric]) (auto simp:0)
thus ?thesis
by simp
qed

lemma expected-coin-usage-of-tspmf:
expected-coin-usage-of-tspmf p = (∑k. ennreal (measure (coin-usage-of-tspmf p) {x. x > enat k})) (is ?L = ?R)
proof –
have ?L = integralN (measure-pmf (coin-usage-of-tspmf p)) ennreal-of-enat
unfolding expected-coin-usage-of-tspmf-def by simp
also have ... = (∑k. emeasure (measure-pmf (coin-usage-of-tspmf p)) {x. enat k < x})
by (subst nn-integral-enat-function) auto

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

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

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

lemma ord-tspmf-trans[trans];
  assumes ord-tspmf \text{p} \text{q} ord-tspmf \text{q} \text{r}
  shows ord-tspmf \text{p} \text{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 \( \forall x. x \leq f x \)
  shows ord-tspmf (map-spmf (apsnd f) \text{p}) \text{p}
  using assms unfolding ord-tspmf-def ord-spmf-map-spmf1
  by (intro ord-spmf-reflI) auto

lemma ord-tspmf-bind-pmf:
  assumes \( \forall x. \) ord-tspmf \((f x) \) \((g x) \)
  shows ord-tspmf (bind-pmf \text{p} f) (bind-pmf \text{p} g)
  using assms unfolding ord-tspmf-def
  by (intro rel-pmf-bindI[where R=\(\Rightarrow\)]) (auto simp:pmf.rel-refl)

lemma ord-tspmf-bind-tspmf:
  assumes \( \forall x. \) ord-tspmf \((f x) \) \((g x) \)
  shows ord-tspmf (bind-tspmf \text{p} f) (bind-tspmf \text{p} g)
  using assms unfolding ord-tspmf-def
  by (intro ord-spmf-bind-reflI) (simp add: case-prod-beta ord-spmf-map-spmf12)

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

lemma use-coins-add:
  use-coins \text{k} (use-coins \text{s} f) = use-coins (\text{k}+\text{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 \( f :: \text{bool} \Rightarrow \text{'b tspmf} \)
  shows (coin-tspmf \(\Rightarrow\ f) = use-coins 1 (coin-spmf \(\Rightarrow\ f) \)
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 \text{k} \text{p}) \text{p}
unfolding use-coins-def by (intro ord-tspmf-map-spmf) auto

64
lemma ord-tspmfa-use-coins-2:
  assumes ord-tspmfa p q
  shows ord-tspmfa (use-coins k p) (use-coins k q)
  using assms unfolding use-coins-def ord-tspmfa-def ord-spmf-map-spmf12 by auto

lemma result-mono:
  assumes ord-tspmfa p q
  shows ord-spmfa (=) (result p) (result q)
  using assms ord-spmfa-mono unfolding result-def ord-tspmfa-def ord-spmf-map-spmf12 by force

lemma result-bind:
  result (bind-tspmfa f g) = result f >>= (\x. result (g x))
  unfolding bind-tspmfa-def result-def map-spmf-conv-bind-spmf by (simp add: case-prod-beta')

lemma result-return:
  result (return-tspmfa x) = return-spmfa x
  unfolding return-tspmfa-def result-def map-spmf-conv-bind-spmf by (simp add: case-prod-beta')

lemma result-coin:
  result (coin-tspmfa) = coin-spmfa
  unfolding coin-tspmfa-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 ⇒ 'a tspmf where
tspmfa-of-ra = spmf-of-ra o track-coin-use

lemma tspmf-of-ra-coin: tspmf-of-ra coin-ra = coin-tspmfa
  unfolding tspmf-of-ra-def comp-def track-coin-use coin-tra-def coin-tspmfa-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-tspmfa x
  unfolding tspmf-of-ra-def comp-def track-coin-use-return return-tra-def return-tspmfa-def
  spmf-of-ra-return by simp

lemma tspmf-of-ra-bind:
  tspmf-of-ra (bind-ra m f) = bind-tspmfa (tspmfa-of-ra m) (\x. tspmf-of-ra (f x))
  unfolding tspmf-of-ra-def comp-def track-coin-use-bind bind-tra-def bind-tspmfa-def
  map-spmf-conv-bind-spmf


lemma tspmf-of-ra-mono:
  assumes ord-ra f g
  shows ord-spmfa (=) (tspmfa-of-ra f) (tspmfa-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-spmfa (tspmfa-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\[where f=\mathrm{spmf-of-ra} \] track-coin-ase-lub assms)
also have ... = \text{lub-spmf} (\text{spmf-of-ra } \cdot \text{track-coin-ase } A)
  by (intro spmf-of-ra-lub-ra 0)
also have ... = \text{?R}
unfolding image-image tspmf-of-ra-def by simp
finally show \text{?thesis} by simp
qed

definition rel-tspmfof-ra :: \text{'a tspmf } \Rightarrow \text{'a random-alg } \Rightarrow \text{bool where}
rel-tspmfof-ra q p \leftrightarrow q = \text{tspmfof-ra } p

lemma admissible-rel-tspmfof-ra:
ccpo.admissible (prod-lub lub-spmf lub-ra) (rel-prod (ord-spmf (\_)) ord-ra) (case-prod rel-tspmfof-ra)
is ccpo.admissible \text{?lub \_ord \_P}
proof (rule ccpo.admissibleI)
fix Y
assume chain: Complete-Partial-Order.chain \text{?ord } Y
  and Y: Y \neq \{}
  and \text{\_R}: \forall (p, q) \in Y. \text{rel-tspmfof-ra } p q
from \text{\_R} have \text{\_R}: \forall p. q. (p, q) \in Y \Rightarrow \text{rel-tspmfof-ra } p q by auto
have \text{chain1}: Complete-Partial-Order.chain (ord-spmf (\_)) (\text{fst } Y)
  and \text{chain2}: Complete-Partial-Order.chain \text{ord-ra } (\text{snd } Y)
using \text{\_chain} by (rule chain-imageI; clarsimp)+
from Y have Y1: \text{fst } Y \neq \{} and Y2: \text{snd } Y \neq \{}
  by auto
have \text{lub-spmf } (\text{fst } Y) = \text{lub-spmf } (\text{tspmfof-ra } (\text{snd } Y))
  unfolding image-image using \text{\_R}
by (intro arg-cong[of - - lub-spmf] image-cong) (auto simp: \text{rel-tspmfof-ra-def})
also have ... = \text{tspmfof-ra } (\text{lub-ra } (\text{snd } Y))
  by (intro tspmf-of-ra-lub[\_symmetric] \text{\_chain2})
finally have \text{rel-tspmfof-ra } (\text{lub-spmf } (\text{fst } Y)) (\text{lub-ra } (\text{snd } Y))
  unfolding \text{rel-tspmfof-ra-def} .
then show \text{?P (\_lub } Y)
  by (simp add: prod-lub-def)
qed

lemma admissible-rel-tspmfof-ra-cont [cont-intro]:
fixes \text{ord}
shows [ mcont \text{lub ord lub-spmf } (ord-spmf (\_)) \text{f}; mcont \text{lub ord lub-ra ord-ra } \text{g } \{}\]
  \Rightarrow \text{ccpo.admissible lub ord } (\lambda x. \text{rel-tspmfof-ra } (\text{f } x ) (\text{g } x))
by (rule admissible-subst[OF \text{admissible-rel-tspmfof-ra}, \text{where } f=\lambda x. (f x \_ g x), \text{simplified}])
  (rule mcont-Pair)

lemma \text{mcont-tspmfof-ra}:
\text{mcont-lub-ra ord-ra lub-spmf } (ord-spmf (\_)) tspmf-of-ra
unfolding \text{mcont-def monotone-def cont-def}
by (auto simp: tspmf-of-ra-mono tspmf-of-ra-lub)

lemmas \text{mcont2mcont-tspmfof-ra } = \text{mcont-tspmfof-ra}\text{[THEN spmf.mcont2mcont]}

context includes lifting-syntax
begin

lemma fixp-rel-tspmfof-ra-parametric[transfer-rule]:
assumes f: \lambda x. \text{mono-spmf } (\lambda f. F \_ f x)
and g: \lambda x. \text{mono-ra } (\lambda f. G \_ f x)
and \text{param: } ((A \Rightarrow \Rightarrow \text{rel-tspmfof-ra}) \Rightarrow \Rightarrow A \Rightarrow \Rightarrow \text{rel-tspmfof-ra}) F G
shows (A \Rightarrow \Rightarrow \text{rel-tspmfof-ra } (\text{spmf.\text{fixp-fun } F}) (\text{random-alg-pf.\text{fixp-fun } G})

66
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))
  ($\lambda x. \ (A ===> rel-tspmf-of-ra) \ (fst x) \ (snd x))$

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) \ (\lambda - . \ lub-spmf \ \{}\) \ (\lambda - . \ 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]: 
  $(=) ===> rel-tspmf-of-ra) \ return-tspmfn \ return-ra$

unfolding rel-fun-def rel-tspmfn-of-ra-def tspmf-of-ra-return
  by simp

lemma bind-ra-transfer [transfer-rule]: 
  $(rel-tspmfn-of-ra === (\lambda \ . \ lub-ra \ \{}\)) \ (\lambda \ . \ lub-ra \ \{})$
  by (auto simp: rel-fun-def rel-tspmfn-of-ra-def 0)

unfolding rel-fun-def rel-tspmfn-of-ra-def tspmf-of-ra-bind
  by simp

lemma coin-ra-transfer [transfer-rule]: 
  $(rel-tspmfn-of-ra coin-spmfn \ coin-ra$

unfolding rel-fun-def rel-tspmfn-of-ra-def tspmf-of-ra-coin
  by simp

end

lemma spmf-of-tspmfn:
  result $(tspmfn-of-ra \ f) = spmf-of-ra \ f$

unfolding tspmf-of-ra-def result-def
  by (simp add: untrack-coin-use spmf-of-ra-map[ symmetric])

lemma coin-usage-of-tspmfn-correct:
  coin-usage-of-tspmfn $(spmfn-of-ra \ p) = \ coin-usage-of-ra \ p \ (is \ ?L = ?R)$

proof –

let $?p = \ Rep-random-alg \ p$

have measure-pmf $(map-spmfn \ snd \ (spmfn-of-ra \ p)) =$
  distr $(distr-rai \ (track-random-bits \ ?p)) \ D \ (map-option \ snd)$

  by simp

also have $... = distr \ B \ D \ (map-option \ snd \ o \ (map-option \ fst \ o \ track-random-bits \ ?p))$

unfolding distr-rai-def
  by (intro distr-distr distr-rai-measurable wf-track-random-bits wf-rep-rand-alg) simp

also have $... = distr \ B \ D \ (\lambda x. \ ?p \ \in \ \{\}) \ (\lambda x. \ consumed-bits \ ?p \ x)\ )$

unfolding track-random-bits-def by (simp add: comp-def map-option-bind case-produc-beta)

also have $... = distr \ B \ D \ (\lambda x. \ consumed-bits \ ?p \ x)\ )$

by (intro arg-cong [where $f = distr \ B \ D$] ext)

also have $... = measure-pmf \ (coin-usage-of-ra-aux \ p)$

unfolding coin-usage-of-ra-aux.rep-eq used-bits-dist-def by simp

finally have measure-pmf $(map-spmfn \ snd \ (spmfn-of-ra \ p)) = measure-pmf \ (coin-usage-of-ra-aux \ p)$

by simp

67
hence $0 = \text{map-spmf} \text{snd} (\text{tspmf-of-ra} p) = \text{coin-usage-of-ra-aux} p$
  using measure-pmf-inject by auto
show ?thesis
unfolding \text{coin-usage-of-tspm-f-def} 0 \text{[symmetric]} \text{coin-usage-of-ra-def} \text{map-pmf-comp}
  by (intro map-pmf-cong) (auto split:option.split)
qed

lemma \text{expected-coin-usage-of-tspm-f-correct}: 
\text{expected-coin-usage-of-tspm-f} (\text{tspmf-of-ra} p) = \text{expected-coin-usage-of-ra} p
unfolding \text{expected-coin-usage-of-tspm-f-def} \text{coin-usage-of-tspm-f-correct}
  by simp
end

8 Dice Roll

theory \text{Dice-Roll}
  imports \text{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].

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

lemma \text{combine-spmf-set-coin-spmf}:
  fixes $f$ :: nat $\Rightarrow$ 'a spmf
  fixes $k$ :: nat
  shows \text{pmf-of-set} \{..<2^k\} > > \text{equal} ($\lambda l \cdot \text{coin-spmf} > > (\lambda b. f (2 * l + \text{of-bool} b))) = 
  \text{pmf-of-set} \{..<2^{(k+1)}\} > > \text{f (is \_L = \_R)}$
proof - 
  let $f = (\lambda (l::nat,b). 2 * l + \text{of-bool} b)$
  let $\text{coin} = \text{pmf-of-set} (\text{UNIV :: bool set})$

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

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

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

  hence $0 = \text{pair-pmf} (\text{pmf-of-set} \{..<(nat)^k\}) \text{coin} = \text{pmf-of-set} \{..<2^k\} \times \text{UNIV}$
  by (intro pmf-eql) simp

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

\^An interesting alternative algorithm, which we did not formalized here, has been introduced by Lambruso [7].
have \(?L = pmf-of-set \{..<2^k\} \gg (\lambda l. \ ?coin \gg (\lambda b. f (2*l + of-bool b)))
\)
unfolding spmf-of-set-def bind-spmf-def spmf-of-pmf-def by (simp add:bind-map-pmf)
also have \(\ldots = pair-pmf \ (pmf-of-set \{..<2^k\}) \ ?coin \gg (\lambda(l, b). f (2*l + of-bool b))
\)
unfolding pair-pmf-def by (simp add:bind-assoc-pmf bind-return-pmf)
also have \(\ldots = map-pmf \ (\lambda(l,b). 2*l + of-bool b) \ (pmf-of-set \ ((..<2^k) \times UNIV)) \gg f
\)
unfolding 0 bind-map-pmf by (simp add:comp-def case-prod-beta)
also have \(\ldots = ?R
\)
unfolding 1 by simp
finally show \(?thesis by simp
qed

lemma count-ints-in-range:
real \(\{x. of-int x \in \{u..v\}\} \geq v-u-1\) (is \(?L \geq ?R
\)
proof (cases \(u \leq v\)
  case True
  have \(0:of-int x \in \{u..v\} \iff x \in \{[u]..[v]\}\) for \(x\) by simp linarith

  have \(v - u - 1 \leq \lfloor v \rfloor - \lfloor u \rfloor + 1\) using True by linarith
  also have \(\ldots = real \ (nat \ (\lfloor v \rfloor - \lfloor u \rfloor + 1))\) using True by (intro of-nat-nat[symmetric]) linarith
  also have \(\ldots = card \ \{\lfloor u \rfloor..\lfloor v \rfloor\}\) by simp
  also have \(\ldots = ?L
\)
  unfolding 0 by (intro arg-cong[where \(f=real\) arg-cong[where \(f=card\)]) auto
finally show \(?thesis by simp
next
  case False
  hence \(v-u-1 \leq 0\) by simp
  thus \(?thesis by simp
qed

partial-function (random-alg) dice-roll-step-ra :: \(\text{real} \Rightarrow \text{real} \Rightarrow \text{int} \ \text{random-alg}\)
where 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 dice-roll-ra n = map-ra nat (dice-roll-step-ra 0 (of-nat n))

partial-function (spmfn) drs-tspmfn :: \(\text{real} \Rightarrow \text{real} \Rightarrow \text{int} \ \text{spmfn}\)
where drs-tspmfn l h = ( if \([l] = [l+h]-1\) then
  return-tspmfn \([l]\)
  else
  do \(\{b \leftarrow \text{coin-tspmfn}; \text{drs-tspmfn} \ (l + (h/2) * \text{of-bool b}) \ (h/2)\}\)
)
definition dice-roll-tspmfn n = drs-tspmfn 0 (of-nat n) \gg (\lambda x. return-tspmfn \ (nat x))

lemma drs-tspmfn: drs-tspmfn l u = tspmf-of-ra \ (dice-roll-step-ra l u)
proof
  include lifting-syntax
  have \((=) \Longrightarrow \ (\gg) \Longrightarrow \ \text{rel-tspmfn-of-ra}) drs-tspmfn dice-roll-step-ra
    unfolding drs-tspmfn-def dice-roll-step-ra-def
    apply (rule rel-funD[OF carry-transfer])
    apply (rule fixp-rel-tspmfn-of-ra-parametric[OF drs-tspmfn.mono dice-roll-step-ra.mono])
  by transfer-prover

69
thus \( \text{thesis} \)

unfolding rel-fun-def rel-tspmfd-of-ra-def by auto

qed

lemma dice-roll-ra-tspmfd: tspmfd-of-ra (dice-roll-ra n) = dice-roll-tspmfd n

drs-tspmfd by simp

lemma dice-roll-step-tspmfd-lb:
assumes \( h > 0 \)
shows \( \text{coin-tspmfd} \gg (\lambda b. \text{drs-tspmfd} (l + (h/2) \cdot \text{of-bool} b) (h/2)) \leq_R \text{drs-tspmfd} l h \)

proof (cases \( |l| = \lceil l + h \rceil - 1 \))
case True
  hence \( 2 \cdot \text{drs-tspmfd} l h = \text{return-tspmfd} \lfloor l \rfloor \)
  by (subst drs-tspmfd. simps) simp

have 0: \( \lfloor l + h / 2 \cdot \text{of-bool} b \rfloor = \lfloor l \rfloor \) for \( b \)

proof –
  have \( \lfloor l + h / 2 \cdot \text{of-bool} b \rfloor \leq \lfloor l + h / 2 \rfloor \)
    using \( h \) by (intro floor-mono add-mono) auto
  also have .. \( = \lfloor l + h \rfloor - 1 \)
    using \( h \) by simp
  finally have \( \lfloor l + h / 2 \cdot \text{of-bool} b \rfloor \leq \lfloor l \rfloor \) by simp
  moreover have \( |l| \leq \lfloor l + h / 2 \cdot \text{of-bool} b \rfloor \)
    using \( h \) by simp
  ultimately show \( \text{thesis} \) by simp

qed

have 1: \( \lfloor l + h / 2 \cdot \text{of-bool} b + h / 2 \rfloor - 1 = \lfloor l \rfloor \) for \( b \)

proof –
  have \( \lfloor l + h / 2 \cdot \text{of-bool} b + h / 2 \rfloor - 1 \leq \lfloor l + h \rfloor - 1 \)
    using \( h \) by simp
  finally have \( \lfloor l + h / 2 \cdot \text{of-bool} b + h / 2 \rfloor - 1 \leq \lfloor l \rfloor \) by simp
  moreover have \( |l| \leq \lfloor l + h / 2 \cdot \text{of-bool} b + h / 2 \rfloor - 1 \)
    using \( h \) by simp
  ultimately show \( \text{thesis} \) by simp

qed

have 3: \( \text{drs-tspmfd} (l + (h/2) \cdot \text{of-bool} b) (h/2) = \text{return-tspmfd} \lfloor l \rfloor \) for \( b \)

using 0 1 by (subst drs-tspmfd.simps) simp

show \( \text{thesis} \)
  unfolding 2 3 bind-tspmfd-def coin-tspmfd-def pair-spmfd-alt-def
  by (simp add:bind-spmfd-const ord-tspmfd-map-spmfd)

next
case False
  thus \( \text{thesis} \)
    by (subst drs-tspmfd.simps) (auto intro:ord-tspmfd-refl)

qed

abbreviation coins \( k \equiv \text{pmf-of-set} \{..<(2::nat)^k\} \)

lemma dice-roll-step-tspmfd-expand:
assumes \( h > 0 \)
shows \( \text{coins} \gg (\lambda l. \text{use-coins} k (\text{drs-tspmfd} \ (\text{real} \ l \cdot h) h)) \leq_R \text{drs-tspmfd} 0 (h \cdot 2^k) \)
using \( h \)
proof (induction k arbitrary:h)

  case 0
  have {..<Suc 0} = {0} by auto
  then show ?case
    by (auto intro:ord-tspmff-use-coins simp:pmf-of-set-singleton bind-return-pmf)
next

  case (Suc k)
  have (coins (k+1) ≨ (λl. use-coins (k+1) (drs-tspmff (real l*h))))) =
    coins k ≨ (λl. use-coins (k+1) (drs-tspmff (real (2*l + of-bool b) * h)))
    by (intro combine-spmff-set-coin-spmff[ symmetric])
  also have ... = coins k ≨ (λl. use-coins (k+1) (coin-spmff ≨
    (λb. drs-tspmff (real l*(2+h) + h * of-bool b)))))
    unfolding use-coins-def map-spmff-cone-bind-spmff by (simp add: algebra-simps)
  also have ... = coins k ≨ (λl. drs-tspmff (real l*(2+h) + h * of-bool b))
    unfolding coin-tspmff-split use-coins-add by simp
  also have ... = coins k ≨ (λl. use-coins k (coin-spmff ≨
    (λb. drs-tspmff (real l*(2*h) + (2*h/2) * of-bool b) ((2*h)/(2))))
    using Suc(2) by simp
  also have ... ≤R coins k ≨ (λl. use-coins k (drs-tspmff (real l*(2 + h)) (2*h)))
    using Suc(2) by (intro ord-tspmff-bind-pmf ord-tspmff-use-coins-2 dice-roll-step-tspmff-lb) simp
  also have ... ≤R drs-tspmff 0 ((2*h) * 2^k)
    using Suc(2) by (intro Suc(1)) auto
  also have ... = drs-tspmff 0 (h*2^(k+1))
    unfolding power-add by (simp add: algebra-simps)
  finally show ?thesis by simp
qed

lemma dice-roll-step-tspmff-approx:
  fixes k :: nat
  assumes h > (0::real)
  defines f ≡ (λl. if ⌈l*h⌉ = ⌈(l+1)*h⌉ - 1 then Some (⌈l*h⌉) else None)
  shows map-pmf f (coins k) ≤R drs-tspmff 0 (h*2^k) (is ⌈L ≤R ⌈R)
proof
  have ⌈L = coins k ≨
    (λl. use-coins k (if ⌈real l*h⌉ = ⌈(l+1)*h⌉ - 1 then return-tspmff ⌈l*h⌉ else return-pmf None))
    unfolding f-def return-tspmff-def use-coins-def map-pmf-def
  by (simp add: if-distribR if-distrib bind-return-pmf algebra-simps cong; if-cong)
  also have ... ≤R coins k ≨ (λl. use-coins k (drs-tspmff (real l*h)))
    by (subst drs-tspmff.simps, intro ord-tspmff-bind-pmf ord-tspmff-use-coins-2)
    (simp add: ord-tspmff-min ord-tspmff-refl algebra-simps)
  also have ... ≤R drs-tspmff 0 (h*2^k)
    by (intro dice-roll-step-tspmff-expand assms)
  finally show ?thesis by simp
qed

lemma dice-roll-step-tspmff-approx:
  fixes k :: nat
  assumes h > (0::real)
  defines f ≡ (λl. if ⌈l*h⌉ = ⌈(l+1)*h⌉ - 1 then Some (⌈l*h⌉) else None)
  shows ord-spmff (=) (map-pmf f (coins k)) (result (drs-tspmff 0 (h*2^k)))
  (is ord-spmff - ⌈L ⌈R)
proof
  have ⌈L = result (map-pmf (λl. if ⌈l*h⌉ = ⌈(l+1)*h⌉ - 1 then Some (⌈l*h⌉) 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-tspmff-approx] by simp
qed

lemma spmf-dice-roll-step-lb:
  assumes j < n
  shows spmf (result (drs-tspmff 0 (of-nat n))) (of-nat j) ≥ 1/n (is ?L ≥ ?R)
proof (rule ccontr)
  assume ¬(spmff (result (drs-tspmff 0 (of-nat n))) (of-nat j) ≥ 1/n)
  hence a:?L < 1/n by simp
  define k :: nat where k = nat [2−log2 (1/n−?L)]
  define h where h = real n/2^k
  define f where f l = (if [l*h]=[⌈(l+1)*h⌉]−1 then Some [l*h] else None) for l :: nat

  have h-gt-0: h > 0 using assms unfolding h-def by auto
  have n-gt-0: real n > 0 using assms by simp

  have 0: x < 2^k if real x ≤ (real j+1)*2^k/n−1 for x
  proof
    have real x ≤ (real j+1)*2^k/n−1 using that by simp
    also have ... ≤ real n * 2^k/n − 1 using assms by (intro diff-mono divide-right-mono mult-right-mono) auto
    also have ... ≤ 2^k−1 by simp
    finally have real x ≤ 2^k−1 by simp
    thus ?thesis using nat-less-real-le by auto
  qed

  have 2: int {x. P (real x)} = {x. P (real-of-int x)} if \(\forall x. P(x) \Rightarrow x \geq 0\) for P
  proof
    have bij-betw int {x. P (real x)} {x. P (real-of-int x)}
      using that by (intro bij-betwI[where g=nat]) force+
    thus ?thesis using bij-betw-imp-surj-on by auto
  qed

  have 1: real j*2^k/n ≥ 0 by auto

  have 3:⌈real l*h⌉≤⌈real (l+1)*h⌉−1 for l
    using h-gt-0 by (intro floor-ceil-minus-one) force

  have 2 = (1/n − ?L)*2 powr (1−log2 (1/n−?L))
    using a n-gt-0 unfolding powr-diff by (subst powr-log-cancel) (auto simp:divide-simps)
  also have ... < (1/n − ?L)*2 powr [2−log2 (1/n−?L)]
    using a by (intro mult-strict-left-mono powr-less-mono) linarith+
  also have ... ≤ (1/n − ?L)*2 powr real k
    using a unfolding k-def by (intro mult-left-mono powr-mono) auto
  also have ... = (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^k by (simp add:field-simps)
  also have ... = (((real j+1)*2^k/n−1)−(real j*2^k/n−1))/2^k
    using n-gt-0 by (simp add:field-simps)
  also have ... ≤ real (card {x. of-int x ∈ {real j*2^k/n..(real j+1)*2^k/n−1}})2^k
    by (intro divide-right-mono count-ints-in-range) auto
  also have ... = real (card (int {x. real x ∈ {real j+2^k/n..(real j+1)*2^k/n−1}}))/2^k
    using order-trans[OF 1] by (subst 2) auto
  also have ... = real (card {x. real x ∈ {real j*2^k/n..(real j+1)*2^k/n−1}})/2^k
    by (subst card-image) auto
  also have ... = real (card {x. real x ∈ {real j+2^k/n..(real j+1)*2^k/n−1}})/2^k
    using 0 by (intro arg-cong[where f=λx. real (card x)/2^k] auto)
  also have ... = real (card {l. l<2^k ∧ real l ≤ real l * h ∧ (1 + real l)*h≤real j+1}) / 2^k
using assms unfolding h-def
by (intro arg-cong[where f=λx. real (card x)/2^k] Collect-conq) (auto simp:field-simps)
also have ... = measure (coins k) {l. real j ≤ real l*h ∧ real ((l+1)*h) ≤ real j + 1} 
  by (subgoal assms unfolding le_less[where a=1] refl)
also have ... = measure (coins k) {l. int j ≤ [real l*h] ∧ [real ((l+1)*h)] - 1 ≤ int j} 
  by (intro arg-cong2[where f=measure] refl Collect-conq refl)
also have ... = measure (coins k) {l. int j = [real l*h] ∧ int j = [real ((l+1)*h)] - 1} 
  using 3 order.trans order-antisym
by (intro arg-cong2[where f=measure] refl Collect-conq iffI blast simp)
also have ... = spmf (map-pmf f (coins k)) j 
  unfolding pmf-map f-def vimage-def
by (intro arg-cong[where f=measure] refl Collect-conq auto)
also have ... ≤ spmf (result (drs-tspmfn 0 (h*2^k))) j 
  unfolding f-def by (intro ord-spmf-le-approx Dice-roll-step-spmf-approx h-gt-0)
also have ... = ?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 result (drs-tspmfn 0 (of-nat n)) = spmf-of-set {0..<n}
proof -
  have weight-spmf (spmf-of-set {0..<int n}) ≥ weight-spmf (result (drs-tspmfn 0 (of-nat n))) 
    using assms unfolding weight-spmf-of-set
    by (simp add:lessThan-iff weight-spmf-le-1)
  moreover have spmf (spmf-of-set {0..<int n}) x ≤ spmf (result (drs-tspmfn 0 (of-nat n))) x 
    for x
  proof (cases x < n ∧ x ≥ 0)
    case True
    hence spmf (spmf-of-set {0..<int n}) x = 1/n 
      unfolding spmf-of-set by auto
    also have ... ≤ spmf (result (drs-tspmfn 0 (of-nat n))) (of-nat (nat x))
      using True by (intro spmf-dice-roll-step-lb auto)
    also have ... = spmf (result (drs-tspmfn 0 (of-nat n))) x 
      using True by (subgoal assms unfolding le_less[where a=1] refl)
    finally show ?thesis by simp
  next
    case False
    hence spmf (spmf-of-set {0..<int n}) x = 0 
      unfolding spmf-of-set by auto
    then show ?thesis by simp
  qed

  hence ord-spmf (=) (spmf-of-set {0..<int n}) (result (drs-tspmfn 0 (of-nat n)))
    by (intro ord-spmf-increase refl refl)
  ultimately show ?thesis
    by (intro eq-iff-or-spmf[where g=def] sym)

qed

theorem dice-roll-correct:
  assumes n > 0
  shows result (dice-roll-tspmfn n) = spmf-of-set ..<n {is ?L = ?R} 
  spmf-of-ra (dice-roll-ra n) = spmf-of-set ..<n
proof -
  have bij:bij-betweennat {0..<int n} ..<n 
    by (intro bij-between[where g=int] auto)
have \(?L = \text{map-spmf} \text{ nat} \ (\text{spmff-of-set} \ \{0..<\text{int} \ n\})\)
  unfolding dice-roll-tspmf-def dice-roll-correct-aux[of \text{assms} \ result-bind \ result-return
  map-spmf\-conv\-bind\-spmff \ by \ simp
also have \(... = \text{spmff-of-set} \ (\text{nat} \ \{0..<\text{int} \ n\})\)
  by (intro map-spmf\-of-set\-inj\-on inj\-onI) auto
also have \(... = \ ?R\)
  using bij\-betw\-imp\-surj\-on[of \text{bij} \ by \ (intro arg\-cong[where \text{f=}spmff-of-set]) \ auto
finally show \(?L = \ ?R \ by \ simp\)
thus \text{spmff-of-ra} \ (\text{dice-roll-ra} \ n) = \ ?R
  using spmff-of-tspmff dice-roll-ra-tspmff by metis
qed

lemma dice-roll-consumption-bound:
assumes \(n > 0\)
shows \(\text{measure} \ (\text{coin-usage-of-tspmff} \ (\text{dice-roll-tspmff} \ n)) \ \{x, x > \text{enat} \ k\} \leq \text{real} \ n/2^k\)
(is \(?L \leq \ ?R\))
proof
  define \(h\) where \(h = \text{real} \ n/2^k\)
  define \(f\) where \(f \ l = (\text{if} \ (\text{int\-\ast} h) = ((l+1)*h) - 1 \ \text{then Some} \ ((\text{int\-\ast} h), k) \ \text{else None}) \ \text{for} \ l :: \text{nat}\)
    have \(\text{h-gt-0}: \ h > 0\)
      using \text{assms} unfolding \text{h-def}
      by (intro divide\-pos\-pos) auto
    have \(0::\text{real} \ n = h \ast 2^k\)
      unfolding \text{h-def} by simp
    have \(1::\text{real} \ l*h < (\text{real} \ l+1)*h\) if \(\text{real} \ l*h \neq (\text{real} \ l+1)*h - 1\) for \(l\)
      proof
        have \(\text{real} \ l*h \leq (\text{real} \ l+1)*h - 1\)
          using \(\text{h-gt-0}\) by (intro floor-le-ceil-minus-one) force
          hence \(\text{real} \ l*h < (\text{real} \ l+1)*h - 1\)
            using that by simp
            also have \(\leq (\text{real} \ l+1)*h\)
              by linarith
            finally show \(\text{thesis} \ by \ simp\)
qed

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

proof -
  define \(k_0\) where \(k_0 = \lceil \log 2 \, n \rceil\)
  define \(\delta\) where \(\delta = \log 2 \, n - \lfloor \log 2 \, n \rfloor\)

have 0: \(\text{ennreal} (\text{measure} (\text{coin-usage-of-tspmf} (\text{dice-roll-tspmf} \, n))) \{ x. \, x > \text{enat} \, k \} \leq \text{ennreal} (\text{min} (\text{real} \, n/2^k) \, 1)\) (is \(\?L \leq \?R\)) for \(k\)
  by (intro \text{ifD2}[OF \text{ennreal-le-if}] \text{min.boundedI} \text{dice-roll-consumption-bound}[OF \text{assms}]) auto

have 1[simp]: (2::ennreal) \(\sim k < \text{Orderings.top}\) for \(k::\text{nat}\)
  using \text{ennreal-numeral-less-top} \text{power-less-top-ennreal} by blast

have \(\sum k. \text{ennreal} ((1/2) \, k)) = \text{ennreal} (\sum k. ((1/2) \, k))\)
  by (intro \text{suminf-ennreal2}) auto
also have ... = \text{ennreal} 2
  by (subst \text{suminf-geometric}) simp-all
finally have 2:(\(\sum k. \text{ennreal} ((1/2) \, k)) = \text{ennreal} 2\)
  by simp

have \(\text{real} \, n \geq 1\)
  using \text{assms} by simp
hence 3: \(\log 2 \, (\text{real} \, n) \geq 0\)
  by simp

have \(\text{real-of-int} \lfloor \log 2 \, (\text{real} \, n) \rfloor \leq 1 + \log 2 \, (\text{real} \, n)\)
  by \text{linarith}
hence 4: \(\delta + 1 \in \{0..1\}\)
  unfolding \(\delta\)-def by (auto \text{simp}; \text{algebra-simps})

have \(?L = (\sum k. \text{ennreal} (\text{measure} (\text{coin-usage-of-tspmf} (\text{dice-roll-tspmf} \, n))) \{ x. \, x > \text{enat} \, k \}))\)
  unfolding \text{expected-coin-usage-of-tspmf} by simp
also have ... \(\leq (\sum k. \text{ennreal} (\text{min} (\text{real} \, n/2^k) \, 1))\)
  by (intro \text{suminf-le summableI} 0)
also have ... = (\(\sum k. \text{ennreal} (\text{min} (\text{real} \, n/2^{(k+k)}))\, 1))+(\(\sum k < k_0. \, \text{ennreal}(\text{min} (\text{real} \, n/2^k) \, 1))\)
  by (intro \text{suminf-offset summableI})
also have ... \(\leq (\sum k. \text{ennreal} (\text{real} \, n/2^{(k+k)})) + (\sum k < k_0. \, 1)\)
  by (intro \text{add-mono suminf-le summableI} \text{sum-mono ifD2}[OF \text{ennreal-le-if}] auto
also have ... = (\(\sum k. \text{ennreal} (\text{real} \, n/2^k) \, * \text{ennreal} ((1/2) \, k)) + \text{of-nat} \, k_0\)
  by (intro \text{suminf-cong arg-cong2}[where \(f=+\)]))
  (simp-all \text{add}; \text{ennreal-mult}[\text{symmetric}] \text{power-add divide-simps})
also have ... = \text{ennreal} (\text{real} \, n/2^k_0) \, * \, (\sum k. \text{ennreal} ((1/2) \, k)) + \text{ennreal} (\text{real} \, k_0)\)
unfolding ennreal-of-nat-eq-real-of-nat by simp
also have ... = ennreal (real n / 2^k0 * 2 + real k0)
unfolding 2 by (subst ennreal-mul[symmetric]) simp-all
also have ... = ennreal (real n / 2 powr k0 * 2 + real k0)
by (subt powr-realpow) auto
also have ... = ennreal (2 powr (log 2 n) * 2 + real k0)
using 3 unfolding k0-def by (subst of-nat-nat) auto
also have ... = ennreal (2 powr (log 2 n - δ) * 2 + real k0)
unfolding δ-def by simp
also have ... = ennreal (2 powr (log 2 n - δ) * 2 + 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-tspmfd (tspmfd-of-ra (dice-roll-ra n))
unfolding expected-coin-usage-of-tspmfd-correct[symmetric] by simp
also have ... = expected-coin-usage-of-tspmfd (dice-roll-tspmfd n)
unfolding dice-roll-ra-tspmfd 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
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-mul :: 64 word
  where pcg-mul = 6364136223846793005

definition pcg-increment :: 64 word
  where pcg-increment = 1442695040888963407

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

76
where \( \text{pcg-rotr} \ x \ r = \text{Bit-Operations.or} \ (\text{drop-bit} \ r \ x) \ (\text{push-bit} \ (32-r) \ x) \)

fun \( \text{pcg-step} :: \) \(64\) word \( \Rightarrow \) \(64\) word  
where \( \text{pcg-step state} = \text{state} \ast \text{pcg-mult} + \text{pcg-increment} \)

Based on [9, Section 6.3.1]:

fun \( \text{pcg-get} :: \) \(64\) word \( \Rightarrow \) \(32\) word  
where \( \text{pcg-get state} = (\text{let} \ \text{count} = \text{unsigned} \ (\text{drop-bit} \ 59 \ \text{state}); x = \text{xor} \ (\text{drop-bit} \ 18 \ \text{state}) \ \text{state} \in \text{pcg-rotr} \ (\text{ucast} \ (\text{drop-bit} \ 27 \ x)) \ \text{count}) \)

fun \( \text{pcg-init} :: \) \(64\) word \( \Rightarrow \) \(64\) word  
where \( \text{pcg-init seed} = \text{pcg-step} \ (\text{seed} + \text{pcg-increment}) \)

definition \( \text{to-bits} :: \) \(32\) word \( \Rightarrow \) bool list  
where \( \text{to-bits} x = \text{map} \ (\lambda k. \text{bit} x k) [0..<32] \)

definition \( \text{random-coins} \)  
where \( \text{random-coins seed} = \text{build-coin-gen} \ (\text{to-bits} \circ \text{pcg-get}) \ \text{pcg-step} \ (\text{pcg-init seed}) \)

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 \( \text{binary-dice-roll} :: \) nat \( \Rightarrow \) nat random-alg  
where \( \text{binary-dice-roll} 0 = \text{return-ra} \ 0 \ | \text{binary-dice-roll} \ (\text{Suc} \ n) = \begin{array}{l} \text{do} \{ h \leftarrow \text{binary-dice-roll} \ n; c \leftarrow \text{coin-ra}; \text{return-ra} \ (\text{of-bool} \ c + 2 \ast h) \} \end{array} \)

Because the algorithm terminates unconditionally it is easy to verify that \( \text{binary-dice-roll} \) terminates almost surely:

lemma \( \text{binary-dice-roll-terminates}: \text{terminates-almost-surely} \ (\text{binary-dice-roll} \ n) \)  
by (\text{induction} \ n) (\text{auto intro:terminates-almost-surely-intros})

The corresponding PMF can be written as:

fun \( \text{binary-dice-roll-pmf} :: \) nat \( \Rightarrow \) nat pmf  
where

77
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.monono binary-geometric.monono])
    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.fisp-induct)
case 1
  thus ?case by simp
next
case 2
  thus ?case by simp
next
case (3 p)
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 ⇒ nat random-alg where
geometric-ra p = do {
b ← bernoulli-ra p;
  if b then return-ra 0 else map-ra ((+ )1) (geometric-ra p)
}
declare geometric-ra.simps[code]

lemma geometric-ra-correct: spmf-of-ra (geometric-ra x) = geometric-spmf x
proof
  include lifting-syntax
  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 ⇒ 'a random-alg ⇒ 'a list random-alg
where
  replicate-ra 0 f = return-ra [] |
  replicate-ra (Suc n) f = do { xh ← f; xt ← replicate-ra n f; return-ra (xh#xt) }

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

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

lemma replicate-spmf-of-pmf: replicate-spmf n (spmf-of-pmf f) = spmf-of-pmf (replicate-pmf n f)
by (induction n) (simp-all add: spmf-of-pmf-bind)

Binomial distribution

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

lemma
  assumes p ∈ {0..1}
  shows spmf-of-ra (binomial-ra n p) = spmf-of-pmf (binomial-pmf n p)
proof
  have spmf-of-ra (replicate-ra n (bernoulli-ra p)) = spmf-of-pmf (replicate-pmf n (bernoulli-pmf p))
    unfolding replicate-ra-correct bernoulli-ra-correct[symmetric] bernoulli-eq-bernoulli-pmf
  by (simp add: replicate-spmf-of-pmf)
thus ?thesis
  unfolding binomial-pmf-altdef[OF assms] binomial-ra-def
by (simp flip: map-spmf-of-pmf add: spmf-of-ra-map)
qed

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

value run-ra (binomial-ra 10 0.5) (random-coins 42)
value run-ra (replicate-ra 20 (bernoulli-ra 0.3)) (random-coins 42)

end

References


