Abstract

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

Contents

1 Introduction 1
2 τ-Additivity 2
3 Coin Flip Space 3
4 Randomized Algorithms (Internal Representation) 11
5 Randomized Algorithms 18
  5.1 Almost surely terminating randomized algorithms . . . . . . . . . . . . . . . . . . 22
6 Tracking Randomized Algorithms 23
7 Tracking SPMFs 26
8 Dice Roll 30
9 A Pseudo-random Number Generator 32
10 Basic Randomized Algorithms 33

1 Introduction

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

In this work, we introduce a representation of randomized algorithms as a parser monad over an external arbitrary source of random coin flips, modelled using a lazy infinite stream of booleans. Using for example a PRG or some other mechanism, like a hardware RNG to supply the coin flips, the generated code for the monad can be executed.
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)
- `tspmof-ra`: Morphism between randomized algorithms and the joint-distribution of their result and coin-flip usage. (Section 7)

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

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

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

## 2 $\tau$-Additivity

theory Tau-Additivity
imports HOL-Analysis.Regularity
begin

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

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

this is shown in \textit{SUP-emeasure-incseq}.

It is possible to generalize that to arbitrary chains\(^1\) of open sets for some measures without the restriction of countability, such measures are called \(\tau\)-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 \textit{HOL—Analysis.Regularity}.

\textbf{definition} \textit{op-stable} \(op\) \(F = (\forall x \ y. \ x \in F \land y \in F \longrightarrow \text{op} \ x \ y \in F)\)

\textbf{lemma} \textit{op-stableD}:  
assumes \textit{op-stable} \(op\) \(F\)  
assumes \(x \in F\) \(y \in F\)  
shows \(\text{op} \ x \ y \in F\)  
⟨\textit{proof}⟩

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

\textbf{lemma} \textit{chain-imp-union-stable}:  
assumes \textit{Complete-Partial-Order.chain} \((\subseteq)\) \(F\)  
shows \(\text{op-stable} \ (\bigcup) \ F\)  
⟨\textit{proof}⟩

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

end

\section{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.

\textbf{theory} \textit{Coin-Space}

\textbf{imports}  
\textit{HOL—Probability.Probability}  
\textit{HOL—Library.Code-Lazy}

\begin{verbatim}

end

\footnote{More generally families closed under pairwise unions.}

\end{verbatim}
lemma stream-eq-iff:
  assumes \( \forall i. x !! i = y !! i \)
  shows \( x = y \)
⟨proof⟩

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 \( f \in M \rightarrow M' \)
  assumes \( g \in N \rightarrow N' \)
  shows \( \text{map-prod} f g \in M \otimes N \rightarrow M' \otimes N' \)
⟨proof⟩

lemma measurable-sigma-sets-with-exception:
  fixes \( f :: 'a \Rightarrow 'b :: \text{countable} \)
  assumes \( \forall x. x \neq d \Rightarrow f - \{ x \} \cap \text{space } M \in \text{sets } M \)
  shows \( f \in M \rightarrow \text{count-space } UNIV \)
⟨proof⟩

lemma restr-empty-eq: \( \text{restrict-space } M \{ \} = \text{restrict-space } N \{ \} \)
⟨proof⟩

lemma (in prob-space) distr-stream-space-snth [simp]:
  assumes \( \text{sets } M = \text{sets } N \)
  shows \( \text{distr} (\text{stream-space } M) N (\lambda xs. \text{snth} xs n) = M \)
⟨proof⟩

lemma (in prob-space) distr-stream-space-shd [simp]:
  assumes \( \text{sets } M = \text{sets } N \)
  shows \( \text{distr} (\text{stream-space } M) N \text{ shd} = M \)
⟨proof⟩

lemma shift-measurable:
  assumes \( \text{set } x \subseteq \text{space } M \)
  shows \( (\lambda bs. x @- bs) \in \text{stream-space } M \rightarrow \text{stream-space } M \)
⟨proof⟩

lemma (in sigma-finite-measure) restrict-space-pair-lift:
  assumes \( A' \in \text{sets } A \)
  shows \( \text{restrict-space } A A' \otimes M = \text{restrict-space } (A \otimes M) (A' \times \text{space } M) \) (is \( ?L = ?R \))
⟨proof⟩

lemma to-stream-comb-seq-eq:
  \( \text{to-stream} (\text{comb-seq } n x y) = \text{stake } n (\text{to-stream } x) @- \text{to-stream } y \)
lemma to-stream-snth: to-stream (!! x) = x
⟨proof⟩

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

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

The type for the coin flip space is isomorphic to bool stream. Nevertheless, we introduce
it as a separate type to be able to introduce a topology and mark it as a lazy type for
code-generation:
codatatype coin-stream = Coin (chd:Boolean) (ctl:coin-stream)

code-lazy-type coin-stream

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

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

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

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

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

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

definition cshift where cshift x y = to-coins (x @- from-coins y)
definition cnth where cnth x n = from-coins x !! n
definition etake where etake 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 ←→ etake (length x) y = x
definition cconst where cconst x = to-coins (sconst x)

context
  includes lifting-syntax
begin

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

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

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

⟨proof⟩

lemma ctake-transfer [transfer-rule]: ((=) === rel-coins ===) stake ctake

⟨proof⟩

lemma cdrop-transfer [transfer-rule]: ((=) === rel-coins === rel-coins) sdrop cdrop

⟨proof⟩

lemma chd-transfer [transfer-rule]: (rel-coins === (=)) shd chd

⟨proof⟩

lemma ctl-transfer [transfer-rule]: (rel-coins === rel-coins) stl ctl

⟨proof⟩

lemma cconst-transfer [transfer-rule]: ((=) === rel-coins)

⟨proof⟩

end

lemma coins-eq-iff:
  assumes \( \forall i. \text{cnth } x i = \text{cnth } y i \)
  shows \( x = y \)

⟨proof⟩

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

⟨proof⟩

lemma ctake-nth[simp]: \( m < n \Longrightarrow \text{ctake } n s ! m = \text{cnth } s m \)

⟨proof⟩

lemma ctake-cdrop: cshift (ctake n s) (cdrop n s) = s

⟨proof⟩

lemma cshift-append[simp]: cshift (p @ q) s = cshift p (cshift q s)

⟨proof⟩

lemma cshift-empty[simp]: cshift [] xs = xs

⟨proof⟩

lemma ctake-null[simp]: ctake 0 xs = []

⟨proof⟩

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

⟨proof⟩

lemma cdrop-null[simp]: cdrop 0 s = s

⟨proof⟩

lemma cdrop-Suc[simp]: cdrop (Suc n) s = cdrop n (ctl s)

⟨proof⟩

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

⟨proof⟩

lemma ctl-shift[simp]: ctl (cshift xs s) = (if xs = [] then ctl s else cshift (tl xs) s)
lemma shd-sconst[simp]: chd (cconst x) = x
  ⟨proof⟩

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

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

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

lemma cprefix-iff: cprefix x y ←→ (∀ i < length x. cnth y i = x ! i) (is ?L ←→ ?R)
  ⟨proof⟩

A non-empty shift is not idempotent:

lemma empty-if-shift-idem:
  assumes ⋀ cs. cshift h cs = cs
  shows h = []
  ⟨proof⟩

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⟩

lemma same-prefix-not-parallel:
  assumes cprefix x bs cprefix y bs
  shows ¬ (x || y)
  ⟨proof⟩

lemma ctake-shift:
  ctake m (cshift xs ys) = (if m ≤ length xs then take m xs else xs @ ctake (m − length xs) ys)
  ⟨proof⟩

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
  ⟨proof⟩

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⟩

lemma cdrop-shift-small [simp];
  m ≤ length xs ⇒ cdrop m (cshift xs ys) = cshift (drop m xs) ys
and cdrop-shift-big [simp];
  m ≥ length xs ⇒ cdrop m (cshift xs ys) = cdrop (m − length xs) ys
  ⟨proof⟩

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 s))

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

**⟨proof⟩**

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

**where**

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

(λ(r,s'). (if tl r = [] then (m s', f s') else (tl r, s')))) (m s, f s)

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

**fixes** f :: 'a ⇒ 'b stream

**assumes** \( \forall x. (\exists \; n \; y. \; n \neq [] \wedge f x = n@ - f y \wedge g x = n@ - g y) \)

**shows** f x = g x

**⟨proof⟩**

**lemma** build-coin-gen:

**assumes** \( \forall x. m x \neq [] \)

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

**⟨proof⟩**

Measure space for coin streams:

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

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

**bundle** coin-space-notation

**begin**

**notation** coin-space (B)

**end**

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

**begin**

**no-notation** coin-space (B)

**end**

**unbundle** coin-space-notation

**lemma** space-coin-space: space B = UNIV

**⟨proof⟩**

**lemma** B-t-eq-distr: B = distr (stream-space (pmf-of-set UNIV)) B to-coins

**⟨proof⟩**

**lemma** from-coins-measurable: from-coins ∈ B → M (stream-space (pmf-of-set UNIV))

**⟨proof⟩**

**lemma** to-coins-measurable: to-coins ∈ (stream-space (pmf-of-set UNIV)) → M B

**⟨proof⟩**

**lemma** chd-measurable: chd ∈ B → M D

**⟨proof⟩**

**lemma** cnth-measurable: (λxs. cnth xs i) ∈ B → M D

**⟨proof⟩**

**lemma** B-eq-distr:

stream-space (pmf-of-set UNIV) = distr B (stream-space (pmf-of-set UNIV)) from-coins

(is ?L = ?R)
lemma B-t-finite: emeasure B (space B) = 1

interpretation coin-space: prob-space coin-space

lemma distr-shd: distr B D chd = pmf-of-set UNIV (is ?L = ?R)

lemma cshift-measurable: cshift x ∈ B →_M B

lemma cdrop-measurable: cdrop x ∈ B →_M B

lemma ctake-measurable: ctake k ∈ B →_M D

lemma branch-coin-space:
  (λ(x, y). cshift (ctake n x) y) ∈ B ⊗_M B →_M B
distr (B ⊗_M B) B (λ(x,y). cshift (ctake n x) y) = B (is ?L = ?R)

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

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

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

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

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

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

instantiation coin-stream :: topological-space
begin
  definition open-coin-stream :: coin-stream set ⇒ bool
  where open-coin-stream U = open (from-coins-t `· U)
  instance ⟨proof⟩
end

definition coin-stream-basis
  where coin-stream-basis = (λx. Collect (cprefix x)) `· UNIV

lemma image-collect-eq: f `· {x. A (f x)} = {x. A x} ∩ range f
  ⟨proof⟩
lemma coin-stream-basis: topological-basis coin-stream-basis
⟨proof⟩

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

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

instantiation coin-stream :: uniformity-dist
begin
definition dist-coin-stream :: coin-stream ⇒ coin-stream ⇒ real
where dist-coin-stream x y = dist (from-coins-t x) (from-coins-t y)
definition uniformity-coin-stream :: (coin-stream × coin-stream) filter
where uniformity-coin-stream = (INF e∈{0<..}. principal {(x, y). dist x y < e})
instance ⟨proof⟩
end

lemma in-from-coins-iff: x ∈ from-coins-t ' U ←→ (to-coins-t x ∈ U)
⟨proof⟩

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

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

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

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

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

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

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

lemma coin-space-is-borel-measure:
  assumes open U
  shows U ∈ sets B
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⟩

**lemma** openin-option-ud: openin option-ud S ←→ (S = UNIV ∨ None ∉ S)
   ⟨proof⟩

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

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

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

end

4 Randomized Algorithms (Internal Representation)

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

**begin**

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

This is the inverse of set-option

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

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

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

**definition** at-most-one :: 'a set ⇒ bool
   where at-most-one S ←→ (∀ 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 {}

11
shows $P S$
⟨proof⟩

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

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

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

**type-synonym** 'a 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]. 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

---

\[1\] Although we were not aware of the technical report, when initially considering this representation.
would pass the left sub-tree to the first algorithm and the right sub-tree to the second algorithm. This model has the dis-advantage that the resulting “monad”, does not fulfill the associativity law. Moreover many PRG’s are designed and tested in the streaming sense, and there is not a lot of research into the performance of PRGs with tree structured output. (A related idea was to still use a stream as input, and split it into two sub-streams for example by the parity of the stream position. This alternative also suffers from the lack of associativity problem and may lead to a lot of unused coin flips.)

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

Monad laws:

**Lemma** `return-bind-rai`: `bind-rai (return-rai x) g = g x`  
(proof)

**Lemma** `bind-rai-assoc`: `bind-rai (bind-rai f g) h = bind-rai f (λx. bind-rai (g x) h)`  
(proof)

**Lemma** `bind-return-rai`: `bind-rai m return-rai = m`  
(proof)

**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)

**Definition** `distr-rai` :: `a random-alg-int ⇒ 'a option measure`  
where `distr-rai f = distr B D (map-option fst ◦ f)`

**Lemma** `wf-randomI`:
  assumes `wf-random m`  
  assumes `∀ x. x ∈ range-rm m ⇒ wf-random (f x)`  
sows `wf-random (m >>= f)`  
(proof)

**Lemma** `wf-on-prefix-bindI`:
  assumes `wf-on-prefix-bind m p r`  
  assumes `wf-on-prefix-bind (f r) q s`  
sows `wf-on-prefix-bind (m >>= f) (p@q) s`  
(proof)

**Lemma** `wf-bind`:
  assumes `wf-random m`  
  assumes `∀ x. x ∈ range-rm m ⇒ wf-random (f x)`  
sows `wf-random (m >>= f)`
lemma wf-return:  
\text{wf-random (return-rai x)}
\langle proof \rangle

lemma wf-coin:  
\text{wf-random (coin-rai)}
\langle proof \rangle

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

definition eval-rm :: `'a random-alg-int ⇒ bool list ⇒ `'a where
\text{eval-rm f p} = \text{fst (the (f (cshift p (cconst False))))}

lemma eval-rmD:  
\text{assumes \text{wf-on-prefix f p r}}  
\text{shows eval-rm f p} = r
\langle proof \rangle

lemma wf-on-prefixD:  
\text{assumes \text{wf-on-prefix f p r}}  
\text{assumes \text{cprefix p bs}}  
\text{shows f bs} = \text{Some (eval-rm f p, cdrop (length p) bs)}
\langle proof \rangle

lemma prefixes-parallel-helper:  
\text{assumes p ∈ ptree-rm f}  
\text{assumes q ∈ ptree-rm f}  
\text{assumes prefix p q}  
\text{shows p} = q
\langle proof \rangle

lemma prefixes-parallel:  
\text{assumes p ∈ ptree-rm f}  
\text{assumes q ∈ ptree-rm f}  
\text{shows p} = q ∨ p \parallel q
\langle proof \rangle

lemma prefixes-singleton:  
\text{assumes p ∈ \{p. p ∈ ptree-rm f ∧ cprefix p bs\}}  
\text{shows \{p ∈ ptree-rm f, cprefix p bs\} = \{p\}}
\langle proof \rangle

lemma prefixes-at-most-one:  
\text{at-most-one \{p ∈ ptree-rm f, cprefix p x\}}
\langle proof \rangle

definition consumed-prefix f bs = \text{the-elem-opt \{p ∈ ptree-rm f, cprefix p bs\}}

lemma wf-random-alt:  
\text{assumes \text{wf-random f}}  
\text{shows f bs} = \text{map-option (λp. (eval-rm f p, cdrop (length p) bs)) (consumed-prefix f bs)}
\langle proof \rangle

lemma range-rm-alt:  
\text{assumes \text{wf-random f}}
shows range-rm f = eval-rm f ` ptree-rm f (is ?L = ?R)
⟨proof⟩

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

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

definition used-bits-distr :: 'a random-alg-int ⇒ nat option measure
where used-bits-distr f = distr B D (consumed-bits f)

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

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

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

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

lemma consumed-bits-measurable: consumed-bits f ∈ B → M D
⟨proof⟩

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

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

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

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

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

lemma none-measure-subprob-algebra:
return \( \mathcal{D} \) None \( \in \) space \((\text{subprob-algebra} \ \mathcal{D})\) 
\(\langle\text{proof}\rangle\)

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

lemma the-f-measurable: \(\text{the} \circ f \in R \rightarrow_{\mathcal{M}} \mathcal{D} \otimes_{\mathcal{M}} \mathcal{B}\)
\(\langle\text{proof}\rangle\)

lemma distr-rai-measurable: \(\text{map-option} \ \text{fst} \circ f \in \mathcal{B} \rightarrow_{\mathcal{M}} \mathcal{D}\)
\(\langle\text{proof}\rangle\)

lemma distr-rai-subprob-space:
\(\text{distr-rai} \ f \in \text{space} \ (\text{subprob-algebra} \ \mathcal{D})\)
\(\langle\text{proof}\rangle\)

lemma fst-the-f-measurable: \(\text{fst} \circ \text{the} \circ f \in R \rightarrow_{\mathcal{M}} \mathcal{D}\)
\(\langle\text{proof}\rangle\)

lemma prob-space-distr-rai:
\(\text{prob-space} \ (\text{distr-rai} \ f)\)
\(\langle\text{proof}\rangle\)

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:
\(\text{distr} \ R \ (\mathcal{D} \otimes_{\mathcal{M}} \mathcal{B}) \ (\text{the} \circ f) = \text{distr} \ R \ \mathcal{D} \ (\text{fst} \circ \text{the} \circ f) \otimes_{\mathcal{M}} \mathcal{B}\)
\(\langle\text{proof}\rangle\)

end

lemma distr-rai-bind:
assumes \(\text{wf-m}: \text{wf-random} \ m\)
assumes \(\text{wf-f}: \exists x. \ x \in \text{range-rm} \ m \Rightarrow \text{wf-random} \ (f \ x)\)
shows \(\text{distr-rai} \ (m \gg f) = \text{distr-rai} \ m \gg \)
\(\lambda x. \text{if } x \in \text{Some} \ (\text{range-rm} \ m) \text{ then } \text{distr-rai} \ (f \ (\text{the} \ x)) \text{ else } \text{return} \ \mathcal{D} \ \text{None}\)
\(\text{is } ?L = ?R\)
\(\langle\text{proof}\rangle\)

lemma return-discrete: \(\text{return} \ \mathcal{D} \ x = \text{return-pmf} \ x\)
\(\langle\text{proof}\rangle\)

lemma distr-rai-return: \(\text{distr-rai} \ (\text{return-rai} \ x) = \text{return} \ \mathcal{D} \ (\text{Some} \ x)\)
\(\langle\text{proof}\rangle\)

lemma distr-rai-return': \(\text{distr-rai} \ (\text{return-rai} \ x) = \text{return-spmf} \ x\)
\(\langle\text{proof}\rangle\)

lemma distr-rai-coin: \(\text{distr-rai} \ \text{coin-rai} = \text{coin-spmf} \ (\text{is } ?L = ?R)\)
\(\langle\text{proof}\rangle\)

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

interpretation random-alg-int-pd: partial-function-definitions ord-rai lub-rai
(proof)

lemma wf-lub-helper:
  assumes ord-rai f g
  assumes wf-on-prefix f p r
  shows wf-on-prefix g p r
(proof)

lemma wf-lub:
  assumes Complete-Partial-Order.chain ord-rai R
  assumes ∀r. r ∈ R ⇒ wf-random r
  shows wf-random (lub-rai R)
(proof)

lemma ord-rai-mono:
  assumes ord-rai f g
  assumes ¬ (P None)
  assumes P (f bs)
  shows P (g bs)
(proof)

lemma lub-rai-empty:
  lub-rai {} = Map.empty
(proof)

lemma distr-rai-lub:
  assumes F ≠ {}
  assumes Complete-Partial-Order.chain ord-rai F
  assumes wf-f: ∀f. f ∈ F ⇒ wf-random f
  assumes None ∉ A
  shows emeasure (distr-rai (lub-rai F)) A = (SUP f ∈ F. emeasure (distr-rai f) A) (is ?L = ?R)
(proof)

lemma distr-rai-ord-rai-mono:
  assumes wf-random f wf-random g ord-rai f g
  assumes None ∉ A
  shows emeasure (distr-rai f) A ≤ emeasure (distr-rai g) A (is ?L ≤ ?R)
(proof)

lemma distr-None: distr (λ-. None) = measure-pmf (return-pmf (None :: 'a option))
(proof)

lemma bind-rai-mono:
  assumes ord-rai f1 f2 ∃y. ord-rai (g1 y) (g2 y)
  shows ord-rai (bind-rai f1 g1) (bind-rai f2 g2)
(proof)

end
5 Randomized Algorithms

This section introduces the \textit{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 \textit{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.

\texttt{theory Randomized-Algorithm}

\texttt{imports}

\hspace{1em} Randomized-Algorithm-Internal

\texttt{begin}

A stronger variant of \textit{pmf-eqI}.

\texttt{lemma pmf-eq-iff-le:}

\hspace{1em} \texttt{fixes p q :: 'a pmf}

\hspace{1em} \texttt{assumes } \Lambda x. \texttt{pmf p x }\leq \texttt{pmf q x}

\hspace{1em} \texttt{shows } \texttt{p = q}

\hspace{1em} \langle \texttt{proof} \rangle

The following is a stronger variant of \textit{ord-spmf-eq-pmf-None-eq}.

\texttt{lemma eq-iff-ord-spmf:}

\hspace{1em} \texttt{assumes } \texttt{weight-spmf p }\geq \texttt{weight-spmf q}

\hspace{1em} \texttt{assumes } \texttt{ord-spmf (=) p q}

\hspace{1em} \texttt{shows } \texttt{p = q}

\hspace{1em} \langle \texttt{proof} \rangle

\texttt{lemma wf-empty: wf-random (\lambda -. None) }

\hspace{1em} \langle \texttt{proof} \rangle

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

\hspace{1em} \langle \texttt{proof} \rangle

\texttt{setup-lifting type-definition-random-alg}

\texttt{lift-definition return-ra :: 'a }\Rightarrow\texttt{'a random-alg is return-ra}

\hspace{1em} \langle \texttt{proof} \rangle

\texttt{lift-definition coin-ra :: bool random-alg is coin-ra}

\hspace{1em} \langle \texttt{proof} \rangle

\texttt{lift-definition bind-ra :: 'a random-alg }\Rightarrow\texttt{(\lambda a \Rightarrow \texttt{'b random-alg}) }\Rightarrow\texttt{'b random-alg is bind-ra}

\hspace{1em} \langle \texttt{proof} \rangle

\texttt{adhoc-overloading Monad-Syntax.bind bind-ra}

Monad laws:

\texttt{lemma return-bind-ra:}

\hspace{1em} \texttt{bind-ra (return-ra x) g }\Rightarrow\texttt{g x}

\hspace{1em} \langle \texttt{proof} \rangle

\texttt{lemma bind-ra-assoc:}

18
bind-ra \((\text{bind-ra } f g) \ h = \text{bind-ra } f (\lambda x. \text{bind-ra } (g \ x) \ h)\)

(proof)

lemma bind-return-ra:
bind-ra m return-ra = m
(proof)

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

lift-definition ord-ra :: 'a random-alg ⇒ 'a random-alg ⇒ bool is ord-rai (proof)

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

context
begin

interpretation pmf-as-measure (proof)

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

lift-definition spmf-of-ra :: 'a random-alg ⇒ 'a spmf is distr-rai
(proof)

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

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

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

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

lemma set-pmf-spmf-of-ra:
set-pmf (spmf-of-ra f) ⊆ Some ' range-rm (Rep-random-alg f) ∪ \{None\}
(proof)

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

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

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

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

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

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

lemma partial-function-image-improved:
  fixes ord
  assumes λA. 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⟩

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

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

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

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

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

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

lemma spmf-of-ra-map: spmf-of-ra (map-ra f p) = map-spmf f (spmf-of-ra p)
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))
  ⟨proof⟩

definition rel-spmf-of-ra :: `'a spmf ⇒ `'a random-alg ⇒ bool where
  rel-spmf-of-ra q p ←→ q = spmf-of-ra p

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

lemma admissible-rel-spmf-of-ra-cont [cont-intro]:
  fixes ord
  shows [ mcont lub ord lub-spmf (ord-spmf (=)) f; mcont lub ord lub-ra ord-ra g ]
  \implies ccpo.admissible lub ord (\lambda x. rel-spmf-of-ra (f x) (g x))
  ⟨proof⟩

lemma mcont2mcont-spmf-of-ra[THEN spmf.mcont2mcont, cont-intro]:
  ⟨proof⟩

context
  includes lifting-syntax
begin

lemma fixp-ra-parametric[transfer-rule]:
  assumes f: \A x. mono-spmf (\lambda f. F f x)
  and g: \A x. mono-ra (\lambda g. G f x)
  and param: (A ===> rel-spmf-of-ra) ===> A ===> rel-spmf-of-ra F G
  ⟨proof⟩

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

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

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

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

end

declare [[function-internals]]
5.1 Almost surely terminating randomized algorithms

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

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

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

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

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

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

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

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

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

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

lemma terminates-almost-surely-bind:
  assumes terminates-almost-surely f
  assumes ∀x. x ∈ set-pmf (pmf-of-ra f) → terminates-almost-surely (g x)
  shows terminates-almost-surely (f >>= g)
  ⟨proof⟩

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

lemmas pmf-of-ra-simps =

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

```
theory Tracking-Randomized-Algorithm
  imports Randomized-Algorithm
begin

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

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

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

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

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

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

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

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

lemma track-rb-bind:
```
\textbf{assumes} \( \text{wf-random } m \)
\textbf{assumes} \( \forall x. x \in \text{range-rm } m \implies \text{wf-random } (f x) \)
\textbf{shows} \( \text{track-random-bits } (m \gg f) = \text{track-random-bits } m \gg \)
\( (\lambda (r,n). \text{track-random-bits } (f r) \gg (\lambda (r',m). \text{return-rai } (r',n+m))) \) (is \(?L = ?R\))
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{track-random-bits-mono}: \)
\textbf{assumes} \( \text{wf-random } f \ \text{wf-random } g \)
\textbf{assumes} \( \text{ord-rai } f \ g \)
\textbf{shows} \( \text{ord-rai } (\text{track-random-bits } f) (\text{track-random-bits } g) \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{wf-track-random-bits}: \)
\textbf{assumes} \( \text{wf-random } f \)
\textbf{shows} \( \text{wf-random } (\text{track-random-bits } f) \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{track-random-bits-lub-rai}: \)
\textbf{assumes} \( \text{Complete-Partial-Order}. \text{chain ord-rai } A \)
\textbf{assumes} \( \forall r. r \in A \implies \text{wf-random } r \)
\textbf{shows} \( \text{track-random-bits } (\text{lub-rai } A) = \text{lub-rai } (\text{track-random-bits } A) \) (is \(?L = ?R\))
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{untrack-random-bits}: \)
\textbf{assumes} \( \text{wf-random } f \)
\textbf{shows} \( \text{track-random-bits } f \gg \)
\( (\lambda x. \text{return-rai } (x,0)) = f \) (is \(?L = ?R\))
\( \langle \text{proof} \rangle \)

\textbf{lift-definition} \( \text{track-coin-use} :: 'a \text{ random-alg } \Rightarrow ('a \times \text{nat}) \text{ random-alg} \)
is \( \text{track-random-bits} \)
\( \langle \text{proof} \rangle \)

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

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

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

\textbf{adhoc-overloading} \( \text{Monad-Syntax}. \text{bind } \text{bind-tra} \)

\textbf{Monad laws:}

\textbf{lemma} \( \text{return-bind-tra}: \)
\( \text{bind-tra } (\text{return-tra } x) \ g = g \ x \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{bind-tra-assoc}: \)
\( \text{bind-tra } (\text{bind-tra } f \ g) \ h = \text{bind-tra } f \ (\lambda x. \text{bind-tra } (g \ x) \ h) \)
lemma bind-return-tra:
bind-tra m return-tra = m
⟨proof⟩

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

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

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

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

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

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

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

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

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

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

lemma admissible-rel-track-coin-use-cont [cont-intro]:
  fixes ord
  shows [ mcont lub ord lub-ra ord-ra f; mcont lub ord lub-ra ord-ra g ]
\[ \Rightarrow \text{ccpo.admissible} \text{ lub ord} \; (\lambda x. \text{rel-track-coin-use} (f \; x) \; (g \; x)) \]

**lemma** mcont-track-coin-use:
\[ \text{mcont} \; \text{lub-ra \; ord-ra \; lub-ra \; ord-ra \; track-coin-use} \]

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

**context** includes lifting-syntax

**begin**

**lemma** fixp-track-coin-use-parametric[transfer-rule]:
\[ \text{assumes} \; f: \forall x. \text{mono-ra} \; (\lambda f. \; F \; f \; x) \]
\[ \text{and} \; g: \forall x. \text{mono-ra} \; (\lambda f. \; G \; f \; x) \]
\[ \text{and} \; \text{param}: \; ((A \; === \; \text{rel-track-coin-use}) \; === \; A \; === \; \text{rel-track-coin-use}) \; F \; G \]
\[ \text{shows} \; (A \; === \; \text{rel-track-coin-use}) \; \text{(random-alg-pf.fixp-fun} \; F) \; \text{(random-alg-pf.fixp-fun} \; G) \]

**lemma** return-ra-tranfer[transfer-rule]:
\[ (\; \text{(=} \; === \; \text{rel-track-coin-use}) \; \text{return-tra} \; \text{return-ra} \]

**lemma** bind-ra-tranfer[transfer-rule]:
\[ (\text{rel-track-coin-use} \; === \; (\; \text{(=} \; === \; \text{rel-track-coin-use}) \; === \; \text{rel-track-coin-use}) \; \text{bind-tra} \]

**lemma** coin-ra-tranfer[transfer-rule]:
\[ \text{rel-track-coin-use} \; \text{coin-tra} \; \text{coin-ra} \]

**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 \times nat) spmf

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

**definition** coin-tspmf :: bool tspmf where
\[ \text{coin-tspmf} = \text{pair-spmf} \; \text{coin-spmf} \; (\text{return-spmf} \; 1) \]

**definition** bind-tspmf :: 'a tspmf \Rightarrow ('a \Rightarrow 'b tspmf) \Rightarrow 'b tspmf where
\[ \text{bind-tspmf} \; f \; g = \text{bind-spmf} \; f \; (\lambda (r,c). \text{map-spmf} \; (\text{apsnd} \; ((+) \; c)) \; (g \; r)) \]
**adhoc-overloading** *Monad-Syntax.bind bind-tspmf*

Monad laws:

**lemma** `return-bind-tspmf`:  
\[ bind-tspmf (return-tspmf x) g = g x \]  
(proof)

**lemma** `bind-tspmf-assoc`:  
\[ bind-tspmf (bind-tspmf f g) h = bind-tspmf f (\lambda x. bind-tspmf (g x) h) \]  
(proof)

**lemma** `bind-return-tspmf`:  
\[ bind-tspmf m return-tspmf = m \]  
(proof)

**lemma** `bind-mono-tspmf-aux`:  
\begin{align*}  &\text{assumes ord-spmf} (=) f1 f2 \land y. \text{ord-spmf} (=) (g1 y) (g2 y) \\  &\text{shows ord-spmf} (=) (\text{bind-tspmf} f1 g1) (\text{bind-tspmf} f2 g2) \end{align*}  
(proof)

**lemma** `bind-mono-tspmf` [partial-function-mono]:  
\begin{align*}  &\text{assumes mono-spmf} B \land y. \text{mono-spmf} (C y) \\  &\text{shows mono-spmf} (\lambda f. \text{bind-tspmf} (B f) (\lambda y. C y f)) \end{align*}  
(proof)

**definition** `ord-tspmf` :: `a tspmf` ⇒ `a tspmf` ⇒ `bool` where  
\[ \text{ord-tspmf} = \text{ord-spmf} (\lambda x y. \text{fst} x = \text{fst} y \land \text{snd} x \geq \text{snd} y) \]

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

**definition** `expected-coin-usage-of-tspmf` :: `a tspmf` ⇒ `ennreal` where  
\[ \text{expected-coin-usage-of-tspmf} p = (\int^+ x. x \partial (\text{map-pmf ennreal-of-enat} (\text{coin-usage-of-tspmf } p))) \]

**definition** `expected-coin-usage-of-ra` where  
\[ \text{expected-coin-usage-of-ra } p = (\int^+ x. x \partial (\text{map-pmf ennreal-of-enat} (\text{coin-usage-of-ra } p))) \]

**definition** `result` :: `a tspmf` ⇒ `a spmf` where  
\[ \text{result} = \text{map-spmf} \text{fst} \]

**lemma** `coin-usage-of-tspmf-alt-def`:  
\[ \text{coin-usage-of-tspmf} p = \text{map-pmf} (\lambda x. \text{case } x \text{ of } \text{None } \Rightarrow \infty | \text{Some } y \Rightarrow \text{enat } y) (\text{map-spmf snd } p) \]  
(proof)
lemma coin-usage-of-tspmf-bind-return:
    coin-usage-of-tspmf (bind-tspmf f (λx. return-tspmf (g x))) = (coin-usage-of-tspmf f)
    ⟨proof⟩

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

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

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⟩

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

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

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

lemma ord-tspmf-map-spmf:
    assumes ∀x. x ≤ f x
    shows ord-tspmf (map-spmf (apsnd f) p) p
    ⟨proof⟩

lemma ord-tspmf-bind-pmf:
    assumes ∀x. ord-tspmf (f x) (g x)
    shows ord-tspmf (bind-pmf p f) (bind-pmf p g)
    ⟨proof⟩

lemma ord-tspmf-bind-tspmf:
    assumes ∀x. ord-tspmf (f x) (g x)
    shows ord-tspmf (bind-tspmf p f) (bind-tspmf p g)
    ⟨proof⟩

definition use-coins :: nat ⇒ ′a tspmf ⇒ ′a tspmf
    where use-coins k = map-spmf (apsnd ((+) k))

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

lemma coin-tspmf-split:
    fixes f :: bool ⇒ ′b tspmf
    shows (coin-tspmf ≧ f) = use-coins 1 (coin-spmf ≧ f)
proof

lemma ord-tspmfa-use-coins:
  ord-tspmfa (use-coins k p) p
  proof

lemma ord-tspmfa-use-coins-2:
  assumes ord-tspmfa p q
  shows ord-tspmfa (use-coins k p) (use-coins k q)
  proof

lemma result-mono:
  assumes ord-tspmfa p q
  shows ord-spmfa (=) (result p) (result q)
  proof

lemma result-bind:
  result (bind-tspmfa f g) = result f \approx (\lambda x. result (g x))
  proof

lemma result-return:
  result (return-tspmfa x) = return-spmfa x
  proof

lemma result-coin:
  result (coin-tspmfa) = coin-spmfa
  proof

definition tspmf-of-ra :: 'a random-alg \Rightarrow 'a tspmf where
tspmfa-of-ra = spmfa-of-ra \circ track-coin-use

lemma tspmf-of-ra-coin: tspmf-of-ra coin-ra = coin-tspmfa
  proof

lemma tspmf-of-ra-return: tspmf-of-ra (return-ra x) = return-tspmfa x
  proof

lemma tspmf-of-ra-bind:
  tspmf-of-ra (bind-ra m f) = bind-tspmfa (tspmfa-of-ra m) (\lambda x. tspmf-of-ra (f x))
  proof


lemma tspmf-of-ra-mono:
  assumes ord-ra f g
  shows ord-spmfa (=) (tspmfa-of-ra f) (tspmfa-of-ra g)
  proof

lemma tspmf-of-ra-lub:
  assumes Complete-Partial-Order.chain ord-ra A
  shows tspmf-of-ra (lub-ra A) = lub-spmfa (tspmfa-of-ra ' A) (is ?L = ?R)
  proof

definition rel-tspmfa-of-ra :: 'a tspmf \Rightarrow 'a random-alg \Rightarrow bool where
rel-tspmfa-of-ra q p \iff q = tspmf-of-ra p

lemma admissible-rel-tspmfa-of-ra:
ccpo.admissible (prod-lub lub-spmfa lub-ra) (rel-prod (ord-spmfa (=)) ord-ra) (case-prod rel-tspmfa-of-ra)
lemma admissible-rel-tspmfof-ra-cont [cont-intro]:
fixes ord
shows \[ \text{mcont\ lub\ ord\ lub-spmf\ (ord-spmf\ (=))\ f; \ mcont\ lub\ ord\ lub-ra\ ord-ra\ g} \] \[ \implies \text{ccpo.admissible\ lub\ ord\ (} \lambda x. \text{rel-tspmfof-ra\ (f x)\ (g x)} \) \]
\<proof> \n
lemma mcont-tspmfof-ra:
\text{mcont\ lub-ra\ ord-ra\ lub-spmf\ (ord-spmf\ (=))\ tspmfof-ra} 
\<proof> 

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

context includes lifting-syntax
begin

lemma fixp-rel-tspmfof-ra-parametric[transfer-rule]:
assumes \( f: \forall x. \text{mono-spmf}\ (\lambda f. F f x) \) 
and \( g: \forall x. \text{mono-ra}\ (\lambda f. G f x) \)
and \( \text{param}: ((A \implies \text{rel-tspmfof-ra}) \implies A \implies \text{rel-tspmfof-ra}) F G \)
shows \( (A \implies \text{rel-tspmfof-ra}) \quad \text{spmf.fixp-fun F} \) \( \text{random-algpf.fixp-fun G} \)
\<proof> 

lemma return-ra-tranfer[transfer-rule]: \( (\text{=} \implies \text{rel-tspmfof-ra}) \quad \text{return-tspmfof-ra} \)
\<proof> 

lemma bind-ra-tranfer[transfer-rule]:
\( (\text{rel-tspmfof-ra} \implies (\text{=} \implies \text{rel-tspmfof-ra}) \implies \text{rel-tspmfof-ra}) \quad \text{bind-tspmfof-ra} \)
\<proof> 

lemma coin-ra-tranfer[transfer-rule]:
\( \text{rel-tspmfof-ra} \quad \text{coin-tspmfof-ra} \quad \text{coin-ra} \)
\<proof> 

end

lemma spmf-of-tspmfof:
\( \text{result} \quad (\text{tspmfof-ra} \quad f) = \text{spmfof-ra} \quad f \)
\<proof> 

lemma coin-usage-of-tspmfof-correct:
\( \text{coin-usage-of-tspmfof} \quad (\text{tspmfof-ra} \quad p) = \text{coin-usage-of-ra} \quad p \quad \text{(is L = R)} \)
\<proof> 

lemma expected-coin-usage-of-tspmfof-correct:
\( \text{expected-coin-usage-of-tspmfof} \quad (\text{tspmfof-ra} \quad p) = \text{expected-coin-usage-of-ra} \quad p \)
\<proof> 

end

8 Dice Roll

theory Dice-Roll
imports Tracking-SPMF
begin
The following is a dice roll algorithm for an arbitrary number of sides \( n \). Besides correctness we also show that the expected number of coin flips is at most \( \log_2 n + 2 \). It is a specialization of the algorithm presented by Hao and Hoshi [4].

**Lemma** \( \text{floor-} \leq \text{ceil-} \text{minus-one} \):

**Fixes** \( x \), \( y :: \text{real} \)
**Shows** \( x < y \implies \lfloor x \rfloor \leq \lceil y \rceil - 1 \)

**Lemma** \( \text{combine-spmf-} \text{set-} \text{coin-} \text{spmf} \):

**Fixes** \( f :: \text{nat} \Rightarrow 'a \text{ spmf} \)
**Fixes** \( k :: \text{nat} \)
**Shows** \( \text{pmf-of-set} \{..<2^k \} \geq (\lambda l. \text{coin-} \text{spmf} \geq (\lambda b. f (2*l+ \text{ of-bool } b))) = \text{pmf-of-set} \{..<2^{(k+1)} \} \geq f (\text{is } ?L = ?R) \)

**Lemma** \( \text{count-} \text{ints-} \text{in-} \text{range} \):

\[ \text{real} \left( \text{card} \{x. \text{of-int } x \in \{u..v\} \} \geq v - u - 1 \ (\text{is } ?L \geq ?R) \right) \]

**Partial-Function** \( \text{(random-} \text{alg)} \text{ dice-roll-step-} \text{ra} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{int} \text{ random-} \text{alg} \)
**Where** \( \text{dice-roll-step-} \text{ra } l \ h = (\)
  \( \text{if } \lfloor l \rfloor = \lfloor l + h \rfloor - 1 \text{ then} \)
  \( \text{return-} \text{ra } \lfloor l \rfloor \)
  \( \text{else} \)
  \( \text{do } \{ b \leftarrow \text{coin-} \text{ra}; \text{dice-roll-step-} \text{ra } (l + (h/2) * \text{ of-bool } b) (h/2) \} \)
\( ) \)

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

**Partial-Function** \( (\text{spmfn}) \text{ drs-tspmf } :: \text{real} \Rightarrow \text{real} \Rightarrow \text{int} \text{ tspmf} \)
**Where** \( \text{drs-tspmf } l \ h = (\)
  \( \text{if } \lfloor l \rfloor = \lfloor l + h \rfloor - 1 \text{ then} \)
  \( \text{return-tspmf } \lfloor l \rfloor \)
  \( \text{else} \)
  \( \text{do } \{ b \leftarrow \text{coin-tspmf}; \text{drs-tspmf } (l + (h/2) * \text{ of-bool } b) (h/2) \} \)
\( ) \)

**Definition** \( \text{dice-roll-tspmf } n = \text{drs-tspmf } 0 (\text{of-nat } n) \geq (\lambda x. \text{return-tspmf } (\text{nat } x)) \)

**Lemma** \( \text{drs-tspmf} : \text{drs-tspmf } l \ u = \text{spmfn-of-} \text{ra } (\text{dice-roll-step-} \text{ra } l \ u) \)

**Include** lifting-syntax

**Lemma** \( \text{dice-roll-} \text{ra-tspmf } : \text{spmfn-of-} \text{ra } (\text{dice-roll-} \text{ra } n) = \text{dice-roll-tspmf } n \)

**Lemma** \( \text{dice-roll-step-tspmf-} \text{lb} \):
**Assumes** \( h > 0 \)
**Shows** \( \text{coin-tspmf} \geq (\lambda b. \text{drs-tspmf } (l + (h/2) * \text{ of-bool } b) (h/2)) \leq_R \text{drs-tspmf } l \ h \)

**Abbreviation** \( \text{coins } k \equiv \text{spmfn-of-} \text{set } \{..<(2::nat)\}^k \)

**Lemma** \( \text{dice-roll-step-tspmf-} \text{expand} \):
**Assumes** \( h > 0 \)

\(^3\text{An interesting alternative algorithm, which we did not formalized here, has been introduced by Lambruso [7].}\)
shows coins k ≥ (λ l. use-coins k (drs-tspmf (real l*h) h)) ≤R drs-tspmf 0 (h*2^k)
⟨proof⟩

lemma dice-roll-step-tspmf-approx:
fixes k :: nat
assumes h > (0::real)
defines f ≡ (λ l. if ⌈l*h⌉=⌈(l+1)*h⌉ then Some (⌈l*h⌉,k) else None)
shows map-pmf f (coins k) ≤R drs-tspmf 0 (h*2^k) (is ?L ≤R ?R)
⟨proof⟩

lemma dice-roll-step-spmf-approx:
fixes k :: nat
assumes h > (0::real)
defines f ≡ (λ l. if ⌈l*h⌉=⌈(l+1)*h⌉ then Some (⌈l*h⌉) else None)
shows ord-spmf (=) (map-pmf f (coins k)) (result (drs-tspmf 0 (h*2^k))
(is ord-spmf - ?L ?R)
⟨proof⟩

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

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

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

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

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

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⟩

end

9 A Pseudo-random Number Generator

In this section we introduce a PRG, that can be used to generate random bits. It is an implementation of O’Neill’s Permuted congruential generator [9] (specifically PCG-XSH-
In empirical tests it ranks high [2, 10] while having a low implementation complexity. This is for easy testing purposes only, the generated code can be run with any source of random bits.

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

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

definition pcg-mult :: 64 word
  where pcg-mult = 6364136223846793005

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

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

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

Based on [9, Section 6.3.1]:

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

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

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

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

10 Basic Randomized Algorithms

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

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

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

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

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

\textbf{lemma} binary-dice-roll-terminates: terminates-almost-surely (binary-dice-roll n)

The corresponding PMF can be written as:

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

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

\textbf{lemma} pmf-of-ra (binary-dice-roll n) = binary-dice-roll-pmf n

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

\textbf{partial-function} (random-alg) binary-geometric :: nat ⇒ nat random-alg
\textbf{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 \textbf{partial-function} directive:

\textbf{declare} binary-geometric.simps[code]

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

\textbf{partial-function} (spmf) binary-geometric-spmf :: nat ⇒ nat spmf
\textbf{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 \texttt{spmf-of-ra} to show the correspondence:

\textbf{lemma} binary-geometric-ra-correct:
  spmf-of-ra (binary-geometric x) = binary-geometric-spmf x

\textbf{include} lifting-syntax

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

\[ \text{partial-function} \ (\text{random-alg}) \ 
\text{bernoulli-ra} :: \ 
\text{real} \Rightarrow \ 
\text{bool} \ 
\text{random-alg} \ 
\text{where} \ 
\begin{align*}
\text{bernoulli-ra} \ p &= \ 
\begin{cases} 
\text{do} \ 
\begin{align*}
\ & b \leftarrow \ 
\text{coin-ra}; 
\ & \text{if} \ b \text{ then return-ra} \ (p \geq 1/2) 
\ & \text{else if} \ p < 1/2 \text{ then bernoulli-ra} \ (2 \ast p) 
\ & \text{else bernoulli-ra} \ (2 \ast p - 1)
\end{align*}
\end{cases}
\end{align*}
\]

\text{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:

\[ \text{lemma} \ bernoulli-ra-correct-aux: \ ord-spmf \ (=) \ (\text{bernoulli} \ x) \ (spmf-of-ra \ (\text{bernoulli-ra} \ x)) \]

\[ \langle \text{proof} \rangle \]

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

\[ \text{lemma} \ bernoulli-ra-correct: \ bernoulli \ x = \ spmf-of-ra \ (\text{bernoulli-ra} \ x) \]

\[ \langle \text{proof} \rangle \]

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

\[ \text{lemma} \ 
\begin{align*}
\ & \text{terminates-almost-surely} \ (\text{bernoulli-ra} \ p) 
\ & \text{bernoulli-pmf} \ p = \ pmf-of-ra \ (\text{bernoulli-ra} \ p)
\end{align*}
\]

\[ \langle \text{proof} \rangle \]

\text{context}

\text{includes lifting-syntax}

\text{begin}

\[ \text{lemma} \ bernoulli-ra-transfer \ [\text{transfer-rule}]: \]

\[ ((=) \Longrightarrow \ rel-spmf-of-ra) \ bernoulli \ bernoulli-ra \]

\[ \langle \text{proof} \rangle \]

\text{end}

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

\[ \text{partial-function} \ (\text{random-alg}) \ 
\text{geometric-ra} :: \ 
\text{real} \Rightarrow \ 
\text{nat} \ 
\text{random-alg} \ 
\text{where} \ 
\begin{align*}
\text{geometric-ra} \ p &= \ 
\begin{cases} 
\text{do} \ 
\begin{align*}
\ & b \leftarrow \ bernoulli-ra \ p; 
\ & \text{if} \ b \text{ then return-ra} \ 0 \text{ else map-ra} \ ((+) \ 1) \ (\text{geometric-ra} \ p)
\end{align*}
\end{cases}
\end{align*}
\]

\text{declare geometric-ra.simps[code]}

\[ \text{lemma} \ geometric-ra-correct: \ spmf-of-ra \ (\text{geometric-ra} \ x) = \text{geometric-spmf} \ x \]

\[ \langle \text{proof} \rangle \]

\text{include lifting-syntax}

\[ \langle \text{proof} \rangle \]

Replication of a distribution

\[ \text{fun} \ replicate-ra :: \ 
\text{nat} \Rightarrow \ 'a \text{ random-alg} \Rightarrow \ 'a \text{ list random-alg} \]

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

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

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

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

Binomial distribution

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

lemma
  assumes p ∈ {0..1}
  shows spmf-of-ra (binomial-ra n p) = spmf-of-pmf (binomial-pmf n p)
(proof)

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

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

end

References