Gambler's Ruin Problem

Zibo Yang

July 25, 2021

Contents

theory *Gambler-Ruin-Problem*
imports *DiscretePricing.Infinite-Coin-Toss-Space*

begin

1 Gambler's ruin problem

In Gamblerruin.thy, we will construct the formalization of a specific random walk model coordinated with gambler's ruin problem.

1.1 Theory Infinite_Coin_Toss_Space

In order to construct the formal method in gambler's ruin problem, we start with the existing formalization in the Theory Infinite_Coin_Toss_Space which constructed the probability space on infinite sequences of independent coin. tosses.

The only concept need to be elaborated is bernoulli. 'a stream is a type of infinite sequence with all element of type 'a. The bernoulli stream is a stream measure which has the space composed of all the elements of type bool stream, measurable sets filled with all the subset of its space. under this specific measure, all the possibility of occurrence of elements in a specific set A can be described as the measure value of A if and only if A is in the measurable sets of this bernoulli stream. In fact, it was set up by producting countable measure of boolean space with measuring {True} to p and {False} to $1 - p$.

1.2 Gambler model

fun (**in** *infinite-coin-toss-space*) *gambler-rand-walk-pre*:: *int* ⇒ *int* ⇒ *int* ⇒ (*nat* ⇒ *bool stream* ⇒ *int*) **where**
 base: *gambler-rand-walk-pre u d v 0 w* = *v*|
 step1: *gambler-rand-walk-pre u d v* (*Suc n*) *w* = ((λ*True* ⇒ *u* | *False* ⇒ *d*) (*snth w n*)) + *gambler-rand-walk-pre u d v n w*

fun (**in** *infinite-coin-toss-space*) *gambler-rand-walk*:: *int* ⇒ *int* ⇒ *int* ⇒ (*enat* ⇒ *bool stream* ⇒ *int*) **where**
gambler-rand-walk u d v n w = (*case n of enat n* ⇒ (*gambler-rand-walk-pre u d v n w*)|∞ ⇒ −*1*)

The function *gambler_ran_walk* extends the fourth parameter by adding ∞ as new input. The reason why we define it is that we found it very tough to describe the position where the specific random walk stops, for the first time, by reaching the threshold if natural number is the only allowed input as what *gambler_ran_walk_pre* defines. Since some infinite random walks will never stop, we must allocate ∞ as the output coordinated with that non-stop case and extend the type of steps from *nat* to *enat*. But if someone wants to base their further analysis on our endeavor here, please be cautious of or even avoid discussing the case that initial number and target number is negative since we map ∞ to -1. The lemma *exist* demonstrates that non-stop random walk will never succeed in reaching the target, which is the best explanation why we allocates -1 as the output of ∞ in *gambler_ran_walk*.

locale *gambler-model* = *infinite-coin-toss-space* +
 fixes *geom-proc*::*int* ⇒ *bool stream* ⇒ *enat* ⇒ *int*
assumes *geometric-process*:*geom-proc init x step* = *gambler-rand-walk 1* (−*1*) *init step x*

begin

1.3 Basic functions

Here we define the all basic functions which will play an invisible role in the further probability analysis. you can just focus on the lemmas and functions where we comment

definition *reach-steps*::*int* ⇒ *bool stream* ⇒ *int* ⇒ *nat set***where**
reach-steps init x target = {*step*::*nat. geom-proc init x step* ∈ {*0*,*target*}}

reach_steps describes all the steps where the input random walk reaches the threshold 0, target

fun *infm*::*nat set* ⇒ *enat* **where**
infm A = (*if A* = {} *then* ∞ *else* $\bigsqcap$ *A*)

infm A = iff A =

lemma *only-inf-infm*:
  assumes*A* ≠ {}
  shows*infm A* ≠ ∞
proof
  assume *infm A* = ∞
  then have *l*:∀ *a*∈*A*. ∞ ≤ *a*
    using *infm.simps*[*of A*] *assms*
    by *auto*
  then have *r*:∀ *a*∈*A*. *a* < ∞
    using *not-enat-eq enat-ord-simps not-infinity-eq*
    by *auto*
  from *l r* **show** *False*
    using *not-enat-eq enat-ord-simps not-infinity-eq*
        assms equals0I
    by *fastforce*
qed

fun *stop-at*::*int* ⇒ *bool stream* ⇒ *int* ⇒ *enat* **where**
stop-at init x target = (*infm* (*reach-steps init x target*))

stop_at describes the first step in the *reach_steps* sets, which means exactly the stopping point in gambler's ruin problem. Be careful, Here the type of output has been extended to *enat*, which means stopping point will be ∞(equivalent to non-existence)

fun *success*::*int* ⇒ *bool stream* ⇒ *int* ⇒ *bool***where**
success init x target = (*geom-proc init x* (*stop-at init x target*) = *target*)

success describes the random walk reaching the target number rather than ruining at stopping point

1.4 Important intermediate conclusions

1.4.1 Successful random walks never stop at ∞

Once we set target to be positive, the weird situation where random walk succeeds at ∞ will disappear

lemma *exist*:
  fixes *init*::*int* **and** *x* **and** *target*::*int*
  assumes $0 \leq init$ $init \leq target$*success init x target*
  shows *stop-at init x target* $\neq \infty$
proof
  assume *stop-at init x target* $= \infty$
  from *this* **have** *geom-proc init x* (*stop-at init x target*) $= -1$
    using *geometric-process* **by** *auto*
  from *this* **show** *False*
    using *assms* **by** *force*
qed

1.4.2 The way we count never change the amount got through specific random walk

lemma *pre1*:$\bigwedge x\ n.$ *snth x* $(n+1) =$ *snth* (*stl x*) *n*
  using *snth.simps*[*of x*] **by** *auto*

lemma additional1 states that the reaching number doesn't change if we want to calculate from the second step

lemma *additional1*:*let init'* = *geom-proc init x 1 in*
geom-proc init' (*stl x*) *n* = *geom-proc init x* (*Suc n*)
proof (*induction n*)
  show *let init'* = *geom-proc init x 1*
    in geom-proc init' (*stl x*) (*enat 0*) = *geom-proc init x* (*enat* (*Suc 0*))
  proof–
    have *let init'* = *geom-proc init x 1 in*
        geom-proc init' (*stl x*) (*enat 0*) = *init'*
     using *geometric-process gambler-rand-walk.simps gambler-rand-walk-pre.simps*
      by *auto*
    from *this* **show** *let init'* = *geom-proc init x 1*
    in geom-proc init' (*stl x*) (*enat 0*) = *geom-proc init x* (*enat* (*Suc 0*))
      by (*metis One-nat-def one-enat-def*)
  qed
next
  fix *n*
  assume *ams*:*let init'* = *geom-proc init x 1*
        in geom-proc init' (*stl x*) (*enat n*) =
          geom-proc init x (*enat* (*Suc n*))
  have *ppp1*:$\bigwedge$ *init1 x1 n. geom-proc init1 x1* (*Suc n*) = *geom-proc init1 x1 n* +
(*case snth x1 n of True* $\Rightarrow$ *1*| *False* $\Rightarrow -1$)
    using *geometric-process gambler-rand-walk.simps gambler-rand-walk-pre.simps*
    by *auto*

from *ams* **show** *let init′ = geom-proc init x 1*
in geom-proc init′ (stl x) (enat (Suc n)) =
geom-proc init x (enat (Suc (Suc n)))
using *ppp1[of init x Suc n]*
ppp1[of geom-proc init x 1 stl x n]
pre1[of x Suc n]
by *auto*
qed

1.4.3 The way we count never change whether the random walk succeeds

lemma *set-up-Inf*:
fixes *A* **and** *a::nat*
assumes $\bigwedge b::nat.\ b \in A \Longrightarrow a \leq b$ $a \in A$
shows $a = Inf\ A$
using *assms(1) assms(2) cInf-eq-minimum*
by *blast*

lemma *Inf-property*:
fixes *a* **and** *A*
assumes $a = Inf\ A$
shows $\bigwedge b::nat.\ b \in A \Longrightarrow a \leq b$
proof−
have*bdd-below A*
using *bdd-below-def[of A]*
by *auto*
then show $\bigwedge b.\ b \in A \Longrightarrow a \leq b$
using *cInf-lower[of - A] assms*
by *auto*
qed

*conditional*2*_pre* states that stopping point doesn't change if we calculate from second step

lemma *conditional2-pre*:
fixes *init′* **and** *Ar* **and** *Al*
assumes *init′ = geom-proc init x 1*
Ar = reach-steps init x target
Al = reach-steps init′ (stl x) target
0 < init
init < target
shows *stop-at init′ (stl x) target + 1 = stop-at init x target*

proof (*cases Ar*= {})
assume *ar-empty*:*Ar* = {}
then have ¬ (∃ *n::nat. geom-proc init x n* ∈ {*0*,*target*})
using *assms(2) reach-steps-def[of init x target]*
by *auto*
then have ¬ (∃ *m::nat. geom-proc init′ (stl x) m* ∈ {*0*,*target*})

```
      using additional1[of init x] assms(1)
      by auto
    then have al-empty:Al = {}
      using assms(3) reach-steps-def[of init′ stl x]
      by auto
    from al-empty have al-inf:stop-at init′ (stl x) target = ∞
      using stop-at.simps assms
      by auto
    from ar-empty have ar-inf:stop-at init x target = ∞
      using assms stop-at.simps
      by auto
    from al-inf ar-inf havestop-at init′ (stl x) target + 1 = stop-at init x target
      using plus-enat-simps
      by auto
    then show ?thesis by auto
  next
    assume ar-nonempty:Ar ≠ {}
    obtain a::nat where a-def:a = stop-at init x target
      unfolding stop-at.simps
      using only-inf-infm[of Ar] ar-nonempty assms(2) not-infinity-eq[of infm Ar]
      by (auto simp add: not-infinity-eq)
    have a ≠ 0
    proof
      assume a = 0
      then have 0 ∈ reach-steps init x target
        using a-def stop-at.simps[of init x target]
              infm.simps[of reach-steps init x target]
              ar-nonempty
              assms(2)
              enat-0-iff[of a]
              enat.inject[of a ⨅ reach-steps init x target]
              Inf-nat-def1[of Ar]
        by auto
      then have geom-proc init x 0 ∈ {0,target}
        using reach-steps-def[of init x target]
        by (simp add: zero-enat-def)
      then have init ∈ {0,target}
        using geometric-process[of init x 0]
              gambler-rand-walk.simps[of 1 −1 init 0 x]
        by (simp add: zero-enat-def)
      then have False
        using assms by auto
      then show False
        by auto
    qed
    obtain a′::nat where a′ + 1 = a
        using ‹a ≠ 0›
          by (metis (no-types) add.commute add.left-neutral add-Suc-right assms
less-imp-Suc-add less-one not-less-eq not-less-less-Suc-eq)
```

```
then have a′ ∈ reach-steps init′ (stl x) target
proof(unfold reach-steps-def)
  assume a′ + 1 = a
  then have geom-proc init′ (stl x) a′ ∈ {0, target}
    using additional1[of init x a′]
          a-def
          stop-at.simps[of init x target]
          infm.simps[of reach-steps init x target]
          Inf-nat-def1[of reach-steps init x target]
          assms(2)
          ar-nonempty
          enat.inject[of a ⨅ reach-steps init x target]
          reach-steps-def[of init x target]
 by (metis (no-types, lifting) add.commute assms(1) mem-Collect-eq plus-1-eq-Suc)
  then show a′ ∈ {xa. geom-proc init′ (stl x) (enat xa) ∈ {0, target}}
    using ‹a′ + 1 = a› by force
qed
then have reach-steps init′ (stl x) target ≠ {}
  by auto
have⋀a b A::nat set. b ∈ A ⟹ a = Inf A ⟹ a ≤ b
proof–
  fix a::nat and b::nat and A
  assume b ∈ A and a = Inf A
  have bdd-below A
    using bdd-below-def[of A]
    by auto
  then show a ≤ b
    using cInf-lower[of b A] ‹a = Inf A› ‹b ∈ A›
    by auto
  qed
have ⋀b. b ∈ reach-steps init′ (stl x) target ⟹ a′ ≤ b
proof(unfold reach-steps-def)
  fix b
  assume b ∈ {xa. geom-proc init′ (stl x) (enat xa) ∈ {0, target}}
  then have geom-proc init′ (stl x) b ∈ {0,target}
    by auto
  then have geom-proc init x (b+1) ∈ {0,target}
    using additional1[of init x b] assms(1)
    by auto
  then have (b + 1) ∈ reach-steps init x target
    using reach-steps-def[of init x target]
    by auto
  then have a ≤ (b + 1)
    using a-def
          stop-at.simps[of init x target]
          infm.simps[of reach-steps init x target]
          ‹Ar ≠ {}›
          assms(2)
          enat.inject[of a ⨅ reach-steps init x target]
```

$\langle\bigwedge a\ b\ A::nat\ set.\ b \in A \Longrightarrow a = Inf\ A \Longrightarrow a \leq b\rangle$
by *auto*
then show $a' \leq b$
using $\langle a' + 1 = a\rangle$
by *force*
qed
then have $a' = \bigsqcap\ (reach\text{-}steps\ init'\ (stl\ x)\ target)$
unfolding *reach-steps-def*
using $cInf\text{-}eq\text{-}minimum[of\ a'\ \{xa.\ geom\text{-}proc\ init'\ (stl\ x)\ (enat\ xa) \in \{0, target\}\}]$
$\langle a' \in reach\text{-}steps\ init'\ (stl\ x)\ target\rangle$
$reach\text{-}steps\text{-}def[of\ init'\ stl\ x\ target]$
by *auto*
then have $a' = stop\text{-}at\ init'\ (stl\ x)\ target$
unfolding *stop-at.simps*
using $\langle reach\text{-}steps\ init'\ (stl\ x)\ target \neq \{\}\rangle$
by *auto*
then show $stop\text{-}at\ init'\ (stl\ x)\ target + 1 = stop\text{-}at\ init\ x\ target$
using $\langle a = stop\text{-}at\ init\ x\ target\rangle$
$\langle a' + 1 = a\rangle$
by $(metis\ one\text{-}enat\text{-}def\ plus\text{-}enat\text{-}simps(1))$
qed

conditional2 states that whether a random walk succeeds or not doesn't change if we calculate from second step

lemma *conditional2*:
fixes *init x target*
assumes $init' = geom\text{-}proc\ init\ x\ 1$
$0 < init$
$init < target$
shows $success\ init'\ (stl\ x)\ target \longleftrightarrow success\ init\ x\ target$
proof
obtain *ar* **where** $ar = reach\text{-}steps\ init\ x\ target$
by *blast*
obtain *al* **where** $al = reach\text{-}steps\ init'\ (stl\ x)\ target$
by *blast*
assume $lhs{:}success\ init'\ (stl\ x)\ target$
then have $lhs1{:}geom\text{-}proc\ init'\ (stl\ x)\ (stop\text{-}at\ init'\ (stl\ x)\ target) = target$
using *success.simps*
by *auto*
then have $stop\text{-}at\ init'\ (stl\ x)\ target \neq \infty$
using $exist[of\ init'\ target\ stl\ x]$
$success.simps[of\ init'\ stl\ x\ target]$
assms
$geometric\text{-}process[of\ init\ x\ 1]$
$gambler\text{-}rand\text{-}walk.simps[of\ 1\ {-1}\ init\ 1\ x]$
gambler-rand-walk-pre.simps
using *geometric-process* **by** *auto*
then obtain $a::nat$ **where** $a = stop\text{-}at\ init'\ (stl\ x)\ target$

using *not-infinity-eq*
by *auto*
with *lhs1* **have** *geom-proc init x* (*stop-at init x target*) = *target*
using *conditional2-pre*[*of init′ init x ar target al*]
assms
‹*ar* = *reach-steps init x target*›
‹*al* = *reach-steps init′* (*stl x*) *target*›
additional1[*of init x a*]
by (*metis Suc-eq-plus1 one-enat-def plus-enat-simps*(*1*))
then show *success init x target*
using *success.simps*
by *auto*
next
obtain *ar* **where** *ar* = *reach-steps init x target*
by *blast*
obtain *al* **where** *al* = *reach-steps init′* (*stl x*) *target*
by *blast*
assume *rhs*:*success init x target*
then have *rhs1*:*geom-proc init x* (*stop-at init x target*) = *target*
using *success.simps*
by *auto*
then have *stop-at init x target* ≠ ∞
using *exist*[*of init target x*]
success.simps[*of init x target*]
assms
geometric-process[*of init x 1*]
gambler-rand-walk.simps[*of 1 −1 init 1 x*]
gambler-rand-walk-pre.simps
using *geometric-process*
by *auto*
then obtain *a′*::*nat* **where** *a′* = *stop-at init x target*
using *not-infinity-eq*
by *auto*
have *a′* ≠ *0*
proof
assume *a′* = *0*
then have *geom-proc init x a′* = *init*
by (*metis* ‹*enat a′* = *stop-at init x target*› *add.commute add.right-neutral assms*(*2*) *assms*(*3*) *conditional2-pre enat-add-left-cancel-less gr-implies-not-zero zero-enat-def zero-less-one*)
then show *False*
using *rhs1*
assms
‹*a′* = *stop-at init x target*›
by *auto*
qed
then obtain *a*::*nat* **where** *a* + *1* = *a′*
by (*metis Suc-eq-plus1 old.nat.exhaust*)
with *rhs1* **have** *geom-proc init′* (*stl x*) (*stop-at init′* (*stl x*) *target*) = *target*

proof−
have *stop-at init′* (*stl x*) *target* + *1* = *stop-at init x target*
using *conditional2-pre*[*of init′ init x ar target al*]
assms
‹*ar* = *reach-steps init x target*›
‹*al* = *reach-steps init′* (*stl x*) *target*›
by *auto*
then have *a* = *stop-at init′* (*stl x*) *target*
using ‹*a′* = *stop-at init x target*›
‹*a* + *1* = *a′*›
eSuc-enat[*of a*]
by (*metis Suc-eq-plus1 eSuc-inject plus-1-eSuc*(*2*))
then have *geom-proc init′* (*stl x*) (*stop-at init′* (*stl x*) *target*) = *geom-proc init x* (*stop-at init x target*)
using *additional1*[*of init x a*]
by (*metis Suc-eq-plus1* ‹*a* + *1* = *a′*› ‹*enat a′* = *stop-at init x target*› *assms*(*1*))

with *rhs1* **show** *geom-proc init′* (*stl x*) (*stop-at init′* (*stl x*) *target*) = *target*
by *auto*
qed
then show *success init′* (*stl x*) *target*
using *success.simps*
by *auto*
qed

1.4.4 The change of initial number

if first step is true, then we add 1 to initial number

lemma *fst-true-plus-one*:
fixes *init x target*
assumes *init′* = *geom-proc init x 1shd x* = *True*
shows *init′* = *init* + *1*
proof−
have *int1*:*gambler-rand-walk 1* (− *1*) *init 1 x* = *1* + *gambler-rand-walk-pre 1* (− *1*) *init 0 x*
using *snth.simps*(*1*)[*of x*]
gambler-rand-walk-pre.simps(*1*)[*of 1* −*1 init x*]
gambler-rand-walk-pre.simps(*2*)[*of 1* −*1 init 0 x*]
gambler-rand-walk.simps[*of 1* −*1 init 1 x*]
one-enat-def zero-enat-def
gambler-rand-walk.simps[*of 1* −*1 init 0 x*]
assms
by (*simp add*: *one-enat-def*)
have *int2*:*gambler-rand-walk-pre 1* (− *1*) *init 0 x* = *init*
using *gambler-rand-walk-pre.simps*(*1*)[*of 1* −*1 init x*]
by *auto*
have *int3*:*gambler-rand-walk 1* (− *1*) *init 1 x* = *init′*
using *geometric-process*[*of init x 1*]
assms(*1*)

gambler-rand-walk.simps[*of* 1 -1 *init* 1 *x*]
by *auto*
show *init'* = *init* + 1
using *int1 int2 int3*
by *auto*
qed

if first step is False, then we reduce 1 to initial number

lemma *fst-true-plus-one-false*:
fixes *init x target*
assumes *init'* = *geom-proc init x 1shd x* = *False*
shows *init'* = *init* − 1
proof−
have *int1*:*gambler-rand-walk* 1 ($-$ 1) *init* 1 *x* = (*gambler-rand-walk-pre* 1 ($-$ 1) *init* 0 *x*) − 1
using *snth.simps*(1)[*of x*]
gambler-rand-walk-pre.simps(1)[*of* 1 -1 *init x*]
gambler-rand-walk-pre.simps(2)[*of* 1 -1 *init* 0 *x*]
gambler-rand-walk.simps[*of* 1 -1 *init* 1 *x*]
one-enat-def zero-enat-def
gambler-rand-walk.simps[*of* 1 -1 *init* 0 *x*]
assms
by (*simp add*: *one-enat-def*)
have *int2*:*gambler-rand-walk-pre* 1 ($-$ 1) *init* 0 *x* = *init*
using *gambler-rand-walk-pre.simps*(1)[*of* 1 -1 *init x*]
by *auto*
have *int3*:*gambler-rand-walk* 1 ($-$ 1) *init* 1 *x* = *init'*
using *geometric-process*[*of init x* 1]
assms(1)
gambler-rand-walk.simps[*of* 1 -1 *init* 1 *x*]
by *auto*
show *init'* = *init* − 1
using *int1 int2 int3*
by *auto*
qed

1.4.5 The way we count never change the successful random walk set

the set where all random walks in it succeeds and their first step are True doesn't change if we calculate from second step

lemma *conditional-set-equation*:
fixes *init target*
assumes
$0 < init$
$init < target$
shows
$\{x::bool\ stream.\ success\ init\ x\ target \wedge shd\ x = True\} =$

$\{x{::}\mathit{bool\ stream}.\ \mathit{success}\ (\mathit{init}{+}1)\ (\mathit{stl}\ x)\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}\}$

proof

  show $\{x.\ \mathit{success}\ \mathit{init}\ x\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}\}$

    $\subseteq \{x.\ \mathit{success}\ (\mathit{init} + 1)\ (\mathit{stl}\ x)\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}\}$

  proof

    fix x

    assume $x \in \{x.\ \mathit{success}\ \mathit{init}\ x\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}\}$

    then have $\mathit{success}\ \mathit{init}\ x\ \mathit{target}\ \mathit{shd}\ x = \mathit{True}$

      by *auto*

    obtain init' **where** $\mathit{init}' = \mathit{geom\text{-}proc}\ \mathit{init}\ x\ 1$

      by *blast*

    with $\langle \mathit{shd}\ x = \mathit{True} \rangle$ **have** $\mathit{init}' = \mathit{init} + 1$

      using $\mathit{fst\text{-}true\text{-}plus\text{-}one}[\mathit{of}\ \mathit{init}'\ \mathit{init}\ x]$

        $\mathit{assms}(1)$

      by *auto*

    then have $\mathit{success}\ (\mathit{init} + 1)\ (\mathit{stl}\ x)\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}$

      using $\mathit{conditional2}[\mathit{of}\ \mathit{init}'\ \mathit{init}\ x\ \mathit{target}]$

        assms

        $\langle \mathit{init}' = \mathit{geom\text{-}proc}\ \mathit{init}\ x\ 1 \rangle$

        $\langle \mathit{shd}\ x = \mathit{True} \rangle$

        $\langle \mathit{success}\ \mathit{init}\ x\ \mathit{target} \rangle$

      by *auto*

    then show $x \in \{x.\ \mathit{success}\ (\mathit{init} + 1)\ (\mathit{stl}\ x)\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}\}$

      by *auto*

  qed

next

  show $\{x.\ \mathit{success}\ (\mathit{init} + 1)\ (\mathit{stl}\ x)\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}\}$

    $\subseteq \{x.\ \mathit{success}\ \mathit{init}\ x\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}\}$

  proof

    fix x

    assume $x \in \{x.\ \mathit{success}\ (\mathit{init} + 1)\ (\mathit{stl}\ x)\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}\}$

    then have $\mathit{success}\ (\mathit{init} + 1)\ (\mathit{stl}\ x)\ \mathit{targetshd}\ x = \mathit{True}$

      by *auto*

    obtain init' **where** $\mathit{init}' = \mathit{geom\text{-}proc}\ \mathit{init}\ x\ 1$

      by *blast*

    with $\langle \mathit{shd}\ x = \mathit{True} \rangle$ **have** $\mathit{init}' = \mathit{init} + 1$

      using $\mathit{fst\text{-}true\text{-}plus\text{-}one}[\mathit{of}\ \mathit{init}'\ \mathit{init}\ x]$

        $\mathit{assms}(1)$

      by *auto*

    then have $\mathit{success}\ \mathit{init}\ x\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}$

      using $\mathit{conditional2}[\mathit{of}\ \mathit{init}'\ \mathit{init}\ x\ \mathit{target}]$

        assms

        $\langle \mathit{init}' = \mathit{geom\text{-}proc}\ \mathit{init}\ x\ 1 \rangle$

        $\langle \mathit{shd}\ x = \mathit{True} \rangle$

        $\langle \mathit{success}\ (\mathit{init} + 1)\ (\mathit{stl}\ x)\ \mathit{target} \rangle$

      by *auto*

    then show $x \in \{x.\ \mathit{success}\ \mathit{init}\ x\ \mathit{target} \land \mathit{shd}\ x = \mathit{True}\}$

      by *auto*

  qed

qed

the set where all random walks in it succeeds and their first step are False doesn't change if we calculate from second step

lemma *conditional-set-equation-false*:
  fixes *init* *target*
  assumes
    $0 < init$
    $init < target$
  shows
$\{x::bool\ stream.\ success\ init\ x\ target \wedge shd\ x = False\} =$
$\{x::bool\ stream.\ success\ (init-1)\ (stl\ x)\ target \wedge shd\ x = False\}$
proof
  show $\{x.\ success\ init\ x\ target \wedge shd\ x = False\}$
    $\subseteq \{x.\ success\ (init - 1)\ (stl\ x)\ target \wedge shd\ x = False\}$
  proof
    fix x
    assume $x \in \{x.\ success\ init\ x\ target \wedge shd\ x = False\}$
    then have *success init x target shd x = False*
      by *auto*
    obtain $init'$ **where** $init' = geom\text{-}proc\ init\ x\ 1$
      by *blast*
    with ‹*shd x = False*› **have** $init' = init - 1$
      using *fst-true-plus-one-false*[*of init' init x*]
        assms(*1*)
      by *auto*
    then have *success* (*init* − *1*) (*stl x*) *target* ∧ *shd x* = *False*
      using *conditional2*[*of init' init x target*]
        assms
        ‹*init' = geom-proc init x 1*›
        ‹*shd x = False*›
        ‹*success init x target*›
      by *auto*
    then show $x \in \{x.\ success\ (init - 1)\ (stl\ x)\ target \wedge shd\ x = False\}$
      by *auto*
  qed
next
  show $\{x.\ success\ (init - 1)\ (stl\ x)\ target \wedge shd\ x = False\}$
    $\subseteq \{x.\ success\ init\ x\ target \wedge shd\ x = False\}$
  proof
    fix x
    assume $x \in \{x.\ success\ (init - 1)\ (stl\ x)\ target \wedge shd\ x = False\}$
    then have *success* (*init* − *1*) (*stl x*) *targetshd x* = *False*
      by *auto*
    obtain $init'$ **where** $init' = geom\text{-}proc\ init\ x\ 1$
      by *blast*
    with ‹*shd x = False*› **have** $init' = init - 1$
      using *fst-true-plus-one-false*[*of init' init x*]
        assms(*1*)

by *auto*
then have *success init x target* $\wedge$ *shd x* = *False*
using *conditional2*[*of init′ init x target*]
assms
‹*init′* = *geom-proc init x 1*›
‹*shd x* = *False*›
‹*success* (*init* − *1*) (*stl x*) *target*›
by *auto*
then show $x \in \{x.$ *success init x target* $\wedge$ *shd x* = *False*$\}$
by *auto*
qed
qed

1.5 Probability equation

Here we start to analyse the probability of successful random walk. To better understand this part please have a look the elaboration in front of lemma *success_measurable*

probability_of_win is the function describing possibility of successful random walks with initial number and target number as inputs

fun *probability-of-win*::*int* ⇒ *int* ⇒ *ennreal***where**
probability-of-win init target = *emeasure M* {*x*∈ *space M*. *success init x target*}

1.5.1 Successful random walk set is measurable

Preimage of function snth is measurable

lemma *snth-measurable*:
fixes *n*::*nat*
shows⋀*k*. (λ*w*. *snth w n*) −‘ {*k*} ∈ *sets M*
proof−
fix *k*
have (λ*w*. *snth w n*) ∈ *measurable M* (*measure-pmf* (*bernoulli-pmf p*))
using *bernoulli p-gt-0 p-lt-1*
by (*simp add*: *bernoulli-stream-def*)
moreover have {*k*} ∈ *sets* (*measure-pmf* (*bernoulli-pmf p*))
by *simp*
ultimately show (λ*w*. *snth w n*) −‘ {*k*} ∈ *sets M*
using *measurable-sets*[*of* λ*w*. *snth w n M measure-pmf* (*bernoulli-pmf p*) {*k*}]
bernoulli-stream-preimage[*of M p* λ*w*. *snth w n* {*k*}]
bernoulli
by *force*
qed

lemma *stake-measurable-pre1*:
fixes *n w k*
assumes *length k* > *n*

shows *stake* (*Suc n*) *w* = *take* (*Suc n*) *k* $\longleftrightarrow$ *stake n w* = *take n k* $\wedge$ *snth w n* = *nth k n*
proof
assume *stake* (*Suc n*) *w* = *take* (*Suc n*) *k*
then show *stake n w* = *take n k* $\wedge$ *w* !! *n* = *k* ! *n*
using *take-hd-drop*[*of n k*]
stake-Suc[*of n w*]
assms
append-eq-append-conv[*of stake n w take n k*[*snth w n*][*nth k n*]]
length-take[*of n k*]
length-stake[*of n w*]
Lattices.linorder-class.min.absorb-iff2[*of n length k*]
by (*metis append1-eq-conv hd-drop-conv-nth*)
next
assume *stake n w* = *take n k* $\wedge$ *snth w n* = *nth k n*
then show *stake* (*Suc n*) *w* = *take* (*Suc n*) *k*
using *take-hd-drop*[*of n k*]
stake-Suc[*of n w*]
assms
append-eq-append-conv[*of stake n w take n k*[*snth w n*][*nth k n*]]
length-take[*of n k*]
length-stake[*of n w*]
Lattices.linorder-class.min.absorb-iff2[*of n length k*]
by (*metis take-Suc-conv-app-nth*)
qed

lemma *stake-measurable-pre*:
fixes *n*
shows $\bigwedge$*k. length k* $\geq$ *n* $\Longrightarrow$ (*stake n* −' {*k*}) $\in$ *sets M*
proof(*induction n*)
fix *k*
show (*stake 0* −' {*k*}) $\in$ *sets M*
proof (*cases k*)
assume *k* = []
have $\forall$ *w. stake 0 w* = []
by *auto*
then have (*stake 0* −' {*k*}) = *space M*
using *bernoulli-stream-space*[*of M p*]
bernoulli
‹*k* = []›
by *fastforce*
then show (*stake 0* −' {*k*}) $\in$ *sets M*
by *auto*
next
fix *a list*
assume *k* = *a* # *list*
then have *k* $\neq$ []

```
        by auto
      have ∀ w. stake 0 w = []
        by auto
      then have (stake 0 −‘ {k}) = {}
        using ‹k ≠ []›
        by force
      then show (stake 0 −‘ {k}) ∈ sets M
        by auto
    qed
next
  fix n k
  assume ⋀k1. n ≤ length k1 ⟹ (stake n −‘ {k1}) ∈ sets M
  thus Suc n ≤ length k ⟹ stake (Suc n) −‘ {k} ∈ events
  proof(cases Suc n = length k)
    assume Suc n ≤ length kSuc n = length k
    have (stake (Suc n) −‘ {take (Suc n) k}) = (stake n −‘ {take n k}) ∩ ((λw.
snth w n) −‘ {nth k n})
    proof
        show stake (Suc n) −‘ {take (Suc n) k}
      ⊆ stake n −‘ {take n k} ∩ (λw. w !! n) −‘ {k ! n}
        proof
          fix w
          assume w ∈ stake (Suc n) −‘ {take (Suc n) k}
          then have stake (Suc n) w = take (Suc n) k
            by auto
          then have stake n w = take n k
            using stake-measurable-pre1[of n k w]
                ‹Suc n ≤ length k›
            by auto
          have snth w n = nth k n
            using stake-measurable-pre1[of n k w]
                ‹Suc n ≤ length k›
                ‹stake (Suc n) w = take (Suc n) k›
            by auto
          show w ∈ (stake n −‘ {take n k}) ∩ ((λw. snth w n) −‘ {nth k n})
            using ‹snth w n = nth k n›
                ‹stake n w = take n k›
            by auto
        qed
      next
        show stake n −‘ {take n k} ∩ (λw. w !! n) −‘ {k ! n}
      ⊆ stake (Suc n) −‘ {take (Suc n) k}
        proof
          fix w
          assume w ∈ (stake n −‘ {take n k}) ∩ ((λw. snth w n) −‘ {nth k n})
          then have snth w n = nth k nstake n w = take n k
            by auto
          then have stake (Suc n) w = take (Suc n) k
            using stake-measurable-pre1[of n k w]
```

```
          ‹Suc n ≤ length k›
        by auto
      then show w ∈ stake (Suc n) −‘ {take (Suc n) k}
        by auto
    qed
  qed
  moreover have take (Suc n) k = k
    using take-all[of k Suc n]
        ‹Suc n = length k›
    by auto
  moreover have stake n −‘ {take n k} ∈ sets M
    using ‹⋀k1. n ≤ length k1 ⟹ (stake n −‘ {k1}) ∈ sets M›
        ‹Suc n = length k›
    by auto
  moreover have ((λw. snth w n) −‘ {nth k n}) ∈ sets M
    using snth-measurable
    by auto
  ultimately show stake (Suc n) −‘ {k} ∈ events
    by simp
next
  assume Suc n ≤ length kSuc n ≠ length k
  then have Suc n < length k
    by auto
  then have stake (Suc n) −‘ {k} = {}
  proof−
    have ⋀k1 . length k1 < length k ⟹ k1 ≠ k
      by auto
    then have ⋀w. stake (Suc n) w ≠ k
      using length-stake[of Suc n w]
          ‹Suc n < length k›
      by auto
    then show stake (Suc n) −‘ {k} = {}
      by force
  qed
  then show stake (Suc n) −‘ {k} ∈ events
    by auto
qed
qed
```

The preimage of any list over function stake is measurable

```
lemma stake-measurable:
  fixes n k
  shows(stake n −‘ {k}) ∈ sets M
proof (cases length k ≥ n)
  assume length k ≥ n
  then show (stake n −‘ {k}) ∈ sets M
    using stake-measurable-pre[of n k]
    by auto
next
```

assume $\neg\ n \leq$ *length k*
then have *length k* $<$ *n*
by *auto*
then have *stake n* $-$‘ $\{k\} = \{\}$
proof–
have $\bigwedge k1$. *length k1* $>$ *length k* $\Longrightarrow$ *k1* $\neq$ *k*
by *auto*
then have $\bigwedge w$. *stake n w* $\neq$ *k*
using *length-stake*[*of n w*]
⟨*length k* $<$ *n*⟩
by *auto*
then show *stake n* $-$‘ $\{k\} = \{\}$
by *force*
qed
then show (*stake n* $-$‘ $\{k\}$) $\in$ *sets M*
using *UN-empty2*[*of l*]
by *auto*
qed

The preimage of any list set over function stake is measurable once the set is finite

lemma *finite-stake-measurable*:
fixes *A* **and** *n*::*nat*
assumes*finite A*
shows(*stake n* $-$‘ *A*) $\in$ *sets M*
proof–
have ($\bigcup x{\in}A$. (*stake n* $-$‘ $\{x\}$)) $=$ (*stake n* $-$‘ *A*)
by *auto*
then show (*stake n* $-$‘ *A*) $\in$ *sets M*
using *stake-measurable*
assms
by (*metis sets.finite-UN*)
qed

The new *geom_proc* function for list

fun *geom-proc-list*::*int* $\Rightarrow$ *bool list* $\Rightarrow$ *int***where**
geom-proc-list init [] $=$ *init*|
geom-proc-list init (*x* # *xs*) $=$ (*case x of True*$\Rightarrow$*1*|*False* $\Rightarrow$ $-$*1*) $+$ *geom-proc-list init xs*

lemma *reverse-construct-pre*:
fixes *init lengthx y*
shows $\bigwedge x$::*bool list*. *lengthx* $=$ *length x* $\Longrightarrow$ *geom-proc-list init* (*x* @ [*y*]) $=$ *geom-proc-list init x* $+$ (*case y of True*$\Rightarrow$*1*|*False*$\Rightarrow$ $-$*1*)
proof(*induction lengthx*)
show $\bigwedge x$. *0* $=$ *length x* $\Longrightarrow$
geom-proc-list init (*x* @ [*y*]) $=$
geom-proc-list init x $+$

(*case y of True* ⇒ *1* | *False* ⇒ − *1*)
by *force*
next
fix *lengthx x*
assume (⋀*x1. lengthx* = *length x1* ⟹
geom-proc-list init (*x1* @ [*y*]) =
geom-proc-list init x1 +
(*case y of True* ⇒ *1* | *False* ⇒ − *1*))
then show *Suc lengthx* = *length x* ⟹
geom-proc-list init (*x* @ [*y*]) =
geom-proc-list init x +
(*case y of True* ⇒ *1* | *False* ⇒ − *1*)
proof−
assume*Suc lengthx* = *length x*
⋀*x1. lengthx* = *length x1* ⟹
geom-proc-list init (*x1* @ [*y*]) =
geom-proc-list init x1 +
(*case y of True* ⇒ *1* | *False* ⇒ − *1*)
have *geom-proc-list init* (*x* @ [*y*]) = *geom-proc-list init* (*hd x* # (*tl x* @ [*y*]))
by (*smt* (*verit*, *del-insts*) *Nil-is-append-conv* ‹*Suc lengthx* = *length x*› *hd-Cons-tl hd-append2 length-Suc-conv list.discI tl-append2*)
moreover have *geom-proc-list init* (*hd x* # (*tl x* @ [*y*])) = (*case hd x of True*⇒*1*|*False*⇒ −*1*) + *geom-proc-list init* (*tl x* @ [*y*])
using *geom-proc-list.simps*
by *auto*
moreover have *geom-proc-list init* (*tl x* @ [*y*]) = *geom-proc-list init* (*tl x*) +
(*case y of True* ⇒ *1* | *False* ⇒ − *1*)
using ‹*Suc lengthx* = *length x*› ‹⋀*x1. lengthx* = *length x1* ⟹ *geom-proc-list init* (*x1* @ [*y*]) = *geom-proc-list init x1* + (*case y of True* ⇒ *1* | *False* ⇒ − *1*)›
by *fastforce*
moreover have *geom-proc-list init* (*tl x*) + (*case hd x of True*⇒*1*|*False*⇒ −*1*) = *geom-proc-list init x*
by (*metis* ‹*Suc lengthx* = *length x*› *add.commute geom-proc-list.simps*(*2*) *hd-Cons-tl length-Suc-conv list.discI*)
ultimately show *geom-proc-list init* (*x* @ [*y*]) =
geom-proc-list init x +
(*case y of True* ⇒ *1* | *False* ⇒ − *1*)
by *auto*
qed
qed

lemma *reverse-construct*:
fixes *init x y*
shows *geom-proc-list init* (*x* @ [*y*]) = *geom-proc-list init x* + (*case y of True*⇒*1*|*False*⇒ −*1*)
using *reverse-construct-pre*[*of length x x init y*]
by *auto*

lemma *success-pre*:

fixes *init x target i*
assumes $0 <$ *initinit* $<$ *target*
shows *geom-proc-list init* (*stake i x*) = *geom-proc init x i*
proof(*induction i*)
show *geom-proc-list init* (*stake 0 x*) = *geom-proc init x* (*enat 0*)
using *stake.simps*(*1*)[*of x*]
geom-proc-list.simps(*1*)[*of init*]
geometric-process[*of init x enat 0*]
gambler-rand-walk.simps(*1*)[*of 1* -1 *init enat 0 x*]
gambler-rand-walk-pre.simps(*1*)[*of 1* -1 *init x*]
enat-0
enat.simps(*4*)[*of* λn. *gambler-rand-walk-pre 1* ($-$ *1*) *init n x* -1 *0*]
by *auto*
next
fix *i*
assume *geom-proc-list init* (*stake i x*) = *geom-proc init x i*
have *geom-proc-list init* (*stake* (*Suc i*) *x*) = *geom-proc-list init* (*stake i x* @ [*x* !! *i*])
using *stake-Suc*[*of i x*]
by *auto*
moreover have *geom-proc-list init* (*stake i x* @ [*x* !! *i*]) = *geom-proc-list init* (*stake i x*) + (*case x* !! *i of True*⇒*1*|*False*⇒ -1)
using *reverse-construct*[*of init stake i x x* !! *i*]
by *auto*
moreover have *geom-proc-list init* (*stake i x*) = *geom-proc init x i*
using ‹*geom-proc-list init* (*stake i x*) = *geom-proc init x i*›
by *auto*
moreover have *geom-proc init x* (*Suc i*) = *geom-proc init x i* + (*case x* !! *i of True*⇒*1*|*False*⇒ -1)
using *geometric-process*
by *auto*
ultimately show *geom-proc-list init* (*stake* (*Suc i*) *x*) = *geom-proc init x* (*Suc i*)
by *auto*
qed

Any natural number smaller than Inf A doesn't belong to A

lemma *not-belong*:
fixes *A* **and** *a*::*nat*
assumes $\bigsqcap$ $A > a$
shows $a \notin A$
proof
assume $a \in A$
then have $a \geq \bigsqcap$ A
using *Inf-property*[*of - A a*]
by *auto*
then show *False*
using *assms*
by *auto*

qed

This is the most important intermediate lemma prepared for lemma *success_measurable.Itclarifiest*

lemma *success-measurable2*:
  fixes *init target* **and** *i*::*nat*
  assumes $0 < init init < target\ 0 \leq i$
  shows$\{x \in space\ M.\ success\ init\ x\ target \wedge stop\text{-}at\ init\ x\ target = i\}$
$= stake\ i -`\ \{c::bool\ list.\ (\forall k<i.\ (geom\text{-}proc\text{-}list\ init\ (take\ k\ c)) \notin \{0,target\}) \wedge$
$length\ c = i \wedge geom\text{-}proc\text{-}list\ init\ c = target\}$
proof
  show $\{x \in space\ M.\ success\ (init)\ x\ (target) \wedge stop\text{-}at\ (init)\ x\ (target) = enat\ i\}$
    $\subseteq stake\ i -`\ \{c.\ (\forall k<i.\ geom\text{-}proc\text{-}list\ (init)\ (take\ k\ c) \notin \{0,\ target\}) \wedge length\ c = i \wedge geom\text{-}proc\text{-}list\ (init)\ c = target\}$
  proof
    fix *x*
    assume $x \in \{x \in space\ M.\ success\ (init)\ x\ (target) \wedge stop\text{-}at\ (init)\ x\ (target) = enat\ i\}$
   then have *lhs1*:$success\ (init)\ x\ (target)$**and** *lhs2*:$stop\text{-}at\ (init)\ x\ (target) = enat\ i$
      by *auto*
    then have $geom\text{-}proc\ init\ x\ i = target$
      by *auto*
    then have $geom\text{-}proc\text{-}list\ init\ (stake\ i\ x) = target$
      using *success-pre*[*of init target i x*]
          assms
      by *auto*
    with *lhs2* **have** $\forall k<i.\ geom\text{-}proc\text{-}list\ (init)\ (take\ k\ (stake\ i\ x)) \notin \{0,\ target\}$
    proof–
      have$\forall k<i.\ take\ k\ (stake\ i\ x) = stake\ k\ x$
        using *take-stake*
        by (*simp add*: *take-stake min.strict-order-iff*)
      then have $\forall k<i.\ geom\text{-}proc\text{-}list\ (init)\ (take\ k\ (stake\ i\ x)) = geom\text{-}proc\text{-}list\ init\ (stake\ k\ x)$
        by *auto*
      then have $\forall k<i.\ geom\text{-}proc\text{-}list\ (init)\ (take\ k\ (stake\ i\ x)) = geom\text{-}proc\ init\ x\ k$
      using *success-pre*[*of init target - x*]
          assms
      by *auto*
    have *nonempty*:$reach\text{-}steps\ (init)\ x$
      $(target) \neq \{\}$
      using ‹$geom\text{-}proc\ init\ x\ i = target$›
          reach-steps-def[*of init x target*]
      by *auto*
    then have $\forall k<i.\ geom\text{-}proc\ init\ x\ k \notin \{0,\ target\}$
      using *lhs2 stop-at.simps*[*of init x target*]
          infm.simps[*of reach-steps init x target*]
          not-belong[*of - reach-steps init x target*]

reach-steps-def[*of init x target*]
by (*metis enat.inject mem-Collect-eq*)
then show $\forall k<i$. *geom-proc-list* (*init*) (*take k* (*stake i x*)) $\notin$ {*0*, *target*}
using ‹$\forall k<i$. *geom-proc init x k* $\notin$ {*0*, *target*}›
‹$\forall k<i$. *geom-proc-list* (*init*) (*take k* (*stake i x*)) = *geom-proc init x k*›
by *auto*
qed
have *length* (*stake i x*) = *i*
by (*simp add*:*length-stake*)
then show $x \in$ *stake i* −‘ {*c*. ($\forall k<i$. *geom-proc-list* (*init*) (*take k c*) $\notin$ {*0*,*target*}) $\wedge$ *length c* = *i* $\wedge$ *geom-proc-list* (*init*) *c* = *target*}
using ‹$\forall k<i$. *geom-proc-list* (*init*) (*take k* (*stake i x*)) $\notin$ {*0*, *target*}›
‹*geom-proc-list init* (*stake i x*) = *target*›
by *auto*
qed
next
show *stake i* −‘ {*c*. ($\forall k<i$. *geom-proc-list* (*init*) (*take k c*) $\notin$ {*0*,*target*}) $\wedge$ *length c* = *i* $\wedge$ *geom-proc-list* (*init*) *c* = *target*}
$\subseteq$ {$x \in$ *space M*. *success* (*init*) *x* (*target*) $\wedge$ *stop-at* (*init*) *x* (*target*) = *enat i*}
proof
fix *x*
assume $x \in$ *stake i* −‘ {*c*. ($\forall k<i$. *geom-proc-list* (*init*) (*take k c*) $\notin$ {*0*, *target*}) $\wedge$ *length c* = *i* $\wedge$ *geom-proc-list* (*init*) *c* = *target*}
then have *rhs1*:$\forall k<i$. *geom-proc-list* (*init*) (*take k* (*stake i x*)) $\notin$ {*0*, *target*}and

rhs2:*length* (*stake i x*) = *i* and
rhs3:*geom-proc-list* (*init*) (*stake i x*) = *target*
by *auto*
from *rhs3* have *geom-proc init x i* = *target*
using *success-pre*[*of init target i x*] *assms*
by *auto*
then have *reach-steps init x target* $\neq$ {}
unfolding *reach-steps-def*
by *auto*
from *rhs1* have $\forall k<i$. *geom-proc init x k* $\notin$ {*0*, *target*}
using *success-pre*[*of init target - x*]
assms
take-stake[*of - i x*]
min.strict-order-iff[*of - i*]
by *force*
then have *stop-at* (*init*) *x* (*target*) = *enat i*
proof
have *stop-at* (*init*) *x* (*target*) = *Inf* (*reach-steps init x target*)
using ‹*reach-steps init x target* $\neq$ {}›
stop-at.simps[*of init x target*]
infm.simps[*of reach-steps init x target*]
by *auto*
moreover have $i \in$ *reach-steps init x target*
using ‹*geom-proc init x i* = *target*›

reach-steps-def[*of init x target*]
by *auto*
moreover have $\forall\, k<i.\ k \notin$ *reach-steps init x target*
using ‹$\forall\, k<i.$ *geom-proc init x k* $\notin \{0,$ *target*$\}$›
reach-steps-def[*of init x target*]
by *auto*
moreover have *Inf* (*reach-steps init x target*) = *i*
using
‹*i* ∈ *reach-steps init x target*›
‹$\forall\, k<i.\ k \notin$ *reach-steps init x target*›
by (*metis le-refl nat-less-le nat-neq-iff set-up-Inf*)
ultimately show *stop-at* (*init*) *x* (*target*) = *enat i*
by *auto*
qed
then have *success init x target*
unfolding *success.simps*
using ‹*geom-proc init x i* = *target*›
by *auto*
then have $\forall\, x$::*bool stream*. *x* ∈ *space M*
using *bernoulli-stream-space*[*of M p*]
bernoulli
by *auto*
then show*x* ∈ {*x* ∈ *space M*. *success* (*init*) *x* (*target*) ∧ *stop-at* (*init*) *x* (*target*) = *enat i*}
using ‹*stop-at* (*init*) *x* (*target*) = *enat i*›
‹*success init x target*›
by *auto*
qed
qed

lemma *stake-space*:*stake n* ‘ *space M* = {*c*::*bool list*. *length c* = *n*}
proof
show *stake n* ‘ *space M* ⊆ {*c*::*bool list*. *length c* = *n*}
proof
fix *x*
assume *x* ∈ *stake n* ‘ *space M*
then show *x* ∈ {*c*::*bool list*. *length c* = *n*}
using *length-stake*
by *force*
qed
next
show{*c*::*bool list*. *length c* = *n*} ⊆ *stake n* ‘ *space M*
proof
fix *x*
assume*x* ∈ {*c*::*bool list*. *length c* = *n*}
then have *length x* = *n*
by *auto*
obtain *k* **where** *shd k* = *True*
by (*metis stream.sel*(*1*))

```
    then obtain k1 wherek1 = x @- k
      by blast
    then have stake n k1 = x
      using stake-shift[of n x k]
      ‹length x = n›
      take-all[of x n]
      stake.simps(1)[of k]
      by auto
    then obtain k2 where stake n k2 = x
      using length-stake[of n -]
      by auto
    then have k2 ∈ space M
      using bernoulli
            bernoulli-stream-space[of M p]
      by auto
    then show  x ∈ (stake n ‘ space M)
      using ‹stake n k2 = x›
      by auto
  qed
qed
```

Set of all the lists with specific length is finite

```
lemma finite-length:finite {c::bool list. length c = n}
proof–
  let ?U = UNIV::bool set
  have ?U = {True, False}
    by auto
  hence finite ?U
    by simp
  moreover have ?U ≠ {}
    by auto
  ultimately have fi: finite (stake n ‘streams ?U)
    using stake-finite-universe-finite[of ?U]
    by simp
  have stake n ‘ streams ?U = stake n ‘ space M
    using bernoulli
          bernoulli-stream-space[of M p]
    by auto
  then have stake n ‘ streams ?U = {c::bool list. length c = n}
    using stake-space[of n]
    by auto
  then show ?thesis
    using ‹finite (stake n ‘streams ?U)›
    by auto
qed
```

```
lemma finite-image:finite {c::bool list. (∀ k<i. (geom-proc-list init (take k c)) ∉
{0,target})∧ length c = i ∧ geom-proc-list init c = target}
  using finite-length[of i]
```

by *auto*

Sets of all successful random walk with specific stop is measurable

lemma *success-measurable3*:
fixes *init* **and** *target* **and** *i*::*nat*
assumes *0<initinit<target0* $\leq$ *i*
shows{*x*$\in$ *space M*. *success init x target* $\wedge$ *stop-at init x target* = *enat i*} $\in$ *sets M*
using *finite-image*[*of i*]
success-measurable2[*of init target i*]
finite-stake-measurable[*of* {*c*. ($\forall$ *k*<*i*.
geom-proc-list init (*take k c*)
$\notin$ {*0*, *target*}) $\wedge$
length c = *i* $\wedge$ *geom-proc-list init c* = *target*}]
assms
by *presburger*

Any successful random walk must stop at specific position described by natural number

lemma *success-measurable1*:
fixes *init target*
assumes *0* < *initinit* < *target*
shows {*x*$\in$ *space M*. *success init x target*}
= ($\bigcup$ *i*::*nat*. {*x*$\in$ *space M*. *success init x target* $\wedge$ *stop-at init x target* = *i*})
proof
show{*x* $\in$ *space M*. *success init x target*} $\subseteq$ ($\bigcup$ *x*. {*xa* $\in$ *space M*. *success init xa target* $\wedge$ *stop-at init xa target* = *enat x*})
proof
fix *x*
assume *x* $\in$ {*x* $\in$ *space M*. *success init x target*}
then have *success init x target*
by *auto*
then have *stop-at init x target* $\neq$ ∞
unfolding *success.simps*
using *assms*(*1*) *assms*(*2*) *exist*
by *force*
then obtain *i* **where** *stop-at init x target* = *enat i*
by *auto*
then have *x* $\in$ {*xa* $\in$ *space M*. *success init xa target* $\wedge$ *stop-at init xa target* = *enat i*}
using ‹*success init x target*›
bernoulli
bernoulli-stream-space[*of M p*]
by *auto*
then show *x* $\in$ ($\bigcup$ *x*. {*xa* $\in$ *space M*.
success init xa target $\wedge$ *stop-at init xa target* = *enat x*})
by *auto*
qed
next

show$(\bigcup x.\ \{xa \in \mathit{space}\ M.\ \mathit{success\ init\ xa\ target} \wedge \mathit{stop\text{-}at\ init\ xa\ target} = \mathit{enat}\ x\}) \subseteq \{x \in \mathit{space}\ M.\ \mathit{success\ init\ x\ target}\}$
proof
fix x
assume $x \in (\bigcup x.\ \{xa \in \mathit{space}\ M.$
$\mathit{success\ init\ xa\ target} \wedge$
$\mathit{stop\text{-}at\ init\ xa\ target} = \mathit{enat}\ x\})$
then have *success init x target*
by *auto*
then show $x \in \{x \in \mathit{space}\ M.\ \mathit{success\ init\ x\ target}\}$
using *bernoulli*
bernoulli-stream-space[*of M p*]
by *auto*
qed
qed

Here we need to elaborate about this most difficult lemma we've met during this model formalization. lemma *success_measurable* asserts that successful random walks set under assumption "$0 \leq initialnumber \leq targetnumber$" is measurable set for measure M. On the one hand, since the probability theory has been set up based on the measure theory, every specific set must be proved to be measurable with respect to fixed measure before we calculate the probability of the set, which severely hinders most of scholars and experts from formalizing the security analysis related to the probability since it's extremely difficult to prove why your set is measurable. That is exactly why our endeavor matters to provide the first example to overcome the difficulty. On the other hand, we are willing to briefly explain the way we prove this lemma since it's nontrivial even for pen-and-paper proof. lemma *finite_stake_measurable* states that for the function (λw. stake n w) taking the first n steps of random walk, the preimage of a finite sets is measurable for measure M. lemma *finite_image* states that sets filled with all bool list of fixed length n is finite. lemma *success_measurable*2 sets up the bijection between successful random walks stopping at fixed step and preimage of successful bool list with identical length. lemma *success_measurable*1 demonstrates that set of successful random walks is countable union of sets of successful random walks stopping at some step. Combining theses 4 lemmas together proves the set of successful random walk is measurable. If you take a closed look at the proofs of these 4 lemmas patiently, you will find it's very hard to finish. Honestly, we will never be able to finish such difficult proofs within one month without the current stochastic process theory library established just in 2021 by Mnacho Echenim, the author of theory *infinite_coin_toss_space*.

lemma *success-measurable*:
fixes *init target*
assumes $0 \leq \mathit{initinit} \leq \mathit{target}$
shows $\{x\in \mathit{space}\ M.\ \mathit{success\ init\ x\ target}\} \in \mathit{sets}\ M$

proof(*cases init* = *target*)
assume *equ*:*init* = *target*
then have $\bigwedge x$. *gambler-rand-walk-pre 1* (−*1*) *init 0 x* = *target*
using *enat-0 gambler-rand-walk-pre.simps*(*1*)[*of 1* −*1 init* -]
by *force*
then have $\bigwedge x$. *geom-proc init x 0* = *target*
unfolding *geometric-process gambler-rand-walk.simps gambler-rand-walk-pre.simps*
using *enat-0 gambler-rand-walk-pre.simps*(*1*)[*of 1* −*1 init* -]
enat.simps(*4*)[*of* λ*n. gambler-rand-walk-pre 1* (−*1*) *init n* - −*1 0*]
by *auto*
then have *belong-0*:∀ *x*. *0* ∈ *reach-steps init x target*
unfolding *reach-steps-def geometric-process gambler-rand-walk.simps gambler-rand-walk-pre.simps*(*1*)
using *equ*
by *auto*
then have ∀ *x*. ⨅ *reach-steps init x target* = *0*
using *not-belong*
by *auto*
then have ∀ *x*. *stop-at init x target* = *0*
unfolding *stop-at.simps infm.simps*
using *belong-0 enat-0*
by *auto*
then have ∀ *x*∈ *space M*. *stop-at init x target* = *0*
using *bernoulli-stream-space*[*of M p*] *bernoulli*
by *blast*
then have ∀ *x*∈ *space M*. *success init x target*
unfolding *success.simps*
using ‹$\bigwedge x$. *geom-proc init x 0* = *target*›
by *auto*
then show *?thesis*
by (*smt* (*verit*, *best*) *Collect-cong Collect-mem-eq sets.top*)
next
assume *init* ≠ *target*
show *?thesis*
proof(*cases init* = *0*)
assume *equ1*:*init* = *0*
then have $\bigwedge x$. *gambler-rand-walk-pre 1* (−*1*) *init 0 x* = *0*
using *enat-0 gambler-rand-walk-pre.simps*(*1*)[*of 1* −*1 init* -]
by *force*
then have $\bigwedge x$. *geom-proc init x 0* = *0*
unfolding *geometric-process gambler-rand-walk.simps gambler-rand-walk-pre.simps*
using *enat-0 gambler-rand-walk-pre.simps*(*1*)[*of 1* −*1 init* -]
enat.simps(*4*)[*of* λ*n. gambler-rand-walk-pre 1* (−*1*) *init n* - −*1 0*]
by *auto*
then have *belong-0*:∀ *x*. *0* ∈ *reach-steps init x target*
unfolding *reach-steps-def geometric-process gambler-rand-walk.simps gambler-rand-walk-pre.simps*(*1*)
using *equ1*
by *auto*

then have $\forall x. \prod$ *reach-steps init x target* $=$ *0*
  using *not-belong*
  by *auto*
then have $\forall x.$ *stop-at init x target* $=$ *0*
  unfolding *stop-at.simps infm.simps*
  using *belong-0 enat-0*
  by *auto*
then have *stop*:$\forall x \in$ *space M. stop-at init x target* $=$ *0*
  using *bernoulli-stream-space*[*of M p*] *bernoulli*
  by *blast*
then have $\forall x \in$ *space M.* $\neg$ *success init x target***if** *target* $\neq$ *0*
  unfolding *success.simps*
  using ‹$\bigwedge x.$ *geom-proc init x 0* $=$ *0*›
    that
  by *auto*
then have $\forall x \in$ *space M. success init x target***if** *target* $=$ *0*
  unfolding *success.simps*
  using ‹$\bigwedge x.$ *geom-proc init x 0* $=$ *0*›
    that
    stop
  by *auto*
then show *?thesis*
  using ‹*target* $\neq$ *0* $\Longrightarrow \forall x \in$*space M.* $\neg$ *success init x target*›
by (*metis* (*no-types*, *lifting*) *Collect-empty-eq* ‹*init* $\neq$ *target*› *equ1 sets.empty-sets*)
next
  assume*init*$\neq$*0*
  then have *0*$<$*initinit*$<$*target*
    using *assms* ‹*init* $\neq$ *target*›
    by *auto*
  then show *?thesis*
    using *success-measurable1*[*of init target*]
      success-measurable3[*of init target* -]
      assms
    by *auto*
qed
qed

The set of all the random walk with first step True is measurable

lemma *success-measurable-shd*:
$\{x \in$ *space M. shd x*$\} \in$ *sets M*
  using *snth-measurable*[*of 0 True*]
    snth.simps(*1*)
    bernoulli
    bernoulli-stream-space[*of M p*]
  by (*simp add*: *insert-compr streams-UNIV*)

The set of all the random walk with first step False is measurable

lemma *success-measurable-shd-false*:
$\{x \in$ *space M.* $\neg$ *shd x*$\} \in$ *sets M*

using *success-measurable-shd*
by *auto*

lemma *success-measurable-final*:
fixes *init target*
assumes $0 < init$ $init < target$
shows$\{x \in space\ M.\ success\ (init{+}1)\ (stl\ x)\ target \wedge shd\ x\} \in sets\ M$
proof–
have $\{x \in space\ M.\ success\ init\ x\ target \wedge shd\ x = True\} = \{x \in space\ M.\ success\ (init + 1)\ (stl\ x)\ target \wedge shd\ x = True\}$
using *conditional-set-equation*[*of init target*]
assms
bernoulli
bernoulli-stream-space[*of M p*]
by *auto*
moreover have $\{x \in space\ M.\ success\ init\ x\ target \wedge shd\ x = True\} \in sets\ M$
using *Sigma-Algebra.sets.Int*[*of* $\{x \in space\ M.\ shd\ x\}$ *M* $\{x \in space\ M.\ success\ init\ x\ target\}$]
success-measurable-shd
success-measurable[*of init target*]
assms
by *auto*
ultimately show *?thesis*
by *auto*
qed

1.5.2 Probability of successful random walk with its first step True

lemma *semi-goal1*:
fixes *init target P*
assumes $0 < init$$init \leq target$$\bigwedge x.\ P\ x = success\ (init{+}1)\ (stl\ x)\ target \wedge shd\ x$
shows $emeasure\ M\ \{x.\ P\ (t\ \#\#\ x)\}$
$= (case\ t\ of\ True \Rightarrow 1 | False \Rightarrow 0) * emeasure\ M\ \{x.\ success\ (init{+}1)\ x\ target\}$
proof (*cases t*)
assume *t*
then have $t = True$
by *auto*
then have $\forall x.\ shd\ (t\ \#\#\ x)$
by *auto*
then have $\forall x.\ P\ (t\ \#\#\ x) \longleftrightarrow success\ (init{+}1)\ x\ target$
using *assms*(*3*) *stream.sel*(*2*)[*of t* -] ‹$\forall x.\ shd\ (t\ \#\#\ x)$›
by *force*
then have $emeasure\ M\ \{x.\ P\ (t\ \#\#\ x)\} = emeasure\ M\ \{x.\ success\ (init{+}1)\ x\ target\}$
by *auto*
then show *?thesis*

```
    using ‹t = True›
    by auto
next
  assume ¬t
  then have t = False
    by auto
  then have ∀x. ¬shd (t ## x)
    by auto
  then have {x. P (t ## x)} = {}
    using assms(3)
    by auto
  then have emeasure M {x. P (t ## x)} = 0
    by auto
  then show ?thesis
    using ‹t = False›
    by auto
qed


lemma semi-goal21:
  fixes p1 e::real and f
  assumes m = measure-pmf (bernoulli-pmf p1)⋀t. f t = ennreal e ∗ (case t of
True ⇒ 1| False ⇒ 0)
  shows simple-function m f
  unfolding simple-function-def
proof
  show finite (f ‘ space m)
  proof−
    have space m = {True, False}
      using assms(1)
      by auto
    then have f ‘ space m = {0, e}
      using assms(2)
      by auto
    then show finite (f ‘ space m)
      by auto
  qed
next
  show ∀x∈f ‘ space m. f −‘ {x} ∩ space m ∈ sets m
  proof (cases e = 0)
    assume e = 0
    have space m = {True, False}
      using assms(1)
      by auto
    then have f ‘ space m = {0, e}
      using assms(2)
      by auto
    then have f −‘ {0} ∩ space m = {True,False}
      using assms ‹space m = {True, False}› ‹e = 0›
```

by *force*
then show *?thesis*
using ‹f ‘ *space* $m = \{0,\ e\}$›‹$e = 0$›
by (*metis* ‹*space* $m = \{True,\ False\}$› *ennreal-0 insert-absorb2 sets.top singletonD*)
next
assume $e \neq 0$
have *space* $m = \{True,\ False\}$
using *assms*(*1*)
by *auto*
then have f ‘ *space* $m = \{0,\ e\}$
using *assms*(*2*)
by *auto*
then show *?thesis*
using *assms* ‹$e \neq 0$› ‹*space* $m = \{True,\ False\}$›
by *simp*
qed
qed

lemma *semi-goal21-false*:
fixes *p1 e*::*real* **and** f
assumes m = *measure-pmf* (*bernoulli-pmf p1*)$\bigwedge t.\ f\ t$ = *ennreal* $e * ($*case* t *of* *True* $\Rightarrow 0|$ *False* $\Rightarrow 1)$
shows *simple-function* m f
unfolding *simple-function-def*
proof
show *finite* (f ‘ *space* m)
proof−
have *space* $m = \{True,\ False\}$
using *assms*(*1*)
by *auto*
then have f ‘ *space* $m = \{0,\ e\}$
using *assms*(*2*)
by *auto*
then show *finite* (f ‘ *space* m)
by *auto*
qed
next
show $\forall x \in f$ ‘ *space* m. f −‘ $\{x\} \cap$ *space* $m \in$ *sets* m
proof (*cases* $e = 0$)
assume $e = 0$
have *space* $m = \{True,\ False\}$
using *assms*(*1*)
by *auto*
then have f ‘ *space* $m = \{0,\ e\}$
using *assms*(*2*)
by *auto*
then have f −‘ $\{0\} \cap$ *space* $m = \{True,False\}$

using *assms* ‹*space m* = {*True*, *False*}› ‹*e* = *0*›
by *force*
then show *?thesis*
using ‹*f* ‘ *space m* = {*0*, *e*}›‹*e* = *0*›
by (*metis* ‹*space m* = {*True*, *False*}› *ennreal-0 insert-absorb2 sets.top singletonD*)
next
assume $e \neq 0$
have *space m* = {*True*, *False*}
using *assms*(*1*)
by *auto*
then have *f* ‘ *space m* = {*0*, *e*}
using *assms*(*2*)
by *auto*
then show *?thesis*
using *assms* ‹$e \neq 0$› ‹*space m* = {*True*, *False*}›
by *simp*
qed
qed

lemma *sum-rephrase*:
fixes *f*::*ennreal* ⇒ *ennreal* **and** *e*
assumes $0 \neq e$
shows *sum f* {*0*,*e*} = *f 0* + *f* (*e*)
using *assms finite.insertI insert-absorb insert-not-empty singleton-insert-inj-eq′ sum-clauses*(*1*)
by *auto*

lemma *semi-goal22*:
fixes *p1 e*::*real*
assumes *m* = *measure-pmf* (*bernoulli-pmf p1*)*0*≤*p1p1*≤*1*
⋀*t*. *f t* = *ennreal e* ∗ (*case t of True* ⇒ *1*| *False* ⇒ *0*)
e > *0*
shows $integral^S$ *m f* = *p1* ∗ *e*
unfolding *simple-integral-def*
proof–
show(∑*x*∈*f* ‘ *space m*. *x* ∗ *emeasure m* (*f* −‘ {*x*} ∩ *space m*)) = *ennreal* (*p1* ∗ *e*)
proof–
have *space m* = {*True*, *False*}
using *assms*(*1*)
by *auto*
have *f* −‘ {*0*} = {*False*}
proof
show*f* −‘ {*0*} ⊆ {*False*}
proof
fix *x*

```
    assume x ∈ f −‘ {0}
    then have f x = 0
      by auto
    then have x = False
      using assms(4)[of x]
            ‹e > 0›
             ennreal-eq-0-iff
      by auto
    then show x ∈ {False}
      by auto
  qed
next
  show {False} ⊆ f −‘ {0}
  proof
    fix x
    assume x ∈ {False}
    then have x = False
      by auto
    then have f x = 0
      using assms
      by auto
    then show x ∈ f −‘ {0}
      by auto
  qed
qed
have emeasure m ({False}) = 1 − p1
  unfolding assms(1) emeasure-pmf-single ennreal-cong
  using pmf-bernoulli-False[of p1]
        assms(2)
        assms(3)
        ennreal-cong[of pmf (bernoulli-pmf p1) False 1 − p1]
  by auto
then have emeasure m (f −‘ {0} ∩ space m) = 1 − p1
  using ‹f −‘ {0} = {False}› ‹space m = {True, False}›
  by auto
have f −‘ {ennreal e} = {True}
proof
  showf −‘ {ennreal e} ⊆ {True}
  proof
    fix x
    assume x ∈ f −‘ {ennreal e}
    then have f x = e
      by auto
    then have x = True
      using assms(4)[of x]
            ‹e > 0›
             ennreal-eq-0-iff
      by (smt (verit, best) mult-zero-right)
    then show x ∈ {True}
```

by *auto*
qed
next
show $\{True\} \subseteq f -` \{ennreal\ e\}$
using *assms*(*4*)
by *auto*
qed
have *emeasure m* ({*True*}) = *p1*
unfolding *assms*(*1*) *emeasure-pmf-single ennreal-cong*
using *pmf-bernoulli-False*[*of p1*]
assms(*2*)
assms(*3*)
ennreal-cong[*of pmf* (*bernoulli-pmf p1*) *False 1* − *p1*]
by *auto*
then have *emeasure m* ($f -` \{ennreal\ e\} \cap space\ m$) = *p1*
using ‹$f -` \{ennreal\ e\} = \{True\}$› ‹*space m* = {*True*, *False*}›
by *auto*
have *f* ‘ *space m* = {*0*, *ennreal e*}
using ‹*space m* = {*True*,*False*}› *assms*(*4*)
by *auto*
then have ($\sum x \in f$ ‘ *space m*. $x * emeasure\ m\ (f -` \{x\} \cap space\ m)$)
$= 0 * emeasure\ m\ (f -` \{0\} \cap space\ m) + ennreal\ e * emeasure\ m\ (f -` \{ennreal\ e\} \cap space\ m)$
using *sum-rephrase*[*of ennreal e* $\lambda x.\ x * emeasure\ m\ (f -` \{x\} \cap space\ m)$]
ennreal-eq-0-iff
‹*e* > *0*›
by *force*
then show *?thesis*
using
‹*space m* = {*True*,*False*}›
‹*emeasure m* ($f -` \{ennreal\ e\} \cap space\ m$) = *p1*›
‹*emeasure m* ($f -` \{0\} \cap space\ m$) = *1* − *p1*›
by (*metis add.left-neutral assms*(*2*) *ennreal-mult*′′ *mult.commute mult-zero-left*)
qed
qed

lemma *semi-goal22-false*:
fixes *p1 e*::*real*
assumes *m* = *measure-pmf* (*bernoulli-pmf p1*)*0*≤*p1p1*≤*1*
$\bigwedge t.\ f\ t = ennreal\ e * (case\ t\ of\ True \Rightarrow 0 |\ False \Rightarrow 1)$
e > *0*
shows $integral^S$ *m f* = (*1*−*p1*) ∗ *e*
unfolding *simple-integral-def*
proof−
show($\sum x \in f$ ‘ *space m*. $x * emeasure\ m\ (f -` \{x\} \cap space\ m)$) = *ennreal* ((*1*−*p1*) ∗ *e*)
proof−
have *space m* = {*True*, *False*}

```
    using assms(1)
    by auto
  have f -` {0} = {True}
  proof
    showf -` {0} ⊆ {True}
    proof
      fix x
      assume x ∈ f -` {0}
      then have f x = 0
        by auto
      then have x = True
        using assms(4)[of x]
              ‹e > 0›
              ennreal-eq-0-iff[of e]
        by (smt (verit, best) mult.right-neutral)
      then show x ∈ {True}
        by auto
    qed
  next
    show {True} ⊆ f -` {0}
    proof
      fix x
      assume x ∈ {True}
      then have x = True
        by auto
      then have f x = 0
        using assms
        by auto
      then show x ∈ f -` {0}
        by auto
    qed
  qed
  have emeasure m ({True}) = p1
    unfolding assms(1) emeasure-pmf-single ennreal-cong
    using pmf-bernoulli-False[of p1]
          assms(2)
          assms(3)
          ennreal-cong[of pmf (bernoulli-pmf p1) False 1 - p1]
    by auto
  then have emeasure m (f -` {0} ∩ space m) = p1
    using ‹f -` {0} = {True}› ‹space m = {True, False}›
    by auto
  have f -` {ennreal e} = {False}
  proof
    showf -` {ennreal e} ⊆ {False}
    proof
      fix x
      assume x ∈ f -` {ennreal e}
      then have f x = e
```

by *auto*
then have $x = False$
using *assms*(*4*)[*of x*]
$\langle e > 0\rangle$
ennreal-eq-0-iff
by (*smt* (*verit*, *best*) *mult-zero-right*)
then show $x \in \{False\}$
by *auto*
qed
next
show $\{False\} \subseteq f -` \{ennreal\ e\}$
using *assms*(*4*)
by *auto*
qed
have *emeasure m* ($\{False\}$) = $1-p1$
unfolding *assms*(*1*) *emeasure-pmf-single ennreal-cong*
using *pmf-bernoulli-False*[*of p1*]
assms(*2*)
assms(*3*)
ennreal-cong[*of pmf* (*bernoulli-pmf p1*) *False* $1 - p1$]
by *auto*
then have *emeasure m* ($f -` \{ennreal\ e\} \cap space\ m$) = $1- p1$
using $\langle f -` \{ennreal\ e\} = \{False\}\rangle$ $\langle space\ m = \{True, False\}\rangle$
by *auto*
have f ` $space\ m = \{0, ennreal\ e\}$
using $\langle space\ m = \{True,False\}\rangle$ *assms*(*4*)
by *auto*
then have ($\sum x \in f$ ` $space\ m.\ x * emeasure\ m\ (f -` \{x\} \cap space\ m)$)
$= 0 * emeasure\ m\ (f -` \{0\} \cap space\ m) + ennreal\ e * emeasure\ m\ (f -` \{ennreal\ e\} \cap space\ m)$
using *sum-rephrase*[*of ennreal e* $\lambda x.\ x * emeasure\ m\ (f -` \{x\} \cap space\ m)$]
ennreal-eq-0-iff
$\langle e > 0\rangle$
by *force*
moreover have $ennreal\ e * ennreal\ (1-p1) + 0 * ennreal\ p1 = ennreal\ ((1-p1)*e)$
using *assms*(*3*) *ennreal-mult'* *mult.commute* **by** *auto*
ultimately show *?thesis*
using
$\langle space\ m = \{True,False\}\rangle$
$\langle emeasure\ m\ (f -` \{ennreal\ e\} \cap space\ m) = 1 - p1\rangle$
$\langle emeasure\ m\ (f -` \{0\} \cap space\ m) = p1\rangle$
by *auto*
qed
qed

lemma *semi-goal23*:
fixes *p1 e*::*real* **and** *f*
assumes $0 \leq p1$

```
        p1 ≤ 1
        e > 0
m = measure-pmf (bernoulli-pmf p1)
⋀t. f t = ennreal e * (case t of True ⇒ 1| False ⇒ 0)
shows ∫+ t. (f t) ∂m = integral^S m f
  using nn-integral-eq-simple-integral semi-goal21[of m p1] assms
  by blast

lemma semi-goal23-false:
  fixes p1 e::real and f
  assumes 0 ≤ p1
        p1 ≤ 1
        e > 0
m = measure-pmf (bernoulli-pmf p1)
⋀t. f t = ennreal e * (case t of True ⇒ 0| False ⇒ 1)
shows ∫+ t. (f t) ∂m = integral^S m f
  using nn-integral-eq-simple-integral semi-goal21-false[of m p1] assms
  by blast


lemma semi-goal2:
  fixes p1 e::real and f
  assumes  0 ≤ p1
        p1 ≤ 1
        e ≥ 0
m = measure-pmf (bernoulli-pmf p1)
⋀t. f t = ennreal e * (case t of True ⇒ 1| False ⇒ 0)
shows ∫+ t. (f t) ∂m = p1 * e
proof(cases e = 0)
  assume e = 0
  then have ⋀t. f t = 0
    using assms
    by force
  then show ?thesis
    unfolding assms(4) ‹e = 0›
    using nn-integral-const[of m 0]
    by force
next
  assume e ≠ 0
  then have e > 0
    using assms
    by auto
  then show ?thesis
  using semi-goal22[of m p1 f e] semi-goal23[of p1 e m f] assms
  by auto
qed

lemma semi-goal2-false:
  fixes p1 e::real and f
```

assumes *0 ≤ p1*
p1 ≤ 1
e ≥ 0
m = measure-pmf (bernoulli-pmf p1)
⋀t. f t = ennreal e ∗ (case t of True ⇒ 0| False ⇒ 1)
shows *∫⁺ t. (f t) ∂m = (1−p1) ∗ e*
proof(*cases e = 0*)
assume *e = 0*
then have *⋀t. f t = 0*
using *assms*
by *force*
then show *?thesis*
unfolding *assms(4)* ‹*e = 0*›
using *nn-integral-const[of m 0]*
by *force*
next
assume *e ≠ 0*
then have *e > 0*
using *assms*
by *auto*
then show *?thesis*
using *semi-goal22-false[of m p1 f e] semi-goal23-false[of p1 e m f] assms*
by *auto*
qed

lemma *semi-goal2-final*:
fixes *p1::real* **and** *e::ennreal* **and** *f*
assumes *0 ≤ p1*
p1 ≤ 1
e ≠ top
m = measure-pmf (bernoulli-pmf p1)
⋀t. f t = e ∗ (case t of True ⇒ 1| False ⇒ 0)
shows *∫⁺ t. (f t) ∂m = p1 ∗ e*
proof−
obtain *e1* **where** *e1≥0***and** *ennreal e1 = e*
using *ennreal-cases[of e] assms(3)*
by *auto*
obtain *f1* **where** *⋀t. f1 t = e1 ∗ (case t of True ⇒ 1| False ⇒ 0)*
by *auto*
then have *⋀t. ennreal (f1 t) = f t*
unfolding *assms(5)* ‹*ennreal e1 = e*› ‹*⋀t. f1 t = e1 ∗ (case t of True ⇒ 1| False ⇒ 0)*›
using ‹*ennreal e1 = e*›
by *(smt (z3) ennreal-0 mult.right-neutral mult-cancel-left1 mult-cancel-right1 mult-zero-right)*
then have *(∫⁺ t. (f1 t) ∂m) = ∫⁺ t. (f t) ∂m*
by *force*
then show *?thesis*

using *assms semi-goal2*[*of p1 e1 m f1*] ‹*e1*≥*0*› ‹$\bigwedge$*t. f1 t* = *e1* ∗ (*case t of True* ⇒ *1*| *False* ⇒ *0*)›
‹$\bigwedge$*t. ennreal* (*f1 t*) = *f t*› ‹*ennreal e1* = *e*› *ennreal-mult''*
by *force*
qed

lemma *semi-goal2-final-false*:
fixes *p1*::*real* **and** *e*::*ennreal* **and** *f*
assumes *0* ≤ *p1*
p1 ≤ *1*
e ≠ *top*
m = *measure-pmf* (*bernoulli-pmf p1*)
$\bigwedge$*t. f t* = *e* ∗ (*case t of True* ⇒ *0*| *False* ⇒ *1*)
shows ∫ ⁺ *t*. (*f t*) ∂*m* =(*1* − *p1*) ∗ *e*
proof–
obtain *e1* **where** *e1*≥*0***and** *ennreal e1* = *e*
using *ennreal-cases*[*of e*] *assms*(*3*)
by *auto*
obtain *f1* **where** $\bigwedge$*t. f1 t* = *e1* ∗ (*case t of True* ⇒ *0*| *False* ⇒ *1*)
by *auto*
then have $\bigwedge$*t. ennreal* (*f1 t*) = *f t*
unfolding *assms*(*5*) ‹*ennreal e1* = *e*› ‹$\bigwedge$*t. f1 t* = *e1* ∗ (*case t of True* ⇒ *0*| *False* ⇒ *1*)›
using ‹*ennreal e1* = *e*›
by (*smt* (*z3*) *ennreal-0 mult.right-neutral mult-cancel-left1 mult-cancel-right1 mult-zero-right*)
then have (∫ ⁺ *t*. (*f1 t*) ∂*m*) = ∫ ⁺ *t*. (*f t*) ∂*m*
by *force*
then show *?thesis*
using *assms semi-goal2-false*[*of p1 e1 m f1*] ‹*e1*≥*0*› ‹$\bigwedge$*t. f1 t* = *e1* ∗ (*case t of True* ⇒ *0*| *False* ⇒ *1*)›
‹$\bigwedge$*t. ennreal* (*f1 t*) = *f t*› ‹*ennreal e1* = *e*› *ennreal-mult''*
by *force*
qed

lemma *fun-description-pre*:
fixes *init target t*
assumes *0* < *initinit* < *target*
shows
emeasure M {*x* ∈ *space M*. *t* ## *x* ∈ {*x*∈ *space M*. *success* (*init*+*1*) (*stl x*) *target* ∧ *shd x*}}
= (*case t of True* ⇒ *1*|*False* ⇒ *0*) ∗ (*emeasure M* {*x*∈ *space M*. *success* (*init*+*1*) (*x*) *target*})
proof(*cases t*)
assume *t*
then have *t* = *True*

```
    by auto
   have {x ∈ space M. t ## x ∈ {x∈ space M. success (init+1) (stl x) target ∧
shd x}} = {x∈ space M. success (init+1) (x) target}
        if  t = True
      proof
        show {x ∈ space M. t ## x ∈ {x∈ space M. success (init+1) (stl x) target
∧ shd x}} ⊆ {x ∈ space M. success (init + 1) x target}
        proof
          fix x
         assume x ∈ {x ∈ space M. (t ## x) ∈ {x∈ space M. success (init+1) (stl
x) target ∧ shd x}}
          then have (t ## x) ∈ {x∈ space M. success (init+1) (stl x) target ∧ shd
x}
            by blast
          then have success (init+1) (stl (t ## x)) target ∧ shd (t ## x)
            by blast
          then have success (init+1) (x) target
            using that
                  stream.sel(2)
            by force
          then show x ∈ {x ∈ space M. success (init + 1) x target}
            using bernoulli-stream-space[of M p]
                  bernoulli
            by auto
        qed
      next
        show {x ∈ space M. success (init + 1) x target} ⊆ {x ∈ space M. t ## x
∈ {x∈ space M. success (init+1) (stl x) target ∧ shd x}}
        proof
          fix x
          assume x ∈ {x ∈ space M. success (init + 1) x target}
          then have x ∈ space Msuccess (init + 1) x target
            by auto
          then have (t ## x) ∈ space M
            using stream-space-Stream
          proof−
            have t ∈ space (measure-pmf (bernoulli-pmf p))
              by fastforce
            then have (t ## x) ∈ space M
              using ‹x ∈ space M›
                    stream-space-Stream[of  t x]
                    bernoulli
                    bernoulli-stream-def[of p]
              by auto
            then show ?thesis
              using bernoulli
                      bernoulli-stream-def[of p]
                      that
              by auto
```

qed
then have *success* (*init*+*1*) (*stl* (*t* ## *x*)) *targetshd* (*t* ## *x*)(*t* ## *x*) ∈ *space M*
using *stream.sel*(*2*) *stream.sel*(*1*) *that*
‹*x* ∈ *space M*›
‹*success* (*init* + *1*) *x target*›
by *auto*
then have *t* ## *x* ∈ {*x*∈ *space M*. *success* (*init*+*1*) (*stl x*) *target* ∧ *shd x*}
unfolding *that*
by *force*
then show *x* ∈ {*x* ∈ *space M*. *t* ## *x* ∈ {*x*∈ *space M*. *success* (*init*+*1*) (*stl x*) *target* ∧ *shd x*}}
using *bernoulli-stream-space*[*of M p*]
bernoulli
‹*x* ∈ *space M*›
by *force*
qed
qed
then have {*x* ∈ *space M*. *t* ## *x* ∈ {*x* ∈ *space M*. *success* (*init* + *1*) (*stl x*) *target* ∧ *shd x*}} = {*x* ∈ *space M*. *success* (*init* + *1*) *x target*}
using ‹*t* = *True*›
by *auto*
then have *emeasure M* {*x* ∈ *space M*. *t* ## *x* ∈ {*x* ∈ *space M*. *success* (*init* + *1*) (*stl x*) *target* ∧ *shd x*}} = *emeasure M* {*x* ∈ *space M*. *success* (*init* + *1*) *x target*}
using ‹*t* = *True*›
by *auto*
then show *?thesis*
using ‹*t* = *True*›
by *auto*
next
assume ¬ *t*
then have *t* = *False*
by *auto*
moreover have {*x* ∈ *space M*. *t* ## *x* ∈ {*x*∈ *space M*. *success* (*init*+*1*) (*stl x*) *target* ∧ *shd x*}} = {}
if *t* = *False*
proof−
have ∀ *x* ∈ *space M*. *t* ## *x* ∉ {*x*∈ *space M*. *success* (*init*+*1*) (*stl x*) *target* ∧ *shd x*}
using *stream.sel*(*1*) *that*
by *auto*
then show *?thesis*
by *blast*
qed
ultimately have *emeasure M* {*x* ∈ *space M*. *t* ## *x* ∈ {*x* ∈ *space M*. *success* (*init* + *1*) (*stl x*) *target* ∧ *shd x*}} = *0*
by *force*
then show *?thesis*

using ‹t = *False*›
by *auto*
qed

lemma *fun-description-pre-false*:
fixes *init target t*
assumes $0 < init init < target$
shows
emeasure M $\{x \in$ *space M*. t ## $x \in \{x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* ∧ ¬ *shd x*}}
= (*case t of True* ⇒ *0*|*False* ⇒ *1*) ∗ (*emeasure M* {$x\in$ *space M*. *success* (*init*−*1*) (x) *target*})
proof(*cases t*)
assume ¬t
then have t = *False*
by *auto*
have {$x \in$ *space M*. t ## $x \in$ {$x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* ∧ ¬ *shd x*}} = {$x\in$ *space M*. *success* (*init*−*1*) (x) *target*}
if t = *False*
proof
show {$x \in$ *space M*. t ## $x \in$ {$x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* ∧ ¬ *shd x*}} ⊆ {$x \in$ *space M*. *success* (*init* − *1*) *x target*}
proof
fix x
assume $x \in$ {$x \in$ *space M*. (t ## x) ∈ {$x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* ∧ ¬ *shd x*}}
then have (t ## x) ∈ {$x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* ∧ ¬ *shd x*}
by *blast*
then have *success* (*init*−*1*) (*stl* (t ## x)) *target* ∧ ¬ *shd* (t ## x)
by *blast*
then have *success* (*init*−*1*) (x) *target*
using *that*
stream.sel(*2*)
by *force*
then show $x \in$ {$x \in$ *space M*. *success* (*init* − *1*) *x target*}
using *bernoulli-stream-space*[*of M p*]
bernoulli
by *auto*
qed
next
show {$x \in$ *space M*. *success* (*init* − *1*) *x target*} ⊆ {$x \in$ *space M*. t ## x ∈ {$x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* ∧ ¬ *shd x*}}
proof
fix x
assume $x \in$ {$x \in$ *space M*. *success* (*init* − *1*) *x target*}
then have $x \in$ *space Msuccess* (*init* − *1*) *x target*
by *auto*
then have (t ## x) ∈ *space M*

using *stream-space-Stream*
proof−
have $t \in$ *space* (*measure-pmf* (*bernoulli-pmf p*))
by *fastforce*
then have (t ## x) $\in$ *space M*
using ‹$x \in$ *space M*›
stream-space-Stream[*of t x*]
bernoulli
bernoulli-stream-def[*of p*]
by *auto*
then show *?thesis*
using *bernoulli*
bernoulli-stream-def[*of p*]
that
by *auto*
qed
then have *success* (*init*−*1*) (*stl* (t ## x)) *target*¬ *shd* (t ## x)(t ## x) $\in$ *space M*
using *stream.sel*(*2*) *stream.sel*(*1*) *that*
‹$x \in$ *space M*›
‹*success* (*init* − *1*) *x target*›
by *auto*
then have t ## $x \in$ {$x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* ∧ ¬ *shd x*}
unfolding *that*
by *force*
then show $x \in$ {$x \in$ *space M*. t ## $x \in$ {$x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* ∧ ¬ *shd x*}}
using *bernoulli-stream-space*[*of M p*]
bernoulli
‹$x \in$ *space M*›
by *force*
qed
qed
then have {$x \in$ *space M*. t ## $x \in$ {$x \in$ *space M*. *success* (*init* − *1*) (*stl x*) *target* ∧ ¬ *shd x*}} = {$x \in$ *space M*. *success* (*init* − *1*) *x target*}
using ‹t = *False*›
by *auto*
then have *emeasure M* {$x \in$ *space M*. t ## $x \in$ {$x \in$ *space M*. *success* (*init* − *1*) (*stl x*) *target* ∧ ¬ *shd x*}} = *emeasure M* {$x \in$ *space M*. *success* (*init* − *1*) *x target*}
using ‹t = *False*›
by *auto*
then show *?thesis*
using ‹t = *False*›
by *auto*
next
assume t
then have t = *True*

by *auto*

moreover have $\{x \in$ *space* $M.$ t ## $x \in \{x\in$ *space* $M.$ *success* (*init*−*1*) (*stl* x) *target* $\wedge \neg$ *shd* $x\}\} = \{\}$

if t = *True*

proof−

have $\forall x \in$ *space* $M.$ t ## $x \notin \{x\in$ *space* $M.$ *success* (*init*−*1*) (*stl* x) *target* $\wedge \neg$ *shd* $x\}$

using *stream.sel*(*1*) *that*

by *auto*

then show *?thesis*

by *blast*

qed

ultimately have *emeasure* M $\{x \in$ *space* $M.$ t ## $x \in \{x \in$ *space* $M.$ *success* (*init* − *1*) (*stl* x) *target* $\wedge \neg$ *shd* $x\}\} = 0$

by *force*

then show *?thesis*

using ‹t = *True*›

by *auto*

qed

term*emeasure-stream-space*

The lemma *semi_goal_true* is the second difficulty we've overcome during the model formalization. It asserts that probability of sets of successful random walk with first step True is equal to probability of sets of random walk times probability of sets of successful random walk with initial number plus 1. Thanks to the lemma *emeasure_stream_space* provided by Mnacho Echenim, the author of $infinite\_coin\_toss\_space$, we could finally use the integral rather than tediously break down the countable product to calculate the probability

lemma *semi-goal-true*:

fixes *init target*

assumes *0* < *initinit* < *target*

shows *emeasure* M $\{x\in$ *space* $M.$ *success* (*init*+*1*) (*stl* x) *target* $\wedge$ *shd* $x\}$ = *emeasure* M $\{x \in$ *space* $M.$ *shd* $x\}$ ∗ *emeasure* M $\{x\in$ *space* $M.$ *success* (*init*+*1*) (x) *target*$\}$

proof−

let *?M* = *measure-pmf* (*bernoulli-pmf* p)

have $\bigwedge X.$ $X \in$ *sets* (*stream-space* *?M*) $\Longrightarrow$ *emeasure* (*stream-space* *?M*) $X = \int^{+} t.$ *emeasure* (*stream-space* *?M*) $\{x \in$ *space* (*stream-space* *?M*). t ## $x \in X\}$ ∂*?M*

using *emeasure-stream-space*

by (*smt* (*verit*, *best*) *Collect-cong nn-integral-cong prob-space.emeasure-stream-space prob-space-measure-pmf*)

moreover have $\{x\in$ *space* $M.$ *success* (*init*+*1*) (*stl* x) *target* $\wedge$ *shd* $x\} \in$ *sets* (*stream-space* *?M*)

using *success-measurable-final*[*of init target*] *assms*

bernoulli

by (*metis bernoulli-stream-def*)
moreover have $\int^{+}$ *t*. *emeasure M* $\{x \in$ *space M*. *t* ## $x \in \{x \in$ *space M*. *success* (*init* + *1*) (*stl x*) *target* $\wedge$ *shd x*$\}\}$
∂*measure-pmf* (*bernoulli-pmf p*) =
ennreal p $*$ *emeasure* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))) $\{x \in$ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))). *success* (*init* + *1*) *x target*$\}$
proof−
have *emeasure* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*)))
$\{x \in$ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))). *success* (*init* + *1*) *x target*$\}$ $\neq$ *top*
using *emeasure-finite*[*of* $\{x \in$ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))).
success (*init* + *1*) *x target*$\}$]
bernoulli
bernoulli-stream-def[*of p*]
by *force*
moreover have ($\bigwedge$*t*. *emeasure M* $\{x \in$ *space M*. *t* ## $x \in \{x \in$ *space M*. *success* (*init* + *1*) (*stl x*) *target* $\wedge$ *shd x*$\}\}$ =
emeasure (*stream-space* (*measure-pmf* (*bernoulli-pmf p*)))
$\{x \in$ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))). *success* (*init* + *1*) *x target*$\}$ $*$
(*case t of True* $\Rightarrow$ *1* | *False* $\Rightarrow$ *0*))
using *fun-description-pre*[*of init target* -] *assms*
bernoulli
bernoulli-stream-def[*of p*]
mult.commute
by *fastforce*
ultimately show *?thesis*
using *semi-goal2-final*[*of p*
emeasure (*stream-space ?M*) $\{x \in$ *space* (*stream-space ?M*). *success* (*init*+*1*) (*x*) *target*$\}$
measure-pmf (*bernoulli-pmf p*)
λ*t*. *emeasure M* $\{x \in$ *space M*. *t* ## $x \in \{x \in$ *space M*. *success* (*init* + *1*) (*stl x*) *target* $\wedge$ *shd x*$\}\}$]
p-gt-0 p-lt-1
bernoulli
bernoulli-stream-def[*of p*]
by *force*
qed
ultimately have *emeasure* (*stream-space ?M*) $\{x\in$ *space M*. *success* (*init*+*1*) (*stl x*) *target* $\wedge$ *shd x*$\}$
= *ennreal p* $*$
emeasure (*stream-space* (*measure-pmf* (*bernoulli-pmf p*)))
$\{x \in$ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))). *success* (*init* + *1*) *x target*$\}$
using *bernoulli*
bernoulli-stream-def[*of p*]
by *force*
moreover have *emeasure M* $\{x \in$ *space M*. *shd x*$\}$ = *p*

proof−
have $\forall n.$ *emeasure M* $\{w \in$ *space M*$.$ *w* !! *n*$\}$ = *ennreal p*
using *bernoulli-stream-component-probability*[*of M p*]
bernoulli
p-gt-0
p-lt-1
snth.simps(*1*)
by *auto*
then show *?thesis*
using *snth.simps*(*1*)
by (*metis* (*no-types*, *lifting*) *Collect-cong*)
qed
ultimately show *?thesis*
using *bernoulli*
bernoulli-stream-def[*of p*]
by *force*
qed

lemma *semi-goal-false*:
fixes *init target*
assumes *0* < *initinit* < *target*
shows *emeasure M* $\{x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* $\wedge\ \neg$ *shd x*$\}$ = *emeasure M* $\{x \in$ *space M*. $\neg$ *shd x*$\}$ ∗ *emeasure M* $\{x\in$ *space M*. *success* (*init*−*1*) (*x*) *target*$\}$
proof−
let *?M* = *measure-pmf* (*bernoulli-pmf p*)
have $\bigwedge X.$ $X \in$ *sets* (*stream-space ?M*) $\Longrightarrow$
emeasure (*stream-space ?M*) *X* = $\int^{+}$ *t*. *emeasure* (*stream-space ?M*) $\{x \in$ *space* (*stream-space ?M*). *t* ## *x* $\in X\}$ ∂*?M*
using *emeasure-stream-space*
by (*smt* (*verit*, *best*) *Collect-cong nn-integral-cong prob-space.emeasure-stream-space prob-space-measure-pmf*)
moreover have $\{x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* $\wedge\ \neg$ *shd x*$\}$ $\in$ *sets* (*stream-space ?M*)
proof−
have $\{x\in$ *space M*. *success* (*init*−*1*) (*stl x*) *target* $\wedge\ \neg$ *shd x*$\}$ = $\{x\in$ *space M*. *success init x target* $\wedge\ \neg$ *shd x*$\}$
using *conditional-set-equation-false*[*of init target*]
assms
bernoulli-stream-space[*of M p*]
bernoulli
by *auto*
moreover have $\{x\in$ *space M*. *success init x target* $\wedge\ \neg$ *shd x*$\}$ $\in$ *sets M*
using *success-measurable-shd-false*
success-measurable[*of init target*]
assms
by *auto*
ultimately show *?thesis*

using *bernoulli*
bernoulli-stream-def[*of p*]
by *force*
qed
moreover have $\int^+ t.$ *emeasure M* $\{x \in$ *space M*. t ## $x \in \{x \in$ *space M*. *success* (*init* − *1*) (*stl x*) *target* ∧ ¬ *shd x*}}
∂*measure-pmf* (*bernoulli-pmf p*) =
ennreal (*1*−*p*) ∗ *emeasure* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))) {*x* ∈ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))). *success* (*init* − *1*) *x target*}
proof–
have *emeasure* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*)))
{*x* ∈ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))). *success* (*init* − *1*) *x target*} ≠ *top*
using *emeasure-finite*[*of* {*x* ∈ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))).
success (*init* − *1*) *x target*}]
bernoulli
bernoulli-stream-def[*of p*]
by *force*
moreover have (⋀*t*. *emeasure M* {*x* ∈ *space M*. *t* ## *x* ∈ {*x* ∈ *space M*. *success* (*init* − *1*) (*stl x*) *target* ∧ ¬ *shd x*}} =
emeasure (*stream-space* (*measure-pmf* (*bernoulli-pmf p*)))
{*x* ∈ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))). *success* (*init* − *1*) *x target*} ∗
(*case t of True* ⇒ *0* | *False* ⇒ *1*))
using *fun-description-pre-false*[*of init target* -] *assms*
bernoulli
bernoulli-stream-def[*of p*]
mult.commute
by *fastforce*
ultimately show *?thesis*
using *semi-goal2-final-false*[*of p*
emeasure (*stream-space ?M*) {*x* ∈ *space* (*stream-space ?M*). *success* (*init*−*1*) (*x*) *target*}
measure-pmf (*bernoulli-pmf p*)
λ*t*. *emeasure M* {*x* ∈ *space M*. (*t*) ## *x* ∈ {*x* ∈ *space M*. *success* (*init* − *1*) (*stl x*) *target* ∧ ¬ *shd x*}}]
p-gt-0 p-lt-1
bernoulli
bernoulli-stream-def[*of p*]
by *force*

qed
ultimately have *emeasure* (*stream-space ?M*) {*x*∈ *space M*. *success* (*init*−*1*) (*stl x*) *target* ∧ ¬ *shd x*}
= *ennreal* (*1*−*p*) ∗
emeasure (*stream-space* (*measure-pmf* (*bernoulli-pmf p*)))
{*x* ∈ *space* (*stream-space* (*measure-pmf* (*bernoulli-pmf p*))). *success* (*init* − *1*) *x target*}

using *bernoulli*
bernoulli-stream-def[*of p*]
by *force*
moreover have *emeasure M* {*x* ∈ *space M*. ¬ *shd x*} = *1*−*p*
proof−
have ∀ *n*. *emeasure M* {*w* ∈ *space M*. ¬ *w* !! *n*} = *ennreal* (*1*−*p*)
using *bernoulli-stream-component-probability-compl*[*of M p*]
bernoulli
p-gt-0
p-lt-1
snth.simps(*1*)
by *auto*
then show *?thesis*
using *snth.simps*(*1*)
by (*metis* (*no-types*, *lifting*) *Collect-cong*)
qed
ultimately show *?thesis*
using *bernoulli*
bernoulli-stream-def[*of p*]
by *force*
qed

1.5.3 Final goal: establish the recursive probability equation

The final probability equation we want to formalize:

$$P_n = pP_{n+1} + (1-p)P_{n-1}$$

lemma *Recursive-probability-equation*:
fixes *init target*
assumes *0* < *init init* < *target*
shows*probability-of-win init target* = *p* ∗ (*probability-of-win* (*init* + *1*) *target*) + (*1* − *p*) ∗ (*probability-of-win* (*init* − *1*) *target*)
unfolding *probability-of-win.simps*
proof−
have *emeasure M* {*x*∈ *space M*. *success init x target*}
= *emeasure M* {*x*∈ *space M*. *success init x target* ∧ *shd x*}
+ *emeasure M* {*x*∈ *space M*. *success init x target* ∧ ¬ (*shd x*)}
proof−
have {*x*∈ *space M*. *success init x target* ∧ ¬ (*shd x*)} ∪ {*x*∈ *space M*. *success init x target* ∧ (*shd x*)} =
{*x*∈ *space M*. *success init x target*}
by *auto*
moreover have {*x*∈ *space M*. *success init x target* ∧ ¬ (*shd x*)} ∩ {*x*∈ *space M*. *success init x target* ∧ (*shd x*)} = {}
by *auto*
moreover have {*x* ∈ *space M*. *success init x target* ∧ ¬ *shd x*} ∈ *sets M*
using *success-measurable-shd-false*
success-measurable-shd

success-measurable[*of init target*]
assms
Sigma-Algebra.sets.Int
by *auto*
moreover have $\{x \in$ *space M. success init x target* $\wedge$ *shd x*$\} \in$ *sets M*
using *success-measurable-shd-false*
success-measurable-shd
success-measurable[*of init target*]
assms
Sigma-Algebra.sets.Int
by *auto*
moreover have *emeasure M* $\{\} = 0$
by *auto*
ultimately show *?thesis*
using *emeasure-Un-Int*[*of* $\{x \in$ *space M. success init x target* $\wedge \neg$ (*shd x*)$\}$ *M* $\{x \in$ *space M. success init x target* $\wedge$(*shd x*)$\}$]
by (*metis* (*no-types*, *lifting*) *add.commute plus-emeasure*)
qed
moreover have *emeasure M* $\{x \in$ *space M. success init x target* $\wedge$ *shd x*$\}$ = *emeasure M* $\{x \in$ *space M. shd x*$\}$ $*$ *emeasure M* $\{x \in$ *space M. success* (*init* + *1*) *x target*$\}$
using *semi-goal-true*[*of init target*]
conditional-set-equation[*of init target*]
assms
by (*smt* (*verit*, *ccfv-SIG*) *Collect-cong mem-Collect-eq*)
moreover have *emeasure M* $\{x \in$ *space M. success init x target* $\wedge \neg$ *shd x*$\}$ = *emeasure M* $\{x \in$ *space M.* $\neg$ *shd x*$\}$ $*$ *emeasure M* $\{x \in$ *space M. success* (*init* − *1*) *x target*$\}$
using *semi-goal-false*[*of init target*]
conditional-set-equation-false[*of init target*]
assms
by (*smt* (*verit*, *ccfv-SIG*) *Collect-cong mem-Collect-eq*)
moreover have *emeasure M* $\{x \in$ *space M.* $\neg$ *shd x*$\}$ = *1−p*
proof−
have $\forall$ *n. emeasure M* $\{w \in$ *space M.* $\neg$ *w* !! *n*$\}$ = *ennreal* (*1−p*)
using *bernoulli-stream-component-probability-compl*[*of M p*]
bernoulli
p-gt-0
p-lt-1
snth.simps(*1*)
by *auto*
then show *?thesis*
using *snth.simps*(*1*)
by (*metis* (*no-types*, *lifting*) *Collect-cong*)
qed
moreover have *emeasure M* $\{x \in$ *space M. shd x*$\}$ = *p*
proof−
have $\forall$ *n. emeasure M* $\{w \in$ *space M. w* !! *n*$\}$ = *ennreal p*
using *bernoulli-stream-component-probability*[*of M p*]

bernoulli
p-gt-0
p-lt-1
snth.simps(*1*)
by *auto*
then show *?thesis*
using *snth.simps*(*1*)
by (*metis* (*no-types*, *lifting*) *Collect-cong*)
qed
ultimately show *emeasure M* $\{x \in$ *space M*. *success init x target*$\}$ =
ennreal p $*$ *emeasure M* $\{x \in$ *space M*. *success* (*init* + *1*) *x target*$\}$ +
ennreal (*1* − *p*) $*$ *emeasure M* $\{x \in$ *space M*. *success* (*init* − *1*) *x target*$\}$
by *force*
qed

end
end