Risk-Free Lending

Matthew Doty

March 17, 2025

Abstract

We construct an abstract ledger supporting the *risk-free lending* protocol. The risk-free lending protocol is a system for issuing and exchanging novel financial products we call *risk-free loans*. The system allows one party to lend money at 0% APY to another party in exchange for a good or service. On every update of the ledger, accounts have interest distributed to them. Holders of lent assets keep interest accrued by those assets. After distributing interest, the system returns a fixed fraction of each loan. These fixed fractions determine *loan periods*. Loans for longer periods have a smaller fixed fraction returned. Loans may be re-lent or used as collateral for other loans. We give a sufficient criterion to enforce that all accounts will forever be solvent. We give a protocol for maintaining this invariant when transferring or lending funds. We also show that this invariant holds after an update. Even though the system does not track counter-party obligations, we show that all credited and debited loans cancel, and the monetary supply grows at a specified interest rate.

Contents

1 Accounts **1**

2 Strictly Solvent **4**

3 Cash **6**

4 Ledgers **7**

4.1 Balanced Ledgers 8
4.2 Transfers 9
4.3 The Valid Transfers Protocol 11
4.4 Embedding Conventional Cash-Only Ledgers 12

5 Interest **14**

5.1 Net Asset Value 14
5.1.1 The Shortest Period for Credited & Debited Assets in an Account 15

5.1.2 Net Asset Value Properties 19
5.2 Distributing Interest 21

6 Update **22**
6.1 Update Preserves Ledger Balance 23
6.2 Strictly Solvent is Forever Strictly Solvent 28

7 Bulk Update **32**
7.1 Decomposition 34
7.2 Simple Transfers 36
7.3 Closed Forms 43

theory *Risk-Free-Lending*
  imports
    Complex-Main
    HOL−Cardinals.Cardinals
begin

1 Accounts

We model accounts as functions from *nat* to *real* with *finite support*.

Index *0* corresponds to an account's *cash* reserve (see §3 for details).

An index greater than *0* may be regarded as corresponding to a financial product. Such financial products are similar to *notes*. Our notes are intended to be as easy to use for exchange as cash. Positive values are debited. Negative values are credited.

We refer to our new financial products as *risk-free loans*, because they may be regarded as 0% APY loans that bear interest for the debtor. They are *risk-free* because we prove a *safety* theorem for them. Our safety theorem proves no account will "be in the red", with more credited loans than debited loans, provided an invariant is maintained. We call this invariant *strictly solvent*. See §7 for details on our safety proof.

Each risk-free loan index corresponds to a progressively shorter *loan period*. Informally, a loan period is the time it takes for 99% of a loan to be returned given a *rate function* ϱ. Rate functions are introduced in §6.

It is unnecessary to track counter-party obligations so we do not. See §4.1 and §4.2 for details.

typedef *account* = (*fin-support 0 UNIV*) :: (*nat* ⇒ *real*) *set*
proof −
  have (λ - . *0*) ∈ *fin-support 0 UNIV*
    unfolding *fin-support-def support-def*
    by *simp*
  thus ∃ *x* :: *nat* ⇒ *real*. *x* ∈ *fin-support 0 UNIV* **by** *fastforce*

qed

The type definition for *account* automatically generates two functions: *Rep-account* and *Rep-account*. *Rep-account* is a left inverse of *Abs-account*. For convenience we introduce the following shorthand notation:

notation *Rep-account* (‹π›)
notation *Abs-account* (‹ι›)

Accounts form an Abelian group. *Summing* accounts will be helpful in expressing how all credited and debited loans can cancel across a ledger. This is done in §4.1.

It is also helpful to think of an account as a transferable quantity. Transferring subtracts values under indexes from one account and adds them to another. Transfers are presented in §4.2.

instantiation *account* :: *ab-group-add*
begin

definition $0 = \iota\ (\lambda\ \text{-}\ .\ 0)$
definition $-\ \alpha = \iota\ (\lambda\ n\ .\ -\ \pi\ \alpha\ n)$
definition $\alpha_1 + \alpha_2 = \iota\ (\lambda\ n.\ \pi\ \alpha_1\ n + \pi\ \alpha_2\ n)$
definition $(\alpha_1 :: \mathit{account}) - \alpha_2 = \alpha_1 + -\ \alpha_2$

lemma *Rep-account-zero* [*simp*]: $\pi\ 0 = (\lambda\ \text{-}\ .\ 0)$
proof –
  have $(\lambda\ \text{-}\ .\ 0) \in$ *fin-support 0 UNIV*
    unfolding *fin-support-def support-def*
    by *simp*
  thus *?thesis*
    unfolding *zero-account-def*
    using *Abs-account-inverse* **by** *blast*
qed

lemma *Rep-account-uminus* [*simp*]:
  $\pi\ (-\ \alpha) = (\lambda\ n\ .\ -\ \pi\ \alpha\ n)$
proof –
  have $\pi\ \alpha \in$ *fin-support 0 UNIV*
    using *Rep-account* **by** *blast*
  hence $(\lambda x.\ -\ \pi\ \alpha\ x) \in$ *fin-support 0 UNIV*
    unfolding *fin-support-def support-def*
    by *force*
  thus *?thesis*
    unfolding *uminus-account-def*
    using *Abs-account-inverse* **by** *blast*
qed

lemma *fin-support-closed-under-addition*:
  fixes $f\ g :: {'a} \Rightarrow \mathit{real}$

```
  assumes f ∈ fin-support 0 A
  and g ∈ fin-support 0 A
  shows (λ x . f x + g x) ∈ fin-support 0 A
  using assms
  unfolding fin-support-def support-def
  by (metis (mono-tags) mem-Collect-eq sum.finite-Collect-op)

lemma Rep-account-plus [simp]:
  π (α1 + α2) = (λ n. π α1 n + π α2 n)
  unfolding plus-account-def
  by (metis (full-types)
        Abs-account-cases
        Abs-account-inverse
        fin-support-closed-under-addition)

instance
proof(standard)
  fix a b c :: account
  have ∀ n. π (a + b) n + π c n = π a n + π (b + c) n
    using Rep-account-plus plus-account-def
    by auto
  thus a + b + c = a + (b + c)
    unfolding plus-account-def
    by force
next
  fix a b :: account
  show a + b = b + a
    unfolding plus-account-def
    by (simp add: add.commute)
next
  fix a :: account
  show 0 + a = a
    unfolding plus-account-def Rep-account-zero
    by (simp add: Rep-account-inverse)
next
  fix a :: account
  show − a + a = 0
    unfolding plus-account-def zero-account-def Rep-account-uminus
    by (simp add: Abs-account-inverse)
next
  fix a b :: account
  show a − b = a + − b
    using minus-account-def by blast
qed

end
```

2 Strictly Solvent

An account is *strictly solvent* when, for every loan period, the sum of all the debited and credited loans for longer periods is positive. This implies that the *net asset value* for the account is positive. The net asset value is the sum of all of the credit and debit in the account. We prove *strictly-solvent* $\alpha \Longrightarrow 0 \leq$ *net-asset-value* α in §5.1.2.

definition *strictly-solvent* :: *account* ⇒ *bool* **where**
  strictly-solvent $\alpha \equiv \forall\ n\ .\ 0 \leq (\sum\ i{\leq}n\ .\ \pi\ \alpha\ i)$

lemma *additive-strictly-solvent*:
  assumes *strictly-solvent* α_1 **and** *strictly-solvent* α_2
  shows *strictly-solvent* $(\alpha_1 + \alpha_2)$
  using *assms Rep-account-plus*
  unfolding *strictly-solvent-def plus-account-def*
  by (*simp add*: *Abs-account-inverse sum.distrib*)

The notion of strictly solvent generalizes to a partial order, making *account* an ordered Abelian group.

instantiation *account* :: *ordered-ab-group-add*
begin

definition *less-eq-account* :: *account* ⇒ *account* ⇒ *bool* **where**
  less-eq-account $\alpha_1\ \alpha_2 \equiv \forall\ n\ .\ (\sum\ i{\leq}n\ .\ \pi\ \alpha_1\ i) \leq (\sum\ i{\leq}n\ .\ \pi\ \alpha_2\ i)$

definition *less-account* :: *account* ⇒ *account* ⇒ *bool* **where**
  less-account $\alpha_1\ \alpha_2 \equiv (\alpha_1 \leq \alpha_2 \land \neg\ \alpha_2 \leq \alpha_1)$

instance
proof(*standard*)
  fix $x\ y$ :: *account*
  show $(x < y) = (x \leq y \land \neg\ y \leq x)$
    unfolding *less-account-def* **..**
next
  fix x :: *account*
  show $x \leq x$
    unfolding *less-eq-account-def* **by** *auto*
next
  fix $x\ y\ z$ :: *account*
  assume $x \leq y$ **and** $y \leq z$
  thus $x \leq z$
    unfolding *less-eq-account-def*
    by (*meson order-trans*)
next
  fix $x\ y$ :: *account*
  assume $x \leq y$ **and** $y \leq x$
  hence ⋆: $\forall\ n\ .\ (\sum\ i{\leq}n\ .\ \pi\ x\ i) = (\sum\ i{\leq}n\ .\ \pi\ y\ i)$
    unfolding *less-eq-account-def*

```
    using dual-order.antisym by blast
  {
    fix n
    have π x n = π y n
    proof (cases n = 0)
      case True
      then show ?thesis using ⋆
        by (metis
              atMost-0
              empty-iff
              finite.intros(1)
              group-cancel.rule0
              sum.empty sum.insert)
    next
      case False
      from this obtain m where
        n = m + 1
        by (metis Nat.add-0-right Suc-eq-plus1 add-eq-if)
      have (∑ i≤n . π x i) = (∑ i≤n . π y i)
        using ⋆ by auto
      hence
        (∑ i≤m . π x i) + π x n =
         (∑ i≤m . π y i) + π y n
        using ‹n = m + 1›
        by simp
      moreover have (∑ i≤m . π x i) = (∑ i≤m . π y i)
        using ⋆ by auto
      ultimately show ?thesis by linarith
    qed
  }
  hence π x = π y by auto
  thus x = y
    by (metis Rep-account-inverse)
next
  fix x y z :: account
  assume x ≤ y
  {
    fix n :: nat
    have
      (∑ i≤n . π (z + x) i) =
       (∑ i≤n . π z i) + (∑ i≤n . π x i)
    and
      (∑ i≤n . π (z + y) i) =
       (∑ i≤n . π z i) + (∑ i≤n . π y i)
      unfolding Rep-account-plus
      by (simp add: sum.distrib)+
    moreover have (∑ i≤n . π x i) ≤ (∑ i≤n . π y i)
      using ‹x ≤ y›
      unfolding less-eq-account-def by blast
```

```
    ultimately have
      (∑ i≤n . π (z + x) i) ≤ (∑ i≤n . π (z + y) i)
      by linarith
  }
  thus z + x ≤ z + y
    unfolding
      less-eq-account-def by auto
qed
end
```

An account is strictly solvent exactly when it is *greater than or equal to 0*, according to the partial order just defined.

```
lemma strictly-solvent-alt-def: strictly-solvent α = (0 ≤ α)
  unfolding
    strictly-solvent-def
    less-eq-account-def
  using zero-account-def
  by force
```

3 Cash

The *cash reserve* in an account is the value under index 0.

Cash is treated with distinction. For instance it grows with interest (see §5). When we turn to balanced ledgers in §4.1, we will see that cash is the only quantity that does not cancel out.

```
definition cash-reserve :: account ⇒ real where
  cash-reserve α = π α 0
```

If α is strictly solvent then it has non-negative cash reserves.

```
lemma strictly-solvent-non-negative-cash:
  assumes strictly-solvent α
  shows 0 ≤ cash-reserve α
  using assms
  unfolding strictly-solvent-def cash-reserve-def
  by (metis
        atMost-0
        empty-iff
        finite.emptyI
        group-cancel.rule0
        sum.empty
        sum.insert)
```

An account consists of *just cash* when it has no other credit or debit other than under the first index.

```
definition just-cash :: real ⇒ account where
  just-cash c = ι (λ n . if n = 0 then c else 0)
```

lemma *Rep-account-just-cash* [*simp*]:
  π (*just-cash c*) = (λ *n* . *if n* = *0 then c else 0*)
proof(*cases c* = *0*)
  case *True*
  hence *just-cash c* = *0*
    unfolding *just-cash-def zero-account-def*
    by *force*
  then show *?thesis*
    using *Rep-account-zero True* **by** *force*
next
  case *False*
  hence *finite* (*support 0 UNIV* (λ *n* :: *nat* . *if n* = *0 then c else 0*))
    unfolding *support-def*
    by *auto*
  hence (λ *n* :: *nat* . *if n* = *0 then c else 0*) $\in$ *fin-support 0 UNIV*
    unfolding *fin-support-def*
    by *blast*
  then show *?thesis*
    unfolding *just-cash-def*
    using *Abs-account-inverse* **by** *auto*
qed

4 Ledgers

We model a *ledger* as a function from an index type *'a* to an *account*. A ledger could be thought of as an *indexed set* of accounts.

type-synonym *'a ledger* = *'a* $\Rightarrow$ *account*

4.1 Balanced Ledgers

We say a ledger is *balanced* when all of the debited and credited loans cancel, and all that is left is just cash.

Conceptually, given a balanced ledger we are justified in not tracking counterparty obligations.

definition (**in** *finite*) *balanced* :: *'a ledger* $\Rightarrow$ *real* $\Rightarrow$ *bool* **where**
  balanced $\mathcal{L}$ *c* $\equiv$ ($\sum$ *a* $\in$ *UNIV*. $\mathcal{L}$ *a*) = *just-cash c*

Provided the total cash is non-negative, a balanced ledger is a special case of a ledger which is globally strictly solvent.

lemma *balanced-strictly-solvent*:
  assumes *0* $\leq$ *c* **and** *balanced* $\mathcal{L}$ *c*
  shows *strictly-solvent* ($\sum$ *a* $\in$ *UNIV*. $\mathcal{L}$ *a*)
  using *assms*
  unfolding *balanced-def strictly-solvent-def*
  by *simp*

lemma (**in** *finite*) *finite-Rep-account-ledger* [*simp*]:
  $\pi\ (\sum\ a \in (A :: 'a\ set).\ \mathcal{L}\ a)\ n = (\sum\ a \in A.\ \pi\ (\mathcal{L}\ a)\ n)$
  using *finite*
  by (*induct A rule*: *finite-induct*, *auto*)

An alternate definition of balanced is that the *cash-reserve* for each account sums to c, and all of the other credited and debited assets cancels out.

lemma (**in** *finite*) *balanced-alt-def*:
  $balanced\ \mathcal{L}\ c =$
    $((\sum\ a \in UNIV.\ cash\text{-}reserve\ (\mathcal{L}\ a)) = c$
    $\wedge\ (\forall\ n > 0.\ (\sum\ a \in UNIV.\ \pi\ (\mathcal{L}\ a)\ n) = 0))$
  (**is** *?lhs* = *?rhs*)
proof (*rule iffI*)
  assume *?lhs*
  hence $(\sum\ a \in UNIV.\ cash\text{-}reserve\ (\mathcal{L}\ a)) = c$
    unfolding *balanced-def cash-reserve-def*
    by (*metis Rep-account-just-cash finite-Rep-account-ledger*)
  moreover
  {
    fix *n* :: *nat*
    assume $n > 0$
    with ‹*?lhs*› **have** $(\sum\ a \in UNIV.\ \pi\ (\mathcal{L}\ a)\ n) = 0$
      unfolding *balanced-def*
      by (*metis*
            Rep-account-just-cash
            less-nat-zero-code
            finite-Rep-account-ledger)
  }
  ultimately show *?rhs* **by** *auto*
next
  assume *?rhs*
  have $cash\text{-}reserve\ (just\text{-}cash\ c) = c$
    unfolding *cash-reserve-def*
    using *Rep-account-just-cash*
    by *presburger*
  also have ... $= (\sum a{\in}UNIV.\ cash\text{-}reserve\ (\mathcal{L}\ a))$ **using** ‹*?rhs*› **by** *auto*
  finally have
    $cash\text{-}reserve\ (\sum\ a \in UNIV.\ \mathcal{L}\ a) = cash\text{-}reserve\ (just\text{-}cash\ c)$
    unfolding *cash-reserve-def*
    by *auto*
  moreover
  {
    fix *n* :: *nat*
    assume $n > 0$
    hence $\pi\ (\sum\ a \in UNIV.\ \mathcal{L}\ a)\ n = 0$ **using** ‹*?rhs*› **by** *auto*
    hence $\pi\ (\sum\ a \in UNIV.\ \mathcal{L}\ a)\ n = \pi\ (just\text{-}cash\ c)\ n$
      unfolding *Rep-account-just-cash* **using** ‹$n > 0$› **by** *auto*
  }

ultimately have
  $\forall\ n\ .\ \pi\ (\sum\ a \in UNIV.\ \mathcal{L}\ a)\ n = \pi\ (just\text{-}cash\ c)\ n$
  unfolding *cash-reserve-def*
  by (*metis gr-zeroI*)
hence $\pi\ (\sum\ a \in UNIV.\ \mathcal{L}\ a) = \pi\ (just\text{-}cash\ c)$
  by *auto*
thus *?lhs*
  unfolding *balanced-def*
  using *Rep-account-inject*
  by *blast*
qed

4.2 Transfers

A *transfer amount* is the same as an *account*. It is just a function from *nat* to *real* with finite support.

type-synonym *transfer-amount* = *account*

When transferring between accounts in a ledger we make use of the Abelian group operations defined in §1.

definition *transfer* :: *'a ledger* ⇒ *transfer-amount* ⇒ *'a* ⇒ *'a* ⇒ *'a ledger* **where**
  $transfer\ \mathcal{L}\ \tau\ a\ b\ x = (if\ a = b\ then\ \mathcal{L}\ x$
    $else\ if\ x = a\ then\ \mathcal{L}\ a - \tau$
    $else\ if\ x = b\ then\ \mathcal{L}\ b + \tau$
    $else\ \mathcal{L}\ x)$

Transferring from an account to itself is a no-op.

lemma *transfer-collapse*:
  $transfer\ \mathcal{L}\ \tau\ a\ a = \mathcal{L}$
  unfolding *transfer-def* **by** *auto*

After a transfer, the sum totals of all credited and debited assets are preserved.

lemma (**in** *finite*) *sum-transfer-equiv*:
  fixes *x y* :: *'a*
  shows $(\sum\ a \in UNIV.\ \mathcal{L}\ a) = (\sum\ a \in UNIV.\ transfer\ \mathcal{L}\ \tau\ x\ y\ a)$
  (**is** - = $(\sum\ a \in UNIV.\ ?\mathcal{L}'\ a)$)
proof (*cases* $x = y$)
  case *True*
  show *?thesis*
    unfolding ‹$x = y$› *transfer-collapse* **..**
next
  case *False*
  let $?sum\text{-}\mathcal{L} = (\sum\ a \in UNIV - \{x,y\}.\ \mathcal{L}\ a)$
  let $?sum\text{-}\mathcal{L}' = (\sum\ a \in UNIV - \{x,y\}.\ ?\mathcal{L}'\ a)$
  have $\forall\ a \in UNIV - \{x,y\}.\ ?\mathcal{L}'\ a = \mathcal{L}\ a$
    by (*simp add*: *transfer-def*)
  hence $?sum\text{-}\mathcal{L}' = ?sum\text{-}\mathcal{L}$

by (*meson sum.cong*)
have $\{x,y\} \subseteq UNIV$ **by** *auto*
have $(\sum a \in UNIV.\ ?\mathcal{L}'\ a) = ?sum\text{-}\mathcal{L}' + (\sum a \in \{x,y\}.\ ?\mathcal{L}'\ a)$
using *finite-UNIV sum.subset-diff* [*OF* ‹$\{x,y\} \subseteq UNIV$›]
by *fastforce*
also have ... $= ?sum\text{-}\mathcal{L}' + ?\mathcal{L}'\ x + ?\mathcal{L}'\ y$
using
‹$x \neq y$›
finite
Diff-empty
Diff-insert-absorb
Diff-subset
group-cancel.add1
insert-absorb
sum.subset-diff
by (*simp add*: *insert-Diff-if*)
also have ... $= ?sum\text{-}\mathcal{L}' + \mathcal{L}\ x - \tau + \mathcal{L}\ y + \tau$
unfolding *transfer-def*
using ‹$x \neq y$›
by *auto*
also have ... $= ?sum\text{-}\mathcal{L}' + \mathcal{L}\ x + \mathcal{L}\ y$
by *simp*
also have ... $= ?sum\text{-}\mathcal{L} + \mathcal{L}\ x + \mathcal{L}\ y$
unfolding ‹$?sum\text{-}\mathcal{L}' = ?sum\text{-}\mathcal{L}$› **..**
also have ... $= ?sum\text{-}\mathcal{L} + (\sum a \in \{x,y\}.\ \mathcal{L}\ a)$
using
‹$x \neq y$›
finite
Diff-empty
Diff-insert-absorb
Diff-subset
group-cancel.add1
insert-absorb
sum.subset-diff
by (*simp add*: *insert-Diff-if*)
ultimately show *?thesis*
by (*metis local.finite sum.subset-diff top-greatest*)
qed

Since the sum totals of all credited and debited assets are preserved after transfer, a ledger is balanced if and only if it is balanced after transfer.

lemma (**in** *finite*) *balanced-transfer*:
balanced $\mathcal{L}$ c = *balanced* (*transfer* $\mathcal{L}$ τ a b) c
unfolding *balanced-def*
using *sum-transfer-equiv*
by *force*

Similarly, the sum total of a ledger is strictly solvent if and only if it is strictly solvent after transfer.

lemma (**in** *finite*) *strictly-solvent-transfer*:
  fixes $x\ y :: 'a$
  shows *strictly-solvent* $(\sum\ a \in UNIV.\ \mathcal{L}\ a) =$
          strictly-solvent $(\sum\ a \in UNIV.\ transfer\ \mathcal{L}\ \tau\ x\ y\ a)$
  using *sum-transfer-equiv*
  by *presburger*

4.3 The Valid Transfers Protocol

In this section we give a *protocol* for safely transferring value from one account to another.

We enforce that every transfer is *valid*. Valid transfers are intended to be intuitive. For instance one cannot transfer negative cash. Nor is it possible for an account that only has $50 to loan out $5,000,000.

A transfer is valid just in case the *transfer-amount* is strictly solvent and the account being credited the transfer will be strictly solvent afterwards.

definition *valid-transfer* :: *account* $\Rightarrow$ *transfer-amount* $\Rightarrow$ *bool* **where**
  valid-transfer $\alpha\ \tau$ = (*strictly-solvent* $\tau \wedge$ *strictly-solvent* $(\alpha - \tau)$)

lemma *valid-transfer-alt-def*: *valid-transfer* $\alpha\ \tau = (0 \leq \tau \wedge \tau \leq \alpha)$
  unfolding *valid-transfer-def strictly-solvent-alt-def*
  by *simp*

Only strictly solvent accounts can make valid transfers to begin with.

lemma *only-strictly-solvent-accounts-can-transfer*:
  assumes *valid-transfer* $\alpha\ \tau$
  shows *strictly-solvent* α
  using *assms*
  unfolding *strictly-solvent-alt-def valid-transfer-alt-def*
  by *auto*

We may now give a key result: accounts remain strictly solvent given a valid transfer.

theorem *strictly-solvent-still-strictly-solvent-after-valid-transfer*:
  assumes *valid-transfer* $(\mathcal{L}\ a)\ \tau$
  and *strictly-solvent* $(\mathcal{L}\ b)$
  shows
    strictly-solvent $((transfer\ \mathcal{L}\ \tau\ a\ b)\ a)$
    strictly-solvent $((transfer\ \mathcal{L}\ \tau\ a\ b)\ b)$
  using *assms*
  unfolding
    strictly-solvent-alt-def
    valid-transfer-alt-def
    transfer-def
  by (*cases* $a = b$, *auto*)

4.4 Embedding Conventional Cash-Only Ledgers

We show that in a sense the ledgers presented generalize conventional ledgers which only track cash.

An account consisting of just cash is strictly solvent if and only if it consists of a non-negative amount of cash.

lemma *strictly-solvent-just-cash-equiv*:
  strictly-solvent (*just-cash* c) = ($0 \leq c$)
  unfolding *strictly-solvent-def*
  using *Rep-account-just-cash just-cash-def* **by** *force*

An empty account corresponds to 0; the account with no cash or debit or credit.

lemma *zero-account-alt-def*: *just-cash* $0 = 0$
  unfolding *zero-account-def just-cash-def*
  by *simp*

Building on *just-cash* $0 = 0$, we have that *just-cash* is an embedding into an ordered subgroup. This means that *just-cash* is an order-preserving group homomorphism from the reals to the universe of accounts.

lemma *just-cash-embed*: ($a = b$) = (*just-cash* a = *just-cash* b)
proof (*rule iffI*)
  assume $a = b$
  thus *just-cash* a = *just-cash* b
    by *force*
next
  assume *just-cash* a = *just-cash* b
  hence *cash-reserve* (*just-cash* a) = *cash-reserve* (*just-cash* b)
    by *presburger*
  thus $a = b$
    unfolding *Rep-account-just-cash cash-reserve-def*
    by *auto*
qed

lemma *partial-nav-just-cash* [*simp*]:
 ($\sum$ $i \leq n$. π (*just-cash* a) i) = a
  unfolding *Rep-account-just-cash*
  by (*induct* n, *auto*)

lemma *just-cash-order-embed*: ($a \leq b$) = (*just-cash* $a \leq$ *just-cash* b)
  unfolding *less-eq-account-def*
  by *simp*

lemma *just-cash-plus* [*simp*]: *just-cash* a + *just-cash* b = *just-cash* ($a + b$)
proof −
  {
    fix x

```
    have π (just-cash a + just-cash b) x = π (just-cash (a + b)) x
    proof (cases x = 0)
      case True
      then show ?thesis
        using Rep-account-just-cash just-cash-def by force
    next
      case False
      then show ?thesis by simp
    qed
  }
  hence π (just-cash a + just-cash b) = π (just-cash (a + b))
    by auto
  thus ?thesis
    by (metis Rep-account-inverse)
qed

lemma just-cash-uminus [simp]: − just-cash a = just-cash (− a)
proof −
  {
    fix x
    have π (− just-cash a) x = π (just-cash (− a)) x
    proof (cases x = 0)
      case True
      then show ?thesis
        using Rep-account-just-cash just-cash-def by force
    next
      case False
      then show ?thesis by simp
    qed
  }
  hence π (− just-cash a) = π (just-cash (− a))
    by auto
  thus ?thesis
    by (metis Rep-account-inverse)
qed

lemma just-cash-subtract [simp]:
  just-cash a − just-cash b = just-cash (a − b)
  by (simp add: minus-account-def)
```

Valid transfers as per *valid-transfer* $?\alpha$ $?\tau$ = ($0 \leq ?\tau \wedge ?\tau \leq ?\alpha$) collapse into inequalities over the real numbers.

```
lemma just-cash-valid-transfer:
  valid-transfer (just-cash c) (just-cash t) = ((0 :: real) ≤ t ∧ t ≤ c)
  unfolding valid-transfer-alt-def
  by (simp add: less-eq-account-def)
```

Finally a ledger consisting of accounts with only cash is trivially *balanced*.

```
lemma (in finite) just-cash-summation:
```

fixes A :: $'a$ *set*
assumes $\forall\ a \in A.\ \exists\ c\ .\ \mathcal{L}\ a = \textit{just-cash}\ c$
shows $\exists\ c\ .\ (\sum\ a \in A\ .\ \mathcal{L}\ a) = \textit{just-cash}\ c$
using *finite assms*
by (*induct A rule*: *finite-induct*, *auto*, *metis zero-account-alt-def*)

lemma (**in** *finite*) *just-cash-UNIV-is-balanced*:
assumes $\forall\ a\ .\ \exists\ c\ .\ \mathcal{L}\ a = \textit{just-cash}\ c$
shows $\exists\ c\ .\ \textit{balanced}\ \mathcal{L}\ c$
unfolding *balanced-def*
using
assms
just-cash-summation [**where** *A*=*UNIV*]
by *simp*

5 Interest

In this section we discuss how to calculate the interest accrued by an account for a period. This is done by looking at the sum of all of the credit and debit in an account. This sum is called the *net asset value* of an account.

5.1 Net Asset Value

The net asset value of an account is the sum of all of the non-zero entries. Since accounts have finite support this sum is always well defined.

definition *net-asset-value* :: *account* $\Rightarrow$ *real* **where**
$\textit{net-asset-value}\ \alpha = (\sum\ i \mid \pi\ \alpha\ i \neq 0\ .\ \pi\ \alpha\ i)$

5.1.1 The Shortest Period for Credited & Debited Assets in an Account

Higher indexes for an account correspond to shorter loan periods. Since accounts only have a finite number of entries, it makes sense to talk about the *shortest* period an account has an entry for. The net asset value for an account has a simpler expression in terms of that account's shortest period.

definition *shortest-period* :: *account* $\Rightarrow$ *nat* **where**
$\textit{shortest-period}\ \alpha =$
$(\textit{if}\ (\forall\ i.\ \pi\ \alpha\ i = 0)$
$\textit{then}\ 0$
$\textit{else}\ \textit{Max}\ \{i\ .\ \pi\ \alpha\ i \neq 0\})$

lemma *shortest-period-uminus*:
$\textit{shortest-period}\ (-\ \alpha) = \textit{shortest-period}\ \alpha$
unfolding *shortest-period-def*
using *Rep-account-uminus uminus-account-def*
by *force*

lemma *finite-account-support*:
  $\mathit{finite}\ \{i\ .\ \pi\ \alpha\ i \neq \mathit{0}\}$
proof $-$
  have $\pi\ \alpha \in \mathit{fin\text{-}support}\ \mathit{0}\ \mathit{UNIV}$
    by (*simp add*: *Rep-account*)
  thus *?thesis*
    unfolding *fin-support-def support-def*
    by *fastforce*
qed

lemma *shortest-period-plus*:
  $\mathit{shortest\text{-}period}\ (\alpha + \beta) \leq \mathit{max}\ (\mathit{shortest\text{-}period}\ \alpha)\ (\mathit{shortest\text{-}period}\ \beta)$
  (**is** - $\leq$ *?MAX*)
proof $(\mathit{cases}\ \forall\ i\ .\ \pi\ (\alpha + \beta)\ i = \mathit{0})$
  case *True*
  then show *?thesis* **unfolding** *shortest-period-def* **by** *auto*
next
  case *False*
  have $\mathit{shortest\text{-}period}\ \alpha \leq \mathit{?MAX}$ **and** $\mathit{shortest\text{-}period}\ \beta \leq \mathit{?MAX}$
    by *auto*
  moreover
  have $\forall\ i > \mathit{shortest\text{-}period}\ \alpha\ .\ \pi\ \alpha\ i = \mathit{0}$
  and $\forall\ i > \mathit{shortest\text{-}period}\ \beta\ .\ \pi\ \beta\ i = \mathit{0}$
    unfolding *shortest-period-def*
    using *finite-account-support Max.coboundedI leD Collect-cong*
    by *auto*
  ultimately
  have $\forall\ i > \mathit{?MAX}\ .\ \pi\ \alpha\ i = \mathit{0}$
  and $\forall\ i > \mathit{?MAX}\ .\ \pi\ \beta\ i = \mathit{0}$
    by *simp+*
  hence $\forall\ i > \mathit{?MAX}\ .\ \pi\ (\alpha + \beta)\ i = \mathit{0}$
    by *simp*
  hence $\forall\ i\ .\ \pi\ (\alpha + \beta)\ i \neq \mathit{0} \longrightarrow i \leq \mathit{?MAX}$
    by (*meson not-le*)
  thus *?thesis*
    unfolding *shortest-period-def*
    using
      finite-account-support [**where** $\alpha = \alpha + \beta$]
      False
    by *simp*
qed

lemma *shortest-period-*π:
  assumes $\pi\ \alpha\ i \neq \mathit{0}$
  shows $\pi\ \alpha\ (\mathit{shortest\text{-}period}\ \alpha) \neq \mathit{0}$
proof $-$
  let $\mathit{?support} = \{i\ .\ \pi\ \alpha\ i \neq \mathit{0}\}$
  have *A*: *finite ?support*

```
    using finite-account-support by blast
  have B: ?support ≠ {} using assms by auto
  have shortest-period α = Max ?support
    using assms
    unfolding shortest-period-def
    by auto
  have shortest-period α ∈ ?support
    unfolding ‹shortest-period α = Max ?support›
    using Max-in [OF A B] by auto
  thus ?thesis
    by auto
qed

lemma shortest-period-bound:
  assumes π α i ≠ 0
  shows i ≤ shortest-period α
proof −
  let ?support = {i . π α i ≠ 0}
  have shortest-period α = Max ?support
    using assms
    unfolding shortest-period-def
    by auto
  have shortest-period α ∈ ?support
    using assms shortest-period-π by force
  thus ?thesis
    unfolding ‹shortest-period α = Max ?support›
    by (simp add: assms finite-account-support)
qed
```

Using *shortest-period* we may give an alternate definition for *net-asset-value*.

```
lemma net-asset-value-alt-def:
  net-asset-value α = (∑ i ≤ shortest-period α. π α i)
proof −
  let ?support = {i . π α i ≠ 0}
  {
    fix k
    have (∑ i | i ≤ k ∧ π α i ≠ 0 . π α i) = (∑ i ≤ k. π α i)
    proof (induct k)
      case 0
      thus ?case
      proof (cases π α 0 = 0)
        case True
        then show ?thesis
          by fastforce
      next
        case False
        {
          fix i
          have (i ≤ 0 ∧ π α i ≠ 0) = (i ≤ 0)
```

using *False*
by *blast*
}
hence $(\sum\ i \mid i \leq 0 \wedge \pi\ \alpha\ i \neq 0.\ \pi\ \alpha\ i) =$
$(\sum i \mid i \leq 0.\ \pi\ \alpha\ i)$
by *presburger*
also have ... $= (\sum i \leq 0.\ \pi\ \alpha\ i)$
by *simp*
ultimately show *?thesis*
by *simp*
qed
next
case $(Suc\ k)$
then show *?case*
proof $(cases\ \pi\ \alpha\ (Suc\ k) = 0)$
case *True*
{
fix i
have $(i \leq Suc\ k \wedge \pi\ \alpha\ i \neq 0) =$
$(i \leq k \wedge \pi\ \alpha\ i \neq 0)$
using *True le-Suc-eq* **by** *blast*
}
hence $(\sum i \mid i \leq Suc\ k \wedge \pi\ \alpha\ i \neq 0.\ \pi\ \alpha\ i) =$
$(\sum i \mid i \leq k \wedge \pi\ \alpha\ i \neq 0.\ \pi\ \alpha\ i)$
by *presburger*
also have ... $= (\sum\ i \leq k.\ \pi\ \alpha\ i)$
using *Suc* **by** *blast*
ultimately show *?thesis* **using** *True*
by *simp*
next
let $?A = \{i\ .\ i \leq Suc\ k \wedge \pi\ \alpha\ i \neq 0\}$
let $?A' = \{i\ .\ i \leq k \wedge \pi\ \alpha\ i \neq 0\}$
case *False*
hence $?A = \{i\ .\ (i \leq k \wedge \pi\ \alpha\ i \neq 0) \vee i = Suc\ k\}$
by *auto*
hence $?A = ?A' \cup \{i\ .\ i = Suc\ k\}$
by (*simp add*: *Collect-disj-eq*)
hence $\star$: $?A = ?A' \cup \{Suc\ k\}$
by *simp*
hence $\heartsuit$: *finite* $(?A' \cup \{Suc\ k\})$
using *finite-nat-set-iff-bounded-le*
by *blast*
hence
$(\sum i \mid i \leq Suc\ k \wedge \pi\ \alpha\ i \neq 0.\ \pi\ \alpha\ i) =$
$(\sum\ i \in ?A' \cup \{Suc\ k\}.\ \pi\ \alpha\ i)$
unfolding $\star$
by *auto*
also have ... $= (\sum\ i \in ?A'.\ \pi\ \alpha\ i) + (\sum\ i \in \{Suc\ k\}.\ \pi\ \alpha\ i)$
using $\heartsuit$

```
        by force
      also have ... = (∑ i ∈ ?A'. π α i) + π α (Suc k)
        by simp
      ultimately show ?thesis
        by (simp add: Suc)
    qed
  qed
}
hence †:
  (∑ i | i ≤ shortest-period α ∧ π α i ≠ 0. π α i) =
        (∑ i ≤ shortest-period α. π α i)
  by auto
{
  fix i
  have (i ≤ shortest-period α ∧ π α i ≠ 0) = (π α i ≠ 0)
    using shortest-period-bound by blast
}
note · = this
show ?thesis
  using †
  unfolding · net-asset-value-def
  by blast
qed

lemma greater-than-shortest-period-zero:
  assumes shortest-period α < m
  shows π α m = 0
proof −
  let ?support = {i . π α i ≠ 0}
  have ∀ i ∈ ?support . i ≤ shortest-period α
    by (simp add: finite-account-support shortest-period-def)
  then show ?thesis
    using assms
    by (meson CollectI leD)
qed
```

An account's *net-asset-value* does not change when summing beyond its *shortest-period*. This is helpful when computing aggregate net asset values across multiple accounts.

```
lemma net-asset-value-shortest-period-ge:
  assumes shortest-period α ≤ n
  shows net-asset-value α = (∑ i ≤ n. π α i)
proof (cases shortest-period α = n)
  case True
  then show ?thesis
    unfolding net-asset-value-alt-def by auto
next
  case False
  hence shortest-period α < n using assms by auto
```

hence $(\sum\ i{=}shortest\text{-}period\ \alpha + 1..\ n.\ \pi\ \alpha\ i) = 0$
(**is** $?\Sigma extra = 0$)
using *greater-than-shortest-period-zero*
by *auto*
moreover have $(\sum\ i \leq n.\ \pi\ \alpha\ i) =$
$(\sum\ i \leq shortest\text{-}period\ \alpha.\ \pi\ \alpha\ i) + ?\Sigma extra$
(**is** $?lhs = ?\Sigma shortest\text{-}period + \text{-}$)
by (*metis*
‹*shortest-period* $\alpha < n$›
Suc-eq-plus1
less-imp-add-positive
sum-up-index-split)
ultimately have $?lhs = ?\Sigma shortest\text{-}period$
by *linarith*
then show *?thesis*
unfolding *net-asset-value-alt-def* **by** *auto*
qed

5.1.2 Net Asset Value Properties

In this section we explore how *net-asset-value* forms an order-preserving group homomorphism from the universe of accounts to the real numbers.

We first observe that *strictly-solvent* implies the more conventional notion of solvent, where an account's net asset value is non-negative.

lemma *strictly-solvent-net-asset-value*:
assumes *strictly-solvent* α
shows $0 \leq$ *net-asset-value* α
using *assms strictly-solvent-def net-asset-value-alt-def* **by** *auto*

Next we observe that *net-asset-value* is a order preserving group homomorphism from the universe of accounts to *real*.

lemma *net-asset-value-zero* [*simp*]: *net-asset-value* $0 = 0$
unfolding *net-asset-value-alt-def*
using *zero-account-def* **by** *force*

lemma *net-asset-value-mono*:
assumes $\alpha \leq \beta$
shows *net-asset-value* $\alpha \leq$ *net-asset-value* β
proof −
let $?r = max\ (shortest\text{-}period\ \alpha)\ (shortest\text{-}period\ \beta)$
have *shortest-period* $\alpha \leq ?r$ **and** *shortest-period* $\beta \leq ?r$ **by** *auto*
hence *net-asset-value* $\alpha = (\sum\ i \leq ?r.\ \pi\ \alpha\ i)$
and *net-asset-value* $\beta = (\sum\ i \leq ?r.\ \pi\ \beta\ i)$
using *net-asset-value-shortest-period-ge* **by** *presburger+*
thus *?thesis* **using** *assms* **unfolding** *less-eq-account-def* **by** *auto*
qed

lemma *net-asset-value-uminus*: *net-asset-value* $(- \alpha) = -$ *net-asset-value* α
  unfolding
    net-asset-value-alt-def
    shortest-period-uminus
    Rep-account-uminus
  by (*simp add*: *sum-negf*)

lemma *net-asset-value-plus*:
  net-asset-value $(\alpha + \beta) =$ *net-asset-value* $\alpha +$ *net-asset-value* β
  (**is** $?lhs = ?\Sigma\alpha + ?\Sigma\beta$)
proof $-$
  let $?r = max$ (*shortest-period* α) (*shortest-period* β)
  have A: *shortest-period* $(\alpha + \beta) \leq ?r$
    and B: *shortest-period* $\alpha \leq ?r$
    and C: *shortest-period* $\beta \leq ?r$
    using *shortest-period-plus* **by** *presburger+*
  have $?lhs = (\sum i \leq ?r.\ \pi\ (\alpha + \beta)\ i)$
    using *net-asset-value-shortest-period-ge* [*OF A*] **.**
  also have $\ldots = (\sum i \leq ?r.\ \pi\ \alpha\ i + \pi\ \beta\ i)$
    using *Rep-account-plus* **by** *presburger*
  ultimately show *?thesis*
    using
      net-asset-value-shortest-period-ge [*OF B*]
      net-asset-value-shortest-period-ge [*OF C*]
    by (*simp add*: *sum.distrib*)
qed

lemma *net-asset-value-minus*:
  net-asset-value $(\alpha - \beta) =$ *net-asset-value* $\alpha -$ *net-asset-value* β
  using *additive.diff additive.intro net-asset-value-plus* **by** *blast*

Finally we observe that *just-cash* is the right inverse of *net-asset-value*.

lemma *net-asset-value-just-cash-left-inverse*:
  net-asset-value (*just-cash* c) $= c$
  using *net-asset-value-alt-def partial-nav-just-cash* **by** *presburger*

5.2 Distributing Interest

We next show that the total interest accrued for a ledger at distribution does not change when one account makes a transfer to another.

definition (**in** *finite*) *total-interest* :: $'a$ *ledger* $\Rightarrow$ *real* $\Rightarrow$ *real*
  where *total-interest* $\mathcal{L}\ i = (\sum a \in UNIV.\ i * $ *net-asset-value* $(\mathcal{L}\ a))$

lemma (**in** *finite*) *total-interest-transfer*:
  total-interest (*transfer* $\mathcal{L}\ \tau\ a\ b$) $i =$ *total-interest* $\mathcal{L}\ i$
  (**is** *total-interest* $?\mathcal{L}'\ i$ = -)
proof (*cases* $a = b$)
  case *True*

```
  show ?thesis
    unfolding ‹a = b› transfer-collapse ..
next
  case False
  have total-interest ?ℒ' i = (∑ a ∈ UNIV . i ∗ net-asset-value (?ℒ' a))
    unfolding total-interest-def ..
  also have ... = (∑ a ∈ UNIV − {a, b} ∪ {a,b}. i ∗ net-asset-value (?ℒ' a))
    by (metis Un-Diff-cancel2 Un-UNIV-left)
  also have ... = (∑ a ∈ UNIV − {a, b}. i ∗ net-asset-value (?ℒ' a)) +
                  i ∗ net-asset-value (?ℒ' a) + i ∗ net-asset-value (?ℒ' b)
    (is - = ?Σ + - + -)
    using ‹a ≠ b›
    by simp
  also have ... = ?Σ +
                   i ∗ net-asset-value (ℒ a − τ) +
                   i ∗ net-asset-value (ℒ b + τ)
    unfolding transfer-def
    using ‹a ≠ b›
    by auto
  also have ... = ?Σ +
                 i ∗ net-asset-value (ℒ a) +
                 i ∗ net-asset-value (− τ) +
                 i ∗ net-asset-value (ℒ b) +
                 i ∗ net-asset-value τ
    unfolding minus-account-def net-asset-value-plus
    by (simp add: distrib-left)
  also have ... = ?Σ +
                 i ∗ net-asset-value (ℒ a) +
                 i ∗ net-asset-value (ℒ b)
    unfolding net-asset-value-uminus
    by linarith
  also have ... = (∑ a ∈ UNIV − {a, b}. i ∗ net-asset-value (ℒ a)) +
                 i ∗ net-asset-value (ℒ a) +
                 i ∗ net-asset-value (ℒ b)
    unfolding transfer-def
    using ‹a ≠ b›
    by force
  also have ... = (∑ a ∈ UNIV − {a, b} ∪ {a,b}. i ∗ net-asset-value (ℒ a))
    using ‹a ≠ b› by force
  ultimately show ?thesis
    unfolding total-interest-def
    by (metis Diff-partition Un-commute top-greatest)
qed
```

6 Update

Periodically the ledger is *updated*. When this happens interest is distributed and loans are returned. Each time loans are returned, a fixed fraction of each loan for each period is returned.

The fixed fraction for returned loans is given by a *rate function*. We denote rate functions with ϱ::*nat* $\Rightarrow$ *real*. In principle this function obeys the rules:

- $\varrho\ 0 = 0$ – Cash is not returned.
- $\forall\, n.\ \varrho\ n < 1$ – The fraction of a loan returned never exceeds 1.
- $\forall\, n\ m.\ n < m \longrightarrow \varrho\ n < \varrho\ m$ – Higher indexes correspond to shorter loan periods. This in turn corresponds to a higher fraction of loans returned at update for higher indexes.

In practice, rate functions determine the time it takes for 99% of the loan to be returned. However, the presentation here abstracts away from time. In §7.3 we establish a closed form for updating. This permits for a production implementation to efficiently (albeit *lazily*) update ever *millisecond* if so desired.

definition *return-loans* :: (*nat* $\Rightarrow$ *real*) $\Rightarrow$ *account* $\Rightarrow$ *account* **where**
  return-loans $\varrho\ \alpha = \iota\ (\lambda\ n\ .\ (1 - \varrho\ n) * \pi\ \alpha\ n)$

lemma *Rep-account-return-loans* [*simp*]:
  π (*return-loans* $\varrho\ \alpha$) $= (\lambda\ n\ .\ (1 - \varrho\ n) * \pi\ \alpha\ n)$
proof –
  have (*support 0 UNIV* $(\lambda\ n\ .\ (1 - \varrho\ n) * \pi\ \alpha\ n)) \subseteq$
        (*support 0 UNIV* $(\pi\ \alpha)$)
    unfolding *support-def*
    by (*simp add*: *Collect-mono*)
  moreover have *finite* (*support 0 UNIV* $(\pi\ \alpha)$)
    using *Rep-account*
    unfolding *fin-support-def* **by** *auto*
  ultimately have *finite* (*support 0 UNIV* $(\lambda\ n\ .\ (1 - \varrho\ n) * \pi\ \alpha\ n)$)
    using *infinite-super* **by** *blast*
  hence $(\lambda\ n\ .\ (1 - \varrho\ n) * \pi\ \alpha\ n) \in$ *fin-support 0 UNIV*
    unfolding *fin-support-def* **by** *auto*
  thus *?thesis*
    using
      Rep-account
      Abs-account-inject
      Rep-account-inverse
      return-loans-def
    by *auto*
qed

As discussed, updating an account involves distributing interest and returning its credited and debited loans.

definition *update-account* :: (*nat* $\Rightarrow$ *real*) $\Rightarrow$ *real* $\Rightarrow$ *account* $\Rightarrow$ *account* **where**

update-account ϱ *i* α = *just-cash* (*i* $*$ *net-asset-value* α) + *return-loans* ϱ α

definition *update-ledger* :: (*nat* $\Rightarrow$ *real*) $\Rightarrow$ *real* $\Rightarrow$ *'a ledger* $\Rightarrow$ *'a ledger*
where
update-ledger ϱ *i* $\mathcal{L}$ *a* = *update-account* ϱ *i* ($\mathcal{L}$ *a*)

6.1 Update Preserves Ledger Balance

A key theorem is that if all credit and debit in a ledger cancel, they will continue to cancel after update. In this sense the monetary supply grows with the interest rate, but is otherwise conserved.

A consequence of this theorem is that while counter-party obligations are not explicitly tracked by the ledger, these obligations are fulfilled as funds are returned by the protocol.

definition *shortest-ledger-period* :: *'a ledger* $\Rightarrow$ *nat* **where**
shortest-ledger-period $\mathcal{L}$ = *Max* (*image shortest-period* (*range* $\mathcal{L}$))

lemma (**in** *finite*) *shortest-ledger-period-bound*:
fixes $\mathcal{L}$ :: *'a ledger*
shows *shortest-period* ($\mathcal{L}$ *a*) $\leq$ *shortest-ledger-period* $\mathcal{L}$
proof −
{
fix α :: *account*
fix *S* :: *account set*
assume *finite S* **and** $\alpha \in S$
hence *shortest-period* $\alpha \leq$ *Max* (*shortest-period* ' *S*)
proof (*induct S rule*: *finite-induct*)
case *empty*
then show *?case* **by** *auto*
next
case (*insert* β *S*)
then show *?case*
proof (*cases* $\alpha = \beta$)
case *True*
then show *?thesis*
by (*simp add*: *insert.hyps*(*1*))
next
case *False*
hence $\alpha \in S$
using *insert.prems* **by** *fastforce*
then show *?thesis*
by (*meson*
Max-ge
finite-imageI
finite-insert
imageI
insert.hyps(*1*)
insert.prems)

```
      qed
    qed
  }
  moreover
  have finite (range ℒ)
    by force
  ultimately show ?thesis
    by (simp add: shortest-ledger-period-def)
qed

theorem (in finite) update-balanced:
  assumes ϱ 0 = 0 and ∀ n. ϱ n < 1 and 0 ≤ i
  shows balanced ℒ c = balanced (update-ledger ϱ i ℒ) ((1 + i) ∗ c)
    (is - = balanced ?ℒ' ((1 + i) ∗ c))
proof
  assume balanced ℒ c
  have ∀ n>0. (∑ a∈UNIV. π (?ℒ' a) n) = 0
  proof (rule allI, rule impI)
    fix n :: nat
    assume n > 0
    {
      fix a
      let ?α' = λn. (1 − ϱ n) ∗ π (ℒ a) n
      have π (?ℒ' a) n = ?α' n
        unfolding
          update-ledger-def
          update-account-def
          Rep-account-plus
          Rep-account-just-cash
          Rep-account-return-loans
        using plus-account-def ‹n > 0›
        by simp
    }
    hence (∑ a∈UNIV. π (?ℒ' a) n) =
            (1 − ϱ n) ∗ (∑ a∈UNIV. π (ℒ a) n)
      using finite-UNIV
      by (metis (mono-tags, lifting) sum.cong sum-distrib-left)
    thus (∑ a∈UNIV. π (?ℒ' a) n) = 0
      using ‹0 < n› ‹balanced ℒ c› local.balanced-alt-def by force
  qed
  moreover
  {
    fix S :: 'a set
    let ?ω = shortest-ledger-period ℒ
    assume (∑ a∈S. cash-reserve (ℒ a)) = c
    and ∀ n>0. (∑ a∈S. π (ℒ a) n) = 0
    have (∑ a∈S. cash-reserve (?ℒ' a)) =
             (∑ a∈S. i ∗ (∑ n ≤ ?ω. π (ℒ a) n) +
                 cash-reserve (ℒ a))
```

```
    using finite
    proof (induct S arbitrary: c rule: finite-induct)
      case empty
      then show ?case
        by auto
    next
      case (insert x S)
      have (∑ a∈insert x S. cash-reserve (?ℒ' a)) =
              (∑ a∈insert x S. i * (∑ n ≤ ?ω. π (ℒ a) n) +
                  cash-reserve (ℒ a))
        unfolding update-ledger-def update-account-def cash-reserve-def
        by (simp add: ‹ϱ 0 = 0›,
            metis (no-types)
                  shortest-ledger-period-bound
                  net-asset-value-shortest-period-ge)
      thus ?case .
    qed
    also have ... = (∑ a∈S. i * (∑ n = 1 .. ?ω. π (ℒ a) n) +
                          i * cash-reserve (ℒ a) + cash-reserve (ℒ a))
      unfolding cash-reserve-def
      by (simp add:
            add.commute
            distrib-left
            sum.atMost-shift
            sum-bounds-lt-plus1)
    also have ... = (∑ a∈S. i * (∑ n = 1 .. ?ω. π (ℒ a) n) +
                         (1 + i) * cash-reserve (ℒ a))
      using finite
      by (induct S rule: finite-induct, auto, simp add: distrib-right)
    also have ... = i * (∑ a∈S. (∑ n = 1 .. ?ω. π (ℒ a) n)) +
                         (1 + i) * (∑ a∈S. cash-reserve (ℒ a))
      by (simp add: sum.distrib sum-distrib-left)
    also have ... = i * (∑ n = 1 .. ?ω. (∑ a∈S. π (ℒ a) n)) +
                         (1 + i) * c
      using ‹(∑ a∈S. cash-reserve (ℒ a)) = c› sum.swap by force
    finally have (∑ a∈S. cash-reserve (?ℒ' a)) = c * (1 + i)
      using ‹∀ n>0. (∑ a∈S. π (ℒ a) n) = 0›
      by simp
  }
  hence (∑ a∈UNIV. cash-reserve (?ℒ' a)) = c * (1 + i)
    using ‹balanced ℒ c›
    unfolding balanced-alt-def
    by fastforce
  ultimately show balanced ?ℒ' ((1 + i) * c)
    unfolding balanced-alt-def
    by auto
next
  assume balanced ?ℒ' ((1 + i) * c)
  have ⋆: ∀ n>0. (∑ a∈UNIV. π (ℒ a) n) = 0
```

```
proof (rule allI, rule impI)
  fix n :: nat
  assume n > 0
  hence 0 = (∑ a∈UNIV. π (?ℒ' a) n)
    using ‹balanced ?ℒ' ((1 + i) * c)›
    unfolding balanced-alt-def
    by auto
  also have ... = (∑ a∈UNIV. (1 − ϱ n) * π (ℒ a) n)
    unfolding
      update-ledger-def
      update-account-def
      Rep-account-return-loans
      Rep-account-just-cash
    using ‹n > 0›
    by auto
  also have ... = (1 − ϱ n) * (∑ a∈UNIV. π (ℒ a) n)
    by (simp add: sum-distrib-left)
  finally show (∑ a∈UNIV. π (ℒ a) n) = 0
    by (metis
          ‹∀ r . ϱ r < 1›
          diff-gt-0-iff-gt
          less-numeral-extra(3)
          mult-eq-0-iff)
qed
moreover
{
  fix S :: 'a set
  let ?ω = shortest-ledger-period ℒ
  assume (∑ a∈S. cash-reserve (?ℒ' a)) = (1 + i) * c
  and ∀ n>0. (∑ a∈S. π (ℒ a) n) = 0
  hence (1 + i) * c = (∑ a∈S. cash-reserve (?ℒ' a))
    by auto
  also have ... = (∑ a∈S. i * (∑ n ≤ ?ω. π (ℒ a) n) +
                  cash-reserve (ℒ a))
  using finite
  proof (induct S rule: finite-induct)
    case empty
    then show ?case
      by auto
  next
    case (insert x S)
    have (∑ a∈insert x S. cash-reserve (?ℒ' a)) =
          (∑ a∈insert x S.
              i * (∑ n ≤ ?ω. π (ℒ a) n) + cash-reserve (ℒ a))
      unfolding update-ledger-def update-account-def cash-reserve-def
      by (simp add: ‹ϱ 0 = 0›,
          metis (no-types)
                shortest-ledger-period-bound
                net-asset-value-shortest-period-ge)
```

```
    thus ?case .
  qed
  also have ... = (∑ a∈S. i * (∑ n = 1 .. ?ω. π (ℒ a) n) +
                          i * cash-reserve (ℒ a) + cash-reserve (ℒ a))
    unfolding cash-reserve-def
    by (simp add:
          add.commute
          distrib-left
          sum.atMost-shift
          sum-bounds-lt-plus1)
  also have ... = (∑ a∈S. i * (∑ n = 1 .. ?ω. π (ℒ a) n) +
                       (1 + i) * cash-reserve (ℒ a))
    using finite
    by (induct S rule: finite-induct, auto, simp add: distrib-right)
  also have ... = i * (∑ a∈S. (∑ n = 1 .. ?ω. π (ℒ a) n)) +
                     (1 + i) * (∑ a∈S. cash-reserve (ℒ a))
    by (simp add: sum.distrib sum-distrib-left)
  also have ... =  i * (∑ n = 1 .. ?ω. (∑ a∈S. π (ℒ a) n)) +
                    (1 + i) * (∑ a∈S. cash-reserve (ℒ a))
    using sum.swap by force
  also have ... = (1 + i) * (∑ a∈S. cash-reserve (ℒ a))
    using ‹∀ n>0. (∑ a∈S. π (ℒ a) n) = 0›
    by simp
  finally have c = (∑ a∈S. cash-reserve (ℒ a))
    using ‹0 ≤ i›
    by force
}
hence (∑ a∈UNIV. cash-reserve (ℒ a)) = c
  unfolding cash-reserve-def
  by (metis
        Rep-account-just-cash
        ‹balanced ?ℒ' ((1 + i) * c)›
        ⋆
        balanced-def
        finite-Rep-account-ledger)
ultimately show balanced ℒ c
  unfolding balanced-alt-def
  by auto
qed
```

6.2 Strictly Solvent is Forever Strictly Solvent

The final theorem presented in this section is that if an account is strictly solvent, it will still be strictly solvent after update.

This theorem is the key to how the system avoids counter party risk. Provided the system enforces that all accounts are strictly solvent and transfers are *valid* (as discussed in §4.2), all accounts will remain strictly solvent forever.

We first prove that *return-loans* is a group homomorphism.

It is order preserving given certain assumptions.

lemma *return-loans-plus*:
  return-loans ϱ $(\alpha + \beta)$ = *return-loans* ϱ α + *return-loans* ϱ β
proof –
  let $?\alpha = \pi\ \alpha$
  let $?\beta = \pi\ \beta$
  let $?\varrho\alpha\beta = \lambda n.\ (1 - \varrho\ n) * (?\alpha\ n + ?\beta\ n)$
  let $?\varrho\alpha = \lambda n.\ (1 - \varrho\ n) * ?\alpha\ n$
  let $?\varrho\beta = \lambda n.\ (1 - \varrho\ n) * ?\beta\ n$
  have *support 0 UNIV* $?\varrho\alpha \subseteq$ *support 0 UNIV* $?\alpha$
      support 0 UNIV $?\varrho\beta \subseteq$ *support 0 UNIV* $?\beta$
      support 0 UNIV $?\varrho\alpha\beta \subseteq$ *support 0 UNIV* $?\alpha \cup$ *support 0 UNIV* $?\beta$
    unfolding *support-def*
    by *auto*
  moreover have
    $?\alpha \in$ *fin-support 0 UNIV*
    $?\beta \in$ *fin-support 0 UNIV*
    using *Rep-account* **by** *force+*
  ultimately have $\star$:
    $?\varrho\alpha \in$ *fin-support 0 UNIV*
    $?\varrho\beta \in$ *fin-support 0 UNIV*
    $?\varrho\alpha\beta \in$ *fin-support 0 UNIV*
    unfolding *fin-support-def*
    using *finite-subset* **by** *auto+*
  {
    fix n
    have π (*return-loans* ϱ $(\alpha + \beta)$) n =
        π (*return-loans* ϱ α + *return-loans* ϱ β) n
      unfolding *return-loans-def Rep-account-plus*
      using $\star$ *Abs-account-inverse distrib-left* **by** *auto*
  }
  hence π (*return-loans* ϱ $(\alpha + \beta)$) =
        π (*return-loans* ϱ α + *return-loans* ϱ β)
    by *auto*
  thus *?thesis*
    by (*metis Rep-account-inverse*)
qed

lemma *return-loans-zero* [*simp*]: *return-loans* ϱ $0 = 0$
proof –
  have $(\lambda n.\ (1 - \varrho\ n) * 0) = (\lambda\text{-}.\ 0)$
    by *force*
  hence $\iota\ (\lambda n.\ (1 - \varrho\ n) * 0) = 0$
    unfolding *zero-account-def*
    by *presburger*
  thus *?thesis*
    unfolding *return-loans-def Rep-account-zero* **.**

qed

lemma *return-loans-uminus*: *return-loans* ϱ $(-\ \alpha) = -$ *return-loans* ϱ α
  by (*metis*
    add.left-cancel
    diff-self
    minus-account-def
    return-loans-plus
    return-loans-zero)

lemma *return-loans-subtract*:
  return-loans ϱ $(\alpha - \beta)$ = *return-loans* ϱ α − *return-loans* ϱ β
  by (*simp add*: *additive.diff additive-def return-loans-plus*)

As presented in §1, each index corresponds to a progressively shorter loan period. This is captured by a monotonicity assumption on the rate function ϱ::*nat* ⇒ *real*. In particular, provided $\forall\, n.\ \varrho\ n < 1$ and $\forall\, n\ m.\ n < m \longrightarrow \varrho\ n < \varrho\ m$ then we know that all outstanding credit is going away faster than loans debited for longer periods.

Given the monotonicity assumptions for a rate function ϱ::*nat* ⇒ *real*, we may in turn prove monotonicity for *return-loans* over (≤)::*account* ⇒ *account* ⇒ *bool*.

lemma *return-loans-mono*:
  assumes $\forall\ n\ .\ \varrho\ n < 1$
  and $\forall\ n\ m\ .\ n \leq m \longrightarrow \varrho\ n \leq \varrho\ m$
  and $\alpha \leq \beta$
  shows *return-loans* ϱ α ≤ *return-loans* ϱ β
proof −
  {
    fix α :: *account*
    assume $0 \leq \alpha$
    {
      fix n :: *nat*
      let $?\alpha = \pi\ \alpha$
      let $?\varrho\alpha = \lambda n.\ (1 - \varrho\ n) * ?\alpha\ n$
      have $\forall\ n\ .\ 0 \leq (\sum\ i{\leq}n\ .\ ?\alpha\ i)$
        using ‹$0 \leq \alpha$›
        unfolding *less-eq-account-def Rep-account-zero*
        by *simp*
      hence $0 \leq (\sum\ i{\leq}n\ .\ ?\alpha\ i)$ **by** *auto*
      moreover have $(1 - \varrho\ n) * (\sum\ i{\leq}n\ .\ ?\alpha\ i) \leq (\sum\ i{\leq}n\ .\ ?\varrho\alpha\ i)$
      proof (*induct n*)
        case *0*
        then show *?case* **by** *simp*
      next
        case (*Suc n*)
        have $0 \leq (1 - \varrho\ (Suc\ n))$
          by (*simp add*: ‹$\forall\ n\ .\ \varrho\ n < 1$› *less-eq-real-def*)

```
        moreover have (1 − ϱ (Suc n)) ≤ (1 − ϱ n)
          using ‹∀ n m . n ≤ m ⟶ ϱ n ≤ ϱ m›
          by simp
        ultimately have
          (1 − ϱ (Suc n)) ∗ (∑ i≤n . ?α i) ≤ (1 − ϱ n) ∗ (∑ i≤n . ?α i)
          using ‹∀ n . 0 ≤ (∑ i≤n . ?α i)›
          by (meson le-less mult-mono′)
        hence
          (1 − ϱ (Suc n)) ∗ (∑ i≤ Suc n . ?α i) ≤
           (1 − ϱ n) ∗ (∑ i≤n . ?α i) + (1 − ϱ (Suc n)) ∗ (?α (Suc n))
          (is - ≤ ?X)
          by (simp add: distrib-left)
        moreover have
          ?X ≤ (∑ i≤ Suc n . ?ϱα i)
          using Suc.hyps by fastforce
        ultimately show ?case by auto
      qed
      moreover have 0 < 1 − ϱ n
        by (simp add: ‹∀ n . ϱ n < 1›)
      ultimately have 0 ≤ (∑ i≤n . ?ϱα i)
        using dual-order.trans by fastforce
    }
    hence strictly-solvent (return-loans ϱ α)
      unfolding strictly-solvent-def Rep-account-return-loans
      by auto
  }
  hence 0 ≤ return-loans ϱ (β − α)
    using ‹α ≤ β›
    by (simp add: strictly-solvent-alt-def)
  thus ?thesis
    by (metis
          add-diff-cancel-left′
          diff-ge-0-iff-ge
          minus-account-def
          return-loans-plus)
qed

lemma return-loans-just-cash:
  assumes ϱ 0 = 0
  shows return-loans ϱ (just-cash c) = just-cash c
proof −
  have (λn. (1 − ϱ n) ∗ π (ι (λn. if n = 0 then c else 0)) n)
       = (λn. if n = 0 then (1 − ϱ n) ∗ c else 0)
    using Rep-account-just-cash just-cash-def by force
  also have . . . = (λn. if n = 0 then c else 0)
    using ‹ϱ 0 = 0›
    by force
  finally show ?thesis
  unfolding return-loans-def just-cash-def
```

by *presburger*
qed

lemma *distribute-interest-plus*:
 just-cash $(i * \textit{net-asset-value}\ (\alpha + \beta)) =$
  just-cash $(i * \textit{net-asset-value}\ \alpha) +$
  just-cash $(i * \textit{net-asset-value}\ \beta)$
 unfolding *just-cash-def net-asset-value-plus*
 by (*metis*
   distrib-left
   just-cash-plus
   just-cash-def)

We now prove that *update-account* is an order-preserving group homomorphism just as *just-cash*, *net-asset-value*, and *return-loans* are.

lemma *update-account-plus*:
 $\textit{update-account}\ \varrho\ i\ (\alpha + \beta) =$
  $\textit{update-account}\ \varrho\ i\ \alpha + \textit{update-account}\ \varrho\ i\ \beta$
 unfolding
  update-account-def
  return-loans-plus
  distribute-interest-plus
 by *simp*

lemma *update-account-zero* [*simp*]: $\textit{update-account}\ \varrho\ i\ 0 = 0$
 by (*metis add-cancel-right-left update-account-plus*)

lemma *update-account-uminus*:
 $\textit{update-account}\ \varrho\ i\ (-\alpha) = -\ \textit{update-account}\ \varrho\ i\ \alpha$
 unfolding *update-account-def*
 by (*simp add*: *net-asset-value-uminus return-loans-uminus*)

lemma *update-account-subtract*:
 $\textit{update-account}\ \varrho\ i\ (\alpha - \beta) =$
  $\textit{update-account}\ \varrho\ i\ \alpha - \textit{update-account}\ \varrho\ i\ \beta$
 by (*simp add*: *additive.diff additive.intro update-account-plus*)

lemma *update-account-mono*:
 assumes $0 \leq i$
 and $\forall\ n\ .\ \varrho\ n < 1$
 and $\forall\ n\ m\ .\ n \leq m \longrightarrow \varrho\ n \leq \varrho\ m$
 and $\alpha \leq \beta$
 shows $\textit{update-account}\ \varrho\ i\ \alpha \leq \textit{update-account}\ \varrho\ i\ \beta$
proof −
 have $\textit{net-asset-value}\ \alpha \leq \textit{net-asset-value}\ \beta$
  using ‹$\alpha \leq \beta$› *net-asset-value-mono* **by** *presburger*
 hence $i * \textit{net-asset-value}\ \alpha \leq i * \textit{net-asset-value}\ \beta$
  by (*simp add*: ‹$0 \leq i$› *mult-left-mono*)
 hence *just-cash* $(i * \textit{net-asset-value}\ \alpha) \leq$

```
      just-cash (i * net-asset-value β)
    by (simp add: just-cash-order-embed)
  moreover
  have return-loans ϱ α ≤ return-loans ϱ β
    using assms return-loans-mono by presburger
  ultimately show ?thesis unfolding update-account-def
    by (simp add: add-mono)
qed
```

It follows from monotonicity and *update-account* ϱ *i* *0* = *0* that strictly solvent accounts remain strictly solvent after update.

```
lemma update-preserves-strictly-solvent:
  assumes 0 ≤ i
  and ∀ n . ϱ n < 1
  and ∀ n m . n ≤ m ⟶ ϱ n ≤ ϱ m
  and strictly-solvent α
  shows strictly-solvent (update-account ϱ i α)
  using assms
  unfolding strictly-solvent-alt-def
  by (metis update-account-mono update-account-zero)
```

7 Bulk Update

In this section we demonstrate there exists a closed form for bulk-updating an account.

```
primrec bulk-update-account ::
   nat ⇒ (nat ⇒ real) ⇒ real ⇒ account ⇒ account
   where
   bulk-update-account 0 - - α = α
 | bulk-update-account (Suc n) ϱ i α =
     update-account ϱ i (bulk-update-account n ϱ i α)
```

As with *update-account*, *bulk-update-account* is an order-preserving group homomorphism.

We now prove that *update-account* is an order-preserving group homomorphism just as *just-cash*, *net-asset-value*, and *return-loans* are.

```
lemma bulk-update-account-plus:
  bulk-update-account n ϱ i (α + β) =
    bulk-update-account n ϱ i α + bulk-update-account n ϱ i β
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case
    using bulk-update-account.simps(2) update-account-plus by presburger
```

qed

lemma *bulk-update-account-zero* [*simp*]: *bulk-update-account* $n\ \varrho\ i\ 0 = 0$
by (*metis add-cancel-right-left bulk-update-account-plus*)

lemma *bulk-update-account-uminus*:
bulk-update-account $n\ \varrho\ i\ (-\alpha) = -$ *bulk-update-account* $n\ \varrho\ i\ \alpha$
by (*metis add-eq-0-iff bulk-update-account-plus bulk-update-account-zero*)

lemma *bulk-update-account-subtract*:
bulk-update-account $n\ \varrho\ i\ (\alpha - \beta) =$
bulk-update-account $n\ \varrho\ i\ \alpha -$ *bulk-update-account* $n\ \varrho\ i\ \beta$
by (*simp add*: *additive.diff additive-def bulk-update-account-plus*)

lemma *bulk-update-account-mono*:
assumes $0 \leq i$
and $\forall\ n .\ \varrho\ n < 1$
and $\forall\ n\ m .\ n \leq m \longrightarrow \varrho\ n \leq \varrho\ m$
and $\alpha \leq \beta$
shows *bulk-update-account* $n\ \varrho\ i\ \alpha \leq$ *bulk-update-account* $n\ \varrho\ i\ \beta$
using *assms*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case*
using *bulk-update-account.simps*(*2*) *update-account-mono* **by** *presburger*
qed

In follows from the fact that *bulk-update-account* is an order-preserving group homomorphism that the update protocol is *safe*. Informally this means that provided we enforce every account is strictly solvent then no account will ever have negative net asset value (ie, be in the red).

theorem *bulk-update-safety*:
assumes $0 \leq i$
and $\forall\ n .\ \varrho\ n < 1$
and $\forall\ n\ m .\ n \leq m \longrightarrow \varrho\ n \leq \varrho\ m$
and *strictly-solvent* α
shows $0 \leq$ *net-asset-value* (*bulk-update-account* $n\ \varrho\ i\ \alpha$)
using *assms*
by (*metis*
bulk-update-account-mono
bulk-update-account-zero
strictly-solvent-alt-def
strictly-solvent-net-asset-value)

7.1 Decomposition

In order to express *bulk-update-account* using a closed formulation, we first demonstrate how to *decompose* an account into a summation of credited and debited loans for different periods.

definition *loan* :: *nat* ⇒ *real* ⇒ *account* (‹δ›)
  where
    $\delta\ n\ x = \iota\ (\lambda\ m\ .\ \mathit{if}\ n = m\ \mathit{then}\ x\ \mathit{else}\ 0)$

lemma *loan-just-cash*: $\delta\ 0\ c = \mathit{just\text{-}cash}\ c$
  unfolding *just-cash-def loan-def*
  by *force*

lemma *Rep-account-loan* [*simp*]:
  $\pi\ (\delta\ n\ x) = (\lambda\ m\ .\ \mathit{if}\ n = m\ \mathit{then}\ x\ \mathit{else}\ 0)$
proof −
  have $(\lambda\ m\ .\ \mathit{if}\ n = m\ \mathit{then}\ x\ \mathit{else}\ 0) \in \mathit{fin\text{-}support}\ 0\ \mathit{UNIV}$
    unfolding *fin-support-def support-def*
    by *force*
  thus *?thesis*
    unfolding *loan-def*
    using *Abs-account-inverse* **by** *blast*
qed

lemma *loan-zero* [*simp*]: $\delta\ n\ 0 = 0$
  unfolding *loan-def*
  using *zero-account-def* **by** *fastforce*

lemma *shortest-period-loan*:
  assumes $c \neq 0$
  shows $\mathit{shortest\text{-}period}\ (\delta\ n\ c) = n$
  using *assms*
  unfolding *shortest-period-def Rep-account-loan*
  by *simp*

lemma *net-asset-value-loan* [*simp*]: $\mathit{net\text{-}asset\text{-}value}\ (\delta\ n\ c) = c$
proof (*cases* $c = 0$)
  case *True*
  then show *?thesis* **by** *simp*
next
  case *False*
  hence $\mathit{shortest\text{-}period}\ (\delta\ n\ c) = n$ **using** *shortest-period-loan* **by** *blast*
  then show *?thesis* **unfolding** *net-asset-value-alt-def* **by** *simp*
qed

lemma *return-loans-loan* [*simp*]: $\mathit{return\text{-}loans}\ \varrho\ (\delta\ n\ c) = \delta\ n\ ((1 - \varrho\ n) * c)$
proof −
  have $\mathit{return\text{-}loans}\ \varrho\ (\delta\ n\ c) =$
      $\iota\ (\lambda na.\ (\mathit{if}\ n = na\ \mathit{then}\ (1 - \varrho\ n) * c\ \mathit{else}\ 0))$

```
    unfolding return-loans-def
    by (metis Rep-account-loan mult.commute mult-zero-left)
  thus ?thesis
    by (simp add: loan-def)
qed

lemma account-decomposition:
  α = (∑ i ≤ shortest-period α. δ i (π α i))
proof −
  let ?p = shortest-period α
  let ?πα = π α
  let ?Σδ = ∑ i ≤ ?p. δ i (?πα i)
  {
    fix n m :: nat
    fix f :: nat ⇒ real
    assume n > m
    hence π (∑ i ≤ m. δ i (f i)) n = 0
      by (induct m, simp+)
  }
  note · = this
  {
    fix n :: nat
    have π ?Σδ n = ?πα n
    proof (cases n ≤ ?p)
      case True
      {
        fix n m :: nat
        fix f :: nat ⇒ real
        assume n ≤ m
        hence π (∑ i ≤ m. δ i (f i)) n = f n
        proof (induct m)
          case 0
          then show ?case by simp
        next
          case (Suc m)
          then show ?case
          proof (cases n = Suc m)
            case True
            then show ?thesis using · by auto
          next
            case False
            hence n ≤ m
              using Suc.prems le-Suc-eq by blast
            then show ?thesis
              by (simp add: Suc.hyps)
          qed
        qed
      }
      then show ?thesis using True by auto
```

```
    next
      case False
      have ?πα n = 0
        unfolding shortest-period-def
        using False shortest-period-bound by blast
      thus ?thesis using False · by auto
    qed
  }
  thus ?thesis
    by (metis Rep-account-inject ext)
qed
```

7.2 Simple Transfers

Building on our decomposition, we can understand the necessary and sufficient conditions to transfer a loan of $\delta\ n\ c$.

We first give a notion of the *reserves for a period n*. This characterizes the available funds for a loan of period n that an account α possesses.

```
definition reserves-for-period :: account ⇒ nat ⇒ real where
  reserves-for-period α n =
      fold
        min
        [(∑ i≤k . π α i) . k ← [n..<shortest-period α+1]]
        (∑ i≤n . π α i)

lemma nav-reserves-for-period:
  assumes shortest-period α ≤ n
  shows reserves-for-period α n = net-asset-value α
proof cases
  assume shortest-period α = n
  hence [n..<shortest-period α+1] = [n]
    by simp
  hence [(∑ i≤k . π α i) . k ← [n..<shortest-period α+1]] =
          [(∑ i≤n . π α i)]
    by simp
  then show ?thesis
    unfolding reserves-for-period-def
    by (simp add: ‹shortest-period α = n› net-asset-value-alt-def)
next
  assume shortest-period α ≠ n
  hence shortest-period α < n
    using assms order-le-imp-less-or-eq by blast
  hence [(∑ i≤k . π α k) . k ← [n..<shortest-period α+1]] = []
    by force
  hence reserves-for-period α n = (∑ i≤n . π α i)
    unfolding reserves-for-period-def by auto
  then show ?thesis
    using assms net-asset-value-shortest-period-ge by presburger
```

qed

lemma *reserves-for-period-exists*:
$\exists\, m \geq n.\ \textit{reserves-for-period}\ \alpha\ n = (\sum\ i \leq m\ .\ \pi\ \alpha\ i)$
$\quad \wedge (\forall\, u \geq n.\ (\sum\ i \leq m\ .\ \pi\ \alpha\ i) \leq (\sum\ i \leq u\ .\ \pi\ \alpha\ i))$
proof $-$
{
fix j
have $\exists\, m \geq n.\ (\sum\ i \leq m\ .\ \pi\ \alpha\ i) =$
fold
min
$[(\sum\ i \leq k\ .\ \pi\ \alpha\ i)\ .\ k \leftarrow [n..<j]]$
$(\sum\ i \leq n\ .\ \pi\ \alpha\ i)$
$\wedge (\forall\, u \geq n.\ u < j \longrightarrow (\sum\ i \leq m\ .\ \pi\ \alpha\ i) \leq (\sum\ i \leq u\ .\ \pi\ \alpha\ i))$
proof (*induct j*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc j*)
then show *?case*
proof *cases*
assume $j \leq n$
thus *?thesis*
by (*simp*, *metis dual-order.refl le-less-Suc-eq*)
next
assume $\neg(j \leq n)$
hence $n < j$ **by** *auto*
obtain m **where**
$m \geq n$
$\forall\, u \geq n.\ u < j \longrightarrow (\sum\ i \leq m\ .\ \pi\ \alpha\ i) \leq (\sum\ i \leq u\ .\ \pi\ \alpha\ i)$
$(\sum\ i \leq m\ .\ \pi\ \alpha\ i) =$
fold
min
$[(\sum\ i \leq k\ .\ \pi\ \alpha\ i)\ .\ k \leftarrow [n..<j]]$
$(\sum\ i \leq n\ .\ \pi\ \alpha\ i)$
using *Suc* **by** *blast*
note $\heartsuit$ = *this*
hence †: $\textit{min}\ (\sum\ i \leq m\ .\ \pi\ \alpha\ i)\ (\sum\ i \leq j\ .\ \pi\ \alpha\ i) =$
fold
min
$[(\sum\ i \leq k\ .\ \pi\ \alpha\ i)\ .\ k \leftarrow [n..<\textit{Suc}\ j]]$
$(\sum\ i \leq n\ .\ \pi\ \alpha\ i)$
(**is** - = *?fold*)
using ‹$n < j$› **by** *simp*
show *?thesis*
proof *cases*
assume $(\sum\ i \leq m\ .\ \pi\ \alpha\ i) < (\sum\ i \leq j\ .\ \pi\ \alpha\ i)$
hence
$\forall\, u \geq n.\ u < \textit{Suc}\ j \longrightarrow (\sum\ i \leq m\ .\ \pi\ \alpha\ i) \leq (\sum\ i \leq u\ .\ \pi\ \alpha\ i)$

```
        by (metis
              ♡(2)
              dual-order.order-iff-strict
              less-Suc-eq)
      thus ?thesis
        using † ‹m ≥ n› by auto
    next
      assume ⋆: ¬ ((∑ i≤m . π α i) < (∑ i≤j . π α i))
      hence
        ∀ u≥n. u < j ⟶ (∑ i≤j . π α i) ≤ (∑ i≤u . π α i)
        using ♡(2)
        by auto
      hence
        ∀ u≥n. u < Suc j ⟶ (∑ i≤j . π α i) ≤ (∑ i≤u . π α i)
        by (simp add: less-Suc-eq)
      also have ?fold = (∑ i≤j . π α i)
        using † ⋆ by linarith
      ultimately show ?thesis
        by (metis ‹n < j› less-or-eq-imp-le)
    qed
  qed
 qed
}
from this obtain m where
    m ≥ n
    (∑ i≤m . π α i) = reserves-for-period α n
    ∀ u≥n. u < shortest-period α + 1
          ⟶ (∑ i≤m . π α i) ≤ (∑ i≤u . π α i)
  unfolding reserves-for-period-def
  by blast
note ◇ = this
hence (∑ i≤m . π α i) ≤ (∑ i≤shortest-period α . π α i)
  by (metis
        less-add-one
        nav-reserves-for-period
        net-asset-value-alt-def
        nle-le)
hence ∀ u≥shortest-period α. (∑ i≤m . π α i) ≤ (∑ i≤u . π α i)
  by (metis
        net-asset-value-alt-def
        net-asset-value-shortest-period-ge)
hence ∀ u≥n. (∑ i≤m . π α i) ≤ (∑ i≤u . π α i)
  by (metis ◇(3) Suc-eq-plus1 less-Suc-eq linorder-not-le)
thus ?thesis
  using ◇(1) ◇(2)
  by metis
qed
```

lemma *permissible-loan-converse*:

assumes *strictly-solvent* $(\alpha - \delta\ n\ c)$
shows $c \leq$ *reserves-for-period* $\alpha\ n$
proof $-$
obtain m **where**
$n \leq m$
reserves-for-period $\alpha\ n = (\sum\ i{\leq}m\ .\ \pi\ \alpha\ i)$
using *reserves-for-period-exists* **by** *blast*
have $(\sum\ i{\leq}m\ .\ \pi\ (\alpha - \delta\ n\ c)\ i) = (\sum\ i{\leq}m\ .\ \pi\ \alpha\ i) - c$
using $\langle n \leq m\rangle$
proof (*induct m*)
case *0*
hence $n = 0$ **by** *auto*
have $\pi\ (\alpha - \delta\ n\ c + \delta\ n\ c)\ 0 = \pi\ (\alpha - \delta\ n\ c)\ 0 + \pi\ (\delta\ n\ c)\ 0$
using *Rep-account-plus* **by** *presburger*
thus *?case*
unfolding $\langle n = 0\rangle$
by *simp*
next
case (*Suc m*)
then show *?case*
proof *cases*
assume $n = Suc\ m$
hence $m < n$ **by** *auto*
hence $(\sum\ i{\leq}m\ .\ \pi\ (\alpha - \delta\ n\ c)\ i) = (\sum\ i{\leq}m\ .\ \pi\ \alpha\ i)$
proof(*induct m*)
case *0*
then show *?case*
by (*metis*
(*no-types*, *opaque-lifting*)
Rep-account-loan
Rep-account-plus
atMost-0 bot-nat-0.not-eq-extremum
diff-0-right
diff-add-cancel
empty-iff
finite.intros(*1*)
sum.empty
sum.insert)
next
case (*Suc m*)
hence $m < n$ **and** $n \neq Suc\ m$
using *Suc-lessD* **by** *blast+*
moreover have
$\pi\ (\alpha - \delta\ n\ c + \delta\ n\ c)\ (Suc\ m) =$
$\pi\ (\alpha - \delta\ n\ c)\ (Suc\ m) + \pi\ (\delta\ n\ c)\ (Suc\ m)$
using *Rep-account-plus* **by** *presburger*
ultimately show *?case* **by** (*simp add*: *Suc.hyps*)
qed
moreover

```
      have π (α − δ (Suc m) c + δ (Suc m) c) (Suc m) =
              π (α − δ (Suc m) c) (Suc m) + π (δ (Suc m) c) (Suc m)
        by (meson Rep-account-plus)
      ultimately show ?thesis
        unfolding ‹n = Suc m›
        by simp
    next
      assume n ≠ Suc m
      hence n ≤ m
        using Suc.prems le-SucE by blast
      have π (α − δ n c + δ n c) (Suc m) =
              π (α − δ n c) (Suc m) + π (δ n c) (Suc m)
        by (meson Rep-account-plus)
      moreover have 0 = (if n = Suc m then c else 0)
        using ‹n ≠ Suc m› by presburger
      ultimately show ?thesis
        by (simp add: Suc.hyps ‹n ≤ m›)
    qed
  qed
  hence 0 ≤ (∑ i≤m . π α i) − c
    by (metis assms strictly-solvent-def)
  thus ?thesis
    by (simp add: ‹reserves-for-period α n = sum (π α) {..m}›)
qed

lemma permissible-loan:
  assumes strictly-solvent α
  shows strictly-solvent (α − δ n c) = (c ≤ reserves-for-period α n)
proof
  assume strictly-solvent (α − δ n c)
  thus c ≤ reserves-for-period α n
    using permissible-loan-converse by blast
next
  assume c ≤ reserves-for-period α n
  {
    fix j
    have 0 ≤ (∑ i≤j . π (α − δ n c) i)
    proof cases
      assume j < n
      hence (∑ i≤j . π (α − δ n c) i) = (∑ i≤j . π α i)
      proof (induct j)
        case 0
        then show ?case
          by (simp,
              metis
                Rep-account-loan
                Rep-account-plus
                ‹j < n›
                add.commute
```

add-0
diff-add-cancel
gr-implies-not-zero)
next
case (*Suc j*)
moreover have π (α − δ *n c* + δ *n c*) (*Suc j*) =
π (α − δ *n c*) (*Suc j*) + π (δ *n c*) (*Suc j*)
using *Rep-account-plus* **by** *presburger*
ultimately show *?case* **by** *simp*
qed
thus *?thesis*
by (*metis assms strictly-solvent-def*)
next
assume ¬ (*j* < *n*)
hence *n* ≤ *j* **by** *auto*
obtain *m* **where**
reserves-for-period α *n* = ($\sum$ *i*≤*m* . π α *i*)
∀ *u*≥*n*. ($\sum$ *i*≤*m* . π α *i*) ≤ ($\sum$ *i*≤*u* . π α *i*)
using *reserves-for-period-exists* **by** *blast*
hence ∀ *u*≥*n*. *c* ≤ ($\sum$ *i*≤*u* . π α *i*)
using ‹*c* ≤ *reserves-for-period* α *n*›
by *auto*
hence *c* ≤ ($\sum$ *i*≤*j* . π α *i*)
using ‹*n* ≤ *j*› **by** *presburger*
hence *0* ≤ ($\sum$ *i*≤*j* . π α *i*) − *c*
by *force*
moreover have ($\sum$ *i*≤*j* . π α *i*) − *c* = ($\sum$ *i*≤*j* . π (α − δ *n c*) *i*)
using ‹*n* ≤ *j*›
proof (*induct j*)
case *0*
hence *n* = *0* **by** *auto*
have π (α − δ *0 c* + δ *0 c*) *0* = π (α − δ *0 c*) *0* + π (δ *0 c*) *0*
using *Rep-account-plus* **by** *presburger*
thus *?case* **unfolding** ‹*n* = *0*› **by** *simp*
next
case (*Suc j*)
then show *?case*
proof *cases*
assume *n* = *Suc j*
hence *j* < *n*
by *blast*
hence ($\sum$ *i*≤*j* . π (α − δ *n c*) *i*) = ($\sum$ *i*≤*j* . π α *i*)
proof (*induct j*)
case *0*
then show *?case*
by (*simp*,
metis
Rep-account-loan
Rep-account-plus

```
              ‹j < n›
              add.commute
              add-0
              diff-add-cancel
              gr-implies-not-zero)
        next
          case (Suc j)
          moreover have π (α − δ n c + δ n c) (Suc j) =
                        π (α − δ n c) (Suc j) + π (δ n c) (Suc j)
            using Rep-account-plus by presburger
          ultimately show ?case by simp
        qed
        moreover have
          π (α − δ (Suc j) c + δ (Suc j) c) (Suc j) =
            π (α − δ (Suc j) c) (Suc j) + π (δ (Suc j) c) (Suc j)
          using Rep-account-plus by presburger
        ultimately show ?thesis
          unfolding ‹n = Suc j›
          by simp
      next
        assume n ≠ Suc j
        hence n ≤ j
          using Suc.prems le-SucE by blast
        hence (∑ i≤j . π α i) − c = (∑ i≤j . π (α − δ n c) i)
          using Suc.hyps by blast
        moreover have π (α − δ n c + δ n c) (Suc j) =
                      π (α − δ n c) (Suc j) + π (δ n c) (Suc j)
          using Rep-account-plus by presburger
        ultimately show ?thesis
          by (simp add: ‹n ≠ Suc j›)
      qed
    qed
    ultimately show ?thesis by auto
  qed
}
thus strictly-solvent (α − δ n c)
  unfolding strictly-solvent-def
  by auto
qed
```

7.3 Closed Forms

We first give closed forms for loans *δ n c*. The simplest closed form is for *just-cash*. Here the closed form is just the compound interest accrued from each update.

```
lemma bulk-update-just-cash-closed-form:
  assumes ϱ 0 = 0
  shows bulk-update-account n ϱ i (just-cash c) =
          just-cash ((1 + i) ^ n ∗ c)
```

```
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have return-loans ϱ (just-cash ((1 + i) ^ n * c)) =
          just-cash ((1 + i) ^ n * c)
    using assms return-loans-just-cash by blast
  thus ?case
    using Suc net-asset-value-just-cash-left-inverse
    by (simp add: update-account-def,
        metis
          add.commute
          mult.commute
          mult.left-commute
          mult-1
          ring-class.ring-distribs(2))
qed

lemma bulk-update-loan-closed-form:
  assumes ϱ k ≠ 1
  and ϱ k > 0
  and ϱ 0 = 0
  and i ≥ 0
  shows bulk-update-account n ϱ i (δ k c) =
          just-cash (c * i * ((1 + i) ^ n − (1 − ϱ k) ^ n) / (i + ϱ k))
          + δ k ((1 − ϱ k) ^ n * c)
proof (induct n)
  case 0
  then show ?case
    by (simp add: zero-account-alt-def)
next
  case (Suc n)
  have i + ϱ k > 0
    using assms(2) assms(4) by force
  hence (i + ϱ k) / (i + ϱ k) = 1
    by force
  hence bulk-update-account (Suc n) ϱ i (δ k c) =
           just-cash
             ((c * i) / (i + ϱ k) * (1 + i) * ((1 + i) ^ n − (1 − ϱ k) ^ n) +
              c * i * (1 − ϱ k) ^ n * ((i + ϱ k) / (i + ϱ k)))
           + δ k ((1 − ϱ k) ^ (n + 1) * c)
    using Suc
    by (simp add:
           return-loans-plus
           ‹ϱ 0 = 0›
           return-loans-just-cash
           update-account-def
           net-asset-value-plus
```

```
          net-asset-value-just-cash-left-inverse
          add.commute
          add.left-commute
          distrib-left
          mult.assoc
          add-divide-distrib
          distrib-right
          mult.commute
          mult.left-commute)
  also have
    ... =
      just-cash
        ((c ∗ i) / (i + ϱ k) ∗ (1 + i) ∗ ((1 + i) ^ n − (1 − ϱ k) ^ n) +
         (c ∗ i) / (i + ϱ k) ∗ (1 − ϱ k) ^ n ∗ (i + ϱ k))
      + δ k ((1 − ϱ k) ^(n + 1) ∗ c)
    by (metis (no-types, lifting) times-divide-eq-left times-divide-eq-right)
  also have
    ... =
      just-cash
        ((c ∗ i) / (i + ϱ k) ∗ (
              (1 + i) ∗ ((1 + i) ^ n − (1 − ϱ k) ^ n)
            + (1 − ϱ k) ^ n ∗ (i + ϱ k)))
      + δ k ((1 − ϱ k) ^(n + 1) ∗ c)
    by (metis (no-types, lifting) mult.assoc ring-class.ring-distribs(1))
  also have
    ... =
      just-cash
        ((c ∗ i) / (i + ϱ k) ∗ ((1 + i) ^(n + 1) − (1 − ϱ k) ^(n + 1)))
      + δ k ((1 − ϱ k) ^(n + 1) ∗ c)
    by (simp add: mult.commute mult-diff-mult)
  ultimately show ?case by simp
qed
```

We next give an *algebraic* closed form. This uses the ordered abelian group that *account*s form.

```
lemma bulk-update-algebraic-closed-form:
  assumes 0 ≤ i
  and ∀ n . ϱ n < 1
  and ∀ n m . n < m ⟶ ϱ n < ϱ m
  and ϱ 0 = 0
  shows bulk-update-account n ϱ i α
          = just-cash (
              (1 + i) ^ n ∗ (cash-reserve α)
              + (∑ k = 1..shortest-period α.
                    (π α k) ∗ i ∗ ((1 + i) ^ n − (1 − ϱ k) ^ n)
                        / (i + ϱ k))
            )
            + (∑ k = 1..shortest-period α. δ k ((1 − ϱ k) ^ n ∗ π α k))
proof −
```

{
  fix *m*
  have $\forall\ k \in \{1..m\}.\ \varrho\ k \neq 1 \wedge \varrho\ k > 0$
    by (*metis*
          assms(2)
          assms(3)
          assms(4)
          atLeastAtMost-iff
          dual-order.refl
          less-numeral-extra(1)
          linorder-not-less
          not-gr-zero)
  hence $\star$: $\forall\ k \in \{1..m\}.$
        bulk-update-account $n\ \varrho\ i\ (\delta\ k\ (\pi\ \alpha\ k))$
      $=$ *just-cash* $((\pi\ \alpha\ k) * i * ((1 + i)\ \widehat{}\ n - (1 - \varrho\ k)\ \widehat{}\ n)$
            $/\ (i + \varrho\ k))$
        $+\ \delta\ k\ ((1 - \varrho\ k)\ \widehat{}\ n * (\pi\ \alpha\ k))$
    using *assms(1) assms(4) bulk-update-loan-closed-form* **by** *blast*
  have *bulk-update-account* $n\ \varrho\ i\ (\sum\ k \leq m.\ \delta\ k\ (\pi\ \alpha\ k))$
      $= (\sum\ k \leq m.$ *bulk-update-account* $n\ \varrho\ i\ (\delta\ k\ (\pi\ \alpha\ k)))$
    by (*induct m*, *simp*, *simp add*: *bulk-update-account-plus*)
  also have
    $\ldots =$ *bulk-update-account* $n\ \varrho\ i\ (\delta\ 0\ (\pi\ \alpha\ 0))$
      $+ (\sum\ k = 1..m.$ *bulk-update-account* $n\ \varrho\ i\ (\delta\ k\ (\pi\ \alpha\ k)))$
    by (*simp add*: *atMost-atLeast0 sum.atLeast-Suc-atMost*)
  also have
    $\ldots =$ *just-cash* $((1 + i)\ \widehat{}\ n *$ *cash-reserve* $\alpha)$
      $+ (\sum\ k = 1..m.$ *bulk-update-account* $n\ \varrho\ i\ (\delta\ k\ (\pi\ \alpha\ k)))$
    using
    $\langle \varrho\ 0 = 0 \rangle$
    bulk-update-just-cash-closed-form
    loan-just-cash
    cash-reserve-def
    by *presburger*
  also have
    $\ldots =$ *just-cash* $((1 + i)\ \widehat{}\ n *$ *cash-reserve* $\alpha)$
      $+ (\sum\ k = 1..m.$
          just-cash $((\pi\ \alpha\ k) * i * ((1 + i)\ \widehat{}\ n - (1 - \varrho\ k)\ \widehat{}\ n)$
            $/\ (i + \varrho\ k))$
        $+\ \delta\ k\ ((1 - \varrho\ k)\ \widehat{}\ n * (\pi\ \alpha\ k)))$
    using $\star$ **by** *auto*
  also have
    $\ldots =$ *just-cash* $((1 + i)\ \widehat{}\ n *$ *cash-reserve* $\alpha)$
      $+ (\sum\ k = 1..m.$
          just-cash $((\pi\ \alpha\ k) * i * ((1 + i)\ \widehat{}\ n - (1 - \varrho\ k)\ \widehat{}\ n)$
            $/\ (i + \varrho\ k)))$
      $+ (\sum\ k = 1..m.\ \delta\ k\ ((1 - \varrho\ k)\ \widehat{}\ n * (\pi\ \alpha\ k)))$
    by (*induct m*, *auto*)
  also have

$\ldots =$ *just-cash* $((1 + i)$ ^ $n * $ *cash-reserve* $\alpha)$
$+$ *just-cash*
$(\sum\ k = 1..m.$
$(\pi\ \alpha\ k) * i * ((1 + i)$ ^ $n - (1 - \varrho\ k)$ ^ $n) / (i + \varrho\ k))$
$+ (\sum\ k = 1..m.\ \delta\ k\ ((1 - \varrho\ k)$ ^ $n * (\pi\ \alpha\ k)))$
by (*induct m*, *auto*, *metis* (*no-types*, *lifting*) *add.assoc just-cash-plus*)
ultimately have
bulk-update-account $n\ \varrho\ i\ (\sum\ k \leq m.\ \delta\ k\ (\pi\ \alpha\ k)) =$
just-cash (
$(1 + i)$ ^ $n *$ *cash-reserve* α
$+ (\sum\ k = 1..m.$
$(\pi\ \alpha\ k) * i * ((1 + i)$ ^ $n - (1 - \varrho\ k)$ ^ $n) / (i + \varrho\ k)))$
$+ (\sum\ k = 1..m.\ \delta\ k\ ((1 - \varrho\ k)$ ^ $n * (\pi\ \alpha\ k)))$
by *simp*
}
note $\cdot =$ *this*
have
bulk-update-account $n\ \varrho\ i\ \alpha$
$=$ *bulk-update-account* $n\ \varrho\ i\ (\sum\ k \leq$ *shortest-period* $\alpha.\ \delta\ k\ (\pi\ \alpha\ k))$
using *account-decomposition* **by** *presburger*
thus *?thesis* **unfolding** $\cdot$ **.**
qed

We finally give a *functional* closed form for bulk updating an account. Since the form is in terms of exponentiation, we may efficiently compute the bulk update output using *exponentiation-by-squaring*.

theorem *bulk-update-closed-form*:
assumes $0 \leq i$
and $\forall\ n\ .\ \varrho\ n < 1$
and $\forall\ n\ m\ .\ n < m \longrightarrow \varrho\ n < \varrho\ m$
and $\varrho\ 0 = 0$
shows *bulk-update-account* $n\ \varrho\ i\ \alpha$
$= \iota\ (\ \lambda\ k\ .$
if $k = 0$ *then*
$(1 + i)$ ^ $n * ($*cash-reserve* $\alpha)$
$+ (\sum\ j = 1..$*shortest-period* $\alpha.$
$(\pi\ \alpha\ j) * i * ((1 + i)$ ^ $n - (1 - \varrho\ j)$ ^ $n)$
$/ (i + \varrho\ j))$
else
$(1 - \varrho\ k)$ ^ $n * \pi\ \alpha\ k$
)
(**is** $- = \iota\ ?\nu$)
proof $-$
obtain ν **where** X: $\nu = ?\nu$ **by** *blast*
moreover obtain ν' **where** Y:
$\nu' = \pi$ (*just-cash* (
$(1 + i)$ ^ $n * ($*cash-reserve* $\alpha)$
$+ (\sum\ j = 1..$*shortest-period* $\alpha.$
$(\pi\ \alpha\ j) * i * ((1 + i)$ ^ $n - (1 - \varrho\ j)$ ^ $n)$

$/ \ (i + \varrho\ j))$
$)$
$+ \ (\sum j = 1..shortest\text{-}period\ \alpha.\ \delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j)))$
by *blast*
moreover
{
fix k
have $\forall\ k > shortest\text{-}period\ \alpha\ .\ \nu\ k = \nu'\ k$
proof (*rule allI*, *rule impI*)
fix k
assume $shortest\text{-}period\ \alpha < k$
hence $\nu\ k = 0$
unfolding X
by (*simp add*: *greater-than-shortest-period-zero*)
moreover have $\nu'\ k = 0$
proof $-$
have $\forall\ c.\ \pi\ (just\text{-}cash\ c)\ k = 0$
using
Rep-account-just-cash
‹$shortest\text{-}period\ \alpha < k$›
just-cash-def
by *auto*
moreover
have $\forall\ m < k.\ \pi\ (\sum j = 1..m.\ \delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j))\ k = 0$
proof (*rule allI*, *rule impI*)
fix m
assume $m < k$
let $?\pi\Sigma\delta = \pi\ (\sum j = 1..m.\ \delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j))$
have $?\pi\Sigma\delta\ k = (\sum j = 1..m.\ \pi\ (\delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j))\ k)$
by (*induct m*, *auto*)
also have $\ldots = (\sum j = 1..m.\ 0)$
using ‹$m < k$›
by (*induct m*, *simp+*)
finally show $?\pi\Sigma\delta\ k = 0$
by *force*
qed
ultimately show *?thesis* **unfolding** Y
using ‹$shortest\text{-}period\ \alpha < k$› **by** *force*
qed
ultimately show $\nu\ k = \nu'\ k$ **by** *auto*
qed
moreover have $\forall\ k\ .\ 0 < k \longrightarrow k \leq shortest\text{-}period\ \alpha \longrightarrow \nu\ k = \nu'\ k$
proof (*rule allI*, (*rule impI*)+)
fix k
assume $0 < k$
and $k \leq shortest\text{-}period\ \alpha$
have $\nu\ k = (1 - \varrho\ k)\ \hat{}\ n * \pi\ \alpha\ k$
unfolding X
using ‹$0 < k$› **by** *fastforce*

moreover have $\nu'\ k = (1 - \varrho\ k)\ \hat{}\ n * \pi\ \alpha\ k$
proof $-$
  have $\forall\ c.\ \pi\ (\mathit{just\text{-}cash}\ c)\ k = 0$
    using $\langle 0 < k\rangle$ **by** *auto*
  moreover
  {
    fix m
    assume $k \leq m$
    have $\pi\ (\sum j = 1..m.\ \delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j))\ k$
      $= (\sum j = 1..m.\ \pi\ (\delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j))\ k)$
      by (*induct m*, *auto*)
    also
    have $\ldots = (1 - \varrho\ k)\ \hat{}\ n * \pi\ \alpha\ k$
      using $\langle 0 < k\rangle$ $\langle k \leq m\rangle$
    proof (*induct m*)
      case *0*
      then show *?case* **by** *simp*
    next
      case (*Suc m*)
      then show *?case*
      proof (*cases* $k = \mathit{Suc}\ m$)
        case *True*
        hence $k > m$ **by** *auto*
        hence $(\sum j = 1..m.\ \pi\ (\delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j))\ k) = 0$
          by (*induct m*, *auto*)
        then show *?thesis*
          using $\langle k > m\rangle$ $\langle k = \mathit{Suc}\ m\rangle$
          by *simp*
      next
        case *False*
        hence $(\sum j = 1..m.\ \pi\ (\delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j))\ k)$
          $= (1 - \varrho\ k)\ \hat{}\ n * \pi\ \alpha\ k$
          using *Suc.hyps Suc.prems*(*1*) *Suc.prems*(*2*) *le-Suc-eq* **by** *blast*
        moreover have $k \leq m$
          using *False Suc.prems*(*2*) *le-Suc-eq* **by** *blast*
        ultimately show *?thesis* **using** $\langle 0 < k\rangle$ **by** *simp*
      qed
    qed
    finally have
      $\pi\ (\sum j = 1..m.\ \delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j))\ k$
        $= (1 - \varrho\ k)\ \hat{}\ n * \pi\ \alpha\ k$ **.**
  }
  hence
    $\forall\ m \geq k.$
      $\pi\ (\sum j = 1..m.\ \delta\ j\ ((1 - \varrho\ j)\ \hat{}\ n * \pi\ \alpha\ j))\ k$
    $= (1 - \varrho\ k)\ \hat{}\ n * \pi\ \alpha\ k$ **by** *auto*
  ultimately show *?thesis*
    unfolding Y
    using $\langle k \leq \mathit{shortest\text{-}period}\ \alpha\rangle$

```
        by force
      qed
      ultimately show ν k = ν' k
        by fastforce
    qed
    moreover have ν 0 = ν' 0
    proof –
      have ν 0 = (1 + i) ^ n * (cash-reserve α)
                 + (∑ j = 1..shortest-period α.
                      (π α j) * i * ((1 + i) ^ n – (1 – ϱ j) ^ n)
                        / (i + ϱ j))
        using X by presburger
      moreover
      have ν' 0 = (1 + i) ^ n * (cash-reserve α)
                  + (∑ j = 1..shortest-period α.
                       (π α j) * i * ((1 + i) ^ n – (1 – ϱ j) ^ n)
                         / (i + ϱ j))
      proof –
        {
          fix m
          have π (∑ j = 1..m. δ j ((1 – ϱ j) ^ n * π α j)) 0 = 0
            by (induct m, simp+)
        }
        thus ?thesis unfolding Y
          by simp
      qed
      ultimately show ?thesis by auto
    qed
    ultimately have ν k = ν' k
      by (metis linorder-not-less not-gr0)
  }
  hence ι ν = ι ν'
    by presburger
  ultimately show ?thesis
    using
      Rep-account-inverse
      assms
      bulk-update-algebraic-closed-form
    by presburger
qed

end
```