The Stern-Brocot Tree

Peter Gammie Andreas Lochbihler

October 27, 2022

Abstract

The Stern-Brocot tree contains all rational numbers exactly once and in their lowest terms. We formalise the Stern-Brocot tree as a coinductive tree using recursive and iterative specifications, which we have proven equivalent, and show that it indeed contains all the numbers as stated. Following Hinze, we prove that the Stern-Brocot tree can be linearised looplessly into Stern’s diatonic sequence (also known as Dijkstra’s fusc function) and that it is a permutation of the Bird tree.

The reasoning stays at an abstract level by appealing to the uniqueness of solutions of guarded recursive equations and lifting algebraic laws point-wise to trees and streams using applicative functors.

Contents

1 A codatatype of infinite binary trees 2
  1.1 Applicative functor for ‘a tree 3
  1.2 Standard tree combinators 6
    1.2.1 Recurse combinator 6
    1.2.2 Tree iteration 6
    1.2.3 Tree traversal 7
    1.2.4 Mirroring 8
  1.3 Pointwise arithmetic on infinite binary trees 8
    1.3.1 Constants and operators 9
    1.3.2 Algebraic instances 10

2 The Stern-Brocot Tree 13
  2.1 Specification via a recursion equation 14
  2.2 Basic properties 15
  2.3 All the rationals 15
  2.4 No repetitions 16
  2.5 Equivalence of recursive and iterative version 22
3 Linearising the Stern-Brocot Tree

3.1 Turning a tree into a stream ........................................... 23
3.2 Split the Stern-Brocot tree into numerators and denumerators 24
3.3 Loopless linearisation of the Stern-Brocot tree. .................. 27
3.4 Equivalence with Dijkstra’s fuse function ......................... 28

4 The Bird tree .................................................................... 28

Acknowledgements  Thanks to Dave Cock for a fruitful discussion about
unique fixed points.

1 A codatatype of infinite binary trees

theory Cotree imports
  Main
  Applicative-Lifting.Applicative
  HOL-Library.BNF-Corec
  HOL-Library.Adhoc-Overloading
begin

context notes [[bnf-internals]]
begin
  codatatype 'a tree = Node (root: 'a) (left: 'a tree) (right: 'a tree)
end

lemma rel-treeD:
  assumes rel-tree A x y
  shows rel-tree-rootD: A (root x) (root y)
and rel-tree-leftD: rel-tree A (left x) (left y)
and rel-tree-rightD: rel-tree A (right x) (right y)
using assms
by(cases x y rule: tree.exhaust[case-product tree.exhaust], simp-all)+

lemmas [simp] = tree.map sel tree.map comp

lemma set-tree-induct[consumes 1, case-names root left right]:
  assumes x: x ∈ set-tree t
and root: ∃t. P (root t) t
and left: ∀x t. [ x ∈ set-tree (left t); P x (left t) ] ⊨ P x t
and right: ∀x t. [ x ∈ set-tree (right t); P x (right t) ] ⊨ P x t
shows P x t
using x
proof(rule tree.set-induct)
  fix l r
  from root[of Node x l r] show P x (Node x l r) by simp
qed(auto intro: left right)
lemma corec-tree-cong:
assumes \( \forall x. \text{stopL} x \Rightarrow \text{STOPL} x = \text{STOPL}' x \)
and \( \forall x. \sim \text{stopL} x \Rightarrow \text{LEFT} x = \text{LEFT}' x \)
and \( \forall x. \text{stopR} x \Rightarrow \text{STOPR} x = \text{STOPR}' x \)
and \( \forall x. \sim \text{stopR} x \Rightarrow \text{RIGHT} x = \text{RIGHT}' x \)
shows corec-tree \( \text{ROOT} \text{STOPL} \text{STOPL} \text{LEFT} \text{STOPR} \text{STOPR} \text{RIGHT} = \\
\text{corec-tree} \text{ROOT} \text{STOPL} \text{STOPR} \text{LEFT} \text{STOPR} \text{STOPR} \text{RIGHT}' \)
(is \( ?\text{lhs} = ?\text{rhs} \))
proof
fix \( x \)
show \( ?\text{lhs} x = ?\text{rhs} x \)
by (coinduction arbitrary; \( x \) rule: tree.coinduct-strong)(auto simp add: assms)
qed

context fixes \( g1 :: 'a \Rightarrow 'b \)
and \( g22 :: 'a \Rightarrow 'a \)
and \( g32 :: 'a \Rightarrow 'a \)
begin

corec unfold-tree :: 'a \Rightarrow 'b tree
where unfold-tree \( a \) = Node \( g1 a \) (unfold-tree \( g22 a \)) (unfold-tree \( g32 a \))

lemma unfold-tree-simps [simp]:
root (unfold-tree \( a \)) = \( g1 a \)
left (unfold-tree \( a \)) = unfold-tree \( g22 a \)
right (unfold-tree \( a \)) = unfold-tree \( g32 a \)
by(subst unfold-tree.code; simp; fail)+
end

lemma unfold-tree-unique:
assumes \( \forall s. \text{root} (f s) = \text{ROOT} s \)
and \( \forall s. \text{left} (f s) = f (\text{LEFT} s) \)
and \( \forall s. \text{right} (f s) = f (\text{RIGHT} s) \)
shows \( f s = \text{unfold-tree} \text{ROOT} \text{LEFT} \text{RIGHT} s \)
by(rule unfold-tree.unique[THEN fun-cong])(auto simp add: fun-eq-iff assms intro: tree.expand)

1.1 Applicative functor for 'a tree

context fixes \( x :: 'a \)
begin

corec pure-tree :: 'a tree
where pure-tree = Node \( x \) pure-tree pure-tree
end

lemmas pure-tree-unfold = pure-tree.code

lemma pure-tree-simps [simp]:
root (pure-tree x) = x
left (pure-tree x) = pure-tree x
right (pure-tree x) = pure-tree x
by (subst pure-tree-unfold; simp; fail)+

adhoc-overloading pure pure-tree

lemma pure-tree-parametric [transfer-rule]: (rel-fun A (rel-tree A)) pure pure
by (rule rel-fun); (coinduction, auto)

lemma map-pure-tree [simp]: map-tree f (pure x) = pure (f x)
by (coinduction arbitrary; x) auto

lemmas pure-tree-unique = pure-tree.unique

primcorec (transfer) ap-tree :: ('a ⇒ 'b) tree ⇒ 'a tree ⇒ 'b tree
where
  root (ap-tree f x) = root f (root x)
| left (ap-tree f x) = ap-tree (left f) (left x)
| right (ap-tree f x) = ap-tree (right f) (right x)

adhoc-overloading Applicative.ap ap-tree

unbundle applicative-syntax

lemma ap-tree-pure-Node [simp]:
  pure f ⋄ Node x l r = Node (f x) (pure f ⋄ l) (pure f ⋄ r)
by (rule tree.expand) auto

lemma ap-tree-Node-Node [simp]:
  Node f fl fr ⋄ Node x l r = Node (f x) (fl ⋄ l) (fr ⋄ r)
by (rule tree.expand) auto

Applicative functor laws

lemma map-tree-ap-tree-pure-tree:
  pure f ⋄ u = map-tree f u
by (coinduction arbitrary: u) auto

lemma ap-tree-identity: pure id ⋄ t = t
by (simp add: map-tree-ap-tree-pure-tree.tree.map-id)

lemma ap-tree-composition:
  pure (○) ⋄ r1 ⋄ r2 ⋄ r3 = r1 ⋄ (r2 ⋄ r3)
by (coinduction arbitrary: r1 r2 r3) auto

lemma ap-tree-homomorphism:
  pure f ⋄ pure x = pure (f x)
by (simp add: map-tree-ap-tree-pure-tree)
lemma ap-tree-interchange:
\[ t \circ pure \, x = pure \, (\lambda f \, x \circ t) \]
by(coinduction arbitrary: \( t \)) auto

lemma ap-tree-K-tree: pure \((\lambda x \, y. \, f \, y \, x) \circ u \circ v = u \circ w \circ v \)
by(coinduction arbitrary: \( u \circ v \)) (auto)

lemma ap-tree-C-tree: pure \((\lambda f \, x \, y. \, f \, y \, x) \circ u \circ v \circ w = u \circ w \circ v \)
by(coinduction arbitrary: \( u \circ v \circ w \)) (auto)

lemma ap-tree-W-tree: pure \((\lambda f \, x. \, f \, x \, x) \circ f \circ x = f \circ x \circ x \)
by(coinduction arbitrary: \( f \circ x \)) (auto)

applicative tree \((K, W)\) for
pure: pure-tree
ap: ap-tree
rel: rel-tree
set: set-tree

proof
  fix \( R : `'c \Rightarrow 'c \Rightarrow bool \) and \( f : (`a \Rightarrow `b) \Rightarrow tree \) and \( g \, x \)
  assume [transfer-rule]: \( rel-tree \, (rel-fun \, (eq-on \, (set-tree \, x))) \, R \, f \, g \)
  have [transfer-rule]: \( rel-tree \, (eq-on \, (set-tree \, x)) \, x \, x \) by(rule tree.rel-refl-strong)
  simp
  show \( rel-tree \, R \, (f \circ x \, \circ (g \circ x)) \) by transfer-prover
qed(rule ap-tree-homomorphism ap-tree-composition[unfolded o-def[abs-def]] ap-tree-K-tree
ap-tree-W-tree ap-tree-interchange pure-tree-parametric)+

declare map-tree-ap-tree-pure-tree[symmetric, applicative-unfold]

lemma ap-tree-strong-extensional:
\[(\forall x. \, f \circ pure \, x = g \circ pure \, x) \Rightarrow f = g\]
proof(coinduction arbitrary: \( f \circ g \))
case [rule-format]: \( Eq-tree \, f \, g \)
  have root \( f = \) root \( g \)
proof
  fix \( x \)
  show root \( f \, x = root \, g \, x \)
  using \( Eq-tree[af \, x] \) by(subst (asm) (1 2) ap-tree.ctr) simp
qed
moreover {
  fix \( x \)
  have left \( f \circ pure \, x = left \, g \circ pure \, x \)
  using \( Eq-tree[af \, x] \) by(subst (asm) (1 2) ap-tree.ctr) simp
}
moreover {
  fix \( x \)
  have right \( f \circ pure \, x = right \, g \circ pure \, x \)
  using \( Eq-tree[af \, x] \) by(subst (asm) (1 2) ap-tree.ctr) simp
}
ultimately show \(?case \) by simp
qed

5
 lemma ap-tree-extensional:
  \((\forall x. f \circ x = g \circ x) \implies f = g\)
by(rule ap-tree-strong-extensional) simp

1.2 Standard tree combinators

1.2.1 Recurse combinator

This will be the main combinator to define trees recursively
Uniqueness for this gives us the unique fixed-point theorem for guarded
recursive definitions.

 lemma map-unfold-tree [simp]:
  fixes l r x
  defines unf \equiv unfold-tree (\lambda f. f x) (\lambda f. f \circ l) (\lambda f. f \circ r)
  shows map-tree G (unf F) = unf (G \circ F)
  by(coinduction arbitrary: F G)(auto 4 3 simp add: unf-def o-assoc)

friend-of-corec map-tree :: ('a \Rightarrow 'a) \Rightarrow 'a tree \Rightarrow 'a tree
  where
    map-tree f t = Node (f (root t)) (map-tree f (left t)) (map-tree f (right t))

subgoal by (rule tree.expand; simp)
subgoal by (fold relator-eq; transfer-prover)
done

context fixes l :: 'a \Rightarrow 'a and r :: 'a \Rightarrow 'a and x :: 'a begin

primcorec tree-recurse :: 'a tree
  where
    tree-recurse = Node x (map-tree l tree-recurse) (map-tree r tree-recurse)
end

 lemma tree-recurse-simps [simp]:
  root (tree-recurse l r x) = x
  left (tree-recurse l r x) = map-tree l (tree-recurse l r x)
  right (tree-recurse l r x) = map-tree r (tree-recurse l r x)
  by(subst tree-recurse.code; simp; fail)+

lemma tree-recurse-unfold:
  tree-recurse l r x = Node x (map-tree l (tree-recurse l r x)) (map-tree r (tree-recurse l r x))
  by(fact tree-recurse.code)

lemma tree-recurse-fusion:
  assumes h \circ l = l' \circ h and h \circ r = r' \circ h
  shows map-tree h (tree-recurse l r x) = tree-recurse l' r' (h x)
  by(rule tree-recurse.unique)(simp add: tree.expand assms)

1.2.2 Tree iteration

context fixes l :: 'a \Rightarrow 'a and r :: 'a \Rightarrow 'a begin

primcorec tree-iterate :: 'a \Rightarrow 'a tree
  where
    tree-iterate s = Node s (tree-iterate l s) (tree-iterate r s)
end
lemma unfold-tree-tree-iterate:
unfold-tree out l r = map-tree out ◦ tree-iterate l r
by(rule ext)(rule unfold-tree-unique[symmetric]; simp)

lemma tree-iterate-fusion:
assumes h ◦ l = l' ◦ h
assumes h ◦ r = r' ◦ h
shows map-tree h (tree-iterate l r x) = tree-iterate l' r' (h x)
apply(coinduction arbitrary: x)
using assms by(auto simp add: fun-eq-iff)

1.2.3 Tree traversal
datatype dir = L | R
type-synonym path = dir list
definition traverse-tree :: path ⇒ 'a tree ⇒ 'a tree
where traverse-tree path ≡ foldr (λd f f ◦ case-dir left right d) path id

lemma traverse-tree-simps[simp]:
traverse-tree [] = id
traverse-tree (d # path) = traverse-tree path ◦ (case d of L ⇒ left | R ⇒ right)
by (simp-all add: traverse-tree-def)

lemma traverse-tree-map-tree [simp]:
traverse-tree path (map-tree f t) = map-tree f (traverse-tree path t)
by (induct path arbitrary: t) (simp-all split: dir.splits)

lemma traverse-tree-append [simp]:
traverse-tree (path @ ext) t = traverse-tree ext (traverse-tree path t)
by (induct path arbitrary: t) simp-all

traverse-tree is an applicative-functor homomorphism.

lemma traverse-tree-pure-tree [simp]:
traverse-tree path (pure x) = pure x
by (induct path arbitrary: x) (simp-all split: dir.splits)

lemma traverse-tree-ap [simp]:
traverse-tree path (f ◦ x) = traverse-tree path f ◦ traverse-tree path x
by (induct path arbitrary: f x) (simp-all split: dir.splits)

context fixes l r :: 'a ⇒ 'a begin
primrec traverse-dir :: dir ⇒ 'a ⇒ 'a
where
traverse-dir L = l
| traverse-dir R = r
abbreviation traverse-path :: path ⇒ 'a ⇒ 'a
where traverse-path ≡ fold traverse-dir

end

lemma traverse-tree-tree-iterate:
traverse-tree path (tree-iterate l r s) =
tree-iterate l r (traverse-path l r path s)
by (induct path arbitrary: s) (simp-all split: dir.splits)

shows that if the tree construction function is suitably monoidal then
recursion and iteration define the same tree.

lemma tree-recurse-iterate:
assumes monoid:
\[ \forall x y z. f (f x y) z = f x (f y z) \]
\[ \forall x. f x x = x \]
\[ \forall x. f x x = x \]
shows tree-recurse (f l) (f r) ε = tree-iterate (λx. f x l) (λx. f x r) ε
apply(rule tree-recurse.unique[symmetric])
apply(rule tree.expand)
apply(simp add: tree-iterate-fusion[where r'=λx. f x r and l'=λx. f x l] fun-eq-iff
monoid)
done

1.2.4 Mirroring

primcorec mirror :: 'a tree ⇒ 'a tree
where
root (mirror t) = root t
| left (mirror t) = mirror (right t)
| right (mirror t) = mirror (left t)

lemma mirror-unfold: mirror (Node x l r) = Node x (mirror r) (mirror l)
by(rule tree.expand) simp

lemma mirror-pure: mirror (pure x) = pure x
by(coinduction rule: tree.coinduct) simp

lemma mirror-ap-tree: mirror (f o x) = mirror f o mirror x
by(coinduction arbitrary: f x) auto

end

1.3 Pointwise arithmetic on infinite binary trees

theory Cotree-Algebra
imports Cotree
begin
1.3.1 Constants and operators

instantiation tree :: (zero) zero begin
definition [applicative-unfold]: 0 = pure-tree 0
instance ..
end

instantiation tree :: (one) one begin
definition [applicative-unfold]: 1 = pure-tree 1
instance ..
end

instantiation tree :: (plus) plus begin
definition [applicative-unfold]: plus x y = pure (+) ⋄ x ⋄ (y :: 'a tree)
instance ..
end

lemma plus-tree-simps [simp]:
   root (t + t') = root t + root t'
   left (t + t') = left t + left t'
   right (t + t') = right t + right t'
by(simp-all add: plus-tree-def)

friend-of-corec plus where t + t' = Node (root t + root t') (left t + left t') (right t + right t')
subgoal by(rule tree.expand; simp)
subgoal by transfer-prover
done

instantiation tree :: (minus) minus begin
definition [applicative-unfold]: minus x y = pure (%) ⋄ x ⋄ (y :: 'a tree)
instance ..
end

lemma minus-tree-simps [simp]:
   root (t - t') = root t - root t'
   left (t - t') = left t - left t'
   right (t - t') = right t - right t'
by(simp-all add: minus-tree-def)

instantiation tree :: (uminus) uminus begin
definition [applicative-unfold tree]: uminus = ((% (pure uminus)) :: 'a tree ⇒ 'a tree)
instance ..
end

instantiation tree :: (times) times begin
definition [applicative-unfold]: times x y = pure (*) ⋄ x ⋄ (y :: 'a tree)
instance ..
end
lemma times-tree-simps [simp]:
  root \((t * t')\) = root \(t\) * root \(t'\)
  left \((t * t')\) = left \(t\) * left \(t'\)
  right \((t * t')\) = right \(t\) * right \(t'\)
  by(simp-all add: times-tree-def)

instance tree :: (Rings dvd) Rings dvd ..

instantiation tree :: (modulo) modulo begin
  definition [applicative-unfold]: \(x \div y\) = pure-tree \((\text{div}) \circ x \circ (y :: 'a tree)\)
  definition [applicative-unfold]: \(x \mod y\) = pure-tree \((\text{mod}) \circ x \circ (y :: 'a tree)\)
  instance ..
end

lemma mod-tree-simps [simp]:
  root \((t \mod t')\) = root \(t\) mod root \(t'\)
  left \((t \mod t')\) = left \(t\) mod left \(t'\)
  right \((t \mod t')\) = right \(t\) mod right \(t'\)
  by(simp-all add: modulo-tree-def)

1.3.2 Algebraic instances

instance tree :: (semigroup-add) semigroup-add
  using add.assoc by intro-classes applicative-lifting

instance tree :: (ab-semigroup-add) ab-semigroup-add
  using add.commute by intro-classes applicative-lifting

instance tree :: (semigroup-mult) semigroup-mult
  using mult.assoc by intro-classes applicative-lifting

instance tree :: (ab-semigroup-mult) ab-semigroup-mult
  using mult.commute by intro-classes applicative-lifting

instance tree :: (monoid-add) monoid-add
  by intro-classes (applicative-lifting, simp)+

instance tree :: (comm-monoid-add) comm-monoid-add
  by intro-classes (applicative-lifting, simp)

instance tree :: (comm-monoid-diff) comm-monoid-diff
  by intro-classes (applicative-lifting, simp add: diff-diff-add)+

instance tree :: (monoid-mult) monoid-mult
  by intro-classes (applicative-lifting, simp)+

instance tree :: (comm-monoid-mult) comm-monoid-mult
  by intro-classes (applicative-lifting, simp)
instance tree :: (cancel-semigroup-add) cancel-semigroup-add
proof
  fix a b c :: 'a tree
  assume a + b = a + c
  thus b = c
proof (coinduction arbitrary: a b c)
  case (Eq-tree a b c)
  hence root (a + b) = root (a + c)
  left (a + b) = left (a + c)
  right (a + b) = right (a + c)
  by simp-all
  thus ?case by (auto)
qed

next
fix a b c :: 'a tree
assume b + a = c + a
thus b = c
proof (coinduction arbitrary: a b c)
  case (Eq-tree a b c)
  hence root (b + a) = root (c + a)
  left (b + a) = left (c + a)
  right (b + a) = right (c + a)
  by simp-all
  thus ?case by (auto)
qed
qed

instance tree :: (cancel-ab-semigroup-add) cancel-ab-semigroup-add
by intro-classes (applicative-lifting, simp add: diff-diff-eq)+

instance tree :: (cancel-comm-monoid-add) cancel-comm-monoid-add ..

instance tree :: (group-add) group-add
by intro-classes (applicative-lifting, simp)+

instance tree :: (ab-group-add) ab-group-add
by intro-classes (applicative-lifting, simp)+

instance tree :: (semiring) semiring
by intro-classes (applicative-lifting, simp add: ring-distribs)+

instance tree :: (mult-zero) mult-zero
by intro-classes (applicative-lifting, simp)+

instance tree :: (semiring-0) semiring-0 ..

instance tree :: (semiring-0-cancel) semiring-0-cancel ..
instance tree :: (comm-semiring) comm-semiring
by intro-classes(rule distrib-right)

instance tree :: (comm-semiring-0) comm-semiring-0 ..
instance tree :: (comm-semiring-0-cancel) comm-semiring-0-cancel ..

lemma pure-tree-inject[simp]: pure-tree x = pure-tree y ←→ x = y
proof
  assume pure-tree x = pure-tree y
  hence root (pure-tree x) = root (pure-tree y) by simp
  thus x = y by simp
qed simp

instance tree :: (zero-neq-one) zero-neq-one
by intro-classes (applicative-unfold tree)

instance tree :: (semiring-1) semiring-1 ..
instance tree :: (comm-semiring-1) comm-semiring-1 ..
instance tree :: (semiring-1-cancel) semiring-1-cancel ..
instance tree :: (comm-semiring-1-cancel) comm-semiring-1-cancel
by(intro-classes; applicative-lifting, rule right-diff-distrib')

instance tree :: (ring) ring ..
instance tree :: (comm-ring) comm-ring ..
instance tree :: (ring-1) ring-1 ..
instance tree :: (comm-ring-1) comm-ring-1 ..
instance tree :: (numeral) numeral ..
instance tree :: (neg-numeral) neg-numeral ..
instance tree :: (semiring-numeral) semiring-numeral ..

lemma of-nat-tree: of-nat n = pure-tree (of-nat n)
proof (induction n)
  case 0 show ?case by (simp add: zero-tree-def)
next
  case (Suc n)
  have 1 + pure (of-nat n) = pure (1 + of-nat n) by applicative-nf rule
  with Suc.IH show ?case by simp
qed
instance tree :: (semiring-char-0) semiring-char-0
by intro-classes (simp add: inj-on-def of-nat-tree)

lemma numeral-tree-simps [simp]:
  root (numeral n) = numeral n
  left (numeral n) = numeral n
  right (numeral n) = numeral n
by (induct n)(auto simp add: numeral.simps plus-tree-def one-tree-def)

lemma numeral-tree-conv-pure [applicative-unfold]: numeral n = pure (numeral n)
by (rule pure-tree-unique)(rule tree.expand; simp)

instance tree :: (ring-char-0) ring-char-0 ..
end

2 The Stern-Brocot Tree

theory Stern-Brocot-Tree
imports
  HOL.Rat
  HOL-Library.Sublist
  Cotree-Algebra
  Applicative-Lifting.Stream-Algebra
begin

The Stern-Brocot tree is discussed at length by Graham et al. (1994, §4.5).
In essence the tree enumerates the rational numbers in their lowest terms
by constructing the mediant of two bounding fractions.

type-synonym fraction = nat × nat

definition mediant :: fraction × fraction ⇒ fraction
where mediant ≡ λ((a, c), (b, d)). (a + b, c + d)

definition stern-brocot :: fraction tree
where
  stern-brocot = unfold-tree
  (λ(lb, ub). mediant (lb, ub))
  (λ(lb, ub). (lb, mediant (lb, ub)))
  (λ(lb, ub). (mediant (lb, ub), ub))
  ((0, 1), (1, 0))

This process is visualised in Figure 2. Intuitively each node is labelled with
the mediant of it’s rightmost and leftmost ancestors.

Our ultimate goal is to show that the Stern-Brocot tree contains all rationals
(in lowest terms), and that each occurs exactly once in the tree. A proof is
sketched in Graham et al. (1994, §4.5).
2.1 Specification via a recursion equation

Hinze (2009) derives the following recurrence relation for the Stern-Brocot tree. We will show in §2.5 that his derivation is sound with respect to the standard iterative definition of the tree shown above.

abbreviation succ :: fraction ⇒ fraction
where succ ≡ λ(m, n). (m + n, n)

abbreviation recip :: fraction ⇒ fraction
where recip ≡ λ(m, n). (n, m)

corec stern-brocot-recurse :: fraction tree
where
stern-brocot-recurse =
  Node (1, 1)
  (map-tree recip (map-tree succ (map-tree recip stern-brocot-recurse)))
  (map-tree succ stern-brocot-recurse)

Actually, we would like to write the specification below, but (∘) cannot be registered as friendly due to varying type parameters

lemma stern-brocot-unfold:
stern-brocot-recurse =
  Node (1, 1)
  (pure recip ∘ (pure succ ∘ (pure recip ∘ stern-brocot-recurse)))
  (pure succ ∘ stern-brocot-recurse)
by(fact stern-brocot-recuse.code[unfolded map-tree-ap-tree-pure-tree[symmetric]])

lemma stern-brocot-simps [simp]:
root stern-brocot-recurse = (1, 1)
left stern-brocot-recurse ≡ pure recip ∘ (pure succ ∘ (pure recip ∘ stern-brocot-recurse))
right stern-brocot-recurse ≡ pure succ ∘ stern-brocot-recurse
by (subst stern-brocot-unfold, simp)+

lemma stern-brocot-conv:
2.2 Basic properties

The recursive definition is useful for showing some basic properties of the tree, such as that the pairs of numbers at each node are coprime, and have non-zero denominators. Both are simple inductions on the path.

**Lemma** stern-brocot-denominator-non-zero:

```
case root (traverse-tree path stern-brocot-recurse) of (m, n) ⇒ m > 0 ∧ n > 0
by (induct path) (auto split: dir.splits)
```

**Lemma** stern-brocot-coprime:

```
case root (traverse-tree path stern-brocot-recurse) of (m, n) ⇒ coprime m n
by (induct path) (auto split: dir.splits simp add: coprime-iff-gcd-eq-1, metis gcd.commute gcd-add1)
```

2.3 All the rationals

For every pair of positive naturals, we can construct a path into the Stern-Brocot tree such that the naturals at the end of the path define the same rational as the pair we started with. Intuitively, the choices made by Euclid’s algorithm define this path.

**Function** mk-path :: nat ⇒ nat ⇒ path where

```
m = n =⇒ mk-path (Suc m) (Suc n) = []
| m < n =⇒ mk-path (Suc m) (Suc n) = L # mk-path (Suc m) (n - m)
| m > n =⇒ mk-path (Suc m) (Suc n) = R # mk-path (m - n) (Suc n)

mk-path 0 - = undefined
mk-path - 0 = undefined
by atomize-elim (auto, arith)
```

**Termination** mk-path by lexicographic-order

**Lemmas** mk-path-induct [case-names equal less greater] = mk-path.induct

**Abbreviation** rat-of :: fraction ⇒ rat

```
rat-of ≡ λ(x, y). Fract (int x) (int y)
```

**Theorem** stern-brocot-rational:

```
[ m > 0; n > 0 ] =⇒
root (traverse-tree (mk-path m n) (pure rat-of ◦ stern-brocot-recurse)) = Fract (int m) (int n)
```

**Proof** (induction m n rule: mk-path-induct)

```
case (less m n)
```
2.4 No repetitions

We establish that the Stern-Brocot tree does not contain repetitions, i.e., that each rational number appears at most once in it. Note that this property is stronger than merely requiring that pairs of naturals not be repeated, though it is implied by that property and stern-brocot-coprime.

Intuitively, the tree enjoys the binary search tree ordering property when we map our pairs of naturals into rationals. This suffices to show that each rational appears at most once in the tree. To establish this seems to require more structure than is present in the recursion equations, and so we follow Backhouse and Ferreira (2008) and Hinze (2009) by introducing another definition of the tree, which summarises the path to each node using a matrix.

We then derive an iterative version and use invariant reasoning on that. We begin by defining some matrix machinery. This is all elementary and primitive (we do not need much algebra).

```plaintext
type-synonym matrix = fraction × fraction

definition times-matrix :: matrix ⇒ matrix ⇒ matrix (infixl ⊗ 70)
where times-matrix = (λ((a, c), (b, d)) ((a’, c’), (b’, d’)).
((a * a’ + b * c’, c * a’ + d * c’),
(a * b’ + b * d’, c * b’ + d * d’)))

definition times-vector :: matrix ⇒ vector ⇒ vector (infixr ⊙ 70)
where times-vector = (λ((a, c), (b, d)) (a’, c’). (a * a’ + b * c’, c * a’ + d * c’))
```

context begin

private definition F :: matrix where F = ((0, 1), (1, 0))
private definition I :: matrix where I = ((1, 0), (0, 1))
private definition LL :: matrix where LL = ((1, 1), (0, 1))
private definition UR :: matrix where UR = ((1, 0), (1, 1))

definition Det :: matrix ⇒ nat where Det ≡ λ((a, c), (b, d)). a * d - b * c

lemma Dets [iff]:

16
\[ \text{Det} \ I = 1 \]
\[ \text{Det} \ LL = 1 \]
\[ \text{Det} \ UR = 1 \]

**Unfolding** \( \text{Det-def} \ I \)-def \( \text{LL-def} \) \( \text{UR-def} \) by \textit{simp-all}

**Lemma** \( \text{LL-UR-Det} \):
\[ \text{Det} \ m = 1 \implies \text{Det} \ (m \otimes LL) = 1 \]
\[ \text{Det} \ m = 1 \implies \text{Det} \ (LL \otimes m) = 1 \]
\[ \text{Det} \ m = 1 \implies \text{Det} \ (m \otimes UR) = 1 \]
\[ \text{Det} \ m = 1 \implies \text{Det} \ (UR \otimes m) = 1 \]

by (cases \( m \), simp add: \( \text{Det-def} \) \( \text{LL-def} \) \( \text{UR-def} \) \( \text{times-matrix-def} \) \( \text{split-def} \) \( \text{field-simps} \))

**Lemma** \( \text{mediant-I-F} \) [simp]:
\[ \text{mediant} \ F = (1, 1) \]
\[ \text{mediant} \ I = (1, 1) \]

by (simp-all add: \( \text{F-def} \) \( \text{I-def} \) \( \text{mediant-def} \))

**Lemma** \( \text{times-matrix-I} \) [simp]:
\[ I \otimes x = x \]
\[ x \otimes I = x \]

by (simp-all add: \( \text{times-matrix-def} \) \( \text{I-def} \) \( \text{split-def} \))

**Lemma** \( \text{times-matrix-assoc} \) [simp]:
\[ (x \otimes y) \otimes z = x \otimes (y \otimes z) \]

by (simp add: \( \text{times-matrix-def} \) \( \text{field-simps} \) \( \text{split-def} \))

**Lemma** \( \text{LL-UR-pos} \):
\[ 0 < \text{snd} (\text{mediant} \ m) \implies 0 < \text{snd} (\text{mediant} \ (m \otimes LL)) \]
\[ 0 < \text{snd} (\text{mediant} \ m) \implies 0 < \text{snd} (\text{mediant} \ (m \otimes UR)) \]

by (cases \( m \)) (simp-all add: \( \text{Det-def} \) \( \text{UR-def} \) \( \text{times-matrix-def} \) \( \text{split-def} \) \( \text{field-simps} \) \( \text{mediant-def} \))

**Lemma** \( \text{recip-succ-recip} \) : \( \text{recip} \circ \text{succ} \circ \text{recip} = (\lambda (x, y). (x, x + y)) \)

by (clarsimp simp: fun-eq-iff)

**Backhouse and Ferreira** work with the identity matrix \( I \) at the root. This has the advantage that all relevant matrices have determinants of \( 1 \).

**Definition** \( \text{stern-brocot-iterate-aux} :: \text{matrix} \Rightarrow \text{matrix tree} \)
where \( \text{stern-brocot-iterate-aux} \equiv \text{tree-iterate} (\lambda s. s \otimes LL) (\lambda s. s \otimes UR) \)

**Definition** \( \text{stern-brocot-iterate} :: \text{fraction tree} \)
where \( \text{stern-brocot-iterate} \equiv \text{map-tree mediant} (\text{stern-brocot-iterate-aux} \ I) \)

**Lemma** \( \text{stern-brocot-recurse-iterate} :: \text{stern-brocot-recurse} = \text{stern-brocot-iterate} \ (\text{is} \ ?\text{lhs} = ?\text{rhs}) \)

**Proof**
- have \( ?\text{rhs} = \text{map-tree mediant} (\text{tree-recurse} (\otimes LL) (\otimes UR) \ I) \)
  using \( \text{tree-recurse-iterate} [\text{where} \ f = (\otimes) \text{ and} \ l = LL \text{ and} \ r = UR \text{ and} \ v = I] \)
  by (simp add: \( \text{stern-brocot-iterate-def} \) \( \text{stern-brocot-iterate-aux-def} \))
also have \ldots = \text{tree-recurse}(\text{LL}) \oplus \text{UR}(1, 1)

unfolding \text{mediant-I-F}(2)[\text{symmetric}],
by (rule \text{tree-recurse-fusion})(\text{simp-all add: fun-eq-iff \text{mediant-def \text{times-vector-def \text{LL-def \text{UR-def}}}}}[2])
also have \ldots = ?lhs
by (simp add: \text{stern-brocot-conv recip-succ-recip \text{times-vector-def \text{LL-def \text{UR-def}}})
finally show ?thesis by simp
qed

The following are the key ordering properties derived by Backhouse and Ferreira (2008). They hinge on the matrices containing only natural numbers.

\text{lemma} \text{tree-ordering-left:}
\text{assumes} \text{DX:} \text{Det} X = 1
\text{assumes} \text{DY:} \text{Det} Y = 1
\text{assumes} \text{MX:} 0 < \text{snd \text{mediant} X}
\text{shows} \text{rat-of \text{mediant} \text{(X \odot LL \odot Y)}} < \text{rat-of \text{mediant} X}
\text{proof –}
\text{from} \text{DX\ DY have F:} 0 < \text{snd \text{mediant} \text{(X \odot LL \odot Y)}}
by (auto simp: \text{Det-def \text{times-vector-def \text{LL-def \text{UR-def}}}}\text{split-def \text{mediant-def}})
\text{obtain} x11 x12 x21 x22 \text{where} X = ((x11, x12), (x21, x22)) \text{by (cases X)}
\text{auto}
\text{obtain} y11 y12 y21 y22 \text{where} Y = ((y11, y12), (y21, y22)) \text{by (cases Y)}
\text{auto}
\text{from} \text{DX\ DY have *} : (x12 \ast x21) \ast (y12 + y22) < (x11 \ast x22) \ast (y12 + y22)
by (simp add: X Y \text{Det-def})(\text{cases y12, simp-all add: \text{field-simps}})
\text{from} \text{DX\ DY\ MX\ F show ?thesis}
\text{apply (simp add: \text{split-def X Y of-nat-mult \text{symmetric}} del: \text{of-nat-mult})}
\text{apply (clarsimp simp add: \text{Det-def \text{times-vector-def \text{LL-def \text{UR-def}}}}\text{split-def \text{mediant-def}})}
\text{using * by (simp add: \text{field-simps})}
\text{qed}

\text{lemma} \text{tree-ordering-right:}
\text{assumes} \text{DX:} \text{Det} X = 1
\text{assumes} \text{DY:} \text{Det} Y = 1
\text{assumes} \text{MX:} 0 < \text{snd \text{mediant} X}
\text{shows} \text{rat-of \text{mediant} \text{(X \odot UR \odot Y)}} < \text{rat-of \text{mediant} \text{(X \odot UR \odot Y)}}
\text{proof –}
\text{from} \text{DX\ DY have F:} 0 < \text{snd \text{mediant} \text{(X \odot UR \odot Y)}}
by (auto simp: \text{Det-def \text{times-vector-def \text{UR-def \text{split-def \text{mediant-def}}}}})
\text{obtain} x11 x12 x21 x22 \text{where} X = ((x11, x12), (x21, x22)) \text{by (cases X)}
\text{auto}
\text{obtain} y11 y12 y21 y22 \text{where} Y = ((y11, y12), (y21, y22)) \text{by (cases Y)}
\text{auto}
\text{show ?thesis using DX\ DY\ MX\ F}
\text{apply (simp add: X Y \text{split-def of-nat-mult \text{symmetric}} del: \text{of-nat-mult})}
\text{apply (simp add: \text{Det-def \text{times-vector-def \text{LL-def \text{UR-def}}}}\text{split-def \text{mediant-def \text{split-def}} \text{algebra-simps})}
\text{apply (simp add: add-mult-distrib2[\text{symmetric}] \text{mult.assoc[\text{symmetric}]}})
apply (cases y21; simp)
done

qed

lemma stern-brocot-iterate-aux-Det:
  assumes Det m = 1 0 < snd (mediant m)
  shows Det (root (traverse-tree path (stern-brocot-iterate-aux m))) = 1
  and 0 < snd (mediant (root (traverse-tree path (stern-brocot-iterate-aux m))))
  using assms
  by (induct path arbitrary: m)

lemma stern-brocot-iterate-aux-decompose:
  ∃ m". m ⊗ m" = root (traverse-tree path (stern-brocot-iterate-aux m)) ∧ Det m" = 1
proof (induction path arbitrary: m)
  case Nil
  next
  case (Cons d ds m)
  from Cons.IH [where m = m ⊗ UR] Cons.IH [where m = m ⊗ LL] show ?case
qed

lemma stern-brocot-fractions-not-repeated-strict-prefix:
  assumes root (traverse-tree path stern-brocot-iterate) = root (traverse-tree path' stern-brocot-iterate)
  assumes pp': strict-prefix path path'
  shows False
proof
  from pp' obtain d ds where pp': path' = path @ [d] @ ds by (auto elim!: strict-prefixE')
  define m where m = root (traverse-tree path (stern-brocot-iterate-aux I))
  then have Dm: Det m = 1 and Pm: 0 < snd (mediant m)
    using stern-brocot-iterate-aux-Det [where path = path and m = I] by simp-all
  define m' where m' = root (traverse-tree path' (stern-brocot-iterate-aux I))
  then have Dm': Det m' = 1
    using stern-brocot-iterate-aux-Det [where path = path' and m = I] by simp
  let ?M = case d of L ⇒ m ⊗ LL | R ⇒ m ⊗ UR
  from pp' have root (traverse-tree ds (stern-brocot-iterate-aux ?M)) = m'
    by (simp add: m-def m'-def stern-brocot-iterate-aux-def traverse-tree-iterate-dir.splits)
  then obtain m" where mm'm'': ?M ⊗ m'' = m' and Dm'': Det m'' = 1
    using stern-brocot-iterate-aux-decompose [where path = ds and m = ?M] by clar-simp
  hence case d of L ⇒ rat-of (mediant m') < rat-of (mediant m) | R ⇒ rat-of (mediant m) < rat-of (mediant m')
using tree-ordering-left[OF Dm Dm' Pm] tree-ordering-right[OF Dm Dm' Pm]
by (simp split: dir.splits)
with assms show False
by (simp add: stern-brocot-iterate-def m-def m'-def split: dir.splits)
qed

lemma stern-brocot-fractions-not-repeated-parallel:
  assumes root (traverse-tree path stern-brocot-iterate) = root (traverse-tree path' stern-brocot-iterate)
  assumes p: path =pref @ d # ds
  assumes p': path' = pref @ d' # ds'
  assumes dd': d ≠ d'
  shows False
proof
  define m where m = root (traverse-tree pref (stern-brocot-iterate-aux I))
  then have Dm: Det m = 1 and Pm: 0 < snd (mediant m)
  using stern-brocot-iterate-aux-det[where path=pref and m=I] by simp-all
  define pm where pm = root (traverse-tree path (stern-brocot-iterate-aux I))
  then have Dpm: Det pm = 1
  using stern-brocot-iterate-aux-det[where path=path and m=I] by simp
  let ?M = case d of L ⇒ m ⊗ LL | R ⇒ m ⊗ UR
  from p
  have root (traverse-tree ds (stern-brocot-iterate-aux ?M)) = pm
  by (simp add: stern-brocot-iterate-aux-def m-def pm-def traverse-tree-tree-iterate split: dir.splits)
  then obtain pm'
    where pm': ?M ⊗ pm' = pm and Dpm': Det pm' = 1
    using stern-brocot-iterate-aux-decompose[where path=ds and m=?M] by clar-simp
  hence case d of L ⇒ rat-of (mediant pm) < rat-of (mediant m) | R ⇒ rat-of (mediant m) < rat-of (mediant pm)
  using tree-ordering-left[OF Dm Dpm' Pm, unfolded pm']
  tree-ordering-right[OF Dm Dpm' Pm, unfolded pm']
  by (simp split: dir.splits)
moreover
  define p'm where p'm = root (traverse-tree path' (stern-brocot-iterate-aux I))
  then have Dp'm: Det p'm = 1
  using stern-brocot-iterate-aux-det[where path=path' and m=I] by simp
  let ?M' = case d' of L ⇒ m ⊗ LL | R ⇒ m ⊗ UR
  from p'
  have root (traverse-tree ds' (stern-brocot-iterate-aux ?M')) = p'm
  by (simp add: stern-brocot-iterate-aux-def m-def p'm-def traverse-tree-tree-iterate split: dir.splits)
  then obtain p'm'
    where p'm': ?M' ⊗ p'm' = p'm and Dp'm': Det p'm' = 1
    using stern-brocot-iterate-aux-decompose[where path=ds' and m=?M'] by clar-simp
  hence case d' of L ⇒ rat-of (mediant p'm) < rat-of (mediant m) | R ⇒ rat-of (mediant m) < rat-of (mediant p'm)
ultimately show False using pm' p'm' assms 
by(simp add: m-def pm-def p'm-def stern-brocot-iterate-def split: dir.splits)

The function Fract is injective under certain conditions.

lemma rat-inv-eq:
assumes Fract a b = Fract c d 
assumes b > 0
assumes d > 0
assumes coprime a b 
assumes coprime c d
shows a = c ∧ b = d

proof
  from \langle b > 0, d > 0 \rangle \langle Fract a b = Fract c d \rangle
  have \*: a * d = c * b by (simp add: eq-rat)
  from arg-cong[where f=sgn, OF this] \langle b > 0, d > 0 \rangle
  have sgn a = sgn c by (simp add: sgn-mult)
  with \* show \?thesis
  using \langle b > 0, d > 0 \rangle coprime-crossproduct-int[OF \langle coprime a b \rangle \langle coprime c d \rangle]
  by (simp add: abs-sgn)
theorem stern-brocot-rationals-not-repeated:
  assumes root (traverse-tree path (pure rat-of ⋄ stern-brocot-recurse))
  = root (traverse-tree path' (pure rat-of ⋄ stern-brocot-recurse))
  shows path = path'
using assms
using stern-brocot-coprime[where path=path]
  stern-brocot-coprime[where path=path']
  stern-brocot-denominator-non-zero[where path=path]
  stern-brocot-denominator-non-zero[where path=path']

2.5 Equivalence of recursive and iterative version

Hinze shows that it does not matter whether we use $I$ or $F$ at the root provided we swap the left and right matrices too.

definition stern-brocot-Hinze-iterate :: fraction tree
where stern-brocot-Hinze-iterate = map-tree mediant (tree-iterate ($\lambda s. s \otimes UR$) ($\lambda s. s \otimes LL$) $F$)

lemma mediant-times-F: mediant ◦ ($\lambda s. s \otimes F$) = mediant
by(simp add: times-matrix-def F-def mediant-def split-def o-def add.commute)

lemma stern-brocot-iterate: stern-brocot = stern-brocot-iterate
proof
  have stern-brocot = stern-brocot-Hinze-iterate
    unfolding stern-brocot-def stern-brocot-Hinze-iterate-def
  also have ... = map-tree mediant (map-tree ($\lambda s. s \otimes F$) (tree-iterate ($\lambda s. s \otimes LL$) $\lambda s. s \otimes LL$) $\lambda s. s \otimes UR$) $I$)
    unfolding stern-brocot-Hinze-iterate-def
    by(subst tree-iterate-fusion[where $t'=\lambda s. s \otimes UR$ and $r'=\lambda s. s \otimes LL$])
    (simp-all add: fun-eq-iff times-matrix-def UR-def LL-def F-def I-def)
  also have ... = stern-brocot-iterate
    by(simp only: tree.map-comp mediant-times-F stern-brocot-iterate-def stern-brocot-iterate-aux-def)
  finally show ?thesis .
qed

theorem stern-brocot-mediant-recurse: stern-brocot = stern-brocot-recurse
by(simp add: stern-brocot-recurse-iterate stern-brocot-iterate)

end

no-notation times-matrix (infixl ⊗ 70)
  and times-vector (infixl ⊗ 70)
3 Linearising the Stern-Brocot Tree

3.1 Turning a tree into a stream

```plaintext
corec tree-chop :: 'a tree ⇒ 'a tree
where tree-chop t = Node (root (left t)) (right t) (tree-chop (left t))

lemma tree-chop-sel [simp]:
root (tree-chop t) = root (left t)
left (tree-chop t) = right t
right (tree-chop t) = tree-chop (left t)
by (subst tree-chop.code; simp; fail)

tree-chop is a idiom homomorphism
lemma tree-chop-pure-tree [simp]:
tree-chop (pure x) = pure x
by (coinduction rule: tree.coinduct-strong) auto

lemma tree-chop-ap-tree [simp]:
tree-chop (f ⋄ x) = tree-chop f ⋄ tree-chop x
by (coinduction arbitrary: f x rule: tree.coinduct-strong) auto

lemma tree-chop-plus: tree-chop (t + t') = tree-chop t + tree-chop t'
by (simp add: plus-tree-def)

corec stream :: 'a tree ⇒ 'a stream
where stream t = root t ## stream (tree-chop t)

lemma stream-sel [simp]:
shd (stream t) = root t
stl (stream t) = stream (tree-chop t)
by (subst stream.code; simp; fail)

stream is an idiom homomorphism.
lemma stream-pure [simp]: stream (pure x) = pure x
by coinduction auto

lemma stream-ap [simp]: stream (f ⋄ x) = stream f ⋄ stream x
by (coinduction arbitrary: f x) auto

lemma stream-plus [simp]: stream (t + t') = stream t + stream t'
by (simp add: plus-stream-def plus-tree-def)

lemma stream-minus [simp]: stream (t - t') = stream t - stream t'
by (simp add: minus-stream-def minus-tree-def)

lemma stream-times [simp]: stream (t * t') = stream t * stream t'
by (simp add: times-stream-def times-tree-def)
```
lemma stream-mod \[\text{simp}]: \text{stream} (t \mod t') = \text{stream} t \mod \text{stream} t' \\
\text{by} (\text{simp add: modulo-stream-def modulo-tree-def})

lemma stream-1 \[\text{simp}]: \text{stream} 1 = 1 \\
\text{by} (\text{simp add: one-tree-def one-stream-def})

lemma stream-numeral \[\text{simp}]: \text{stream} (\text{numeral} n) = \text{numeral} n \\
\text{by} (\text{induct} n) (\text{simp-all only: numeral.simps stream-plus stream-1})

3.2 Split the Stern-Brocot tree into numerators and denominators

corec num-den :: bool \Rightarrow nat tree \\
where 
num-den x = 
  Node 1 
  (if x then num-den True else num-den True + num-den False) 
  (if x then num-den True + num-den False else num-den False)

abbreviation num where num \equiv num-den True 
abbreviation den where den \equiv num-den False

lemma num-unfold: num = Node 1 num (num + den) \\
\text{by} (\text{subst} num-den.code; \text{simp})

lemma den-unfold: den = Node 1 (num + den) den \\
\text{by} (\text{subst} num-den.code; \text{simp})

lemma num-simps \[\text{simp}]: 
  root num = 1 
  left num = num 
  right num = num + den 
\text{by} (\text{subst} num-unfold, \text{simp})+

lemma den-simps \[\text{simp}]: 
  root den = 1 
  left den = num + den 
  right den = den 
\text{by} (\text{subst} den-unfold, \text{simp})+

lemma stern-brocot-num-den: 
  pure-tree Pair \diamond\ num \diamond den = stern-brocot-recurse 
\text{apply} (\text{rule stern-brocot-recurse.unique}) 
\text{apply} (\text{subst} den-unfold) 
\text{apply} (\text{subst} num-unfold) 
\text{apply} (\text{simp; intro conjI}) 
\text{apply} (\text{applicative-lifting; simp})+ 
\text{done}
lemma den-eq-chop-num: den = tree-chop num
by(coinduction rule: tree.coinduct-strong) simp

lemma num-conv: num = pure fst ∘ stern-brocot-recurse
unfolding stern-brocot-num-den[symmetric]
apply(simp add: map-tree-ap-tree-pure-tree stern-brocot-num-den[symmetric])
apply(applicative-lifting; simp)
done

lemma den-conv: den = pure snd ∘ stern-brocot-recurse
unfolding stern-brocot-num-den[symmetric]
apply(simp add: map-tree-ap-tree-pure-tree stern-brocot-num-den[symmetric])
apply(applicative-lifting; simp)
done

corec num-mod-den :: nat tree
where num-mod-den = Node 0 num num-mod-den

lemma num-mod-den-simps [simp]:
  root num-mod-den = 0
  left num-mod-den = num
  right num-mod-den = num-mod-den
by(subst num-mod-den.code; simp; fail)+

The arithmetic transformations need the precondition that den contains only positive numbers, no 0. Hinze (2009, p502) gets a bit sloppy here; it is not straightforward to adapt his lifting framework Hinze (2010) to conditional equations.

lemma mod-tree-lemma1:
  fixes x :: nat tree
  assumes ∀i∈set-tree y. 0 < i
  shows x mod (x + y) = x
proof –
  have rel-tree (=) (x mod (x + y)) x by applicative-lifting(simp add: assms)
  thus ?thesis by(unfold tree.rel-eq)
qed

lemma mod-tree-lemma2:
  fixes x y :: 'a :: unique-euclidean-semiring tree
  shows (x + y) mod y = x mod y
by applicative-lifting simp

lemma set-tree-pathD: x ∈ set-tree t =⇒ ∃p. x = root (traverse-tree p t)
by(induct rule: set-tree-induct)(auto intro: exI[where x=[]] exI[where x=L # p for p]
for p] exI[where x=R # p for p])

lemma den-gt-0: 0 < x if x ∈ set-tree den
proof –
  from that obtain p where x = root (traverse-tree p den) by(blast dest: set-tree-pathD)
with stern-brocot-denominator-non-zero[of p] show 0 < x by(simp add: den-conv split-beta)
qed

lemma num-mod-den: num mod den = num-mod-den
by(rule num-mod-den.unique)(rule tree.expand, simp add: mod-tree-lemma2 mod-tree-lemma1 den-gt-0)

lemma tree-chop-den: tree-chop den = num + den - 2 * (num mod den)
proof -
  have le: 0 < y \implies 2 * (x mod y) \leq x + y for x y :: nat
    by (simp add: mult-2 add-mono)

We switch to int such that all cancellation laws are available.

define den' where den' = pure int \diamond den
define num' where num' = pure int \diamond num
define num-mod-den' where num-mod-den' = pure int \diamond num-mod-den

have [simp]: root num' = 1 left num' = num' unfolding den'-def num'-def by simp-all
  have [simp]: right num' = num' + den' unfolding den'-def num'-def ap-tree.sel
    pure-tree-simps num-simps
    by applicative-lifting simp

have num-mod-den'-simp [simp]: root num-mod-den' = 0 left num-mod-den' = num' right num-mod-den' = num-mod-den'
  by(simp-all add: num-mod-den'-def num'-def)
have den'-eq-chop-num': den' = tree-chop num' by(simp add: den'-def num'-def den-eq-chop-num)
  have num'-eq-chop-num': num' = num' - tree-chop den' = 2 * num-mod-den'
    by(corec-unique)(rule tree.expand; simp)
have num'+plus-den'-minus-chop-den': num' + den' - tree-chop den' = 2 * num-mod-den'
  by(rule num-mod-den'-2-unique)(rule tree.expand, simp add: tree-chop-plus den'-eq-chop-num')

have tree-chop den = pure nat \diamond (tree-chop den')
  unfolding den-conv tree-chop-ap-tree tree-chop-pure-tree den'-def by applicative-nf simp
also have tree-chop den' = num' + den' - tree-chop den' + tree-chop den' - 2 * num-mod-den'
  by(subst num'-plus-den'-minus-chop-den') simp
also have . . . = num' + den' - 2 * (num' mod den')
  unfolding num-mod-den'-def num'-def den'-def num-mod-den[symmetric]
  by applicative-lifting(simp add: zmod-int)
also have [unfolded tree.rel-eq]: rel-tree (=) . . . (pure int \diamond (num + den - 2 * (num mod den)))
  unfolding num'-def den'-def by(applicative-lifting)(simp add: of-nat-diff zmod-int le den-gt-0)
also have pure nat ⊙ (pure int ⊙ (num + den - 2 * (num mod den))) = num + den - 2 * (num mod den) by (applicative-nf) simp

finally show thesis.

qed

3.3 Loopless linearisation of the Stern-Brocot tree.

This is a loopless linearisation of the Stern-Brocot tree that gives Stern’s diatomic sequence, which is also known as Dijkstra’s fusc function Dijkstra (1982a,b). Loopless à la Bird (2006) means that the first element of the stream can be computed in linear time and every further element in constant time.

friend-of-corec smap :: ('a ⇒ 'a) ⇒ 'a stream
where smap f xs = SCons (f (shd xs)) (smap f (stl xs))
subgoal by (rule stream.expand) simp
subgoal by (fold relator-eq) (transfer-prover)
done

definition step :: nat × nat ⇒ nat × nat
where step = (λ(n, d). (d, n + d - 2 * (n mod d)))

corec stern-brocot-loopless :: fraction stream
where stern-brocot-loopless = (1, 1) ## smap step stern-brocot-loopless

lemmas stern-brocot-loopless-rec = stern-brocot-loopless.code

friend-of-corec plus where s + s' = (shd s + shd s') ## (stl s + stl s')
subgoal by (rule stream.expand; simp add: plus-stream-shd plus-stream-stl)
subgoal by (transfer-prover)
done

friend-of-corec minus where t − t' = (shd t − shd t') ## (stl t − stl t')
subgoal by (rule stream.expand; simp add: minus-stream-def)
subgoal by (transfer-prover)
done

friend-of-corec times where t * t' = (shd t * shd t') ## (stl t * stl t')
subgoal by (rule stream.expand; simp add: times-stream-def)
subgoal by (transfer-prover)
done

friend-of-corec modulo where t mod t' = (shd t mod shd t') ## (stl t mod stl t')
subgoal by (rule stream.expand; simp add: modulo-stream-def)
subgoal by (transfer-prover)
done

corec fusc' :: nat stream
where fusc' = 1 ## (((1 ## fusc') + fusc') − 2 * ((1 ## fusc') mod fusc'))
definition fusc where fusc = 1 ## fusc'

lemma fusc-unfold: fusc = 1 ## fusc' by (fact fusc-def)

lemma fusc'-unfold: fusc' = 1 ## (fusc + fusc' - 2 * (fusc mod fusc'))
by (subst fusc'.code) (simp add: fusc-def)

lemma fusc-simps [simp]:
    shd fusc = 1
    stl fusc = fusc'
by (simp-all add: fusc-unfold)

lemma fusc'-simps [simp]:
    shd fusc' = 1
    stl fusc' = fusc + fusc' - 2 * (fusc mod fusc')
by (subst fusc'-unfold, simp+)

3.4 Equivalence with Dijkstra’s fusc function

lemma stern-brocot-loopless-siterate:
    stern-brocot-loopless = siterate step (1, 1)
by (rule stern-brocot-loopless.unique[symmetric]) (rule stream.expand; simp add: smap-siterate[symmetric])

lemma fusc-fusc'-iterate:
    pure Pair ⋄ fusc ⋄ fusc' = stern-brocot-loopless
apply (rule stern-brocot-loopless.unique)
apply (rule stream.expand; simp add: step-def)
apply (applicative-lifting; simp)
done

theorem stern-brocot-loopless:
    stream stern-brocot-recurse = stern-brocot-loopless (is ?lhs = ?rhs)
proof (rule stern-brocot-loopless.unique)
have eq: ?lhs = stream (pure-tree Pair ⋄ num ⋄ den) by (simp only: stern-brocot-num-den)
    have num: stream num = 1 ## stream den
      by (rule stream.expand) (simp add: den-eq-chop-num)
    have den: stream den = 1 ## (stream num + stream den - 2 * (stream num mod stream den))
      by (rule stream.expand) (simp add: tree-chop-den)
    show ?lhs = (1, 1) ## smap step ?lhs unfolding eq
qed

end
4 The Bird tree

We define the Bird tree following Hinze (2009) and prove that it is a permutation of the Stern-Brocot tree. As a corollary, we derive that the Bird tree also contains all rational numbers in lowest terms exactly once.

theory Bird-Tree imports Stern-Brocot-Tree begin

corec bird :: fraction tree
where
  bird = Node (1, 1) (map-tree recip (map-tree succ bird)) (map-tree succ (map-tree recip bird))

lemma bird-unfold:
  bird = Node (1, 1) (pure recip ∘ (pure succ ∘ bird)) (pure succ ∘ (pure recip ∘ bird))
using bird.code unfolding map-tree-ap-tree-pure-tree[symmetric].

lemma bird-simps [simp]:
  root bird = (1, 1)
  left bird = pure recip ∘ (pure succ ∘ bird)
  right bird = pure succ ∘ (pure recip ∘ bird)
by (subst bird-unfold, simp+)

lemma mirror-bird: mirror bird = pure recip ∘ bird (is ?lhs = ?rhs)
proof (rule sym)
  let ?F = λt. Node (1, 1) (map-tree succ (map-tree recip t)) (map-tree recip (map-tree succ t))
  have ∗: mirror bird = ?F (mirror bird)
  by (rule tree.expand; simp add: mirror-ap-tree mirror-pure map-tree-ap-tree-pure-tree[symmetric])
show t = mirror bird when t = ?F t for t using that by corec-unique (fact ∗)
show pure recip ∘ bird = ?F (pure recip ∘ bird)
  by (rule tree.expand; simp add: map-tree-ap-tree-pure-tree; applicative-lifting; simp add: split-beta)
qed

primcorec even-odd-mirror :: bool ⇒ 'a tree ⇒ 'a tree
where
  | even. root (even-odd-mirror even t) = root t
  | even. left (even-odd-mirror even t) = even-odd-mirror (∼ even) (if even then right t else left t)
  | even. right (even-odd-mirror even t) = even-odd-mirror (∼ even) (if even then left t else right t)

definition even-mirror :: 'a tree ⇒ 'a tree
where even-mirror = even-odd-mirror True

definition odd-mirror :: 'a tree ⇒ 'a tree
where odd-mirror = even-odd-mirror False

29
lemma even-mirror-simps [simp]:
  root (even-mirror t) = root t
  left (even-mirror t) = odd-mirror (right t)
  right (even-mirror t) = odd-mirror (left t)
and odd-mirror-simps [simp]:
  root (odd-mirror t) = root t
  left (odd-mirror t) = even-mirror (left t)
  right (odd-mirror t) = even-mirror (right t)
by (simp-all add: even-mirror-def odd-mirror-def)

lemma even-odd-mirror-pure [simp]:
  fixes even
  shows even-odd-mirror even (pure-tree x) = pure-tree x
by (coinduction arbitrary: even) auto

lemma even-odd-mirror-ap-tree [simp]:
  fixes even
  shows even-odd-mirror even (f ⋄ x) = even-odd-mirror even f ⋄ even-odd-mirror even x
by (coinduction arbitrary: even f x) auto

lemma [simp]:
  shows even-mirror-pure: even-mirror (pure-tree x) = pure-tree x
  and odd-mirror-pure: odd-mirror (pure-tree x) = pure-tree x
by (simp-all add: even-mirror-def odd-mirror-def)

lemma [simp]:
  shows even-mirror-ap-tree: even-mirror (f ⋄ x) = even-mirror f ⋄ even-mirror x
  and odd-mirror-ap-tree: odd-mirror (f ⋄ x) = odd-mirror f ⋄ odd-mirror x
by (simp-all add: even-mirror-def odd-mirror-def)

fun even-mirror-path :: path ⇒ path
  and odd-mirror-path :: path ⇒ path
where
  even-mirror-path [] = []
  | even-mirror-path (d # ds) = (case d of L ⇒ R | R ⇒ L) # odd-mirror-path ds
  | odd-mirror-path (d # ds) = d # even-mirror-path ds

lemma even-mirror-traverse-tree [simp]:
  root (traverse-tree path (even-mirror t)) = root (traverse-tree (even-mirror-path path) t)
and odd-mirror-traverse-tree [simp]:
  root (traverse-tree path (odd-mirror t)) = root (traverse-tree (odd-mirror-path path) t)
by (induct path arbitrary: t) (simp-all split: dir.splits)

lemma even-odd-mirror-path-involution [simp]:
  even-mirror-path (even-mirror-path path) = path
  odd-mirror-path (odd-mirror-path path) = path
by (induct path) (simp-all split: dir.splits)
lemma even-odd-mirror-path-injective [simp]:
even-mirror-path path = even-mirror-path path' \iff path = path'  
odd-mirror-path path = odd-mirror-path path' \iff path = path'  
by (induct path arbitrary; path') (case_tac path', simp-all split: dir.splits)+

lemma odd-mirror-bird-stern-brocot:  
odd-mirror bird = stern-brocot-recurse  
proof –  
let ?rsrs = map-tree (recip \circ succ \circ recip \circ succ)  
let ?rssr = map-tree (recip \circ succ \circ recip \circ succ)  
let ?srrs = map-tree (succ \circ recip \circ recip \circ succ)  
let ?srsr = map-tree (succ \circ recip \circ succ \circ recip)  
let ?R = \lambda t. Node (1, 1) (Node (1, 2) (?rssr t) (?rsrs t)) (Node (2, 1) (?srss t) (?rsrs t))  

have \ast: stern-brocot-recurse = ?R stern-brocot-recurse  
by (rule tree.expand; simp; intro conjI; rule tree.expand; simp; intro conjI) —  
Expand the tree twice  
(applicative-lifting, simp add: split-beta)+  
show f = stern-brocot-recurse when f = ?R f for f using \ast by corec-unique  
show odd-mirror bird = ?R (odd-mirror bird)  
by (rule tree.expand; simp; intro conjI; rule tree.expand; simp; intro conjI) —  
Expand the tree twice  
(applicative-lifting; simp)+  
qed  

theorem bird-rationals:  
assumes m > 0 n > 0  
shows root (traverse-tree (odd-mirror-path (mk-path m n))) (pure rat-of \circ bird)) = Fract (int m) (int n)  
using stern-brocot-rationals[OF assms]  
by (simp add: odd-mirror-bird-stern-brocot[symmetric])

theorem bird-rationals-not-repeated:  
root (traverse-tree path (pure rat-of \circ bird)) = root (traverse-tree path' (pure rat-of \circ bird))  
\implies path = path'  
using stern-brocot-rationals-not-repeated[where path=odd-mirror-path path and path'=odd-mirror-path path']  
by (simp add: odd-mirror-bird-stern-brocot[symmetric])

end

References
Roland Backhouse and João F. Ferreira. Recounting the rationals: Twice!  
In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathemat-


