The Stern-Brocot Tree

Peter Gammie  Andreas Lochbihler

August 16, 2018

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 5
    1.2.1 Recurse combinator 5
    1.2.2 Tree iteration 6
    1.2.3 Tree traversal 6
    1.2.4 Mirroring 7
  1.3 Pointwise arithmetic on infinite binary trees 8
    1.3.1 Constants and operators 8
    1.3.2 Algebraic instances 9

2 The Stern-Brocot Tree 11
  2.1 Specification via a recursion equation 12
  2.2 Basic properties 13
  2.3 All the rationals 13
  2.4 No repetitions 14
  2.5 Equivalence of recursive and iterative version 17
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)

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)
⟨proof⟩

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
⟨proof⟩

lemma corec-tree-cong:
  assumes ∃x. stopL x ⇒ STOPL x = STOPL' x
  and ∃x. ¬ stopL x ⇒ LEFT x = LEFT' x
  and ∃x. stopR x ⇒ STOPR x = STOPR' x
  and ∃x. ¬ stopR x ⇒ RIGHT x = RIGHT' x
shows corec-tree ROOT stopL STOPL LEFT stopR STOPR RIGHT = 
corec-tree ROOT stopL STOPL’ LEFT’ stopR STOPR’ RIGHT’
(is ?lhs = ?rhs)
⟨proof⟩

context
  fixes g1 :: 'a ⇒ 'b
  and g22 :: 'a ⇒ 'a
  and g32 :: 'a ⇒ 'a
begin

corec unfold-tree :: 'a ⇒ '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)
⟨proof⟩

end

lemma unfold-tree-unique:
  assumes ⊓ s. root (f s) = ROOT s
  and ⊓ s. left (f s) = f (LEFT s)
  and ⊓ s. right (f s) = f (RIGHT s)
  shows f s = unfold-tree ROOT LEFT RIGHT s
⟨proof⟩

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
⟨proof⟩

adhoc-overloading pure pure-tree

lemma pure-tree-parametric [transfer-rule]: (rel-fun A (rel-tree A)) pure pure
⟨proof⟩
lemma map-pure-tree [simp]: \( \text{map-tree } f (\text{pure } x) = \text{pure } (f x) \)
(proof)

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

primcorec (transfer) ap-tree :: ('a ⇒ 'b) tree ⇒ 'a tree ⇒ 'b tree
where
  \begin{align*}
  \text{root } (\text{ap-tree } f x) &= \text{root } f \text{ (root } x) \\
  \text{left } (\text{ap-tree } f x) &= \text{ap-tree } (\text{left } f) \text{ (left } x) \\
  \text{right } (\text{ap-tree } f x) &= \text{ap-tree } (\text{right } f) \text{ (right } x)
  \end{align*}

adhoc-overloading Applicative.ap ap-tree

unbundle applicative-syntax

lemma ap-tree-pure-Node [simp]:
  \( \text{pure } f \circ \text{Node } x l r = \text{Node } (f x) \text{ (pure } f \circ l) \text{ (pure } f \circ r) \)
(proof)

lemma ap-tree-Node-Node [simp]:
  \( \text{Node } f fl fr \circ \text{Node } x l r = \text{Node } (f x) \text{ (fl } \circ l) \text{ (fr } \circ r) \)
(proof)

Applicative functor laws

lemma map-tree-ap-tree-pure-tree:
  \( \text{pure } f \circ u = \text{map-tree } f u \)
(proof)

lemma ap-tree-identity: \( \text{pure } \text{id } \circ t = t \)
(proof)

lemma ap-tree-composition:
  \( \text{pure } (\circ) \circ r1 \circ r2 \circ r3 = r1 \circ (r2 \circ r3) \)
(proof)

lemma ap-tree-homomorphism:
  \( \text{pure } f \circ \text{pure } x = \text{pure } (f x) \)
(proof)

lemma ap-tree-interchange:
  \( t \circ \text{pure } x = \text{pure } (\lambda f. f x) \circ t \)
(proof)

lemma ap-tree-K-tree: \( \text{pure } (\lambda x y. x) \circ u \circ v = u \)
(proof)

lemma ap-tree-C-tree: \( \text{pure } (\lambda f x y. f y x) \circ u \circ v \circ w = u \circ w \circ v \)
(proof)
lemma ap-tree-W-tree: pure \((\lambda f. f x x) \circ f \circ x = f \circ x \circ x\)
(proof)

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

declare map-tree-ap-tree-pure-tree

lemma ap-tree-strong-extensional:
\((\forall x. f \circ pure x = g \circ pure x) \Rightarrow f = g\)
(proof)

lemma ap-tree-extensional:
\((\forall x. f \circ x = g \circ x) \Rightarrow f = g\)
(proof)

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 \circ l) (\lambda f \circ r)
shows map-tree G (unf F) = unf (G \circ F)
(proof)

friend-of-corec map-tree :: \(\forall a \Rightarrow a \Rightarrow a\) tree \Rightarrow a tree
where
map-tree f t = Node (f (root t)) (map-tree f (left t)) (map-tree f (right t))
(proof)

context fixes l :: \(\forall a \Rightarrow a\) and r :: \(\forall a \Rightarrow a\) and x :: \(\forall a\) begin
corec tree-recurse :: \(\forall a\) 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)
(proof)

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))
(proof)

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)
(proof)

1.2.2 Tree iteration
context fixes l :: 'a ⇒ 'a and r :: 'a ⇒ 'a begin
primcorec tree-iterate :: 'a ⇒ '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 \circ tree-iterate l r
(proof)

lemma tree-iterate-fusion:
  assumes h \circ l = l' \circ h
  assumes h \circ r = r' \circ h
  shows map-tree h (tree-iterate l r x) = tree-iterate l' r' (h x)
(proof)

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 \circ case-dir left right d) path id

lemma traverse-tree-simps[simp]:
  traverse-tree [] = id
  traverse-tree (d @ path) = traverse-tree path \circ (case d of L ⇒ left | R ⇒ right)
(proof)

lemma traverse-tree-map-tree [simp]:
  traverse-tree path (map-tree f t) = map-tree f (traverse-tree path t)
(proof)

lemma traverse-tree-append [simp]:
  traverse-tree (path @ ext) t = traverse-tree ext (traverse-tree path t)
(proof)

traverse-tree is an applicative-functor homomorphism.

lemma traverse-tree-pure-tree [simp]:
  traverse-tree path (pure x) = pure x
lemma traverse-tree-ap [simp]:
  traverse-tree path (f ⋄ x) = traverse-tree path f ⋄ traverse-tree path x
⟨proof⟩
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)
⟨proof⟩

? shows that if the tree construction function is suitably monoidal then recursion and iteration define the same tree.
lemma tree-recurse-iterate:
  assumes monoid:
    \( \forall x y z. f (f x y) z = f x (f y z) \)
    \( \forall x. f x x = x \)
    \( \forall x. f x \varepsilon = x \)
  shows tree-recurse (f l) (f r) ε = tree-iterate (λx. f x l) (λx. f x r) ε
⟨proof⟩

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)
⟨proof⟩
lemma mirror-pure: mirror (pure x) = pure x
⟨proof⟩
lemma mirror-ap-tree: mirror (f ⋄ x) = mirror f ⋄ mirror x
⟨proof⟩

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 ⟨proof⟩ end

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

instantiation tree :: (plus) plus begin
definition [applicative-unfold]: plus x y = pure (+) ∘ x ∘ (y :: 'a tree)
instance ⟨proof⟩ 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'
⟨proof⟩

friend-of-corec plus where t + t' = Node (root t + root t') (left t + left t')
⟨proof⟩

instantiation tree :: (minus) minus begin
definition [applicative-unfold]: minus x y = pure (−) ∘ x ∘ (y :: 'a tree)
instance ⟨proof⟩ 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'
⟨proof⟩

instantiation tree :: (uminus) uminus begin
definition [applicative-unfold tree]: uminus = ((∘) (pure uminus) :: 'a tree ⇒ 'a tree)

end
instance ⟨proof⟩
end

instantiation tree :: (times) times begin
definition [applicative-unfold]: times x y = pure ( *) ⊙ x ⊙ (y :: 'a tree)
instance ⟨proof⟩
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'
⟨proof⟩

instance tree :: (Rings.dvd) Rings.dvd ⟨proof⟩

instantiation tree :: (modulo) modulo begin
definition [applicative-unfold]: x div y = pure-tree (div) ⊙ x ⊙ (y :: 'a tree)
definition [applicative-unfold]: x mod y = pure-tree (mod) ⊙ x ⊙ (y :: 'a tree)
instance ⟨proof⟩
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'
⟨proof⟩

1.3.2 Algebraic instances

instance tree :: (semigroup-add) semigroup-add ⟨proof⟩

instance tree :: (ab-semigroup-add) ab-semigroup-add ⟨proof⟩

instance tree :: (semigroup-mult) semigroup-mult ⟨proof⟩

instance tree :: (ab-semigroup-mult) ab-semigroup-mult ⟨proof⟩

instance tree :: (monoid-add) monoid-add ⟨proof⟩

instance tree :: (comm-monoid-add) comm-monoid-add ⟨proof⟩

instance tree :: (comm-monoid-diff) comm-monoid-diff
instance tree :: (monoid-mult) monoid-mult
(proof)

instance tree :: (comm-monoid-mult) comm-monoid-mult
(proof)

instance tree :: (cancel-semigroup-add) cancel-semigroup-add
(proof)

instance tree :: (cancel-ab-semigroup-add) cancel-ab-semigroup-add
(proof)

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

instance tree :: (group-add) group-add
(proof)

instance tree :: (ab-group-add) ab-group-add
(proof)

instance tree :: (semiring) semiring
(proof)

instance tree :: (mult-zero) mult-zero
(proof)

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

instance tree :: (semiring-0-cancel) semiring-0-cancel (proof)

instance tree :: (comm-semiring) comm-semiring
(proof)

instance tree :: (comm-semiring-0) comm-semiring-0 (proof)

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

lemma pure-tree-inject[simp]: pure-tree x = pure-tree y \iff x = y
(proof)

instance tree :: (zero-neq-one) zero-neq-one
(proof)

instance tree :: (semiring-1) semiring-1 (proof)

instance tree :: (comm-semiring-1) comm-semiring-1 (proof)
instance tree :: (semiring-1-cancel) semiring-1-cancel ⟨proof⟩

instance tree :: (comm-semiring-1-cancel) comm-semiring-1-cancel ⟨proof⟩

instance tree :: (ring) ring ⟨proof⟩

instance tree :: (comm-ring) comm-ring ⟨proof⟩

instance tree :: (ring-1) ring-1 ⟨proof⟩

instance tree :: (comm-ring-1) comm-ring-1 ⟨proof⟩

instance tree :: (numeral) numeral ⟨proof⟩

instance tree :: (neg-numeral) neg-numeral ⟨proof⟩

instance tree :: (semiring-numeral) semiring-numeral ⟨proof⟩

lemma of-nat-tree: of-nat n = pure-tree (of-nat n) ⟨proof⟩

instance tree :: (semiring-char-0) semiring-char-0 ⟨proof⟩

lemma numeral-tree-simps [simp]:
  root (numeral n) = numeral n
  left (numeral n) = numeral n
  right (numeral n) = numeral n ⟨proof⟩

lemma numeral-tree-conv-pure [applicative-unfold]: numeral n = pure (numeral n) ⟨proof⟩

instance tree :: (ring-char-0) ring-char-0 ⟨proof⟩

end

2 The Stern-Brocot Tree

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

11
Figure 1: Constructing the Stern-Brocot tree iteratively.

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)
⟨proof⟩

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
⟨proof⟩

lemma stern-brocot-conv:
  stern-brocot-recurse = tree-recurse (recip ∘ succ ∘ recip) succ (1, 1)
⟨proof⟩

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
⟨proof⟩

lemma stern-brocot-coprime:
  case root (traverse-tree path stern-brocot-recurse) of (m, n) ⇒ coprime m n
⟨proof⟩

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
\[ m = n \Rightarrow \text{mk-path} (\text{Suc } m) (\text{Suc } n) = [] \]
| \( m < n \Rightarrow \text{mk-path} (\text{Suc } m) (\text{Suc } n) = \text{L} \# \text{mk-path} (\text{Suc } m) (n - m) \)
| \( m > n \Rightarrow \text{mk-path} (\text{Suc } m) (\text{Suc } n) = \text{R} \# \text{mk-path} (m - n) (\text{Suc } n) \)
| \(\text{mk-path} \emptyset \cdot = \text{undefined} \)
| \(\text{mk-path} \cdot \emptyset = \text{undefined} \)

\(\langle\text{proof}\rangle\)

\textbf{termination} \text{mk-path} \(\langle\text{proof}\rangle\)

\textbf{lemmas} \text{mk-path-induct} [\text{case-names equal less greater}] = \text{mk-path.induct}

\textbf{abbreviation} \text{rat-of} :: \text{fraction} \Rightarrow \text{rat}
\text{where} \text{rat-of} \equiv \lambda (x, y). \text{Fract} (\text{int } x) (\text{int } y)

\textbf{theorem} \text{stern-brocot-rationals}:
\[ [m > 0; n > 0] \Rightarrow \text{root} (\text{traverse-tree} (\text{mk-path } m \ n) (\text{pure } \text{rat-of} \circ \text{stern-brocot-recurse})) = \text{Fract} (\text{int } m) (\text{int } n) \]
\(\langle\text{proof}\rangle\)

### 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 \text{stern-brocot-coprime}.

Intuitively, the tree enjoys the \textit{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 \text{Backhouse and Ferreira (2008)} and \text{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).

\textbf{type-synonym} \text{matrix} = \text{fraction} \times \text{fraction}
\textbf{type-synonym} \text{vector} = \text{fraction}

\textbf{definition} \text{times-matrix} :: \text{matrix} \Rightarrow \text{matrix} \Rightarrow \text{matrix} \text{ (infixl } \odot 70)\text{ (\textbf{where})}
\text{where} \text{times-matrix} = (\lambda ((a, c), (b, d)) ((a', c'), (b', d'))).
\[ ((a * a' + b * c', c * a' + d * c'),\]
\[ (a * b' + b * d', c * b' + d * d')) \]

\textbf{definition} \text{times-vector} :: \text{matrix} \Rightarrow \text{vector} \Rightarrow \text{vector} \text{ (infixr } \odot 70)\text{ (\textbf{where})}
\text{where} \text{times-vector} = (\lambda ((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 $Det :: matrix \Rightarrow \text{nat}$ where $Det \equiv \lambda ((a, c), (b, d)). a \cdot d - b \cdot c$

lemma $\text{Dets}$ \[iff\]:
$Det I = 1$
$Det LL = 1$
$Det UR = 1$
⟨proof⟩

lemma $\text{LL-UR-Det}$:
$Det m = 1 \implies Det (m \otimes LL) = 1$
$Det m = 1 \implies Det (LL \otimes m) = 1$
$Det m = 1 \implies Det (m \otimes UR) = 1$
$Det m = 1 \implies Det (UR \otimes m) = 1$
⟨proof⟩

lemma $\text{mediant-I-F}$ \[simp\]:
 mediated $F = (1, 1)$
 mediated $I = (1, 1)$
⟨proof⟩

lemma $\text{times-matrix-I}$ \[simp\]:
$I \otimes x = x$
$x \otimes I = x$
⟨proof⟩

lemma $\text{times-matrix-assoc}$ \[simp\]:
$(x \otimes y) \otimes z = x \otimes (y \otimes z)$
⟨proof⟩

lemma $\text{LL-UR-pos}$:
$0 < \text{snd} (\text{mediant m}) \implies 0 < \text{snd} (\text{mediant (m \otimes LL)})$
$0 < \text{snd} (\text{mediant m}) \implies 0 < \text{snd} (\text{mediant (m \otimes UR)})$
⟨proof⟩

lemma $\text{recip-succ-recip}$: $\text{recip} \circ \text{succ} \circ \text{recip} = (\lambda(x, y). (x, x + y))$
⟨proof⟩

Backhouse and Ferreira work with the identity matrix $I$ at the root. This has the advantage that all relevant matrices have determinants of $I$.

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

15
definition stern-brocot-iterate :: fraction tree
where stern-brocot-iterate ≡ map-tree mediant (stern-brocot-iterate-aux I)

lemma stern-brocot-recurse-iterate: stern-brocot-recurse = stern-brocot-iterate (is lhs = rhs)
⟨proof⟩

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

lemma tree-ordering-left:
  assumes DX: Det X = 1
  assumes DY: Det Y = 1
  assumes MX: 0 < snd (mediant X)
  shows rat-of (mediant (X ⊗ LL ⊗ Y)) < rat-of (mediant X)
⟨proof⟩

lemma tree-ordering-right:
  assumes DX: Det X = 1
  assumes DY: Det Y = 1
  assumes MX: 0 < snd (mediant X)
  shows rat-of (mediant X) < rat-of (mediant (X ⊗ UR ⊗ Y))
⟨proof⟩

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)))))
⟨proof⟩

lemma stern-brocot-iterate-aux-decompose:
  ∃ m". m ⊗ m" = root (traverse-tree path (stern-brocot-iterate-aux m)) ∧ Det m"
  = 1
⟨proof⟩

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⟩

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⟩
lemma lists-not-eq:
  assumes $xs \neq ys$
  obtains
  $(c1)$ strict-prefix $xs ys$
  | $(c2)$ strict-prefix $ys xs$
  | $(c3)$ $ps x y zs' ys'$
  where $xs = ps @ x \# xs'$ and $ys = ps @ y \# ys'$ and $x \neq y$
⟨proof⟩

lemma stern-brocot-fractions-not-repeated:
  assumes root $(\text{traverse-tree path stern-brocot-iterate}) =$
  root $(\text{traverse-tree path'}$
  stern-brocot-iterate)$
  shows path = path'$
⟨proof⟩

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 $a = c \land b = d$
⟨proof⟩

theorem stern-brocot-rationals-not-repeated:
  assumes root $(\text{traverse-tree path (pure rat-of } \circ \text{ stern-brocot-recurse)})$
  = root $(\text{traverse-tree path'} (\text{pure rat-of } \circ \text{ stern-brocot-recurse}))$
  shows path = path'$
⟨proof⟩

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 ($\lambda s. s \otimes UR$)
($\lambda s \otimes LL$) $F$)

lemma mediant-times-F: mediant $\circ (\lambda s \otimes F) = \text{mediant}$
⟨proof⟩

lemma stern-brocot-iterate: stern-brocot = stern-brocot-iterate
⟨proof⟩

theorem stern-brocot-mediant-recurse: stern-brocot = stern-brocot-recurse
⟨proof⟩
3 Linearising the Stern-Brocot Tree

3.1 Turning a tree into a stream

```latex
\begin{verbatim}
corec tree-chop :: 'a tree ⇒ 'a tree
where tree-chop t = Node (root (left t)) (right t) (tree-chop (left t))

lemma tree-chop-set [simp]:
  root (tree-chop t) = root (left t)
  left (tree-chop t) = right t
  right (tree-chop t) = tree-chop (left t)
⟨proof⟩

  tree-chop is a idiom homomorphism

lemma tree-chop-pure-tree [simp]:
  tree-chop (pure x) = pure x
⟨proof⟩

lemma tree-chop-ap-tree [simp]:
  tree-chop (f ⋄ x) = tree-chop f ⋄ tree-chop x
⟨proof⟩

lemma tree-chop-plus:
  tree-chop (t + t') = tree-chop t + tree-chop t'
⟨proof⟩

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

lemma stream-set [simp]:
  shd (stream t) = root t
  stl (stream t) = stream (tree-chop t)
⟨proof⟩

  stream is an idiom homomorphism.

lemma stream-pure [simp]: stream (pure x) = pure x
⟨proof⟩

lemma stream-ap [simp]: stream (f ⋄ x) = stream f ⋄ stream x
⟨proof⟩

lemma stream-plus [simp]: stream (t + t') = stream t + stream t'
⟨proof⟩
\end{verbatim}
```
**Lemma** stream-minus [simp]: \( \text{stream } (t - t') = \text{stream } t - \text{stream } t' \)

**Lemma** stream-times [simp]: \( \text{stream } (t * t') = \text{stream } t * \text{stream } t' \)

**Lemma** stream-mod [simp]: \( \text{stream } (t \mod t') = \text{stream } t \mod \text{stream } t' \)

**Lemma** stream-1 [simp]: \( \text{stream } 1 = 1 \)

**Lemma** stream-numeral [simp]: \( \text{stream } (\text{numeral } n) = \text{numeral } n \)

### 3.2 Split the Stern-Brocot tree into numerators and denominators

**Corec** num-den :: bool ⇒ nat tree

**Where**

\[
\text{num-den } x = \\
\text{Node } 1 \\
\quad \text{(if } x \text{ then num-den True else num-den True + num-den False)} \\
\quad \text{(if } x \text{ then num-den True + num-den False else num-den False)}
\]

**Abstraction** num where num ≡ num-den True

**Abstraction** den where den ≡ num-den False

**Lemma** num-unfold: num = Node 1 num (num + den)

**Lemma** den-unfold: den = Node 1 (num + den) den

**Lemma** num-simps [simp]:

\[
\begin{align*}
\text{root num} &= 1 \\
\text{left num} &= \text{num} \\
\text{right num} &= \text{num} + \text{den}
\end{align*}
\]

**Lemma** den-simps [simp]:

\[
\begin{align*}
\text{root den} &= 1 \\
\text{left den} &= \text{num} + \text{den} \\
\text{right den} &= \text{den}
\end{align*}
\]

**Lemma** stern-brocot-num-den:

\[
\text{pure-tree Pair } \odot \text{num } \odot \text{den } = \text{stern-brocot-recurse}
\]
lemma den-eq-chop-num: den = tree-chop num
⟨proof⟩

lemma num-conv: num = pure fst ⨿ stern-brocot-recurse
⟨proof⟩

lemma den-conv: den = pure snd ⨿ stern-brocot-recurse
⟨proof⟩

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
⟨proof⟩

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⟩

lemma mod-tree-lemma2:
fixes x y :: 'a :: unique-euclidean-semiring tree
shows (x + y) mod y = x mod y
⟨proof⟩

lemma set-tree-pathD: x ∈ set-tree t → ∃ p. x = root (traverse-tree p t)
⟨proof⟩

lemma den-gt-0: 0 < x if x ∈ set-tree den
⟨proof⟩

lemma num-mod-den: num mod den = num-mod-den
⟨proof⟩

lemma tree-chop-den: tree-chop den = num + den - 2 * (num mod den)
⟨proof⟩
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 ⇒ 'a stream
where smap f xs = SCons (f (shd xs)) (smap f (stl xs))
⟨proof⟩

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')
⟨proof⟩

friend-of-corec minus where t − t' = (shd t − shd t') ## (stl t − stl t')
⟨proof⟩

friend-of-corec times where t ∗ t' = (shd t ∗ shd t') ## (stl t ∗ stl t')
⟨proof⟩

friend-of-corec modulo where t mod t' = (shd t mod shd t') ## (stl t mod stl t')
⟨proof⟩

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' (proof)

lemma fusc'-unfold: fusc' = 1 ## (fusc + fusc' − 2 ∗ (fusc mod fusc'))
⟨proof⟩

lemma fusc-simps [simp]:
  shd fusc = 1
  stl fusc = fusc'
⟨proof⟩

lemma fusc'-simps [simp]:
\[
\begin{align*}
\text{shd } & \text{fusc} = 1 \\
\text{stl } & \text{fusc} = \text{fusc} + \text{fusc}' - 2 \times (\text{fusc mod fusc}')
\end{align*}
\]

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

### 3.4 Equivalence with Dijkstra’s fusc function

**Lemma** stern-brocot-loopless-siterate: \(\text{stern-brocot-loopless} = \text{siterate step} \ (1, 1)\)

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

**Lemma** fusc-fusc’-iterate: \(\text{pure Pair } \circ \text{fusc } \circ \text{fusc}' = \text{stern-brocot-loopless}\)

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

**Theorem** stern-brocot-loopless:

\(\text{stream stern-brocot-recurse} = \text{stern-brocot-loopless} \ (\text{is lhs} = ?\text{rhs})\)

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

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

\[\text{where} \]

\(\text{bird} = \text{Node} \ (1, 1) \ (\text{map-tree recip} \ (\text{map-tree succ} \ \text{bird})) \ (\text{map-tree succ} \ (\text{map-tree recip} \ \text{bird}))\)

**Lemma** bird-unfold:

\(\text{bird} = \text{Node} \ (1, 1) \ (\text{pure recip} \circ (\text{pure succ} \circ \text{bird})) \ (\text{pure succ} \circ (\text{pure recip} \circ \text{bird}))\)

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

**Lemma** bird-simps [simp]:

\(\text{root bird} = (1, 1)\)

\(\text{left bird} = \text{pure recip} \circ (\text{pure succ} \circ \text{bird})\)

\(\text{right bird} = \text{pure succ} \circ (\text{pure recip} \circ \text{bird})\)

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

**Lemma** mirror-bird: \(\text{mirror bird} = \text{pure recip} \circ \text{bird} \ (\text{is lhs} = ?\text{rhs})\)

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

**Primcorec** even-odd-mirror :: bool \(\Rightarrow \ ‘a \ tree \Rightarrow ‘a \ tree\)

\[\text{where} \]

\(\forall \text{even. root (even-odd-mirror even t)} = \text{root t}\)

end
\[ \forall \text{even. left (even-odd-mirror even t)} = \text{even-odd-mirror (\sim \text{even}) (if even then right t else left t)} \]
\[ \forall \text{even. right (even-odd-mirror even t)} = \text{even-odd-mirror (\sim \text{even}) (if even then left t else right t)} \]

definition even-mirror :: 'a tree \Rightarrow 'a tree
where
\text{even-mirror = even-odd-mirror True}

definition odd-mirror :: 'a tree \Rightarrow 'a tree
where
\text{odd-mirror = even-odd-mirror False}

lemma even-mirror-simps [simp]:
\begin{align*}
\text{root (even-mirror t)} &= \text{root t} \\
\text{left (even-mirror t)} &= \text{odd-mirror (right t)} \\
\text{right (even-mirror t)} &= \text{odd-mirror (left t)} \\
\end{align*}

and odd-mirror-simps [simp]:
\begin{align*}
\text{root (odd-mirror t)} &= \text{root t} \\
\text{left (odd-mirror t)} &= \text{even-mirror (left t)} \\
\text{right (odd-mirror t)} &= \text{even-mirror (right t)} \\
\end{align*}

lemma even-odd-mirror-pure [simp]: \text{fixes even} shows
\text{even-odd-mirror even (pure-tree x) = pure-tree x}

lemma even-odd-mirror-ap-tree [simp]: \text{fixes even} shows
\text{even-odd-mirror even (f \odot x) = even-odd-mirror even f \odot even-odd-mirror even x}

lemma [simp]:
\begin{align*}
\text{shows even-mirror-pure: even-mirror (pure-tree x) = pure-tree x} \\
\text{and odd-mirror-pure: odd-mirror (pure-tree x) = pure-tree x} \\
\end{align*}

lemma [simp]:
\begin{align*}
\text{shows even-mirror-ap-tree: even-mirror (f \odot x) = even-mirror f \odot even-mirror x} \\
\text{and odd-mirror-ap-tree: odd-mirror (f \odot x) = odd-mirror f \odot odd-mirror x} \\
\end{align*}

fun even-mirror-path :: path \Rightarrow path

and odd-mirror-path :: path \Rightarrow path

where
\begin{align*}
\text{even-mirror-path []} &= [] \\
\text{even-mirror-path (d # ds)} &= \text{(case d of \text{L} \Rightarrow \text{R} | \text{R} \Rightarrow \text{L}) \# odd-mirror-path ds} \\
\text{odd-mirror-path []} &= [] \\
\text{odd-mirror-path (d # ds)} &= d \# \text{even-mirror-path ds} \\
\end{align*}
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)
⟨proof⟩

lemma even-odd-mirror-path-involution [simp]:
  even-mirror-path (even-mirror-path path) = path
  odd-mirror-path (odd-mirror-path path) = path
⟨proof⟩

lemma even-odd-mirror-path-injective [simp]:
  even-mirror-path path = even-mirror-path path′ ↔ path = path′
  odd-mirror-path path = odd-mirror-path path′ ↔ path = path′
⟨proof⟩

lemma odd-mirror-bird-stern-brocot:
  odd-mirror bird = stern-brocot-recurse
⟨proof⟩

theorem bird-rationals:
  assumes m > 0 n > 0
  shows root (traverse-tree (odd-mirror-path (mk-path m n)) (pure rat-of ⋄ bird)) = Fract (int m) (int n)
⟨proof⟩

theorem bird-rationals-not-repeated:
  root (traverse-tree path (pure rat-of ⋄ bird)) = root (traverse-tree path′ (pure rat-of ⋄ bird))
  ⇒ path = path′
⟨proof⟩

end

References


Edsger W. Dijkstra. An exercise for Dr. R. M. Burstall. In *Selected


