

The Stern-Brocot Tree

Peter Gammie Andreas Lochbihler

October 11, 2017

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 <i>'a tree</i>	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

3	Linearising the Stern-Brocot Tree	18
3.1	Turning a tree into a stream	18
3.2	Split the Stern-Brocot tree into numerators and denominators	19
3.3	Loopless linearisation of the Stern-Brocot tree.	20
3.4	Equivalence with Dijkstra's fusc function	22
4	The Bird tree	22

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)
  ⟨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:  $\bigwedge t. P$  (root t) t
  and left:  $\bigwedge x t. \llbracket x \in \text{set-tree (left t); } P x \text{ (left t)} \rrbracket \implies P x t$ 
  and right:  $\bigwedge x t. \llbracket x \in \text{set-tree (right t); } P x \text{ (right t)} \rrbracket \implies P x t$ 
  shows P x t
  ⟨proof⟩

lemma corec-tree-cong:
  assumes  $\bigwedge x. \text{stopL } x \implies \text{STOPL } x = \text{STOPL}' x$ 
  and  $\bigwedge x. \sim \text{stopL } x \implies \text{LEFT } x = \text{LEFT}' x$ 
  and  $\bigwedge x. \text{stopR } x \implies \text{STOPR } x = \text{STOPR}' x$ 
  and  $\bigwedge x. \neg \text{stopR } x \implies \text{RIGHT } x = \text{RIGHT}' x$ 

```

```

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  $\bigwedge s. \text{root } (f\ s) = \text{ROOT } s$ 
  and  $\bigwedge s. \text{left } (f\ s) = f\ (\text{LEFT } s)$ 
  and  $\bigwedge s. \text{right } (f\ s) = f\ (\text{RIGHT } s)$ 
  shows  $f\ s = \text{unfold-tree } \text{ROOT } \text{LEFT } \text{RIGHT } s$ 
<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 \Rightarrow 'b) tree \Rightarrow 'a tree \Rightarrow 'b tree
where

$\text{root } (\text{ap-tree } f \ x) = \text{root } f \ (\text{root } x)$
 | $\text{left } (\text{ap-tree } f \ x) = \text{ap-tree } (\text{left } f) \ (\text{left } x)$
 | $\text{right } (\text{ap-tree } f \ x) = \text{ap-tree } (\text{right } f) \ (\text{right } x)$

adhoc-overloading *Applicative.ap ap-tree*

unbundle *applicative-syntax*

lemma *ap-tree-pure-Node* [*simp*]:
 $\text{pure } f \ \diamond \ \text{Node } x \ l \ r = \text{Node } (f \ x) \ (\text{pure } f \ \diamond \ l) \ (\text{pure } f \ \diamond \ r)$
<proof>

lemma *ap-tree-Node-Node* [*simp*]:
 $\text{Node } f \ fl \ fr \ \diamond \ \text{Node } x \ l \ r = \text{Node } (f \ x) \ (fl \ \diamond \ l) \ (fr \ \diamond \ r)$
<proof>

Applicative functor laws

lemma *map-tree-ap-tree-pure-tree*:
 $\text{pure } f \ \diamond \ u = \text{map-tree } f \ u$
<proof>

lemma *ap-tree-identity*: $\text{pure } id \ \diamond \ t = t$
<proof>

lemma *ap-tree-composition*:
 $\text{pure } (op \ \circ) \ \diamond \ r1 \ \diamond \ r2 \ \diamond \ r3 = r1 \ \diamond \ (r2 \ \diamond \ r3)$
<proof>

lemma *ap-tree-homomorphism*:
 $\text{pure } f \ \diamond \ \text{pure } x = \text{pure } (f \ x)$
<proof>

lemma *ap-tree-interchange*:
 $t \ \diamond \ \text{pure } x = \text{pure } (\lambda f. f \ x) \ \diamond \ t$
<proof>

lemma *ap-tree-K-tree*: $\text{pure } (\lambda x \ y. x) \ \diamond \ u \ \diamond \ v = u$
<proof>

lemma *ap-tree-C-tree*: $\text{pure } (\lambda f \ x \ y. f \ y \ x) \ \diamond \ u \ \diamond \ v \ \diamond \ w = u \ \diamond \ w \ \diamond \ v$
<proof>

lemma *ap-tree-W-tree*: $\text{pure } (\lambda f x. f x x) \diamond f \diamond x = f \diamond x \diamond x$
 ⟨*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*[*symmetric, applicative-unfold*]

lemma *ap-tree-strong-extensional*:

$(\bigwedge x. f \diamond \text{pure } x = g \diamond \text{pure } x) \implies f = g$
 ⟨*proof*⟩

lemma *ap-tree-extensional*:

$(\bigwedge x. f \diamond x = g \diamond x) \implies f = g$
 ⟨*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 $\text{unf} \equiv \text{unfold-tree } (\lambda f. f x) (\lambda f. f \circ l) (\lambda f. f \circ r)$

shows $\text{map-tree } G (\text{unf } F) = \text{unf } (G \circ F)$

⟨*proof*⟩

friend-of-corec *map-tree* :: $'a \Rightarrow 'a) \Rightarrow 'a \text{ tree} \Rightarrow 'a \text{ tree}$ **where**

$\text{map-tree } f t = \text{Node } (f (\text{root } t)) (\text{map-tree } f (\text{left } t)) (\text{map-tree } f (\text{right } t))$

⟨*proof*⟩

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

corec *tree-recurse* :: $'a \text{ tree}$

where $\text{tree-recurse} = \text{Node } x (\text{map-tree } l \text{ tree-recurse}) (\text{map-tree } r \text{ tree-recurse})$

end

lemma *tree-recurse-simps* [*simp*]:

$\text{root } (\text{tree-recurse } l r x) = x$

$\text{left } (\text{tree-recurse } l r x) = \text{map-tree } l (\text{tree-recurse } l r x)$

$\text{right } (\text{tree-recurse } l r x) = \text{map-tree } r (\text{tree-recurse } l r x)$

⟨*proof*⟩

lemma *tree-recurse-unfold*:

$tree-recurse\ l\ r\ x = Node\ x\ (map-tree\ l\ (tree-recurse\ l\ r\ x))\ (map-tree\ r\ (tree-recurse\ l\ r\ x))$
 $\langle proof \rangle$

lemma *tree-recurse-fusion*:
assumes $h \circ l = l' \circ h$ **and** $h \circ r = r' \circ h$
shows $map-tree\ h\ (tree-recurse\ l\ r\ x) = tree-recurse\ l'\ r'\ (h\ x)$
 $\langle proof \rangle$

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*:
 $unfold-tree\ out\ l\ r = map-tree\ out \circ tree-iterate\ l\ r$
 $\langle proof \rangle$

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

1.2.3 Tree traversal

datatype *dir* = $L \mid R$
type-synonym *path* = $dir\ list$

definition *traverse-tree* $:: path \Rightarrow 'a\ tree \Rightarrow 'a\ tree$
where $traverse-tree\ path \equiv foldr\ (\lambda d\ f.\ f \circ case-dir\ left\ right\ d)\ path\ id$

lemma *traverse-tree-simps* [*simp*]:
 $traverse-tree\ [] = id$
 $traverse-tree\ (d \# path) = traverse-tree\ path \circ (case\ d\ of\ L \Rightarrow left \mid R \Rightarrow right)$
 $\langle proof \rangle$

lemma *traverse-tree-map-tree* [*simp*]:
 $traverse-tree\ path\ (map-tree\ f\ t) = map-tree\ f\ (traverse-tree\ path\ t)$
 $\langle proof \rangle$

lemma *traverse-tree-append* [*simp*]:
 $traverse-tree\ (path\ @\ ext)\ t = traverse-tree\ ext\ (traverse-tree\ path\ t)$
 $\langle proof \rangle$

traverse-tree is an applicative-functor homomorphism.

lemma *traverse-tree-pure-tree* [*simp*]:
 $traverse-tree\ path\ (pure\ x) = pure\ x$

<proof>

lemma *traverse-tree-ap* [*simp*]:

traverse-tree path (f \diamond x) = traverse-tree path f \diamond traverse-tree path x

<proof>

context *fixes* $l\ r :: 'a \Rightarrow 'a$ **begin**

primrec *traverse-dir* :: $dir \Rightarrow 'a \Rightarrow 'a$

where

traverse-dir L = l

| *traverse-dir R = r*

abbreviation *traverse-path* :: $path \Rightarrow 'a \Rightarrow 'a$

where *traverse-path* \equiv *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*:

$\bigwedge x\ y\ z. f (f\ x\ y)\ z = f\ x (f\ y\ z)$

$\bigwedge x. f\ x\ \varepsilon = x$

$\bigwedge x. f\ \varepsilon\ x = x$

shows *tree-recurse (f l) (f r) ε = tree-iterate ($\lambda x. f\ x\ l$) ($\lambda x. f\ x\ r$) ε*

<proof>

1.2.4 Mirroring

primcorec *mirror* :: $'a\ tree \Rightarrow 'a\ tree$

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 \diamond x) = mirror f \diamond mirror x*

<proof>

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]:  $0 = \text{pure-tree } 0$   
instance  $\langle \text{proof} \rangle$   
end
```

```
instantiation tree :: (one) one begin  
definition [applicative-unfold]:  $1 = \text{pure-tree } 1$   
instance  $\langle \text{proof} \rangle$   
end
```

```
instantiation tree :: (plus) plus begin  
definition [applicative-unfold]:  $\text{plus } x \ y = \text{pure op } + \diamond x \diamond (y :: 'a \ \text{tree})$   
instance  $\langle \text{proof} \rangle$   
end
```

```
lemma plus-tree-simps [simp]:  
   $\text{root } (t + t') = \text{root } t + \text{root } t'$   
   $\text{left } (t + t') = \text{left } t + \text{left } t'$   
   $\text{right } (t + t') = \text{right } t + \text{right } t'$   
 $\langle \text{proof} \rangle$ 
```

```
friend-of-corec plus where  $t + t' = \text{Node } (\text{root } t + \text{root } t') (\text{left } t + \text{left } t')$   
 $(\text{right } t + \text{right } t')$   
 $\langle \text{proof} \rangle$ 
```

```
instantiation tree :: (minus) minus begin  
definition [applicative-unfold]:  $\text{minus } x \ y = \text{pure op } - \diamond x \diamond (y :: 'a \ \text{tree})$   
instance  $\langle \text{proof} \rangle$   
end
```

```
lemma minus-tree-simps [simp]:  
   $\text{root } (t - t') = \text{root } t - \text{root } t'$   
   $\text{left } (t - t') = \text{left } t - \text{left } t'$   
   $\text{right } (t - t') = \text{right } t - \text{right } t'$   
 $\langle \text{proof} \rangle$ 
```

```
instantiation tree :: (uminus) uminus begin  
definition [applicative-unfold tree]:  $\text{uminus} = (\text{op } \diamond (\text{pure } \text{uminus}) :: 'a \ \text{tree} \Rightarrow 'a \ \text{tree})$   
end
```


instance $\langle proof \rangle$
end

instantiation $tree :: (times) times$ **begin**
definition $[applicative-unfold]: times\ x\ y = pure\ op\ * \diamond x \diamond (y :: 'a\ tree)$
instance $\langle proof \rangle$
end

lemma $times-tree-simps$ $[simp]:$
 $root\ (t * t') = root\ t * root\ t'$
 $left\ (t * t') = left\ t * left\ t'$
 $right\ (t * t') = right\ t * right\ t'$
 $\langle proof \rangle$

instance $tree :: (Rings.dvd) Rings.dvd$ $\langle proof \rangle$

instantiation $tree :: (modulo) modulo$ **begin**
definition $[applicative-unfold]: x\ div\ y = pure-tree\ op\ div \diamond x \diamond (y :: 'a\ tree)$
definition $[applicative-unfold]: x\ mod\ y = pure-tree\ op\ mod \diamond x \diamond (y :: 'a\ tree)$
instance $\langle proof \rangle$
end

lemma $mod-tree-simps$ $[simp]:$
 $root\ (t mod t') = root\ t mod root\ t'$
 $left\ (t mod t') = left\ t mod left\ t'$
 $right\ (t mod t') = right\ t mod right\ t'$
 $\langle proof \rangle$

1.3.2 Algebraic instances

instance $tree :: (semigroup-add) semigroup-add$
 $\langle proof \rangle$

instance $tree :: (ab-semigroup-add) ab-semigroup-add$
 $\langle proof \rangle$

instance $tree :: (semigroup-mult) semigroup-mult$
 $\langle proof \rangle$

instance $tree :: (ab-semigroup-mult) ab-semigroup-mult$
 $\langle proof \rangle$

instance $tree :: (monoid-add) monoid-add$
 $\langle proof \rangle$

instance $tree :: (comm-monoid-add) comm-monoid-add$
 $\langle proof \rangle$

instance $tree :: (comm-monoid-diff) comm-monoid-diff$

<proof>

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* \longleftrightarrow *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

```

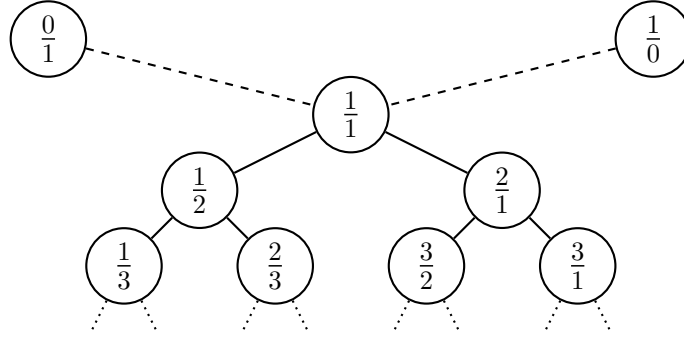


Figure 1: Constructing the Stern-Brocot tree iteratively.

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 $\text{fraction} = \text{nat} \times \text{nat}$

definition $\text{mediant} :: \text{fraction} \times \text{fraction} \Rightarrow \text{fraction}$
where $\text{mediant} \equiv \lambda((a, c), (b, d)). (a + b, c + d)$

definition $\text{stern-brocot} :: \text{fraction tree}$
where

$\text{stern-brocot} = \text{unfold-tree}$
 $(\lambda(\text{lb}, \text{ub}). \text{mediant} (\text{lb}, \text{ub}))$
 $(\lambda(\text{lb}, \text{ub}). (\text{lb}, \text{mediant} (\text{lb}, \text{ub})))$
 $(\lambda(\text{lb}, \text{ub}). (\text{mediant} (\text{lb}, \text{ub}), \text{ub}))$
 $((0, 1), (1, 0))$

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 $\text{succ} :: \text{fraction} \Rightarrow \text{fraction}$
where $\text{succ} \equiv \lambda(m, n). (m + n, n)$

abbreviation $\text{recip} :: \text{fraction} \Rightarrow \text{fraction}$
where $\text{recip} \equiv \lambda(m, n). (n, m)$

```

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 $op \diamond$ cannot be registered as friendly due to varying type parameters

```

lemma stern-brocot-unfold:
  stern-brocot-recurse =
    Node (1, 1)
      (pure recip  $\diamond$  (pure succ  $\diamond$  (pure recip  $\diamond$  stern-brocot-recurse)))
      (pure succ  $\diamond$  stern-brocot-recurse)
<proof>

```

```

lemma stern-brocot-simps [simp]:
  root stern-brocot-recurse = (1, 1)
  left stern-brocot-recurse = pure recip  $\diamond$  (pure succ  $\diamond$  (pure recip  $\diamond$  stern-brocot-recurse))
  right stern-brocot-recurse = pure succ  $\diamond$  stern-brocot-recurse
<proof>

```

```

lemma stern-brocot-conv:
  stern-brocot-recurse = tree-recurse (recip  $\circ$  succ  $\circ$  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)  $\Rightarrow$  m > 0  $\wedge$  n > 0
<proof>

```

```

lemma stern-brocot-coprime:
  case root (traverse-tree path stern-brocot-recurse) of (m, n)  $\Rightarrow$  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  $\Rightarrow$  nat  $\Rightarrow$  path where

```

$m = n \implies \text{mk-path } (\text{Suc } m) (\text{Suc } n) = []$
 $| m < n \implies \text{mk-path } (\text{Suc } m) (\text{Suc } n) = L \# \text{mk-path } (\text{Suc } m) (n - m)$
 $| m > n \implies \text{mk-path } (\text{Suc } m) (\text{Suc } n) = R \# \text{mk-path } (m - n) (\text{Suc } n)$
 $| \text{mk-path } 0 - = \text{undefined}$
 $| \text{mk-path } - 0 = \text{undefined}$
 $\langle \text{proof} \rangle$
termination $\text{mk-path } \langle \text{proof} \rangle$

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

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

theorem $\text{stern-brocot-rationals}$:

$\llbracket m > 0; n > 0 \rrbracket \implies$
 $\text{root } (\text{traverse-tree } (\text{mk-path } m \ n) (\text{pure } \text{rat-of} \ \diamond \ \text{stern-brocot-recurse})) = \text{Fract}$
 $(\text{int } m) (\text{int } n)$
 $\langle \text{proof} \rangle$

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 *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).

type-synonym $\text{matrix} = \text{fraction} \times \text{fraction}$

type-synonym $\text{vector} = \text{fraction}$

definition $\text{times-matrix} :: \text{matrix} \Rightarrow \text{matrix} \Rightarrow \text{matrix}$ (**infixl** \otimes 70)

where $\text{times-matrix} = (\lambda((a, c), (b, d)) ((a', c'), (b', d')).$

$((a * a' + b * c', c * a' + d * c'),$
 $(a * b' + b * d', c * b' + d * d'))$

definition $\text{times-vector} :: \text{matrix} \Rightarrow \text{vector} \Rightarrow \text{vector}$ (**infixr** \odot 70)

where $\text{times-vector} = (\lambda((a, c), (b, d)) (a', c'). (a * a' + b * c', c * a' + d * c'))$

context begin

private definition $F :: \text{matrix}$ **where** $F = ((0, 1), (1, 0))$

private definition $I :: \text{matrix}$ **where** $I = ((1, 0), (0, 1))$

private definition $LL :: \text{matrix}$ **where** $LL = ((1, 1), (0, 1))$

private definition $UR :: \text{matrix}$ **where** $UR = ((1, 0), (1, 1))$

definition $Det :: \text{matrix} \Rightarrow \text{nat}$ **where** $Det \equiv \lambda((a, c), (b, d)). a * d - b * c$

lemma $Dets$ [iff]:

$$Det\ I = 1$$

$$Det\ LL = 1$$

$$Det\ UR = 1$$

$\langle \text{proof} \rangle$

lemma $LL\text{-}UR\text{-}Det$:

$$Det\ m = 1 \implies Det\ (m \otimes LL) = 1$$

$$Det\ m = 1 \implies Det\ (LL \otimes m) = 1$$

$$Det\ m = 1 \implies Det\ (m \otimes UR) = 1$$

$$Det\ m = 1 \implies Det\ (UR \otimes m) = 1$$

$\langle \text{proof} \rangle$

lemma $mediant\text{-}I\text{-}F$ [simp]:

$$mediant\ F = (1, 1)$$

$$mediant\ I = (1, 1)$$

$\langle \text{proof} \rangle$

lemma $times\text{-}matrix\text{-}I$ [simp]:

$$I \otimes x = x$$

$$x \otimes I = x$$

$\langle \text{proof} \rangle$

lemma $times\text{-}matrix\text{-}assoc$ [simp]:

$$(x \otimes y) \otimes z = x \otimes (y \otimes z)$$

$\langle \text{proof} \rangle$

lemma $LL\text{-}UR\text{-}pos$:

$$0 < snd\ (mediant\ m) \implies 0 < snd\ (mediant\ (m \otimes LL))$$

$$0 < snd\ (mediant\ m) \implies 0 < snd\ (mediant\ (m \otimes UR))$$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

[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 $stern\text{-}brocot\text{-}iterate\text{-}aux :: \text{matrix} \Rightarrow \text{matrix tree}$

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

definition *stern-brocot-iterate* :: fraction tree
where *stern-brocot-iterate* \equiv 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 \otimes LL \otimes 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 \otimes UR \otimes 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*:
 $\exists m''. m \otimes m'' = \text{root (traverse-tree path (stern-brocot-iterate-aux m))} \wedge \text{Det } m'' = 1$
 ⟨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 \neq 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 xs' ys'*

where $xs = ps @ x \# xs'$ **and** $ys = ps @ y \# ys'$ **and** $x \neq y$

<proof>

lemma *stern-brocot-fractions-not-repeated*:

assumes $root (traverse-tree path stern-brocot-iterate) = root (traverse-tree path' stern-brocot-iterate)$

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 $a = c \wedge b = d$

<proof>

theorem *stern-brocot-rationals-not-repeated*:

assumes $root (traverse-tree path (pure rat-of \diamond stern-brocot-recurse))$

$= root (traverse-tree path' (pure rat-of \diamond stern-brocot-recurse))$

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

lemma *mediant-times-F*: $mediant \circ (\lambda s. s \otimes F) = mediant$

<proof>

lemma *stern-brocot-iterate*: $stern-brocot = stern-brocot-iterate$

<proof>

theorem *stern-brocot-mediante-recurse*: $stern-brocot = stern-brocot-recurse$

<proof>

end

no-notation *times-matrix* (**infixl** \otimes 70)
and *times-vector* (**infixl** \odot 70)

3 Linearising the Stern-Brocot Tree

3.1 Turning a tree into a stream

corec *tree-chop* :: 'a tree \Rightarrow 'a stream
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)
<proof>

tree-chop is an idiom homomorphism

lemma *tree-chop-pure-tree* [*simp*]:
 tree-chop (pure x) = pure x
<proof>

lemma *tree-chop-ap-tree* [*simp*]:
 tree-chop (f \diamond x) = *tree-chop* f \diamond *tree-chop* x
<proof>

lemma *tree-chop-plus*: *tree-chop* (t + t') = *tree-chop* t + *tree-chop* t'
<proof>

corec *stream* :: 'a tree \Rightarrow '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)
<proof>

stream is an idiom homomorphism.

lemma *stream-pure* [*simp*]: *stream* (pure x) = pure x
<proof>

lemma *stream-ap* [*simp*]: *stream* (f \diamond x) = *stream* f \diamond *stream* x
<proof>

lemma *stream-plus* [*simp*]: *stream* (t + t') = *stream* t + *stream* t'
<proof>

lemma *stream-minus* [simp]: $\text{stream } (t - t') = \text{stream } t - \text{stream } t'$
(proof)

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

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

lemma *stream-1* [simp]: $\text{stream } 1 = 1$
(proof)

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

3.2 Split the Stern-Brocot tree into numerators and denominators

corec *num-den* :: $\text{bool} \Rightarrow \text{nat tree}$

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** $\text{num} \equiv \text{num-den True}$

abbreviation *den* **where** $\text{den} \equiv \text{num-den False}$

lemma *num-unfold*: $\text{num} = \text{Node } 1 \text{ num } (\text{num} + \text{den})$
(proof)

lemma *den-unfold*: $\text{den} = \text{Node } 1 (\text{num} + \text{den}) \text{ den}$
(proof)

lemma *num-simps* [simp]:

root $\text{num} = 1$

left $\text{num} = \text{num}$

right $\text{num} = \text{num} + \text{den}$

(proof)

lemma *den-simps* [simp]:

root $\text{den} = 1$

left $\text{den} = \text{num} + \text{den}$

right $\text{den} = \text{den}$

(proof)

lemma *stern-brocot-num-den*:

pure-tree $\text{Pair } \diamond \text{ num } \diamond \text{ den} = \text{stern-brocot-recurse}$

(proof)

lemma *den-eq-chop-num*: $den = tree\text{-}chop\ num$
<proof>

lemma *num-conv*: $num = pure\ fst \diamond\ stern\text{-}brocot\text{-}recurse$
<proof>

lemma *den-conv*: $den = pure\ snd \diamond\ stern\text{-}brocot\text{-}recurse$
<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 $\forall i \in set\text{-}tree\ y. 0 < i$
 shows $x\ mod\ (x + y) = x$
<proof>

lemma *mod-tree-lemma2*:
 fixes *x y* :: '*a* :: *semiring-div tree*
 shows $(x + y)\ mod\ y = x\ mod\ y$
<proof>

lemma *set-tree-pathD*: $x \in set\text{-}tree\ t \implies \exists p. x = root\ (traverse\text{-}tree\ p\ t)$
<proof>

lemma *den-gt-0*: $0 < x$ **if** $x \in set\text{-}tree\ den$
<proof>

lemma *num-mod-den*: $num\ mod\ den = num\text{-}mod\text{-}den$
<proof>

lemma *tree-chop-den*: $tree\text{-}chop\ den = num + den - 2 * (num\ mod\ den)$
<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 $smap :: ('a \Rightarrow 'a) \Rightarrow 'a \text{ stream} \Rightarrow 'a \text{ stream}$
where $smap f xs = SCons (f (shd xs)) (smap f (stl xs))$
 $\langle proof \rangle$

definition $step :: nat \times nat \Rightarrow nat \times nat$
where $step = (\lambda(n, d). (d, n + d - 2 * (n \text{ mod } d)))$

corec $stern-brocot-loopless :: fraction \text{ stream}$
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** $s + s' = (shd s + shd s') \#\# (stl s + stl s')$
 $\langle proof \rangle$

friend-of-corec $minus$ **where** $t - t' = (shd t - shd t') \#\# (stl t - stl t')$
 $\langle proof \rangle$

friend-of-corec $times$ **where** $t * t' = (shd t * shd t') \#\# (stl t * stl t')$
 $\langle proof \rangle$

friend-of-corec $modulo$ **where** $t \text{ mod } t' = (shd t \text{ mod } shd t') \#\# (stl t \text{ mod } stl t')$
 $\langle proof \rangle$

corec $fusc' :: nat \text{ stream}$
where $fusc' = 1 \#\# (((1 \#\# fusc') + fusc') - 2 * ((1 \#\# fusc') \text{ mod } fusc'))$

definition $fusc$ **where** $fusc = 1 \#\# fusc'$

lemma $fusc-unfold: fusc = 1 \#\# fusc' \langle proof \rangle$

lemma $fusc'-unfold: fusc' = 1 \#\# (fusc + fusc' - 2 * (fusc \text{ mod } fusc'))$
 $\langle proof \rangle$

lemma $fusc-simps [simp]:$
 $shd fusc = 1$
 $stl fusc = fusc'$
 $\langle proof \rangle$

lemma $fusc'-simps [simp]:$

$shd\ fusc' = 1$
 $stl\ fusc' = fusc + fusc' - 2 * (fusc\ mod\ fusc')$
 <proof>

3.4 Equivalence with Dijkstra's fusc function

lemma *stern-brocot-loopless-siterate*: *stern-brocot-loopless = siterate step (1, 1)*
 <proof>

lemma *fusc-fusc'-iterate*: *pure Pair \diamond fusc \diamond fusc' = stern-brocot-loopless*
 <proof>

theorem *stern-brocot-loopless*:
 $stream\ stern-brocot-recurse = stern-brocot-loopless$ (**is** ?lhs = ?rhs)
 <proof>

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*:

$bird = Node\ (1, 1)\ (pure\ recip\ \diamond\ (pure\ succ\ \diamond\ bird))\ (pure\ succ\ \diamond\ (pure\ recip\ \diamond\ bird))$

<proof>

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

$root\ bird = (1, 1)$

$left\ bird = pure\ recip\ \diamond\ (pure\ succ\ \diamond\ bird)$

$right\ bird = pure\ succ\ \diamond\ (pure\ recip\ \diamond\ bird)$

<proof>

lemma *mirror-bird*: $mirror\ bird = pure\ recip\ \diamond\ bird$ (**is** ?lhs = ?rhs)

<proof>

primcorec *even-odd-mirror* :: *bool \Rightarrow 'a tree \Rightarrow 'a tree*

where

$\bigwedge even. root\ (even-odd-mirror\ even\ t) = root\ t$

| \wedge *even*. *left* (*even-odd-mirror even t*) = *even-odd-mirror* (\neg *even*) (if *even* then *right t* else *left t*)
| \wedge *even*. *right* (*even-odd-mirror even t*) = *even-odd-mirror* (\neg *even*) (if *even* then *left t* else *right t*)

definition *even-mirror* :: 'a tree \Rightarrow 'a tree
where *even-mirror* = *even-odd-mirror True*

definition *odd-mirror* :: 'a tree \Rightarrow 'a tree
where *odd-mirror* = *even-odd-mirror False*

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*)
<*proof*>

lemma *even-odd-mirror-pure* [*simp*]: **fixes** *even* **shows**
even-odd-mirror even (*pure-tree x*) = *pure-tree x*
<*proof*>

lemma *even-odd-mirror-ap-tree* [*simp*]: **fixes** *even* **shows**
even-odd-mirror even (*f* \diamond *x*) = *even-odd-mirror even f* \diamond *even-odd-mirror even*
x
<*proof*>

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*
<*proof*>

lemma [*simp*]:
shows *even-mirror-ap-tree*: *even-mirror* (*f* \diamond *x*) = *even-mirror f* \diamond *even-mirror*
x
and *odd-mirror-ap-tree*: *odd-mirror* (*f* \diamond *x*) = *odd-mirror f* \diamond *odd-mirror x*
<*proof*>

fun *even-mirror-path* :: *path* \Rightarrow *path*
and *odd-mirror-path* :: *path* \Rightarrow *path*
where

even-mirror-path [] = []
| *even-mirror-path* (*d* # *ds*) = (*case d of L* \Rightarrow *R* | *R* \Rightarrow *L*) # *odd-mirror-path ds*
| *odd-mirror-path* [] = []
| *odd-mirror-path* (*d* # *ds*) = *d* # *even-mirror-path ds*

lemma *even-mirror-traverse-tree* [simp]:
 $\text{root } (\text{traverse-tree path } (\text{even-mirror } t)) = \text{root } (\text{traverse-tree } (\text{even-mirror-path path}) t)$
and *odd-mirror-traverse-tree* [simp]:
 $\text{root } (\text{traverse-tree path } (\text{odd-mirror } t)) = \text{root } (\text{traverse-tree } (\text{odd-mirror-path path}) t)$
 ⟨proof⟩

lemma *even-odd-mirror-path-involution* [simp]:
 $\text{even-mirror-path } (\text{even-mirror-path path}) = \text{path}$
 $\text{odd-mirror-path } (\text{odd-mirror-path path}) = \text{path}$
 ⟨proof⟩

lemma *even-odd-mirror-path-injective* [simp]:
 $\text{even-mirror-path path} = \text{even-mirror-path path}' \longleftrightarrow \text{path} = \text{path}'$
 $\text{odd-mirror-path path} = \text{odd-mirror-path path}' \longleftrightarrow \text{path} = \text{path}'$
 ⟨proof⟩

lemma *odd-mirror-bird-stern-brocot*:
 $\text{odd-mirror bird} = \text{stern-brocot-recurse}$
 ⟨proof⟩

theorem *bird-rationals*:
assumes $m > 0 \ n > 0$
shows $\text{root } (\text{traverse-tree } (\text{odd-mirror-path } (\text{mk-path } m \ n)) (\text{pure rat-of } \diamond \text{ bird})) = \text{Fract } (\text{int } m) (\text{int } n)$
 ⟨proof⟩

theorem *bird-rationals-not-repeated*:
 $\text{root } (\text{traverse-tree path } (\text{pure rat-of } \diamond \text{ bird})) = \text{root } (\text{traverse-tree path}' (\text{pure rat-of } \diamond \text{ bird}))$
 $\implies \text{path} = \text{path}'$
 ⟨proof⟩

end

References

- Roland Backhouse and João F. Ferreira. Recounting the rationals: Twice! In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction (MPC 2008)*, volume 5133 of *LNCS*, pages 79–91. Springer, 2008. doi: 10.1007/978-3-540-70594-9_6.
- Richard S. Bird. Loopless functional algorithms. In Uustalu Tarmo, editor, *Mathematics of Program Construction*, volume 4014 of *LNCS*, pages 90–114. Springer, 2006. doi: 10.1007/11783596_9.
- Edsger W. Dijkstra. An exercise for Dr. R. M. Burstall. In *Selected*

Writings on Computing: A personal Perspective, Texts and Monographs in Computer Science, pages 215–216. Springer, 1982a. doi: 10.1007/978-1-4612-5695-3_36.

Edsger W. Dijkstra. More about the function “fusc” (a sequel to EWD570). In *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science, pages 230–232. Springer, 1982b. doi: 10.1007/978-1-4612-5695-3_41.

R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics—a Foundation for Computer Science*. Addison-Wesley, 2nd edition, 1994.

Ralf Hinze. The Bird tree. *Journal of Functional Programming*, 19(5):491–508, 2009. doi: 10.1017/S0956796809990116.

Ralf Hinze. Lifting operators and laws. <http://www.cs.ox.ac.uk/ralf.hinze/Lifting.pdf>, 2010.