

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	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	23
3.1	Turning a tree into a stream	23
3.2	Split the Stern-Brocot tree into numerators and denominators	24
3.3	Loopless linearisation of the Stern-Brocot tree.	27
3.4	Equivalence with Dijkstra's fusc function	28
4	The Bird tree	29

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:  $\bigwedge t. P$  (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
using x
proof(rule tree.set-induct)
  fix l x 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  $\bigwedge x. \text{stopL } x \implies \text{STOPL } x = \text{STOPL}' x$ 
  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
  fix x
  show ?lhs x = ?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  $\bigwedge s. \text{root } (f s) = \text{ROOT } s$ 
  and  $\bigwedge s. \text{left } (f s) = f (\text{LEFT } s)$ 
  and  $\bigwedge s. \text{right } (f s) = f (\text{RIGHT } s)$ 
  shows  $f s = \text{unfold-tree } \text{ROOT } \text{LEFT } \text{RIGHT } s$ 
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-funI)(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 \Rightarrow 'b) tree \Rightarrow 'a tree \Rightarrow '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]:
 $pure\ f\ \diamond\ Node\ x\ l\ r = Node\ (f\ x)\ (pure\ f\ \diamond\ l)\ (pure\ f\ \diamond\ r)$
by(rule tree.expand) auto

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

Applicative functor laws

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

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

lemma ap-tree-composition:
 $pure\ (op\ \circ)\ \diamond\ r1\ \diamond\ r2\ \diamond\ r3 = r1\ \diamond\ (r2\ \diamond\ r3)$
by(coinduction arbitrary: r1 r2 r3) auto

lemma ap-tree-homomorphism:
 $pure\ f\ \diamond\ pure\ x = pure\ (f\ x)$
by(simp add: map-tree-ap-tree-pure-tree)

lemma *ap-tree-interchange*:
 $t \diamond \text{pure } x = \text{pure } (\lambda f. f x) \diamond t$
by(*coinduction arbitrary*: t)(*auto*)

lemma *ap-tree-K-tree*: $\text{pure } (\lambda x y. x) \diamond u \diamond v = u$
by(*coinduction arbitrary*: $u v$)(*auto*)

lemma *ap-tree-C-tree*: $\text{pure } (\lambda f x y. f y x) \diamond u \diamond v \diamond w = u \diamond w \diamond v$
by(*coinduction arbitrary*: $u v w$)(*auto*)

lemma *ap-tree-W-tree*: $\text{pure } (\lambda f x. f x x) \diamond f \diamond x = f \diamond x \diamond x$
by(*coinduction arbitrary*: $f x$)(*auto*)

applicative *tree* (K, W) **for**

pure: *pure-tree*
ap: *ap-tree*
rel: *rel-tree*
set: *set-tree*

proof –

fix $R :: 'b \Rightarrow 'c \Rightarrow \text{bool}$ **and** $f :: ('a \Rightarrow 'b)$ *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* $R (f \diamond x) (g \diamond x)$ **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*:

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

proof(*coinduction arbitrary*: $f g$)

case [*rule-format*]: (*Eq-tree* $f g$)

have $\text{root } f = \text{root } g$

proof

fix x

show $\text{root } f x = \text{root } g x$

using *Eq-tree[of x]* **by**(*subst (asm) (1 2) ap-tree.ctr*) *simp*

qed

moreover {

fix x

have $\text{left } f \diamond \text{pure } x = \text{left } g \diamond \text{pure } x$

using *Eq-tree[of x]* **by**(*subst (asm) (1 2) ap-tree.ctr*) *simp*

} **moreover** {

fix x

have $\text{right } f \diamond \text{pure } x = \text{right } g \diamond \text{pure } x$

using *Eq-tree[of x]* **by**(*subst (asm) (1 2) ap-tree.ctr*) *simp*

} **ultimately show** *?case* **by** *simp*

qed

lemma *ap-tree-extensional*:
 $(\bigwedge x. f \diamond x = g \diamond x) \implies f = g$
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 $unf \equiv unfold-tree\ (\lambda f. f\ x)\ (\lambda f. f\ o\ l)\ (\lambda f. f\ o\ r)$
shows $map-tree\ G\ (unf\ F) = unf\ (G\ o\ F)$
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**
corec *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 \circ tree-iterate l r
by (*rule ext*)(*rule unfold-tree-unique[symmetric]*; *simp*)

lemma *tree-iterate-fusion*:

assumes $h \circ l = l' \circ h$
assumes $h \circ r = r' \circ h$
shows $\text{map-tree } h (\text{tree-iterate } l \ r \ x) = \text{tree-iterate } l' \ r' (h \ x)$
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* \Rightarrow '*a tree* \Rightarrow '*a tree*

where *traverse-tree path* \equiv *foldr* ($\lambda d \ f. f \circ \text{case-dir left right d}$) *path id*

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

traverse-tree [] = id
traverse-tree (d # path) = traverse-tree path \circ (case d of L \Rightarrow left | R \Rightarrow 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 \diamond x) = traverse-tree path f \diamond traverse-tree path x
by (*induct path arbitrary: f x*) (*simp-all split: dir.splits*)

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)

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*:

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

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

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

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

apply(*rule tree-recurse.unique[symmetric]*)

apply(*rule tree.expand*)

apply(*simp add: tree-iterate-fusion[where $r' = \lambda x. f x r$ and $l' = \lambda x. f x l$] fun-eq-iff monoid*)

done

1.2.4 Mirroring

primcorec *mirror* :: $'a \text{ tree} \Rightarrow 'a \text{ tree}$

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 \diamond x) = mirror f \diamond 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*]: $0 = \text{pure-tree } 0$
instance ..
end

instantiation *tree* :: (*one*) *one* **begin**
definition [*applicative-unfold*]: $1 = \text{pure-tree } 1$
instance ..
end

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

lemma *plus-tree-simps* [*simp*]:
 $\text{root } (t + t') = \text{root } t + \text{root } t'$
 $\text{left } (t + t') = \text{left } t + \text{left } t'$
 $\text{right } (t + t') = \text{right } t + \text{right } t'$
by(*simp-all add: plus-tree-def*)

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

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

lemma *minus-tree-simps* [*simp*]:
 $\text{root } (t - t') = \text{root } t - \text{root } t'$
 $\text{left } (t - t') = \text{left } t - \text{left } t'$
 $\text{right } (t - t') = \text{right } t - \text{right } t'$
by(*simp-all add: minus-tree-def*)

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

instantiation *tree* :: (*times*) *times* **begin**
definition [*applicative-unfold*]: $\text{times } x y = \text{pure } op * \diamond x \diamond (y :: 'a \text{ tree})$
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*]: $x \text{ div } y = \text{pure-tree op div } \diamond x \diamond (y :: 'a \text{ tree})$

definition [*applicative-unfold*]: $x \text{ mod } y = \text{pure-tree op mod } \diamond x \diamond (y :: 'a \text{ tree})$

instance ..

end

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

$root (t \text{ mod } t') = root t \text{ mod } root t'$

$left (t \text{ mod } t') = left t \text{ mod } left t'$

$right (t \text{ mod } t') = right t \text{ mod } right t'$

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-distrib)+

```

```

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  $\leftrightarrow$  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-neg-one) zero-neg-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

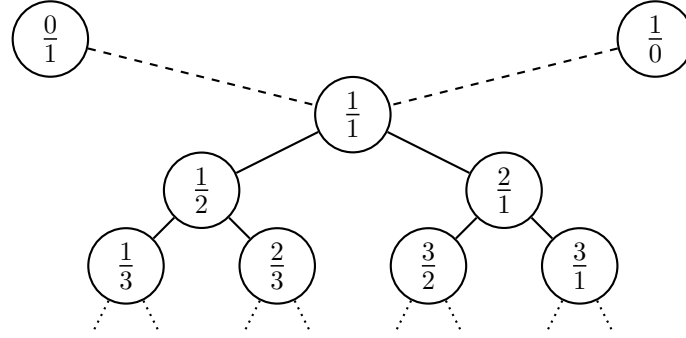


Figure 1: Constructing the Stern-Brocot tree iteratively.

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 *op* ◊ 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-recurse.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*:

stern-brocot-recurse = *tree-recurse* (*recip* \circ *succ* \circ *recip*) *succ* (1, 1)
apply(*rule tree-recurse.unique*)
apply(*subst stern-brocot-unfold*)
apply(*simp add: o-assoc*)
apply(*rule conjI; applicative-nf; simp*)
done

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 $m > 0 \wedge n > 0$
by(*induct path*)(*auto split: dir.splits*)

lemma *stern-brocot-coprime*:

case root (traverse-tree path stern-brocot-recurse) of (m, n) \Rightarrow coprime m n
by(*induct path*)(*clarsimp simp: field-simps split: dir.splits*)⁺

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* \Rightarrow *mk-path* (*Suc m*) (*Suc n*) = []
| *m* < *n* \Rightarrow *mk-path* (*Suc m*) (*Suc n*) = *L* # *mk-path* (*Suc m*) (*n - m*)
| *m* > *n* \Rightarrow *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* \Rightarrow *rat*

where *rat-of* \equiv $\lambda(x, y). \text{Fract } (\text{int } x) (\text{int } y)$

theorem *stern-brocot-rationals*:

$\llbracket m > 0; n > 0 \rrbracket \Rightarrow$
root (*traverse-tree* (*mk-path* *m n*) (*pure rat-of* \diamond *stern-brocot-recurse*)) = *Fract*
(*int m*) (*int n*)

```

proof(induction m n rule: mk-path-induct)
  case (less m n)
  with stern-brocot-denominator-non-zero[where path=mk-path (Suc m) (n - m)]
  show ?case
    by (simp add: eq-rat field-simps of-nat-diff split: prod.split-asm)
next
  case (greater m n)
  with stern-brocot-denominator-non-zero[where path=mk-path (m - n) (Suc n)]
  show ?case
    by (simp add: eq-rat field-simps of-nat-diff split: prod.split-asm)
qed (simp-all add: eq-rat)

```

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 *matrix* = *fraction* × *fraction*

type-synonym *vector* = *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 $Det :: matrix \Rightarrow nat$ **where** $Det \equiv \lambda((a, c), (b, d)). a * d - b * c$

lemma $Dets$ [iff]:

$$Det\ I = 1$$

$$Det\ LL = 1$$

$$Det\ UR = 1$$

unfolding Det -def I -def LL -def UR -def **by** $simp$ -all

lemma LL - UR - Det :

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

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

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

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

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

lemma $mediant$ - I - F [$simp$]:

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

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

by ($simp$ -all add : F -def I -def $mediant$ -def)

lemma $times$ -matrix- I [$simp$]:

$$I \otimes x = x$$

$$x \otimes I = x$$

by ($simp$ -all add : $times$ -matrix-def I -def $split$ -def)

lemma $times$ -matrix- $assoc$ [$simp$]:

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

by ($simp\ add$: $times$ -matrix-def $field$ -simps $split$ -def)

lemma LL - UR - pos :

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

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

by ($cases\ m$) ($simp$ -all add : LL -def UR -def $times$ -matrix-def $split$ -def $field$ -simps $mediant$ -def)

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

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 $stern$ - $brocot$ - $iterate$ - aux :: $matrix \Rightarrow matrix\ tree$

where $stern$ - $brocot$ - $iterate$ - $aux \equiv tree$ -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 –

```

have ?rhs = map-tree mediant (tree-recurse (op ⊗ LL) (op ⊗ UR) I)
  using tree-recurse-iterate[where f=op ⊗ and l=LL and r=UR and ε=I]
  by (simp add: stern-brocot-iterate-def stern-brocot-iterate-aux-def)
also have ... = tree-recurse (op ⊙ LL) (op ⊙ UR) (1, 1)
  unfolding mediant-I-F(2)[symmetric]
  by (rule tree-recurse-fusion)(simp-all add: fun-eq-iff mediant-def times-matrix-def
times-vector-def LL-def UR-def)[2]
also have ... = ?lhs
  by (simp add: stern-brocot-conv recip-succ-recip times-vector-def LL-def 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.

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 –
  from DX DY have F: 0 < snd (mediant (X ⊗ LL ⊗ Y))
    by (auto simp: Det-def times-matrix-def LL-def split-def mediant-def)
  obtain x11 x12 x21 x22 where X: X = ((x11, x12), (x21, x22)) by(cases X)
  auto
  obtain y11 y12 y21 y22 where Y: Y = ((y11, y12), (y21, y22)) by(cases Y)
  auto
  from DX DY have *: (x12 * x21) * (y12 + y22) < (x11 * x22) * (y12 + y22)
    by(simp add: X Y Det-def)(cases y12, simp-all add: field-simps)
  from DX DY MX F show ?thesis
    apply (simp add: split-def X Y of-nat-mult [symmetric] del: of-nat-mult)
    apply (clarsimp simp: Det-def times-matrix-def LL-def UR-def mediant-def
split-def)
    using * by (simp add: field-simps)
qed

```

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 –
  from DX DY have F: 0 < snd (mediant (X ⊗ UR ⊗ Y))
    by (auto simp: Det-def times-matrix-def UR-def split-def mediant-def)
  obtain x11 x12 x21 x22 where X: X = ((x11, x12), (x21, x22)) by(cases X)
  auto
  obtain y11 y12 y21 y22 where Y: Y = ((y11, y12), (y21, y22)) by(cases Y)
  auto
  show ?thesis using DX DY MX F
    apply (simp add: X Y split-def of-nat-mult [symmetric] del: of-nat-mult)

```

apply (*simp add: Det-def times-matrix-def LL-def UR-def mediant-def split-def algebra-simps*)
apply (*simp add: add-mult-distrib2[symmetric] mult.assoc[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*)
(simp-all add: stern-brocot-iterate-aux-def LL-UR-Det LL-UR-pos split: dir.splits)

lemma stern-brocot-iterate-aux-decompose:
 $\exists m''. m \otimes m'' = \text{root} (\text{traverse-tree path} (\text{stern-brocot-iterate-aux } m)) \wedge \text{Det } m'' = 1$
proof (*induction path arbitrary: m*)
case *Nil show ?case*
by (*auto simp add: stern-brocot-iterate-aux-def intro: exI[where x=I] simp del: split-paired-Ex*)
next
case (*Cons d ds m*)
from *Cons.IH[where m=m \otimes UR] Cons.IH[where m=m \otimes LL] show ?case*
by (*simp add: stern-brocot-iterate-aux-def split: dir.splits del: split-paired-Ex*)(*fastforce simp: LL-UR-Det*)
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')*
def *m \equiv 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*
def *m' \equiv 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 \Rightarrow m \otimes LL | R \Rightarrow m \otimes 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-tree-iterate split: dir.splits*)
then obtain *m'' where mm'm'': ?M \otimes m'' = m' and Dm'': Det m'' = 1*
using *stern-brocot-iterate-aux-decompose[where path=ds and m=?M] by*

clarsimp
hence case d of $L \Rightarrow \text{rat-of (mediant } m') < \text{rat-of (mediant } m) \mid R \Rightarrow \text{rat-of (mediant } m) < \text{rat-of (mediant } m')$
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 $p: \text{path} = \text{pref} @ d \# ds$

assumes $p': \text{path}' = \text{pref} @ d' \# ds'$

assumes $dd': d \neq d'$

shows *False*

proof –

def $m \equiv \text{root (traverse-tree pref (stern-brocot-iterate-aux } I))$

then have $Dm: \text{Det } m = 1$ **and** $Pm: 0 < \text{snd (mediant } m)$

using *stern-brocot-iterate-aux-Det*[**where** $\text{path}=\text{pref}$ **and** $m=I$] **by** *simp-all*

def $pm \equiv \text{root (traverse-tree path (stern-brocot-iterate-aux } I))$

then have $Dpm: \text{Det } pm = 1$

using *stern-brocot-iterate-aux-Det*[**where** $\text{path}=\text{path}$ **and** $m=I$] **by** *simp*

let $?M = \text{case } d \text{ of } L \Rightarrow m \otimes LL \mid R \Rightarrow m \otimes UR$

from p

have $\text{root (traverse-tree } ds \text{ (stern-brocot-iterate-aux } ?M)) = pm$

by(*simp add: stern-brocot-iterate-aux-def m-def pm-def traverse-tree-tree-iterate split: dir.splits*)

then obtain pm'

where $pm': ?M \otimes pm' = pm$ **and** $Dpm': \text{Det } pm' = 1$

using *stern-brocot-iterate-aux-decompose*[**where** $\text{path}=ds$ **and** $m=?M$] **by**

clarsimp

hence case d of $L \Rightarrow \text{rat-of (mediant } pm) < \text{rat-of (mediant } m) \mid R \Rightarrow \text{rat-of (mediant } m) < \text{rat-of (mediant } pm)$

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

def $p'm \equiv \text{root (traverse-tree path' (stern-brocot-iterate-aux } I))$

then have $Dp'm: \text{Det } p'm = 1$

using *stern-brocot-iterate-aux-Det*[**where** $\text{path}=\text{path}'$ **and** $m=I$] **by** *simp*

let $?M' = \text{case } d' \text{ of } L \Rightarrow m \otimes LL \mid R \Rightarrow m \otimes UR$

from p'

have $\text{root (traverse-tree } ds' \text{ (stern-brocot-iterate-aux } ?M')) = p'm$

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 $p'm': ?M' \otimes p'm' = p'm$ **and** $Dp'm': \text{Det } p'm' = 1$

using *stern-brocot-iterate-aux-decompose*[**where** $path=ds'$ **and** $m=?M'$] **by**
clarsimp
hence $case\ d'\ of\ L \Rightarrow rat-of\ (mediant\ p'm) < rat-of\ (mediant\ m) \mid R \Rightarrow rat-of\ (mediant\ m) < rat-of\ (mediant\ p'm)$
using *tree-ordering-left*[*OF Dm Dp'm' Pm, unfolded pm'*]
tree-ordering-right[*OF Dm Dp'm' Pm, unfolded pm'*]
by (*simp split: dir.splits*)
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*)
qed

lemma *lists-not-eq*:

assumes $xs \neq ys$
obtains
 $(c1)\ strict-prefix\ xs\ ys$
 $\mid (c2)\ strict-prefix\ ys\ xs$
 $\mid (c3)\ ps\ x\ y\ xs'\ ys'$
where $xs = ps @ x \# xs'$ **and** $ys = ps @ y \# ys'$ **and** $x \neq y$
using *assms*
by (*cases xs ys rule: prefix-cases*)
(blast dest: parallel-decomp prefix-order.neq-le-trans)+

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(*rule ccontr*)
assume $path \neq path'$
then show *False using assms*
by (*cases path path' rule: lists-not-eq*)
(blast intro: stern-brocot-fractions-not-repeated-strict-prefix sym stern-brocot-fractions-not-repeated-parallel)+
qed

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 –
from $\langle b > 0 \rangle \langle d > 0 \rangle \langle Fract\ a\ b = Fract\ c\ d \rangle$
have $*$: $a * d = c * b$ **by** (*simp add: eq-rat*)
from *arg-cong*[**where** $f=sgn$, *OF this*] $\langle b > 0 \rangle \langle d > 0 \rangle$
have $sgn\ a = sgn\ c$ **by** (*simp add: sgn-mult*)
with $*$ **show** *?thesis*
using $\langle b > 0 \rangle \langle d > 0 \rangle$ *coprime-crossproduct-int*[*OF* $\langle coprime\ a\ b \rangle \langle coprime\ c$

d)]
 by (simp add: abs-sgn)
 qed

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'
 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']
 by(*auto simp: transfer-int-nat-gcd dest!: rat-inv-eq intro: stern-brocot-fractions-not-repeated*
simp add: stern-brocot-recurse-iterate[symmetric] split: prod.splits)

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*
 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*
 by(*subst unfold-tree-tree-iterate*)(*simp add: F-def times-matrix-def mediant-def*
UR-def LL-def split-def)
also have \dots = map-tree *mediant* (map-tree ($\lambda s. s \otimes F$) (tree-iterate ($\lambda s. s \otimes$
 LL) ($\lambda s. s \otimes UR$) I))
unfolding *stern-brocot-Hinze-iterate-def*
 by(*subst tree-iterate-fusion*[**where** $l' = \lambda s. s \otimes UR$ **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 \dots = *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-mediante-recurse*: *stern-brocot* = *stern-brocot-recurse*
 by(*simp add: stern-brocot-recurse-iterate stern-brocot-iterate*)

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 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 \diamond x) = *tree-chop* f \diamond *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 \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)
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 \diamond x) = *stream* f \diamond *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* [*simp*]: *stream (t mod t') = stream t mod stream t'*
by(*simp add: modulo-stream-def modulo-tree-def*)

lemma *stream-1* [*simp*]: *stream 1 = 1*
by(*simp add: one-tree-def one-stream-def*)

lemma *stream-numeral* [*simp*]: *stream (numeral n) = numeral n*
by(*induct n*)(*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)*
by(*subst num-den.code; simp*)

lemma *den-unfold*: *den = Node 1 (num + den) den*
by(*subst num-den.code; simp*)

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

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

lemma *stern-brocot-num-den*:
pure-tree Pair \diamond num \diamond den = stern-brocot-recurse
apply(*rule stern-brocot-recurse.unique*)
apply(*subst den-unfold*)
apply(*subst num-unfold*)
apply(*simp; intro conjI*)
apply(*applicative-lifting; simp*)+

done

lemma *den-eq-chop-num*: $den = tree\text{-}chop\ num$
by(*coinduction rule*: *tree.coinduct-strong*) *simp*

lemma *num-conv*: $num = pure\ fst \diamond\ stern\text{-}brocot\text{-}recurse$
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*: $den = pure\ snd \diamond\ stern\text{-}brocot\text{-}recurse$
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 $\forall i \in set\text{-}tree\ y. 0 < i$
 shows $x\ mod\ (x + y) = x$
proof –
 have *rel-tree op* = $(x\ mod\ (x + y))\ x$ **by** *applicative-lifting*(*simp add*: *assms*)
 thus *?thesis* **by**(*unfold tree.rel-eq*)
qed

lemma *mod-tree-lemma2*:
 fixes *x y* :: '*a* :: *semiring-div tree*
 shows $(x + y)\ mod\ y = x\ mod\ y$
by *applicative-lifting simp*

lemma *set-tree-pathD*: $x \in set\text{-}tree\ t \implies \exists p. x = root\ (traverse\text{-}tree\ p\ t)$
by(*induct rule*: *set-tree-induct*)(*auto intro*: *exI*[**where** *x=*[]] *exI*[**where** *x=L* # *p*
for *p*] *exI*[**where** *x=R* # *p* **for** *p*])

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

proof –
from *that obtain p where* $x = \text{root } (\text{traverse-tree } p \text{ den})$ **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: $\bigwedge x y :: \text{nat. } 0 < y \implies 2 * (x \text{ mod } y) \leq x + y$*
by (*metis Divides.mod-less-eq-dividend add-le-mono mod-le-divisor mult-2*)

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

def *den' \equiv pure int \diamond den*
def *num' \equiv pure int \diamond num*
def *num-mod-den' \equiv 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'-simps* [*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-mod-den'2-unique: $\bigwedge x. x = \text{Node } 0 (2 * \text{num}') x \implies x = 2 * \text{num-mod-den}'$*

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 $\dots = \text{num}' + \text{den}' - 2 * (\text{num}' \text{ mod } \text{den}')$

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 op = \dots (pure int \diamond (num + den - 2 * num-mod-den'))*

```

(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  $\diamond$  (pure int  $\diamond$  (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  $\Rightarrow$  'a)  $\Rightarrow$  'a stream  $\Rightarrow$  '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  $\times$  nat  $\Rightarrow$  nat  $\times$  nat
where step = ( $\lambda$ (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 $fusc' = 1 \ \#\# \ (((1 \ \#\# \ fusc') + fusc') - 2 * ((1 \ \#\# \ fusc') \bmod fusc'))$

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

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

lemma $fusc'$ -unfold: $fusc' = 1 \ \#\# \ (fusc + fusc' - 2 * (fusc \bmod fusc'))$
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 \bmod 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: $pure\ Pair \diamond fusc \diamond fusc' = stern$ -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 \diamond num \diamond 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*

by(*rule stream.expand*)(*simp add: den-eq-chop-num[symmetric] tree-chop-den; applicative-lifting; simp add: step-def*)

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* \diamond (*pure succ* \diamond *bird*)) (*pure succ* \diamond (*pure recip* \diamond *bird*))

using *bird.code* **unfolding** *map-tree-ap-tree-pure-tree[symmetric]* .

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*)

by(*subst bird-unfold, simp*)**+**

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

proof(*rule sym*)

let ?*F* = $\lambda t.$ *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* \diamond *bird* = ?*F* (*pure recip* \diamond *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* \Rightarrow '*a tree* \Rightarrow '*a tree*

where

\wedge *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]:
 $\text{root } (\text{even-mirror } t) = \text{root } t$
 $\text{left } (\text{even-mirror } t) = \text{odd-mirror } (\text{right } t)$
 $\text{right } (\text{even-mirror } t) = \text{odd-mirror } (\text{left } t)$
and *odd-mirror-simps* [simp]:
 $\text{root } (\text{odd-mirror } t) = \text{root } t$
 $\text{left } (\text{odd-mirror } t) = \text{even-mirror } (\text{left } t)$
 $\text{right } (\text{odd-mirror } t) = \text{even-mirror } (\text{right } t)$
by(*simp-all add: even-mirror-def odd-mirror-def*)

lemma *even-odd-mirror-pure* [simp]: **fixes** *even shows*
 $\text{even-odd-mirror even } (\text{pure-tree } x) = \text{pure-tree } x$
by(*coinduction arbitrary: even*) *auto*

lemma *even-odd-mirror-ap-tree* [simp]: **fixes** *even shows*
 $\text{even-odd-mirror even } (f \diamond x) = \text{even-odd-mirror even } f \diamond \text{even-odd-mirror even } x$
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* ($f \diamond x$) = *even-mirror* $f \diamond \text{even-mirror } x$
and *odd-mirror-ap-tree: odd-mirror* ($f \diamond x$) = *odd-mirror* $f \diamond \text{odd-mirror } x$
by(*simp-all add: even-mirror-def odd-mirror-def*)

fun *even-mirror-path* :: *path* \Rightarrow *path*
and *odd-mirror-path* :: *path* \Rightarrow *path*
where
 $\text{even-mirror-path } [] = []$
 $|\text{ even-mirror-path } (d \# ds) = (\text{case } d \text{ of } L \Rightarrow R \mid R \Rightarrow L) \# \text{odd-mirror-path } ds$
 $|\text{ odd-mirror-path } [] = []$
 $|\text{ odd-mirror-path } (d \# ds) = d \# \text{even-mirror-path } ds$

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]:
 $\text{root } (\text{traverse-tree path } (\text{odd-mirror } t)) = \text{root } (\text{traverse-tree } (\text{odd-mirror-path path}) t)$
by (*induct path arbitrary: t*) (*simp-all split: dir.splits*)

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

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'* \longleftrightarrow *path* = *path'*
odd-mirror-path path = *odd-mirror-path path'* \longleftrightarrow *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 *?rskr* = *map-tree* (*recip* \circ *succ* \circ *succ* \circ *recip*)
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*) (*?rskr t*) (*?rsrs t*)) (*Node* (*2*, *1*) (*?srsr t*) (*?srrs t*))

have *: *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** *that* * **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* \diamond *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* \diamond *bird*)) = *root* (*traverse-tree path'* (*pure rat-of* \diamond *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, *Mathematics of Program Construction (MPC 2008)*, volume 5133 of *LNCIS*, 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 *LNCIS*, 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.