The Stern-Brocot Tree

Peter Gammie Andreas Lochbihler

February 23, 2021

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

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*: $x \in$ *set-tree t*
and *root*: $\bigwedge t.\ P\ (root\ t)\ t$
and *left*: $\bigwedge x\ t.\ [\![\ x \in set\text{-}tree\ (left\ t);\ P\ x\ (left\ t)\ ]\!] \Longrightarrow P\ x\ t$
and *right*: $\bigwedge x\ t.\ [\![\ x \in set\text{-}tree\ (right\ t);\ P\ x\ (right\ t)\ ]\!] \Longrightarrow P\ x\ t$
shows $P\ x\ t$
⟨*proof*⟩

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

shows *corec-tree ROOT stopL STOPL LEFT stopR STOPR RIGHT* =
corec-tree ROOT stopL STOPL′ LEFT′ stopR STOPR′ RIGHT′
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

context
fixes *g1* :: *′a* ⇒ *′b*
and *g22* :: *′a* ⇒ *′a*
and *g32* :: *′a* ⇒ *′a*
begin

corec *unfold-tree* :: *′a* ⇒ *′b tree*
where *unfold-tree a* = *Node* (*g1 a*) (*unfold-tree* (*g22 a*)) (*unfold-tree* (*g32 a*))

lemma *unfold-tree-simps* [*simp*]:
root (*unfold-tree a*) = *g1 a*
left (*unfold-tree a*) = *unfold-tree* (*g22 a*)
right (*unfold-tree a*) = *unfold-tree* (*g32 a*)
⟨*proof*⟩

end

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

1.1 Applicative functor for *′a tree*

context fixes *x* :: *′a* **begin**
corec *pure-tree* :: *′a tree*
where *pure-tree* = *Node x pure-tree pure-tree*
end

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

lemma *pure-tree-simps* [*simp*]:
root (*pure-tree x*) = *x*
left (*pure-tree x*) = *pure-tree x*
right (*pure-tree x*) = *pure-tree x*
⟨*proof*⟩

adhoc-overloading *pure pure-tree*

lemma *pure-tree-parametric* [*transfer-rule*]: (*rel-fun A* (*rel-tree A*)) *pure pure*
⟨*proof*⟩

lemma *map-pure-tree* [*simp*]: *map-tree f* (*pure x*) = *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
  root (*ap-tree f x*) = *root f* (*root x*)
| *left* (*ap-tree f x*) = *ap-tree* (*left f*) (*left x*)
| *right* (*ap-tree f x*) = *ap-tree* (*right f*) (*right x*)

adhoc-overloading *Applicative.ap ap-tree*

unbundle *applicative-syntax*

lemma *ap-tree-pure-Node* [*simp*]:
  pure f ⋄ *Node x l r* = *Node* (*f x*) (*pure f* ⋄ *l*) (*pure f* ⋄ *r*)
⟨*proof*⟩

lemma *ap-tree-Node-Node* [*simp*]:
  Node f fl fr ⋄ *Node x l r* = *Node* (*f x*) (*fl* ⋄ *l*) (*fr* ⋄ *r*)
⟨*proof*⟩

Applicative functor laws

lemma *map-tree-ap-tree-pure-tree*:
  pure f ⋄ *u* = *map-tree f u*
⟨*proof*⟩

lemma *ap-tree-identity*: *pure id* ⋄ *t* = *t*
⟨*proof*⟩

lemma *ap-tree-composition*:
  pure (∘) ⋄ *r1* ⋄ *r2* ⋄ *r3* = *r1* ⋄ (*r2* ⋄ *r3*)
⟨*proof*⟩

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

lemma *ap-tree-interchange*:
  t ⋄ *pure x* = *pure* ($\lambda f.\ f\ x$) ⋄ *t*
⟨*proof*⟩

lemma *ap-tree-K-tree*: *pure* ($\lambda x\ y.\ x$) ⋄ *u* ⋄ *v* = *u*
⟨*proof*⟩

lemma *ap-tree-C-tree*: *pure* ($\lambda f\ x\ y.\ f\ y\ x$) ⋄ *u* ⋄ *v* ⋄ *w* = *u* ⋄ *w* ⋄ *v*
⟨*proof*⟩

lemma *ap-tree-W-tree*: $pure\ (\lambda f\ x.\ f\ x\ x) \diamond f \diamond x = f \diamond x \diamond x$
$\langle proof \rangle$

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

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

lemma *ap-tree-strong-extensional*:
  $(\bigwedge x.\ f \diamond pure\ x = g \diamond pure\ x) \Longrightarrow f = g$
$\langle proof \rangle$

lemma *ap-tree-extensional*:
  $(\bigwedge x.\ f \diamond x = g \diamond x) \Longrightarrow f = g$
$\langle proof \rangle$

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 $unf \equiv unfold\text{-}tree\ (\lambda f.\ f\ x)\ (\lambda f.\ f \circ l)\ (\lambda f.\ f \circ r)$
 shows $map\text{-}tree\ G\ (unf\ F) = unf\ (G \circ F)$
$\langle proof \rangle$

friend-of-corec *map-tree* :: $('a \Rightarrow 'a) \Rightarrow 'a\ tree \Rightarrow 'a\ tree$ **where**
  $map\text{-}tree\ f\ t = Node\ (f\ (root\ t))\ (map\text{-}tree\ f\ (left\ t))\ (map\text{-}tree\ f\ (right\ t))$
$\langle proof \rangle$

context fixes $l :: 'a \Rightarrow 'a$ **and** $r :: 'a \Rightarrow 'a$ **and** $x :: 'a$ **begin**
corec *tree-recurse* :: $'a\ tree$
where $tree\text{-}recurse = Node\ x\ (map\text{-}tree\ l\ tree\text{-}recurse)\ (map\text{-}tree\ r\ tree\text{-}recurse)$
end

lemma *tree-recurse-simps* [*simp*]:
  $root\ (tree\text{-}recurse\ l\ r\ x) = x$
  $left\ (tree\text{-}recurse\ l\ r\ x) = map\text{-}tree\ l\ (tree\text{-}recurse\ l\ r\ x)$
  $right\ (tree\text{-}recurse\ l\ r\ x) = map\text{-}tree\ r\ (tree\text{-}recurse\ l\ r\ x)$
$\langle proof \rangle$

lemma *tree-recurse-unfold*:

$tree\text{-}recurse\ l\ r\ x = Node\ x\ (map\text{-}tree\ l\ (tree\text{-}recurse\ l\ r\ x))\ (map\text{-}tree\ r\ (tree\text{-}recurse\ l\ r\ x))$
⟨*proof*⟩

lemma *tree-recurse-fusion*:
assumes $h \circ l = l' \circ h$ **and** $h \circ r = r' \circ h$
shows $map\text{-}tree\ h\ (tree\text{-}recurse\ l\ r\ x) = tree\text{-}recurse\ l'\ r'\ (h\ x)$
⟨*proof*⟩

1.2.2 Tree iteration

context fixes $l :: 'a \Rightarrow 'a$ **and** $r :: 'a \Rightarrow 'a$ **begin**
primcorec $tree\text{-}iterate :: \ 'a \Rightarrow 'a\ tree$
where $tree\text{-}iterate\ s = Node\ s\ (tree\text{-}iterate\ (l\ s))\ (tree\text{-}iterate\ (r\ s))$
end

lemma *unfold-tree-tree-iterate*:
$unfold\text{-}tree\ out\ l\ r = map\text{-}tree\ out \circ tree\text{-}iterate\ l\ r$
⟨*proof*⟩

lemma *tree-iterate-fusion*:
assumes $h \circ l = l' \circ h$
assumes $h \circ r = r' \circ h$
shows $map\text{-}tree\ h\ (tree\text{-}iterate\ l\ r\ x) = tree\text{-}iterate\ l'\ r'\ (h\ x)$
⟨*proof*⟩

1.2.3 Tree traversal

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

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

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

lemma *traverse-tree-map-tree* [*simp*]:
$traverse\text{-}tree\ path\ (map\text{-}tree\ f\ t) = map\text{-}tree\ f\ (traverse\text{-}tree\ path\ t)$
⟨*proof*⟩

lemma *traverse-tree-append* [*simp*]:
$traverse\text{-}tree\ (path\ @\ ext)\ t = traverse\text{-}tree\ ext\ (traverse\text{-}tree\ path\ t)$
⟨*proof*⟩

traverse-tree is an applicative-functor homomorphism.

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

$\langle proof \rangle$

lemma *traverse-tree-ap* [*simp*]:
  $traverse\text{-}tree\ path\ (f \diamond x) = traverse\text{-}tree\ path\ f \diamond traverse\text{-}tree\ path\ x$
$\langle proof \rangle$

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

primrec $traverse\text{-}dir :: dir \Rightarrow 'a \Rightarrow 'a$
where
  $traverse\text{-}dir\ L = l$
| $traverse\text{-}dir\ R = r$

abbreviation $traverse\text{-}path :: path \Rightarrow 'a \Rightarrow 'a$
where $traverse\text{-}path \equiv fold\ traverse\text{-}dir$

end

lemma *traverse-tree-tree-iterate*:
  $traverse\text{-}tree\ path\ (tree\text{-}iterate\ l\ r\ s) =$
   $tree\text{-}iterate\ l\ r\ (traverse\text{-}path\ l\ r\ path\ s)$
$\langle proof \rangle$

? 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 $tree\text{-}recurse\ (f\ l)\ (f\ r)\ \varepsilon = tree\text{-}iterate\ (\lambda x.\ f\ x\ l)\ (\lambda x.\ f\ x\ r)\ \varepsilon$
$\langle proof \rangle$

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*: $mirror\ (Node\ x\ l\ r) = Node\ x\ (mirror\ r)\ (mirror\ l)$
$\langle proof \rangle$

lemma *mirror-pure*: $mirror\ (pure\ x) = pure\ x$
$\langle proof \rangle$

lemma *mirror-ap-tree*: $mirror\ (f \diamond x) = mirror\ f \diamond mirror\ x$
$\langle proof \rangle$

end

1.3 Pointwise arithmetic on infinite binary trees

theory *Cotree-Algebra*
imports *Cotree*
begin

1.3.1 Constants and operators

instantiation *tree* :: (*zero*) *zero* **begin**
definition [*applicative-unfold*]: *0* = *pure-tree 0*
instance ⟨*proof*⟩
end

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

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

lemma *plus-tree-simps* [*simp*]:
  root (t + t') = *root* t + *root* t'
  left (t + t') = *left* t + *left* t'
  right (t + t') = *right* t + *right* t'
⟨*proof*⟩

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

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

lemma *minus-tree-simps* [*simp*]:
  root (t − t') = *root* t − *root* t'
  left (t − t') = *left* t − *left* t'
  right (t − t') = *right* t − *right* t'
⟨*proof*⟩

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

instance $\langle proof \rangle$
end

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

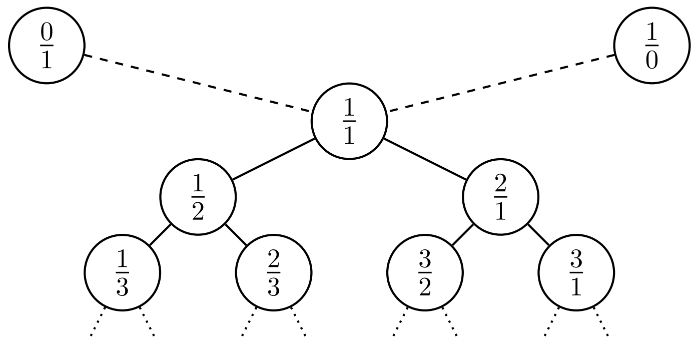

Figure 1: Constructing the Stern-Brocot tree iteratively.

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 *fraction* = *nat* × *nat*

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

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

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

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

2.1 Specification via a recursion equation

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

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

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

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

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

lemma *stern-brocot-unfold*:
stern-brocot-recurse =
Node (*1*, *1*)
(*pure recip* ⋄ (*pure succ* ⋄ (*pure recip* ⋄ *stern-brocot-recurse*)))
(*pure succ* ⋄ *stern-brocot-recurse*)
⟨*proof*⟩

lemma *stern-brocot-simps* [*simp*]:
root stern-brocot-recurse = (*1*, *1*)
left stern-brocot-recurse = *pure recip* ⋄ (*pure succ* ⋄ (*pure recip* ⋄ *stern-brocot-recurse*))
right stern-brocot-recurse = *pure succ* ⋄ *stern-brocot-recurse*
⟨*proof*⟩

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

2.2 Basic properties

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

lemma *stern-brocot-denominator-non-zero*:
case root (*traverse-tree path stern-brocot-recurse*) *of* (*m*, *n*) ⇒ *m* > *0* ∧ *n* > *0*
⟨*proof*⟩

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

2.3 All the rationals

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

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

$m = n \Longrightarrow$ *mk-path* $(Suc\ m)$ $(Suc\ n) = []$
| $m < n \Longrightarrow$ *mk-path* $(Suc\ m)$ $(Suc\ n) = L\ \#$ *mk-path* $(Suc\ m)$ $(n - m)$
| $m > n \Longrightarrow$ *mk-path* $(Suc\ m)$ $(Suc\ n) = R\ \#$ *mk-path* $(m - n)$ $(Suc\ n)$
| *mk-path 0 - = undefined*
| *mk-path - 0 = undefined*
⟨*proof*⟩
termination *mk-path* ⟨*proof*⟩

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

abbreviation *rat-of* :: *fraction* ⇒ *rat*
where *rat-of* ≡ $\lambda(x, y)$. *Fract* (*int* x) (*int* y)

theorem *stern-brocot-rationals*:
⟦ $m > 0$; $n > 0$ ⟧ ⟹
root (*traverse-tree* (*mk-path* m n) (*pure rat-of* ⋄ *stern-brocot-recurse*)) = *Fract* (*int* m) (*int* n)
⟨*proof*⟩

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* = $(\lambda((a, c), (b, d))\ ((a', c'), (b', d'))$.
$((a * a' + b * c', c * a' + d * c'),$
$(a * b' + b * d', c * b' + d * d')))$

definition *times-vector* :: *matrix* ⇒ *vector* ⇒ *vector* (**infixr** ⊙ *70*)
where *times-vector* = $(\lambda((a, c), (b, d))\ (a', c').\ (a * a' + b * c', c * a' + d * c'))$

context begin

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

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

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

lemma *LL-UR-Det*:
  $Det\ m = 1 \Longrightarrow Det\ (m \otimes LL) = 1$
  $Det\ m = 1 \Longrightarrow Det\ (LL \otimes m) = 1$
  $Det\ m = 1 \Longrightarrow Det\ (m \otimes UR) = 1$
  $Det\ m = 1 \Longrightarrow Det\ (UR \otimes m) = 1$
⟨*proof*⟩

lemma *mediant-I-F* [*simp*]:
  $mediant\ F = (1, 1)$
  $mediant\ I = (1, 1)$
⟨*proof*⟩

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

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

lemma *LL-UR-pos*:
  $0 < snd\ (mediant\ m) \Longrightarrow 0 < snd\ (mediant\ (m \otimes LL))$
  $0 < snd\ (mediant\ m) \Longrightarrow 0 < snd\ (mediant\ (m \otimes UR))$
⟨*proof*⟩

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

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

definition *stern-brocot-iterate-aux* :: *matrix* ⇒ *matrix tree*
where *stern-brocot-iterate-aux* ≡ *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*⟩

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

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

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

lemma *stern-brocot-iterate-aux-Det*:
  assumes *Det m* = *1* *0* < *snd* (*mediant m*)
  shows *Det* (*root* (*traverse-tree path* (*stern-brocot-iterate-aux m*))) = *1*
  and *0* < *snd* (*mediant* (*root* (*traverse-tree path* (*stern-brocot-iterate-aux m*))))
⟨*proof*⟩

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

lemma *stern-brocot-fractions-not-repeated-strict-prefix*:
  assumes *root* (*traverse-tree path stern-brocot-iterate*) = *root* (*traverse-tree path' stern-brocot-iterate*)
  assumes *pp'*: *strict-prefix path path'*
  shows *False*
⟨*proof*⟩

lemma *stern-brocot-fractions-not-repeated-parallel*:
  assumes *root* (*traverse-tree path stern-brocot-iterate*) = *root* (*traverse-tree path' stern-brocot-iterate*)
  assumes *p*: *path* = *pref* @ *d* # *ds*
  assumes *p'*: *path'* = *pref* @ *d'* # *ds'*
  assumes *dd'*: *d* ≠ *d'*
  shows *False*
⟨*proof*⟩

lemma *lists-not-eq*:
assumes $xs \neq ys$
obtains
$(c1)$ *strict-prefix xs ys*
| $(c2)$ *strict-prefix ys xs*
| $(c3)$ *ps x y 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* ◇ *stern-brocot-recurse*))
= *root* (*traverse-tree path'* (*pure rat-of* ◇ *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* ∘ ($\lambda s.\ s \otimes F$) = *mediant*
⟨*proof*⟩

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

theorem *stern-brocot-mediant-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* $\Rightarrow$ $'a$ *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)
⟨*proof*⟩

tree-chop is a idiom homomorphism

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

lemma *tree-chop-ap-tree* [*simp*]:
 tree-chop ($f \diamond x$) = *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*]: *stream* ($t - t'$) = *stream* t − *stream* t'

⟨*proof*⟩

lemma *stream-times* [*simp*]: $stream\ (t * t') = stream\ t * stream\ t'$
⟨*proof*⟩

lemma *stream-mod* [*simp*]: $stream\ (t\ mod\ t') = stream\ t\ mod\ stream\ t'$
⟨*proof*⟩

lemma *stream-1* [*simp*]: $stream\ 1 = 1$
⟨*proof*⟩

lemma *stream-numeral* [*simp*]: $stream\ (numeral\ n) = numeral\ n$
⟨*proof*⟩

3.2 Split the Stern-Brocot tree into numerators and denumerators

corec *num-den* :: *bool* ⇒ *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* ≡ *num-den True*
abbreviation *den* **where** *den* ≡ *num-den False*

lemma *num-unfold*: *num* = *Node 1 num* (*num* + *den*)
⟨*proof*⟩

lemma *den-unfold*: *den* = *Node 1* (*num* + *den*) *den*
⟨*proof*⟩

lemma *num-simps* [*simp*]:
  root num = *1*
  left num = *num*
  right num = *num* + *den*
⟨*proof*⟩

lemma *den-simps* [*simp*]:
  root den = *1*
  left den = *num* + *den*
  right den = *den*
⟨*proof*⟩

lemma *stern-brocot-num-den*:
  pure-tree Pair ⋄ *num* ⋄ *den* = *stern-brocot-recurse*
⟨*proof*⟩

lemma *den-eq-chop-num*: *den* = *tree-chop num*
⟨*proof*⟩

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

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

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

lemma *num-mod-den-simps* [*simp*]:
  root num-mod-den = *0*
  left num-mod-den = *num*
  right num-mod-den = *num-mod-den*
⟨*proof*⟩

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

lemma *mod-tree-lemma1*:
  fixes *x* :: *nat tree*
  assumes ∀ *i*∈*set-tree y*. *0* < *i*
  shows *x mod* (*x* + *y*) = *x*
⟨*proof*⟩

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

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

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

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

lemma *tree-chop-den*: *tree-chop den* = *num* + *den* − *2* ∗ (*num mod den*)
⟨*proof*⟩

3.3 Loopless linearisation of the Stern-Brocot tree.

This is a loopless linearisation of the Stern-Brocot tree that gives Stern's diatomic sequence, which is also known as Dijkstra's fusc function Dijkstra

(1982a,b). Loopless à la Bird (2006) means that the first element of the stream can be computed in linear time and every further element in constant time.

friend-of-corec *smap* :: $('a \Rightarrow 'a) \Rightarrow 'a\ stream \Rightarrow 'a\ stream$
where *smap f xs* = *SCons* (*f* (*shd xs*)) (*smap f* (*stl xs*))
⟨*proof*⟩

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** $s + s' = (shd\ s + shd\ s')$ ## $(stl\ s + stl\ s')$
⟨*proof*⟩

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

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

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

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

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

lemma *fusc-unfold*: *fusc* = *1* ## *fusc'* ⟨*proof*⟩

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

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

lemma *fusc'-simps* [*simp*]:
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* ⋄ *fusc* ⋄ *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* ⋄ (*pure succ* ⋄ *bird*)) (*pure succ* ⋄ (*pure recip* ⋄ *bird*))
⟨*proof*⟩

lemma *bird-simps* [*simp*]:
root bird = (*1*, *1*)
left bird = *pure recip* ⋄ (*pure succ* ⋄ *bird*)
right bird = *pure succ* ⋄ (*pure recip* ⋄ *bird*)
⟨*proof*⟩

lemma *mirror-bird*: *mirror bird* = *pure recip* ⋄ *bird* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

primcorec *even-odd-mirror* :: *bool* ⇒ *′a tree* ⇒ *′a tree*
where
⋀*even*. *root* (*even-odd-mirror even t*) = *root t*
| ⋀*even*. *left* (*even-odd-mirror even t*) = *even-odd-mirror* (¬ *even*) (*if even then right t else left t*)
| ⋀*even*. *right* (*even-odd-mirror even t*) = *even-odd-mirror* (¬ *even*) (*if even then left t else right t*)

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

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

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* ⋄ *x*) = *even-odd-mirror even f* ⋄ *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* ⋄ *x*) = *even-mirror f* ⋄ *even-mirror x*
and *odd-mirror-ap-tree*: *odd-mirror* (*f* ⋄ *x*) = *odd-mirror f* ⋄ *odd-mirror x*
⟨*proof*⟩

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

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

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

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

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

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

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

end

References

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.