FingerTrees

Benedikt Nordhoff Stefan Körner Peter Lammich

April 18, 2024

Abstract

We implement and prove correct 2-3 finger trees. Finger trees are a general purpose data structure, that can be used to efficiently implement other data structures, such as priority queues. Intuitively, a finger tree is an annotated sequence, where the annotations are elements of a monoid. Apart from operations to access the ends of the sequence, the main operation is to split the sequence at the point where a *monotone predicate* over the sum of the left part of the sequence becomes true for the first time. The implementation follows the paper of Hinze and Paterson[1]. The code generator can be used to get efficient, verified code.

Contents

1 2-3 Finger Trees

theory *FingerTree*
imports *Main*
begin

We implement and prove correct 2-3 finger trees as described by Ralf Hinze and Ross Paterson[1].

This theory is organized as follows: Section 1.1 contains the finger-tree datatype, its invariant and its abstraction function to lists. The Section 1.2 contains the operations on finger trees and their correctness lemmas. Section 1.3 contains a finger tree datatype with implicit invariant, and, finally, Section 1.4 contains a documentation of the implemented operations.

Technical Issues As Isabelle lacks proper support of namespaces, we try to simulate namespaces by locales.

The problem is, that we define lots of internal functions that should not be exposed to the user at all. Moreover, we define some functions with names equal to names from Isabelle's standard library. These names make perfect sense in the context of FingerTrees, however, they shall not be exposed to anyone using this theory indirectly, hiding the standard library names there.

Our approach puts all functions and lemmas inside the locale *FingerTree_loc*, and then interprets this locale with the prefix *FingerTree*. This makes all definitions visible outside the locale, with qualified names. Inside the locale, however, one can use unqualified names.

1.1 Datatype definition

locale *FingerTreeStruc-loc*

Nodes: Non empty 2-3 trees, with all elements stored within the leafs plus a cached annotation

datatype *('e,'a) Node = Tip 'e 'a |*
  Node2 'a ('e,'a) Node ('e,'a) Node |
  Node3 'a ('e,'a) Node ('e,'a) Node ('e,'a) Node

Digit: one to four ordered Nodes

datatype *('e,'a) Digit = One ('e,'a) Node |*
  Two ('e,'a) Node ('e,'a) Node |
  Three ('e,'a) Node ('e,'a) Node ('e,'a) Node |
  Four ('e,'a) Node ('e,'a) Node ('e,'a) Node ('e,'a) Node

FingerTreeStruc: The empty tree, a single node or some nodes and a deeper tree

```
datatype ('e, 'a) FingerTreeStruc =
  Empty |
  Single ('e,'a) Node |
  Deep 'a ('e,'a) Digit ('e,'a) FingerTreeStruc ('e,'a) Digit
```

1.1.1 Invariant

```
context FingerTreeStruc-loc
begin
```

Auxiliary functions

Readout the cached annotation of a node

```
primrec gmn :: ('e,'a::monoid-add) Node ⇒ 'a where
  gmn (Tip e a) = a |
  gmn (Node2 a - -) = a |
  gmn (Node3 a - - -) = a
```

The annotation of a digit is computed on the fly

```
primrec gmd :: ('e,'a::monoid-add) Digit ⇒ 'a where
  gmd (One a) = gmn a |
  gmd (Two a b) = (gmn a) + (gmn b)|
  gmd (Three a b c) = (gmn a) + (gmn b) + (gmn c)|
  gmd (Four a b c d) = (gmn a) + (gmn b) + (gmn c) + (gmn d)
```

Readout the cached annotation of a finger tree

```
primrec gmft :: ('e,'a::monoid-add) FingerTreeStruc ⇒ 'a where
  gmft Empty = 0 |
  gmft (Single nd) = gmn nd |
  gmft (Deep a - - -) = a
```

Depth and cached annotations have to be correct

```
fun is-leveln-node :: nat ⇒ ('e,'a) Node ⇒ bool where
  is-leveln-node 0 (Tip - -) ⟷ True |
  is-leveln-node (Suc n) (Node2 - n1 n2) ⟷
    is-leveln-node n n1 ∧ is-leveln-node n n2 |
  is-leveln-node (Suc n) (Node3 - n1 n2 n3) ⟷
    is-leveln-node n n1 ∧ is-leveln-node n n2 ∧ is-leveln-node n n3 |
  is-leveln-node - - ⟷ False
```

```
primrec is-leveln-digit :: nat ⇒ ('e,'a) Digit ⇒ bool where
  is-leveln-digit n (One n1) ⟷ is-leveln-node n n1 |
  is-leveln-digit n (Two n1 n2) ⟷ is-leveln-node n n1 ∧
    is-leveln-node n n2 |
  is-leveln-digit n (Three n1 n2 n3) ⟷ is-leveln-node n n1 ∧
```

is-leveln-node n n2 ∧ *is-leveln-node n n3* |
is-leveln-digit n (*Four n1 n2 n3 n4*) ⟷ *is-leveln-node n n1* ∧
is-leveln-node n n2 ∧ *is-leveln-node n n3* ∧ *is-leveln-node n n4*

primrec *is-leveln-ftree* :: *nat* ⇒ (*'e*,*'a*) *FingerTreeStruc* ⇒ *bool* **where**
is-leveln-ftree n Empty ⟷ *True* |
is-leveln-ftree n (*Single nd*) ⟷ *is-leveln-node n nd* |
is-leveln-ftree n (*Deep - l t r*) ⟷ *is-leveln-digit n l* ∧
is-leveln-digit n r ∧ *is-leveln-ftree* (*Suc n*) *t*

primrec *is-measured-node* :: (*'e*,*'a*::*monoid-add*) *Node* ⇒ *bool* **where**
is-measured-node (*Tip - -*) ⟷ *True* |
is-measured-node (*Node2 a n1 n2*) ⟷ ((*is-measured-node n1*) ∧
(*is-measured-node n2*)) ∧ (*a* = (*gmn n1*) + (*gmn n2*)) |
is-measured-node (*Node3 a n1 n2 n3*) ⟷ ((*is-measured-node n1*) ∧
(*is-measured-node n2*) ∧ (*is-measured-node n3*)) ∧
(*a* = (*gmn n1*) + (*gmn n2*) + (*gmn n3*))

primrec *is-measured-digit* :: (*'e*,*'a*::*monoid-add*) *Digit* ⇒ *bool* **where**
is-measured-digit (*One a*) = *is-measured-node a* |
is-measured-digit (*Two a b*) =
((*is-measured-node a*) ∧ (*is-measured-node b*))|
is-measured-digit (*Three a b c*) =
((*is-measured-node a*) ∧ (*is-measured-node b*) ∧ (*is-measured-node c*))|
is-measured-digit (*Four a b c d*) = ((*is-measured-node a*) ∧
(*is-measured-node b*) ∧ (*is-measured-node c*) ∧ (*is-measured-node d*))

primrec *is-measured-ftree* :: (*'e*,*'a*::*monoid-add*) *FingerTreeStruc* ⇒ *bool* **where**
is-measured-ftree Empty ⟷ *True* |
is-measured-ftree (*Single n1*) ⟷ (*is-measured-node n1*) |
is-measured-ftree (*Deep a l m r*) ⟷ ((*is-measured-digit l*) ∧
(*is-measured-ftree m*) ∧ (*is-measured-digit r*)) ∧
(*a* = ((*gmd l*) + (*gmft m*) + (*gmd r*)))

Structural invariant for finger trees

definition *ft-invar t* == *is-leveln-ftree 0 t* ∧ *is-measured-ftree t*

1.1.2 Abstraction to Lists

primrec *nodeToList* :: (*'e*,*'a*) *Node* ⇒ (*'e* × *'a*) *list* **where**
nodeToList (*Tip e a*) = [(*e*,*a*)]|
nodeToList (*Node2 - a b*) = (*nodeToList a*) @ (*nodeToList b*)|
nodeToList (*Node3 - a b c*)
= (*nodeToList a*) @ (*nodeToList b*) @ (*nodeToList c*)

primrec *digitToList* :: (*'e*,*'a*) *Digit* ⇒ (*'e* × *'a*) *list* **where**
digitToList (*One a*) = *nodeToList a*|
digitToList (*Two a b*) = (*nodeToList a*) @ (*nodeToList b*)|
digitToList (*Three a b c*)

  = (*nodeToList a*) @ (*nodeToList b*) @ (*nodeToList c*)|
 digitToList (*Four a b c d*)
  = (*nodeToList a*) @ (*nodeToList b*) @ (*nodeToList c*) @ (*nodeToList d*)

List representation of a finger tree

primrec *toList* :: (*'e* ,*'a*) *FingerTreeStruc* ⇒ (*'e* × *'a*) *list* **where**
 toList Empty = []|
 toList (*Single a*) = *nodeToList a*|
 toList (*Deep - pr m sf*) = (*digitToList pr*) @ (*toList m*) @ (*digitToList sf*)

lemma *nodeToList-empty*: *nodeToList nd* ≠ *Nil*
 ⟨*proof*⟩

lemma *digitToList-empty*: *digitToList d* ≠ *Nil*
 ⟨*proof*⟩

Auxiliary lemmas

lemma *gmn-correct*:
 assumes *is-measured-node nd*
 shows *gmn nd* = *sum-list* (*map snd* (*nodeToList nd*))
 ⟨*proof*⟩

lemma *gmd-correct*:
 assumes *is-measured-digit d*
 shows *gmd d* = *sum-list* (*map snd* (*digitToList d*))
 ⟨*proof*⟩

lemma *gmft-correct*: *is-measured-ftree t*
 ⟹ (*gmft t*) = *sum-list* (*map snd* (*toList t*))
 ⟨*proof*⟩
lemma *gmft-correct2*: *ft-invar t* ⟹ (*gmft t*) = *sum-list* (*map snd* (*toList t*))
 ⟨*proof*⟩

1.2 Operations

1.2.1 Empty tree

lemma *Empty-correct*[*simp*]:
 toList Empty = []
 ft-invar Empty
 ⟨*proof*⟩

Exactly the empty finger tree represents the empty list

lemma *toList-empty*: *toList t* = [] ⟷ *t* = *Empty*
 ⟨*proof*⟩

1.2.2 Annotation

Sum of annotations of all elements of a finger tree

```
definition annot :: ('e,'a::monoid-add) FingerTreeStruc ⇒ 'a
  where annot t = gmft t

lemma annot-correct:
  ft-invar t ⟹ annot t = sum-list (map snd (toList t))
  ⟨proof⟩
```

1.2.3 Appending

Auxiliary functions to fill in the annotations

```
definition deep:: ('e,'a::monoid-add) Digit ⇒ ('e,'a) FingerTreeStruc
    ⇒ ('e,'a) Digit ⇒ ('e, 'a) FingerTreeStruc where
  deep pr m sf = Deep ((gmd pr) + (gmft m) + (gmd sf)) pr m sf
definition node2 where
  node2 nd1 nd2 = Node2 ((gmn nd1)+(gmn nd2)) nd1 nd2
definition node3 where
  node3 nd1 nd2 nd3 = Node3 ((gmn nd1)+(gmn nd2)+(gmn nd3)) nd1 nd2 nd3
```

Append a node at the left end

```
fun nlcons :: ('e,'a::monoid-add) Node ⇒ ('e,'a) FingerTreeStruc
    ⇒ ('e,'a) FingerTreeStruc
where
— Recursively we append a node, if the digit is full we push down a node3
  nlcons a Empty = Single a |
  nlcons a (Single b) = deep (One a) Empty (One b) |
  nlcons a (Deep - (One b) m sf) = deep (Two a b) m sf |
  nlcons a (Deep - (Two b c) m sf) = deep (Three a b c) m sf |
  nlcons a (Deep - (Three b c d) m sf) = deep (Four a b c d) m sf |
  nlcons a (Deep - (Four b c d e) m sf)
    = deep (Two a b) (nlcons (node3 c d e) m) sf
```

Append a node at the right end

```
fun nrcons :: ('e,'a::monoid-add) FingerTreeStruc
    ⇒ ('e,'a) Node ⇒ ('e,'a) FingerTreeStruc  where
  — Recursively we append a node, if the digit is full we push down a node3
  nrcons Empty a = Single a |
  nrcons (Single b) a = deep (One b) Empty (One a) |
  nrcons (Deep - pr m (One b)) a = deep pr m (Two  b a)|
  nrcons (Deep - pr m (Two b c)) a = deep pr m (Three b c a) |
  nrcons (Deep - pr m (Three b c d)) a = deep pr m (Four b c d a) |
  nrcons (Deep - pr m (Four b c d e)) a
    = deep pr (nrcons m (node3 b c d)) (Two e a)

lemma nlcons-invlevel: ⟦is-leveln-ftree n t; is-leveln-node n nd⟧
  ⟹ is-leveln-ftree n (nlcons nd t)
  ⟨proof⟩

lemma nlcons-invmeas: ⟦is-measured-ftree t; is-measured-node nd⟧
```

$\Longrightarrow$ *is-measured-ftree* (*nlcons nd t*)
⟨*proof*⟩

lemmas *nlcons-inv* = *nlcons-invlevel nlcons-invmeas*

lemma *nlcons-list*: *toList* (*nlcons a t*) = (*nodeToList a*) @ (*toList t*)
⟨*proof*⟩

lemma *nrcons-invlevel*: ⟦*is-leveln-ftree n t*; *is-leveln-node n nd*⟧
$\Longrightarrow$ *is-leveln-ftree n* (*nrcons t nd*)
⟨*proof*⟩

lemma *nrcons-invmeas*: ⟦*is-measured-ftree t*; *is-measured-node nd*⟧
$\Longrightarrow$ *is-measured-ftree* (*nrcons t nd*)
⟨*proof*⟩

lemmas *nrcons-inv* = *nrcons-invlevel nrcons-invmeas*

lemma *nrcons-list*: *toList* (*nrcons t a*) = (*toList t*) @ (*nodeToList a*)
⟨*proof*⟩

Append an element at the left end

definition *lcons* :: ('*e* × '*a*::*monoid-add*)
  ⇒ ('*e*,'*a*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *FingerTreeStruc* (**infixr** ◁ *65*) **where**
a ◁ *t* = *nlcons* (*Tip* (*fst a*) (*snd a*)) *t*

lemma *lcons-correct*:
assumes *ft-invar t*
shows *ft-invar* (*a* ◁ *t*) **and** *toList* (*a* ◁ *t*) = *a* # (*toList t*)
⟨*proof*⟩

lemma *lcons-inv*:*ft-invar t* $\Longrightarrow$ *ft-invar* (*a* ◁ *t*)
⟨*proof*⟩

lemma *lcons-list*[*simp*]: *toList* (*a* ◁ *t*) = *a* # (*toList t*)
⟨*proof*⟩

Append an element at the right end

definition *rcons*
:: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e* × '*a*) ⇒ ('*e*,'*a*) *FingerTreeStruc*
  (**infixl** ▷ *65*) **where**
t ▷ *a* = *nrcons t* (*Tip* (*fst a*) (*snd a*))

lemma *rcons-correct*:
assumes *ft-invar t*
shows *ft-invar* (*t* ▷ *a*) **and** *toList* (*t* ▷ *a*) = (*toList t*) @ [*a*]
⟨*proof*⟩

lemma *rcons-inv*:*ft-invar t* $\Longrightarrow$ *ft-invar* (*t* ▷ *a*)

⟨*proof*⟩

lemma *rcons-list*[*simp*]: *toList* (*t* ⊳ *a*) = (*toList t*) @ [*a*]
⟨*proof*⟩

1.2.4 Convert list to tree

```
primrec toTree :: ('e × 'a::monoid-add) list ⇒ ('e,'a) FingerTreeStruc where
  toTree [] = Empty|
  toTree (a#xs) = a ◁ (toTree xs)
```

```
lemma toTree-correct[simp]:
  ft-invar (toTree l)
  toList (toTree l) = l
  ⟨proof⟩
```

Note that this lemma is a completeness statement of our implementation, as it can be read as: „All lists of elements have a valid representation as a finger tree."

1.2.5 Detaching leftmost/rightmost element

```
primrec digitToTree :: ('e,'a::monoid-add) Digit ⇒ ('e,'a) FingerTreeStruc
  where
  digitToTree (One a) = Single a|
  digitToTree (Two a b) = deep (One a) Empty (One b)|
  digitToTree (Three a b c) = deep (Two a b) Empty (One c)|
  digitToTree (Four a b c d) = deep (Two a b) Empty (Two c d)
```

```
primrec nodeToDigit :: ('e,'a) Node ⇒ ('e,'a) Digit where
  nodeToDigit (Tip e a) = One (Tip e a)|
  nodeToDigit (Node2 - a b) = Two a b|
  nodeToDigit (Node3 - a b c) = Three a b c
```

```
fun nlistToDigit :: ('e,'a) Node list ⇒ ('e,'a) Digit where
  nlistToDigit [a] = One a |
  nlistToDigit [a,b] = Two a b |
  nlistToDigit [a,b,c] = Three a b c |
  nlistToDigit [a,b,c,d] = Four a b c d
```

```
primrec digitToNlist :: ('e,'a) Digit ⇒ ('e,'a) Node list where
  digitToNlist (One a) = [a] |
  digitToNlist (Two a b) = [a,b]  |
  digitToNlist (Three a b c) = [a,b,c] |
  digitToNlist (Four a b c d) = [a,b,c,d]
```

Auxiliary function to unwrap a Node element

```
primrec n-unwrap:: ('e,'a) Node ⇒ ('e × 'a) where
  n-unwrap (Tip e a) = (e,a)|
```

n-unwrap (*Node2* - *a b*) = *undefined*|
n-unwrap (*Node3* - *a b c*) = *undefined*

type-synonym ('*e*,'*a*) *ViewnRes* = (('*e*,'*a*) *Node* × ('*e*,'*a*) *FingerTreeStruc*) *option*
lemma *viewnres-cases*:
fixes *r* :: ('*e*,'*a*) *ViewnRes*
obtains (*Nil*) *r*=*None* |
(*Cons*) *a t* **where** *r*=*Some* (*a*,*t*)
⟨*proof*⟩

lemma *viewnres-split*:
P (*case-option f1* (*case-prod f2*) *x*) =
((x = *None* $\longrightarrow$ *P f1*) $\wedge$ ($\forall$ *a b*. x = *Some* (*a*,*b*) $\longrightarrow$ *P* (*f2 a b*)))
⟨*proof*⟩

Detach the leftmost node. Return *None* on empty finger tree.

fun *viewLn* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *ViewnRes* **where**
viewLn Empty = *None*|
viewLn (*Single a*) = *Some* (*a*, *Empty*)|
viewLn (*Deep* - (*Two a b*) *m sf*) = *Some* (*a*, (*deep* (*One b*) *m sf*))|
viewLn (*Deep* - (*Three a b c*) *m sf*) = *Some* (*a*, (*deep* (*Two b c*) *m sf*))|
viewLn (*Deep* - (*Four a b c d*) *m sf*) = *Some* (*a*, (*deep* (*Three b c d*) *m sf*))|
viewLn (*Deep* - (*One a*) *m sf*) =
(*case viewLn m of*
None ⇒ *Some* (*a*, (*digitToTree sf*)) |
Some (*b*, *m2*) ⇒ *Some* (*a*, (*deep* (*nodeToDigit b*) *m2 sf*)))

Detach the rightmost node. Return *None* on empty finger tree.

fun *viewRn* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *ViewnRes* **where**
viewRn Empty = *None* |
viewRn (*Single a*) = *Some* (*a*, *Empty*) |
viewRn (*Deep* - *pr m* (*Two a b*)) = *Some* (*b*, (*deep pr m* (*One a*))) |
viewRn (*Deep* - *pr m* (*Three a b c*)) = *Some* (*c*, (*deep pr m* (*Two a b*))) |
viewRn (*Deep* - *pr m* (*Four a b c d*)) = *Some* (*d*, (*deep pr m* (*Three a b c*))) |
viewRn (*Deep* - *pr m* (*One a*)) =
(*case viewRn m of*
None ⇒ *Some* (*a*, (*digitToTree pr*))|
Some (*b*, *m2*) ⇒ *Some* (*a*, (*deep pr m2* (*nodeToDigit b*))))

lemma
digitToTree-inv: *is-leveln-digit n d* ⟹ *is-leveln-ftree n* (*digitToTree d*)
is-measured-digit d ⟹ *is-measured-ftree* (*digitToTree d*)
⟨*proof*⟩

lemma *digitToTree-list*: *toList* (*digitToTree d*) = *digitToList d*

⟨*proof*⟩

lemma *nodeToDigit-inv*:
is-leveln-node (*Suc n*) *nd* $\Longrightarrow$ *is-leveln-digit n* (*nodeToDigit nd*)
is-measured-node nd $\Longrightarrow$ *is-measured-digit* (*nodeToDigit nd*)
⟨*proof*⟩

lemma *nodeToDigit-list*: *digitToList* (*nodeToDigit nd*) = *nodeToList nd*
⟨*proof*⟩

lemma *viewLn-empty*: $t \neq$ *Empty* $\longleftrightarrow$ (*viewLn t*) $\neq$ *None*
⟨*proof*⟩

lemma *viewLn-inv*: ⟦
is-measured-ftree t; *is-leveln-ftree n t*; *viewLn t* = *Some* (*nd*, *s*)
⟧ $\Longrightarrow$ *is-measured-ftree s* $\wedge$ *is-measured-node nd* $\wedge$
is-leveln-ftree n s $\wedge$ *is-leveln-node n nd*
⟨*proof*⟩

lemma *viewLn-list*: *viewLn t* = *Some* (*nd*, *s*)
$\Longrightarrow$ *toList t* = (*nodeToList nd*) @ (*toList s*)
⟨*proof*⟩

lemma *viewRn-empty*: $t \neq$ *Empty* $\longleftrightarrow$ (*viewRn t*) $\neq$ *None*
⟨*proof*⟩

lemma *viewRn-inv*: ⟦
is-measured-ftree t; *is-leveln-ftree n t*; *viewRn t* = *Some* (*nd*, *s*)
⟧ $\Longrightarrow$ *is-measured-ftree s* $\wedge$ *is-measured-node nd* $\wedge$
is-leveln-ftree n s $\wedge$ *is-leveln-node n nd*
⟨*proof*⟩

lemma *viewRn-list*: *viewRn t* = *Some* (*nd*, *s*)
$\Longrightarrow$ *toList t* = (*toList s*) @ (*nodeToList nd*)
⟨*proof*⟩

type-synonym ('*e*,'*a*) *viewres* = (('*e* × '*a*) × ('*e*,'*a*) *FingerTreeStruc*) *option*

Detach the leftmost element. Return *None* on empty finger tree.

definition *viewL* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *viewres*
where
viewL t = (*case viewLn t of*
None ⇒ *None* |
(*Some* (*a*, *t2*)) ⇒ *Some* ((*n-unwrap a*), *t2*))

lemma *viewL-correct*:
assumes *INV*: *ft-invar t*

shows
$(t{=}Empty \Longrightarrow viewL\ t = None)$
$(t{\neq}Empty \Longrightarrow (\exists\, a\ s.\ viewL\ t = Some\ (a,\ s) \wedge ft\text{-}invar\ s$
$\wedge\ toList\ t = a\ \#\ toList\ s))$
⟨*proof*⟩

lemma *viewL-correct-empty*[*simp*]: *viewL Empty = None*
⟨*proof*⟩

lemma *viewL-correct-nonEmpty*:
assumes *ft-invar t t ≠ Empty*
obtains *a s* **where**
viewL t = Some (a, s) ft-invar s toList t = a # toList s
⟨*proof*⟩

Detach the rightmost element. Return *None* on empty finger tree.

definition *viewR* :: *('e,'a::monoid-add) FingerTreeStruc ⇒ ('e,'a) viewres*
where
viewR t = (case viewRn t of
None ⇒ None |
(Some (a, t2)) ⇒ Some ((n-unwrap a), t2))

lemma *viewR-correct*:
assumes *INV*: *ft-invar t*
shows
$(t = Empty \Longrightarrow viewR\ t = None)$
$(t \neq Empty \Longrightarrow (\exists\ a\ s.\ viewR\ t = Some\ (a,\ s) \wedge ft\text{-}invar\ s$
$\wedge\ toList\ t = toList\ s\ @\ [a]))$
⟨*proof*⟩

lemma *viewR-correct-empty*[*simp*]: *viewR Empty = None*
⟨*proof*⟩

lemma *viewR-correct-nonEmpty*:
assumes *ft-invar t t ≠ Empty*
obtains *a s* **where**
viewR t = Some (a, s) ft-invar s ∧ toList t = toList s @ [a]
⟨*proof*⟩

Finger trees viewed as a double-ended queue. The head and tail functions here are only defined for non-empty queues, while the view-functions were also defined for empty finger trees.

Check for emptiness

definition *isEmpty* :: *('e,'a) FingerTreeStruc ⇒ bool* **where**
[*code del*]: *isEmpty t = (t = Empty)*
lemma *isEmpty-correct*: *isEmpty t ⟷ toList t = []*
⟨*proof*⟩
lemma [*code*]: *isEmpty t = (case t of Empty ⇒ True | - ⇒ False)*

⟨*proof*⟩

Leftmost element

definition *head* :: (′*e*,′*a*::*monoid-add*) *FingerTreeStruc* ⇒ ′*e* × ′*a* **where**
head t = (*case viewL t of* (*Some* (*a*, -)) ⇒ *a*)
lemma *head-correct*:
assumes *ft-invar t t* ≠ *Empty*
shows *head t* = *hd* (*toList t*)
⟨*proof*⟩

All but the leftmost element

definition *tail*
:: (′*e*,′*a*::*monoid-add*) *FingerTreeStruc* ⇒ (′*e*,′*a*) *FingerTreeStruc*
where
tail t = (*case viewL t of* (*Some* (-, *m*)) ⇒ *m*)
lemma *tail-correct*:
assumes *ft-invar t t* ≠ *Empty*
shows *toList* (*tail t*) = *tl* (*toList t*) **and** *ft-invar* (*tail t*)
⟨*proof*⟩

Rightmost element

definition *headR* :: (′*e*,′*a*::*monoid-add*) *FingerTreeStruc* ⇒ ′*e* × ′*a* **where**
headR t = (*case viewR t of* (*Some* (*a*, -)) ⇒ *a*)
lemma *headR-correct*:
assumes *ft-invar t t* ≠ *Empty*
shows *headR t* = *last* (*toList t*)
⟨*proof*⟩

All but the rightmost element

definition *tailR*
:: (′*e*,′*a*::*monoid-add*) *FingerTreeStruc* ⇒ (′*e*,′*a*) *FingerTreeStruc*
where
tailR t = (*case viewR t of* (*Some* (-, *m*)) ⇒ *m*)
lemma *tailR-correct*:
assumes *ft-invar t t* ≠ *Empty*
shows *toList* (*tailR t*) = *butlast* (*toList t*) **and** *ft-invar* (*tailR t*)
⟨*proof*⟩

1.2.6 Concatenation

primrec *lconsNlist* :: (′*e*,′*a*::*monoid-add*) *Node list*
⇒ (′*e*,′*a*) *FingerTreeStruc* ⇒ (′*e*,′*a*) *FingerTreeStruc* **where**
lconsNlist [] *t* = *t* |
lconsNlist (*x*#*xs*) *t* = *nlcons x* (*lconsNlist xs t*)
primrec *rconsNlist* :: (′*e*,′*a*::*monoid-add*) *FingerTreeStruc*
⇒ (′*e*,′*a*) *Node list* ⇒ (′*e*,′*a*) *FingerTreeStruc* **where**
rconsNlist t [] = *t* |
rconsNlist t (*x*#*xs*) = *rconsNlist* (*nrcons t x*) *xs*

fun *nodes* :: (*'e*,*'a*::*monoid-add*) *Node list* ⇒ (*'e*,*'a*) *Node list* **where**
  nodes [*a*, *b*] = [*node2 a b*] |
  nodes [*a*, *b*, *c*] = [*node3 a b c*] |
  nodes [*a*,*b*,*c*,*d*] = [*node2 a b*, *node2 c d*] |
  nodes (*a*#*b*#*c*#*xs*) = (*node3 a b c*) # (*nodes xs*)

Recursively we concatenate two FingerTreeStrucs while we keep the inner Nodes in a list

fun *app3* :: (*'e*,*'a*::*monoid-add*) *FingerTreeStruc* ⇒ (*'e*,*'a*) *Node list*
    ⇒ (*'e*,*'a*) *FingerTreeStruc* ⇒ (*'e*,*'a*) *FingerTreeStruc* **where**
  app3 Empty xs t = *lconsNlist xs t* |
  app3 t xs Empty = *rconsNlist t xs* |
  app3 (*Single x*) *xs t* = *nlcons x* (*lconsNlist xs t*) |
  app3 t xs (*Single x*) = *nrcons* (*rconsNlist t xs*) *x* |
  app3 (*Deep - pr1 m1 sf1*) *ts* (*Deep - pr2 m2 sf2*) =
    deep pr1 (*app3 m1*
      (*nodes* ((*digitToNlist sf1*) @ *ts* @ (*digitToNlist pr2*))) *m2*) *sf2*

lemma *lconsNlist-inv*:
  assumes *is-leveln-ftree n t*
  and *is-measured-ftree t*
  and ∀ *x*∈*set xs*. (*is-leveln-node n x* ∧ *is-measured-node x*)
  shows
  is-leveln-ftree n (*lconsNlist xs t*) ∧ *is-measured-ftree* (*lconsNlist xs t*)
  ⟨*proof*⟩

lemma *rconsNlist-inv*:
  assumes *is-leveln-ftree n t*
  and *is-measured-ftree t*
  and ∀ *x*∈*set xs*. (*is-leveln-node n x* ∧ *is-measured-node x*)
  shows
  is-leveln-ftree n (*rconsNlist t xs*) ∧ *is-measured-ftree* (*rconsNlist t xs*)
  ⟨*proof*⟩

lemma *nodes-inv*:
  assumes ∀ *x* ∈ *set ts*. *is-leveln-node n x* ∧ *is-measured-node x*
  and *length ts* ≥ *2*
  shows ∀ *x* ∈ *set* (*nodes ts*). *is-leveln-node* (*Suc n*) *x* ∧ *is-measured-node x*
⟨*proof*⟩

lemma *nodes-inv2*:
  assumes *is-leveln-digit n sf1*
  and *is-measured-digit sf1*
  and *is-leveln-digit n pr2*
  and *is-measured-digit pr2*
  and ∀ *x* ∈ *set ts*. *is-leveln-node n x* ∧ *is-measured-node x*
  shows
  ∀*x*∈*set* (*nodes* (*digitToNlist sf1* @ *ts* @ *digitToNlist pr2*)).
        is-leveln-node (*Suc n*) *x* ∧ *is-measured-node x*

⟨*proof*⟩

lemma *app3-inv*:
  assumes *is-leveln-ftree n t1*
  and *is-leveln-ftree n t2*
  and *is-measured-ftree t1*
  and *is-measured-ftree t2*
  and $\forall$ *x*$\in$*set xs*. (*is-leveln-node n x* $\wedge$ *is-measured-node x*)
  shows *is-leveln-ftree n* (*app3 t1 xs t2*) $\wedge$ *is-measured-ftree* (*app3 t1 xs t2*)
⟨*proof*⟩

primrec *nlistToList*:: ((*'e*, *'a*) *Node*) *list* $\Rightarrow$ (*'e* $\times$ *'a*) *list* **where**
  nlistToList [] = []|
  nlistToList (*x#xs*) = (*nodeToList x*) @ (*nlistToList xs*)

lemma *nodes-list*: *length xs* $\geq$ *2* $\Longrightarrow$ *nlistToList* (*nodes xs*) = *nlistToList xs*
  ⟨*proof*⟩

lemma *nlistToList-app*:
  nlistToList (*xs*@*ys*) = (*nlistToList xs*) @ (*nlistToList ys*)
  ⟨*proof*⟩

lemma *nlistListLCons*: *toList* (*lconsNlist xs t*) = (*nlistToList xs*) @ (*toList t*)
  ⟨*proof*⟩

lemma *nlistListRCons*: *toList* (*rconsNlist t xs*) = (*toList t*) @ (*nlistToList xs*)
  ⟨*proof*⟩

lemma *app3-list-lem1*:
  nlistToList (*nodes* (*digitToNlist sf1* @ *ts* @ *digitToNlist pr2*)) =
      digitToList sf1 @ *nlistToList ts* @ *digitToList pr2*
⟨*proof*⟩

lemma *app3-list*:
  toList (*app3 t1 xs t2*) = (*toList t1*) @ (*nlistToList xs*) @ (*toList t2*)
  ⟨*proof*⟩

definition *app*
  :: (*'e*,*'a*::*monoid-add*) *FingerTreeStruc* $\Rightarrow$ (*'e*,*'a*) *FingerTreeStruc*
      $\Rightarrow$ (*'e*,*'a*) *FingerTreeStruc*
  where *app t1 t2* = *app3 t1* [] *t2*

lemma *app-correct*:
  assumes *ft-invar t1 ft-invar t2*
  shows *toList* (*app t1 t2*) = (*toList t1*) @ (*toList t2*)
    and *ft-invar* (*app t1 t2*)
  ⟨*proof*⟩

lemma *app-inv*: ⟦*ft-invar t1*;*ft-invar t2*⟧ ⟹ *ft-invar* (*app t1 t2*)
  ⟨*proof*⟩

lemma *app-list*[*simp*]: *toList* (*app t1 t2*) = (*toList t1*) @ (*toList t2*)
  ⟨*proof*⟩

1.2.7 Splitting

```
type-synonym ('e,'a) SplitDigit =
  ('e,'a) Node list  × ('e,'a) Node × ('e,'a) Node list
type-synonym ('e,'a) SplitTree  =
  ('e,'a) FingerTreeStruc × ('e,'a) Node × ('e,'a) FingerTreeStruc
```

Auxiliary functions to create a correct finger tree even if the left or right digit is empty

```
fun deepL :: ('e,'a::monoid-add) Node list ⇒ ('e,'a) FingerTreeStruc
    ⇒ ('e,'a) Digit ⇒ ('e,'a) FingerTreeStruc where
  deepL [] m sf = (case (viewLn m) of None ⇒ digitToTree sf |
                              (Some (a, m2)) ⇒ deep (nodeToDigit a) m2 sf) |
  deepL pr m sf = deep (nlistToDigit pr) m sf
fun deepR :: ('e,'a::monoid-add) Digit ⇒ ('e,'a) FingerTreeStruc
    ⇒ ('e,'a) Node list ⇒ ('e,'a) FingerTreeStruc where
  deepR pr m [] = (case (viewRn m) of None ⇒ digitToTree pr |
                              (Some (a, m2)) ⇒ deep pr m2 (nodeToDigit a)) |
  deepR pr m sf = deep pr m (nlistToDigit sf)
```

Splitting a list of nodes

```
fun splitNlist :: ('a::monoid-add ⇒ bool) ⇒ 'a ⇒ ('e,'a) Node list
    ⇒ ('e,'a) SplitDigit where
  splitNlist p i [a]    = ([],a,[]) |
  splitNlist p i (a#b) =
    (let i2 = (i + gmn a) in
      (if (p i2)
        then ([],a,b)
        else
         (let (l,x,r) = (splitNlist p i2 b) in ((a#l),x,r))))
```

Splitting a digit by converting it into a list of nodes

```
definition splitDigit :: ('a::monoid-add ⇒ bool) ⇒ 'a ⇒ ('e,'a) Digit
    ⇒ ('e,'a) SplitDigit where
  splitDigit p i d = splitNlist p i (digitToNlist d)
```

Creating a finger tree from list of nodes

```
definition nlistToTree :: ('e,'a::monoid-add) Node list
    ⇒ ('e,'a) FingerTreeStruc where
  nlistToTree xs = lconsNlist xs Empty
```

Recursive splitting into a left and right tree and a center node

```
fun nsplitTree :: ('a::monoid-add ⇒ bool) ⇒ 'a ⇒ ('e,'a) FingerTreeStruc
    ⇒ ('e,'a) SplitTree where
  nsplitTree p i Empty = (Empty, Tip undefined undefined, Empty)
      — Making the function total |
  nsplitTree p i (Single ea) = (Empty,ea,Empty) |
  nsplitTree p i (Deep - pr m sf) =
     (let
        vpr = (i + gmd pr);
        vm = (vpr + gmft m)
      in
        if (p vpr) then
          (let (l,x,r) = (splitDigit p i pr) in
            (nlistToTree l,x,deepL r m sf))
        else (if (p vm) then
          (let (ml,xs,mr) = (nsplitTree p vpr m);
            (l,x,r) = (splitDigit p (vpr + gmft ml) (nodeToDigit xs)) in
              (deepR pr ml l,x,deepL r mr sf))
        else
          (let (l,x,r) = (splitDigit p vm sf) in
            (deepR pr m l,x,nlistToTree r))
      ))
```

lemma *nlistToTree-inv*:
  $\forall\ x \in set\ nl.\ is\text{-}measured\text{-}node\ x \Longrightarrow is\text{-}measured\text{-}ftree\ (nlistToTree\ nl)$
  $\forall\ x \in set\ nl.\ is\text{-}leveln\text{-}node\ n\ x \Longrightarrow is\text{-}leveln\text{-}ftree\ n\ (nlistToTree\ nl)$
  ⟨*proof*⟩

lemma *nlistToTree-list*: *toList* (*nlistToTree nl*) = *nlistToList nl*
  ⟨*proof*⟩

lemma *deepL-inv*:
  assumes *is-leveln-ftree* (*Suc n*) *m* ∧ *is-measured-ftree m*
  and *is-leveln-digit n sf* ∧ *is-measured-digit sf*
  and ∀ *x* ∈ *set pr*. (*is-measured-node x* ∧ *is-leveln-node n x*) ∧ *length pr* ≤ *4*
  shows *is-leveln-ftree n* (*deepL pr m sf*) ∧ *is-measured-ftree* (*deepL pr m sf*)
  ⟨*proof*⟩

lemma *nlistToDigit-list*:
  assumes *1* ≤ *length xs* ∧ *length xs* ≤ *4*
  shows *digitToList*(*nlistToDigit xs*) = *nlistToList xs*
  ⟨*proof*⟩

lemma *deepL-list*:
  assumes *is-leveln-ftree* (*Suc n*) *m* ∧ *is-measured-ftree m*
  and *is-leveln-digit n sf* ∧ *is-measured-digit sf*
  and ∀ *x* ∈ *set pr*. (*is-measured-node x* ∧ *is-leveln-node n x*) ∧ *length pr* ≤ *4*

shows *toList* (*deepL pr m sf*) = *nlistToList pr* @ *toList m* @ *digitToList sf*
⟨*proof*⟩

lemma *deepR-inv*:
  assumes *is-leveln-ftree* (*Suc n*) *m* ∧ *is-measured-ftree m*
  and *is-leveln-digit n pr* ∧ *is-measured-digit pr*
  and ∀ *x* ∈ *set sf*. (*is-measured-node x* ∧ *is-leveln-node n x*) ∧ *length sf* ≤ *4*
  shows *is-leveln-ftree n* (*deepR pr m sf*) ∧ *is-measured-ftree* (*deepR pr m sf*)
  ⟨*proof*⟩

lemma *deepR-list*:
  assumes *is-leveln-ftree* (*Suc n*) *m* ∧ *is-measured-ftree m*
  and *is-leveln-digit n pr* ∧ *is-measured-digit pr*
  and ∀ *x* ∈ *set sf*. (*is-measured-node x* ∧ *is-leveln-node n x*) ∧ *length sf* ≤ *4*
  shows *toList* (*deepR pr m sf*) = *digitToList pr* @ *toList m* @ *nlistToList sf*
⟨*proof*⟩

primrec *gmnl*:: (*'e*, *'a*::*monoid-add*) *Node list* ⇒ *'a* **where**
gmnl [] = *0*|
gmnl (*x#xs*) = *gmn x* + *gmnl xs*

lemma *gmnl-correct*:
  assumes ∀ *x* ∈ *set xs*. *is-measured-node x*
  shows *gmnl xs* = *sum-list* (*map snd* (*nlistToList xs*))
  ⟨*proof*⟩

lemma *splitNlist-correct*: ⟦
  ⋀(*a*::*'a*) (*b*::*'a*). *p a* ⟹ *p* (*a* + *b*);
  ¬ *p i*;
  p (*i* + *gmnl* (*nl* ::(*'e*,*'a*::*monoid-add*) *Node list*));
  splitNlist p i nl = (*l*, *n*, *r*)
  ⟧ ⟹
  ¬ *p* (*i* + (*gmnl l*))
  ∧
  p (*i* + (*gmnl l*) + (*gmn n*))
  ∧
  nl = *l* @ *n* # *r*

⟨*proof*⟩

lemma *digitToNlist-inv*:
  is-measured-digit d ⟹ (∀ *x* ∈ *set* (*digitToNlist d*). *is-measured-node x*)
  is-leveln-digit n d ⟹ (∀ *x* ∈ *set* (*digitToNlist d*). *is-leveln-node n x*)
  ⟨*proof*⟩

lemma *gmnl-gmd*:
  is-measured-digit d ⟹ *gmnl* (*digitToNlist d*) = *gmd d*
  ⟨*proof*⟩

lemma *gmn-gmd*:
  $is\text{-}measured\text{-}node\ nd \Longrightarrow gmd\ (nodeToDigit\ nd) = gmn\ nd$
  ⟨*proof*⟩

lemma *splitDigit-inv*:
  $[\![$
  $\bigwedge (a{::}'a)\ (b{::}'a).\ p\ a \Longrightarrow p\ (a + b);$
  $\neg\ p\ i;$
  $is\text{-}measured\text{-}digit\ d;$
  $is\text{-}leveln\text{-}digit\ n\ d;$
  $p\ (i + gmd\ (d\ {::}('e,'a{::}monoid\text{-}add)\ Digit));$
  $splitDigit\ p\ i\ d = (l,\ nd,\ r)$
  $]\!] \Longrightarrow$
  $\neg\ p\ (i + (gmnl\ l))$
  $\wedge$
  $p\ (i + (gmnl\ l) + (gmn\ nd))$
  $\wedge$
  $(\forall\ x \in set\ l.\ (is\text{-}measured\text{-}node\ x \wedge is\text{-}leveln\text{-}node\ n\ x))$
  $\wedge$
  $(\forall\ x \in set\ r.\ (is\text{-}measured\text{-}node\ x \wedge is\text{-}leveln\text{-}node\ n\ x))$
  $\wedge$
  $(is\text{-}measured\text{-}node\ nd \wedge is\text{-}leveln\text{-}node\ n\ nd\ )$

⟨*proof*⟩

lemma *splitDigit-inv′*:
  $[\![$
  $splitDigit\ p\ i\ d = (l,\ nd,\ r);$
  $is\text{-}measured\text{-}digit\ d;$
  $is\text{-}leveln\text{-}digit\ n\ d$
  $]\!] \Longrightarrow$
  $(\forall\ x \in set\ l.\ (is\text{-}measured\text{-}node\ x \wedge is\text{-}leveln\text{-}node\ n\ x))$
  $\wedge$
  $(\forall\ x \in set\ r.\ (is\text{-}measured\text{-}node\ x \wedge is\text{-}leveln\text{-}node\ n\ x))$
  $\wedge$
  $(is\text{-}measured\text{-}node\ nd \wedge is\text{-}leveln\text{-}node\ n\ nd\ )$

  ⟨*proof*⟩

lemma *splitDigit-list*: $splitDigit\ p\ i\ d = (l,n,r) \Longrightarrow$
  $(digitToList\ d) = (nlistToList\ l)\ @\ (nodeToList\ n)\ @\ (nlistToList\ r)$
  $\wedge\ length\ l \leq 4\ \wedge\ length\ r \leq 4$
  ⟨*proof*⟩

lemma *gmnl-gmft*: $\forall\ x \in set\ nl.\ is\text{-}measured\text{-}node\ x \Longrightarrow$
  $gmft\ (nlistToTree\ nl) = gmnl\ nl$

⟨*proof*⟩

lemma *gmftR-gmnl*:
  assumes *is-leveln-ftree* (*Suc n*) *m* ∧ *is-measured-ftree m*
  and *is-leveln-digit n pr* ∧ *is-measured-digit pr*
  and ∀ *x* ∈ *set sf*. (*is-measured-node x* ∧ *is-leveln-node n x*) ∧ *length sf* ≤ *4*
  shows *gmft* (*deepR pr m sf*) = *gmd pr* + *gmft m* + *gmnl sf*
⟨*proof*⟩

lemma *nsplitTree-invpres*: ⟦
  is-leveln-ftree n (*s*:: (′*e*,′*a*::*monoid-add*) *FingerTreeStruc*);
  is-measured-ftree s;
  ¬ *p i*;
  p (*i* + (*gmft s*));
  (*nsplitTree p i s*) = (*l*, *nd*, *r*)⟧
  ⟹
  is-leveln-ftree n l
  ∧
  is-measured-ftree l
  ∧
  is-leveln-ftree n r
  ∧
  is-measured-ftree r
  ∧
  is-leveln-node n nd
  ∧
  is-measured-node nd

⟨*proof*⟩

lemma *nsplitTree-correct*: ⟦
  is-leveln-ftree n (*s*:: (′*e*,′*a*::*monoid-add*) *FingerTreeStruc*);
  is-measured-ftree s;
  ⋀(*a*::′*a*) (*b*::′*a*). *p a* ⟹ *p* (*a* + *b*);
  ¬ *p i*;
  p (*i* + (*gmft s*));
  (*nsplitTree p i s*) = (*l*, *nd*, *r*)⟧
  ⟹ (*toList s*) = (*toList l*) @ (*nodeToList nd*) @ (*toList r*)
  ∧
  ¬ *p* (*i* + (*gmft l*))
  ∧
  p (*i* + (*gmft l*) + (*gmn nd*))
  ∧
  is-leveln-ftree n l
  ∧
  is-measured-ftree l
  ∧
  is-leveln-ftree n r
  ∧

is-measured-ftree r
$\wedge$
is-leveln-node n nd
$\wedge$
is-measured-node nd

⟨*proof*⟩

A predicate on the elements of a monoid is called *monotone*, iff, when it holds for some value a, it also holds for all values $a + b$:

Split a finger tree by a monotone predicate on the annotations, using a given initial value. Intuitively, the elements are summed up from left to right, and the split is done when the predicate first holds for the sum. The predicate must not hold for the initial value of the summation, and must hold for the sum of all elements.

definition *splitTree*
  :: (*'a*::*monoid-add* ⇒ *bool*) ⇒ *'a* ⇒ (*'e*, *'a*) *FingerTreeStruc*
    ⇒ (*'e*, *'a*) *FingerTreeStruc* × (*'e* × *'a*) × (*'e*, *'a*) *FingerTreeStruc*
  where
  splitTree p i t = (*let* (*l*, *x*, *r*) = *nsplitTree p i t in* (*l*, (*n-unwrap x*), *r*))

lemma *splitTree-invpres*:
  assumes *inv*: *ft-invar* (*s*:: (*'e*,*'a*::*monoid-add*) *FingerTreeStruc*)
  assumes *init-ff*: ¬ *p i*
  assumes *sum-tt*: *p* (*i* + *annot s*)
  assumes *fmt*: (*splitTree p i s*) = (*l*, (*e*,*a*), *r*)
  shows *ft-invar l* **and** *ft-invar r*
⟨*proof*⟩

lemma *splitTree-correct*:
  assumes *inv*: *ft-invar* (*s*:: (*'e*,*'a*::*monoid-add*) *FingerTreeStruc*)
  assumes *mono*: ∀ *a b*. *p a* ⟶ *p* (*a* + *b*)
  assumes *init-ff*: ¬ *p i*
  assumes *sum-tt*: *p* (*i* + *annot s*)
  assumes *fmt*: (*splitTree p i s*) = (*l*, (*e*,*a*), *r*)
  shows (*toList s*) = (*toList l*) @ (*e*,*a*) # (*toList r*)
  and   ¬ *p* (*i* + *annot l*)
  and   *p* (*i* + *annot l* + *a*)
  and   *ft-invar l* **and** *ft-invar r*
⟨*proof*⟩

lemma *splitTree-correctE*:
  assumes *inv*: *ft-invar* (*s*:: (*'e*,*'a*::*monoid-add*) *FingerTreeStruc*)
  assumes *mono*: ∀ *a b*. *p a* ⟶ *p* (*a* + *b*)
  assumes *init-ff*: ¬ *p i*
  assumes *sum-tt*: *p* (*i* + *annot s*)
  obtains *l e a r* **where**

$(splitTree\ p\ i\ s) = (l,\ (e,a),\ r)$ **and**
$(toList\ s) = (toList\ l)\ @\ (e,a)\ \#\ (toList\ r)$ **and**
$\neg\ p\ (i + annot\ l)$ **and**
$p\ (i + annot\ l + a)$ **and**
ft-invar l **and** *ft-invar r*
⟨*proof*⟩

1.2.8 Folding

fun *foldl-node* :: $('s \Rightarrow 'e \times 'a \Rightarrow 's) \Rightarrow 's \Rightarrow ('e,'a)\ Node \Rightarrow 's$ **where**
foldl-node f σ *(Tip e a) = f* σ *(e,a)*|
foldl-node f σ *(Node2 - a b) = foldl-node f (foldl-node f* σ *a) b*|
foldl-node f σ *(Node3 - a b c) =*
foldl-node f (foldl-node f (foldl-node f σ *a) b) c*

primrec *foldl-digit* :: $('s \Rightarrow 'e \times 'a \Rightarrow 's) \Rightarrow 's \Rightarrow ('e,'a)\ Digit \Rightarrow 's$ **where**
foldl-digit f σ *(One n1) = foldl-node f* σ *n1*|
foldl-digit f σ *(Two n1 n2) = foldl-node f (foldl-node f* σ *n1) n2*|
foldl-digit f σ *(Three n1 n2 n3) =*
foldl-node f (foldl-node f (foldl-node f σ *n1) n2) n3*|
foldl-digit f σ *(Four n1 n2 n3 n4) =*
foldl-node f (foldl-node f (foldl-node f (foldl-node f σ *n1) n2) n3) n4*

primrec *foldr-node* :: $('e \times 'a \Rightarrow 's \Rightarrow 's) \Rightarrow ('e,'a)\ Node \Rightarrow 's \Rightarrow 's$ **where**
foldr-node f (Tip e a) σ *= f (e,a)* σ |
foldr-node f (Node2 - a b) σ *= foldr-node f a (foldr-node f b* σ*)*|
foldr-node f (Node3 - a b c) σ
= foldr-node f a (foldr-node f b (foldr-node f c σ*))*

primrec *foldr-digit* :: $('e \times 'a \Rightarrow 's \Rightarrow 's) \Rightarrow ('e,'a)\ Digit \Rightarrow 's \Rightarrow 's$ **where**
foldr-digit f (One n1) σ *= foldr-node f n1* σ|
foldr-digit f (Two n1 n2) σ *= foldr-node f n1 (foldr-node f n2* σ*)*|
foldr-digit f (Three n1 n2 n3) σ *=*
foldr-node f n1 (foldr-node f n2 (foldr-node f n3 σ*))*|
foldr-digit f (Four n1 n2 n3 n4) σ *=*
foldr-node f n1 (foldr-node f n2 (foldr-node f n3 (foldr-node f n4 σ*)))*

lemma *foldl-node-correct*:
foldl-node f σ *nd = List.foldl f* σ *(nodeToList nd)*
⟨*proof*⟩

lemma *foldl-digit-correct*:
foldl-digit f σ *d = List.foldl f* σ *(digitToList d)*
⟨*proof*⟩

lemma *foldr-node-correct*:
foldr-node f nd σ *= List.foldr f (nodeToList nd)* σ
⟨*proof*⟩

lemma *foldr-digit-correct*:
  foldr-digit f d σ = *List.foldr f* (*digitToList d*) σ
  ⟨*proof*⟩

Fold from left

primrec *foldl* :: ($'s \Rightarrow 'e \times 'a \Rightarrow 's$) $\Rightarrow 's \Rightarrow$ ($'e,'a$) *FingerTreeStruc* $\Rightarrow 's$
  where
  foldl f σ *Empty* = σ|
  foldl f σ (*Single nd*) = *foldl-node f* σ *nd*|
  foldl f σ (*Deep - d1 m d2*) =
    foldl-digit f (*foldl f* (*foldl-digit f* σ *d1*) *m*) *d2*

lemma *foldl-correct*:
  foldl f σ *t* = *List.foldl f* σ (*toList t*)
  ⟨*proof*⟩

Fold from right

primrec *foldr* :: ($'e \times 'a \Rightarrow 's \Rightarrow 's$) $\Rightarrow$ ($'e,'a$) *FingerTreeStruc* $\Rightarrow 's \Rightarrow 's$
  where
  foldr f Empty σ = σ|
  foldr f (*Single nd*) σ = *foldr-node f nd* σ|
  foldr f (*Deep - d1 m d2*) σ
    = *foldr-digit f d1* (*foldr f m*(*foldr-digit f d2* σ))

lemma *foldr-correct*:
  foldr f t σ = *List.foldr f* (*toList t*) σ
  ⟨*proof*⟩

1.2.9 Number of elements

primrec *count-node* :: ($'e$, $'a$) *Node* $\Rightarrow$ *nat* **where**
  count-node (*Tip - a*) = *1* |
  count-node (*Node2 - a b*) = *count-node a* + *count-node b* |
  count-node (*Node3 - a b c*) = *count-node a* + *count-node b* + *count-node c*

primrec *count-digit* :: ($'e,'a$) *Digit* $\Rightarrow$ *nat* **where**
  count-digit (*One a*) = *count-node a* |
  count-digit (*Two a b*) = *count-node a* + *count-node b* |
  count-digit (*Three a b c*) = *count-node a* + *count-node b* + *count-node c* |
  count-digit (*Four a b c d*)
    = *count-node a* + *count-node b* + *count-node c* + *count-node d*

lemma *count-node-correct*:
  count-node n = *length* (*nodeToList n*)
  ⟨*proof*⟩

lemma *count-digit-correct*:
  count-digit d = *length* (*digitToList d*)

⟨*proof*⟩

primrec *count* :: (*'e,'a*) *FingerTreeStruc* ⇒ *nat* **where**
count Empty = *0* |
count (*Single a*) = *count-node a* |
count (*Deep - pr m sf*) = *count-digit pr* + *count m* + *count-digit sf*

lemma *count-correct*[*simp*]:
count t = *length* (*toList t*)
⟨*proof*⟩
end

interpretation *FingerTreeStruc*: *FingerTreeStruc-loc* ⟨*proof*⟩

no-notation *FingerTreeStruc.lcons* (**infixr** ◁ *65*)
no-notation *FingerTreeStruc.rcons* (**infixl** ▷ *65*)

1.3 Hiding the invariant

In this section, we define the datatype of all FingerTrees that fulfill their invariant, and define the operations to work on this datatype. The advantage is, that the correctness lemmas do no longer contain explicit invariant predicates, what makes them more handy to use.

1.3.1 Datatype

typedef (**overloaded**) (*'e*, *'a*) *FingerTree* =
{*t* :: (*'e*, *'a*::*monoid-add*) *FingerTreeStruc*. *FingerTreeStruc.ft-invar t*}
⟨*proof*⟩

lemma *Rep-FingerTree-invar*[*simp*]: *FingerTreeStruc.ft-invar* (*Rep-FingerTree t*)
⟨*proof*⟩

lemma [*simp*]:
FingerTreeStruc.ft-invar t ⟹ *Rep-FingerTree* (*Abs-FingerTree t*) = *t*
⟨*proof*⟩

lemma [*simp*, *code abstype*]: *Abs-FingerTree* (*Rep-FingerTree t*) = *t*
⟨*proof*⟩

typedef (**overloaded**) (*'e,'a*) *viewres* =
{ *r*:: ((*'e* × *'a*) × (*'e,'a*::*monoid-add*) *FingerTreeStruc*) *option* .
case r of None ⇒ *True* | *Some* (*a,t*) ⇒ *FingerTreeStruc.ft-invar t*}
⟨*proof*⟩

lemma [*simp*, *code abstype*]: *Abs-viewres* (*Rep-viewres x*) = *x*
⟨*proof*⟩

```
lemma Abs-viewres-inverse-None[simp]:
  Rep-viewres (Abs-viewres None) = None
  ⟨proof⟩

lemma Abs-viewres-inverse-Some:
  FingerTreeStruc.ft-invar t ⟹
    Rep-viewres (Abs-viewres (Some (a,t))) = Some (a,t)
  ⟨proof⟩

definition [code]: extract-viewres-isNone r == Rep-viewres r = None
definition [code]: extract-viewres-a r ==
    case (Rep-viewres r) of Some (a,t) ⇒ a
definition extract-viewres-t r ==
  case (Rep-viewres r) of None ⇒ Abs-FingerTree Empty
                        | Some (a,t) ⇒ Abs-FingerTree t
lemma [code abstract]: Rep-FingerTree (extract-viewres-t r) =
    (case (Rep-viewres r) of None ⇒ Empty | Some (a,t) ⇒ t)
  ⟨proof⟩

definition extract-viewres r ==
    if extract-viewres-isNone r then None
    else Some (extract-viewres-a r, extract-viewres-t r)

typedef (overloaded) ('e,'a) splitres =
 { ((l,a,r):: (('e,'a) FingerTreeStruc × ('e × 'a) × ('e,'a::monoid-add) FingerTreeStruc))
   | l a r.
       FingerTreeStruc.ft-invar l ∧ FingerTreeStruc.ft-invar r}
  ⟨proof⟩

lemma [simp, code abstype]: Abs-splitres (Rep-splitres x) = x
  ⟨proof⟩

lemma Abs-splitres-inverse:
  FingerTreeStruc.ft-invar r ⟹ FingerTreeStruc.ft-invar s ⟹
      Rep-splitres (Abs-splitres ((r,a,s))) = (r,a,s)
  ⟨proof⟩

definition [code]: extract-splitres-a r == case (Rep-splitres r) of (l,a,s) ⇒ a
definition extract-splitres-l r == case (Rep-splitres r) of (l,a,r) ⇒
    Abs-FingerTree l
lemma [code abstract]: Rep-FingerTree (extract-splitres-l r) = (case
    (Rep-splitres r) of (l,a,r) ⇒ l)
  ⟨proof⟩
definition extract-splitres-r r == case (Rep-splitres r) of (l,a,r) ⇒
    Abs-FingerTree r
lemma [code abstract]: Rep-FingerTree (extract-splitres-r r) = (case
  (Rep-splitres r) of (l,a,r) ⇒ r)
  ⟨proof⟩
```

definition *extract-splitres r* ==
  (*extract-splitres-l r*,
  extract-splitres-a r,
  extract-splitres-r r)

1.3.2 Definition of Operations

locale *FingerTree-loc*
begin
  definition [*code*]: *toList t* == *FingerTreeStruc.toList* (*Rep-FingerTree t*)
  definition *empty* **where** *empty* == *Abs-FingerTree FingerTreeStruc.Empty*
  lemma [*code abstract*]: *Rep-FingerTree empty* = *FingerTreeStruc.Empty*
    ⟨*proof*⟩

  lemma *empty-rep*: *t*=*empty* ⟷ *Rep-FingerTree t* = *Empty*
    ⟨*proof*⟩

  definition [*code*]: *annot t* == *FingerTreeStruc.annot* (*Rep-FingerTree t*)
  definition *toTree t* == *Abs-FingerTree* (*FingerTreeStruc.toTree t*)
  lemma [*code abstract*]: *Rep-FingerTree* (*toTree t*) = *FingerTreeStruc.toTree t*
    ⟨*proof*⟩
  definition *lcons a t* ==
    Abs-FingerTree (*FingerTreeStruc.lcons a* (*Rep-FingerTree t*))
  lemma [*code abstract*]:
    Rep-FingerTree (*lcons a t*) = (*FingerTreeStruc.lcons a* (*Rep-FingerTree t*))
    ⟨*proof*⟩
  definition *rcons t a* ==
    Abs-FingerTree (*FingerTreeStruc.rcons* (*Rep-FingerTree t*) *a*)
  lemma [*code abstract*]:
    Rep-FingerTree (*rcons t a*) = (*FingerTreeStruc.rcons* (*Rep-FingerTree t*) *a*)
    ⟨*proof*⟩

  definition *viewL-aux t* ==
    Abs-viewres (*FingerTreeStruc.viewL* (*Rep-FingerTree t*))
  definition *viewL t* == *extract-viewres* (*viewL-aux t*)
  lemma [*code abstract*]:
    Rep-viewres (*viewL-aux t*) = (*FingerTreeStruc.viewL* (*Rep-FingerTree t*))
    ⟨*proof*⟩

  definition *viewR-aux t* ==
    Abs-viewres (*FingerTreeStruc.viewR* (*Rep-FingerTree t*))
  definition *viewR t* == *extract-viewres* (*viewR-aux t*)
  lemma [*code abstract*]:
    Rep-viewres (*viewR-aux t*) = (*FingerTreeStruc.viewR* (*Rep-FingerTree t*))
    ⟨*proof*⟩

  definition [*code*]: *isEmpty t* == *FingerTreeStruc.isEmpty* (*Rep-FingerTree t*)

definition [*code*]: *head t* = *FingerTreeStruc.head* (*Rep-FingerTree t*)
definition *tail t* ≡
  if t=empty then
    empty
  else
    Abs-FingerTree (*FingerTreeStruc.tail* (*Rep-FingerTree t*))
  — Make function total, to allow abstraction
lemma [*code abstract*]: *Rep-FingerTree* (*tail t*) =
  (*if* (*FingerTreeStruc.isEmpty* (*Rep-FingerTree t*)) *then Empty*
   else FingerTreeStruc.tail (*Rep-FingerTree t*))
  ⟨*proof*⟩

definition [*code*]: *headR t* = *FingerTreeStruc.headR* (*Rep-FingerTree t*)
definition *tailR t* ≡
  if t=empty then
    empty
  else
    Abs-FingerTree (*FingerTreeStruc.tailR* (*Rep-FingerTree t*))
lemma [*code abstract*]: *Rep-FingerTree* (*tailR t*) =
  (*if* (*FingerTreeStruc.isEmpty* (*Rep-FingerTree t*)) *then Empty*
  else FingerTreeStruc.tailR (*Rep-FingerTree t*))
  ⟨*proof*⟩

definition *app s t* = *Abs-FingerTree* (
  FingerTreeStruc.app (*Rep-FingerTree s*) (*Rep-FingerTree t*))
lemma [*code abstract*]:
  Rep-FingerTree (*app s t*) =
    FingerTreeStruc.app (*Rep-FingerTree s*) (*Rep-FingerTree t*)
  ⟨*proof*⟩

definition *splitTree-aux p i t* == *if* (¬*p i* ∧ *p* (*i*+*annot t*)) *then*
  Abs-splitres (*FingerTreeStruc.splitTree p i* (*Rep-FingerTree t*))
else
  Abs-splitres (*Empty*,*undefined*,*Empty*)
definition *splitTree p i t* == *extract-splitres* (*splitTree-aux p i t*)

lemma [*code abstract*]:
  Rep-splitres (*splitTree-aux p i t*) = (*if* (¬*p i* ∧ *p* (*i*+*annot t*)) *then*
    (*FingerTreeStruc.splitTree p i* (*Rep-FingerTree t*))
  else
    (*Empty*,*undefined*,*Empty*))
  ⟨*proof*⟩

definition *foldl* **where**
  [*code*]: *foldl f σ t* == *FingerTreeStruc.foldl f σ* (*Rep-FingerTree t*)
definition *foldr* **where**
  [*code*]: *foldr f t σ* == *FingerTreeStruc.foldr f* (*Rep-FingerTree t*) *σ*
definition *count* **where**
  [*code*]: *count t* == *FingerTreeStruc.count* (*Rep-FingerTree t*)

1.3.3 Correctness statements

lemma *empty-correct*: $toList\ t = [] \longleftrightarrow t{=}empty$
⟨*proof*⟩

lemma *toList-of-empty*[*simp*]: $toList\ empty = []$
⟨*proof*⟩

lemma *annot-correct*: $annot\ t = sum\text{-}list\ (map\ snd\ (toList\ t))$
⟨*proof*⟩

lemma *toTree-correct*: $toList\ (toTree\ l) = l$
⟨*proof*⟩

lemma *lcons-correct*: $toList\ (lcons\ a\ t) = a\#toList\ t$
⟨*proof*⟩

lemma *rcons-correct*: $toList\ (rcons\ t\ a) = toList\ t@[a]$
⟨*proof*⟩

lemma *viewL-correct*:
$t = empty \Longrightarrow viewL\ t = None$
$t \neq empty \Longrightarrow \exists\, a\ s.\ viewL\ t = Some\ (a,s) \wedge toList\ t = a\#toList\ s$
⟨*proof*⟩

lemma *viewL-empty*[*simp*]: $viewL\ empty = None$
⟨*proof*⟩

lemma *viewL-nonEmpty*:
assumes $t{\neq}empty$
obtains $a\ s$ **where** $viewL\ t = Some\ (a,s)\ toList\ t = a\#toList\ s$
⟨*proof*⟩

lemma *viewR-correct*:
$t = empty \Longrightarrow viewR\ t = None$
$t \neq empty \Longrightarrow \exists\, a\ s.\ viewR\ t = Some\ (a,s) \wedge toList\ t = toList\ s@[a]$
⟨*proof*⟩

lemma *viewR-empty*[*simp*]: $viewR\ empty = None$
⟨*proof*⟩

lemma *viewR-nonEmpty*:
assumes $t{\neq}empty$
obtains $a\ s$ **where** $viewR\ t = Some\ (a,s)\ toList\ t = toList\ s@[a]$
⟨*proof*⟩

lemma *isEmpty-correct*: $isEmpty\ t \longleftrightarrow t{=}empty$
⟨*proof*⟩

lemma *head-correct*: $t{\neq}empty \Longrightarrow head\ t = hd\ (toList\ t)$

⟨*proof*⟩

lemma *tail-correct*: *t*≠*empty* ⟹ *toList* (*tail t*) = *tl* (*toList t*)
⟨*proof*⟩

lemma *headR-correct*: *t*≠*empty* ⟹ *headR t* = *last* (*toList t*)
⟨*proof*⟩

lemma *tailR-correct*: *t*≠*empty* ⟹ *toList* (*tailR t*) = *butlast* (*toList t*)
⟨*proof*⟩

lemma *app-correct*: *toList* (*app s t*) = *toList s* @ *toList t*
⟨*proof*⟩

lemma *splitTree-correct*:
assumes *mono*: ∀ *a b*. *p a* ⟶ *p* (*a* + *b*)
assumes *init-ff*: ¬ *p i*
assumes *sum-tt*: *p* (*i* + *annot s*)
assumes *fmt*: (*splitTree p i s*) = (*l*, (*e*,*a*), *r*)
shows (*toList s*) = (*toList l*) @ (*e*,*a*) # (*toList r*)
and ¬ *p* (*i* + *annot l*)
and *p* (*i* + *annot l* + *a*)
⟨*proof*⟩

lemma *splitTree-correctE*:
assumes *mono*: ∀ *a b*. *p a* ⟶ *p* (*a* + *b*)
assumes *init-ff*: ¬ *p i*
assumes *sum-tt*: *p* (*i* + *annot s*)
obtains *l e a r* **where**
(*splitTree p i s*) = (*l*, (*e*,*a*), *r*) **and**
(*toList s*) = (*toList l*) @ (*e*,*a*) # (*toList r*) **and**
¬ *p* (*i* + *annot l*) **and**
p (*i* + *annot l* + *a*)
⟨*proof*⟩

lemma *foldl-correct*: *foldl f σ t* = *List.foldl f σ* (*toList t*)
⟨*proof*⟩

lemma *foldr-correct*: *foldr f t σ* = *List.foldr f* (*toList t*) *σ*
⟨*proof*⟩

lemma *count-correct*: *count t* = *length* (*toList t*)
⟨*proof*⟩

end

interpretation *FingerTree*: *FingerTree-loc* ⟨*proof*⟩

1.4 Interface Documentation

In this section, we list all supported operations on finger trees, along with a short plaintext documentation and their correctness statements.

$\underline{\textit{FingerTree.toList}::(\textit{'a}, \textit{'b})\ \textit{FingerTree} \Rightarrow (\textit{'a} \times \textit{'b})\ \textit{list}}$
Convert to list ($O(n)$)

$\underline{\textit{FingerTree.empty}::(\textit{'a}, \textit{'b})\ \textit{FingerTree}}$
The empty finger tree ($O(1)$)
Spec *FingerTree.empty-correct*:

$(\textit{FingerTree.toList}\ ?t = []) = (?t = \textit{FingerTree.empty})$

$\underline{\textit{FingerTree.annot}::(\textit{'a}, \textit{'b})\ \textit{FingerTree} \Rightarrow \textit{'b}}$
Return sum of all annotations ($O(1)$)
Spec *FingerTree.annot-correct*:

$\textit{FingerTree.annot}\ ?t = \textit{sum-list}\ (\textit{map snd}\ (\textit{FingerTree.toList}\ ?t))$

$\underline{\textit{FingerTree.toTree}::(\textit{'a} \times \textit{'b})\ \textit{list} \Rightarrow (\textit{'a}, \textit{'b})\ \textit{FingerTree}}$
Convert list to finger tree ($O(n \log(n))$)
Spec *FingerTree.toTree-correct*:

$\textit{FingerTree.toList}\ (\textit{FingerTree.toTree}\ ?l) = ?l$

$\underline{\textit{FingerTree.lcons}::\textit{'a} \times \textit{'b} \Rightarrow (\textit{'a}, \textit{'b})\ \textit{FingerTree} \Rightarrow (\textit{'a}, \textit{'b})\ \textit{FingerTree}}$
Append element at the left end ($O(\log(n))$, $O(1)$ amortized)
Spec *FingerTree.lcons-correct*:

$\textit{FingerTree.toList}\ (\textit{FingerTree.lcons}\ ?a\ ?t) = ?a\ \#\ \textit{FingerTree.toList}\ ?t$

$\underline{\textit{FingerTree.rcons}::(\textit{'a}, \textit{'b})\ \textit{FingerTree} \Rightarrow \textit{'a} \times \textit{'b} \Rightarrow (\textit{'a}, \textit{'b})\ \textit{FingerTree}}$
Append element at the right end ($O(\log(n))$, $O(1)$ amortized)
Spec *FingerTree.rcons-correct*:

$\textit{FingerTree.toList}\ (\textit{FingerTree.rcons}\ ?t\ ?a) = \textit{FingerTree.toList}\ ?t\ @\ [?a]$

$\underline{\textit{FingerTree.viewL}::(\textit{'a}, \textit{'b})\ \textit{FingerTree} \Rightarrow ((\textit{'a} \times \textit{'b}) \times (\textit{'a}, \textit{'b})\ \textit{FingerTree})\ \textit{option}}$
Detach leftmost element ($O(\log(n))$, $O(1)$ amortized)
Spec *FingerTree.viewL-correct*:

$?t = \textit{FingerTree.empty} \Longrightarrow \textit{FingerTree.viewL}\ ?t = \textit{None}$
$?t \neq \textit{FingerTree.empty} \Longrightarrow$
$\exists\, a\ s.\ \textit{FingerTree.viewL}\ ?t = \textit{Some}\ (a, s) \wedge$
$\quad \textit{FingerTree.toList}\ ?t = a\ \#\ \textit{FingerTree.toList}\ s$

$\underline{\textit{FingerTree.viewR}::(\textit{'a}, \textit{'b})\ \textit{FingerTree} \Rightarrow ((\textit{'a} \times \textit{'b}) \times (\textit{'a}, \textit{'b})\ \textit{FingerTree})\ \textit{option}}$
Detach rightmost element ($O(\log(n))$, $O(1)$ amortized)
Spec *FingerTree.viewR-correct*:

$?t = \mathit{FingerTree.empty} \Longrightarrow \mathit{FingerTree.viewR}\ ?t = \mathit{None}$
$?t \neq \mathit{FingerTree.empty} \Longrightarrow$
$\exists\, a\ s.\ \mathit{FingerTree.viewR}\ ?t = \mathit{Some}\ (a,\ s) \wedge$
$\quad \mathit{FingerTree.toList}\ ?t = \mathit{FingerTree.toList}\ s\ @\ [a]$

$\underline{\mathit{FingerTree.isEmpty}{::}('a,\ 'b)\ \mathit{FingerTree} \Rightarrow \mathit{bool}}$
Check whether tree is empty ($O(1)$)
Spec *FingerTree.isEmpty-correct*:

$\mathit{FingerTree.isEmpty}\ ?t = (?t = \mathit{FingerTree.empty})$

$\underline{\mathit{FingerTree.head}{::}('a,\ 'b)\ \mathit{FingerTree} \Rightarrow 'a \times 'b}$
Get leftmost element of non-empty tree ($O(\log(n))$)
Spec *FingerTree.head-correct*:

$?t \neq \mathit{FingerTree.empty} \Longrightarrow \mathit{FingerTree.head}\ ?t = \mathit{hd}\ (\mathit{FingerTree.toList}\ ?t)$

$\underline{\mathit{FingerTree.tail}{::}('a,\ 'b)\ \mathit{FingerTree} \Rightarrow ('a,\ 'b)\ \mathit{FingerTree}}$
Get all but leftmost element of non-empty tree ($O(\log(n))$)
Spec *FingerTree.tail-correct*:

$?t \neq \mathit{FingerTree.empty} \Longrightarrow$
$\mathit{FingerTree.toList}\ (\mathit{FingerTree.tail}\ ?t) = \mathit{tl}\ (\mathit{FingerTree.toList}\ ?t)$

$\underline{\mathit{FingerTree.headR}{::}('a,\ 'b)\ \mathit{FingerTree} \Rightarrow 'a \times 'b}$
Get rightmost element of non-empty tree ($O(\log(n))$)
Spec *FingerTree.headR-correct*:

$?t \neq \mathit{FingerTree.empty} \Longrightarrow \mathit{FingerTree.headR}\ ?t = \mathit{last}\ (\mathit{FingerTree.toList}\ ?t)$

$\underline{\mathit{FingerTree.tailR}{::}('a,\ 'b)\ \mathit{FingerTree} \Rightarrow ('a,\ 'b)\ \mathit{FingerTree}}$
Get all but rightmost element of non-empty tree ($O(\log(n))$)
Spec *FingerTree.tailR-correct*:

$?t \neq \mathit{FingerTree.empty} \Longrightarrow$
$\mathit{FingerTree.toList}\ (\mathit{FingerTree.tailR}\ ?t) = \mathit{butlast}\ (\mathit{FingerTree.toList}\ ?t)$

$\underline{\mathit{FingerTree.app}{::}('a,\ 'b)\ \mathit{FingerTree} \Rightarrow ('a,\ 'b)\ \mathit{FingerTree} \Rightarrow ('a,\ 'b)\ \mathit{FingerTree}}$
Concatenate two finger trees ($O(\log(m + n))$)
Spec *FingerTree.app-correct*:

$\mathit{FingerTree.toList}\ (\mathit{FingerTree.app}\ ?s\ ?t) =$
$\mathit{FingerTree.toList}\ ?s\ @\ \mathit{FingerTree.toList}\ ?t$

$\underline{\mathit{FingerTree.splitTree}}$

$\mathit{FingerTree.splitTree}{::}('a \Rightarrow \mathit{bool})$
$\qquad \Rightarrow 'a \Rightarrow ('b,\ 'a)\ \mathit{FingerTree}$
$\qquad\qquad \Rightarrow ('b,\ 'a)\ \mathit{FingerTree} \times$
$\qquad\qquad\quad ('b \times 'a) \times ('b,\ 'a)\ \mathit{FingerTree}$

Split tree by a monotone predicate. ($O(\log(n))$)

A predicate p over the annotations is called monotone, iff, for all annotations a, b with $p(a)$, we have already $p(a + b)$.

Splitting is done by specifying a monotone predicate p that does not hold for the initial value i of the summation, but holds for i plus the sum of all annotations. The tree is then split at the position where p starts to hold for the sum of all elements up to that position.
Spec *FingerTree.splitTree-correct*:

$$\begin{array}{l}
[\![\forall\, a\ b.\ ?p\ a \longrightarrow ?p\ (a + b);\ \neg\ ?p\ ?i;\ ?p\ (?i + \mathit{FingerTree.annot}\ ?s);\\
\ \mathit{FingerTree.splitTree}\ ?p\ ?i\ ?s = (?l,\ (?e,\ ?a),\ ?r)]\!]\\
\Longrightarrow \mathit{FingerTree.toList}\ ?s =\\
\quad\ \ \mathit{FingerTree.toList}\ ?l\ @\ (?e,\ ?a)\ \#\ \mathit{FingerTree.toList}\ ?r\\
[\![\forall\, a\ b.\ ?p\ a \longrightarrow ?p\ (a + b);\ \neg\ ?p\ ?i;\ ?p\ (?i + \mathit{FingerTree.annot}\ ?s);\\
\ \mathit{FingerTree.splitTree}\ ?p\ ?i\ ?s = (?l,\ (?e,\ ?a),\ ?r)]\!]\\
\Longrightarrow \neg\ ?p\ (?i + \mathit{FingerTree.annot}\ ?l)\\
[\![\forall\, a\ b.\ ?p\ a \longrightarrow ?p\ (a + b);\ \neg\ ?p\ ?i;\ ?p\ (?i + \mathit{FingerTree.annot}\ ?s);\\
\ \mathit{FingerTree.splitTree}\ ?p\ ?i\ ?s = (?l,\ (?e,\ ?a),\ ?r)]\!]\\
\Longrightarrow\ ?p\ (?i + \mathit{FingerTree.annot}\ ?l + ?a)
\end{array}$$

FingerTree.foldl

$\mathit{FingerTree.foldl}::('a \Rightarrow 'b \times 'c \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('b,\ 'c)\ \mathit{FingerTree} \Rightarrow 'a$

Fold with function from left
Spec *FingerTree.foldl-correct*:

$\mathit{FingerTree.foldl}\ ?f\ ?\sigma\ \ ?t = \mathit{foldl}\ ?f\ ?\sigma\ (\mathit{FingerTree.toList}\ ?t)$

FingerTree.foldr

$\mathit{FingerTree.foldr}::('a \times 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a,\ 'b)\ \mathit{FingerTree} \Rightarrow 'c \Rightarrow 'c$

Fold with function from right
Spec *FingerTree.foldr-correct*:

$\mathit{FingerTree.foldr}\ ?f\ ?t\ ?\sigma = \mathit{foldr}\ ?f\ (\mathit{FingerTree.toList}\ ?t)\ ?\sigma$

$\mathit{FingerTree.count}::('a,\ 'b)\ \mathit{FingerTree} \Rightarrow \mathit{nat}$
Return the number of elements
Spec *FingerTree.count-correct*:

$\mathit{FingerTree.count}\ ?t = \mathit{length}\ (\mathit{FingerTree.toList}\ ?t)$

end

2 Related work

Finger trees were originally introduced by Hinze and Paterson[1], who give an implementation in Haskell. Our implementation closely follows this original implementation.

There is also a machine-checked formalization of 2-3 finger trees in Coq [2]. Like ours, it closely follows the original paper of Hinze and Paterson. The main difference is that the Coq-formalization encodes the invariants directly into the datatype for finger trees, while we first define the bigger algebraic datatype *FingerTreeStruc* along with the predicate *ft-invar* that checks the invariant. This bigger type and the *ft-invar*-predicate is then wrapped into the datatype *FingerTree*, that, however, exposes no algebraic structure any more. Our approach greatly simplifies matters in the context of Isabelle/HOL, as it can be realized with Isabelle's datatype-package.

References

[1] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

[2] M. Sozeau. Program-ing finger trees in coq. In *ICFP '07*, pages 13–24, New York, NY, USA, 2007. ACM.