

FingerTrees

Benedikt Nordhoff Stefan Körner Peter Lammich

February 23, 2021

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	2
1.1	Datatype definition	2
1.1.1	Invariant	3
1.1.2	Abstraction to Lists	4
1.2	Operations	5
1.2.1	Empty tree	5
1.2.2	Annotation	5
1.2.3	Appending	6
1.2.4	Convert list to tree	8
1.2.5	Detaching leftmost/rightmost element	8
1.2.6	Concatenation	12
1.2.7	Splitting	15
1.2.8	Folding	21
1.2.9	Number of elements	22
1.3	Hiding the invariant	23
1.3.1	Datatype	23
1.3.2	Definition of Operations	25
1.3.3	Correctness statements	27
1.4	Interface Documentation	30

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  $\Rightarrow$  '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  $\Rightarrow$  '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  $\Rightarrow$  '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  $\Rightarrow$  ('e, 'a) Node  $\Rightarrow$  bool where
  is-leveln-node 0 (Tip -)  $\longleftrightarrow$  True |
  is-leveln-node (Suc n) (Node2 - n1 n2)  $\longleftrightarrow$ 
    is-leveln-node n n1  $\wedge$  is-leveln-node n n2 |
  is-leveln-node (Suc n) (Node3 - n1 n2 n3)  $\longleftrightarrow$ 
    is-leveln-node n n1  $\wedge$  is-leveln-node n n2  $\wedge$  is-leveln-node n n3 |
  is-leveln-node -  $\longleftrightarrow$  False
```

```
primrec is-leveln-digit :: nat  $\Rightarrow$  ('e, 'a) Digit  $\Rightarrow$  bool where
  is-leveln-digit n (One n1)  $\longleftrightarrow$  is-leveln-node n n1 |
  is-leveln-digit n (Two n1 n2)  $\longleftrightarrow$  is-leveln-node n n1  $\wedge$ 
    is-leveln-node n n2 |
  is-leveln-digit n (Three n1 n2 n3)  $\longleftrightarrow$  is-leveln-node n n1  $\wedge$ 
```

$$\begin{aligned}
& \text{is-leveln-node } n \ n2 \wedge \text{is-leveln-node } n \ n3 \mid \\
& \text{is-leveln-digit } n \ (\text{Four } n1 \ n2 \ n3 \ n4) \longleftrightarrow \text{is-leveln-node } n \ n1 \wedge \\
& \text{is-leveln-node } n \ n2 \wedge \text{is-leveln-node } n \ n3 \wedge \text{is-leveln-node } n \ n4
\end{aligned}$$

primrec *is-leveln-ftree* :: $\text{nat} \Rightarrow ('e, 'a) \text{FingerTreeStruc} \Rightarrow \text{bool}$ **where**
is-leveln-ftree *n Empty* $\longleftrightarrow \text{True}$ |
is-leveln-ftree *n (Single nd)* $\longleftrightarrow \text{is-leveln-node } n \ nd$ |
is-leveln-ftree *n (Deep - l t r)* $\longleftrightarrow \text{is-leveln-digit } n \ l \wedge$
is-leveln-digit } n \ r \wedge \text{is-leveln-ftree } (Suc \ n) \ t

primrec *is-measured-node* :: $('e, 'a::\text{monoid-add}) \text{Node} \Rightarrow \text{bool}$ **where**
is-measured-node (*Tip - -*) $\longleftrightarrow \text{True}$ |
is-measured-node (*Node2 a n1 n2*) $\longleftrightarrow ((\text{is-measured-node } n1) \wedge$
*(is-measured-node } n2)) \wedge (a = (\text{gmn } n1) + (\text{gmn } n2)) |
is-measured-node (*Node3 a n1 n2 n3*) $\longleftrightarrow ((\text{is-measured-node } n1) \wedge$
*(is-measured-node } n2) \wedge (\text{is-measured-node } n3)) \wedge
*(a = (\text{gmn } n1) + (\text{gmn } n2) + (\text{gmn } n3))***

primrec *is-measured-digit* :: $('e, 'a::\text{monoid-add}) \text{Digit} \Rightarrow \text{bool}$ **where**
is-measured-digit (*One a*) = *is-measured-node a* |
is-measured-digit (*Two a b*) =
((is-measured-node } a) \wedge (\text{is-measured-node } b)) |
is-measured-digit (*Three a b c*) =
((is-measured-node } a) \wedge (\text{is-measured-node } b) \wedge (\text{is-measured-node } c)) |
is-measured-digit (*Four a b c d*) = *((is-measured-node } a) \wedge*
(is-measured-node } b) \wedge (\text{is-measured-node } c) \wedge (\text{is-measured-node } d))

primrec *is-measured-ftree* :: $('e, 'a::\text{monoid-add}) \text{FingerTreeStruc} \Rightarrow \text{bool}$ **where**
is-measured-ftree Empty $\longleftrightarrow \text{True}$ |
is-measured-ftree (Single n1) $\longleftrightarrow (\text{is-measured-node } n1)$ |
is-measured-ftree (Deep a l m r) $\longleftrightarrow ((\text{is-measured-digit } l) \wedge$
*(is-measured-ftree } m) \wedge (\text{is-measured-digit } r)) \wedge
*(a = ((gmd } l) + (\text{gmft } m) + (\text{gmd } r)))**

Structural invariant for finger trees

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

1.1.2 Abstraction to Lists

primrec *nodeToList* :: $('e, 'a) \text{Node} \Rightarrow ('e \times 'a) \text{list}$ **where**
nodeToList (*Tip e a*) = $[(e, a)]$ |
nodeToList (*Node2 - a b*) = $(\text{nodeToList } a) @ (\text{nodeToList } b)$ |
nodeToList (*Node3 - a b c*)
= $(\text{nodeToList } a) @ (\text{nodeToList } b) @ (\text{nodeToList } c)$

primrec *digitToList* :: $('e, 'a) \text{Digit} \Rightarrow ('e \times 'a) \text{list}$ **where**
digitToList (*One a*) = *nodeToList a* |
digitToList (*Two a b*) = $(\text{nodeToList } a) @ (\text{nodeToList } b)$ |
digitToList (*Three a b c*)

$$\begin{aligned}
&= (\text{nodeToList } a) @ (\text{nodeToList } b) @ (\text{nodeToList } c) | \\
&\text{digitToList } (\text{Four } a \ b \ c \ d) \\
&= (\text{nodeToList } a) @ (\text{nodeToList } b) @ (\text{nodeToList } c) @ (\text{nodeToList } d)
\end{aligned}$$

List representation of a finger tree

primrec *toList* :: ('e, 'a) FingerTreeStruc \Rightarrow ('e \times '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 \neq Nil
 <proof>

lemma *digitToList-empty*: digitToList d \neq 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
 \implies (*gmft* t) = sum-list (map snd (toList t))
 <proof>

lemma *gmft-correct2*: *ft-invar* t \implies (*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 = [] \longleftrightarrow t = Empty
 <proof>

1.2.2 Annotation

Sum of annotations of all elements of a finger tree

definition *annot* :: ('e,'a::monoid-add) FingerTreeStruc \Rightarrow 'a
where *annot* t = gmft t

lemma *annot-correct*:

ft-invar t \implies *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 \Rightarrow ('e,'a) FingerTreeStruc
 \Rightarrow ('e,'a) Digit \Rightarrow ('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 \Rightarrow ('e,'a) FingerTreeStruc
 \Rightarrow ('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
 \Rightarrow ('e,'a) Node \Rightarrow ('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*: \llbracket is-leveln-ftree n t; is-leveln-node n nd \rrbracket
 \implies is-leveln-ftree n (nlcons nd t)
 <proof>

lemma *nlcons-invmeas*: \llbracket is-measured-ftree t; is-measured-node nd \rrbracket

\implies *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*: \llbracket *is-leveln-ftree* n t; *is-leveln-node* n nd \rrbracket
 \implies *is-leveln-ftree* n (nrcons t nd)
 ⟨proof⟩

lemma *nrcons-invmeas*: \llbracket *is-measured-ftree* t; *is-measured-node* nd \rrbracket
 \implies *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)
 \Rightarrow ('e,'a) *FingerTreeStruc* \Rightarrow ('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 \implies *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* \Rightarrow ('e × 'a) \Rightarrow ('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 \implies *ft-invar* (t > a)

<proof>

lemma *rcons-list[simp]*: $toList (t \triangleright a) = (toList t) @ [a]$
<proof>

1.2.4 Convert list to tree

primrec *toTree* :: $(e \times 'a :: monoid-add) list \Rightarrow (e, 'a) FingerTreeStruc$ **where**
toTree [] = *Empty* |
toTree (a#xs) = a \triangleleft (*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* :: $(e, 'a :: monoid-add) Digit \Rightarrow (e, 'a) FingerTreeStruc$
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* :: $(e, 'a) Node \Rightarrow (e, 'a) Digit$ **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* :: $(e, 'a) Node list \Rightarrow (e, 'a) Digit$ **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* :: $(e, 'a) Digit \Rightarrow (e, 'a) Node list$ **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*:: $(e, 'a) Node \Rightarrow (e \times 'a)$ **where**
n-unwrap (*Tip* e a) = (e,a) |

$n\text{-unwrap } (\text{Node2} - a\ b) = \text{undefined}$
 $n\text{-unwrap } (\text{Node3} - a\ b\ c) = \text{undefined}$

type-synonym $(e, 'a)$ *ViewnRes* = $((e, 'a)$ *Node* \times $(e, 'a)$ *FingerTreeStruc*) *option*

lemma *viewnres-cases*:

fixes $r :: (e, 'a)$ *ViewnRes*

obtains $(\text{Nil})\ r = \text{None}$ |

$(\text{Cons})\ a\ t$ **where** $r = \text{Some } (a, t)$

$\langle \text{proof} \rangle$

lemma *viewnres-split*:

P $(\text{case-option } f1\ (\text{case-prod } f2)\ x) =$

$((x = \text{None} \longrightarrow P\ f1) \wedge (\forall a\ b.\ x = \text{Some } (a, b) \longrightarrow P\ (f2\ a\ b)))$

$\langle \text{proof} \rangle$

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

fun *viewLn* :: $(e, 'a :: \text{monoid-add})$ *FingerTreeStruc* \Rightarrow $(e, 'a)$ *ViewnRes* **where**

viewLn *Empty* = *None* |

viewLn $(\text{Single } a) = \text{Some } (a, \text{Empty})$ |

viewLn $(\text{Deep} - (\text{Two } a\ b)\ m\ sf) = \text{Some } (a, (\text{deep } (\text{One } b)\ m\ sf))$ |

viewLn $(\text{Deep} - (\text{Three } a\ b\ c)\ m\ sf) = \text{Some } (a, (\text{deep } (\text{Two } b\ c)\ m\ sf))$ |

viewLn $(\text{Deep} - (\text{Four } a\ b\ c\ d)\ m\ sf) = \text{Some } (a, (\text{deep } (\text{Three } b\ c\ d)\ m\ sf))$ |

viewLn $(\text{Deep} - (\text{One } a)\ m\ sf) =$

$(\text{case } \text{viewLn } m\ \text{of}$

$\text{None} \Rightarrow \text{Some } (a, (\text{digitToTree } sf))$ |

$\text{Some } (b, m2) \Rightarrow \text{Some } (a, (\text{deep } (\text{nodeToDigit } b)\ m2\ sf))$)

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

fun *viewRn* :: $(e, 'a :: \text{monoid-add})$ *FingerTreeStruc* \Rightarrow $(e, 'a)$ *ViewnRes* **where**

viewRn *Empty* = *None* |

viewRn $(\text{Single } a) = \text{Some } (a, \text{Empty})$ |

viewRn $(\text{Deep} - \text{pr } m\ (\text{Two } a\ b)) = \text{Some } (b, (\text{deep } \text{pr } m\ (\text{One } a)))$ |

viewRn $(\text{Deep} - \text{pr } m\ (\text{Three } a\ b\ c)) = \text{Some } (c, (\text{deep } \text{pr } m\ (\text{Two } a\ b)))$ |

viewRn $(\text{Deep} - \text{pr } m\ (\text{Four } a\ b\ c\ d)) = \text{Some } (d, (\text{deep } \text{pr } m\ (\text{Three } a\ b\ c)))$ |

viewRn $(\text{Deep} - \text{pr } m\ (\text{One } a)) =$

$(\text{case } \text{viewRn } m\ \text{of}$

$\text{None} \Rightarrow \text{Some } (a, (\text{digitToTree } \text{pr}))$ |

$\text{Some } (b, m2) \Rightarrow \text{Some } (a, (\text{deep } \text{pr } m2\ (\text{nodeToDigit } b)))$)

lemma

digitToTree-inv: $\text{is-leveln-digit } n\ d \Longrightarrow \text{is-leveln-ftree } n\ (\text{digitToTree } d)$

is-measured-digit $d \Longrightarrow \text{is-measured-ftree } (\text{digitToTree } d)$

$\langle \text{proof} \rangle$

lemma *digitToTree-list*: $\text{toList } (\text{digitToTree } d) = \text{digitToList } d$

$\langle \text{proof} \rangle$

lemma *nodeToDigit-inv*:

is-leveln-node (Suc n) nd \implies *is-leveln-digit* n (nodeToDigit nd)
is-measured-node nd \implies *is-measured-digit* (nodeToDigit nd)
<proof>

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

<proof>

lemma *viewLn-empty*: $t \neq \text{Empty} \longleftrightarrow (\text{viewLn } t) \neq \text{None}$

<proof>

lemma *viewLn-inv*: \llbracket

is-measured-ftree t; *is-leveln-ftree* n t; *viewLn* t = *Some* (nd, s)
 $\rrbracket \implies$ *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)

\implies *toList* t = (*nodeToList* nd) @ (*toList* s)

<proof>

lemma *viewRn-empty*: $t \neq \text{Empty} \longleftrightarrow (\text{viewRn } t) \neq \text{None}$

<proof>

lemma *viewRn-inv*: \llbracket

is-measured-ftree t; *is-leveln-ftree* n t; *viewRn* t = *Some* (nd, s)
 $\rrbracket \implies$ *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)

\implies *toList* t = (*toList* s) @ (*nodeToList* nd)

<proof>

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

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

definition *viewL* :: ('e,'a::*monoid-add*) *FingerTreeStruc* \Rightarrow ('e,'a) *viewres*

where

viewL t = (case *viewLn* t of

None \Rightarrow *None* |

Some (a, t2) \Rightarrow *Some* ((n-unwrap a), t2))

lemma *viewL-correct*:

assumes *INV*: *ft-invar* t

shows

$(t = \text{Empty} \implies \text{viewL } t = \text{None})$
 $(t \neq \text{Empty} \implies (\exists a \ s. \text{viewL } t = \text{Some } (a, s) \wedge \text{ft-invar } s$
 $\quad \wedge \text{toList } t = a \# \text{toList } s))$

$\langle \text{proof} \rangle$

lemma *viewL-correct-empty[simp]*: $\text{viewL } \text{Empty} = \text{None}$
 $\langle \text{proof} \rangle$

lemma *viewL-correct-nonEmpty*:
assumes *ft-invar* $t \neq \text{Empty}$
obtains $a \ s$ **where**
 $\text{viewL } t = \text{Some } (a, s) \wedge \text{ft-invar } s \wedge \text{toList } t = a \# \text{toList } s$
 $\langle \text{proof} \rangle$

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

definition *viewR* :: $(\text{'e}, \text{'a} :: \text{monoid-add}) \text{ FingerTreeStruc} \Rightarrow (\text{'e}, \text{'a}) \text{ viewres}$
where
 $\text{viewR } t = (\text{case viewRn } t \text{ of}$
 $\quad \text{None} \Rightarrow \text{None} \mid$
 $\quad \text{Some } (a, t2) \Rightarrow \text{Some } ((n\text{-unwrap } a), t2))$

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

lemma *viewR-correct-empty[simp]*: $\text{viewR } \text{Empty} = \text{None}$
 $\langle \text{proof} \rangle$

lemma *viewR-correct-nonEmpty*:
assumes *ft-invar* $t \neq \text{Empty}$
obtains $a \ s$ **where**
 $\text{viewR } t = \text{Some } (a, s) \wedge \text{ft-invar } s \wedge \text{toList } t = \text{toList } s @ [a]$
 $\langle \text{proof} \rangle$

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* :: $(\text{'e}, \text{'a}) \text{ FingerTreeStruc} \Rightarrow \text{bool}$ **where**
 $[\text{code del}]: \text{isEmpty } t = (t = \text{Empty})$

lemma *isEmpty-correct*: $\text{isEmpty } t \iff \text{toList } t = []$
 $\langle \text{proof} \rangle$

lemma $[\text{code}]: \text{isEmpty } t = (\text{case } t \text{ of } \text{Empty} \Rightarrow \text{True} \mid - \Rightarrow \text{False})$
 $\langle \text{proof} \rangle$

Leftmost element

definition *head* :: ('e,'a::monoid-add) FingerTreeStruc \Rightarrow 'e \times 'a **where**
 head t = (case viewL t of (Some (a, -)) \Rightarrow a)

lemma *head-correct*:

assumes *ft-invar* t t \neq Empty

shows *head* t = hd (toList t)

<proof>

All but the leftmost element

definition *tail*

 :: ('e,'a::monoid-add) FingerTreeStruc \Rightarrow ('e,'a) FingerTreeStruc

where

tail t = (case viewL t of (Some (-, m)) \Rightarrow m)

lemma *tail-correct*:

assumes *ft-invar* t t \neq Empty

shows toList (*tail* t) = tl (toList t) **and** *ft-invar* (*tail* t)

<proof>

Rightmost element

definition *headR* :: ('e,'a::monoid-add) FingerTreeStruc \Rightarrow 'e \times 'a **where**
 headR t = (case viewR t of (Some (a, -)) \Rightarrow a)

lemma *headR-correct*:

assumes *ft-invar* t t \neq Empty

shows *headR* t = last (toList t)

<proof>

All but the rightmost element

definition *tailR*

 :: ('e,'a::monoid-add) FingerTreeStruc \Rightarrow ('e,'a) FingerTreeStruc

where

tailR t = (case viewR t of (Some (-, m)) \Rightarrow m)

lemma *tailR-correct*:

assumes *ft-invar* t t \neq 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

\Rightarrow ('e,'a) FingerTreeStruc \Rightarrow ('e,'a) FingerTreeStruc **where**

lconsNlist [] t = t |

lconsNlist (x#xs) t = nlcons x (lconsNlist xs t)

primrec *rconsNlist* :: ('e,'a::monoid-add) FingerTreeStruc

\Rightarrow ('e,'a) Node list \Rightarrow ('e,'a) FingerTreeStruc **where**

rconsNlist t [] = t |

rconsNlist t (x#xs) = rconsNlist (nrcons t x) xs

fun *nodes* :: ('e,'a::monoid-add) Node list \Rightarrow ('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  $\forall x \in \text{set } xs. (is-leveln-node\ n\ x \wedge is-measured-node\ x)$ 
shows
is-leveln-ftree n (lconsNlist xs t)  $\wedge$  is-measured-ftree (lconsNlist xs t)
<proof>

```

lemma *rconsNlist-inv*:

```

assumes is-leveln-ftree n t
and is-measured-ftree t
and  $\forall x \in \text{set } xs. (is-leveln-node\ n\ x \wedge is-measured-node\ x)$ 
shows
is-leveln-ftree n (rconsNlist t xs)  $\wedge$  is-measured-ftree (rconsNlist t xs)
<proof>

```

lemma *nodes-inv*:

```

assumes  $\forall x \in \text{set } ts. is-leveln-node\ n\ x \wedge is-measured-node\ x$ 
and length ts  $\geq 2$ 
shows  $\forall x \in \text{set } (nodes\ ts). is-leveln-node\ (Suc\ n)\ x \wedge is-measured-node\ x$ 
<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  $\forall x \in \text{set } ts. is-leveln-node\ n\ x \wedge is-measured-node\ x$ 
shows
 $\forall x \in \text{set } (nodes\ (digitToNlist\ sf1\ @\ ts\ @\ digitToNlist\ pr2)).$ 
  is-leveln-node (Suc n) x  $\wedge$  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 $\forall x \in \text{set } xs. (\text{is-leveln-node } n \ x \wedge \text{is-measured-node } x)$
shows *is-leveln-ftree n (app3 t1 xs t2) \wedge is-measured-ftree (app3 t1 xs t2)*
<proof>

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

lemma *nodes-list*: $\text{length } xs \geq 2 \implies \text{nlistToList (nodes } xs) = \text{nlistToList } xs$
<proof>

lemma *nlistToList-app*:
nlistToList (xs@ys) = (nlistToList xs) @ (nlistToList ys)
<proof>

lemma *nlistListLCons*: $\text{toList (lconsNlist } xs \ t) = (\text{nlistToList } xs) @ (\text{toList } t)$
<proof>

lemma *nlistListRCons*: $\text{toList (rconsNlist } t \ xs) = (\text{toList } t) @ (\text{nlistToList } xs)$
<proof>

lemma *app3-list-lem1*:
nlistToList (nodes (digitToNlist sf1 @ ts @ digitToNlist pr2)) =
digitToNlist sf1 @ nlistToList ts @ digitToNlist pr2
<proof>

lemma *app3-list*:
 $\text{toList (app3 } t1 \ xs \ t2) = (\text{toList } t1) @ (\text{nlistToList } xs) @ (\text{toList } t2)$
<proof>

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

lemma *app-correct*:
assumes *ft-invar t1 ft-invar t2*
shows $\text{toList (app } t1 \ t2) = (\text{toList } t1) @ (\text{toList } t2)$
and *ft-invar (app t1 t2)*
<proof>

lemma *app-inv*: $\llbracket ft\text{-invar } t1; ft\text{-invar } t2 \rrbracket \implies ft\text{-invar } (app\ t1\ t2)$
 ⟨*proof*⟩

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

1.2.7 Splitting

type-synonym $(e, 'a)\ SplitDigit =$
 $(e, 'a)\ Node\ list \times (e, 'a)\ Node \times (e, 'a)\ Node\ list$
type-synonym $(e, 'a)\ SplitTree =$
 $(e, 'a)\ FingerTreeStruc \times (e, 'a)\ Node \times (e, 'a)\ FingerTreeStruc$

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

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

Splitting a list of nodes

fun *splitNlist* :: $(a::monoid\text{-add} \Rightarrow bool) \Rightarrow 'a \Rightarrow (e, 'a)\ Node\ list$
 $\Rightarrow (e, 'a)\ SplitDigit$ **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* :: $(a::monoid\text{-add} \Rightarrow bool) \Rightarrow 'a \Rightarrow (e, 'a)\ Digit$
 $\Rightarrow (e, 'a)\ SplitDigit$ **where**
splitDigit p i d = *splitNlist p i* (*digitToNlist d*)

Creating a finger tree from list of nodes

definition *nlistToTree* :: $(e, 'a)::monoid\text{-add})\ Node\ list$
 $\Rightarrow (e, 'a)\ FingerTreeStruc$ **where**
nlistToTree xs = *lconsNlist xs Empty*

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

```

fun nsplitTree :: ('a::monoid-add  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  ('e,'a) FingerTreeStruc
 $\Rightarrow$  ('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*:

$\forall x \in \text{set } nl. \text{is-measured-node } x \implies \text{is-measured-ftree } (nlistToTree \text{ } nl)$
 $\forall x \in \text{set } nl. \text{is-leveln-node } n \ x \implies \text{is-leveln-ftree } n \ (nlistToTree \text{ } nl)$
<proof>

lemma *nlistToTree-list*: $toList \ (nlistToTree \text{ } nl) = nlistToList \ nl$

<proof>

lemma *deepL-inv*:

assumes $\text{is-leveln-ftree } (Suc \ n) \ m \wedge \text{is-measured-ftree } m$
and $\text{is-leveln-digit } n \ sf \wedge \text{is-measured-digit } sf$
and $\forall x \in \text{set } pr. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x) \wedge \text{length } pr \leq 4$
shows $\text{is-leveln-ftree } n \ (\text{deepL } pr \ m \ sf) \wedge \text{is-measured-ftree } (\text{deepL } pr \ m \ sf)$
<proof>

lemma *nlistToDigit-list*:

assumes $1 \leq \text{length } xs \wedge \text{length } xs \leq 4$
shows $\text{digitToList}(\text{nlistToDigit } xs) = \text{nlistToList } xs$
<proof>

lemma *deepL-list*:

assumes $\text{is-leveln-ftree } (Suc \ n) \ m \wedge \text{is-measured-ftree } m$
and $\text{is-leveln-digit } n \ sf \wedge \text{is-measured-digit } sf$
and $\forall x \in \text{set } pr. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x) \wedge \text{length } pr \leq 4$

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

lemma *deepR-inv*:

assumes $is-leveln-ftree (Suc n) m \wedge is-measured-ftree m$
and $is-leveln-digit n pr \wedge is-measured-digit pr$
and $\forall x \in set sf. (is-measured-node x \wedge is-leveln-node n x) \wedge length sf \leq 4$
shows $is-leveln-ftree n (deepR pr m sf) \wedge is-measured-ftree (deepR pr m sf)$
 ⟨proof⟩

lemma *deepR-list*:

assumes $is-leveln-ftree (Suc n) m \wedge is-measured-ftree m$
and $is-leveln-digit n pr \wedge is-measured-digit pr$
and $\forall x \in set sf. (is-measured-node x \wedge is-leveln-node n x) \wedge length sf \leq 4$
shows $toList (deepR pr m sf) = digitToList pr @ toList m @ nlistToList sf$
 ⟨proof⟩

primrec *gmn*:: ('e, 'a::monoid-add) Node list \Rightarrow 'a **where**

$gmn [] = 0$
 $gmn (x\#xs) = gmn x + gmn xs$

lemma *gmn-correct*:

assumes $\forall x \in set xs. is-measured-node x$
shows $gmn xs = sum-list (map snd (nlistToList xs))$
 ⟨proof⟩

lemma *splitNlist-correct*: \llbracket

$\bigwedge (a::'a) (b::'a). p a \implies p (a + b);$
 $\neg p i;$
 $p (i + gmn (nl :: ('e, 'a::monoid-add) Node list));$
 $splitNlist p i nl = (l, n, r)$
 $\rrbracket \implies$
 $\neg p (i + (gmn l))$
 \wedge
 $p (i + (gmn l) + (gmn n))$
 \wedge
 $nl = l @ n \# r$

⟨proof⟩

lemma *digitToNlist-inv*:

$is-measured-digit d \implies (\forall x \in set (digitToNlist d). is-measured-node x)$
 $is-leveln-digit n d \implies (\forall x \in set (digitToNlist d). is-leveln-node n x)$
 ⟨proof⟩

lemma *gmn-gmd*:

$is-measured-digit d \implies gmn (digitToNlist d) = gmd d$
 ⟨proof⟩

lemma *gmn-gmd*:

is-measured-node nd \implies *gmd* (*nodeToDigit nd*) = *gmn nd*
 ⟨*proof*⟩

lemma *splitDigit-inv*:

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

⟨*proof*⟩

lemma *splitDigit-inv'*:

⌈
splitDigit p i d = (*l*, *nd*, *r*);
is-measured-digit d;
is-leveln-digit n d
 ⌋ \implies
 $(\forall x \in \text{set } l. (is-measured-node\ x \wedge is-leveln-node\ n\ x))$
 \wedge
 $(\forall x \in \text{set } r. (is-measured-node\ x \wedge is-leveln-node\ n\ x))$
 \wedge
 $(is-measured-node\ nd \wedge is-leveln-node\ n\ nd)$

⟨*proof*⟩

lemma *splitDigit-list*: *splitDigit p i d* = (*l*, *n*, *r*) \implies

(*digitToList d*) = (*nlistToList l*) @ (*nodeToList n*) @ (*nlistToList r*)
 \wedge *length l* \leq 4 \wedge *length r* \leq 4
 ⟨*proof*⟩

lemma *gmnl-gmft*: $\forall x \in \text{set } nl. is-measured-node\ x \implies$

gmft (*nlistToTree nl*) = *gmnl nl*

$\langle \text{proof} \rangle$

lemma *gmftR-gmnl*:

assumes *is-leveln-ftree* (Suc *n*) *m* \wedge *is-measured-ftree* *m*

and *is-leveln-digit* *n* *pr* \wedge *is-measured-digit* *pr*

and $\forall x \in \text{set } sf. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x) \wedge \text{length } sf \leq 4$

shows *gmft* (deepR *pr* *m* *sf*) = *gmd* *pr* + *gmft* *m* + *gmnl* *sf*

$\langle \text{proof} \rangle$

lemma *nsplitTree-invpres*: \llbracket

is-leveln-ftree *n* (*s*::('e,'a)::monoid-add) *FingerTreeStruc*);

is-measured-ftree *s*;

$\neg p$ *i*;

p (*i* + (*gmft* *s*));

(*nsplitTree* *p* *i* *s*) = (*l*, *nd*, *r*) \rrbracket

\implies

is-leveln-ftree *n* *l*

\wedge

is-measured-ftree *l*

\wedge

is-leveln-ftree *n* *r*

\wedge

is-measured-ftree *r*

\wedge

is-leveln-node *n* *nd*

\wedge

is-measured-node *nd*

$\langle \text{proof} \rangle$

lemma *nsplitTree-correct*: \llbracket

is-leveln-ftree *n* (*s*::('e,'a)::monoid-add) *FingerTreeStruc*);

is-measured-ftree *s*;

$\bigwedge(a::'a) (b::'a). p \ a \implies p \ (a + b)$;

$\neg p$ *i*;

p (*i* + (*gmft* *s*));

(*nsplitTree* *p* *i* *s*) = (*l*, *nd*, *r*) \rrbracket

$\implies (\text{toList } s) = (\text{toList } l) \ @ \ (\text{nodeToList } nd) \ @ \ (\text{toList } r)$

\wedge

$\neg p$ (*i* + (*gmft* *l*))

\wedge

p (*i* + (*gmft* *l*) + (*gmnl* *nd*))

\wedge

is-leveln-ftree *n* *l*

\wedge

is-measured-ftree *l*

\wedge

is-leveln-ftree *n* *r*

\wedge

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

$\langle proof \rangle$

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*

$:: ('a::monoid-add \Rightarrow bool) \Rightarrow 'a \Rightarrow ('e, 'a) FingerTreeStruc$
 $\Rightarrow ('e, 'a) FingerTreeStruc \times ('e \times 'a) \times ('e, 'a) FingerTreeStruc$

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*: $\neg p i$

assumes *sum-tt*: $p (i + annot s)$

assumes *fnt*: (*splitTree* p i s) = ($l, (e, a), r$)

shows *ft-invar* l **and** *ft-invar* r

$\langle proof \rangle$

lemma *splitTree-correct*:

assumes *inv*: *ft-invar* ($s:: ('e, 'a)::monoid-add$) *FingerTreeStruc*)

assumes *mono*: $\forall a b. p a \longrightarrow p (a + b)$

assumes *init-ff*: $\neg p i$

assumes *sum-tt*: $p (i + annot s)$

assumes *fnt*: (*splitTree* p i s) = ($l, (e, a), r$)

shows (*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

$\langle proof \rangle$

lemma *splitTree-correctE*:

assumes *inv*: *ft-invar* ($s:: ('e, 'a)::monoid-add$) *FingerTreeStruc*)

assumes *mono*: $\forall a b. p a \longrightarrow p (a + b)$

assumes *init-ff*: $\neg p i$

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**
 $foldl-node\ f\ \sigma\ (Tip\ e\ a) = f\ \sigma\ (e,a)|$
 $foldl-node\ f\ \sigma\ (Node2 - a\ b) = foldl-node\ f\ (foldl-node\ f\ \sigma\ a)\ b|$
 $foldl-node\ f\ \sigma\ (Node3 - a\ b\ c) =$
 $foldl-node\ f\ (foldl-node\ f\ (foldl-node\ f\ \sigma\ a)\ b)\ c$

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

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

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

lemma $foldl-node-correct:$
 $foldl-node\ f\ \sigma\ nd = List.foldl\ f\ \sigma\ (nodeToList\ nd)$
 <proof>

lemma $foldl-digit-correct:$
 $foldl-digit\ f\ \sigma\ d = List.foldl\ f\ \sigma\ (digitToList\ d)$
 <proof>

lemma $foldr-node-correct:$
 $foldr-node\ f\ nd\ \sigma = List.foldr\ f\ (nodeToList\ nd)\ \sigma$
 <proof>

lemma *foldr-digit-correct*:

$foldr\text{-}digit\ f\ d\ \sigma = List.foldr\ f\ (digitToList\ d)\ \sigma$
(*proof*)

Fold from left

primrec *foldl* :: ('s \Rightarrow 'e \times 'a \Rightarrow 's) \Rightarrow 's \Rightarrow ('e,'a) FingerTreeStruc \Rightarrow 's

where

$foldl\ f\ \sigma\ Empty = \sigma$ |
 $foldl\ f\ \sigma\ (Single\ nd) = foldl\text{-}node\ f\ \sigma\ nd$ |
 $foldl\ f\ \sigma\ (Deep\ -\ d1\ m\ d2) =$
 $foldl\text{-}digit\ f\ (foldl\ f\ (foldl\text{-}digit\ f\ \sigma\ d1)\ m)\ d2$

lemma *foldl-correct*:

$foldl\ f\ \sigma\ t = List.foldl\ f\ \sigma\ (toList\ t)$
(*proof*)

Fold from right

primrec *foldr* :: ('e \times 'a \Rightarrow 's \Rightarrow 's) \Rightarrow ('e,'a) FingerTreeStruc \Rightarrow 's \Rightarrow 's

where

$foldr\ f\ Empty\ \sigma = \sigma$ |
 $foldr\ f\ (Single\ nd)\ \sigma = foldr\text{-}node\ f\ nd\ \sigma$ |
 $foldr\ f\ (Deep\ -\ d1\ m\ d2)\ \sigma$
 $= foldr\text{-}digit\ f\ d1\ (foldr\ f\ m\ (foldr\text{-}digit\ f\ d2\ \sigma))$

lemma *foldr-correct*:

$foldr\ f\ t\ \sigma = List.foldr\ f\ (toList\ t)\ \sigma$
(*proof*)

1.2.9 Number of elements

primrec *count-node* :: ('e, 'a) Node \Rightarrow nat **where**

$count\text{-}node\ (Tip\ -\ a) = 1$ |
 $count\text{-}node\ (Node2\ -\ a\ b) = count\text{-}node\ a + count\text{-}node\ b$ |
 $count\text{-}node\ (Node3\ -\ a\ b\ c) = count\text{-}node\ a + count\text{-}node\ b + count\text{-}node\ c$

primrec *count-digit* :: ('e,'a) Digit \Rightarrow nat **where**

$count\text{-}digit\ (One\ a) = count\text{-}node\ a$ |
 $count\text{-}digit\ (Two\ a\ b) = count\text{-}node\ a + count\text{-}node\ b$ |
 $count\text{-}digit\ (Three\ a\ b\ c) = count\text{-}node\ a + count\text{-}node\ b + count\text{-}node\ c$ |
 $count\text{-}digit\ (Four\ a\ b\ c\ d)$
 $= count\text{-}node\ a + count\text{-}node\ b + count\text{-}node\ c + count\text{-}node\ d$

lemma *count-node-correct*:

$count\text{-}node\ n = length\ (nodeToList\ n)$
(*proof*)

lemma *count-digit-correct*:

$count\text{-}digit\ d = length\ (digitToList\ d)$

<proof>

primrec *count* :: ('e,'a) *FingerTreeStruc* \Rightarrow *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** \triangleleft 65)
no-notation *FingerTreeStruc.rcons* (**infixl** \triangleright 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 \Longrightarrow *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 \times 'a) \times ('e,'a::monoid-add) *FingerTreeStruc*) *option* .
 case r of None \Rightarrow *True* | *Some (a,t)* \Rightarrow *FingerTreeStruc.ft-invar t* }
<proof>

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

lemma *Abs-viewres-inverse-None*[simp]:
 $Rep\text{-}viewres (Abs\text{-}viewres None) = None$
 ⟨proof⟩

lemma *Abs-viewres-inverse-Some*:
 $FingerTreeStruc.ft\text{-}invar t \implies$
 $Rep\text{-}viewres (Abs\text{-}viewres (Some (a,t))) = Some (a,t)$
 ⟨proof⟩

definition [code]: *extract-viewres-isNone* $r == Rep\text{-}viewres r = None$

definition [code]: *extract-viewres-a* $r ==$
 $case (Rep\text{-}viewres r) \text{ of } Some (a,t) \Rightarrow a$

definition *extract-viewres-t* $r ==$
 $case (Rep\text{-}viewres r) \text{ of } None \Rightarrow Abs\text{-}FingerTree Empty$
 $| Some (a,t) \Rightarrow Abs\text{-}FingerTree t$

lemma [code abstract]: *Rep-FingerTree (extract-viewres-t r) =*
 $(case (Rep\text{-}viewres r) \text{ of } None \Rightarrow Empty | Some (a,t) \Rightarrow t)$
 ⟨proof⟩

definition *extract-viewres* $r ==$
 $if \text{ extract-viewres-isNone } r \text{ then } None$
 $else Some (\text{extract-viewres-a } r, \text{extract-viewres-t } r)$

typedef (overloaded) $(e,'a)$ *splitres* =
 $\{ ((l,a,r):: ((e,'a) FingerTreeStruc \times (e \times 'a) \times (e,'a::monoid\text{-}add) Fin\text{-}gerTreeStruc))$
 $| l a r.$
 $FingerTreeStruc.ft\text{-}invar l \wedge FingerTreeStruc.ft\text{-}invar r \}$
 ⟨proof⟩

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

lemma *Abs-splitres-inverse*:
 $FingerTreeStruc.ft\text{-}invar r \implies FingerTreeStruc.ft\text{-}invar s \implies$
 $Rep\text{-}splitres (Abs\text{-}splitres ((r,a,s))) = (r,a,s)$
 ⟨proof⟩

definition [code]: *extract-splitres-a* $r == case (Rep\text{-}splitres r) \text{ of } (l,a,s) \Rightarrow a$

definition *extract-splitres-l* $r == case (Rep\text{-}splitres r) \text{ of } (l,a,r) \Rightarrow$
 $Abs\text{-}FingerTree l$

lemma [code abstract]: *Rep-FingerTree (extract-splitres-l r) = (case*
 $(Rep\text{-}splitres r) \text{ of } (l,a,r) \Rightarrow l)$
 ⟨proof⟩

definition *extract-splitres-r* $r == case (Rep\text{-}splitres r) \text{ of } (l,a,r) \Rightarrow$
 $Abs\text{-}FingerTree r$

lemma [code abstract]: *Rep-FingerTree (extract-splitres-r r) = (case*
 $(Rep\text{-}splitres r) \text{ of } (l,a,r) \Rightarrow r)$

<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*: $t = \text{empty} \longleftrightarrow \text{Rep-FingerTree } t = \text{Empty}$
<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 $tail\ t \equiv$

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 $tailR\ t \equiv$

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 $splitTree-aux\ p\ i\ t == if\ (\neg p\ i \wedge p\ (i+annot\ t))\ then$

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]:

$Rep-splitres\ (splitTree-aux\ p\ i\ t) = (if\ (\neg p\ i \wedge p\ (i+annot\ t))\ then$

(FingerTreeStruc.splitTree p i (Rep-FingerTree t))

else

(Empty,undefined,Empty)

<proof>

definition *foldl where*

[code]: $foldl\ f\ \sigma\ t == FingerTreeStruc.foldl\ f\ \sigma\ (Rep-FingerTree\ t)$

definition *foldr where*

[code]: $foldr\ f\ t\ \sigma == FingerTreeStruc.foldr\ f\ (Rep-FingerTree\ t)\ \sigma$

definition *count where*

[code]: $\text{count } t == \text{FingerTreeStruc.count } (\text{Rep-FingerTree } t)$

1.3.3 Correctness statements

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

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

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

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

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

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

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

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

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

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

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

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

lemma *isEmpty-correct*: $\text{isEmpty } t \longleftrightarrow t = \text{empty}$

$\langle proof \rangle$

lemma *head-correct*: $t \neq empty \implies head\ t = hd\ (toList\ t)$
 $\langle proof \rangle$

lemma *tail-correct*: $t \neq empty \implies toList\ (tail\ t) = tl\ (toList\ t)$
 $\langle proof \rangle$

lemma *headR-correct*: $t \neq empty \implies headR\ t = last\ (toList\ t)$
 $\langle proof \rangle$

lemma *tailR-correct*: $t \neq empty \implies toList\ (tailR\ t) = butlast\ (toList\ t)$
 $\langle proof \rangle$

lemma *app-correct*: $toList\ (app\ s\ t) = toList\ s\ @\ toList\ t$
 $\langle proof \rangle$

lemma *splitTree-correct*:

assumes *mono*: $\forall a\ b.\ p\ a \longrightarrow p\ (a + b)$

assumes *init-ff*: $\neg p\ i$

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 $\neg p\ (i + annot\ l)$

and $p\ (i + annot\ l + a)$

$\langle proof \rangle$

lemma *splitTree-correctE*:

assumes *mono*: $\forall a\ b.\ p\ a \longrightarrow p\ (a + b)$

assumes *init-ff*: $\neg p\ i$

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**

$p\ (i + annot\ l + a)$

$\langle proof \rangle$

lemma *foldl-correct*: $foldl\ f\ \sigma\ t = List.foldl\ f\ \sigma\ (toList\ t)$
 $\langle proof \rangle$

lemma *foldr-correct*: $foldr\ f\ t\ \sigma = List.foldr\ f\ (toList\ t)\ \sigma$
 $\langle proof \rangle$

lemma *count-correct*: $count\ t = length\ (toList\ t)$
 $\langle proof \rangle$

end

interpretation *FingerTree*: *FingerTree-loc* \langle *proof* \rangle

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.

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

$\frac{\textit{FingerTree.empty}::('a, 'b) \textit{FingerTree}}{\text{The empty finger tree } (O(1))}$

Spec *FingerTree.empty-correct*:

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

$\frac{\textit{FingerTree.annot}::('a, 'b) \textit{FingerTree} \Rightarrow 'b}{\text{Return sum of all annotations } (O(1))}$

Spec *FingerTree.annot-correct*:

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

$\frac{\textit{FingerTree.toTree}::('a \times 'b) \textit{list} \Rightarrow ('a, 'b) \textit{FingerTree}}{\text{Convert list to finger tree } (O(n \log(n)))}$

Spec *FingerTree.toTree-correct*:

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

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

Spec *FingerTree.lcons-correct*:

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

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

Spec *FingerTree.rcons-correct*:

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

$\frac{\textit{FingerTree.viewL}::('a, 'b) \textit{FingerTree} \Rightarrow (('a \times 'b) \times ('a, 'b) \textit{FingerTree}) \textit{option}}{\text{Detach leftmost element } (O(\log(n)), O(1) \text{ amortized})}$

Spec *FingerTree.viewL-correct*:

$?t = \textit{FingerTree.empty} \implies \textit{FingerTree.viewL} \ ?t = \textit{None}$

$?t \neq \textit{FingerTree.empty} \implies$

$\exists a \ s. \textit{FingerTree.viewL} \ ?t = \textit{Some} (a, s) \wedge$

$\textit{FingerTree.toList} \ ?t = a \# \textit{FingerTree.toList} \ s$

$\frac{\textit{FingerTree.viewR}::('a, 'b) \textit{FingerTree} \Rightarrow (('a \times 'b) \times ('a, 'b) \textit{FingerTree}) \textit{option}}{\text{Detach rightmost element } (O(\log(n)), O(1) \text{ amortized})}$

Spec *FingerTree.viewR-correct*:

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

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

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

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

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

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

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

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

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

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

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

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

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

$\text{FingerTree.splitTree}$

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

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*:

$$\begin{aligned} & \llbracket \forall a b. ?p a \longrightarrow ?p (a + b); \neg ?p ?i; ?p (?i + \text{FingerTree.annot } ?s); \\ & \quad \text{FingerTree.splitTree } ?p ?i ?s = (?l, (?e, ?a), ?r) \rrbracket \\ \implies & \text{FingerTree.toList } ?s = \\ & \quad \text{FingerTree.toList } ?l @ (?e, ?a) \# \text{FingerTree.toList } ?r \\ & \llbracket \forall a b. ?p a \longrightarrow ?p (a + b); \neg ?p ?i; ?p (?i + \text{FingerTree.annot } ?s); \\ & \quad \text{FingerTree.splitTree } ?p ?i ?s = (?l, (?e, ?a), ?r) \rrbracket \\ \implies & \neg ?p (?i + \text{FingerTree.annot } ?l) \\ & \llbracket \forall a b. ?p a \longrightarrow ?p (a + b); \neg ?p ?i; ?p (?i + \text{FingerTree.annot } ?s); \\ & \quad \text{FingerTree.splitTree } ?p ?i ?s = (?l, (?e, ?a), ?r) \rrbracket \\ \implies & ?p (?i + \text{FingerTree.annot } ?l + ?a) \end{aligned}$$

FingerTree.foldl

FingerTree.foldl::('a ⇒ 'b × 'c ⇒ 'a) ⇒ 'a ⇒ ('b, 'c) *FingerTree* ⇒ 'a

Fold with function from left

Spec *FingerTree.foldl-correct*:

FingerTree.foldl ?f ?σ ?t = *foldl* ?f ?σ (*FingerTree.toList* ?t)

FingerTree.foldr

FingerTree.foldr::('a × 'b ⇒ 'c ⇒ 'c) ⇒ ('a, 'b) *FingerTree* ⇒ 'c ⇒ 'c

Fold with function from right

Spec *FingerTree.foldr-correct*:

FingerTree.foldr ?f ?t ?σ = *foldr* ?f (*FingerTree.toList* ?t) ?σ

FingerTree.count::('a, 'b) *FingerTree* ⇒ nat

Return the number of elements

Spec *FingerTree.count-correct*:

FingerTree.count ?t = *length* (*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.