A Verified Imperative Implementation of B-Trees

Niels Mündler

Abstract

In this work, we use the interactive theorem prover Isabelle/HOL to verify an imperative implementation of the classical B-tree data structure [1]. The implementation supports set membership and insertion queries with efficient binary search for intra-node navigation. This is accomplished by first specifying the structure abstractly in the functional modeling language HOL and proving functional correctness. Using manual refinement, we derive an imperative implementation in Imperative/HOL. We show the validity of this refinement using the separation logic utilities from the Isabelle Refinement Framework [2]. The code can be exported to the programming languages SML and Scala. We examine the runtime of all operations indirectly by reproducing results of the logarithmic relationship between height and the number of nodes. The results are discussed in greater detail in the related Bachelor's Thesis [3].

Contents

theory *BTree*
imports *Main HOL−Data-Structures.Sorted-Less HOL−Data-Structures.Cmp*
begin

hide-const (**open**) *Sorted-Less.sorted*
abbreviation *sorted-less* ≡ *Sorted-Less.sorted*

1 Definition of the B-Tree

1.1 Datatype definition

B-trees can be considered to have all data stored interleaved as child nodes and separating elements (also keys or indices). We define them to either be a Node that holds a list of pairs of children and indices or be a completely empty Leaf.

datatype *'a btree = Leaf | Node ('a btree * 'a) list 'a btree*

type-synonym *'a btree-list = ('a btree * 'a) list*
type-synonym *'a btree-pair = ('a btree * 'a)*

abbreviation *subtrees* **where** *subtrees xs ≡ (map fst xs)*
abbreviation *separators* **where** *separators xs ≡ (map snd xs)*

1.2 Inorder and Set

The set of B-tree elements is defined automatically.

thm *btree.set*
value *set-btree (Node [(Leaf, (0::nat)), (Node [(Leaf, 1), (Leaf, 10)] Leaf, 12), (Leaf, 30), (Leaf, 100)] Leaf)*

The inorder view is defined with the help of the concat function.

fun *inorder :: 'a btree ⇒ 'a list* **where**
inorder Leaf = [] |
inorder (Node ts t) = concat (map (λ (sub, sep). inorder sub @ [sep]) ts) @ inorder t

abbreviation *inorder-pair ≡ λ(sub,sep). inorder sub @ [sep]*
abbreviation *inorder-list ts ≡ concat (map inorder-pair ts)*

thm *inorder.simps*

value *inorder (Node [(Leaf, (0::nat)), (Node [(Leaf, 1), (Leaf, 10)] Leaf, 12), (Leaf, 30), (Leaf, 100)] Leaf)*

1.3 Height and Balancedness

class *height =*
fixes *height :: 'a ⇒ nat*

instantiation *btree :: (type) height*
begin

fun *height-btree :: 'a btree ⇒ nat* **where**
height Leaf = 0 |

height (*Node ts t*) = *Suc* (*Max* (*height* ‘ (*set* (*subtrees ts*@[*t*])))))

instance ⟨*proof*⟩

end

Balancedness is defined is close accordance to the definition by Ernst

fun *bal*:: *'a btree* ⇒ *bool* **where**
bal Leaf = *True* |
bal (*Node ts t*) = (
(∀ *sub* ∈ *set* (*subtrees ts*). *height sub* = *height t*) ∧
(∀ *sub* ∈ *set* (*subtrees ts*). *bal sub*) ∧ *bal t*
)

value *height* (*Node* [(*Leaf*, (*0*::*nat*)), (*Node* [(*Leaf*, *1*), (*Leaf*, *10*)] *Leaf*, *12*), (*Leaf*, *30*), (*Leaf*, *100*)] *Leaf*)

1.4 Order

The order of a B-tree is defined just as in the original paper by Bayer.

fun *order*:: *nat* ⇒ *'a btree* ⇒ *bool* **where**
order k Leaf = *True* |
order k (*Node ts t*) = (
(*length ts* ≥ *k*) ∧
(*length ts* ≤ *2*∗*k*) ∧
(∀ *sub* ∈ *set* (*subtrees ts*). *order k sub*) ∧ *order k t*
)

The special condition for the root is called *root_order*

fun *root-order*:: *nat* ⇒ *'a btree* ⇒ *bool* **where**
root-order k Leaf = *True* |
root-order k (*Node ts t*) = (
(*length ts* > *0*) ∧
(*length ts* ≤ *2*∗*k*) ∧
(∀ *s* ∈ *set* (*subtrees ts*). *order k s*) ∧ *order k t*
)

1.5 Auxiliary Lemmas

lemma *separators-split*:
set (*separators* (*l*@(*a*,*b*)#*r*)) = *set* (*separators l*) ∪ *set* (*separators r*) ∪ {*b*}
⟨*proof*⟩

lemma *subtrees-split*:
set (*subtrees* (*l*@(*a*,*b*)#*r*)) = *set* (*subtrees l*) ∪ *set* (*subtrees r*) ∪ {*a*}
⟨*proof*⟩

lemma *finite-set-ins-swap*:
assumes *finite A*
shows *max a (Max (Set.insert b A)) = max b (Max (Set.insert a A))*
⟨*proof*⟩

lemma *finite-set-in-idem*:
assumes *finite A*
shows *max a (Max (Set.insert a A)) = Max (Set.insert a A)*
⟨*proof*⟩

lemma *height-Leaf*: *height t = 0 ⟷ t = Leaf*
⟨*proof*⟩

lemma *height-btree-order*:
height (Node (ls@[a]) t) = height (Node (a#ls) t)
⟨*proof*⟩

lemma *height-btree-sub*:
height (Node ((sub,x)#ls) t) = max (height (Node ls t)) (Suc (height sub))
⟨*proof*⟩

lemma *height-btree-last*:
height (Node ((sub,x)#ts) t) = max (height (Node ts sub)) (Suc (height t))
⟨*proof*⟩

lemma *set-btree-inorder*: *set (inorder t) = set-btree t*
⟨*proof*⟩

lemma *child-subset*: *p ∈ set t ⟹ set-btree (fst p) ⊆ set-btree (Node t n)*
⟨*proof*⟩

lemma *some-child-sub*:
assumes *(sub,sep) ∈ set t*
shows *sub ∈ set (subtrees t)*
and *sep ∈ set (separators t)*
⟨*proof*⟩

lemma *bal-all-subtrees-equal*: *bal (Node ts t) ⟹ (∀ s1 ∈ set (subtrees ts). ∀ s2 ∈ set (subtrees ts). height s1 = height s2)*
⟨*proof*⟩

lemma *fold-max-set*: $\forall x \in set\ t.\ x = f \Longrightarrow fold\ max\ t\ f = f$
⟨*proof*⟩

lemma *height-bal-tree*: $bal\ (Node\ ts\ t) \Longrightarrow height\ (Node\ ts\ t) = Suc\ (height\ t)$
⟨*proof*⟩

lemma *bal-split-last*:
assumes $bal\ (Node\ (ls@(sub,sep)\#rs)\ t)$
shows $bal\ (Node\ (ls@rs)\ t)$
and $height\ (Node\ (ls@(sub,sep)\#rs)\ t) = height\ (Node\ (ls@rs)\ t)$
⟨*proof*⟩

lemma *bal-split-right*:
assumes $bal\ (Node\ (ls@rs)\ t)$
shows $bal\ (Node\ rs\ t)$
and $height\ (Node\ rs\ t) = height\ (Node\ (ls@rs)\ t)$
⟨*proof*⟩

lemma *bal-split-left*:
assumes $bal\ (Node\ (ls@(a,b)\#rs)\ t)$
shows $bal\ (Node\ ls\ a)$
and $height\ (Node\ ls\ a) = height\ (Node\ (ls@(a,b)\#rs)\ t)$
⟨*proof*⟩

lemma *bal-substitute*: $[\![bal\ (Node\ (ls@(a,b)\#rs)\ t);\ height\ t = height\ c;\ bal\ c]\!] \Longrightarrow bal\ (Node\ (ls@(c,b)\#rs)\ t)$
⟨*proof*⟩

lemma *bal-substitute-subtree*: $[\![bal\ (Node\ (ls@(a,b)\#rs)\ t);\ height\ a = height\ c;\ bal\ c]\!] \Longrightarrow bal\ (Node\ (ls@(c,b)\#rs)\ t)$
⟨*proof*⟩

lemma *bal-substitute-separator*: $bal\ (Node\ (ls@(a,b)\#rs)\ t) \Longrightarrow bal\ (Node\ (ls@(a,c)\#rs)\ t)$
⟨*proof*⟩

lemma *order-impl-root-order*: $[\![k > 0;\ order\ k\ t]\!] \Longrightarrow root\text{-}order\ k\ t$
⟨*proof*⟩

lemma *sorted-inorder-list-separators*: *sorted-less* (*inorder-list ts*) $\Longrightarrow$ *sorted-less* (*separators ts*)
⟨*proof*⟩

corollary *sorted-inorder-separators*: *sorted-less* (*inorder* (*Node ts t*)) $\Longrightarrow$ *sorted-less* (*separators ts*)
⟨*proof*⟩

lemma *sorted-inorder-list-subtrees*:
sorted-less (*inorder-list ts*) $\Longrightarrow \forall$ *sub* $\in$ *set* (*subtrees ts*). *sorted-less* (*inorder sub*)
⟨*proof*⟩

corollary *sorted-inorder-subtrees*: *sorted-less* (*inorder* (*Node ts t*)) $\Longrightarrow \forall$ *sub* $\in$ *set* (*subtrees ts*). *sorted-less* (*inorder sub*)
⟨*proof*⟩

lemma *sorted-inorder-list-induct-subtree*:
sorted-less (*inorder-list* (*ls*@(*sub*,*sep*)#*rs*)) $\Longrightarrow$ *sorted-less* (*inorder sub*)
⟨*proof*⟩

corollary *sorted-inorder-induct-subtree*:
sorted-less (*inorder* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)) $\Longrightarrow$ *sorted-less* (*inorder sub*)
⟨*proof*⟩

lemma *sorted-inorder-induct-last*: *sorted-less* (*inorder* (*Node ts t*)) $\Longrightarrow$ *sorted-less* (*inorder t*)
⟨*proof*⟩

end
theory *BTree-Height*
imports *BTree*
begin

2 Maximum and minimum height

Textbooks usually provide some proofs relating the maxmimum and minimum height of the BTree for a given number of nodes. We therefore introduce this counting and show the respective proofs.

2.1 Definition of node/size

thm *BTree.btree.size*

value *size* (*Node* [(*Leaf*, (*0*::*nat*)), (*Node* [(*Leaf*, *1*), (*Leaf*, *10*)] *Leaf*, *12*), (*Leaf*, *30*), (*Leaf*, *100*)] *Leaf*)

The default size function does not suit our needs as it regards the length of the list in each node. We would like to count the number of nodes in the tree only, not regarding the number of keys.

fun *nodes*::*'a btree* ⇒ *nat* **where**
nodes Leaf = *0* |
nodes (*Node ts t*) = *1* + (∑ *t*←*subtrees ts*. *nodes t*) + (*nodes t*)

value *nodes* (*Node* [(*Leaf*, (*0*::*nat*)), (*Node* [(*Leaf*, *1*), (*Leaf*, *10*)] *Leaf*, *12*), (*Leaf*, *30*), (*Leaf*, *100*)] *Leaf*)

2.2 Maximum number of nodes for a given height

lemma *sum-list-replicate*: *sum-list* (*replicate n c*) = *n*∗*c*
⟨*proof*⟩

abbreviation *bound k h* ≡ ((*k*+*1*)^*h* − *1*)

lemma *nodes-height-upper-bound*:
⟦*order k t*; *bal t*⟧ ⟹ *nodes t* ∗ (*2*∗*k*) ≤ *bound* (*2*∗*k*) (*height t*)
⟨*proof*⟩

To verify our lower bound is sharp, we compare it to the height of artificially constructed full trees.

fun *full-node*::*nat* ⇒ *'a* ⇒ *nat* ⇒ *'a btree* **where**
full-node k c 0 = *Leaf*|
full-node k c (*Suc n*) = (*Node* (*replicate* (*2*∗*k*) ((*full-node k c n*),*c*)) (*full-node k c n*))

value *let k* = (*2*::*nat*) *in map* (λ*x*. *nodes x* ∗ *2*∗*k*) (*map* (*full-node k* (*1*::*nat*)) [*0*,*1*,*2*,*3*,*4*])
value *let k* = (*2*::*nat*) *in map* (λ*x*. ((*2*∗*k*+(*1*::*nat*))^(*x*)−*1*)) [*0*,*1*,*2*,*3*,*4*]

lemma *compow-comp-id*: *c* > *0* ⟹ *f* ∘ *f* = *f* ⟹ (*f* ^^ *c*) = *f*
⟨*proof*⟩

lemma *compow-id-point*: *f x* = *x* ⟹ (*f* ^^ *c*) *x* = *x*
⟨*proof*⟩

lemma *height-full-node*: *height* (*full-node k a h*) = *h*
⟨*proof*⟩

lemma *bal-full-node*: *bal* (*full-node k a h*)
⟨*proof*⟩

lemma *order-full-node*: *order k* (*full-node k a h*)
⟨*proof*⟩

lemma *full-btrees-sharp*: *nodes* (*full-node k a h*) ∗ (*2*∗*k*) = *bound* (*2*∗*k*) *h*

⟨*proof*⟩

lemma *upper-bound-sharp-node*:
t = *full-node k a h* ⟹ *height t* = *h* ∧ *order k t* ∧ *bal t* ∧ *bound* (2∗*k*) *h* = *nodes t* ∗ (2∗*k*)
⟨*proof*⟩

2.3 Maximum height for a given number of nodes

lemma *nodes-height-lower-bound*:
⟦*order k t*; *bal t*⟧ ⟹ *bound k* (*height t*) ≤ *nodes t* ∗ *k*
⟨*proof*⟩

To verify our upper bound is sharp, we compare it to the height of artificially constructed minimally filled (=slim) trees.

fun *slim-node*::*nat* ⇒ *'a* ⇒ *nat* ⇒ *'a btree* **where**
slim-node k c 0 = *Leaf*|
slim-node k c (*Suc n*) = (*Node* (*replicate k* ((*slim-node k c n*),*c*)) (*slim-node k c n*))

value *let k* = (2::*nat*) *in map* (λ*x*. *nodes x* ∗ *k*) (*map* (*slim-node k* (1::*nat*)) [0,1,2,3,4])
value *let k* = (2::*nat*) *in map* (λ*x*. ((*k*+1::*nat*)^(*x*)−1)) [0,1,2,3,4]

lemma *height-slim-node*: *height* (*slim-node k a h*) = *h*
⟨*proof*⟩

lemma *bal-slim-node*: *bal* (*slim-node k a h*)
⟨*proof*⟩

lemma *order-slim-node*: *order k* (*slim-node k a h*)
⟨*proof*⟩

lemma *slim-nodes-sharp*: *nodes* (*slim-node k a h*) ∗ *k* = *bound k h*
⟨*proof*⟩

lemma *lower-bound-sharp-node*:
t = *slim-node k a h* ⟹ *height t* = *h* ∧ *order k t* ∧ *bal t* ∧ *bound k h* = *nodes t* ∗ *k*
⟨*proof*⟩

Since BTrees have special roots, we need to show the overall nodes seperately

lemma *nodes-root-height-lower-bound*:
assumes *root-order k t*
and *bal t*
shows 2∗((*k*+1)^(*height t* − 1) − 1) + (*of-bool* (*t* ≠ *Leaf*))∗*k* ≤ *nodes t* ∗ *k*
⟨*proof*⟩

lemma *nodes-root-height-upper-bound*:
assumes *root-order k t*
and *bal t*
shows *nodes t ∗ (2∗k) ≤ (2∗k+1)^(height t) − 1*
⟨*proof*⟩

lemma *root-order-imp-divmuleq*: *root-order k t ⟹ (nodes t ∗ k) div k = nodes t*
⟨*proof*⟩

lemma *nodes-root-height-lower-bound-simp*:
assumes *root-order k t*
and *bal t*
and *k > 0*
shows *(2∗((k+1)^(height t − 1) − 1)) div k + (of-bool (t ≠ Leaf)) ≤ nodes t*
⟨*proof*⟩

lemma *nodes-root-height-upper-bound-simp*:
assumes *root-order k t*
and *bal t*
shows *nodes t ≤ ((2∗k+1)^(height t) − 1) div (2∗k)*
⟨*proof*⟩

definition *full-tree = full-node*

fun *slim-tree* **where**
slim-tree k c 0 = Leaf |
slim-tree k c (Suc h) = Node [(slim-node k c h, c)] (slim-node k c h)

lemma *lower-bound-sharp*:
k > 0 ⟹ t = slim-tree k a h ⟹ height t = h ∧ root-order k t ∧ bal t ∧ nodes t ∗ k = 2∗((k+1)^(height t − 1) − 1) + (of-bool (t ≠ Leaf))∗k
⟨*proof*⟩

lemma *upper-bound-sharp*:
k > 0 ⟹ t = full-tree k a h ⟹ height t = h ∧ root-order k t ∧ bal t ∧ ((2∗k+1)^(height t) − 1) = nodes t ∗ (2∗k)
⟨*proof*⟩

end
theory *BTree-Set*
imports *BTree*
HOL−Data-Structures.Set-Specs
begin

3 Set interpretation

3.1 Auxiliary functions

fun *split-half*:: ('*a btree*×'*a*) *list* ⇒ (('*a btree*×'*a*) *list* × ('*a btree*×'*a*) *list*) **where**
split-half xs = (*take* (*length xs div 2*) *xs*, *drop* (*length xs div 2*) *xs*)

lemma *drop-not-empty*: *xs* ≠ [] ⟹ *drop* (*length xs div 2*) *xs* ≠ []
⟨*proof*⟩

lemma *split-half-not-empty*: *length xs* ≥ *1* ⟹ ∃ *ls sub sep rs*. *split-half xs* = (*ls*,(*sub*,*sep*)#*rs*)
⟨*proof*⟩

3.2 The split function locale

Here, we abstract away the inner workings of the split function for B-tree operations.

locale *split* =
fixes *split* :: ('*a btree*×'*a*::*linorder*) *list* ⇒ '*a* ⇒ (('*a btree*×'*a*) *list* × ('*a btree*×'*a*) *list*)
assumes *split-req*:
⟦*split xs p* = (*ls*,*rs*)⟧ ⟹ *xs* = *ls* @ *rs*
⟦*split xs p* = (*ls*@[(*sub*,*sep*)],*rs*); *sorted-less* (*separators xs*)⟧ ⟹ *sep* < *p*
⟦*split xs p* = (*ls*,(*sub*,*sep*)#*rs*); *sorted-less* (*separators xs*)⟧ ⟹ *p* ≤ *sep*
begin

lemmas *split-conc* = *split-req*(*1*)
lemmas *split-sorted* = *split-req*(*2*,*3*)

lemma [*termination-simp*]:(*ls*, (*sub*, *sep*) # *rs*) = *split ts y* ⟹
size sub < *Suc* (*size-list* (λ*x*. *Suc* (*size* (*fst x*))) *ts* + *size l*)
⟨*proof*⟩

fun *invar-inorder* **where** *invar-inorder k t* = (*bal t* ∧ *root-order k t*)

definition *empty-btree* = *Leaf*

3.3 Membership

fun *isin*:: '*a btree* ⇒ '*a* ⇒ *bool* **where**
isin (*Leaf*) *y* = *False* |
isin (*Node ts t*) *y* = (
case split ts y of (-,(*sub*,*sep*)#*rs*) ⇒ (
if y = *sep then*
True

```
        else
          isin sub y
    )
  | (-,[]) ⇒ isin t y
 )
```

3.4 Insertion

The insert function requires an auxiliary data structure and auxiliary invariant functions.

```
datatype 'b up_i = T_i 'b btree | Up_i 'b btree 'b 'b btree

fun order-up_i where
  order-up_i k (T_i sub) = order k sub |
  order-up_i k (Up_i l a r) = (order k l ∧ order k r)

fun root-order-up_i where
  root-order-up_i k (T_i sub) = root-order k sub |
  root-order-up_i k (Up_i l a r) = (order k l ∧ order k r)


fun height-up_i where
  height-up_i (T_i t) = height t |
  height-up_i (Up_i l a r) = max (height l) (height r)

fun bal-up_i where
  bal-up_i (T_i t) = bal t |
  bal-up_i (Up_i l a r) = (height l = height r ∧ bal l ∧ bal r)

fun inorder-up_i where
  inorder-up_i (T_i t) = inorder t |
  inorder-up_i (Up_i l a r) = inorder l @ [a] @ inorder r
```

The following function merges two nodes and returns separately split nodes if an overflow occurs

```
fun node_i:: nat ⇒ ('a btree × 'a) list ⇒ 'a btree ⇒ 'a up_i where
  node_i k ts t = (
  if length ts ≤ 2*k then T_i (Node ts t)
  else (
    case split-half ts of (ls, (sub,sep)#rs) ⇒
      Up_i (Node ls sub) sep (Node rs t)
    )
  )


fun ins:: nat ⇒ 'a ⇒ 'a btree ⇒ 'a up_i where
  ins k x Leaf = (Up_i Leaf x Leaf) |
  ins k x (Node ts t) = (
```

```
  case split ts x of
    (ls,(sub,sep)#rs) ⇒
      (if sep = x then
        T_i (Node ts t)
      else
        (case ins k x sub of
          Up_i l a r ⇒
            node_i k (ls @ (l,a)#(r,sep)#rs) t |
          T_i a ⇒
            T_i (Node (ls @ (a,sep) # rs) t))) |
    (ls, []) ⇒
      (case ins k x t of
        Up_i l a r ⇒
          node_i k (ls@[(l,a)]) r |
        T_i a ⇒
          T_i (Node ls a)
  )
)
```

```
fun tree_i::'a up_i ⇒ 'a btree where
  tree_i (T_i sub) = sub |
  tree_i (Up_i l a r) = (Node [(l,a)] r)

fun insert::nat ⇒ 'a ⇒ 'a btree ⇒ 'a btree where
  insert k x t = tree_i (ins k x t)
```

3.5 Deletion

The following deletion method is inspired by Bayer (70) and Fielding (??). Rather than stealing only a single node from the neighbour, the neighbour is fully merged with the potentially underflowing node. If the resulting node is still larger than allowed, the merged node is split again, using the rules known from insertion splits. If the resulting node has admissable size, it is simply kept in the tree.

```
fun rebalance-middle-tree where
  rebalance-middle-tree k ls Leaf sep rs Leaf = (
  Node (ls@(Leaf,sep)#rs) Leaf
) |
  rebalance-middle-tree k ls (Node mts mt) sep rs (Node tts tt) = (
  if length mts ≥ k ∧ length tts ≥ k then
    Node (ls@(Node mts mt,sep)#rs) (Node tts tt)
  else (
    case rs of [] ⇒ (
      case node_i k (mts@(mt,sep)#tts) tt of
        T_i u ⇒
          Node ls u |
```

```
      Up_i l a r ⇒
       Node (ls@[(l,a)]) r) |
    (Node rts rt,rsep)#rs ⇒ (
      case node_i k (mts@(mt,sep)#rts) rt of
      T_i u ⇒
        Node (ls@(u,rsep)#rs) (Node tts tt) |
      Up_i l a r ⇒
        Node (ls@(l,a)#(r,rsep)#rs) (Node tts tt))
))
```

Deletion

All trees are merged with the right neighbour on underflow. Obviously for the last tree this would not work since it has no right neighbour. Therefore this tree, as the only exception, is merged with the left neighbour. However since we it does not make a difference, we treat the situation as if the second to last tree underflowed.

```
fun rebalance-last-tree where
  rebalance-last-tree k ts t = (
case last ts of (sub,sep) ⇒
   rebalance-middle-tree k (butlast ts) sub sep [] t
)
```

Rather than deleting the minimal key from the right subtree, we remove the maximal key of the left subtree. This is due to the fact that the last tree can easily be accessed and the left neighbour is way easier to access than the right neighbour, it resides in the same pair as the separating element to be removed.

```
fun split-max where
  split-max k (Node ts t) = (case t of Leaf ⇒ (
  let (sub,sep) = last ts in
    (Node (butlast ts) sub, sep)
)|
- ⇒
case split-max k t of (sub, sep) ⇒
  (rebalance-last-tree k ts sub, sep)
)

fun del where
  del k x Leaf = Leaf |
  del k x (Node ts t) = (
  case split ts x of
    (ls,[]) ⇒
     rebalance-last-tree k ls (del k x t)
  | (ls,(sub,sep)#rs) ⇒ (
      if sep ≠ x then
        rebalance-middle-tree k ls (del k x sub) sep rs t
      else if sub = Leaf then
```

```
      Node (ls@rs) t
    else let (sub-s, max-s) = split-max k sub in
      rebalance-middle-tree k ls sub-s max-s rs t
  )
)

fun reduce-root where
  reduce-root Leaf = Leaf |
  reduce-root (Node ts t) = (case ts of
   [] ⇒ t |
   - ⇒ (Node ts t)
)


fun delete where delete k x t = reduce-root (del k x t)
```

An invariant for intermediate states at deletion. In particular we allow for an underflow to 0 subtrees.

```
fun almost-order where
  almost-order k Leaf = True |
  almost-order k (Node ts t) = (
  (length ts ≤ 2∗k) ∧
  (∀ s ∈ set (subtrees ts). order k s) ∧
   order k t
)
```

A recursive property of the "spine" we want to walk along for splitting off the maximum of the left subtree.

```
fun nonempty-lasttreebal where
  nonempty-lasttreebal Leaf = True |
  nonempty-lasttreebal (Node ts t) = (
    (∃ ls tsub tsep. ts = (ls@[(tsub,tsep)]) ∧ height tsub = height t) ∧
     nonempty-lasttreebal t
  )
```

3.6 Proofs of functional correctness

```
lemma split-set:
  assumes split ts z = (ls,(a,b)#rs)
  shows (a,b) ∈ set ts
    and (x,y) ∈ set ls ⟹ (x,y) ∈ set ts
    and (x,y) ∈ set rs ⟹ (x,y) ∈ set ts
    and set ls ∪ set rs ∪ {(a,b)} = set ts
    and ∃ x ∈ set ts. b ∈ Basic-BNFs.snds x
  ⟨proof⟩

lemma split-length:
  split ts x = (ls, rs) ⟹ length ls + length rs = length ts
  ⟨proof⟩
```

Isin proof

thm *isin-simps*

lemma *sorted-ConsD*: *sorted-less* $(y \# xs) \Longrightarrow x \leq y \Longrightarrow x \notin set\ xs$
⟨*proof*⟩

lemma *sorted-snocD*: *sorted-less* $(xs\ @\ [y]) \Longrightarrow y \leq x \Longrightarrow x \notin set\ xs$
⟨*proof*⟩

lemmas *isin-simps2* = *sorted-lems sorted-ConsD sorted-snocD*

lemma *isin-sorted*: *sorted-less* $(xs@a\#ys) \Longrightarrow$
$(x \in set\ (xs@a\#ys)) = (if\ x < a\ then\ x \in set\ xs\ else\ x \in set\ (a\#ys))$
⟨*proof*⟩

lemma *isin-sorted-split*:
assumes *sorted-less* (*inorder* (*Node ts t*))
and *split ts x* = (*ls*, *rs*)
shows $x \in set\ (inorder\ (Node\ ts\ t)) = (x \in set\ (inorder\text{-}list\ rs\ @\ inorder\ t))$
⟨*proof*⟩

lemma *isin-sorted-split-right*:
assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
and *sorted-less* (*inorder* (*Node ts t*))
and $sep \neq x$
shows $x \in set\ (inorder\text{-}list\ ((sub,sep)\#rs)\ @\ inorder\ t) = (x \in set\ (inorder\ sub))$
⟨*proof*⟩

theorem *isin-set-inorder*: *sorted-less* $(inorder\ t) \Longrightarrow isin\ t\ x = (x \in set\ (inorder\ t))$
⟨*proof*⟩

lemma $node_i$*-cases*: *length xs* $\leq k \vee (\exists\ ls\ sub\ sep\ rs.\ split\text{-}half\ xs = (ls,(sub,sep)\#rs))$
⟨*proof*⟩

lemma $root\text{-}order\text{-}tree_i$: $root\text{-}order\text{-}up_i\ (Suc\ k)\ t = root\text{-}order\ (Suc\ k)\ (tree_i\ t)$
⟨*proof*⟩

lemma *node$_i$-root-order*:
assumes *length ts* > *0*
and *length ts* ≤ *4∗k+1*
and ∀ *x* ∈ *set* (*subtrees ts*). *order k x*
and *order k t*
shows *root-order-up$_i$ k* (*node$_i$ k ts t*)
⟨*proof*⟩

lemma *node$_i$-order-helper*:
assumes *length ts* ≥ *k*
and *length ts* ≤ *4∗k+1*
and ∀ *x* ∈ *set* (*subtrees ts*). *order k x*
and *order k t*
shows *case* (*node$_i$ k ts t*) *of T$_i$ t* ⇒ *order k t* | - ⇒ *True*
⟨*proof*⟩

lemma *node$_i$-order*:
assumes *length ts* ≥ *k*
and *length ts* ≤ *4∗k+1*
and ∀ *x* ∈ *set* (*subtrees ts*). *order k x*
and *order k t*
shows *order-up$_i$ k* (*node$_i$ k ts t*)

⟨*proof*⟩

lemma *ins-order*:
order k t ⟹ *order-up$_i$ k* (*ins k x t*)
⟨*proof*⟩

lemma *ins-root-order*:
assumes *root-order k t*
shows *root-order-up$_i$ k* (*ins k x t*)
⟨*proof*⟩

lemma *height-list-split*: *height-up$_i$* (*Up$_i$* (*Node ls a*) *b* (*Node rs t*)) = *height* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*)
⟨*proof*⟩

lemma *node$_i$-height*: *height-up$_i$* (*node$_i$ k ts t*) = *height* (*Node ts t*)
⟨*proof*⟩

lemma *bal-up$_i$-tree*: *bal-up$_i$ t = bal (tree$_i$ t)*
⟨*proof*⟩

lemma *bal-list-split*: *bal (Node (ls@(a,b)#rs) t) ⟹ bal-up$_i$ (Up$_i$ (Node ls a) b (Node rs t))*
⟨*proof*⟩

lemma *node$_i$-bal*:
assumes *bal (Node ts t)*
shows *bal-up$_i$ (node$_i$ k ts t)*
⟨*proof*⟩

lemma *height-up$_i$-merge*: *height-up$_i$ (Up$_i$ l a r) = height t ⟹ height (Node (ls@(t,x)#rs) tt) = height (Node (ls@(l,a)#(r,x)#rs) tt)*
⟨*proof*⟩

lemma *ins-height*: *height-up$_i$ (ins k x t) = height t*
⟨*proof*⟩

lemma *ins-bal*: *bal t ⟹ bal-up$_i$ (ins k x t)*
⟨*proof*⟩

lemma *node$_i$-inorder*: *inorder-up$_i$ (node$_i$ k ts t) = inorder (Node ts t)*
⟨*proof*⟩

corollary *node$_i$-inorder-simps*:
node$_i$ k ts t = T$_i$ t' ⟹ inorder t' = inorder (Node ts t)
node$_i$ k ts t = Up$_i$ l a r ⟹ inorder l @ a # inorder r = inorder (Node ts t)
⟨*proof*⟩

lemma *ins-sorted-inorder*: *sorted-less (inorder t) ⟹ (inorder-up$_i$ (ins k (x::('a::linorder)) t)) = ins-list x (inorder t)*
⟨*proof*⟩

lemma *ins-list-split*:
assumes *split ts x = (ls, rs)*
and *sorted-less (inorder (Node ts t))*
shows *ins-list x (inorder (Node ts t)) = inorder-list ls @ ins-list x (inorder-list rs @ inorder t)*
⟨*proof*⟩

lemma *ins-list-split-right-general*:
assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
and *sorted-less* (*inorder-list ts*)
and *sep* ≠ *x*
shows *ins-list x* (*inorder-list* ((*sub*,*sep*)#*rs*) @ *zs*) = *ins-list x* (*inorder sub*) @ *sep* # *inorder-list rs* @ *zs*
⟨*proof*⟩

corollary *ins-list-split-right*:
assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
and *sorted-less* (*inorder* (*Node ts t*))
and *sep* ≠ *x*
shows *ins-list x* (*inorder-list* ((*sub*,*sep*)#*rs*) @ *inorder t*) = *ins-list x* (*inorder sub*) @ *sep* # *inorder-list rs* @ *inorder t*
⟨*proof*⟩

lemma *ins-list-idem-eq-isin*: *sorted-less xs* ⟹ *x* ∈ *set xs* ⟷ (*ins-list x xs* = *xs*)
⟨*proof*⟩

lemma *ins-list-contains-idem*: ⟦*sorted-less xs*; *x* ∈ *set xs*⟧ ⟹ (*ins-list x xs* = *xs*)
⟨*proof*⟩

declare *node*$_i$.*simps* [*simp del*]
declare *node*$_i$-*inorder* [*simp add*]

lemma *ins-inorder*: *sorted-less* (*inorder t*) ⟹ (*inorder-up*$_i$ (*ins k x t*)) = *ins-list x* (*inorder t*)
⟨*proof*⟩

declare *node*$_i$.*simps* [*simp add*]
declare *node*$_i$-*inorder* [*simp del*]

thm *ins.induct*
thm *btree.induct*

lemma *tree*$_i$-*bal*: *bal-up*$_i$ *u* ⟹ *bal* (*tree*$_i$ *u*)
⟨*proof*⟩

lemma *tree*$_i$-*order*: ⟦*k* > *0*; *root-order-up*$_i$ *k u*⟧ ⟹ *root-order k* (*tree*$_i$ *u*)
⟨*proof*⟩

lemma *tree$_i$-inorder*: *inorder-up$_i$ u = inorder (tree$_i$ u)*
⟨*proof*⟩

lemma *insert-bal*: *bal t ⟹ bal (insert k x t)*
⟨*proof*⟩

lemma *insert-order*: *⟦k > 0; root-order k t⟧ ⟹ root-order k (insert k x t)*
⟨*proof*⟩

lemma *insert-inorder*: *sorted-less (inorder t) ⟹ inorder (insert k x t) = ins-list x (inorder t)*
⟨*proof*⟩

Deletion proofs

thm *list.simps*

lemma *rebalance-middle-tree-height*:
assumes *height t = height sub*
and *case rs of (rsub,rsep) # list ⇒ height rsub = height t | [] ⇒ True*
shows *height (rebalance-middle-tree k ls sub sep rs t) = height (Node (ls@(sub,sep)#rs) t)*
⟨*proof*⟩

lemma *rebalance-last-tree-height*:
assumes *height t = height sub*
and *ts = list@[(sub,sep)]*
shows *height (rebalance-last-tree k ts t) = height (Node ts t)*
⟨*proof*⟩

lemma *split-max-height*:
assumes *split-max k t = (sub,sep)*
and *nonempty-lasttreebal t*
and *t ≠ Leaf*
shows *height sub = height t*
⟨*proof*⟩

lemma *order-bal-nonempty-lasttreebal*: *⟦k > 0; root-order k t; bal t⟧ ⟹ nonempty-lasttreebal t*
⟨*proof*⟩

lemma *bal-sub-height*: *bal (Node (ls@a#rs) t) ⟹ (case rs of [] ⇒ True | (sub,sep)#- ⇒ height sub = height t)*
⟨*proof*⟩

lemma *del-height*: *⟦k > 0; root-order k t; bal t⟧ ⟹ height (del k x t) = height t*

⟨*proof*⟩

lemma *rebalance-middle-tree-inorder*:
assumes *height t* = *height sub*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
shows *inorder* (*rebalance-middle-tree k ls sub sep rs t*) = *inorder* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
⟨*proof*⟩

lemma *rebalance-last-tree-inorder*:
assumes *height t* = *height sub*
and *ts* = *list*@[(*sub*,*sep*)]
shows *inorder* (*rebalance-last-tree k ts t*) = *inorder* (*Node ts t*)
⟨*proof*⟩

lemma *butlast-inorder-app-id*: *xs* = *xs'* @ [(*sub*,*sep*)] ⟹ *inorder-list xs'* @ *inorder sub* @ [*sep*] = *inorder-list xs*
⟨*proof*⟩

lemma *split-max-inorder*:
assumes *nonempty-lasttreebal t*
and *t* ≠ *Leaf*
shows *inorder-pair* (*split-max k t*) = *inorder t*
⟨*proof*⟩

lemma *height-bal-subtrees-merge*: ⟦*height* (*Node as a*) = *height* (*Node bs b*); *bal* (*Node as a*); *bal* (*Node bs b*)⟧
⟹ ∀ *x* ∈ *set* (*subtrees as*) ∪ {*a*}. *height x* = *height b*
⟨*proof*⟩

lemma *bal-list-merge*:
assumes *bal-up*$_i$ (*Up*$_i$ (*Node as a*) *x* (*Node bs b*))
shows *bal* (*Node* (*as*@(*a*,*x*)#*bs*) *b*)
⟨*proof*⟩

lemma *node*$_i$*-bal-up*$_i$:
assumes *bal-up*$_i$ (*node*$_i$ *k ts t*)
shows *bal* (*Node ts t*)
⟨*proof*⟩

lemma *node*$_i$*-bal-simp*: *bal-up*$_i$ (*node*$_i$ *k ts t*) = *bal* (*Node ts t*)
⟨*proof*⟩

lemma *rebalance-middle-tree-bal*: *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) ⟹ *bal* (*rebalance-middle-tree k ls sub sep rs t*)

⟨*proof*⟩

lemma *rebalance-last-tree-bal*: ⟦*bal* (*Node ts t*); *ts* ≠ []⟧ ⟹ *bal* (*rebalance-last-tree k ts t*)
⟨*proof*⟩

lemma *split-max-bal*:
assumes *bal t*
and *t* ≠ *Leaf*
and *nonempty-lasttreebal t*
shows *bal* (*fst* (*split-max k t*))
⟨*proof*⟩

lemma *del-bal*:
assumes *k* > *0*
and *root-order k t*
and *bal t*
shows *bal* (*del k x t*)
⟨*proof*⟩

lemma *rebalance-middle-tree-order*:
assumes *almost-order k sub*
and ∀ *s* ∈ *set* (*subtrees* (*ls*@*rs*)). *order k s order k t*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
and *length* (*ls*@(*sub*,*sep*)#*rs*) ≤ *2*∗*k*
and *height sub* = *height t*
shows *almost-order k* (*rebalance-middle-tree k ls sub sep rs t*)
⟨*proof*⟩

lemma *rebalance-middle-tree-last-order*:
assumes *almost-order k t*
and ∀ *s* ∈ *set* (*subtrees* (*ls*@(*sub*,*sep*)#*rs*)). *order k s*
and *rs* = []
and *length* (*ls*@(*sub*,*sep*)#*rs*) ≤ *2*∗*k*
and *height sub* = *height t*
shows *almost-order k* (*rebalance-middle-tree k ls sub sep rs t*)
⟨*proof*⟩

lemma *rebalance-last-tree-order*:
assumes *ts* = *ls*@[(*sub*,*sep*)]
and ∀ *s* ∈ *set* (*subtrees* (*ts*)). *order k s almost-order k t*
and *length ts* ≤ *2*∗*k*
and *height sub* = *height t*
shows *almost-order k* (*rebalance-last-tree k ts t*)
⟨*proof*⟩

lemma *split-max-order*:
assumes *order k t*
and $t \neq Leaf$
and *nonempty-lasttreebal t*
shows *almost-order k* (*fst* (*split-max k t*))
⟨*proof*⟩

lemma *del-order*:
assumes $k > 0$
and *root-order k t*
and *bal t*
shows *almost-order k* (*del k x t*)
⟨*proof*⟩

thm *del-list-sorted*

lemma *del-list-split*:
assumes *split ts x* = (*ls*, *rs*)
and *sorted-less* (*inorder* (*Node ts t*))
shows *del-list x* (*inorder* (*Node ts t*)) = *inorder-list ls* @ *del-list x* (*inorder-list rs* @ *inorder t*)
⟨*proof*⟩

lemma *del-list-split-right*:
assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
and *sorted-less* (*inorder* (*Node ts t*))
and $sep \neq x$
shows *del-list x* (*inorder-list* ((*sub*,*sep*)#*rs*) @ *inorder t*) = *del-list x* (*inorder sub*) @ *sep* # *inorder-list rs* @ *inorder t*
⟨*proof*⟩

thm *del-list-idem*

lemma *del-inorder*:
assumes $k > 0$
and *root-order k t*
and *bal t*
and *sorted-less* (*inorder t*)
shows *inorder* (*del k x t*) = *del-list x* (*inorder t*)
⟨*proof*⟩

lemma *reduce-root-order*: ⟦$k > 0$; *almost-order k t*⟧ ⟹ *root-order k* (*reduce-root t*)

⟨*proof*⟩

lemma *reduce-root-bal*: *bal* (*reduce-root t*) = *bal t*
⟨*proof*⟩

lemma *reduce-root-inorder*: *inorder* (*reduce-root t*) = *inorder t*
⟨*proof*⟩

lemma *delete-order*: ⟦*k* > *0*; *bal t*; *root-order k t*⟧ ⟹ *root-order k* (*delete k x t*)
⟨*proof*⟩

lemma *delete-bal*: ⟦*k* > *0*; *bal t*; *root-order k t*⟧ ⟹ *bal* (*delete k x t*)
⟨*proof*⟩

lemma *delete-inorder*: ⟦*k* > *0*; *bal t*; *root-order k t*; *sorted-less* (*inorder t*)⟧ ⟹
inorder (*delete k x t*) = *del-list x* (*inorder t*)
⟨*proof*⟩

3.7 Set specification by inorder

interpretation *S-ordered*: *Set-by-Ordered* **where**
empty = *empty-btree* **and**
insert = *insert* (*Suc k*) **and**
delete = *delete* (*Suc k*) **and**
isin = *isin* **and**
inorder = *inorder* **and**
inv = *invar-inorder* (*Suc k*)
⟨*proof*⟩

declare *node*$_i$.*simps*[*simp del*]

end

end
theory *BTree-Split*
imports *BTree-Set*
begin

4 Abstract split functions

4.1 Linear split

Finally we show that the split axioms are feasible by providing an example split function

fun *linear-split-help*:: (-×'*a*::*linorder*) *list* ⇒ - ⇒ (-×-) *list* ⇒ ((-×-) *list* × (-×-) *list*) **where**
linear-split-help [] *x prev* = (*prev*, []) |
linear-split-help ((*sub*, *sep*)#*xs*) *x prev* = (*if sep* < *x then linear-split-help xs x* (*prev* @ [(*sub*, *sep*)]) *else* (*prev*, (*sub*,*sep*)#*xs*))

fun *linear-split*:: (-×'*a*::*linorder*) *list* ⇒ - ⇒ ((-×-) *list* × (-×-) *list*) **where**
linear-split xs x = *linear-split-help xs x* []

Linear split is similar to well known functions, therefore a quick proof can be done.

lemma *linear-split-alt*: *linear-split xs x* = (*takeWhile* (λ(-,*s*). *s*<*x*) *xs*, *dropWhile* (λ(-,*s*). *s*<*x*) *xs*)
⟨*proof*⟩

global-interpretation *btree-linear-search*: *split linear-split*

defines *btree-ls-isin* = *btree-linear-search.isin*
and *btree-ls-ins* = *btree-linear-search.ins*
and *btree-ls-insert* = *btree-linear-search.insert*
and *btree-ls-del* = *btree-linear-search.del*
and *btree-ls-delete* = *btree-linear-search.delete*
⟨*proof*⟩

Some examples follow to show that the implementation works and the above lemmas make sense. The examples are visualized in the thesis.

abbreviation $btree_i \equiv$ *btree-ls-insert*
abbreviation $btree_d \equiv$ *btree-ls-delete*

value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
root-order k x
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
bal x
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
sorted-less (*inorder x*)
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
x
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
$btree_i$ *k 9 x*
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
$btree_i$ *k 1* ($btree_i$ *k 9 x*)
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*

btree$_d$ k 10 (btree$_i$ k 1 (btree$_i$ k 9 x))
value *let k=2::nat; x::nat btree = (Node [(Leaf, 3),(Leaf, 5),(Leaf, 6)] Leaf, 10)] (Node [(Leaf, 14), (Leaf, 20)] Leaf)) in*
btree$_d$ k 3 (btree$_d$ k 10 (btree$_i$ k 1 (btree$_i$ k 9 x)))

For completeness, we also proved an explicit proof of the locale requirements.

lemma *some-child-sm*: *linear-split-help t y xs = (ls,(sub,sep)#rs) ⟹ y ≤ sep*
⟨*proof*⟩

lemma *linear-split-append*: *linear-split-help xs p ys = (ls,rs) ⟹ ls@rs = ys@xs*
⟨*proof*⟩

lemma *linear-split-sm*: ⟦*linear-split-help xs p ys = (ls,rs)*; *sorted-less (separators (ys@xs))*; ∀ *sep* ∈ *set (separators ys). p > sep*⟧ ⟹ ∀ *sep* ∈ *set (separators ls). p > sep*
⟨*proof*⟩

value *linear-split [((Leaf::nat btree), 2)] (1::nat)*

lemma *linear-split-gr*:
⟦*linear-split-help xs p ys = (ls,rs)*; *sorted-less (separators (ys@xs))*; ∀ *(sub,sep) ∈ set ys. p > sep*⟧ ⟹
(case rs of [] ⇒ True | (-,sep)#- ⇒ p ≤ sep)
⟨*proof*⟩

lemma *linear-split-req*:
assumes *linear-split xs p = (ls,(sub,sep)#rs)*
and *sorted-less (separators xs)*
shows *p ≤ sep*
⟨*proof*⟩

lemma *linear-split-req2*:
assumes *linear-split xs p = (ls@[(sub,sep)],rs)*
and *sorted-less (separators xs)*
shows *sep < p*
⟨*proof*⟩

interpretation *split linear-split*
⟨*proof*⟩

4.2 Binary split

It is possible to define a binary split predicate. However, even proving that it terminates is uncomfortable.

function (*sequential*) *binary-split-help*:: (-×′*a*::*linorder*) *list* ⇒ (-×′*a*) *list* ⇒ (-×′*a*) *list* ⇒ ′*a* ⇒ ((-×-) *list* × (-×-) *list*) **where**
binary-split-help ls [] *rs x* = (*ls*,*rs*) |
binary-split-help ls as rs x = (*let* (*mls*, *mrs*) = *split-half as in* (
case mrs of (*sub*,*sep*)#*mrrs* ⇒ (
if x < *sep then binary-split-help ls mls* (*mrs*@*rs*) *x*
else if x > *sep then binary-split-help* (*ls*@*mls*@[(*sub*,*sep*)]) *mrrs rs x*
else (*ls*@*mls*, *mrs*@*rs*)
)
)
)
⟨*proof*⟩
termination
⟨*proof*⟩

fun *binary-split* **where**
binary-split as x = *binary-split-help* [] *as* [] *x*

We can show that it will return sublists that concatenate to the original list again but will not show that it fulfils sortedness properties.

lemma *binary-split-help as bs cs x* = (*ls*,*rs*) ⟹ (*as*@*bs*@*cs*) = (*ls*@*rs*)
⟨*proof*⟩

lemma ⟦*sorted-less* (*separators* (*as*@*bs*@*cs*)); *binary-split-help as bs cs x* = (*ls*,*rs*); ∀ *y* ∈ *set* (*separators as*). *y* < *x*⟧
⟹ ∀ *y* ∈ *set* (*separators ls*). *y* < *x*
⟨*proof*⟩

end
theory *Array-SBlit*
imports *Separation-Logic-Imperative-HOL.Array-Blit*
begin

5 Same array Blit

The standard framework already provides a function to copy array elements.

term *blit*
thm *blit-rule*
thm *blit.simps*

definition *sblit a s d l* ≡ *blit a s a d l*

When copying values within arrays, blit only works for moving elements to the left.

lemma *sblit-rule*[*sep-heap-rules*]:

assumes *LEN*:
si+len ≤ *length lsrc*
and *DST-SM*: *di* ≤ *si*
shows
< *src* ↦$_a$ *lsrc* >
sblit src si di len
<λ-. *src* ↦$_a$ (*take di lsrc* @ *take len* (*drop si lsrc*) @ *drop* (*di+len*) *lsrc*)
>
⟨*proof*⟩

5.1 A reverse blit

The function rblit may be used to copy elements a defined offset to the right

primrec *rblit* :: - *array* ⇒ *nat* ⇒ - *array* ⇒ *nat* ⇒ *nat* ⇒ *unit Heap* **where**
rblit - - - - *0* = *return* ()
| *rblit src si dst di* (*Suc l*) = *do* {
x ← *Array.nth src* (*si+l*);
Array.upd (*di+l*) *x dst*;
rblit src si dst di l
}

For separated arrays it is equivalent to normal blit. The proof follows similarly to the corresponding proof for blit.

lemma *rblit-rule*[*sep-heap-rules*]:
assumes *LEN*: *si+len* ≤ *length lsrc di+len* ≤ *length ldst*
shows
< *src* ↦$_a$ *lsrc*
∗ *dst* ↦$_a$ *ldst* >
rblit src si dst di len
<λ-. *src* ↦$_a$ *lsrc*
∗ *dst* ↦$_a$ (*take di ldst* @ *take len* (*drop si lsrc*) @ *drop* (*di+len*) *ldst*)
>
⟨*proof*⟩

definition *srblit a s d l* ≡ *rblit a s a d l*

However, within arrays we can now copy to the right.

lemma *srblit-rule*[*sep-heap-rules*]:
assumes *LEN*:
di+len ≤ *length lsrc*
and *DST-GR*: *di* ≥ *si*
shows
< *src* ↦$_a$ *lsrc* >
srblit src si di len
<λ-. *src* ↦$_a$ (*take di lsrc* @ *take len* (*drop si lsrc*) @ *drop* (*di+len*) *lsrc*)
>
⟨*proof*⟩

5.2 Modeling target language blit

For convenience, a function that is oblivious to the direction of the shift is defined.

definition *safe-sblit a s d l* ≡
  if s > d then
    sblit a s d l
  else
    srblit a s d l

We obtain a heap rule similar to the one of blit, but for copying within one array.

lemma *safe-sblit-rule*[*sep-heap-rules*]:
  assumes *LEN*:
    len > 0 ⟶ *di+len* ≤ *length lsrc* ∧ *si+len* ≤ *length lsrc*
  shows
    < *src* ↦$_a$ *lsrc* >
    safe-sblit src si di len
    <λ-. *src* ↦$_a$ (*take di lsrc* @ *take len* (*drop si lsrc*) @ *drop* (*di+len*) *lsrc*)
    >
  ⟨*proof*⟩

thm *blit-rule*
thm *safe-sblit-rule*

5.3 Code Generator Setup

Note that the requirement for correctness is even weaker here than in SML. We therefore manually handle the case where length is 0 (in which case nothing happens at all).

code-printing code-module *array-sblit* ⇀ (*SML*)
  ‹
   fun array-sblit src si di len = (
     if len > 0 then
       ArraySlice.copy {
         di = *IntInf.toInt di*,
         src = *ArraySlice.slice* (*src*,*IntInf.toInt si*,*SOME* (*IntInf.toInt len*)),
         dst = *src*}
     else ()
)
›

definition *safe-sblit′* **where**
  [*code del*]: *safe-sblit′ src si di len*
    = *safe-sblit src* (*nat-of-integer si*) (*nat-of-integer di*)
      (*nat-of-integer len*)

lemma [*code*]:
safe-sblit src si di len
= *safe-sblit′ src* (*integer-of-nat si*) (*integer-of-nat di*)
(*integer-of-nat len*) ⟨*proof*⟩

code-printing constant *safe-sblit′* ⇀
(*SML*) (*fn*/ ()/ => /*array′-sblit* - - - -)
and (*Scala*) { ('-: *Unit*)/=>/
def safescopy(*src*: *Array*['-], *srci*: *Int*, *dsti*: *Int*, *len*: *Int*) = {
if (*len* > *0*)
System.arraycopy(*src*, *srci*, *src*, *dsti*, *len*)
else
()
}
safescopy(-.*array*,-.*toInt*,-.*toInt*,-.*toInt*)
}

export-code *safe-sblit* **checking** *SML Scala*

5.4 Derived operations

definition *array-shr* **where**
array-shr a i k ≡ *do* {
l ← *Array.len a*;
safe-sblit a i (*i*+*k*) (*l*−(*i*+*k*))
}

find-theorems *Array.len*

lemma *array-shr-rule*[*sep-heap-rules*]:
< *src* ↦$_a$ *lsrc* >
array-shr src i k
<λ-. *src* ↦$_a$ (*take* (*i*+*k*) *lsrc* @ *take* (*length lsrc* − (*i*+*k*)) (*drop i lsrc*))
>
⟨*proof*⟩

lemma *array-shr-rule-alt*:
< *src* ↦$_a$ *lsrc* >
array-shr src i k
<λ-. *src* ↦$_a$ (*take* (*length lsrc*) (*take* (*i*+*k*) *lsrc* @ (*drop i lsrc*)))
>
⟨*proof*⟩

definition *array-shl* **where**
array-shl a i k ≡ *do* {
l ← *Array.len a*;
safe-sblit a i (*i*−*k*) (*l*−*i*)

}

lemma *array-shl-rule*[*sep-heap-rules*]:

$< src \mapsto_a lsrc >$
array-shl src i k
$<\lambda$-. $src \mapsto_a (take\ (i-k)\ lsrc\ @\ (drop\ i\ lsrc)\ @\ drop\ (i - k + (length\ lsrc - i))\ lsrc)$
$>$
⟨*proof*⟩

lemma *array-shl-rule-alt*:

$[\![i \leq length\ lsrc;\ k \leq i]\!] \Longrightarrow$
$< src \mapsto_a lsrc >$
array-shl src i k
$<\lambda$-. $src \mapsto_a (take\ (i-k)\ lsrc\ @\ (drop\ i\ lsrc)\ @\ drop\ (length\ lsrc - k)\ lsrc)$
$>$
⟨*proof*⟩

end
theory *Partially-Filled-Array*
imports
Refine-Imperative-HOL.IICF-Array-List
Array-SBlit
begin

6 Partially Filled Arrays

An array that is only partially filled. The number of actual elements contained is kept in a second element. This represents a weakened version of the array_list from IICF.

type-synonym *'a pfarray* = *'a array-list*

6.1 Operations on Partly Filled Arrays

definition *is-pfa* **where**
is-pfa c l ≡ $\lambda(a,n).\ \exists_A\ l'.\ a \mapsto_a l' *\ \uparrow(c = length\ l' \wedge n \leq c \wedge l = (take\ n\ l'))$

lemma *is-pfa-prec*[*safe-constraint-rules*]: *precise* (*is-pfa c*)
⟨*proof*⟩

definition *pfa-init* **where**
pfa-init cap v n ≡ *do* {

a ← *Array.new cap v*;
return (*a*,*n*)
}

lemma *pfa-init-rule*[*sep-heap-rules*]: *n* ≤ *N* ⟹ < *emp* > *pfa-init N x n* <*is-pfa N* (*replicate n x*)>
⟨*proof*⟩

definition *pfa-empty* **where**
pfa-empty cap ≡ *pfa-init cap default 0*

lemma *pfa-empty-rule*[*sep-heap-rules*]: < *emp* > *pfa-empty N* <*is-pfa N* []>
⟨*proof*⟩

definition *pfa-length* ≡ *arl-length*

lemma *pfa-length-rule*[*sep-heap-rules*]:
<*is-pfa c l a*>
pfa-length a
<λ*r*. *is-pfa c l a* ∗ ↑(*r*=*length l*)>
⟨*proof*⟩

definition *pfa-capacity* ≡ λ(*a*,*n*). *Array.len a*

lemma *pfa-capacity-rule*[*sep-heap-rules*]:
<*is-pfa c l a*>
pfa-capacity a
<λ*r*. *is-pfa c l a* ∗ ↑(*c*=*r*)>
⟨*proof*⟩

definition *pfa-is-empty* ≡ *arl-is-empty*

lemma *pfa-is-empty-rule*[*sep-heap-rules*]:
<*is-pfa c l a*>
pfa-is-empty a
<λ*r*. *is-pfa c l a* ∗ ↑(*r*⟷(*l*=[]))>
⟨*proof*⟩

definition *pfa-append* ≡ λ(*a*,*n*) *x*. *do* {

Array.upd n x a;
return (*a*,*n*+*1*)
}

lemma *pfa-append-rule*[*sep-heap-rules*]:
$n < c \Longrightarrow$
< *is-pfa c l* (*a*,*n*) >
pfa-append (*a*,*n*) *x*
<λ(*a'*,*n'*). *is-pfa c* (*l*@[*x*]) (*a'*,*n'*) ∗ ↑(*a'* = *a* ∧ *n'* = *n*+*1*) >
⟨*proof*⟩

definition *pfa-last* ≡ *arl-last*

lemma *pfa-last-rule*[*sep-heap-rules*]:
l≠[] ⟹
<*is-pfa c l a*>
pfa-last a
<λ*r*. *is-pfa c l a* ∗ ↑(*r*=*last l*)>
⟨*proof*⟩

definition *pfa-butlast* :: *'a*::*heap pfarray* ⇒ *'a pfarray Heap* **where**
pfa-butlast ≡ λ(*a*,*n*).
return (*a*,*n*−*1*)

lemma *pfa-butlast-rule*[*sep-heap-rules*]:
<*is-pfa c l* (*a*,*n*)>
pfa-butlast (*a*,*n*)
<λ(*a'*,*n'*). *is-pfa c* (*butlast l*) (*a'*,*n'*) ∗ ↑(*a'* = *a*)>
⟨*proof*⟩

definition *pfa-get* ≡ *arl-get*

lemma *pfa-get-rule*[*sep-heap-rules*]:
i < *length l* ⟹
< *is-pfa c l a*>
pfa-get a i
<λ*r*. *is-pfa c l a* ∗ ↑((*l*!*i*) = *r*)>
⟨*proof*⟩

definition *pfa-set* ≡ *arl-set*

lemma *pfa-set-rule*[*sep-heap-rules*]:

i<$length\ l \Longrightarrow$
<$is\text{-}pfa\ c\ l\ a$>
$pfa\text{-}set\ a\ i\ x$
<$\lambda a'.\ is\text{-}pfa\ c\ (l[i:=x])\ a' * \uparrow(a' = a)$>
⟨*proof*⟩

definition *pfa-shrink* :: *nat* ⇒ *'a*::*heap pfarray* ⇒ *'a pfarray Heap* **where**
pfa-shrink k ≡ λ(*a*,*n*).
return (*a*,*k*)

lemma *pfa-shrink-rule*[*sep-heap-rules*]:
$k \le length\ l \Longrightarrow$
< *is-pfa c l* (*a*,*n*) >
pfa-shrink k (*a*,*n*)
<λ(*a'*,*n'*). *is-pfa c* (*take k l*) (*a'*,*n'*) ∗ ↑(*n'* = *k* ∧ *a'*=*a*) >
⟨*proof*⟩

definition *pfa-shrink-cap* :: *nat* ⇒ *'a*::*heap pfarray* ⇒ *'a pfarray Heap* **where**
pfa-shrink-cap k ≡ λ(*a*,*n*). *do* {
a' ← *array-shrink a k*;
return (*a'*,*min k n*)
}

lemma *pfa-shrink-cap-rule-preserve*[*sep-heap-rules*]:
⟦*n* ≤ *k*; *k* ≤ *c*⟧ ⟹
< *is-pfa c l* (*a*,*n*) >
pfa-shrink-cap k (*a*,*n*)
<λ*a'*. *is-pfa k l a'* >$_t$
⟨*proof*⟩

lemma *pfa-shrink-cap-rule*:
⟦*k* ≤ *c*⟧ ⟹
< *is-pfa c l a* >
pfa-shrink-cap k a
<λ*a'*. *is-pfa k* (*take k l*) *a'* >$_t$
⟨*proof*⟩

definition *array-ensure a s x* ≡ *do* {

```
    l←Array.len a;
    if l≥s then
      return a
    else do {
      a'←Array.new s x;
      blit a 0 a' 0 l;
      return a'
    }
  }

lemma array-ensure-rule[sep-heap-rules]:
  shows
      < a↦_a la >
        array-ensure a s x
      <λa'. a'↦_a (la @ replicate (s−length la) x)>_t
  ⟨proof⟩


definition pfa-ensure :: 'a::{heap,default} pfarray ⇒ nat ⇒ 'a pfarray Heap where
  pfa-ensure ≡ λ(a,n) k. do {
  a' ← array-ensure a k default;
  return (a',n)
}


lemma pfa-ensure-rule[sep-heap-rules]:
    < is-pfa c l (a,n) >
      pfa-ensure (a,n) k
    <λ(a',n'). is-pfa (max c k) l (a',n') ∗ ↑(n' = n ∧ c ≥ n)>_t
  ⟨proof⟩


definition pfa-copy ≡ arl-copy


lemma pfa-copy-rule[sep-heap-rules]:
  < is-pfa c l a >
   pfa-copy a
   <λr. is-pfa c l a ∗ is-pfa c l r>_t
  ⟨proof⟩

definition pfa-blit :: 'a::heap pfarray ⇒ nat ⇒ 'a::heap pfarray ⇒ nat ⇒ nat ⇒
unit Heap where
  pfa-blit ≡ λ(src,sn) si (dst,dn) di l. blit src si dst di l


lemma min-nat: min a (a+b) = (a::nat)
  ⟨proof⟩
```

lemma *pfa-blit-rule*[*sep-heap-rules*]:
assumes *LEN*: *si*+*len* ≤ *sn* *di* ≤ *dn* *di*+*len* ≤ *dc*
shows
< *is-pfa sc src* (*srci*,*sn*)
∗ *is-pfa dc dst* (*dsti*,*dn*) >
pfa-blit (*srci*,*sn*) *si* (*dsti*,*dn*) *di len*
<λ-. *is-pfa sc src* (*srci*,*sn*)
∗ *is-pfa dc* (*take di dst* @ *take len* (*drop si src*) @ *drop* (*di*+*len*) *dst*) (*dsti*,*max* (*di*+*len*) *dn*)
>
⟨*proof*⟩

definition *pfa-drop* :: ('*a*::*heap*) *pfarray* ⇒ *nat* ⇒ '*a pfarray* ⇒ '*a pfarray Heap* **where**
pfa-drop ≡ λ(*src*,*sn*) *si* (*dst*,*dn*). *do* {
blit src si dst 0 (*sn*−*si*);
return (*dst*,(*sn*−*si*))
}

lemma *pfa-drop-rule*[*sep-heap-rules*]:
assumes *LEN*: *k* ≤ *sn* (*sn*−*k*) ≤ *dc*
shows
< *is-pfa sc src* (*srci*,*sn*)
∗ *is-pfa dc dst* (*dsti*,*dn*) >
pfa-drop (*srci*,*sn*) *k* (*dsti*,*dn*)
<λ(*dsti*′,*dn*′). *is-pfa sc src* (*srci*,*sn*)
∗ *is-pfa dc* (*drop k src*) (*dsti*′,*dn*′)
∗ ↑(*dsti*′ = *dsti*)
>
⟨*proof*⟩

definition *pfa-append-grow* ≡ λ(*a*,*n*) *x*. *do* {
l ← *pfa-capacity* (*a*,*n*);
a′ ← *if l* = *n*
then array-grow a (*l*+*1*) *x*
else Array.upd n x a;
return (*a*′,*n*+*1*)
}

lemma *pfa-append-grow-full-rule*[*sep-heap-rules*]: *n* = *c* ⟹
<*is-pfa c l* (*a*,*n*)>
array-grow a (*c*+*1*) *x*
<λ*a*′. *is-pfa* (*c*+*1*) (*l*@[*x*]) (*a*′,*n*+*1*)>$_t$

⟨*proof*⟩

lemma *pfa-append-grow-less-rule*: $n < c \Longrightarrow$
<*is-pfa c l (a,n)*>
Array.upd n x a
<$\lambda a'$. *is-pfa c (l@[x]) (a',n+1)*>$_t$
⟨*proof*⟩

lemma *pfa-append-grow-rule*[*sep-heap-rules*]:
<*is-pfa c l (a,n)*>
pfa-append-grow (a,n) x
<$\lambda(a',n')$. *is-pfa (if c = n then c+1 else c) (l@[x]) (a',n')* $*$ $\uparrow(n'=n+1 \wedge c \geq n)$>$_t$
⟨*proof*⟩

definition *pfa-append-grow'* $\equiv \lambda(a,n)\ x.$ *do* {
$a' \leftarrow$ *pfa-ensure (a,n) (n+1)*;
$a'' \leftarrow$ *pfa-append a' x*;
return a''
}

lemma *pfa-append-grow'-rule*[*sep-heap-rules*]:
<*is-pfa c l (a,n)*>
pfa-append-grow' (a,n) x
<$\lambda(a',n')$. *is-pfa (max (n+1) c) (l@[x]) (a',n')* $*$ $\uparrow(n'=n+1 \wedge c \geq n)$>$_t$
⟨*proof*⟩

definition *pfa-insert* $\equiv \lambda(a,n)\ i\ x.$ *do* {
$a' \leftarrow$ *array-shr a i 1*;
$a'' \leftarrow$ *Array.upd i x a*;
return (a'',n+1)
}

lemma *list-update-last*: *length ls = Suc i* $\Longrightarrow$ *ls*[*i*:=*x*] = (*take i ls*)@[*x*]
⟨*proof*⟩

lemma *pfa-insert-rule*[*sep-heap-rules*]:
$[\![i \leq n;\ n < c]\!] \Longrightarrow$
<*is-pfa c l (a,n)*>
pfa-insert (a,n) i x
<$\lambda(a',n')$. *is-pfa c (take i l@x#drop i l) (a',n')* $*$ $\uparrow(n' = n+1 \wedge a=a')$>
⟨*proof*⟩

definition *pfa-insert-grow* :: *'a*::{*heap,default*} *pfarray* $\Rightarrow$ *nat* $\Rightarrow$ *'a* $\Rightarrow$ *'a pfarray Heap*

where *pfa-insert-grow* ≡ λ(*a*,*n*) *i x*. *do* {
a′ ← *pfa-ensure* (*a*,*n*) (*n*+*1*);
a″ ← *pfa-insert a′ i x*;
return a″
}

lemma *pfa-insert-grow-rule*:
i ≤ *n* ⟹
<*is-pfa c l* (*a*,*n*)>
pfa-insert-grow (*a*,*n*) *i x*
<λ(*a′*,*n′*). *is-pfa* (*max c* (*n*+*1*)) (*take i l*@*x*#*drop i l*) (*a′*,*n′*) ∗ ↑(*n′*=*n*+*1* ∧ *c* ≥ *n*)>$_t$
⟨*proof*⟩

definition *pfa-extend* **where**
pfa-extend ≡ λ (*a*,*n*) (*b*,*m*). *do*{
blit b 0 a n m;
return (*a*,*n*+*m*)
}

lemma *pfa-extend-rule*:
n+*m* ≤ *c* ⟹
<*is-pfa c l1* (*a*,*n*) ∗ *is-pfa d l2* (*b*,*m*)>
pfa-extend (*a*,*n*) (*b*,*m*)
<λ(*a′*,*n′*). *is-pfa c* (*l1*@*l2*) (*a′*,*n′*) ∗ ↑(*a′* = *a* ∧ *n′*=*n*+*m*) ∗ *is-pfa d l2* (*b*,*m*)>$_t$
⟨*proof*⟩

definition *pfa-extend-grow* **where**
pfa-extend-grow ≡ λ (*a*,*n*) (*b*,*m*). *do*{
a′ ← *array-ensure a* (*n*+*m*) *default*;
blit b 0 a′ n m;
return (*a′*,*n*+*m*)
}

lemma *pfa-extend-grow-rule*:
<*is-pfa c l1* (*a*,*n*) ∗ *is-pfa d l2* (*b*,*m*)>
pfa-extend-grow (*a*,*n*) (*b*,*m*)
<λ(*a′*,*n′*). *is-pfa* (*max c* (*n*+*m*)) (*l1*@*l2*) (*a′*,*n′*) ∗ ↑(*n′*=*n*+*m* ∧ *c* ≥ *n*) ∗ *is-pfa d l2* (*b*,*m*)>$_t$
⟨*proof*⟩

definition *pfa-append-extend-grow* **where**
pfa-append-extend-grow ≡ λ (*a*,*n*) *x* (*b*,*m*). *do*{
a′ ← *array-ensure a* (*n*+*m*+*1*) *default*;
a″ ← *Array.upd n x a′*;
blit b 0 a″ (*n*+*1*) *m*;
return (*a″*,*n*+*m*+*1*)

}

lemma *pfa-append-extend-grow-rule*:
<*is-pfa c l1 (a,n) ∗ is-pfa d l2 (b,m)*>
pfa-append-extend-grow (a,n) x (b,m)
<*λ(a′,n′). is-pfa (max c (n+m+1)) (l1@x#l2) (a′,n′) ∗ ↑(n′=n+m+1 ∧ c ≥ n) ∗ is-pfa d l2 (b,m)*>$_t$
⟨*proof*⟩

definition *pfa-delete* ≡ *λ(a,n) i. do* {
array-shl a (i+1) 1;
return (a,n−1)
}

lemma *pfa-delete-rule*[*sep-heap-rules*]:
i < n ⟹
<*is-pfa c l (a,n)*>
pfa-delete (a,n) i
<*λ(a′,n′). is-pfa c (take i l@drop (i+1) l) (a′,n′) ∗ ↑(n′ = n−1 ∧ a=a′)*>
⟨*proof*⟩

end
theory *Basic-Assn*
imports
Refine-Imperative-HOL.Sepref-HOL-Bindings
Refine-Imperative-HOL.Sepref-Basic
begin

7 Auxilary imperative assumptions

The following auxiliary assertion types and lemmas were provided by Peter Lammich

7.1 List-Assn

lemma *list-assn-cong*[*fundef-cong*]:
⟦ *xs=xs′*; *ys=ys′*; ⋀*x y*. ⟦ *x∈set xs*; *y∈set ys* ⟧ ⟹ *A x y = A′ x y* ⟧ ⟹ *list-assn A xs ys = list-assn A′ xs′ ys′*
⟨*proof*⟩

lemma *list-assn-app-one*: *list-assn P (l1@*[*x*]*) (l1′@*[*y*]*) = list-assn P l1 l1′ ∗ P x y*
⟨*proof*⟩

lemma *list-assn-len*: $h \models$ *list-assn A xs ys* $\Longrightarrow$ *length xs* = *length ys*
⟨*proof*⟩

lemma *list-assn-append-Cons-left*: *list-assn A* (*xs*@*x*#*ys*) *zs* = (∃$_A$ *zs1 z zs2*. *list-assn A xs zs1* ∗ *A x z* ∗ *list-assn A ys zs2* ∗ ↑(*zs1*@*z*#*zs2* = *zs*))
⟨*proof*⟩

lemma *list-assn-aux-append-Cons*:
shows *length xs* = *length zsl* $\Longrightarrow$ *list-assn A* (*xs*@*x*#*ys*) (*zsl*@*z*#*zsr*) = (*list-assn A xs zsl* ∗ *A x z* ∗ *list-assn A ys zsr*)
⟨*proof*⟩

7.2 Prod-Assn

lemma *prod-assn-cong*[*fundef-cong*]:
⟦ *p*=*p*′; *pi*=*pi*′; *A* (*fst p*) (*fst pi*) = *A*′ (*fst p*) (*fst pi*); *B* (*snd p*) (*snd pi*) = *B*′ (*snd p*) (*snd pi*) ⟧
$\Longrightarrow$ (*A*×$_a$*B*) *p pi* = (*A*′×$_a$*B*′) *p*′ *pi*′
⟨*proof*⟩

end
theory *BTree-Imp*
imports
BTree
Partially-Filled-Array
Basic-Assn
begin

8 Imperative B-tree Definition

The heap data type definition. Anything stored on the heap always contains data, leafs are represented as None.

datatype *'a btnode* =
Btnode (*'a btnode ref option*∗*'a*) *pfarray 'a btnode ref option*

Selector Functions

primrec *kvs* :: *'a*::*heap btnode* ⇒ (*'a btnode ref option*∗*'a*) *pfarray* **where**
[*sep-dflt-simps*]: *kvs* (*Btnode ts* -) = *ts*

primrec *last* :: *'a*::*heap btnode* ⇒ *'a btnode ref option* **where**
[*sep-dflt-simps*]: *last* (*Btnode* - *t*) = *t*

term *arrays-update*

Encoding to natural numbers, as required by Imperative/HOL

fun
btnode-encode :: *'a::heap btnode* ⇒ *nat*
where
btnode-encode (*Btnode ts t*) = *to-nat* (*ts*, *t*)

instance *btnode* :: (*heap*) *heap*
⟨*proof*⟩

The refinement relationship to abstract B-trees.

fun *btree-assn* :: *nat* ⇒ *'a::heap btree* ⇒ *'a btnode ref option* ⇒ *assn* **where**
btree-assn k Leaf None = *emp* |
btree-assn k (*Node ts t*) (*Some a*) =
(∃$_A$ *tsi ti tsi'*.
a ↦$_r$ *Btnode tsi ti*
∗ *btree-assn k t ti*
∗ *is-pfa* (*2∗k*) *tsi' tsi*
∗ *list-assn* ((*btree-assn k*) ×$_a$ *id-assn*) *ts tsi'*
) |
btree-assn - - - = *false*

With the current definition of deletion, we would also need to directly reason on nodes and not only on references to them.

fun *btnode-assn* :: *nat* ⇒ *'a::heap btree* ⇒ *'a btnode* ⇒ *assn* **where**
btnode-assn k (*Node ts t*) (*Btnode tsi ti*) =
(∃$_A$ *tsi'*.
btree-assn k t ti
∗ *is-pfa* (*2∗k*) *tsi' tsi*
∗ *list-assn* ((*btree-assn k*) ×$_a$ *id-assn*) *ts tsi'*
) |
btnode-assn - - - = *false*

abbreviation *blist-assn k* ≡ *list-assn* ((*btree-assn k*) ×$_a$ *id-assn*)

end
theory *BTree-ImpSet*
imports
BTree-Imp
BTree-Set
begin

9 Imperative Set operations

9.1 Auxiliary operations

definition *split-relation xs* ≡
λ(*as*,*bs*) *i*. *i* ≤ *length xs* ∧ *as* = *take i xs* ∧ *bs* = *drop i xs*

lemma *split-relation-alt*:
split-relation as (*ls*,*rs*) *i* = (*as* = *ls*@*rs* ∧ *i* = *length ls*)
⟨*proof*⟩

lemma *split-relation-length*: *split-relation xs* (*ls*,*rs*) (*length xs*) = (*ls* = *xs* ∧ *rs* = [])
⟨*proof*⟩

lemma *list-assn-prod-map*: *list-assn* (*A* ×$_a$ *B*) *xs ys* = *list-assn B* (*map snd xs*) (*map snd ys*) ∗ *list-assn A* (*map fst xs*) (*map fst ys*)
⟨*proof*⟩

lemma *id-assn-list*: *h* ⊨ *list-assn id-assn* (*xs*::'*a list*) *ys* ⟹ *xs* = *ys*
⟨*proof*⟩

lemma *snd-map-help*:
x ≤ *length tsi* ⟹
(∀ *j*<*x*. *snd* (*tsi* ! *j*) = ((*map snd tsi*)!*j*))
x < *length tsi* ⟹ *snd* (*tsi*!*x*) = ((*map snd tsi*)!*x*)
⟨*proof*⟩

lemma *split-ismeq*: ((*a*::*nat*) ≤ *b* ∧ *X*) = ((*a* < *b* ∧ *X*) ∨ (*a* = *b* ∧ *X*))
⟨*proof*⟩

lemma *split-relation-map*: *split-relation as* (*ls*,*rs*) *i* ⟹ *split-relation* (*map f as*) (*map f ls*, *map f rs*) *i*
⟨*proof*⟩

lemma *split-relation-access*: ⟦*split-relation as* (*ls*,*rs*) *i*; *rs* = *r*#*rrs*⟧ ⟹ *as*!*i* = *r*
⟨*proof*⟩

lemma *index-to-elem-all*: (∀ *j*<*length xs*. *P* (*xs*!*j*)) = (∀ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemma *index-to-elem*: *n* < *length xs* ⟹ (∀ *j*<*n*. *P* (*xs*!*j*)) = (∀ *x* ∈ *set* (*take n xs*). *P x*)
⟨*proof*⟩

definition *split-half* :: ('*a*::*heap* × '*b*::{*heap*}) *pfarray* ⇒ *nat Heap*
where

```
  split-half a ≡ do {
  l ← pfa-length a;
  return (l div 2)
}

lemma split-half-rule[sep-heap-rules]: <
    is-pfa c tsi a
  * list-assn R ts tsi>
    split-half a
  <λi.
      is-pfa c tsi a
    * list-assn R ts tsi
    * ↑(i = length ts div 2 ∧  split-relation ts (BTree-Set.split-half ts) i)>
  ⟨proof⟩
```

9.2 The imperative split locale

This locale extends the abstract split locale, assuming that we are provided with an imperative program that refines the abstract split function.

```
locale imp-split = abs-split: BTree-Set.split split
  for split::
    ('a btree × 'a::{heap,default,linorder}) list ⇒ 'a
      ⇒ ('a btree × 'a) list × ('a btree × 'a) list +
  fixes imp-split:: ('a btnode ref option × 'a::{heap,default,linorder}) pfarray ⇒ 'a
⇒ nat Heap
  assumes imp-split-rule [sep-heap-rules]:sorted-less (separators ts) ⟹
   <is-pfa c tsi (a,n)
  * blist-assn k ts tsi>
    imp-split (a,n) p
  <λi.
    is-pfa c tsi (a,n)
    * blist-assn k ts tsi
    * ↑(split-relation ts (split ts p) i)>t
begin
```

9.3 Membership

```
partial-function (heap) isin :: 'a btnode ref option ⇒ 'a ⇒  bool Heap
  where
    isin p x =
  (case p of
     None ⇒ return False |
     (Some a) ⇒ do {
       node ← !a;
       i ← imp-split (kvs node) x;
       tsl ← pfa-length (kvs node);
       if i < tsl then do {
         s ← pfa-get (kvs node) i;
         let (sub,sep) = s in
```

```
        if x = sep then
          return True
        else
          isin sub x
      } else
          isin (last node) x
    }
)
```

9.4 Insertion

```
datatype 'b btupi =
  T_i 'b btnode ref option |
  Up_i 'b btnode ref option 'b 'b btnode ref option

fun btupi-assn where
  btupi-assn k (abs-split.T_i l) (T_i li) =
   btree-assn k l li |
  btupi-assn k (abs-split.Up_i l a r) (Up_i li ai ri) =
   btree-assn k l li * id-assn a ai * btree-assn k r ri |
  btupi-assn - - - = false



definition node_i :: nat ⇒ ('a btnode ref option × 'a) pfarray ⇒ 'a btnode ref
option ⇒ 'a btupi Heap where
  node_i k a ti ≡ do {
    n ← pfa-length a;
    if n ≤ 2*k then do {
      a' ← pfa-shrink-cap (2*k) a;
      l ← ref (Btnode a' ti);
      return (T_i (Some l))
    }
    else do {
      b ← (pfa-empty (2*k) :: ('a btnode ref option × 'a) pfarray Heap);
      i ← split-half a;
      m ← pfa-get a i;
      b' ← pfa-drop a (i+1) b;
      a' ← pfa-shrink i a;
      a'' ← pfa-shrink-cap (2*k) a';
      let (sub,sep) = m in do {
        l ← ref (Btnode a'' sub);
        r ← ref (Btnode b' ti);
        return (Up_i (Some l) sep (Some r))
      }
    }
}
```

```
partial-function (heap) ins :: nat ⇒ 'a ⇒ 'a btnode ref option ⇒ 'a btupi Heap
  where
    ins k x apo = (case apo of
  None ⇒
    return (Up_i None x None) |
  (Some ap) ⇒ do {
    a ← !ap;
    i ← imp-split (kvs a) x;
    tsl ← pfa-length (kvs a);
    if i < tsl then do {
      s ← pfa-get (kvs a) i;
      let (sub,sep) = s in
      if sep = x then
        return (T_i apo)
      else do {
        r ← ins k x sub;
        case r of
          (T_i lp) ⇒ do {
            pfa-set (kvs a) i (lp,sep);
            return (T_i apo)
          } |
          (Up_i lp x' rp) ⇒ do {
            pfa-set (kvs a) i (rp,sep);
            if tsl < 2*k then do {
                kvs' ← pfa-insert (kvs a) i (lp,x');
                ap := (Btnode kvs' (last a));
                return (T_i apo)
            } else do {
              kvs' ← pfa-insert-grow (kvs a) i (lp,x');
              node_i k kvs' (last a)
            }
          }
      }
    }
    else do {
      r ← ins k x (last a);
      case r of
        (T_i lp) ⇒ do {
          ap := (Btnode (kvs a) lp);
          return (T_i apo)
        } |
        (Up_i lp x' rp) ⇒
          if tsl < 2*k then do {
            kvs' ← pfa-append (kvs a) (lp,x');
            ap := (Btnode kvs' rp);
            return (T_i apo)
          } else do {
            kvs' ← pfa-append-grow' (kvs a) (lp,x');
            node_i k kvs' rp
```

```
      }
    }
  }
)
```

```
definition insert :: nat ⇒ ('a::{heap,default,linorder}) ⇒ 'a btnode ref option ⇒
'a btnode ref option Heap where
  insert ≡ λk x ti. do {
  ti' ← ins k x ti;
  case ti' of
     T_i sub ⇒ return sub |
     Up_i l a r ⇒ do {
       kvs ← pfa-init (2*k) (l,a) 1;
       t' ← ref (Btnode kvs r);
       return (Some t')
     }
}
```

9.5 Deletion

Note that the below operations have not been verified to refine the abstract set operations.

```
definition rebalance-middle-tree:: nat ⇒ (('a::{default,heap,linorder}) btnode ref
option × 'a) pfarray ⇒ nat ⇒ 'a btnode ref option ⇒ 'a btnode Heap
  where
    rebalance-middle-tree ≡ λ k tsi i r-ti. (
  case r-ti of
  None ⇒ do {
    return (Btnode tsi r-ti)
  } |
  Some p-t ⇒ do {
      ti ← !p-t;
      (r-sub,sep) ← pfa-get tsi i;
      case r-sub of (Some p-sub) ⇒  do {
      sub ← !p-sub;
      l-sub ← pfa-length (kvs sub);
      l-tts ← pfa-length (kvs ti);
      if l-sub ≥ k ∧ l-tts ≥ k then do {
        return (Btnode tsi r-ti)
      } else do {
        l-tsi ← pfa-length tsi;
        if l-tsi = i then do {
          mts' ← pfa-append-extend-grow (kvs sub) (last sub,sep) (kvs ti);
          res-node_i ← node_i k mts' (last ti);
          case res-node_i of
            T_i u ⇒ return (Btnode tsi u) |
```

```
            Up_i l a r ⇒ do {
              tsi' ← pfa-append tsi (l,a);
              return (Btnode tsi' r)
            }
          } else do {
            (r-rsub,rsep) ← pfa-get tsi (i+1);
            case r-rsub of Some p-rsub ⇒ do {
              rsub ← !p-rsub;
              mts' ← pfa-append-extend-grow (kvs sub) (last sub,sep) (kvs rsub);
              res-node_i ← node_i k mts' (last rsub);
              case res-node_i of
               T_i u ⇒ do {
                tsi' ← pfa-set tsi (i+1) (u,rsep);
                tsi'' ← pfa-delete tsi' i;
                return (Btnode tsi'' r-ti)
              } |
               Up_i l a r ⇒ do {
                tsi' ← pfa-set tsi i (l,a);
                tsi'' ← pfa-set tsi' (i+1) (r,rsep);
                return (Btnode tsi'' r-ti)
              }
            }
          }
        }
    }
})
```

definition *rebalance-last-tree*:: *nat* ⇒ ((*'a*::{*default*,*heap*,*linorder*}) *btnode ref option* × *'a*) *pfarray* ⇒ *'a btnode ref option* ⇒ *'a btnode Heap*
  where
    rebalance-last-tree ≡ λ*k tsi ti*. *do* {
   l-tsi ← *pfa-length tsi*;
   rebalance-middle-tree k tsi (*l-tsi*−*1*) *ti*
}

partial-function (*heap*) *split-max* ::*nat* ⇒ (*'a*::{*default*,*heap*,*linorder*}) *btnode ref option* ⇒ (*'a btnode ref option* × *'a*) *Heap*
  where
    split-max k r-t = (*case r-t of Some p-t* ⇒ *do* {
   t ← !*p-t*;
   (*case t of Btnode tsi r-ti* ⇒
     case r-ti of None ⇒ *do* {
      (*sub*,*sep*) ← *pfa-last tsi*;
      tsi' ← *pfa-butlast tsi*;
      p-t := *Btnode tsi' sub*;
      return (*Some p-t*, *sep*)
  } |

```
    - ⇒ do {
      (sub,sep) ← split-max k r-ti;
      p-t' ← rebalance-last-tree k tsi sub;
      p-t := p-t';
      return (Some p-t, sep)
  })
})
```

```
partial-function (heap) del ::nat ⇒ 'a ⇒ ('a::{default,heap,linorder}) btnode ref
option ⇒ 'a btnode ref option Heap
  where
    del k x ti = (case ti of None ⇒ return None |
   Some p ⇒ do {
   node ← !p;
   i ← imp-split (kvs node) x;
   tsl ← pfa-length (kvs node);
   if i < tsl then do {
     s ← pfa-get (kvs node) i;
     let (sub,sep) = s in
     if x ≠ sep then do {
       sub' ← del k x sub;
       kvs' ← pfa-set (kvs node) i (sub',sep);
       node' ← rebalance-middle-tree k kvs' i (last node);
       ti' ← ref node';
       return (Some ti')
      }
     else if sub = None then do{
       pfa-delete (kvs node) i;
       return ti
     }
     else do {
        sm ← split-max k sub;
        kvs' ← pfa-set (kvs node) i sm;
        node' ← rebalance-middle-tree k kvs' i (last node);
        ti' ← ref node';
        return (Some ti')
     }
   } else do {
        t' ← del k x (last node);
        node' ← rebalance-last-tree k (kvs node) t';
        ti' ← ref node';
        return (Some ti')
    }
})
```

```
partial-function (heap) reduce-root ::('a::{default,heap,linorder}) btnode ref op-
tion ⇒ 'a btnode ref option Heap
```

where
reduce-root ti = (*case ti of*
None ⇒ *return None* |
Some p-t ⇒ *do* {
node ← !*p-t*;
tsl ← *pfa-length* (*kvs node*);
case tsl of 0 ⇒ *return* (*last node*) |
- ⇒ *return ti*
})

partial-function (*heap*) *delete* ::*nat* ⇒ *'a* ⇒ (*'a*::{*default*,*heap*,*linorder*}) *btnode ref option* ⇒ *'a btnode ref option Heap*
where
delete k x ti = *do* {
ti' ← *del k x ti*;
reduce-root ti'
}

9.6 Refinement of the abstract B-tree operations

definition *empty* ::(*'a*::{*default*,*heap*,*linorder*}) *btnode ref option Heap*
where *empty* = *return None*

lemma *P-imp-Q-implies-P*: *P* ⟹ (*Q* ⟶ *P*)
⟨*proof*⟩

lemma *sorted-less* (*inorder t*) ⟹
<*btree-assn k t ti*>
isin ti x
<λ*r*. *btree-assn k t ti* ∗ ↑(*abs-split.isin t x* = *r*)>$_t$
⟨*proof*⟩

declare *abs-split.node$_i$.simps* [*simp add*]
lemma *node$_i$-rule*: **assumes** *c-cap*: *2∗k* ≤ *c c* ≤ *4∗k+1*
shows <*is-pfa c tsi* (*a*,*n*) ∗ *list-assn* ((*btree-assn k*) ×$_a$ *id-assn*) *ts tsi* ∗ *btree-assn k t ti*>
node$_i$ k (*a*,*n*) *ti*
<λ*r*. *btupi-assn k* (*abs-split.node$_i$ k ts t*) *r* >$_t$
⟨*proof*⟩
declare *abs-split.node$_i$.simps* [*simp del*]

lemma *node$_i$-no-split*: *length ts* ≤ *2∗k* ⟹ *abs-split.node$_i$ k ts t* = *abs-split.T$_i$* (*Node ts t*)
⟨*proof*⟩

lemma *node$_i$-rule-app*: $[\![2{*}k \leq c;\ c \leq 4{*}k{+}1]\!] \Longrightarrow$
$<$*is-pfa c (tsi′ @ [(li, ai)]) (aa, al) ∗*
blist-assn k ls tsi′ ∗
btree-assn k l li ∗
id-assn a ai ∗
btree-assn k r ri$>$ *node$_i$ k (aa, al) ri*
$<$*btupi-assn k (abs-split.node$_i$ k (ls @ [(l, a)]) r)*$>_t$
⟨*proof*⟩

lemma *node$_i$-rule-ins2*: $[\![2{*}k \leq c;\ c \leq 4{*}k{+}1;$ *length ls = length lsi*$]\!] \Longrightarrow$
$<$*is-pfa c (lsi @ (li, ai) # (ri,a′i) # rsi) (aa, al) ∗*
blist-assn k ls lsi ∗
btree-assn k l li ∗
id-assn a ai ∗
btree-assn k r ri ∗
id-assn a′ a′i ∗
blist-assn k rs rsi ∗
btree-assn k t ti$>$ *node$_i$ k (aa, al)*
ti $<$*btupi-assn k (abs-split.node$_i$ k (ls @ (l, a) # (r,a′) # rs) t)*$>_t$
⟨*proof*⟩

lemma *ins-rule*:
sorted-less (inorder t) $\Longrightarrow$ $<$*btree-assn k t ti*$>$
ins k x ti
$<\lambda r.$ *btupi-assn k (abs-split.ins k x t) r*$>_t$
⟨*proof*⟩

The imperative insert refines the abstract insert.

lemma *insert-rule*:
assumes *k > 0 sorted-less (inorder t)*
shows $<$*btree-assn k t ti*$>$
insert k x ti
$<\lambda r.$ *btree-assn k (abs-split.insert k x t) r*$>_t$
⟨*proof*⟩

The ”pure” resulting rule follows automatically.

lemma *insert-rule′*:
shows $<$*btree-assn (Suc k) t ti ∗ ↑(abs-split.invar-inorder (Suc k) t ∧ sorted-less (inorder t))*$>$
insert (Suc k) x ti
$<\lambda ri.\exists_A r.$ *btree-assn (Suc k) r ri ∗ ↑(abs-split.invar-inorder (Suc k) r ∧ sorted-less (inorder r) ∧ inorder r = (ins-list x (inorder t)))*$>_t$
⟨*proof*⟩

lemma *node$_i$-rule-ins*: $[\![2{*}k \leq c;\ c \leq 4{*}k{+}1;$ *length ls = length lsi*$]\!] \Longrightarrow$
$<$*is-pfa c (lsi @ (li, ai) # rsi) (aa, al) ∗*

blist-assn k ls lsi ∗
btree-assn k l li ∗
id-assn a ai ∗
blist-assn k rs rsi ∗
btree-assn k t ti> $node_i$ *k* (*aa*, *al*)
ti <*btupi-assn k* (*abs-split*.$node_i$ *k* (*ls* @ (*l*, *a*) # *rs*) *t*)>$_t$
⟨*proof*⟩

lemma *empty-rule*:
shows <*emp*>
empty
<λ*r*. *btree-assn k* (*abs-split.empty-btree*) *r*>
⟨*proof*⟩

end

end
theory *Imperative-Loops*
imports
Refine-Imperative-HOL.Sepref-HOL-Bindings
Refine-Imperative-HOL.Pf-Mono-Prover
Refine-Imperative-HOL.Pf-Add

begin

10 Imperative Loops

An auxiliary while rule provided by Peter Lammich.

lemma *heap-WHILET-rule*:
assumes
wf R
P ⟹$_A$ *I s*
⋀*s*. <*I s* ∗ *true*> *bi s* <λ*r*. *I s* ∗ ↑(*r* ⟷ *b s*)>$_t$
⋀*s*. *b s* ⟹ <*I s* ∗ *true*> *f s* <λ*s*′. *I s*′ ∗ ↑((*s*′, *s*) ∈ *R*)>$_t$
⋀*s*. ¬ *b s* ⟹ *I s* ⟹$_A$ *Q s*
shows <*P* ∗ *true*> *heap-WHILET bi f s* <*Q*>$_t$
⟨*proof*⟩

lemma *heap-WHILET-rule′*:
assumes
wf R
P ⟹$_A$ *I s si* ∗ *F*
⋀*si s*. <*I s si* ∗ *F*> *bi si* <λ*r*. *I s si* ∗ *F* ∗ ↑(*r* ⟷ *b s*)>$_t$
⋀*si s*. *b s* ⟹ <*I s si* ∗ *F*> *f si* <λ*si*′. ∃$_A$*s*′. *I s*′ *si*′ ∗ *F* ∗ ↑((*s*′, *s*) ∈ *R*)>$_t$
⋀*si s*. ¬ *b s* ⟹ *I s si* ∗ *F* ⟹$_A$ *Q s si*

shows $<P>$ *heap-WHILET bi f si* $<\lambda si.\ \exists_A s.\ Q\ s\ si>_t$
⟨*proof*⟩

I derived my own version, simply because it was a better fit to my use case.

corollary *heap-WHILET-rule''*:
assumes
wf R
$P \Longrightarrow_A I\ s$
$\bigwedge s.\ <I\ s * true>\ bi\ s\ <\lambda r.\ I\ s * \uparrow(r \longleftrightarrow b\ s)>_t$
$\bigwedge s.\ b\ s \Longrightarrow <I\ s * true>\ f\ s\ <\lambda s'.\ I\ s' * \uparrow((s',\ s) \in R)>_t$
$\bigwedge s.\ \neg\ b\ s \Longrightarrow I\ s \Longrightarrow_A Q\ s$
shows $<P>$ *heap-WHILET bi f s* $<Q>_t$
⟨*proof*⟩

end
theory *BTree-ImpSplit*
imports
BTree-ImpSet
BTree-Split
Imperative-Loops
begin

11 Imperative split operations

So far, we have only given a functional specification of a possible split. We will now provide imperative split functions that refine the functional specification. However, rather than tracing the execution of the abstract specification, the imperative versions are implemented using while-loops.

11.1 Linear split

The linear split is the most simple split function for binary trees. It serves a good example on how to use while-loops in Imperative/HOL and how to prove Hoare-Triples about its application using loop invariants.

definition *lin-split* :: ('*a*::*heap* × '*b*::{*heap*,*linorder*}) *pfarray* ⇒ '*b* ⇒ *nat Heap*
where
lin-split ≡ λ (*a*,*n*) *p*. *do* {

i ← *heap-WHILET*
(λ*i*. *if i*<*n then do* {
(-,*s*) ← *Array.nth a i*;
return (*s*<*p*)
} *else return False*)
(λ*i*. *return* (*i*+*1*))
0;

```
  return i
}
```

lemma *lin-split-rule*:
$<$ *is-pfa c xs* $(a,n)>$
lin-split (a,n) *p*
$<\lambda i.$ *is-pfa c xs* $(a,n) * \uparrow(i \leq n \wedge (\forall j<i.$ *snd* $(xs!j) < p) \wedge (i<n \longrightarrow$ *snd* $(xs!i) \geq p))>_t$
⟨*proof*⟩

11.2 Binary split

To obtain an efficient B-Tree implementation, we prefer a binary split function. To explore the searching procedure and the resulting proof, we first implement the split on singleton arrays.

```
definition bin'-split :: 'b::{heap,linorder} array-list ⇒ 'b ⇒ nat Heap
  where
    bin'-split ≡ λ(a,n) p. do {
  (low',high') ← heap-WHILET
    (λ(low,high). return (low < high))
    (λ(low,high). let mid = ((low  + high) div 2) in
     do {
      s ← Array.nth a mid;
      if p < s then
         return (low, mid)
      else if p > s then
         return (mid+1, high)
      else return (mid,mid)
     })
    (0::nat,n);
  return low'
}
```

thm *sorted-wrt-nth-less*

lemma *bin'-split-rule*:
sorted-less xs $\Longrightarrow$
$<$ *is-pfa c xs* $(a,n)>$
bin'-split (a,n) *p*
$<\lambda l.$ *is-pfa c xs* $(a,n) * \uparrow(l \leq n \wedge (\forall j<l.\ xs!j < p) \wedge (l<n \longrightarrow xs!l \geq p)) >_t$
⟨*proof*⟩

Then, using the same loop invariant, a binary split for B-tree-like arrays is derived in a straightforward manner.

definition *bin-split* :: ('*a*::*heap* × '*b*::{*heap*,*linorder*}) *pfarray* ⇒ '*b* ⇒ *nat Heap*
where
bin-split ≡ λ(*a*,*n*) *p*. *do* {
(*low'*,*high'*) ← *heap-WHILET*
(λ(*low*,*high*). *return* (*low* < *high*))
(λ(*low*,*high*). *let mid* = ((*low* + *high*) *div* 2) *in*
do {
(-,*s*) ← *Array.nth a mid*;
if p < *s then*
return (*low*, *mid*)
else if p > *s then*
return (*mid*+1, *high*)
else return (*mid*,*mid*)
})
(0::*nat*,*n*);
return low'
}

thm *nth-take*

lemma *nth-take-eq*: *take n ls* = *take n ls'* ⟹ *i* < *n* ⟹ *ls*!*i* = *ls'*!*i*
⟨*proof*⟩

lemma *map-snd-sorted-less*: ⟦*sorted-less* (*map snd xs*); *i* < *j*; *j* < *length xs*⟧
⟹ *snd* (*xs* ! *i*) < *snd* (*xs* ! *j*)
⟨*proof*⟩

lemma *map-snd-sorted-lesseq*: ⟦*sorted-less* (*map snd xs*); *i* ≤ *j*; *j* < *length xs*⟧
⟹ *snd* (*xs* ! *i*) ≤ *snd* (*xs* ! *j*)
⟨*proof*⟩

lemma *bin-split-rule*:
sorted-less (*map snd xs*) ⟹
< *is-pfa c xs* (*a*,*n*)>
bin-split (*a*,*n*) *p*
<λ*l*. *is-pfa c xs* (*a*,*n*) ∗ ↑(*l* ≤ *n* ∧ (∀ *j*<*l*. *snd*(*xs*!*j*) < *p*) ∧ (*l*<*n* ⟶ *snd*(*xs*!*l*)≥*p*))
>$_t$

⟨*proof*⟩

11.3 Refinement of an abstract split

We provide a certain abstract split function that is particularly easy to analyse. The idea of this function is due to Peter Lammich.

definition *abs-split xs x* = (*takeWhile* (λ(-,*s*). *s*<*x*) *xs*, *dropWhile* (λ(-,*s*). *s*<*x*) *xs*)

interpretation *btree-abs-search*: *split abs-split*

⟨*proof*⟩

Any function that yields the heap rule we have obtained for bin_split and lin_split also refines this abstract split.

locale *imp-split-smeq* =
fixes *split-fun* :: ('*a*::{*heap*,*default*,*linorder*} *btnode ref option* × '*a*) *array* × *nat* ⇒ '*a* ⇒ *nat Heap*
assumes *split-rule*: *sorted-less* (*separators xs*) ⟹
<*is-pfa c xs* (*a*, *n*)>
split-fun (*a*, *n*) (*p*::'*a*)
<λ*r*. *is-pfa c xs* (*a*, *n*) ∗
↑ (*r* ≤ *n* ∧
(∀ *j*<*r*. *snd* (*xs* ! *j*) < *p*) ∧
(*r* < *n* ⟶ *p* ≤ *snd* (*xs* ! *r*)))>$_t$
begin

lemma *abs-split-full*: ∀ (-,*s*) ∈ *set xs*. *s* < *p* ⟹ *abs-split xs p* = (*xs*,[])
⟨*proof*⟩

lemma *abs-split-split*:
assumes *n* < *length xs*
and (∀ (-,*s*) ∈ *set* (*take n xs*). *s* < *p*)
and (*case* (*xs*!*n*) *of* (-,*s*) ⇒ ¬(*s* < *p*))
shows *abs-split xs p* = (*take n xs*, *drop n xs*)
⟨*proof*⟩

lemma *split-rule-abs-split*:
shows
sorted-less (*separators ts*) ⟹ <
is-pfa c tsi (*a*,*n*)
∗ *list-assn* (*A* ×$_a$ *id-assn*) *ts tsi*>
split-fun (*a*,*n*) *p*
<λ*i*.
is-pfa c tsi (*a*,*n*)
∗ *list-assn* (*A* ×$_a$ *id-assn*) *ts tsi*
∗ ↑(*split-relation ts* (*abs-split ts p*) *i*)>$_t$
⟨*proof*⟩

sublocale *imp-split abs-split split-fun*
⟨*proof*⟩

end

11.4 Obtaining executable code

In order to obtain fully defined functions, we need to plug our split function implementations into the locales we introduced previously.

interpretation *btree-imp-linear-split*: *imp-split-smeq lin-split*
⟨*proof*⟩

Obtaining actual code turns out to be slightly more difficult due to the use of locales. However, we successfully obtain the B-tree insertion and membership query with binary search splitting.

global-interpretation *btree-imp-binary-split*: *imp-split-smeq bin-split*
defines *btree-isin* = *btree-imp-binary-split.isin*
and *btree-ins* = *btree-imp-binary-split.ins*
and *btree-insert* = *btree-imp-binary-split.insert*
and *btree-empty* = *btree-imp-binary-split.empty*
⟨*proof*⟩

thm *btree-imp-binary-split.ins.simps*
declare *btree-imp-binary-split.ins.simps*[*code*] *btree-imp-binary-split.isin.simps*[*code*]

export-code *btree-empty btree-isin btree-insert* **checking** *SML Scala*
export-code *btree-empty btree-isin btree-insert* **in** *SML* **module-name** *BTreeInsert*
export-code *btree-empty btree-isin btree-insert* **in** *Scala* **module-name** *BTreeInsert*

end

References

[1] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683. URL https://doi.org/10.1007/BF00288683.

[2] Peter Lammich. The imperative refinement framework. *Archive of Formal Proofs*, August 2016. ISSN 2150-914x. https://isa-afp.org/entries/Refine_Imperative_HOL.html, Formal proof development.

[3] Niels Mündler. A verified imperative implementation of b-trees. Bachelor's thesis, Technische Universität München, München, 2021. URL https://mediatum.ub.tum.de/1596550.