

Region Quadtrees

Tobias Nipkow
Technical University of Munich

January 31, 2024

Abstract

These theories formalize *region quadtrees*, which are traditionally used to represent two-dimensional images of (black and white) pixels. Building on these quadtrees, addition and multiplication of recursive block matrices are verified. The generalization of region quadtrees to k dimensions is also formalized.

1 Introduction

These theories formalize so-called *region quadtrees*, as opposed to *point quadtrees* [5, 6, 1]. The following variants are covered:

- Ordinary region quadtrees.
- Block matrices based on region quadtrees. Operations: matrix addition and multiplication. Based on the work of Wise [7, 8, 9, 10, 11].
- A k -dimensional generalization of region quadtrees. This is inspired by the k -dimensional point trees by Bentley [2, 3] which have already been formalized by Rau [4].

For the details of the operations covered see the individual theories.

Contents

1	Introduction	1
2	Quad Tree Basics	3
3	Quad Trees	3
3.1	Compression	3
3.2	Abstraction function	4
3.3	Boolean Quadtrees	5
3.3.1	Abstraction of boolean quadtrees to sets of points	5

3.3.2	Union, Intersection Difference and Complement	6
3.4	Operation <i>put</i>	10
3.5	Extract Square	11
3.6	From Matrix to Quadtree	14
3.6.1	Matrix as function	14
3.6.2	Matrix as list of lists	14
3.7	From Quadtree to Matrix	16
4	Block Matrices via Quad Trees	17
4.1	Square Matrices	17
4.2	Matrix Lemmas	18
4.3	Real Quad Trees and Abstraction to Matrices	18
4.4	Matrix Operations on Trees	19
4.5	Correctness of Quad Tree Implementations	20
4.5.1	<i>add</i>	20
4.5.2	<i>mult</i>	22
5	K-dimensional Region Trees	23
5.1	Subtree	24
5.2	Shifting a coordinate by a boolean vector	24
5.3	Points in a tree	25
5.4	Compression	26
5.5	Extracting a point from a tree	26
5.6	Modifying a point in a tree	27
5.7	Union	29
6	K-dimensional Region Trees - Version 2	30
6.1	Subtree	31
6.2	Shifting a coordinate by a boolean vector	31
6.3	Points in a tree	32
6.4	Compression	33
6.5	Union	33
6.6	Extracting a point from a tree	34
7	K-dimensional Region Trees - Nested Trees	35

2 Quad Tree Basics

```
theory Quad-Base
imports HOL-Library.Tree
begin

datatype 'a qtree = L 'a | Q 'a qtree 'a qtree 'a qtree 'a qtree

instantiation qtree :: (type)height
begin

fun height-qtree :: 'a qtree  $\Rightarrow$  nat where
  height (L _) = 0 |
  height (Q t0 t1 t2 t3) =
    Max {height t0, height t1, height t2, height t3} + 1

instance ..

end

end
```

3 Quad Trees

```
theory Quad-Tree
imports Quad-Base
begin

lemma diff-shunt:  $(\{\} = x - y) \iff (x \leq y)$ 
  by blast

lemma mod-minus:  $\llbracket i < 2*m; \neg i < m \rrbracket \implies i \bmod m = i - (m::nat)$ 
  by (simp add: div-if modulo-nat-def)

definition select :: bool  $\Rightarrow$  bool  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'a where
  select x y t0 t1 t2 t3 =
    (if x then
      if y then t0 else t1
    else
      if y then t2 else t3)

abbreviation qf where
  qf q f i j d  $\equiv$  q (f i j) (f i (j+d)) (f (i+d) j) (f (i+d) (j+d))
```

3.1 Compression

```
fun compressed :: 'a qtree  $\Rightarrow$  bool where
  compressed (L _) = True |
```

$compressed (Q t0 t1 t2 t3) = ((compressed t0 \wedge compressed t1 \wedge compressed t2 \wedge compressed t3) \wedge \neg (\exists x. t0 = L x \wedge t1 = t0 \wedge t2 = t0 \wedge t3 = t0))$

fun $Qc :: 'a\ qtrees \Rightarrow 'a\ qtrees \Rightarrow 'a\ qtrees \Rightarrow 'a\ qtrees \Rightarrow 'a\ qtrees$ **where**
 $Qc (L x0) (L x1) (L x2) (L x3) =$
 $(if\ x0=x1 \wedge x1=x2 \wedge x2=x3\ then\ L\ x0\ else\ Q (L x0) (L x1) (L x2) (L x3)) \mid$
 $Qc\ t0\ t1\ t2\ t3 = Q\ t0\ t1\ t2\ t3$

Compressing version of Q :

lemma $compressed\text{-}Qc$: $\llbracket compressed\ t0; compressed\ t1; compressed\ t2; compressed\ t3 \rrbracket \Longrightarrow$
 $compressed (Qc\ t0\ t1\ t2\ t3)$
by($induction\ t0\ t1\ t2\ t3$ rule: $Qc.induct$) ($auto\ split!$: $qtrees.split$)

lemma $compressedQD$: $compressed (Q\ t1\ t2\ t3\ t4)$
 $\Longrightarrow compressed\ t1 \wedge compressed\ t2 \wedge compressed\ t3 \wedge compressed\ t4$
using $compressed.simps(2)$ **by** $blast$

lemma $height\text{-}Qc\text{-}Q$: $\llbracket height\ s0 \leq n; height\ s1 \leq n; height\ s2 \leq n; height\ s3 \leq n \rrbracket$
 $\Longrightarrow height (Qc\ s0\ s1\ s2\ s3) \leq Suc\ n$
apply($cases (s0,s1,s2,s3)$ rule: $Qc.cases$)
using $\llbracket simp\text{-}depth\text{-}limit=1 \rrbracket$ **apply** $simp\text{-}all$
done

Modify a quadrant addressed by x and y , and put things back together with Qc :

fun $modify :: ('a\ qtrees \Rightarrow 'a\ qtrees) \Rightarrow bool \Rightarrow bool \Rightarrow 'a\ qtrees * 'a\ qtrees * 'a\ qtrees * 'a\ qtrees \Rightarrow 'a\ qtrees$ **where**
 $modify\ f\ x\ y (t0, t1, t2, t3) =$
 $(if\ x\ then$
 $\quad if\ y\ then\ Qc (f\ t0) t1\ t2\ t3\ else\ Qc\ t0 (f\ t1) t2\ t3$
 $\quad else$
 $\quad if\ y\ then\ Qc\ t0\ t1 (f\ t2) t3\ else\ Qc\ t0\ t1\ t2 (f\ t3))$

3.2 Abstraction function

fun $get :: nat \Rightarrow 'a\ qtrees \Rightarrow nat \Rightarrow nat \Rightarrow 'a$ **where**
 $get\ n (L\ b) - - = b \mid$
 $get (Suc\ n) (Q\ t0\ t1\ t2\ t3) i\ j =$
 $get\ n (select (i < 2^n) (j < 2^n) t0\ t1\ t2\ t3) (i\ mod\ 2^n) (j\ mod\ 2^n)$

lemma $get\text{-}Qc$:
 $height(Q\ t0\ t1\ t2\ t3) \leq n \Longrightarrow get\ n (Qc\ t0\ t1\ t2\ t3) i\ j = get\ n (Q\ t0\ t1\ t2\ t3) i\ j$
apply($cases\ n$)
apply $simp$
apply($cases (t0,t1,t2,t3)$ rule: $Qc.cases$)

apply(*simp-all add: select-def*)
done

3.3 Boolean Quadrees

type-synonym *qtb* = *bool qtree*

3.3.1 Abstraction of boolean quadrees to sets of points

Superseded by the more general *get* abstraction.

type-synonym *points* = (*nat* × *nat*) *set*

abbreviation *sq* :: *nat* ⇒ *points* **where**
sq (*n*::*nat*) ≡ {*0*..*2* [^] *n*} × {*0*..*2* [^] *n*}

definition *shift* :: *nat* ⇒ *nat* ⇒ *nat* * *nat* ⇒ *nat* * *nat* **where**
shift *di dj* = (λ(*i,j*). (*i*+*di*, *j*+*dj*))

lemma *shift-pair*[*simp*]: *shift* *di dj* (*a,b*) = (*a*+*di*,*b*+*dj*)
by(*simp add: shift-def*)

lemma *in-shift-image*: (*x,y*) ∈ *shift* *di dj* ‘ *M* ⟷ *di* ≤ *x* ∧ *dj* ≤ *y* ∧ (*x*-*di*,*y*-*dj*) ∈ *M*
by(*force simp: shift-def*)

lemma *inj-shift*: *inj* (*shift* *i j*)
by (*auto simp: inj-def*)

lemma *shift-disj-shift*: [*s* ⊆ *sq* *n*; *s'* ⊆ *sq* *n*;
i ≥ *i'* + *2* [^] *n* ∨ *i'* ≥ *i* + *2* [^] *n* ∨ *j* ≥ *j'* + *2* [^] *n* ∨ *j'* ≥ *j* + *2* [^] *n*] ⇒
shift *i j* ‘ *s* ∩ *shift* *i' j'* ‘ *s'* = {}
by (*auto simp add: in-shift-image*)

Convention: *A, B* :: *points*

The layout of the 4 subquadrants *Q t0 t1 t2 t3* / *Qsq A0 A1 A2 A3*:
 $\begin{matrix} 1 & & & \\ 3 & 0 & 2 & \\ \end{matrix}$ That is, the *x* and *y* coordinates are shifted as follows (where $1 = 2^n$):
(*0,1*) (*1,1*) (*0,0*) (*1,0*)

definition *Qsq* :: *nat* ⇒ *points* ⇒ *points* ⇒ *points* ⇒ *points* ⇒ *points* **where**
Qsq *n A0 A1 A2 A3* =
shift *0 0* ‘ *A0* ∪ *shift* *0* (*2* [^] *n*) ‘ *A1* ∪ *shift* (*2* [^] *n*) *0* ‘ *A2* ∪ *shift* (*2* [^] *n*) (*2* [^] *n*) ‘ *A3*

lemma *sq-Suc-Qsq*: {*0*..*2* * *2* [^] *n*} × {*0*..*2* * *2* [^] *n*} = *Qsq* *n* (*sq* *n*) (*sq* *n*)
(*sq* *n*) (*sq* *n*)
by(*auto simp: in-shift-image Qsq-def*)

fun *points* :: *nat* ⇒ *qtb* ⇒ (*nat* * *nat*) *set* **where**
points *n* (*L b*) = (if *b* then *sq* *n* else {}) |
points (*Suc* *n*) (*Q t0 t1 t2 t3*) = *Qsq* *n* (*points* *n* *t0*) (*points* *n* *t1*) (*points* *n* *t2*)
(*points* *n* *t3*)

```

lemma points-subset:  $height\ t \leq n \implies points\ n\ t \subseteq sq\ n$ 
proof(induction  $n\ t$  rule: points.induct)
  case 1
  then show ?case by simp
next
  case ( $2\ n\ t0\ t1\ t2\ t3$ )
  from  $2.prem$ s have  $h$ :  $height\ t0 \leq n\ height\ t1 \leq n\ height\ t2 \leq n\ height\ t3 \leq n$ 
  by (auto)
  thus ?case
  using  $2.prem$ s  $2.IH(1)[OF\ h(1)]\ 2.IH(2)[OF\ h(2)]\ 2.IH(3)[OF\ h(3)]\ 2.IH(4)[OF\ h(4)]$ 
  by (auto simp add: Let-def shift-def Qsq-def)
next
  case 3 thus ?case
  by simp
qed

```

```

lemma point-Suc-Qc[simp]:  $points\ (Suc\ n)\ (Qc\ t0\ t1\ t2\ t3) = points\ (Suc\ n)\ (Q\ t0\ t1\ t2\ t3)$ 
by(induction  $t0\ t1\ t2\ t3$  rule: Qc.induct) (auto simp: in-shift-image Qsq-def)

```

```

lemma get-points:  $\llbracket height\ t \leq n; (i,j) \in sq\ n \rrbracket \implies get\ n\ t\ i\ j = ((i,j) \in points\ n\ t)$ 
proof(induction  $n\ t\ i\ j$  rule: get.induct)
  case 1
  then show ?case by simp
next
  case ( $2\ n\ t0\ t1\ t2\ t3$ )
  thus ?case using points-subset[of  $t0\ n$ ] points-subset[of  $t1\ n$ ] points-subset[of  $t2\ n$ ]
  by(auto simp: select-def in-shift-image mod-minus Qsq-def)
next
  case 3
  then show ?case by simp
qed

```

3.3.2 Union, Intersection Difference and Complement

```

fun union ::  $qtb \Rightarrow qtb \Rightarrow qtb$  where
  union ( $L\ b$ )  $t = (if\ b\ then\ L\ True\ else\ t) \mid$ 
  union  $t$  ( $L\ b$ ) = (if  $b$  then  $L\ True$  else  $t$ )  $\mid$ 
  union ( $Q\ s1\ s2\ s3\ s4$ ) ( $Q\ t1\ t2\ t3\ t4$ ) =  $Qc\ (union\ s1\ t1)\ (union\ s2\ t2)\ (union\ s3\ t3)\ (union\ s4\ t4)$ 

```

```

fun inter ::  $qtb \Rightarrow qtb \Rightarrow qtb$  where
  inter ( $L\ b$ )  $t = (if\ b\ then\ t\ else\ L\ False) \mid$ 
  inter  $t$  ( $L\ b$ ) = (if  $b$  then  $t$  else  $L\ False$ )  $\mid$ 
  inter ( $Q\ s1\ s2\ s3\ s4$ ) ( $Q\ t1\ t2\ t3\ t4$ ) =  $Qc\ (inter\ s1\ t1)\ (inter\ s2\ t2)\ (inter\ s3\ t3)\ (inter\ s4\ t4)$ 

```

$t3$) (*inter* $s4$ $t4$)

fun *negate* :: $qtb \Rightarrow qtb$ **where**

negate (L b) = $L(\neg b)$ |

negate (Q $t1$ $t2$ $t3$ $t4$) = Q (*negate* $t1$) (*negate* $t2$) (*negate* $t3$) (*negate* $t4$)

fun *diff* :: $qtb \Rightarrow qtb \Rightarrow qtb$ **where**

diff (L b) t = (*if* b *then* *negate* t *else* L *False*) |

diff t (L b) = (*if* b *then* L *False* *else* t) |

diff (Q $s1$ $s2$ $s3$ $s4$) (Q $t1$ $t2$ $t3$ $t4$) = Qc (*diff* $s1$ $t1$) (*diff* $s2$ $t2$) (*diff* $s3$ $t3$) (*diff* $s4$ $t4$)

lemma *Qsq-union*:

Qsq n $A0$ $A1$ $A2$ $A3 \cup Qsq$ n $B0$ $B1$ $B2$ $B3$ = Qsq n ($A0 \cup B0$) ($A1 \cup B1$) ($A2 \cup B2$) ($A3 \cup B3$)

by(*auto simp: Qsq-def*)

lemma *points-union*:

max (*height* $t1$) (*height* $t2$) $\leq n \implies points$ n (*union* $t1$ $t2$) = $points$ n $t1 \cup points$ n $t2$

proof(*induction* $t1$ $t2$ *arbitrary: n rule: union.induct*)

case 1 **thus** ?*case* **using** *Un-absorb2[OF points-subset]* **by** *simp*

next

case 2 **thus** ?*case* **using** *Un-absorb1[OF points-subset]* **by** *simp*

next

case 3

from 3.*prems* **obtain** m **where** $n = Suc$ m **by** (*auto dest: Suc-le-D*)

thus ?*case* **using** 3 **by** (*simp add: Qsq-union*)

qed

lemma *height-union*: $height$ (*union* $t1$ $t2$) $\leq max$ (*height* $t1$) (*height* $t2$)

proof(*induction* $t1$ $t2$ *rule: union.induct*)

case 3 **then show** ?*case*

by(*auto simp add: height-Qc-Q le-max-iff-disj simp del: max.absorb1 max.absorb2 max.absorb3 max.absorb4*)

qed *auto*

lemma *height-union2*: $\llbracket height$ $t1 \leq n; height$ $t2 \leq n \rrbracket \implies height$ (*union* $t1$ $t2$) $\leq n$

by (*meson height-union le-trans max.bounded-iff*)

lemma *get-union*:

max (*height* $t1$) (*height* $t2$) $\leq n \implies get$ n (*union* $t1$ $t2$) i j = (get n $t1$ i $j \vee get$ n $t2$ i j)

proof(*induction* $t1$ $t2$ *arbitrary: i j n rule: union.induct*)

case 3

from 3.*prems* **obtain** m **where** $n = Suc$ m **by** (*auto dest: Suc-le-D*)

thus ?*case* **using** 3 **by** (*auto simp add: get-Qc height-union2 select-def*)

qed *auto*

lemma *compressed-union*: $\text{compressed } t1 \implies \text{compressed } t2 \implies \text{compressed}(\text{union } t1 \ t2)$

proof(*induction t1 t2 arbitrary: rule: union.induct*)

case 1 thus *?case using Un-absorb2[OF points-subset] by simp*

next

case 2 thus *?case using Un-absorb1[OF points-subset] by simp*

next

case 3

thus *?case*

by (*metis compressedQD compressed-Qc union.simps(3)*)

qed

lemma *Qsq-inter*:

$\llbracket A0 \subseteq \text{sq } n; A1 \subseteq \text{sq } n; A2 \subseteq \text{sq } n; A3 \subseteq \text{sq } n;$

$B0 \subseteq \text{sq } n; B1 \subseteq \text{sq } n; B2 \subseteq \text{sq } n; B3 \subseteq \text{sq } n \rrbracket$

$\implies \text{Qsq } n \ A0 \ A1 \ A2 \ A3 \ \cap \ \text{Qsq } n \ B0 \ B1 \ B2 \ B3 = \text{Qsq } n \ (A0 \ \cap \ B0) \ (A1 \ \cap \ B1)$
 $(A2 \ \cap \ B2) \ (A3 \ \cap \ B3)$

by(*simp add: Qsq-def Int-Un-distrib Int-Un-distrib2 shift-disj-shift image-Int inj-shift*)

lemma *points-inter*: $n \geq \max(\text{height } t1) (\text{height } t2) \implies$

$\text{points } n \ (\text{inter } t1 \ t2) = \text{points } n \ t1 \ \cap \ \text{points } n \ t2$

proof(*induction t1 t2 arbitrary: n rule: inter.induct*)

case 1 thus *?case by (simp add: inf-absorb2[OF points-subset])*

next

case 2 thus *?case by (simp add: inf-absorb1[OF points-subset])*

next

case 3

from *3.prem*s **obtain** *m* **where** $n = \text{Suc } m$ **by** (*auto dest: Suc-le-D*)

thus *?case using 3.prem*s *3.IH*[*of m*]

by (*simp add: Qsq-inter points-subset*)

qed

lemma *height-inter*: $\text{height}(\text{inter } t1 \ t2) \leq \max(\text{height } t1) (\text{height } t2)$

proof(*induction t1 t2 rule: inter.induct*)

case 3 then show *?case*

by(*auto simp add: height-Qc-Q le-max-iff-disj simp del: max.absorb1 max.absorb2 max.absorb3 max.absorb4*)

qed *auto*

lemma *height-inter2*: $\llbracket \text{height } t1 \leq n; \text{height } t2 \leq n \rrbracket \implies \text{height}(\text{inter } t1 \ t2) \leq n$

by (*meson height-inter le-trans max.bounded-iff*)

lemma *get-inter*:

$\llbracket \text{height } t1 \leq n; \text{height } t2 \leq n \rrbracket \implies \text{get } n \ (\text{inter } t1 \ t2) \ i \ j = (\text{get } n \ t1 \ i \ j \ \wedge \ \text{get } n \ t2 \ i \ j)$

proof(*induction t1 t2 arbitrary: i j n rule: union.induct*)

case 3
from 3.prem_s **obtain** m **where** $n = \text{Suc } m$ **by** (auto dest: Suc-le-D)
thus ?case **using** 3 **by** (auto simp add: get-Qc height-inter2 select-def)
qed auto

lemma compressed-inter: compressed $t1 \implies$ compressed $t2 \implies$ compressed(inter $t1$ $t2$)

proof(induction $t1$ $t2$ arbitrary: rule: inter.induct)
case 1 **thus** ?case **using** Un-absorb2[OF points-subset] **by** simp
next
case 2 **thus** ?case **using** Un-absorb1[OF points-subset] **by** simp
next
case 3
thus ?case
by (metis compressedQD compressed-Qc inter.simps(3))
qed

lemma Qsq-diff: $\llbracket B0 \subseteq \text{sq } n; B1 \subseteq \text{sq } n; B2 \subseteq \text{sq } n; B3 \subseteq \text{sq } n; A0 \subseteq \text{sq } n; A1 \subseteq \text{sq } n; A2 \subseteq \text{sq } n; A3 \subseteq \text{sq } n \rrbracket \implies$
 $\text{Qsq } n$ $B0$ $B1$ $B2$ $B3 - \text{Qsq } n$ $A0$ $A1$ $A2$ $A3 = \text{Qsq } n$ $(B0 - A0)$ $(B1 - A1)$
 $(B2 - A2)$ $(B3 - A3)$
by (auto simp add: in-shift-image Qsq-def)

lemma points-negate: $n \geq \text{height } t \implies \text{points } n$ (negate t) = $\text{sq } n - \text{points } n$ t
proof(induction t arbitrary: n rule: negate.induct)

case 1 **thus** ?case **by** (simp)
next
case (2 $t0$ $t1$ $t2$ $t3$)
obtain m **where** [simp]: $n = \text{Suc } m$ **using** Suc-le-D 2.prem_s **by** auto
thus ?case **using** 2.prem_s 2.IH[of m]
by(simp add: sq-Suc-Qsq Qsq-diff points-subset)
qed

lemma negate-eq-L-iff: compressed $t \implies \text{negate } t = L$ $x \iff t = L(\neg x)$
by(cases t) auto

lemma compressed-negate: compressed $t \implies$ compressed(negate t)
proof(induction t)

case L **thus** ?case **by** simp
next
case Q
thus ?case **using** negate-eq-L-iff **by** force
qed

lemma points-diff: $n \geq \max(\text{height } t1)$ ($\text{height } t2$) \implies
 $\text{points } n$ (diff $t1$ $t2$) = $\text{points } n$ $t1 - \text{points } n$ $t2$

proof(induction $t1$ $t2$ arbitrary: n rule: diff.induct)
case 1 **thus** ?case **by** (simp add: points-negate)

```

next
  case 2 thus ?case using points-subset by (simp add: diff-shunt)
next
  case 3
  from 3.prem3 obtain m where n = Suc m by (auto dest: Suc-le-D)
  thus ?case using 3.prem3 3.IH[of m]
  by (simp add: Qsq-diff points-subset)
qed

lemma compressed-diff: compressed t1  $\implies$  compressed t2  $\implies$  compressed(diff t1
t2)
proof(induction t1 t2 arbitrary: rule: diff.induct)
  case 1 thus ?case
  by (simp add: compressed-negate)
next
  case 2 thus ?case by simp
next
  case 3
  thus ?case
  by (metis compressedQD compressed-Qc diff.simps(3))
qed

```

3.4 Operation put

```

fun put :: nat  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  nat  $\Rightarrow$  'a qtree  $\Rightarrow$  'a qtree where
  put i j a 0 (L -) = L a |
  put i j a (Suc n) t = modify (put (i mod 2 $\hat{~}$ n) (j mod 2 $\hat{~}$ n) a n) (i < 2 $\hat{~}$ n) (j <
2 $\hat{~}$ n)
  (case t of L b  $\Rightarrow$  (L b, L b, L b, L b) | Q t0 t1 t2 t3  $\Rightarrow$  (t0,t1,t2,t3))

```

```

lemma points-put:  $\llbracket$  height t  $\leq$  n; (i,j)  $\in$  sq n  $\rrbracket \implies$ 
  points n (put i j b n t) = (if b then points n t  $\cup$  {(i,j)} else points n t - {(i,j)})

```

```

proof(induction i j b n t rule: put.induct)
  case 1
  then show ?case by (simp)
next
  case 2
  thus ?case unfolding mem-Sigma-iff using points-subset
  apply(simp add: select-def sq-Suc-Qsq Qsq-def mod-minus split: qtree.split)
  by(fastforce simp: mod-minus in-shift-image)
qed auto

```

```

lemma height-put: height t  $\leq$  n  $\implies$  height (put i j a n t)  $\leq$  n

```

```

proof(induction i j a n t rule: put.induct)
  case 2
  then show ?case by (auto simp: height-Qc-Q split: qtree.split)
qed auto

```

```

lemma get-put:  $\llbracket$  height t  $\leq$  n; (i,j)  $\in$  sq n; (i',j')  $\in$  sq n  $\rrbracket \implies$ 

```

```

  get n (put i j a n t) i' j' = (if i'=i ∧ j'=j then a else get n t i' j')
proof(induction i j a n t arbitrary: i' j' rule: put.induct)
  case 1
  then show ?case by (auto)
next
  case 2
  thus ?case
  by(auto simp add: select-def mod-minus get-Qc height-put less-diff-conv2 split!:
  qtree.split)
qed auto

```

```

lemma compressed-put:
  [ height t ≤ n; compressed t ] ⇒ compressed (put i j a n t)
proof(induction i j a n t rule: put.induct)
  case 1
  then show ?case by (simp)
next
  case 2
  thus ?case by (auto simp add: compressed-Qc split: qtree.split)
qed auto

```

3.5 Extract Square

```

fun get-sq :: nat ⇒ 'a qtree ⇒ nat ⇒ nat ⇒ nat ⇒ 'a qtree where
  get-sq n (L b) m i j = L b |
  get-sq n t 0 i j = L (get n t i j) |
  get-sq (Suc n) (Q t0 t1 t2 t3) (Suc m) i j =
    (if i mod 2n + 2(m+1) ≤ 2n ∧ j mod 2n + 2(m+1) ≤ 2n
     then get-sq n (select (i < 2n) (j < 2n) t0 t1 t2 t3) (m+1) (i mod 2n) (j
     mod 2n)
     else qf Qc (get-sq (Suc n) (Q t0 t1 t2 t3) m) i j (2m))

```

```

lemma shift-shift: shift i j ' (shift i' j' ' s) = shift (i+i') (j+j') ' s
using image-iff by(fastforce simp add: shift-def)

```

```

lemma shift-shift2: shift i j ' (shift i' j' ' s) = shift (i'+i) (j'+j) ' s
by(simp add: shift-shift Groups.add-ac)

```

```

lemma shift-split: shift i j ' s =
  shift (i - i mod 2n) (j - j mod 2n) ' (shift (i mod 2n) (j mod 2n) ' s)
by (simp add: shift-shift)

```

```

lemma plus-pow-aux: (i::nat) + 2m ≤ 2*2n ⇒ i < 2 * 2n
by (metis add-leD1 le-neq-implies-less less-exp nat-add-left-cancel-less not-add-less1)

```

```

lemma Qsq-lem: [ A0 ⊆ sq n; A1 ⊆ sq n; A2 ⊆ sq n; A3 ⊆ sq n;
  i + 2m ≤ 2n Suc n; j + 2m ≤ 2n Suc n;
  i mod 2n + 2m ≤ 2n; j mod 2n + 2m ≤ 2n ] ⇒
  Qsq n A0 A1 A2 A3 ∩ shift i j ' sq m =
  shift (i - i mod 2n) (j - j mod 2n) ' select (i < 2n) (j < 2n) A0 A1

```

$A2 A3 \cap \text{shift } i j \text{ ' } sq m$
by (*auto simp: select-def Qsq-def mod-minus plus-pow-aux*)

lemma *f-select*: $f (\text{select } x y a b c d) = \text{select } x y (f a) (f b) (f c) (f d)$
by(*simp add: select-def*)

lemma *height-get-sq*: $m \leq n \implies \text{height } (\text{get-sq } n t m i j) \leq m$

proof(*induction n t m i j rule: get-sq.induct*)

case ($\exists n t0 t1 t2 t3 m i j$)

have *: $i \bmod 2^{\wedge} n + 2 * 2^{\wedge} m \leq 2^{\wedge} n \implies \text{Suc } m \leq n$

using *power-le-imp-le-exp[of 2::nat Suc m n]* **by** *simp*

show *?case*

using *3.IH 3.prem s ** **by** (*auto simp add: height-Qc-Q Let-def*)

qed *auto*

lemma *shift-Qsq*: $\text{shift } i j \text{ ' } Qsq n A0 A1 A2 A3 =$

$Qsq n (\text{shift } i j \text{ ' } A0) (\text{shift } i j \text{ ' } A1) (\text{shift } i j \text{ ' } A2) (\text{shift } i j \text{ ' } A3)$

by(*simp add: Qsq-def image-Un shift-shift add commute*)

lemma *points-get-sq*:

$\llbracket \text{height } t \leq n; i + 2^{\wedge} m \leq 2^{\wedge} n; j + 2^{\wedge} m \leq 2^{\wedge} n \rrbracket \implies$

$\text{shift } i j \text{ ' } \text{points } m (\text{get-sq } n t m i j) = \text{points } n t \cap (\text{shift } i j \text{ ' } sq m)$

proof (*induction n t m i j rule: get-sq.induct*)

case *2*

then show *?case* **by** (*auto simp: get-points*)

next

case ($\exists n t0 t1 t2 t3 m1 i j$)

define *m* **where** $m = \text{Suc } m1$

let *?t* = $Q t0 t1 t2 t3$

show *?case*

proof (*cases i mod 2^{\wedge} n + 2^{\wedge} m \leq 2^{\wedge} n \wedge j mod 2^{\wedge} n + 2^{\wedge} m \leq 2^{\wedge} n*)

case *True*

let *?sel* = $\text{select } (i < 2^{\wedge} n) (j < 2^{\wedge} n) t0 t1 t2 t3$

let *?i* = $i \bmod 2^{\wedge} n$ **let** *?j* = $j \bmod 2^{\wedge} n$

have *1*: $\text{height } ?sel \leq n$ **using** *3.prem s* **by**(*auto simp: select-def*)

have *2*: $\text{points } m (\text{get-sq } (\text{Suc } n) ?t m i j) = \text{points } m (\text{get-sq } n ?sel m ?i ?j)$

using *True unfolding get-sq.simps m-def* **by**(*simp add: Let-def*)

have *3*: $\text{shift } ?i ?j \text{ ' } \text{points } m (\text{get-sq } n ?sel m ?i ?j) = \text{points } n ?sel \cap \text{shift } ?i ?j \text{ ' } sq m$

using *3.IH(1) 1 True* **by** (*simp add: m-def*)

have $\text{shift } i j \text{ ' } \text{points } (\text{Suc } m1) (\text{get-sq } (\text{Suc } n) ?t (\text{Suc } m1) i j) =$

$\text{shift } i j \text{ ' } \text{points } m (\text{get-sq } n ?sel m ?i ?j)$

using *True unfolding get-sq.simps m-def* **by**(*simp add: Let-def*)

also have $\dots = \text{shift } (i - ?i) (j - ?j) \text{ ' } \text{points } m (\text{get-sq } n ?sel m ?i ?j)$

by (*meson shift-split*)

also have $\dots = \text{shift } (i - ?i) (j - ?j) \text{ ' } (\text{points } n ?sel \cap \text{shift } ?i ?j \text{ ' } sq m)$

using *3.IH(1) 1 True* **by** (*simp add: m-def*)

also have $\dots = \text{shift } (i - ?i) (j - ?j) \text{ ' } \text{points } n ?sel \cap \text{shift } i j \text{ ' } sq m$

```

    using image-Int[OF inj-shift] shift-split by presburger
    also have ... = shift (i - ?i) (j - ?j) ' select (i < 2 ^ n) (j < 2 ^ n) (points
n t0) (points n t1) (points n t2) (points n t3) ∩ shift i j ' sq m
      by(simp add: f-select)
    also have ... = points (Suc n) (Q t0 t1 t2 t3) ∩ shift i j ' sq (Suc m1)
      using 3.prem1 True
      apply(subst Qsq-lem[symmetric])
      by(auto simp: points-subset m-def)
    finally show ?thesis .
next
case False
have shift i j ' points (Suc m1) (get-sq (Suc n) (Q t0 t1 t2 t3) (Suc m1) i j) =
  shift i j ' qf (Qsq m1) (λx y. points m1 (get-sq (Suc n) ?t m1 x y)) i j (2 ^ m1)
  using False unfolding get-sq.simps m-def
  by(simp add: Let-def m-def del: de-Morgan-conj)
also have ... = qf (Qsq m1) (λx y. shift i j ' points m1 (get-sq (Suc n) ?t m1
x y)) i j (2 ^ m1)
  by(simp add: shift-Qsq)
also have ... = points (Suc n) (Q t0 t1 t2 t3) ∩ shift i j ' sq (Suc m1)
  using 3.IH(2-5) 3.prem1 False unfolding get-sq.simps m-def
  by(simp add: sq-Suc-Qsq Qsq-def shift-shift2 image-Int[OF inj-shift] image-Un
Int-Un-distrib add commute)
  finally show ?thesis .
qed
qed auto

```

lemma *get-get-sq*:

```

[[ height t ≤ n; i + 2 ^ m ≤ 2 ^ n; j + 2 ^ m ≤ 2 ^ n; i' < 2 ^ m; j' < 2 ^ m ]] ⇒
  get m (get-sq n t m i j) i' j' = get n t (i+i') (j+j')
proof (induction n t m i j arbitrary: i' j' rule: get-sq.induct)
case (3 n t0 t1 t2 t3 m i j)
let ?t = Q t0 t1 t2 t3
let ?sel = select (i < 2 ^ n) (j < 2 ^ n) t0 t1 t2 t3
show ?case
proof (cases i mod 2 ^ n + 2 ^ (m+1) ≤ 2 ^ n ∧ j mod 2 ^ n + 2 ^ (m+1) ≤ 2 ^ n)
case True
have get (Suc m) (get-sq (Suc n) ?t (Suc m) i j) i' j'
  = get (m+1) (get-sq n ?sel (m+1) (i mod 2 ^ n) (j mod 2 ^ n)) i' j'
  using True by(simp)
also have ... = get n ?sel (i mod 2 ^ n + i') (j mod 2 ^ n + j')
  using True 3.prem1 by(subst 3.IH(1))(simp-all add: select-def)
also have ... = get (Suc n) ?t (i + i') (j + j')
  using True 3.prem1 by(auto simp add: select-def mod-minus)
  finally show ?thesis .
next
case False
have *: i + 2 * 2 ^ m ≤ 2 * 2 ^ n ⇒ m ≤ Suc n
  using power-le-imp-le-exp[of 2::nat m n] by linarith
show ?thesis using False 3.prem1

```

by(auto simp add: 3.IH(2-5) get-Qc mod-minus select-def height-Qc-Q
height-get-sq *)

qed
qed auto

lemma compressed-get-sq:

$\llbracket \text{height } t \leq n; \text{ compressed } t \rrbracket \implies \text{compressed } (\text{get-sq } n \ t \ m \ i \ j)$

proof(induction n t m i j rule: get-sq.induct)

case ($\exists n \ t0 \ t1 \ t2 \ t3 \ m \ i \ j$)

then show ?case by (simp add: compressed-Qc select-def)

qed auto

3.6 From Matrix to Quadtree

3.6.1 Matrix as function

definition shift-mx where

$\text{shift-mx } mx \ x \ y = (\lambda i \ j. \ mx \ (i+x) \ (j+y))$

fun qt-of-fun :: (nat \Rightarrow nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a qtree where

$\text{qt-of-fun } mx \ (\text{Suc } n) = \text{qf } Qc \ (\lambda x \ y. \ \text{qt-of-fun } (\text{shift-mx } mx \ x \ y) \ n) \ 0 \ 0 \ (2^{\wedge}n) \ |$

$\text{qt-of-fun } mx \ 0 = L(mx \ 0 \ 0)$

lemma points-qt-of-fun: points n (qt-of-fun mx n) = $\{(i,j) \in \text{sq } n. \ mx \ i \ j\}$

proof(induction n arbitrary: mx)

case 0

then show ?case by (auto)

next

case (Suc n)

then show ?case by(auto simp add: shift-mx-def Suc-length-conv sq-Suc-Qsq
Qsq-def Let-def)

qed

lemma compressed-qt-of-fun: compressed (qt-of-fun mx n)

proof(induction n arbitrary: mx)

case 0

then show ?case by simp

next

case (Suc n)

then show ?case by(simp add: compressed-Qc)

qed

3.6.2 Matrix as list of lists

type-synonym 'a mx = 'a list list

definition sq-mx n mx = (length mx = $2^{\wedge}n \wedge (\forall xs \in \text{set } mx. \text{length } xs = 2^{\wedge}n)$)

lemma sq-mx-0: sq-mx 0 mx = ($\exists x. \ mx = \llbracket x \rrbracket$)

by(auto simp: sq-mx-def length-Suc-conv)

Decompose matrix into submatrices

definition *decomp* **where**

decomp n $mx = (\text{let } mx01 = \text{take } (2^{\wedge}n) \text{ } mx; mx23 = \text{drop } (2^{\wedge}n) \text{ } mx$
in $(\text{map } (\text{take } (2^{\wedge}n)) \text{ } mx01, \text{map } (\text{drop } (2^{\wedge}n)) \text{ } mx01, \text{map } (\text{take } (2^{\wedge}n)) \text{ } mx23,$
 $\text{map } (\text{drop } (2^{\wedge}n)) \text{ } mx23))$

lemma *decomp-sq-mx*: $sq\text{-}mx \text{ } (Suc \text{ } n) \text{ } mx \implies (mx0, mx1, mx2, mx3) = \text{decomp } n$
 $mx \implies$

$sq\text{-}mx \text{ } n \text{ } mx0 \wedge sq\text{-}mx \text{ } n \text{ } mx1 \wedge sq\text{-}mx \text{ } n \text{ } mx2 \wedge sq\text{-}mx \text{ } n \text{ } mx3$

by $(\text{auto simp add: } sq\text{-}mx\text{-def min-def decomp-def Let-def dest: in-set-takeD in-set-dropD})$

Quadtree of matrix:

fun *qt-of* :: $nat \Rightarrow 'a \text{ } mx \Rightarrow 'a \text{ } qtrees$ **where**

qt-of $(Suc \text{ } n) \text{ } mx =$

$(\text{let } (mx0, mx1, mx2, mx3) = \text{decomp } n \text{ } mx$

in $Qc \text{ } (qt\text{-of } n \text{ } mx0) \text{ } (qt\text{-of } n \text{ } mx1) \text{ } (qt\text{-of } n \text{ } mx2) \text{ } (qt\text{-of } n \text{ } mx3)) \mid$

qt-of $0 \text{ } [[x]] = L \text{ } x$

lemma *height-qt-of*: $sq\text{-}mx \text{ } n \text{ } mx \implies \text{height}(qt\text{-of } n \text{ } mx) \leq n$

proof $(\text{induction } n \text{ } mx \text{ rule: } qt\text{-of.}induct)$

case $(1 \text{ } n \text{ } mx)$

obtain $mx0 \text{ } mx1 \text{ } mx2 \text{ } mx3$ **where** $*$: $\text{decomp } n \text{ } mx = (mx0, mx1, mx2, mx3)$ **by**
 $(metis \text{ prod-cases4})$

show $?case$

using $* \text{ } 1$ **by** $(\text{fastforce simp: height-Qc-Q dest!: decomp-sq-mx})$

qed $(\text{auto simp: } sq\text{-}mx\text{-def})$

lemma *compressed-qt-of*: $sq\text{-}mx \text{ } n \text{ } mx \implies \text{compressed}(qt\text{-of } n \text{ } mx)$

proof $(\text{induction } n \text{ } mx \text{ rule: } qt\text{-of.}induct)$

case $(1 \text{ } n \text{ } mx)$

obtain $mx0 \text{ } mx1 \text{ } mx2 \text{ } mx3$ **where** $*$: $\text{decomp } n \text{ } mx = (mx0, mx1, mx2, mx3)$ **by**
 $(metis \text{ prod-cases4})$

show $?case$

using $* \text{ } 1 \text{ } \text{decomp-sq-mx}[OF \text{ } 1.\text{prems}]$

by $(\text{simp add: compressed-Qc})$

qed $(\text{auto simp: } sq\text{-}mx\text{-def})$

lemma *points-qt-of*: $sq\text{-}mx \text{ } n \text{ } mx \implies \text{points } n \text{ } (qt\text{-of } n \text{ } mx) = \{(i, j) \in sq \text{ } n. \text{ } mx !$
 $i ! j\}$

proof $(\text{induction } n \text{ } arbitrary: \text{ } mx)$

case 0

then show $?case$ **by** $(\text{auto simp: } sq\text{-}mx\text{-0 split: if-splits})$

next

case $(Suc \text{ } n)$

obtain $mx0 \text{ } mx1 \text{ } mx2 \text{ } mx3$ **where** $*$: $(mx0, mx1, mx2, mx3) = \text{decomp } n \text{ } mx$ **by**
 $(metis \text{ prod-cases4})$

note $** = \text{decomp-sq-mx}[OF \text{ } Suc.\text{prems } *]$

show $?case$ **using** $Suc \text{ } * \text{ } **$

by $(\text{auto simp: } Qsq\text{-def decomp-def Let-def sq-mx-def add.commute in-shift-image})$

mult-2)
qed

lemma *get-qt-of*: $\llbracket \text{sq-mx } n \text{ mx}; (i,j) \in \text{sq } n \rrbracket \implies \text{get } n \text{ (qt-of } n \text{ mx)} \ i \ j = \text{mx } ! \ i \ ! \ j$
proof(*safe,induction n arbitrary: mx i j*)
 case 0
 then show ?*case* **by** (*auto simp: sq-mx-0 split: if-splits*)
next
 case (*Suc n*)
 obtain *mx0 mx1 mx2 mx3* **where** *: (*mx0,mx1,mx2,mx3*) = *decomp n mx* **by**
(*metis prod-cases4*)
 note ** = *decomp-sq-mx[OF Suc.prem1] **
 show ?*case* **using** *Suc * ***
 by(*simp add: decomp-def Let-def get-Qc height-qt-of select-def sq-mx-def mod-minus*)
qed

3.7 From Quadtree to Matrix

definition *Qmx* :: 'a mx \Rightarrow 'a mx \Rightarrow 'a mx \Rightarrow 'a mx \Rightarrow 'a mx **where**
Qmx mx0 mx1 mx2 mx3 = *map2 (@) mx0 mx1 @ map2 (@) mx2 mx3*

fun *mx-of* :: nat \Rightarrow 'a qtree \Rightarrow 'a mx **where**
mx-of n (*L x*) = *replicate (2^n) (replicate (2^n) x) |*
mx-of (Suc n) (Q t0 t1 t2 t3) =
 Qmx (mx-of n t0) (mx-of n t1) (mx-of n t2) (mx-of n t3)

lemma *nth-Qmx-select*: $\llbracket \text{sq-mx } n \text{ mx0}; \text{sq-mx } n \text{ mx1}; \text{sq-mx } n \text{ mx2}; \text{sq-mx } n \text{ mx3};$
 $i < 2 * 2^n; j < 2 * 2^n \rrbracket \implies$
 Qmx mx0 mx1 mx2 mx3 ! i ! j = *select (i < 2^n) (j < 2^n) mx0 mx1 mx2 mx3*
 !*(i mod 2^n) ! (j mod 2^n)*
by(*auto simp: sq-mx-def Qmx-def select-def nth-append mod-minus*)

lemma *sq-mx-mx-of*: $\text{height } t \leq n \implies \text{sq-mx } n \text{ (mx-of } n \ t)$
by(*induction n t rule: mx-of.induct*)
 (*auto simp: sq-mx-def Qmx-def mult-2 elim: in-set-zipE*)

lemma *mx-of-points*: $\text{height } t \leq n \implies \text{points } n \ t = \{(i,j) \in \text{sq } n. \text{mx-of } n \ t \ ! \ i \ ! \ j\}$

proof(*induction n t rule: mx-of.induct*)
 case (*2 n t0 t1 t2 t3*)
 then show ?*case*
 by (*auto simp: Qsq-def nth-Qmx-select[of n] sq-mx-mx-of select-def in-shift-image mod-if split!: if-splits*)
qed *auto*

lemma *mx-of-get*: $\llbracket \text{height } t \leq n; (i,j) \in \text{sq } n \rrbracket \implies \text{mx-of } n \ t \ ! \ i \ ! \ j = \text{get } n \ t \ i \ j$
proof(*induction n t arbitrary: i j rule: mx-of.induct*)


```

case (2 n)
then show ?case
  by (simp add: nth-Qmx-select[of n] sq-mx-mx-of select-def)
qed auto

```

end

4 Block Matrices via Quad Trees

theory *Quad-Matrix*

imports

Complex-Main

Quad-Base

begin

There are two possible representations of matrices as quadtrees. In this file we use the standard quadtree with two constructors L and Q . $L x$ represents the x -diagonal matrix of arbitrary dimension. In particular $L 0$ is the "empty" case. Because $L x$ can be of arbitrary dimension, it can be added and multiplied with Q .

In the second representation (not covered in this theory) $L x$ is the 1×1 matrix x . The advantage is that there are fewer cases in function definitions because one cannot add/multiply L and Q : they have different dimensions. However, $L 0$ is special: it still represents the 0 matrix of arbitrary dimension. This leads to a more complicated invariant wrt dimension. Or one introduces a new constructor, eg *Empty*.

4.1 Square Matrices

type-synonym $ma = nat \Rightarrow nat \Rightarrow real$

Implicitly entries outside the dimensions of the matrix are 0. This is maintained by addition; multiplication and diagonal need an explicit argument n to maintain it.

definition $mk\text{-}sq :: nat \Rightarrow ma \Rightarrow ma$ **where**

$mk\text{-}sq\ n\ a = (\lambda i\ j. \text{if } i < 2^n \wedge j < 2^n \text{ then } a\ i\ j \text{ else } 0)$

abbreviation $sq\text{-}ma\ n\ (a :: ma) \equiv (\forall i\ j. 2^n \leq i \vee 2^n \leq j \longrightarrow a\ i\ j = 0)$

Without $mk\text{-}sq$ a number of lemmas like *mult-ma-diag-ma-diag-ma* don't hold.

definition $diag\text{-}ma :: nat \Rightarrow real \Rightarrow ma$ **where**

$diag\text{-}ma\ n\ x = mk\text{-}sq\ n\ (\lambda i\ j. \text{if } i=j \text{ then } x \text{ else } 0)$

definition $add\text{-}ma :: ma \Rightarrow ma \Rightarrow ma$ **where**

$add_ma\ a\ b = (\lambda i\ j. a\ i\ j + b\ i\ j)$

definition $mult_ma :: nat \Rightarrow ma \Rightarrow ma \Rightarrow ma$ **where**
 $mult_ma\ n\ a\ b = (\lambda i\ j. \sum_{k=0..<2^{\widehat{n}}} a\ i\ k * b\ k\ j)$

4.2 Matrix Lemmas

lemma $add_ma_diag_ma[simp]$: $add_ma\ (diag_ma\ n\ x)\ (diag_ma\ n\ y) = diag_ma\ n\ (x+y)$

by ($simp\ add$: $diag_ma_def\ add_ma_def\ mk_sq_def\ fun_eq_iff$)

lemma $add_ma_diag_ma-0[simp]$: $add_ma\ (diag_ma\ n\ 0)\ a = a$

by ($auto\ simp\ add$: $add_ma_def\ diag_ma_def\ mk_sq_def\ fun_eq_iff$)

lemma $add_ma_diag_ma-02[simp]$: $add_ma\ a\ (diag_ma\ n\ 0) = a$

by ($auto\ simp\ add$: $add_ma_def\ diag_ma_def\ mk_sq_def\ fun_eq_iff$)

lemma $mult_ma_diag_ma-0[simp]$: $mult_ma\ n\ (diag_ma\ n\ 0)\ a = diag_ma\ n\ 0$

by ($auto\ simp\ add$: $mult_ma_def\ diag_ma_def\ mk_sq_def\ fun_eq_iff$)

lemma $mult_ma_diag_ma-02[simp]$: $mult_ma\ n\ a\ (diag_ma\ n\ 0) = diag_ma\ n\ 0$

by ($auto\ simp\ add$: $mult_ma_def\ diag_ma_def\ mk_sq_def\ fun_eq_iff$)

lemma $mult_ma_diag_ma_diag_ma[simp]$: $mult_ma\ n\ (diag_ma\ n\ x)\ (diag_ma\ n\ y) = diag_ma\ n\ (x*y)$

apply ($auto\ simp\ add$: $mult_ma_def\ diag_ma_def\ mk_sq_def\ fun_eq_iff\ sum.neutral$)

subgoal for i

apply ($simp\ add$: $sum.remove[where\ x=i]$)

done

done

4.3 Real Quad Trees and Abstraction to Matrices

type-synonym $qtr = real\ qtree$

fun $compressed :: qtr \Rightarrow bool$ **where**

$compressed\ (L\ x) = True$ |

$compressed\ (Q\ (L\ x0)\ (L\ x1)\ (L\ x2)\ (L\ x3)) = (\neg (x1=0 \wedge x2=0 \wedge x0=x3))$ |

$compressed\ (Q\ t0\ t1\ t2\ t3) = (compressed\ t0 \wedge compressed\ t1 \wedge compressed\ t2 \wedge compressed\ t3)$

lemma $compressed-Q$:

$compressed\ (Q\ t1\ t2\ t3\ t4) \implies (compressed\ t1 \wedge compressed\ t2 \wedge compressed\ t3 \wedge compressed\ t4)$

by ($cases\ Q\ t1\ t2\ t3\ t4\ rule: compressed.cases$)($auto$)

definition $Qma :: nat \Rightarrow ma \Rightarrow ma \Rightarrow ma \Rightarrow ma \Rightarrow ma$ **where**

$Qma\ n\ a\ b\ c\ d =$

$(\lambda i\ j. \text{if } i < 2^{\widehat{n}} \text{ then if } j < 2^{\widehat{n}} \text{ then } a\ i\ j \text{ else } b\ i\ (j - 2^{\widehat{n}}) \text{ else}$

$\text{if } j < 2^{\widehat{n}} \text{ then } c\ (i - 2^{\widehat{n}})\ j \text{ else } d\ (i - 2^{\widehat{n}})\ (j - 2^{\widehat{n}}))$

lemma *add-ma-Qma*:

$add\text{-}ma\ (Qma\ n\ a\ b\ c\ d)\ (Qma\ n\ a'\ b'\ c'\ d') =$
 $Qma\ n\ (add\text{-}ma\ a\ a')\ (add\text{-}ma\ b\ b')\ (add\text{-}ma\ c\ c')\ (add\text{-}ma\ d\ d')$
by(*simp add: Qma-def add-ma-def mk-sq-def fun-eq-iff*)

lemma *add-ma-diag-ma-Qma*: $add\text{-}ma\ (diag\text{-}ma\ (Suc\ n)\ x)\ (Qma\ n\ a\ b\ c\ d) =$

$Qma\ n\ (add\text{-}ma\ (diag\text{-}ma\ n\ x)\ a)\ b\ c\ (add\text{-}ma\ (diag\text{-}ma\ n\ x)\ d)$
by(*auto simp add: Qma-def diag-ma-def add-ma-def mk-sq-def fun-eq-iff*)

lemma *add-ma-Qma-diag-ma*: $add\text{-}ma\ (Qma\ n\ a\ b\ c\ d)\ (diag\text{-}ma\ (Suc\ n)\ x) =$

$Qma\ n\ (add\text{-}ma\ a\ (diag\text{-}ma\ n\ x))\ b\ c\ (add\text{-}ma\ d\ (diag\text{-}ma\ n\ x))$
by(*auto simp add: Qma-def diag-ma-def add-ma-def mk-sq-def fun-eq-iff*)

lemma *diag-ma-Suc*: $diag\text{-}ma\ (Suc\ n)\ x = Qma\ n\ (diag\text{-}ma\ n\ x)\ (diag\text{-}ma\ n\ 0)$
 $(diag\text{-}ma\ n\ 0)\ (diag\text{-}ma\ n\ x)$

by(*auto simp add: diag-ma-def Qma-def mk-sq-def fun-eq-iff*)

Abstraction function:

fun *ma* :: $nat \Rightarrow qtr \Rightarrow ma$ **where**

$ma\ n\ (L\ x) = diag\text{-}ma\ n\ x \mid$
 $ma\ (Suc\ n)\ (Q\ t0\ t1\ t2\ t3) =$
 $Qma\ n\ (ma\ n\ t0)\ (ma\ n\ t1)\ (ma\ n\ t2)\ (ma\ n\ t3)$

4.4 Matrix Operations on Trees

fun *Qc* :: $qtr \Rightarrow qtr \Rightarrow qtr \Rightarrow qtr \Rightarrow qtr$ **where**

$Qc\ (L\ x0)\ (L\ x1)\ (L\ x2)\ (L\ x3) =$
 $(if\ x1=0 \wedge x2=0 \wedge x0=x3\ then\ L\ x0\ else\ Q\ (L\ x0)\ (L\ x1)\ (L\ x2)\ (L\ x3)) \mid$
 $Qc\ t1\ t2\ t3\ t4 = Q\ t1\ t2\ t3\ t4$

lemma *ma-Suc-Qc*: $ma\ (Suc\ n)\ (Qc\ t0\ t1\ t2\ t3) = ma\ (Suc\ n)\ (Q\ t0\ t1\ t2\ t3)$

by(*induction t0 t1 t2 t3 rule: Qc.induct*)(*auto simp: diag-ma-Suc*)

lemma *compressed-Qc*:

$compressed\ (Qc\ t0\ t1\ t2\ t3) = (compressed\ t0 \wedge compressed\ t1 \wedge compressed\ t2$
 $\wedge\ compressed\ t3)$

by(*induction t0 t1 t2 t3 rule: Qc.induct*)(*auto*)

lemma *height-Qc-Q*:

$height\ (Qc\ t0\ t1\ t2\ t3) \leq height\ (Q\ t0\ t1\ t2\ t3)$

proof(*induction t0 t1 t2 t3 rule: Qc.induct*)

case $(1\ x0\ x1\ x2\ x3)$

then show *?case* **by** *simp*

qed (*insert Qc.simps,presburger+*)

fun *add* :: $qtr \Rightarrow qtr \Rightarrow qtr$ **where**

$add\ (Q\ s0\ s1\ s2\ s3)\ (Q\ t0\ t1\ t2\ t3) = Qc\ (add\ s0\ t0)\ (add\ s1\ t1)\ (add\ s2\ t2)$
 $(add\ s3\ t3) \mid$
 $add\ (L\ x)\ (L\ y) = L(x+y) \mid$

$add (L x) (Q t0 t1 t2 t3) = Qc (add (L x) t0) t1 t2 (add (L x) t3) \mid$
 $add (Q t0 t1 t2 t3) (L x) = Qc (add t0 (L x)) t1 t2 (add t3 (L x))$

fun *mult* :: *qtr* \Rightarrow *qtr* \Rightarrow *qtr* **where**
 $mult (Q s0 s1 s2 s3) (Q t0 t1 t2 t3) =$
 $Qc (add (mult s0 t0) (mult s1 t2))$
 $(add (mult s0 t1) (mult s1 t3))$
 $(add (mult s2 t0) (mult s3 t2))$
 $(add (mult s2 t1) (mult s3 t3)) \mid$
 $mult (L x) (Q t0 t1 t2 t3) =$
 $Qc (mult (L x) t0)$
 $(mult (L x) t1)$
 $(mult (L x) t2)$
 $(mult (L x) t3) \mid$
 $mult (Q t0 t1 t2 t3) (L x) =$
 $Qc (mult t0 (L x))$
 $(mult t1 (L x))$
 $(mult t2 (L x))$
 $(mult t3 (L x)) \mid$
 $mult (L x) (L y) = L(x*y)$

Initialization of *qtr* from *ma*

fun *qtr* :: *nat* \Rightarrow *ma* \Rightarrow *qtr* **where**
 $qtr 0 a = L(a 0 0) \mid$
 $qtr (Suc n) a =$
 $(let t0 = qtr n a; t1 = qtr n (\lambda i j. a i (j+2\hat{n}));$
 $t2 = qtr n (\lambda i j. a (i+2\hat{n}) j); t3 = qtr n (\lambda i j. a (i+2\hat{n}) (j+2\hat{n}));$
 $in Q t0 t1 t2 t3)$

4.5 Correctness of Quad Tree Implementations

4.5.1 *add*

lemma *ma-add*: $\llbracket height\ s \leq n; height\ t \leq n \rrbracket \implies$

$ma\ n\ (add\ s\ t) = add\ ma\ (ma\ n\ s)\ (ma\ n\ t)$

proof(*induction s t arbitrary: n rule: add.induct*)

case 1

then show *?case* **by**(*simp add: less-eq-nat.simps(2) add-ma-Qma ma-Suc-Qc split: nat.splits*)

next

case 2

then show *?case* **by**(*simp*)

next

case 3

then show *?case* **by**(*simp add: add-ma-diag-ma-Qma ma-Suc-Qc less-eq-nat.simps(2) split: nat.splits*)

next

case 4

then show *?case* **by**(*simp add: add-ma-Qma-diag-ma ma-Suc-Qc less-eq-nat.simps(2) split: nat.splits*)

qed

lemma *height-add*: $\text{height } (\text{add } s \ t) \leq \max (\text{height } s) (\text{height } t)$

proof(*induction s t rule: add.induct*)

case (1 *s1 s2 s3 s4 t1 t2 t3 t4*)

thus ?*case*

using *height-Qc-Q*[of *add s1 t1 add s2 t2 add s3 t3 add s4 t4*]

by (*auto simp: max.coboundedI1 max.coboundedI2*)

simp del: max.absorb1 max.absorb2 max.absorb3 max.absorb4 elim!: le-trans)

next

case (3 *x t1 t2 t3 t4*)

thus ?*case* **using** *height-Qc-Q*[of *add (L x) t1 t2 t3 add (L x) t4*]

by *auto*

next

case (4 *t1 t2 t3 t4 x*)

then show ?*case* **using** *height-Qc-Q*[of *add t1 (L x) t2 t3 add t4 (L x)*]

by *auto*

qed *simp*

lemma *compressed-add*: $\llbracket \text{compressed } s; \text{ compressed } t \rrbracket \implies \text{compressed } (\text{add } s \ t)$

by(*induction s t rule: add.induct*) (*auto simp: compressed-Qc dest: compressed-Q*)

lemma *Max4*: $\text{Max}\{n0, n1, n2, n3\} = \max n0 (\max n1 (\max n2 n3))$ **by** *simp*

lemma *height-mult*: $\text{height } (\text{mult } s \ t) \leq \max (\text{height } s) (\text{height } t)$

proof(*induction s t rule: mult.induct*)

case (1 *s1 s2 s3 s4 t1 t2 t3 t4*)

let ?*m11* = *mult s1 t1* **let** ?*m23* = *mult s2 t3* **let** ?*m12* = *mult s1 t2* **let** ?*m24* = *mult s2 t4*

let ?*m31* = *mult s3 t1* **let** ?*m43* = *mult s4 t3* **let** ?*m32* = *mult s3 t2* **let** ?*m44* = *mult s4 t4*

show ?*case*

using 1 *height-Qc-Q*[of *add ?m11 ?m23 add ?m12 ?m24 add ?m31 ?m43 add ?m32 ?m44*]

height-add[of ?*m11* ?*m23*] *height-add*[of ?*m12* ?*m24*] *height-add*[of ?*m31* ?*m43*] *height-add*[of ?*m32* ?*m44*]

unfolding *mult.simps height-qtreesimps One-nat-def add-Suc-right add-0-right max-Suc-Suc Max4*

by (*smt (z3) order.trans le-max-iff-disj not-less-eq-eq*)

next

case (2 *x t0 t1 t2 t3*)

thus ?*case* **using** *height-Qc-Q*[of *mult (L x) t0 mult (L x) t1 mult (L x) t2 mult (L x) t3*]

by (*simp*)

next

case (3 *t0 t1 t2 t3 x*)

thus ?*case* **using** *height-Qc-Q*[of *mult t0 (L x) mult t1 (L x) mult t2 (L x) mult t3 (L x)*]

by *simp*

qed (simp)

4.5.2 mult

lemma *bij-betw-minus-ivlco-nat*: $n \leq a \implies C = \{a - n..<b - n\} \implies \text{bij-betw } (\lambda k::\text{nat. } k - n) \{a..<b\} C$

by(auto simp add: *bij-betw-def inj-on-def image-minus-const-atLeastLessThan-nat*)

lemma *mult-ma-Qma-Qma*:

$$\begin{aligned} \text{mult-ma } (\text{Suc } n) (Qma \ n \ a \ b \ c \ d) (Qma \ n \ a' \ b' \ c' \ d') = \\ (Qma \ n \ (\text{add-ma } (\text{mult-ma } \ n \ a \ a') (\text{mult-ma } \ n \ b \ c')) \\ (\text{add-ma } (\text{mult-ma } \ n \ a \ b') (\text{mult-ma } \ n \ b \ d')) \\ (\text{add-ma } (\text{mult-ma } \ n \ c \ a') (\text{mult-ma } \ n \ d \ c')) \\ (\text{add-ma } (\text{mult-ma } \ n \ c \ b') (\text{mult-ma } \ n \ d \ d'))) \end{aligned}$$

by(auto simp add: *mult-ma-def add-ma-def Qma-def mk-sq-def fun-eq-iff sum-Un ivl-disj-un(17)[of 0 2ⁿ 2*2ⁿ,symmetric]*)

*intro:sum.reindex-bij-betw[of $\lambda k. k - 2^n \{2^n..<2 * 2^n\} \{0..<2^n\}$, OF *bij-betw-minus-ivlco-nat*]*)

lemma *ma-mult*: $\llbracket \text{height } s \leq n; \text{height } t \leq n \rrbracket \implies$

$$\text{ma } n \ (\text{mult } s \ t) = \text{mult-ma } n \ (\text{ma } n \ s) \ (\text{ma } n \ t)$$

proof(*induction s t arbitrary: n rule: mult.induct*)

case (1 *s1 s2 s3 s4 t1 t2 t3 t4*) **thus** ?case

by(simp add: *mult-ma-Qma-Qma ma-add ma-Suc-Qc le-trans[OF height-mult] less-eq-nat.simps(2) split: nat.splits*)

next

case 2 **thus** ?case

by(simp add: *diag-ma-Suc ma-Suc-Qc mult-ma-Qma-Qma less-eq-nat.simps(2) split: nat.splits*)

next

case 3 **thus** ?case

by(simp add: *diag-ma-Suc ma-Suc-Qc mult-ma-Qma-Qma less-eq-nat.simps(2) split: nat.splits*)

qed simp

lemma *compressed-mult*: $\llbracket \text{compressed } s; \text{compressed } t \rrbracket \implies \text{compressed } (\text{mult } s \ t)$

proof(*induction s t rule: mult.induct*)

case 1 **thus** ?case **unfolding** *mult.simps* **by** (*metis compressed-Q compressed-Qc compressed-add*)

next

case 2 **thus** ?case **unfolding** *mult.simps* **by** (*metis compressed-Q compressed-Qc*)

next

case 3 **thus** ?case **unfolding** *mult.simps* **by** (*metis compressed-Q compressed-Qc*)

next

case 4 **thus** ?case **by** *simp*

qed

end

5 K-dimensional Region Trees

theory *KD-Region-Tree*

imports

HOL-Library.NList

HOL-Library.Tree

begin

lemma *nlists-Suc*: $nlists (Suc\ n)\ A = (\bigcup_{a \in A} (\#)\ a\ \text{'nlists}\ n\ A)$

by(*auto simp: set-eq-iff image-iff in-nlists-Suc-iff*)

lemma *in-nlists-UNIV*: $xs \in nlists\ k\ UNIV \iff length\ xs = k$

unfolding *nlists-def* **by**(*auto*)

lemma *nlists-singleton*: $nlists\ n\ \{a\} = \{replicate\ n\ a\}$

unfolding *nlists-def* **by**(*auto simp: replicate-length-same dest!: subset-singletonD*)

Generalizes quadtrees. Instead of having 2^n direct children of a node, the children are arranged in a binary tree where each *Split* splits along one dimension.

datatype *'a kdt* = *Box 'a* | *Split 'a kdt 'a kdt*

datatype-compat *kdt*

type-synonym *kdtb* = *bool kdt*

A *kdt* is most easily explained by showing how quad trees are represented: $Q\ t0\ t1\ t2\ t3$ becomes $Split\ (Split\ t0'\ t1')\ (Split\ t2'\ t3')$ where ti' is the representation of ti ; $L\ a$ becomes $Box\ a$. In general, each level of an abstract k dimensional tree subdivides space into 2^k subregions. This subdivision is represented by a *kdt* of depth at most k . Further subdivisions of the subregions are seamlessly represented as the subtrees at depth k . $Box\ a$ represents a subregion entirely filled with a 's. In contrast to quad trees, cubes can also occur half way down the subdivision. For example, $Q\ (L\ a)\ (L\ b)\ (L\ c)$ becomes $Split\ (Box\ a)\ (Split\ (Box\ b)\ (Box\ c))$.

instantiation *kdt* :: *(type)height*

begin

fun *height-kdt* :: *'a kdt* \Rightarrow *nat* **where**

height (*Box* $-$) = 0 |

height (*Split* $l\ r$) = $\max\ (height\ l)\ (height\ r) + 1$

instance ..

end

lemma *height-0-iff*: $\text{height } t = 0 \longleftrightarrow (\exists x. t = \text{Box } x)$
by(*cases t*)*auto*

definition *bits* :: $\text{nat} \Rightarrow \text{bool list set}$ **where**
bits n = nlists n UNIV

lemma *bits-0*[*code*]: $\text{bits } 0 = \{\{\}\}$
by(*simp add:bits-def*)

lemma *bits-Suc*[*code*]:
bits (Suc n) = (let B = bits n in (#) True ‘ B \cup (#) False ‘ B)
by(*simp-all add: bits-def nlists-Suc UN-bool-eq Let-def*)

5.1 Subtree

fun *subtree* :: $'a \text{ kdt} \Rightarrow \text{bool list} \Rightarrow 'a \text{ kdt}$ **where**
subtree t [] = t |
subtree (Box x) _ = Box x |
subtree (Split l r) (b#bs) = subtree (if b then r else l) bs

lemma *subtree-Box*[*simp*]: $\text{subtree } (\text{Box } x) \text{ bs} = \text{Box } x$
by(*cases bs*)*auto*

lemma *height-subtree*: $\text{height } (\text{subtree } t \text{ bs}) \leq \text{height } t - \text{length } \text{bs}$
by(*induction t bs rule: subtree.induct*) *auto*

lemma *height-subtree2*: $\llbracket \text{height } t \leq k * (\text{Suc } n); \text{length } \text{bs} = k \rrbracket \Longrightarrow \text{height } (\text{subtree } t \text{ bs}) \leq k * n$
using *height-subtree*[*of t bs*] **by** *auto*

lemma *subtree-Split-Box*: $\text{length } \text{bs} \neq 0 \Longrightarrow \text{subtree } (\text{Split } (\text{Box } b) (\text{Box } b)) \text{ bs} = \text{Box } b$
by(*auto simp: neq-Nil-conv*)

5.2 Shifting a coordinate by a boolean vector

definition *mv* :: $\text{nat} \Rightarrow \text{bool list} \Rightarrow \text{nat list} \Rightarrow \text{nat list}$ **where**
mv d = map2 ($\lambda b x. x + (\text{if } b \text{ then } 0 \text{ else } d)$)

lemma *map-zip1*: $\llbracket \text{length } \text{xs} = \text{length } \text{ys}; \forall p \in \text{set}(\text{zip } \text{xs } \text{ys}). f p = \text{fst } p \rrbracket \Longrightarrow$
 $\text{map } f (\text{zip } \text{xs } \text{ys}) = \text{xs}$
by (*metis (no-types, lifting) map-eq-conv map-fst-zip*)

lemma *map-mv1*: $\llbracket \text{ps} \in \text{nlists } (\text{length } \text{bs}) \{0..<n\}; \text{length } \text{ps} = \text{length } \text{bs} \rrbracket$
 $\Longrightarrow \text{map } (\lambda i. i < n) (\text{mv } (n) \text{ bs } \text{ps}) = \text{bs}$

by(*fastforce simp: mv-def intro!: map-zip1 dest: set-zip-rightD nlistsE-set split: if-splits*)

lemma *map-zip2*: $\llbracket \text{length } xs = \text{length } ys; \forall p \in \text{set}(\text{zip } xs \text{ } ys). f \text{ } p = \text{snd } p \rrbracket \implies \text{map } f (\text{zip } xs \text{ } ys) = ys$
by (*metis (no-types, lifting) map-eq-conv map-snd-zip*)

lemma *map-mv2*: $\llbracket ps \in \text{nlists } (\text{length } bs) \{0..<2^{\hat{n}}\} \rrbracket \implies \text{map } (\lambda x. x \text{ mod } 2^{\hat{n}}) (mv (2^{\hat{n}}) bs ps) = ps$
by(*fastforce simp: mv-def dest: set-zip-rightD nlistsE-set intro!: map-zip2*)

lemma *mv-map-map*: $\text{set } ps \subseteq \{0..<2 * n\} \implies mv (n) (\text{map } (\lambda x. x < n) ps)$
 $(\text{map } (\lambda x. x \text{ mod } n) ps) = ps$
unfolding *nlists-def mv-def*
by(*auto simp: map-eq-conv[where xs=ps and g=id,simplified] map2-map-map not-less le-iff-add*)

lemma *mv-in-nlists*:
 $\llbracket p \in \text{nlists } k \{0..<2^{\hat{n}}\}; bs \in \text{bits } k \rrbracket \implies mv (2^{\hat{n}}) bs p \in \text{nlists } k \{0..<2 * 2^{\hat{n}}\}$
unfolding *mv-def nlists-def bits-def*
by (*fastforce dest: set-zip-rightD*)

lemma *in-nlists2D*: $xs \in \text{nlists } k \{0..<2 * 2^{\hat{n}}\} \implies \exists bs \in \text{bits } k. xs \in mv (2^{\hat{n}}) bs \text{ ' nlists } k \{0..<2^{\hat{n}}\}$
unfolding *nlists-def bits-def image-def*
apply(*rule bexI[where x = map (\lambda x. x < 2^{\hat{n}}) xs]*)
apply(*simp*)
apply(*rule exI[where x = map (\lambda i. i mod 2^{\hat{n}}) xs]*)
apply (*auto simp add: mv-map-map*)
done

lemma *nlists2-simp*: $\text{nlists } k \{0..<2 * 2^{\hat{n}}\} = (\bigcup bs \in \text{bits } k. mv (2^{\hat{n}}) bs \text{ ' nlists } k \{0..<2^{\hat{n}}\})$
by (*auto simp: mv-in-nlists in-nlists2D*)

lemma *in-mv-image*: $\llbracket ps \in \text{nlists } k \{0..<2*2^{\hat{n}}\}; Ps \subseteq \text{nlists } k \{0..<2^{\hat{n}}\}; bs \in \text{bits } k \rrbracket \implies ps \in mv (2^{\hat{n}}) bs \text{ ' } Ps \iff \text{map } (\lambda x. x \text{ mod } 2^{\hat{n}}) ps \in Ps \wedge (bs = \text{map } (\lambda i. i < 2^{\hat{n}}) ps)$
by (*auto simp: map-mv1 map-mv2 mv-map-map bits-def intro!: image-eqI*)

5.3 Points in a tree

fun *cube* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list set}$ **where**
cube *k n* = $\text{nlists } k \{0..<2^{\hat{n}}\}$

fun *points* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{kdtb} \Rightarrow \text{nat list set}$ **where**
points *k n* (*Box* *b*) = (*if* *b* *then* *cube* *k n* *else* $\{\}$) |

$points\ k\ (Suc\ n)\ t = (\bigcup bs \in bits\ k.\ mv\ (2^{\wedge}n)\ bs\ 'points\ k\ n\ (subtree\ t\ bs))$

lemma *points-Suc*: $points\ k\ (Suc\ n)\ t = (\bigcup bs \in bits\ k.\ mv\ (2^{\wedge}n)\ bs\ 'points\ k\ n\ (subtree\ t\ bs))$
by(*cases t*) (*simp-all add: nlists2-simp*)

lemma *points-subset*: $height\ t \leq k*n \implies points\ k\ n\ t \subseteq nlists\ k\ \{0..<2^{\wedge}n\}$
proof(*induction k n t rule: points.induct*)
case ($2\ k\ n\ l\ r$)
have $\bigwedge bs.\ bs \in bits\ k \implies height\ (subtree\ (Split\ l\ r)\ bs) \leq k*n$
unfolding *bits-def* **using** *2.prem1 height-subtree2 in-nlists-UNIV* **by** *blast*
with *2.IH* **show** *?case*
by(*auto intro: mv-in-nlists dest: subsetD*)
qed *auto*

5.4 Compression

Compressing Split:

fun *SplitC* :: $'a\ kdt \Rightarrow 'a\ kdt \Rightarrow 'a\ kdt$ **where**
SplitC (*Box b1*) (*Box b2*) = (*if b1=b2 then Box b1 else Split (Box b1) (Box b2)*) |
SplitC t1 t2 = Split t1 t2

fun *compressed* :: $'a\ kdt \Rightarrow bool$ **where**
compressed (*Box -*) = *True* |
compressed (*Split l r*) = (*compressed l* \wedge *compressed r* \wedge $\neg(\exists b.\ l = Box\ b \wedge r = Box\ b)$)

lemma *compressedI*: $\llbracket compressed\ l;\ compressed\ r \rrbracket \implies compressed\ (SplitC\ l\ r)$
by(*induction l r rule: SplitC.induct*) *auto*

lemma *subtree-SplitC*:
 $1 \leq length\ bs \implies subtree\ (SplitC\ l\ r)\ bs = subtree\ (Split\ l\ r)\ bs$
by(*induction l r rule: SplitC.induct*)
(simp-all add: subtree-Split-Box Suc-le-eq)

lemma *height-SplitC*: $height(SplitC\ l\ r) \leq Suc\ (max\ (height\ l)\ (height\ r))$
by(*cases (l,r) rule: SplitC.cases*)(*auto*)

lemma *height-SplitC2*: $\llbracket height\ l \leq n;\ height\ r \leq n \rrbracket \implies height(SplitC\ l\ r) \leq Suc\ n$
using *height-SplitC[of l r]* **by** *simp*

5.5 Extracting a point from a tree

Also the abstraction function.

fun *get* :: $nat \Rightarrow 'a\ kdt \Rightarrow nat\ list \Rightarrow 'a$ **where**
get - (Box b) - = b |
get (Suc n) t ps = get n (subtree t (map (\i. i < 2^{\wedge}n) ps)) (map (\i. i mod 2^{\wedge}n) ps)

```

lemma get-Suc: get (Suc n) t ps =
  get n (subtree t (map ( $\lambda i. i < 2^{\wedge} n$ ) ps)) (map ( $\lambda i. i \bmod 2^{\wedge} n$ ) ps)
by(cases t)auto

lemma points-get:  $\llbracket \text{height } t \leq k * n; ps \in \text{nlists } k \{0..<2^{\wedge} n\} \rrbracket \implies$ 
  get n t ps = (ps  $\in$  points k n t)
proof(induction n arbitrary: k t ps)
  case 0
  then show ?case by(clarsimp simp add: height-0-iff)
next
  case (Suc n)
  show ?case
  proof (cases t)
    case Box
    thus ?thesis using Suc.prem1(2) by(simp)
  next
    case (Split l r)
    obtain k0 where k = Suc k0 using Suc.prem1(1) Split
    by(cases k) auto
    hence ps  $\neq$  []
    using Suc.prem1(2) by (auto simp: in-nlists-Suc-iff)
    then show ?thesis using Suc.prem1 Split Suc.IH[OF height-subtree2[OF Suc.prem1(1)]]
  in-nlists2D
  by(simp add: height-subtree2 in-mv-image points-subset bits-def)
qed
qed

```

5.6 Modifying a point in a tree

```

fun modify :: ('a kdt  $\Rightarrow$  'a kdt)  $\Rightarrow$  bool list  $\Rightarrow$  'a kdt  $\Rightarrow$  'a kdt where
  modify f [] t = f t |
  modify f (b # bs) (Split l r) = (if b then SplitC l (modify f bs r) else SplitC (modify
  f bs l) r) |
  modify f (b # bs) (Box a) =
    (let t = modify f bs (Box a) in if b then SplitC (Box a) t else SplitC t (Box a))

fun put :: nat list  $\Rightarrow$  'a  $\Rightarrow$  nat  $\Rightarrow$  'a kdt  $\Rightarrow$  'a kdt where
  put ps a 0 (Box -) = Box a |
  put ps a (Suc n) t = modify (put (map ( $\lambda i. i \bmod 2^{\wedge} n$ ) ps) a n) (map ( $\lambda i. i <$ 
   $2^{\wedge} n$ ) ps) t

```

```

lemma height-modify:  $\llbracket \forall t. \text{height } t \leq nk \longrightarrow \text{height } (f t) \leq nk;$ 
  height t  $\leq k + nk$ ; length bs = k]
 $\implies$  height (modify f bs t)  $\leq k + nk$ 
apply(induction f bs t arbitrary: k rule: modify.induct)
by (auto simp: height-SplitC2 Let-def)

```

lemma *height-put*: $\text{height } t \leq n * \text{length } ps \implies \text{height } (\text{put } ps \text{ a } n \ t) \leq n * \text{length } ps$

proof(*induction ps a n t rule: put.induct*)

case 2

then show ?*case* **by** (*auto simp: height-modify*)

qed *auto*

lemma *subtree-modify*: $\llbracket \text{length } bs' = \text{length } bs \rrbracket$

$\implies \text{subtree } (\text{modify } f \ bs \ t) \ bs' = (\text{if } bs' = bs \ \text{then } f(\text{subtree } t \ bs) \ \text{else } \text{subtree } t \ bs')$

apply(*induction f bs t arbitrary: bs' rule: modify.induct*)

apply(*auto simp add: length-Suc-conv Let-def subtree-SplitC split: if-splits*)

done

lemma *mod-eq1*: $\llbracket y < 2 * n; ya < 2 * n; \neg ya < n; \neg y < n; ya \bmod n = y \bmod n \rrbracket$

$\implies ya = (y::nat)$

by(*simp add: mod-if mult-2 split: if-splits*)

lemma *nlist-eq-mod*: $\llbracket ps \in \text{nlists } k \ \{0..<(2::nat) * 2^{\wedge} n\}; ps' \in \text{nlists } k \ \{0..<2 * 2^{\wedge} n\};$

$\text{map } (\lambda i. i < 2^{\wedge} n) \ ps' = \text{map } (\lambda i. i < 2^{\wedge} n) \ ps; ps' \neq ps \rrbracket \implies$

$\text{map } (\lambda i. i \bmod 2^{\wedge} n) \ ps' \neq \text{map } (\lambda i. i \bmod 2^{\wedge} n) \ ps$

apply(*induction k arbitrary: ps ps'*)

apply *simp*

apply (*fastforce simp: in-nlists-Suc-iff mod-eq1*)

done

lemma *get-put*: $\llbracket \text{height } t \leq k*n; ps \in \text{cube } k \ n; ps' \in \text{cube } k \ n \rrbracket \implies$

$\text{get } n \ (\text{put } ps \ \text{a } n \ t) \ ps' = (\text{if } ps' = ps \ \text{then } a \ \text{else } \text{get } n \ t \ ps')$

proof(*induction ps a n t arbitrary: ps' rule: put.induct*)

case 1

then show ?*case* **by** (*simp add: nlists-singleton*)

next

case 2

thus ?*case* **using** *in-nlists2D*

by(*auto simp add: subtree-modify get-Suc height-subtree2 nlist-eq-mod in-mv-image*)

qed *auto*

lemma *compressed-modify*: $\llbracket \text{compressed } t; \text{compressed } (f \ (\text{subtree } t \ bs)) \rrbracket \implies$
 $\text{compressed } (\text{modify } f \ bs \ t)$

by(*induction f bs t rule: modify.induct*) (*auto simp: compressedI Let-def*)

lemma *compressed-subtree*: $\text{compressed } t \implies \text{compressed } (\text{subtree } t \ bs)$

by(*induction t bs rule: subtree.induct*) *auto*

lemma *compressed-put*:

$\llbracket \text{height } t \leq k*n; k = \text{length } ps; \text{compressed } t \rrbracket \implies \text{compressed } (\text{put } ps \ \text{a } n \ t)$

proof(*induction ps a n t rule: put.induct*)

```

  case 1
  then show ?case by (simp)
next
  case 2
  thus ?case by (simp add: compressed-modify compressed-subtree height-subtree2)
qed auto

```

5.7 Union

```

fun union :: kdtb  $\Rightarrow$  kdtb  $\Rightarrow$  kdtb where
  union (Box b) t = (if b then Box True else t) |
  union t (Box b) = (if b then Box True else t) |
  union (Split l1 r1) (Split l2 r2) = SplitC (union l1 l2) (union r1 r2)

```

```

lemma union-Box2: union t (Box b) = (if b then Box True else t)
by(cases t) auto

```

```

lemma subtree-union: subtree (union t1 t2) bs = union (subtree t1 bs) (subtree t2
bs)
proof(induction t1 t2 arbitrary: bs rule: union.induct)
  case 2 thus ?case by(auto simp: union-Box2)
next
  case 3 thus ?case by(cases bs) (auto simp: subtree-SplitC)
qed auto

```

```

lemma points-union:
   $\llbracket \max (\text{height } t1) (\text{height } t2) \leq k * n \rrbracket \implies$ 
  points k n (union t1 t2) = points k n t1  $\cup$  points k n t2
proof(induction n arbitrary: t1 t2)
  case 0 thus ?case by(clarsimp simp add: height-0-iff)
next
  case (Suc n)
  have height t1  $\leq$  k * Suc n height t2  $\leq$  k * Suc n
  using Suc.prem1 by auto
  from height-subtree2[OF this(1)] height-subtree2[OF this(2)] show ?case
  by(auto simp: Suc.IH subtree-union points-Suc bits-def)
qed

```

```

lemma get-union:
   $\llbracket \max (\text{height } t1) (\text{height } t2) \leq \text{length } ps * n \rrbracket \implies$ 
  get n (union t1 t2) ps = (get n t1 ps  $\vee$  get n t2 ps)
proof(induction n arbitrary: t1 t2 ps)
  case 0 thus ?case by(clarsimp simp add: height-0-iff)
next
  case (Suc n)
  have height t1  $\leq$  length ps * Suc n height t2  $\leq$  length ps * Suc n
  using Suc.prem1(1) by auto
  from height-subtree2[OF this(1)] height-subtree2[OF this(2)] show ?case
  by(simp add: Suc.IH subtree-union get-Suc)

```

qed

lemma *height-union*: $\text{height } (\text{union } t1 \ t2) \leq \max (\text{height } t1) (\text{height } t2)$
by(*induction* *t1 t2 rule: union.induct*) (*auto simp: height-SplitC2*)

lemma *compressed-union*: $\text{compressed } t1 \implies \text{compressed } t2 \implies \text{compressed}(\text{union } t1 \ t2)$
by(*induction* *t1 t2 rule: union.induct*) (*simp-all add: compressedI*)

end

6 K-dimensional Region Trees - Version 2

theory *KD-Region-Tree2*

imports

HOL-Library.NList

HOL-Library.Tree

begin

lemma *nlists-Suc*: $\text{nlists } (\text{Suc } n) \ A = (\bigcup a \in A. (\#) \ a \ \text{nlists } \ n \ A)$
by(*auto simp: set-eq-iff image-iff in-nlists-Suc-iff*)

lemma *in-nlists-UNIV*: $xs \in \text{nlists } \ k \ \text{UNIV} \iff \text{length } \ xs = k$
unfolding *nlists-def* **by**(*auto*)

datatype *'a kdt* = *Box 'a* | *Split 'a kdt 'a kdt*

datatype-compat *kdt*

type-synonym *kdtb* = *bool kdt*

A *kdt* is most easily explained by showing how quad trees are represented: $Q \ t0 \ t1 \ t2 \ t3$ becomes $\text{Split } (\text{Split } \ t0' \ t1') (\text{Split } \ t2' \ t3')$ where ti' is the representation of ti ; $L \ a$ becomes $\text{Box } a$. In general, each level of an abstract k dimensional tree subdivides space into 2^k subregions. This subdivision is represented by a *kdt* of depth at most k . Further subdivisions of the subregions are seamlessly represented as the subtrees at depth k . $\text{Box } a$ represents a subregion entirely filled with a 's. In contrast to quad trees, cubes can also occur half way down the subdivision. For example, $Q \ (L \ a) \ (L \ b) \ (L \ c)$ becomes $\text{Split } (\text{Box } a) (\text{Split } (\text{Box } b) (\text{Box } c))$.

instantiation *kdt* :: $(\text{type})\text{height}$

begin

fun *height-kdt* :: $'a \ kdt \Rightarrow \text{nat}$ **where**

$height (Box _) = 0 \mid$
 $height (Split \ l \ r) = \max (height \ l) (height \ r) + 1$

instance ..

end

lemma *height-0-iff*: $height \ t = 0 \longleftrightarrow (\exists x. t = Box \ x)$
by(*cases t*)*auto*

definition *bits* :: $nat \Rightarrow bool \ list \ set$ **where**
bits $n \equiv nlists \ n \ UNIV$

lemma *bits-Suc*[*code*]:
 $bits (Suc \ n) = (let \ B = bits \ n \ in \ (\#) \ True \ ' \ B \cup \ (\#) \ False \ ' \ B)$
by(*simp-all add: bits-def nlists-Suc UN-bool-eq Let-def*)

6.1 Subtree

fun *subtree* :: $'a \ kdt \Rightarrow bool \ list \Rightarrow 'a \ kdt$ **where**
 $subtree \ t \ [] = t \mid$
 $subtree (Box \ x) _ = Box \ x \mid$
 $subtree (Split \ l \ r) (b\#bs) = subtree (if \ b \ then \ r \ else \ l) \ bs$

lemma *subtree-Box*[*simp*]: $subtree (Box \ x) \ bs = Box \ x$
by(*cases bs*)*auto*

lemma *height-subtree*: $height (subtree \ t \ bs) \leq height \ t - length \ bs$
by(*induction t bs rule: subtree.induct*) *auto*

lemma *height-subtree2*: $\llbracket height \ t \leq k * (Suc \ n); length \ bs = k \rrbracket \Longrightarrow height (subtree \ t \ bs) \leq k * n$
using *height-subtree*[*of t bs*] **by** *auto*

lemma *subtree-Split-Box*: $length \ bs \neq 0 \Longrightarrow subtree (Split (Box \ b) (Box \ b)) \ bs = Box \ b$
by(*auto simp: neq-Nil-conv*)

6.2 Shifting a coordinate by a boolean vector

The ?

definition *mv* :: $bool \ list \Rightarrow nat \ list \Rightarrow nat \ list$ **where**
 $mv = map2 (\lambda b \ x. 2*x + (if \ b \ then \ 0 \ else \ 1))$

lemma *map-zip1*: $\llbracket length \ xs = length \ ys; \forall p \in set(zip \ xs \ ys). f \ p = fst \ p \rrbracket \Longrightarrow map \ f (zip \ xs \ ys) = xs$
by (*metis (no-types, lifting) map-eq-conv map-fst-zip*)

lemma *map-mv1*: $\llbracket \text{length } ps = \text{length } bs \rrbracket \implies \text{map even } (mv \text{ } bs \text{ } ps) = bs$
by(*auto simp: mv-def intro!: map-zip1 split: if-splits*)

lemma *map-zip2*: $\llbracket \text{length } xs = \text{length } ys; \forall p \in \text{set}(zip \text{ } xs \text{ } ys). f \text{ } p = \text{snd } p \rrbracket \implies$
 $\text{map } f \text{ } (zip \text{ } xs \text{ } ys) = ys$
by (*metis (no-types, lifting) map-eq-conv map-snd-zip*)

lemma *map-mv2*: $\llbracket \text{length } ps = \text{length } bs \rrbracket \implies \text{map } (\lambda x. x \text{ div } 2) (mv \text{ } bs \text{ } ps) =$
 ps
by(*auto simp: mv-def intro!: map-zip2*)

lemma *mv-map-map*: $mv (\text{map even } ps) (\text{map } (\lambda x. x \text{ div } 2) ps) = ps$
unfolding *nlists-def mv-def*
by(*auto simp add: map-eq-conv[where xs=ps and g=id,simplified] map2-map-map*)

lemma *mv-in-nlists*:
 $\llbracket p \in \text{nlists } k \{0..<2^{\wedge} n\}; bs \in \text{bits } k \rrbracket \implies mv \text{ } bs \text{ } p \in \text{nlists } k \{0..<2 * 2^{\wedge} n\}$
unfolding *mv-def nlists-def bits-def*
by (*fastforce dest: set-zip-rightD*)

lemma *in-nlists2D*: $xs \in \text{nlists } k \{0..<2 * 2^{\wedge} n\} \implies \exists bs \in \text{bits } k. xs \in mv \text{ } bs \text{ } \text{'nlists } k$
 $\{0..<2^{\wedge} n\}$
unfolding *nlists-def bits-def*
apply(*rule bexI[where x = map even xs]*)
apply(*auto simp: image-def*)[1]
apply(*rule exI[where x = map (\lambda i. i div 2) xs]*)
apply(*auto simp add: mv-map-map*)
done

lemma *nlists2-simp*: $\text{nlists } k \{0..<2 * 2^{\wedge} n\} = (\bigcup bs \in \text{bits } k. mv \text{ } bs \text{ } \text{'nlists } k$
 $\{0..<2^{\wedge} n\})$
by (*auto simp: mv-in-nlists in-nlists2D*)

6.3 Points in a tree

fun *cube* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list set}$ **where**
cube $k \text{ } n = \text{nlists } k \{0..<2^{\wedge} n\}$

fun *points* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{kdtb} \Rightarrow \text{nat list set}$ **where**
points $k \text{ } n \text{ } (Box \text{ } b) = (\text{if } b \text{ then } \text{cube } k \text{ } n \text{ else } \{\}) \mid$
points $k \text{ } (Suc \text{ } n) \text{ } t = (\bigcup bs \in \text{bits } k. mv \text{ } bs \text{ } \text{'points } k \text{ } n \text{ } (\text{subtree } t \text{ } bs))$

lemma *points-Suc*: $\text{points } k \text{ } (Suc \text{ } n) \text{ } t = (\bigcup bs \in \text{bits } k. mv \text{ } bs \text{ } \text{'points } k \text{ } n \text{ } (\text{subtree}$
 $t \text{ } bs))$
by(*cases t*) (*simp-all add: nlists2-simp*)

lemma *points-subset*: $\text{height } t \leq k * n \implies \text{points } k \text{ } n \text{ } t \subseteq \text{nlists } k \{0..<2^{\wedge} n\}$
proof(*induction k n t rule: points.induct*)


```

case (2 k n l r)
have  $\bigwedge bs. bs \in \text{bits } k \implies \text{height } (\text{subtree } (\text{Split } l r) bs) \leq k * n$ 
  unfolding bits-def using 2.prem1 height-subtree2 in-nlists-UNIV by blast
with 2.IH show ?case
by(auto intro: mv-in-nlists dest: subsetD)
qed auto

```

6.4 Compression

Compressing Split:

```

fun SplitC :: 'a kdt  $\Rightarrow$  'a kdt  $\Rightarrow$  'a kdt where
SplitC (Box b1) (Box b2) = (if b1=b2 then Box b1 else Split (Box b1) (Box b2)) |
SplitC t1 t2 = Split t1 t2

```

```

fun compressed :: 'a kdt  $\Rightarrow$  bool where
compressed (Box _) = True |
compressed (Split l r) = (compressed l  $\wedge$  compressed r  $\wedge$   $\neg(\exists b. l = \text{Box } b \wedge r = \text{Box } b)$ )

```

```

lemma compressedI:  $\llbracket \text{compressed } t1; \text{compressed } t2 \rrbracket \implies \text{compressed } (\text{SplitC } t1 t2)$ 
by(induction t1 t2 rule: SplitC.induct) auto

```

```

lemma subtree-SplitC:
   $1 \leq \text{length } bs \implies \text{subtree } (\text{SplitC } l r) bs = \text{subtree } (\text{Split } l r) bs$ 
by(induction l r rule: SplitC.induct)
  (simp-all add: subtree-Split-Box Suc-le-eq)

```

6.5 Union

```

fun union :: kdtb  $\Rightarrow$  kdtb  $\Rightarrow$  kdtb where
union (Box b) t = (if b then Box True else t) |
union t (Box b) = (if b then Box True else t) |
union (Split l1 r1) (Split l2 r2) = SplitC (union l1 l2) (union r1 r2)

```

```

lemma union-Box2: union t (Box b) = (if b then Box True else t)
by(cases t) auto

```

```

lemma in-mv-image:  $\llbracket ps \in \text{nlists } k \{0..<2*2^n\}; Ps \subseteq \text{nlists } k \{0..<2^n\}; bs \in \text{bits } k \rrbracket \implies$ 
   $ps \in \text{mv } bs \text{ ' } Ps \iff \text{map } (\lambda x. x \text{ div } 2) ps \in Ps \wedge (bs = \text{map even } ps)$ 
by (auto simp: map-mv1 map-mv2 mv-map-map bits-def intro!: image-eqI)

```

```

lemma subtree-union: subtree (union t1 t2) bs = union (subtree t1 bs) (subtree t2 bs)
proof(induction t1 t2 arbitrary: bs rule: union.induct)
  case 2 thus ?case by(auto simp: union-Box2)
next
  case 3 thus ?case by(cases bs) (auto simp: subtree-SplitC)

```

qed *auto*

lemma *points-union*:

$\llbracket \max (\text{height } t1) (\text{height } t2) \leq k * n \rrbracket \implies$
 $\text{points } k \ n (\text{union } t1 \ t2) = \text{points } k \ n \ t1 \cup \text{points } k \ n \ t2$

proof(*induction n arbitrary: t1 t2*)

case 0 thus *?case by*(*clarsimp simp add: height-0-iff*)

next

case (*Suc n*)

have $\text{height } t1 \leq k * \text{Suc } n$ $\text{height } t2 \leq k * \text{Suc } n$

using *Suc.prem*s **by** *auto*

from *height-subtree2[OF this(1)] height-subtree2[OF this(2)] show* *?case*

by(*auto simp: Suc.IH subtree-union points-Suc bits-def*)

qed

lemma *compressed-union*: $\text{compressed } t1 \implies \text{compressed } t2 \implies \text{compressed}(\text{union } t1 \ t2)$

by(*induction t1 t2 rule: union.induct*) (*simp-all add: compressedI*)

6.6 Extracting a point from a tree

lemma *size-subtree*: $bs \neq [] \implies (\forall b. t \neq \text{Box } b) \implies \text{size}(\text{subtree } t \ bs) < \text{size } t$

by (*induction t bs rule: subtree.induct*) *force+*

For termination of *get*:

corollary *size-subtree-Split*[*termination-simp*]:

$bs \neq [] \implies \text{size}(\text{subtree}(\text{Split } l \ r) \ bs) < \text{Suc}(\text{size } l + \text{size } r)$

using *size-subtree* **by** *fastforce*

fun *get* :: 'a kdt \Rightarrow nat list \Rightarrow 'a **where**

get (*Box b*) = *b* |

get *t ps* = (*if ps*=[] *then undefined else get* (*subtree t* (*map even ps*)) (*map* ($\lambda i. i \text{ div } 2$) *ps*))

lemma *points-get*: $\llbracket \text{height } t \leq k * n; ps \in \text{nlists } k \ \{0..<2^n\} \rrbracket \implies$

$\text{get } t \ ps = (ps \in \text{points } k \ n \ t)$

proof(*induction n arbitrary: k t ps*)

case 0

then show *?case by*(*clarsimp simp add: height-0-iff*)

next

case (*Suc n*)

show *?case*

proof (*cases t*)

case *Box*

thus *?thesis using Suc.prem*s(2) **by**(*simp*)

next

case (*Split l r*)

obtain *k0* **where** $k = \text{Suc } k0$ **using** *Suc.prem*s(1) *Split*

by(*cases k*) *auto*

hence $ps \neq []$

```

    using Suc.prem(2) by (auto simp: in-nlists-Suc-iff)
  then show ?thesis using Suc.prem Split Suc.IH[OF height-subtree2[OF Suc.prem(1)]]
in-nlists2D
  by(simp add: height-subtree2 in-mv-image points-subset bits-def)
qed
qed
end

```

7 K-dimensional Region Trees - Nested Trees

```

theory KD-Region-Nested
imports HOL-Library.NList
begin

```

```

lemma nlists-Suc: nlists (Suc n) A = (⋃ a∈A. (#) a ‘ nlists n A)
  by(auto simp: set-eq-iff image-iff in-nlists-Suc-iff)
lemma nlists-singleton: nlists n {a} = {replicate n a}
  unfolding nlists-def by(auto simp: replicate-length-same dest!: subset-singletonD)

```

```

fun cube :: nat ⇒ nat ⇒ nat list set where
  cube k n = nlists k {0.. $2^n$ }

```

```

datatype 'a tree1 = Lf 'a | Br 'a tree1 'a tree1
datatype 'a kdt = Cube 'a | Dims 'a kdt tree1

```

```

datatype-compat tree1
datatype-compat kdt

```

```

type-synonym kdtb = bool kdt

```

```

lemma set-tree1-finite-ne: finite (set-tree1 t) ∧ set-tree1 t ≠ {}
  by(induction t) auto

```

```

lemma kdt-tree1-term[termination-simp]: x ∈ set-tree1 t ⇒ size-kdt f x < Suc
(size-tree1 (size-kdt f) t)
  by(induction t)(auto)

```

```

fun h-tree1 :: 'a tree1 ⇒ nat where
  h-tree1 (Lf _) = 0 |
  h-tree1 (Br l r) = max (h-tree1 l) (h-tree1 r) + 1

```

```

function (sequential) h-kdt :: 'a kdt ⇒ nat where
  h-kdt (Cube _) = 0 |
  h-kdt (Dims t) = Max (h-kdt ‘ (set-tree1 t)) + 1
  by pat-completeness auto
termination

```

by(*relation measure (size-kdt (λ-. 1))*)
(auto simp add: wf-lex-prod kdt-tree1-term)

function (*sequential*) *inv-kdt* :: *nat* ⇒ *'a kdt* ⇒ *bool* **where**
inv-kdt k (Cube b) = True |
inv-kdt k (Dims t) = (h-tree1 t ≤ k ∧ (∀ kt ∈ set-tree1 t. inv-kdt k kt))
by *pat-completeness auto*

termination
by(*relation {} <*lex*> measure (size-kdt (λ-. 1))*)
(auto simp add: wf-lex-prod kdt-tree1-term)

definition *bits* :: *nat* ⇒ *bool list set* **where**
bits n = nlists n UNIV

lemma *bits-0*[*code*]: *bits 0 = {[]}*
by (*auto simp: bits-def*)

lemma *bits-Suc*[*code*]: *bits (Suc n) = (let B = bits n in (#) True ‘ B ∪ (#) False ‘ B)*
unfolding *bits-def nlists-Suc UN-bool-eq* **by** *metis*

fun *leaf* :: *'a tree1* ⇒ *bool list* ⇒ *'a* **where**
leaf (Lf x) - = x |
leaf (Br l r) (b#bs) = leaf (if b then r else l) bs |
leaf (Br l r) [] = leaf l []

definition *mv* :: *bool list* ⇒ *nat list* ⇒ *nat list* **where**
*mv = map2 (λb x. 2*x + (if b then 0 else 1))*

fun *points* :: *nat* ⇒ *nat* ⇒ *kdtb* ⇒ *nat list set* **where**
points k n (Cube b) = (if b then cube k n else {}) |
points k (Suc n) (Dims t) = (∪ bs ∈ bits k. mv bs ‘ points k n (leaf t bs))

lemma *bits-nonempty*: *bits n ≠ {}*
by(*auto simp: bits-def Ex-list-of-length*)

lemma *finite-bits*: *finite (bits n)*
by (*metis List.finite-set List.set-insert UNIV-bool bits-def empty-set nlists-set*)

lemma *mv-in-nlists*:
 $\llbracket p \in \text{nlists } k \{0..<2 \wedge n\}; bs \in \text{bits } k \rrbracket \implies mv \text{ } bs \text{ } p \in \text{nlists } k \{0..<2 * 2 \wedge n\}$
unfolding *mv-def nlists-def bits-def*
by (*fastforce dest: set-zip-rightD*)

lemma *leaf-append*: *length bs ≥ h-tree1 t* ⇒ *leaf t (bs@bs') = leaf t bs*
by (*induction t bs arbitrary: bs' rule: leaf.induct*) *auto*

lemma *leaf-take*: *length bs ≥ h-tree1 t* ⇒ *leaf t (bs) = leaf t (take (h-tree1 t) bs)*
by (*metis append-take-drop-id leaf-append length-take min.absorb2 order-refl*)

lemma *Union-bits-le*:
 $h\text{-tree1 } t \leq n \implies (\bigcup bs \in \text{bits } n. \{\text{leaf } t \text{ } bs\}) = (\bigcup bs \in \text{bits } (h\text{-tree1 } t). \{\text{leaf } t \text{ } bs\})$
unfolding *bits-def nlists-def*
apply *rule*
using *leaf-take apply (force)*
by *auto (metis Ex-list-of-length order.refl le-add-diff-inverse leaf-append length-append)*

lemma *set-tree1-leafs*:
 $set\text{-tree1 } t = (\bigcup bs \in \text{bits } (h\text{-tree1 } t). \{\text{leaf } t \text{ } bs\})$
proof(*induction t*)
case (*Lf x*)
then **show** *?case by (simp add: bits-nonempty)*
next
case (*Br t1 t2*)
then **show** *?case using Union-bits-le[of t1 h-tree1 t2] Union-bits-le[of t2 h-tree1 t1]*
by (*auto simp add: Let-def bits-Suc max-def*)
qed

lemma *points-subset*: $inv\text{-kdt } k \ t \implies h\text{-kdt } t \leq n \implies \text{points } k \ n \ t \subseteq \text{nlists } k \ \{0..<2^n\}$
proof(*induction k n t rule: points.induct*)
case (*2 k n t*)
have $mv \ bs \ ps \in \text{nlists } k \ \{0..<2 * 2^n\}$ **if** $*$: $bs \in \text{bits } k \ ps \in \text{points } k \ n \ (\text{leaf } t \text{ } bs)$ **for** $bs \ ps$
proof –
have $inv\text{-kdt } k \ (\text{leaf } t \text{ } bs)$ **using** $*(1) \ 2.\text{prems}(1)$
by(*auto simp: set-tree1-leafs*)
(metis bits-def leaf-take length-take min.absorb2 nlistsE-length nlistsI subset-UNIV)
moreover **have** $h\text{-kdt } (\text{leaf } t \text{ } bs) \leq n$ **using** $*(1) \ 2.\text{prems}$
by(*auto simp add: set-tree1-leafs bits-nonempty finite-bits*)
(metis bits-def leaf-take length-take min.absorb2 nlistsE-length nlistsI subset-UNIV)
ultimately **show** *?thesis using * 2.IH[of bs] mv-in-nlists by(auto)*
qed
thus *?case by(auto)*
qed *auto*

fun *comb1* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \text{ tree1} \Rightarrow 'a \text{ tree1} \Rightarrow 'a \text{ tree1}$ **where**
 $comb1 \ f \ (Lf \ x1) \ (Lf \ x2) = Lf \ (f \ x1 \ x2) \ |$
 $comb1 \ f \ (Br \ l1 \ r1) \ (Br \ l2 \ r2) = Br \ (comb1 \ f \ l1 \ l2) \ (comb1 \ f \ r1 \ r2) \ |$
 $comb1 \ f \ (Br \ l1 \ r1) \ (Lf \ x) = Br \ (comb1 \ f \ l1 \ (Lf \ x)) \ (comb1 \ f \ r1 \ (Lf \ x)) \ |$
 $comb1 \ f \ (Lf \ x) \ (Br \ l2 \ r2) = Br \ (comb1 \ f \ (Lf \ x) \ l2) \ (comb1 \ f \ (Lf \ x) \ r2)$

The last two equations cover cases that do not arise but are needed to prove that *comb1* only applies *f* to elements of the two trees, which implies this congruence lemma:

lemma *comb1-cong*[*fundef-cong*]:
 $\llbracket s1 = t1; s2 = t2; \bigwedge x y. x \in \text{set-tree1 } t1 \implies y \in \text{set-tree1 } t2 \implies f x y = g x y \rrbracket \implies \text{comb1 } f s1 s2 = \text{comb1 } g t1 t2$
apply(*induction* *f* *t1* *t2* *arbitrary: s1 s2* *rule: comb1.induct*)
apply *auto*
done

This congruence lemma in turn implies that *union* terminates because the recursive calls of *union* via *comb1* only involve elements from the two trees, which are smaller.

function (*sequential*) *union* :: *kdtb* \Rightarrow *kdtb* \Rightarrow *kdtb* **where**
union (*Cube* *b*) *t* = (*if* *b* *then* *Cube* *True* *else* *t*) |
union *t* (*Cube* *b*) = (*if* *b* *then* *Cube* *True* *else* *t*) |
union (*Dims* *t1*) (*Dims* *t2*) = *Dims* (*comb1* *union* *t1* *t2*)
by *pat-completeness* *auto*
termination
by(*relation* *measure* (*size-kdt* ($\lambda-. 1$)) *<*lex*>* { })
(*auto simp* *add: wf-lex-prod kdt-tree1-term*)

lemma *leaf-comb1*:
 $\llbracket \text{length } bs \geq \max (\text{h-tree1 } t1) (\text{h-tree1 } t2) \rrbracket \implies$
 $\text{leaf } (\text{comb1 } f t1 t2) bs = f (\text{leaf } t1 bs) (\text{leaf } t2 bs)$
apply(*induction* *f* *t1* *t2* *arbitrary: bs* *rule: comb1.induct*)
apply (*auto simp: Suc-le-length-iff split: if-splits*)
done

lemma *leaf-in-set-tree1*: $\llbracket \text{length } bs \geq \text{h-tree1 } t \rrbracket \implies \text{leaf } t bs \in \text{set-tree1 } t$
apply(*auto simp* *add: set-tree1-leafs bits-def intro: nlistsI*)
by (*metis* *leaf-take length-take min.absorb2 nlistsI subset-UNIV*)

lemma *leaf-in-set-tree2*: $\llbracket x \in \text{nlists } k \text{ UNIV}; \text{h-tree1 } t1 \leq k \rrbracket \implies \text{leaf } t1 x \in \text{set-tree1 } t1$
by (*metis* *leaf-in-set-tree1 leaf-take length-take min.absorb2 nlistsE-length*)

lemma *points-union*:
 $\llbracket \text{inv-kdt } k t1; \text{inv-kdt } k t2; n \geq \max (\text{h-kdt } t1) (\text{h-kdt } t2) \rrbracket \implies$
 $\text{points } k n (\text{union } t1 t2) = \text{points } k n t1 \cup \text{points } k n t2$
proof(*induction* *t1* *t2* *arbitrary: n* *rule: union.induct*)
case 1 **thus** *?case* **using** *Un-absorb2[OF points-subset]* **by** *simp*
next
case 2 **thus** *?case* **using** *Un-absorb1[OF points-subset]* **by** *simp*
next
case ($\exists t1 t2$)
from \exists .*prems* **obtain** *m* **where** $n = \text{Suc } m$ **by** (*auto* *dest: Suc-le-D*)
with \exists **show** *?case*
by (*simp* *add: leaf-comb1 bits-def leaf-in-set-tree2 set-tree1-finite-ne image-Un UN-Un-distrib*)
qed

lemma *size-leaf*[*termination-simp*]: $\text{size} (\text{leaf } t (\text{map } f \text{ ps})) < \text{Suc} (\text{size-tree1 } \text{size } t)$
apply(*induction t map f ps arbitrary: ps rule: leaf.induct*)
apply *simp*
apply *fastforce*
apply *fastforce*
done

fun *get* :: 'a kdt \Rightarrow nat list \Rightarrow 'a **where**
get (*Cube b*) - = *b* |
get (*Dims t*) *ps* = *get* (*leaf t (map even ps)*) (*map* ($\lambda x. x \text{ div } 2$) *ps*)

lemma *map-zip1*: $\llbracket \text{length } xs = \text{length } ys; \forall p \in \text{set}(\text{zip } xs \text{ ys}). f \text{ p} = \text{fst } p \rrbracket \Longrightarrow$
 $\text{map } f (\text{zip } xs \text{ ys}) = xs$
by (*metis* (*no-types*, *lifting*) *map-eq-conv map-fst-zip*)

lemma *map-mv1*: $\llbracket \text{length } ps = \text{length } bs \rrbracket \Longrightarrow \text{map even} (\text{mv } bs \text{ ps}) = bs$
unfolding *nlists-def mv-def* **by**(*auto intro!: map-zip1 split: if-splits*)

lemma *map-zip2*: $\llbracket \text{length } xs = \text{length } ys; \forall p \in \text{set}(\text{zip } xs \text{ ys}). f \text{ p} = \text{snd } p \rrbracket \Longrightarrow$
 $\text{map } f (\text{zip } xs \text{ ys}) = ys$
by (*metis* (*no-types*, *lifting*) *map-eq-conv map-snd-zip*)

lemma *map-mv2*: $\llbracket \text{length } ps = \text{length } bs \rrbracket \Longrightarrow \text{map} (\lambda x. x \text{ div } 2) (\text{mv } bs \text{ ps}) =$
 ps
unfolding *nlists-def mv-def* **by**(*auto intro!: map-zip2*)

lemma *mv-map-map*: $\text{mv} (\text{map even } ps) (\text{map} (\lambda x. x \text{ div } 2) \text{ ps}) = ps$
unfolding *nlists-def mv-def*
by(*auto simp add: map-eq-conv*[**where** *xs=ps and g=id,simplified*] *map2-map-map*)

lemma *in-mv-image*: $\llbracket ps \in \text{nlists } k \{0..<2*2^{\wedge}n\}; Ps \subseteq \text{nlists } k \{0..<2^{\wedge}n\}; bs \in$
 $\text{bits } k \rrbracket \Longrightarrow$
 $ps \in \text{mv } bs \text{ ' } Ps \iff \text{map} (\lambda x. x \text{ div } 2) \text{ ps} \in Ps \wedge (bs = \text{map even } ps)$
by (*auto simp: map-mv1 map-mv2 mv-map-map bits-def intro!: image-eqI*)

lemma *get-points*: $\llbracket \text{inv-kdt } k \text{ t}; h\text{-kdt } t \leq n; ps \in \text{nlists } k \{0..<2^{\wedge}n\} \rrbracket \Longrightarrow$
 $\text{get } t \text{ ps} = (ps \in \text{points } k \text{ n } t)$
proof(*induction t ps arbitrary: n rule: get.induct*)
case (*2 t ps*)
obtain *m* **where** [*simp*]: $n = \text{Suc } m$ **using** $\langle h\text{-kdt } (\text{Dims } t) \leq n \rangle$ **by** (*auto dest: Suc-le-D*)
have $\forall bs. \text{length } bs = k \longrightarrow \text{inv-kdt } k (\text{leaf } t \text{ bs}) \wedge h\text{-kdt } (\text{leaf } t \text{ bs}) \leq m$
using *2.prem*s **by** (*auto simp add: leaf-in-set-tree1 set-tree1-finite-ne*)
moreover **have** $\text{map} (\lambda x. x \text{ div } 2) \text{ ps} \in \text{nlists } k \{0..<2^{\wedge}m\}$
using *2.prem*s(3) **by**(*fastforce intro!: nlistsI dest: nlistsE-set*)
ultimately show ?*case* **using** *2.prem*s *2.IH*[*of m*] *points-subset*[*of k - m*]
by(*auto simp add: in-mv-image bits-def intro: nlistsI*)
qed *auto*

```

fun modify :: ('a ⇒ 'a) ⇒ bool list ⇒ 'a tree1 ⇒ 'a tree1 where
  modify f [] (Lf x) = Lf (f x) |
  modify f (b#bs) (Lf x) = (if b then Br (Lf x) (modify f bs (Lf x)) else Br (modify
  f bs (Lf x)) (Lf x)) |
  modify f (b#bs) (Br l r) = (if b then Br l (modify f bs r) else Br (modify
  f bs l) r)

```

```

fun put :: 'a ⇒ nat ⇒ nat list ⇒ 'a kdt ⇒ 'a kdt where
  put b' 0 ps (Cube -) = Cube b' |
  put b' (Suc n) ps t =
    Dims (modify (put b' n (map (λi. i div 2) ps)) (map even ps)
    (case t of Cube b ⇒ Lf (Cube b) | Dims t ⇒ t))

```

```

lemma leaf-modify: [ h-tree1 t ≤ length bs; length bs' = length bs ] ⇒
  leaf (modify f bs t) bs' = (if bs' = bs then f(leaf t bs) else leaf t bs')
apply(induction f bs t arbitrary: bs' rule: modify.induct)
apply(auto simp: length-Suc-conv split: if-splits)
done

```

```

lemma in-nlists2D: xs ∈ nlists k {0..<2 * 2 ^ n} ⇒ ∃ bs ∈ nlists k UNIV. xs ∈
  mv bs ' nlists k {0..<2 ^ n}
unfolding nlists-def
apply(rule bexI[where x = map even xs])
apply(auto simp: image-def)[1]
apply(rule exI[where x = map (λi. i div 2) xs])
apply(auto simp add: mv-map-map)
done

```

```

lemma nlists2-simp: nlists k {0..<2 * 2 ^ n} = (∪ bs ∈ nlists k UNIV. mv bs '
  nlists k {0..<2 ^ n})
by (auto simp: mv-in-nlists bits-def in-nlists2D)

```

```

lemma mv-diff:
  [ length qs = length bs; ∀ as ∈ A. length as = length bs ] ⇒ mv bs ' (A - {qs})
  = mv bs ' A - {mv bs qs}
by (auto) (metis map-mv2)

```

```

lemma put-points: [ inv-kdt k t; h-kdt t ≤ n; ps ∈ nlists k {0..<2 ^ n} ] ⇒
  points k n (put b n ps t) = (if b then points k n t ∪ {ps} else points k n t - {ps})
proof(induction b n ps t rule: put.induct)
  case 1 thus ?case by (simp add: nlists-singleton)
next
  case (2 b' n ps t)
  have *: ∀ x bs. t = Dims x ⇒ length bs = length ps ⇒ inv-kdt k (leaf x bs)
  using 2.premis(1,3) leaf-in-set-tree1 by fastforce
  have **: t = Dims x ⇒ length bs = length ps ⇒ h-kdt (leaf x bs) ≤ n for x bs
  using leaf-in-set-tree1[of x] 2.premis set-tree1-finite-ne[of x] by auto

```


have ***: $\llbracket t = \text{Dims } x; \text{length } bs = \text{length } ps \rrbracket \implies$
 $(\forall qs \in \text{points } (\text{length } ps) \ n \ (\text{leaf } x \ bs). \ \text{length } qs = \text{length } ps) = \text{True}$ **for** $x \ bs$
using $2.\text{prems}(3)$ **by** $(\text{metis} \ * \ ** \ \text{nlistsE-length points-subset subset-iff})$

have $\text{Union-diff-aux}: a \in A \implies (\bigcup x \in A. F \ x) = F \ a \cup (\bigcup x \in A - \{a\}. F \ x)$
for $a \ A \ F$
by blast

have $\text{notin-aux}: \forall x \in \text{nlists } (\text{length } ps) \ \text{UNIV} - \{\text{map even } ps\}. \forall qs \in A \ x. \ \text{length } qs = \text{length } ps \implies$
 $ps \notin (\bigcup x \in \text{nlists } (\text{length } ps) \ \text{UNIV} - \{\text{map even } ps\}. \text{mv } x \ ' \ A \ x)$ **for** A
by $(\text{smt } (\text{verit } \text{DiffE UN-E image-iff insert-iff map-mv1 nlistsE-length}))$
have $\text{set1}: \bigwedge x \ y. \ \{x. \ x \neq y\} = \text{UNIV} - \{y\}$ **by** blast
have $\text{nlists-map}: \bigwedge n \ xs \ f \ A. \ n = \text{size } xs \implies (\text{map } f \ xs \in \text{nlists } n \ A) = (f \ ' \ \text{set } xs \subseteq A)$ **by** simp

have $(\lambda i. \ i \ \text{div } 2) \ ' \ \text{set } ps \subseteq \{0..<2 \wedge n\}$ **using** $\text{nlistsE-set}[OF \ 2.\text{prems}(3)]$ **by** auto

moreover **have** $\forall x. \ t = \text{Dims } x \longrightarrow \text{inv-kdt } k \ (\text{Dims } x)$
using $2.\text{prems}(1)$ **by** blast

moreover **have** $t = \text{Dims } x \implies \text{length } bs = \text{length } ps \implies \text{points } (\text{length } ps) \ n$
 $(\text{leaf } x \ bs) \subseteq \text{nlists } (\text{length } ps) \ \{0..<2 \wedge n\}$ **for** $x \ bs$
using $2.\text{prems}(3)$ **by** $(\text{metis} \ * \ ** \ \text{nlistsE-length points-subset})$

moreover **have** $\text{length } ps = k$ **using** $2.\text{prems}(3)$ **by** simp

moreover **from** $2 \ * \ ** \ \text{calculation}$ **show** $?case$
by $(\text{clarsimp } \text{simp}: \text{leaf-modify}[of \ \text{map even } ps] \ \text{mv-map-map } \text{nlists-map bits-def}$
 $\text{nlistsE-length}[of \ \text{--}: \text{bool list } k \ \text{UNIV}] \ \text{nlists2-simp } \text{Union-diff-aux}[of \ \text{map even } ps]$
 $\text{mv-diff} \ *** \ \text{Diff-insert0}[OF \ \text{notin-aux}]$
 $\text{insert-absorb } \text{Diff-insert-absorb } \text{Int-absorb1 } \text{set1 } \text{Diff-Int-distrib } \text{Un-Diff}$
 $\text{split}: \text{kdt.split})$

qed simp

end

References

- [1] S. Aluru. Quadrees and octrees. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2nd edition, 2017.
- [2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509517, 1975.
- [3] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209226, 1977.

- [4] M. Rau. Multidimensional binary search trees. *Archive of Formal Proofs*, May 2019. https://isa-afp.org/entries/KD_Tree.html, Formal proof development.
- [5] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.
- [6] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
- [7] D. S. Wise. Representing matrices as quadtrees for parallel processors: extended abstract. *SIGSAM Bull.*, 18(3):24–25, 1984.
- [8] D. S. Wise. Representing matrices as quadtrees for parallel processors. *Inf. Process. Lett.*, 20(4):195–199, 1985.
- [9] D. S. Wise. Parallel decomposition of matrix inversion using quadtrees. In *International Conference on Parallel Processing, ICPP'86*, pages 92–99. IEEE Computer Society Press, 1986.
- [10] D. S. Wise. Matrix algebra and applicative programming. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274, pages 134–153, 1987.
- [11] D. S. Wise. Matrix algorithms using quadtrees (invited talk). In G. Hains and L. M. R. Mullin, editors, *ATABLE-92, Intl. Workshop on Arrays, Functional Languages and Parallel Systems*, 1992.