Region Quadtrees

Tobias Nipkow
Technical University of Munich

April 18, 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-dimenstional 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

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 ⇒ nat* **where**
height (L -) = 0 |
height (Q t0 t1 t2 t3) =
Max {height t0, height t1, height t2, height t3} + 1

instance ⟨*proof*⟩

end

end

3 Quad Trees

theory *Quad-Tree*
imports *Quad-Base*
begin

lemma *diff-shunt*: *({} = x − y) ⟷ (x ≤ y)*
⟨*proof*⟩

lemma *mod-minus*: *⟦ i < 2∗m; ¬ i < m ⟧ ⟹ i mod m = i − (m::nat)*
⟨*proof*⟩

definition *select* :: *bool ⇒ bool ⇒ 'a ⇒ 'a ⇒ 'a ⇒ '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 ≡ q (f i j) (f i (j+d)) (f (i+d) j) (f (i+d) (j+d))

3.1 Compression

fun *compressed* :: *'a qtree ⇒ bool* **where**
compressed (L -) = True |

compressed (Q t0 t1 t2 t3) = ((compressed t0 ∧ compressed t1 ∧ compressed t2 ∧ compressed t3)
∧ ¬ (∃ x. t0 = L x ∧ t1 = t0 ∧ t2 = t0 ∧ t3 = t0))

fun *Qc* :: *'a qtree ⇒ 'a qtree ⇒ 'a qtree ⇒ 'a qtree ⇒ 'a qtree* **where**
Qc (L x0) (L x1) (L x2) (L x3) =
(if x0=x1 ∧ x1=x2 ∧ x2=x3 then L x0 else Q (L x0) (L x1) (L x2) (L x3)) |
Qc t0 t1 t2 t3 = Q t0 t1 t2 t3

Compressing version of *Q*:

lemma *compressed-Qc*: ⟦*compressed t0*; *compressed t1*; *compressed t2*; *compressed t3* ⟧ ⟹
compressed (Qc t0 t1 t2 t3)
⟨*proof*⟩

lemma *compressedQD*: *compressed (Q t1 t2 t3 t4)*
⟹ *compressed t1 ∧ compressed t2 ∧ compressed t3 ∧ compressed t4*
⟨*proof*⟩

lemma *height-Qc-Q*: ⟦ *height s0 ≤ n*; *height s1 ≤ n*; *height s2 ≤ n*; *height s3 ≤ n* ⟧
⟹ *height (Qc s0 s1 s2 s3) ≤ Suc n*
⟨*proof*⟩

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

fun *modify* :: *('a qtree ⇒ 'a qtree) ⇒ bool ⇒ bool ⇒ 'a qtree ∗'a qtree ∗'a qtree ∗'a qtree ⇒ 'a qtree* **where**
modify f x y (t0, t1, t2, t3) =
(if x then
if y then Qc (f t0) t1 t2 t3 else Qc t0 (f t1) t2 t3
else
if y then Qc t0 t1 (f t2) t3 else Qc t0 t1 t2 (f t3))

3.2 Abstraction function

fun *get* :: *nat ⇒ 'a qtree ⇒ nat ⇒ nat ⇒ 'a* **where**
get n (L b) - - = b |
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-Qc*:
height(Q t0 t1 t2 t3) ≤ n ⟹ get n (Qc t0 t1 t2 t3) i j = get n (Q t0 t1 t2 t3) i j
⟨*proof*⟩

3.3 Boolean Quadtrees

type-synonym *qtb = bool qtree*

3.3.1 Abstraction of boolean quadtrees to sets of points

Superceded 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*)
⟨*proof*⟩

lemma *in-shift-image*: (*x*,*y*) ∈ *shift di dj* ‘ *M* ⟷ *di* ≤ *x* ∧ *dj* ≤ *y* ∧ (*x*−*di*,*y*−*dj*) ∈ *M*
⟨*proof*⟩

lemma *inj-shift*: *inj* (*shift i j*)
⟨*proof*⟩

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*′ = {}
⟨*proof*⟩

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

The layout of the 4 subquadrants *Q t0 t1 t2 t3* / *Qsq A0 A1 A2 A3*: *1 3 0 2* 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*)
⟨*proof*⟩

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* ≤ *n* ⟹ *points n t* ⊆ *sq n*
⟨*proof*⟩

lemma *point-Suc-Qc*[*simp*]: *points* (*Suc n*) (*Qc t0 t1 t2 t3*) = *points* (*Suc n*) (*Q t0 t1 t2 t3*)
⟨*proof*⟩

lemma *get-points*: ⟦ *height t* ≤ *n*; (*i*,*j*) ∈ *sq n* ⟧ ⟹ *get n t i j* = ((*i*,*j*) ∈ *points n t*)
⟨*proof*⟩

3.3.2 Union, Intersection Difference and Complement

fun *union* :: *qtb* ⇒ *qtb* ⇒ *qtb* **where**
union (*L b*) *t* = (*if b then L True else t*) |
union t (*L b*) = (*if b then L True else t*) |
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* ⇒ *qtb* ⇒ *qtb* **where**
inter (*L b*) *t* = (*if b then t else L False*) |
inter t (*L b*) = (*if b then t else L False*) |
inter (*Q s1 s2 s3 s4*) (*Q t1 t2 t3 t4*) = *Qc* (*inter s1 t1*) (*inter s2 t2*) (*inter s3 t3*) (*inter s4 t4*)

fun *negate* :: *qtb* ⇒ *qtb* **where**
negate (*L b*) = *L*(¬*b*) |
negate (*Q t1 t2 t3 t4*) = *Q* (*negate t1*) (*negate t2*) (*negate t3*) (*negate t4*)

fun *diff* :: *qtb* ⇒ *qtb* ⇒ *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 ∪ *Qsq n B0 B1 B2 B3* = *Qsq n* (*A0* ∪ *B0*) (*A1* ∪ *B1*) (*A2* ∪ *B2*) (*A3* ∪ *B3*)
⟨*proof*⟩

lemma *points-union*:
max (*height t1*) (*height t2*) ≤ *n* ⟹ *points n* (*union t1 t2*) = *points n t1* ∪ *points n t2*
⟨*proof*⟩

lemma *height-union*: *height* (*union t1 t2*) ≤ *max* (*height t1*) (*height t2*)
⟨*proof*⟩

lemma *height-union2*: ⟦ *height t1* ≤ *n*; *height t2* ≤ *n* ⟧ ⟹ *height* (*union t1 t2*) ≤ *n*
⟨*proof*⟩

lemma *get-union*:
max (*height t1*) (*height t2*) ≤ *n* ⟹ *get n* (*union t1 t2*) *i j* = (*get n t1 i j* ∨ *get*

$n\ t2\ i\ j$)
⟨*proof*⟩

lemma *compressed-union*: *compressed t1* ⟹ *compressed t2* ⟹ *compressed*(*union t1 t2*)
⟨*proof*⟩

lemma *Qsq-inter*:
⟦ $A0 \subseteq sq\ n$; $A1 \subseteq sq\ n$; $A2 \subseteq sq\ n$; $A3 \subseteq sq\ n$;
$B0 \subseteq sq\ n$; $B1 \subseteq sq\ n$; $B2 \subseteq sq\ n$; $B3 \subseteq sq\ n$ ⟧
⟹ $Qsq\ n\ A0\ A1\ A2\ A3 \cap Qsq\ n\ B0\ B1\ B2\ B3 = Qsq\ n\ (A0 \cap B0)\ (A1 \cap B1)\ (A2 \cap B2)\ (A3 \cap B3)$
⟨*proof*⟩

lemma *points-inter*: $n \geq max\ (height\ t1)\ (height\ t2)$ ⟹
points n (*inter t1 t2*) = *points n t1* ∩ *points n t2*
⟨*proof*⟩

lemma *height-inter*: $height\ (inter\ t1\ t2) \leq max\ (height\ t1)\ (height\ t2)$
⟨*proof*⟩

lemma *height-inter2*: ⟦ $height\ t1 \leq n$; $height\ t2 \leq n$ ⟧ ⟹ $height\ (inter\ t1\ t2) \leq n$
⟨*proof*⟩

lemma *get-inter*:
⟦ $height\ t1 \leq n$; $height\ t2 \leq n$ ⟧ ⟹ $get\ n\ (inter\ t1\ t2)\ i\ j = (get\ n\ t1\ i\ j \wedge get\ n\ t2\ i\ j)$
⟨*proof*⟩

lemma *compressed-inter*: *compressed t1* ⟹ *compressed t2* ⟹ *compressed*(*inter t1 t2*)
⟨*proof*⟩

lemma *Qsq-diff*: ⟦ $B0 \subseteq sq\ n$; $B1 \subseteq sq\ n$; $B2 \subseteq sq\ n$; $B3 \subseteq sq\ n$; $A0 \subseteq sq\ n$; $A1 \subseteq sq\ n$; $A2 \subseteq sq\ n$; $A3 \subseteq sq\ n$ ⟧ ⟹
$Qsq\ n\ B0\ B1\ B2\ B3 - Qsq\ n\ A0\ A1\ A2\ A3 = Qsq\ n\ (B0 - A0)\ (B1 - A1)\ (B2 - A2)\ (B3 - A3)$
⟨*proof*⟩

lemma *points-negate*: $n \geq height\ t \Longrightarrow points\ n\ (negate\ t) = sq\ n - points\ n\ t$
⟨*proof*⟩

lemma *negate-eq-L-iff*: *compressed t* ⟹ *negate t* = *L x* ⟷ *t* = *L*(¬*x*)
⟨*proof*⟩

lemma *compressed-negate*: *compressed t* ⟹ *compressed*(*negate t*)
⟨*proof*⟩

lemma *points-diff*: $n \geq max\ (height\ t1)\ (height\ t2) \Longrightarrow$
$points\ n\ (diff\ t1\ t2) = points\ n\ t1 - points\ n\ t2$
⟨*proof*⟩

lemma *compressed-diff*: $compressed\ t1 \Longrightarrow compressed\ t2 \Longrightarrow compressed(diff\ t1\ t2)$
⟨*proof*⟩

3.4 Operation *put*

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

lemma *points-put*: $[\![\ height\ t \leq n;\ (i,j) \in sq\ n\ ]\!] \Longrightarrow$
$points\ n\ (put\ i\ j\ b\ n\ t) = (if\ b\ then\ points\ n\ t \cup \{(i,j)\}\ else\ points\ n\ t - \{(i,j)\})$
⟨*proof*⟩

lemma *height-put*: $height\ t \leq n \Longrightarrow height\ (put\ i\ j\ a\ n\ t) \leq n$
⟨*proof*⟩

lemma *get-put*: $[\![\ height\ t \leq n;\ (i,j) \in sq\ n;\ (i',j') \in sq\ n\ ]\!] \Longrightarrow$
$get\ n\ (put\ i\ j\ a\ n\ t)\ i'\ j' = (if\ i'{=}i \wedge j'{=}j\ then\ a\ else\ get\ n\ t\ i'\ j')$
⟨*proof*⟩

lemma *compressed-put*:
$[\![\ height\ t \leq n;\ compressed\ t\ ]\!] \Longrightarrow compressed\ (put\ i\ j\ a\ n\ t)$
⟨*proof*⟩

3.5 Extract Square

fun $get\text{-}sq :: nat \Rightarrow 'a\ qtree \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ qtree$ **where**
$get\text{-}sq\ n\ (L\ b)\ m\ i\ j = L\ b\ |$
$get\text{-}sq\ n\ t\ 0\ i\ j = L\ (get\ n\ t\ i\ j)\ |$
$get\text{-}sq\ (Suc\ n)\ (Q\ t0\ t1\ t2\ t3)\ (Suc\ m)\ i\ j =$
$(if\ i\ mod\ 2\hat{}n + 2\hat{}(m{+}1) \leq 2\hat{}n \wedge j\ mod\ 2\hat{}n + 2\hat{}(m{+}1) \leq 2\hat{}n$
$then\ get\text{-}sq\ n\ (select\ (i < 2\hat{}n)\ (j < 2\hat{}n)\ t0\ t1\ t2\ t3)\ (m{+}1)\ (i\ mod\ 2\hat{}n)\ (j\ mod\ 2\hat{}n)$
$else\ qf\ Qc\ (get\text{-}sq\ (Suc\ n)\ (Q\ t0\ t1\ t2\ t3)\ m)\ i\ j\ (2\hat{}m))$

lemma *shift-shift*: $shift\ i\ j\ `\ (shift\ i'\ j'\ `\ s) = shift\ (i{+}i')\ (j{+}j')\ `\ s$
⟨*proof*⟩
lemma *shift-shift2*: $shift\ i\ j\ `\ (shift\ i'\ j'\ `\ s) = shift\ (i'{+}i)\ (j'{+}j)\ `\ s$
⟨*proof*⟩

lemma *shift-split*: $shift\ i\ j\ `\ s =$
$shift\ (i - i\ mod\ 2\hat{}n)\ (j - j\ mod\ 2\hat{}n)\ `\ (shift\ (i\ mod\ 2\hat{}n)\ (j\ mod\ 2\hat{}n)\ `\ s)$

⟨*proof*⟩

lemma *plus-pow-aux*: $(i::nat) + 2\hat{}m \leq 2*2\hat{}n \Longrightarrow i < 2 * 2 \hat{} n$
⟨*proof*⟩

lemma *Qsq-lem*: $[\![\ A0 \subseteq sq\ n;\ A1 \subseteq sq\ n;\ A2 \subseteq sq\ n;\ A3 \subseteq sq\ n;$
$i + 2 \hat{} m \leq 2 \hat{} Suc\ n;\ j + 2 \hat{} m \leq 2 \hat{} Suc\ n;$
$i\ mod\ 2 \hat{} n + 2 \hat{} m \leq 2 \hat{} n;\ j\ mod\ 2 \hat{} n + 2 \hat{} m \leq 2 \hat{} n\ ]\!] \Longrightarrow$
$Qsq\ n\ A0\ A1\ A2\ A3 \cap shift\ i\ j\ `\ sq\ m =$
$shift\ (i - i\ mod\ 2\hat{}n)\ (j - j\ mod\ 2\hat{}n)\ `\ select\ (i < 2 \hat{} n)\ (j < 2 \hat{} n)\ A0\ A1\ A2\ A3 \cap shift\ i\ j\ `\ sq\ m$
⟨*proof*⟩

lemma *f-select*: $f\ (select\ x\ y\ a\ b\ c\ d) = select\ x\ y\ (f\ a)\ (f\ b)\ (f\ c)\ (f\ d)$
⟨*proof*⟩

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

lemma *shift-Qsq*: $shift\ i\ j\ `\ Qsq\ n\ A0\ A1\ A2\ A3 =$
$Qsq\ n\ (shift\ i\ j\ `\ A0)\ (shift\ i\ j\ `\ A1)\ (shift\ i\ j\ `\ A2)\ (shift\ i\ j\ `\ A3)$
⟨*proof*⟩

lemma *points-get-sq*:
$[\![\ height\ t \leq n;\ i + 2\hat{}m \leq 2\hat{}n;\ j + 2\hat{}m \leq 2\hat{}n\ ]\!] \Longrightarrow$
$shift\ i\ j\ `\ points\ m\ (get\text{-}sq\ n\ t\ m\ i\ j) = points\ n\ t \cap (shift\ i\ j\ `\ sq\ m)$
⟨*proof*⟩

lemma *get-get-sq*:
$[\![\ height\ t \leq n;\ i + 2\hat{}m \leq 2\hat{}n;\ j + 2\hat{}m \leq 2\hat{}n;\ i' < 2\hat{}m;\ j' < 2\hat{}m\ ]\!] \Longrightarrow$
$get\ m\ (get\text{-}sq\ n\ t\ m\ i\ j)\ i'\ j' = get\ n\ t\ (i+i')\ (j+j')$
⟨*proof*⟩

lemma *compressed-get-sq*:
$[\![\ height\ t \leq n;\ compressed\ t\ ]\!] \Longrightarrow compressed\ (get\text{-}sq\ n\ t\ m\ i\ j)$
⟨*proof*⟩

3.6 From Matrix to Quadtree

3.6.1 Matrix as function

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

fun *qt-of-fun* :: $(nat \Rightarrow nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a\ qtree$ **where**
$qt\text{-}of\text{-}fun\ mx\ (Suc\ n) = qf\ Qc\ (\lambda x\ y.\ qt\text{-}of\text{-}fun\ (shift\text{-}mx\ mx\ x\ y)\ n)\ 0\ 0\ (2\hat{}n)\ |$
$qt\text{-}of\text{-}fun\ mx\ 0 = L(mx\ 0\ 0)$

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

lemma *compressed-qt-of-fun*: *compressed* (*qt-of-fun mx n*)
⟨*proof*⟩

3.6.2 Matrix as list of lists

type-synonym *'a mx* = *'a list list*

definition *sq-mx n mx* = (*length mx* = *2^n* ∧ (∀ *xs* ∈ *set mx*. *length xs* = *2^n*))

lemma *sq-mx-0*: *sq-mx 0 mx* = (∃ *x*. *mx* = [[*x*]])
⟨*proof*⟩

Decompose matrix into submatrices

definition *decomp* **where**
decomp n mx = (*let mx01* = *take* (*2^n*) *mx*; *mx23* = *drop* (*2^n*) *mx*
in (*map* (*take* (*2^n*)) *mx01*, *map* (*drop* (*2^n*)) *mx01*, *map* (*take* (*2^n*)) *mx23*, *map* (*drop* (*2^n*)) *mx23*))

lemma *decomp-sq-mx*: *sq-mx* (*Suc n*) *mx* ⟹ (*mx0*,*mx1*,*mx2*,*mx3*) = *decomp n mx* ⟹
sq-mx n mx0 ∧ *sq-mx n mx1* ∧ *sq-mx n mx2* ∧ *sq-mx n mx3*
⟨*proof*⟩

Quadtree of matrix:

fun *qt-of* :: *nat* ⇒ *'a mx* ⇒ *'a qtree* **where**
qt-of (*Suc n*) *mx* =
(*let* (*mx0*,*mx1*,*mx2*,*mx3*) = *decomp n mx*
in Qc (*qt-of n mx0*) (*qt-of n mx1*) (*qt-of n mx2*) (*qt-of n mx3*)) |
qt-of 0 [[*x*]] = *L x*

lemma *height-qt-of*: *sq-mx n mx* ⟹ *height*(*qt-of n mx*) ≤ *n*
⟨*proof*⟩

lemma *compressed-qt-of*: *sq-mx n mx* ⟹ *compressed*(*qt-of n mx*)
⟨*proof*⟩

lemma *points-qt-of*: *sq-mx n mx* ⟹ *points n* (*qt-of n mx*) = {(*i*,*j*) ∈ *sq n*. *mx* ! *i* ! *j*}
⟨*proof*⟩

lemma *get-qt-of*: ⟦ *sq-mx n mx*; (*i*,*j*) ∈ *sq n* ⟧ ⟹ *get n* (*qt-of n mx*) *i j* = *mx* ! *i* ! *j*
⟨*proof*⟩

3.7 From Quadtree to Matrix

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

fun *mx-of* :: *nat* ⇒ *'a qtree* ⇒ *'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*: ⟦ *sq-mx n mx0*; *sq-mx n mx1*; *sq-mx n mx2*; *sq-mx n mx3*; *i* < *2*2^n*; *j* < *2*2^n* ⟧ ⟹
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*)
⟨*proof*⟩

lemma *sq-mx-mx-of*: *height t* ≤ *n* ⟹ *sq-mx n* (*mx-of n t*)
⟨*proof*⟩

lemma *mx-of-points*: *height t* ≤ *n* ⟹ *points n t* = {(*i*,*j*) ∈ *sq n*. *mx-of n t* ! *i* ! *j*}
⟨*proof*⟩

lemma *mx-of-get*: ⟦ *height t* ≤ *n*; (*i*,*j*) ∈ *sq n* ⟧ ⟹ *mx-of n t* ! *i* ! *j* = *get n t i j*
⟨*proof*⟩

end

4 Block Matrices via Quad Trees

theory *Quad-Matrix*
imports
Complex-Main
Quad-Base
begin

There are two possible representations of marices as quadtrees. In this file we use the standard quadtree with two constructors *L* and *Q*. *L x* represents the *x*-diagonal ma 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 1x1 ma *x*. The advantage is that there are fewer cases in function definitions because one cannot add/multiply *L* and *Q*: the have different dimensions. However, *L 0* is special: it still represents the 0 ma 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* ⇒ *nat* ⇒ *real*

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

definition *mk-sq* :: *nat* ⇒ *ma* ⇒ *ma* **where**
mk-sq n a = (λ*i j*. *if i* < 2^*n* ∧ *j* < 2^*n then a i j else 0*)

abbreviation *sq-ma n* (*a*::*ma*) ≡ (∀ *i j*. 2^*n* ≤ *i* ∨ 2^*n* ≤ *j* ⟶ *a i j* = *0*)

Without *mk-sq* a number of lemmas like *mult-ma-diag-ma-diag-ma* don't hold.

definition *diag-ma* :: *nat* ⇒ *real* ⇒ *ma* **where**
diag-ma n x = *mk-sq n* (λ*i j*. *if i*=*j then x else 0*)

definition *add-ma* :: *ma* ⇒ *ma* ⇒ *ma* **where**
add-ma a b = (λ*i j*. *a i j* + *b i j*)

definition *mult-ma* :: *nat* ⇒ *ma* ⇒ *ma* ⇒ *ma* **where**
mult-ma n a b = (λ*i j*. ∑ *k*=*0*..<2^*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*)
⟨*proof*⟩

lemma *add-ma-diag-ma-0*[*simp*]: *add-ma* (*diag-ma n 0*) *a* = *a*
⟨*proof*⟩

lemma *add-ma-diag-ma-02*[*simp*]: *add-ma a* (*diag-ma n 0*) = *a*
⟨*proof*⟩

lemma *mult-ma-diag-ma-0*[*simp*]: *mult-ma n* (*diag-ma n 0*) *a* = *diag-ma n 0*
⟨*proof*⟩

lemma *mult-ma-diag-ma-02*[*simp*]: *mult-ma n a* (*diag-ma n 0*) = *diag-ma n 0*
⟨*proof*⟩

lemma *mult-ma-diag-ma-diag-ma*[*simp*]: *mult-ma n* (*diag-ma n x*) (*diag-ma n y*) = *diag-ma n* (*x*∗*y*)
⟨*proof*⟩

4.3 Real Quad Trees and Abstraction to Matrices

type-synonym *qtr* = *real qtree*

fun *compressed* :: *qtr* ⇒ *bool* **where**
compressed (*L x*) = *True* |
compressed (*Q* (*L x0*) (*L x1*) (*L x2*) (*L x3*)) = (¬ (*x1*=*0* ∧ *x2*=*0* ∧ *x0*=*x3*)) |

compressed (Q t0 t1 t2 t3) = (compressed t0 ∧ compressed t1 ∧ compressed t2 ∧ compressed t3)

lemma *compressed-Q*:
compressed (Q t1 t2 t3 t4) ⟹ (compressed t1 ∧ compressed t2 ∧ compressed t3 ∧ compressed t4)
⟨*proof*⟩

definition *Qma* :: *nat* ⇒ *ma* ⇒ *ma* ⇒ *ma* ⇒ *ma* ⇒ *ma* **where**
Qma n a b c d =
(λ*i j*. *if i* < 2^*n then if j* < 2^*n then a i j else b i* (*j* − 2^*n*) *else*
if j < 2^*n then c* (*i* − 2^*n*) *j else d* (*i* − 2^*n*) (*j* − 2^*n*))

lemma *add-ma-Qma*:
add-ma (*Qma n a b c d*) (*Qma n a′ b′ c′ d′*) =
Qma n (*add-ma a a′*) (*add-ma b b′*) (*add-ma c c′*) (*add-ma d d′*)
⟨*proof*⟩

lemma *add-ma-diag-ma-Qma*: *add-ma* (*diag-ma* (*Suc n*) *x*) (*Qma n a b c d*) =
Qma n (*add-ma* (*diag-ma n x*) *a*) *b c* (*add-ma* (*diag-ma n x*) *d*)
⟨*proof*⟩

lemma *add-ma-Qma-diag-ma*: *add-ma* (*Qma n a b c d*) (*diag-ma* (*Suc n*) *x*) =
Qma n (*add-ma a* (*diag-ma n x*)) *b c* (*add-ma d* (*diag-ma n x*))
⟨*proof*⟩

lemma *diag-ma-Suc*: *diag-ma* (*Suc n*) *x* = *Qma n* (*diag-ma n x*) (*diag-ma n 0*) (*diag-ma n 0*) (*diag-ma n x*)
⟨*proof*⟩

Abstraction function:

fun *ma* :: *nat* ⇒ *qtr* ⇒ *ma* **where**
ma n (*L x*) = *diag-ma n x* |
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* ⇒ *qtr* ⇒ *qtr* ⇒ *qtr* ⇒ *qtr* **where**
Qc (*L x0*) (*L x1*) (*L x2*) (*L x3*) =
(*if x1*=*0* ∧ *x2*=*0* ∧ *x0*=*x3 then L x0 else Q* (*L x0*) (*L x1*) (*L x2*) (*L x3*)) |
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*)
⟨*proof*⟩

lemma *compressed-Qc*:
compressed (Qc t0 t1 t2 t3) = (compressed t0 ∧ compressed t1 ∧ compressed t2 ∧ compressed t3)
⟨*proof*⟩

lemma *height-Qc-Q*:
height (Qc t0 t1 t2 t3) ≤ height (Q t0 t1 t2 t3)
⟨*proof*⟩

fun *add* :: *qtr ⇒ qtr ⇒ 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) |
add (L x) (L y) = L(x+y) |
add (L x) (Q t0 t1 t2 t3) = Qc (add (L x) t0) t1 t2 (add (L x) t3) |
add (Q t0 t1 t2 t3) (L x) = Qc (add t0 (L x)) t1 t2 (add t3 (L x))

fun *mult* :: *qtr ⇒ qtr ⇒ 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)) |
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) |
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)) |
mult (L x) (L y) = L(x∗y)

Initialization of *qtr* from ma

fun *qtr* :: *nat ⇒ ma ⇒ qtr* **where**
qtr 0 a = L(a 0 0) |
qtr (Suc n) a =
(let t0 = qtr n a; t1 = qtr n (λi j. a i (j+2^n));
t2 = qtr n (λi j. a (i+2^n) j); t3 = qtr n (λi j. a (i+2^n) (j+2^n))
in Q t0 t1 t2 t3)

4.5 Correctness of Quad Tree Implementations

4.5.1 *add*

lemma *ma-add*: ⟦ *height s ≤ n*; *height t ≤ n* ⟧ ⟹
ma n (add s t) = add-ma (ma n s) (ma n t)
⟨*proof*⟩

lemma *height-add*: *height (add s t) ≤ max (height s) (height t)*
⟨*proof*⟩

lemma *compressed-add*: ⟦ *compressed s*; *compressed t* ⟧ ⟹ *compressed (add s t)*

⟨*proof*⟩

lemma *Max4*: *Max*{*n0*,*n1*,*n2*,*n3*} = *max n0* (*max n1* (*max n2 n3*))⟨*proof*⟩

lemma *height-mult*: *height* (*mult s t*) ≤ *max* (*height s*) (*height t*)
⟨*proof*⟩

4.5.2 *mult*

lemma *bij-betw-minus-ivlco-nat*: *n* ≤ *a* ⟹ *C* = {*a*−*n*..<*b*−*n*} ⟹ *bij-betw* (λ*k*::*nat*. *k*−*n*) {*a*..<*b*} *C*
⟨*proof*⟩

lemma *mult-ma-Qma-Qma*:
mult-ma (*Suc n*) (*Qma n a b c d*) (*Qma n a′ b′ c′ d′*) =
(*Qma n* (*add-ma* (*mult-ma n a a′*) (*mult-ma n b c′*))
(*add-ma* (*mult-ma n a b′*) (*mult-ma n b d′*))
(*add-ma* (*mult-ma n c a′*) (*mult-ma n d c′*))
(*add-ma* (*mult-ma n c b′*) (*mult-ma n d d′*)))
⟨*proof*⟩

lemma *ma-mult*: ⟦ *height s* ≤ *n*; *height t* ≤ *n* ⟧ ⟹
ma n (*mult s t*) = *mult-ma n* (*ma n s*) (*ma n t*)
⟨*proof*⟩

lemma *compressed-mult*: ⟦ *compressed s*; *compressed t* ⟧ ⟹ *compressed* (*mult s t*)
⟨*proof*⟩

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* = (⋃ *a*∈*A*. (#) *a* ‘ *nlists n A*)
⟨*proof*⟩
lemma *in-nlists-UNIV*: *xs* ∈ *nlists k UNIV* ⟷ *length xs* = *k*
⟨*proof*⟩
lemma *nlists-singleton*: *nlists n* {*a*} = {*replicate n a*}
⟨*proof*⟩

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 a*) (*L b*) (*L c*) becomes *Split* (*Box a*) (*Split* (*Box b*) (*Box c*)).

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

fun *height-kdt* :: *'a kdt* ⇒ *nat* **where**
height (*Box* -) = *0* |
height (*Split l r*) = *max* (*height l*) (*height r*) + *1*

instance ⟨*proof*⟩

end

lemma *height-0-iff*: *height t = 0* ⟷ (∃ *x*. *t = Box x*)
⟨*proof*⟩

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

lemma *bits-0*[*code*]: *bits 0* = {[]}
⟨*proof*⟩

lemma *bits-Suc*[*code*]:
bits (*Suc n*) = (*let B = bits n in* (#) *True* ‘ *B* ∪ (#) *False* ‘ *B*)
⟨*proof*⟩

5.1 Subtree

fun *subtree* :: *'a kdt* ⇒ *bool list* ⇒ *'a 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*]: *subtree (Box x) bs = Box x*
⟨*proof*⟩

lemma *height-subtree*: *height (subtree t bs) ≤ height t − length bs*
⟨*proof*⟩

lemma *height-subtree2*: ⟦ *height t ≤ k ∗ (Suc n)*; *length bs = k*⟧ ⟹ *height (subtree t bs) ≤ k ∗ n*
⟨*proof*⟩

lemma *subtree-Split-Box*: *length bs ≠ 0* ⟹ *subtree (Split (Box b) (Box b)) bs = Box b*
⟨*proof*⟩

5.2 Shifting a coordinate by a boolean vector

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

lemma *map-zip1*: ⟦ *length xs = length ys*; ∀ *p* ∈ *set*(*zip xs ys*). *f p = fst p* ⟧ ⟹ *map f (zip xs ys) = xs*
⟨*proof*⟩

lemma *map-mv1*: ⟦ *ps* ∈ *nlists (length bs) {0..<n}*; *length ps = length bs* ⟧
⟹ *map (λi. i < n) (mv (n) bs ps) = bs*
⟨*proof*⟩

lemma *map-zip2*: ⟦ *length xs = length ys*; ∀ *p* ∈ *set*(*zip xs ys*). *f p = snd p* ⟧ ⟹ *map f (zip xs ys) = ys*
⟨*proof*⟩

lemma *map-mv2*: ⟦ *ps* ∈ *nlists (length bs) {0..<2^n}* ⟧ ⟹ *map (λx. x mod 2^n) (mv (2^n) bs ps) = ps*
⟨*proof*⟩

lemma *mv-map-map*: *set ps* ⊆ *{0..<2 ∗ n}* ⟹ *mv (n) (map (λx. x < n) ps) (map (λx. x mod n) ps) = ps*
⟨*proof*⟩

lemma *mv-in-nlists*:
⟦ *p* ∈ *nlists k {0..<2 ^ n}*; *bs* ∈ *bits k* ⟧ ⟹ *mv (2^n) bs p* ∈ *nlists k {0..<2 ∗ 2 ^ n}*
⟨*proof*⟩

lemma *in-nlists2D*: *xs* ∈ *nlists k {0..<2 ∗ 2^n}* ⟹ ∃ *bs*∈ *bits k. xs* ∈ *mv (2^n)*

bs ‘ nlists k {0..<2^n}
⟨*proof*⟩

lemma *nlists2-simp*: *nlists k {0..<2 * 2 ^ n} = (⋃ bs∈ bits k. mv (2^n) bs ‘ nlists k {0..<2 ^ n})*
⟨*proof*⟩

lemma *in-mv-image*: ⟦ *ps ∈ nlists k {0..<2*2^n}*; *Ps ⊆ nlists k {0..<2^n}*; *bs ∈ bits k* ⟧ ⟹
ps ∈ mv (2^n) bs ‘ Ps ⟷ map (λx. x mod 2^n) ps ∈ Ps ∧ (bs = map (λi. i < 2^n) ps)
⟨*proof*⟩

5.3 Points in a tree

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

fun *points* :: *nat ⇒ nat ⇒ kdtb ⇒ nat list set* **where**
points k n (Box b) = (if b then cube k n else {}) |
points k (Suc n) t = (⋃ bs ∈ bits k. mv (2^n) bs ‘ points k n (subtree t bs))

lemma *points-Suc*: *points k (Suc n) t = (⋃ bs ∈ bits k. mv (2^n) bs ‘ points k n (subtree t bs))*
⟨*proof*⟩

lemma *points-subset*: *height t ≤ k*n ⟹ points k n t ⊆ nlists k {0..<2^n}*
⟨*proof*⟩

5.4 Compression

Compressing Split:

fun *SplitC* :: *'a kdt ⇒ 'a kdt ⇒ '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 ⇒ bool* **where**
compressed (Box -) = True |
compressed (Split l r) = (compressed l ∧ compressed r ∧ ¬(∃ b. l = Box b ∧ r = Box b))

lemma *compressedI*: ⟦ *compressed l*; *compressed r* ⟧ ⟹ *compressed (SplitC l r)*
⟨*proof*⟩

lemma *subtree-SplitC*:
1 ≤ length bs ⟹ subtree (SplitC l r) bs = subtree (Split l r) bs
⟨*proof*⟩

lemma *height-SplitC*: *height(SplitC l r) ≤ Suc (max (height l) (height r))*

⟨*proof*⟩

lemma *height-SplitC2*: ⟦ *height l* ≤ *n*; *height r* ≤ *n* ⟧ ⟹ *height*(*SplitC l r*) ≤ *Suc n*
⟨*proof*⟩

5.5 Extracting a point from a tree

Also the abstraction function.

fun *get* :: *nat* ⇒ *'a kdt* ⇒ *nat list* ⇒ *'a* **where**
get - (Box b) - = b |
get (Suc n) t ps = get n (subtree t (map (λi. i < 2^n) ps)) (map (λi. i mod 2^n) ps)

lemma *get-Suc*: *get (Suc n) t ps =*
get n (subtree t (map (λi. i < 2 ^ n) ps)) (map (λi. i mod 2 ^ n) ps)
⟨*proof*⟩

lemma *points-get*: ⟦ *height t* ≤ *k∗n*; *ps* ∈ *nlists k {0..<2^n}* ⟧ ⟹
get n t ps = (ps ∈ *points k n t)*
⟨*proof*⟩

5.6 Modifying a point in a tree

fun *modify* :: (*'a kdt* ⇒ *'a kdt*) ⇒ *bool list* ⇒ *'a kdt* ⇒ *'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* ⇒ *'a* ⇒ *nat* ⇒ *'a kdt* ⇒ *'a kdt* **where**
put ps a 0 (Box -) = Box a |
put ps a (Suc n) t = modify (put (map (λi. i mod 2^n) ps) a n) (map (λi. i < 2^n) ps) t

lemma *height-modify*: ⟦ ∀ *t*. *height t* ≤ *nk* ⟶ *height (f t)* ≤ *nk*;
height t ≤ *k* + *nk*; *length bs* = *k*⟧
⟹ *height (modify f bs t)* ≤ *k* + *nk*
⟨*proof*⟩

lemma *height-put*: *height t* ≤ *n* ∗ *length ps* ⟹ *height (put ps a n t)* ≤ *n* ∗ *length ps*
⟨*proof*⟩

lemma *subtree-modify*: ⟦ *length bs'* = *length bs* ⟧
⟹ *subtree (modify f bs t) bs'* = *(if bs'* = *bs then f*(*subtree t bs*) *else subtree t bs')*

⟨*proof*⟩

lemma *mod-eq1*: ⟦ $y < 2 * n$; $ya < 2 * n$; $\neg\ ya < n$; $\neg\ y < n$; $ya\ mod\ n = y\ mod\ n$⟧
$\Longrightarrow ya = (y::nat)$
⟨*proof*⟩

lemma *nlist-eq-mod*: ⟦ $ps \in nlists\ k\ \{0..<(2::nat) * 2\ \hat{}\ n\}$; $ps' \in nlists\ k\ \{0..<2 * 2\ \hat{}\ n\}$;
$map\ (\lambda i.\ i < 2\ \hat{}\ n)\ ps' = map\ (\lambda i.\ i < 2\ \hat{}\ n)\ ps$; $ps' \neq ps$ ⟧ $\Longrightarrow$
$map\ (\lambda i.\ i\ mod\ 2\ \hat{}\ n)\ ps' \neq map\ (\lambda i.\ i\ mod\ 2\ \hat{}\ n)\ ps$
⟨*proof*⟩

lemma *get-put*: ⟦ $height\ t \leq k*n$; $ps \in cube\ k\ n$; $ps' \in cube\ k\ n$ ⟧ $\Longrightarrow$
$get\ n\ (put\ ps\ a\ n\ t)\ ps' = (if\ ps' = ps\ then\ a\ else\ get\ n\ t\ ps')$
⟨*proof*⟩

lemma *compressed-modify*: ⟦ $compressed\ t$; $compressed\ (f\ (subtree\ t\ bs))$ ⟧ $\Longrightarrow$ $compressed\ (modify\ f\ bs\ t)$
⟨*proof*⟩

lemma *compressed-subtree*: $compressed\ t \Longrightarrow compressed\ (subtree\ t\ bs)$
⟨*proof*⟩

lemma *compressed-put*:
⟦ $height\ t \leq k*n$; $k = length\ ps$; $compressed\ t$ ⟧ $\Longrightarrow$ $compressed\ (put\ ps\ a\ n\ t)$
⟨*proof*⟩

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*: $union\ t\ (Box\ b) = (if\ b\ then\ Box\ True\ else\ t)$
⟨*proof*⟩

lemma *subtree-union*: $subtree\ (union\ t1\ t2)\ bs = union\ (subtree\ t1\ bs)\ (subtree\ t2\ bs)$
⟨*proof*⟩

lemma *points-union*:
⟦ $max\ (height\ t1)\ (height\ t2) \leq k*n$ ⟧ $\Longrightarrow$
$points\ k\ n\ (union\ t1\ t2) = points\ k\ n\ t1 \cup points\ k\ n\ t2$
⟨*proof*⟩

lemma *get-union*:
⟦ $max\ (height\ t1)\ (height\ t2) \leq length\ ps * n$ ⟧ $\Longrightarrow$

get n (union t1 t2) ps = (get n t1 ps ∨ get n t2 ps)
⟨*proof*⟩

lemma *height-union*: *height (union t1 t2) ≤ max (height t1) (height t2)*
⟨*proof*⟩

lemma *compressed-union*: *compressed t1 ⟹ compressed t2 ⟹ compressed*(*union t1 t2*)
⟨*proof*⟩

end

6 K-dimensional Region Trees - Version 2

theory *KD-Region-Tree2*
imports
HOL−Library.NList
HOL−Library.Tree
begin

lemma *nlists-Suc*: *nlists (Suc n) A* = (⋃ *a*∈*A*. (#) *a* ‘ *nlists n A*)
⟨*proof*⟩

lemma *in-nlists-UNIV*: *xs ∈ nlists k UNIV ⟷ length xs = k*
⟨*proof*⟩

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 a*) (*L b*) (*L c*) becomes *Split* (*Box a*) (*Split* (*Box b*) (*Box c*)).

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

fun *height-kdt* :: *'a kdt* ⇒ *nat* **where**
height (*Box* -) = *0* |
height (*Split l r*) = *max* (*height l*) (*height r*) + *1*

instance ⟨*proof*⟩

end

lemma *height-0-iff*: *height t* = *0* ⟷ (∃ *x*. *t* = *Box x*)
⟨*proof*⟩

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

lemma *bits-Suc*[*code*]:
bits (*Suc n*) = (*let B* = *bits n in* (#) *True* ‘ *B* ∪ (#) *False* ‘ *B*)
⟨*proof*⟩

6.1 Subtree

fun *subtree* :: *'a kdt* ⇒ *bool list* ⇒ *'a 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*]: *subtree* (*Box x*) *bs* = *Box x*
⟨*proof*⟩

lemma *height-subtree*: *height* (*subtree t bs*) ≤ *height t* − *length bs*
⟨*proof*⟩

lemma *height-subtree2*: ⟦ *height t* ≤ *k* ∗ (*Suc n*); *length bs* = *k*⟧ ⟹ *height* (*subtree t bs*) ≤ *k* ∗ *n*
⟨*proof*⟩

lemma *subtree-Split-Box*: *length bs* ≠ *0* ⟹ *subtree* (*Split* (*Box b*) (*Box b*)) *bs* = *Box b*
⟨*proof*⟩

6.2 Shifting a coordinate by a boolean vector

The ?

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

lemma *map-zip1*: ⟦ *length xs* = *length ys*; ∀ *p* ∈ *set*(*zip xs ys*). *f p* = *fst p* ⟧ ⟹ *map f* (*zip xs ys*) = *xs*
⟨*proof*⟩

lemma *map-mv1*: ⟦ *length ps* = *length bs* ⟧ ⟹ *map even* (*mv bs ps*) = *bs*
⟨*proof*⟩

lemma *map-zip2*: ⟦ *length xs* = *length ys*; ∀ *p* ∈ *set*(*zip xs ys*). *f p* = *snd p* ⟧ ⟹ *map f* (*zip xs ys*) = *ys*
⟨*proof*⟩

lemma *map-mv2*: ⟦ *length ps* = *length bs* ⟧ ⟹ *map* (λ*x*. *x div 2*) (*mv bs ps*) = *ps*
⟨*proof*⟩

lemma *mv-map-map*: *mv* (*map even ps*) (*map* (λ*x*. *x div 2*) *ps*) = *ps*
⟨*proof*⟩

lemma *mv-in-nlists*:
⟦ *p* ∈ *nlists k* {*0..<2* ^ *n*}; *bs* ∈ *bits k* ⟧ ⟹ *mv bs p* ∈ *nlists k* {*0..<2* ∗ *2* ^ *n*}
⟨*proof*⟩

lemma *in-nlists2D*: *xs* ∈ *nlists k* {*0..<2* ∗ *2* ^ *n*} ⟹ ∃ *bs*∈ *bits k*. *xs* ∈ *mv bs* ‘ *nlists k* {*0..<2* ^ *n*}
⟨*proof*⟩

lemma *nlists2-simp*: *nlists k* {*0..<2* ∗ *2* ^ *n*} = (⋃ *bs*∈ *bits k*. *mv bs* ‘ *nlists k* {*0..<2* ^ *n*})
⟨*proof*⟩

6.3 Points in a tree

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

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

lemma *points-Suc*: *points k* (*Suc n*) *t* = (⋃ *bs* ∈ *bits k*. *mv bs* ‘ *points k n* (*subtree t bs*))
⟨*proof*⟩

lemma *points-subset*: *height t* ≤ *k*∗*n* ⟹ *points k n t* ⊆ *nlists k* {*0..<2*^*n*}
⟨*proof*⟩

6.4 Compression

Compressing Split:

fun *SplitC* :: *'a kdt* ⇒ *'a kdt* ⇒ *'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 ⇒ bool* **where**
compressed (Box -) = True |
compressed (Split l r) = (compressed l ∧ compressed r ∧ ¬(∃ b. l = Box b ∧ r = Box b))

lemma *compressedI*: ⟦ *compressed t1*; *compressed t2* ⟧ ⟹ *compressed (SplitC t1 t2)*
⟨*proof*⟩

lemma *subtree-SplitC*:
1 ≤ length bs ⟹ subtree (SplitC l r) bs = subtree (Split l r) bs
⟨*proof*⟩

6.5 Union

fun *union* :: *kdtb ⇒ kdtb ⇒ 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)*
⟨*proof*⟩

lemma *in-mv-image*: ⟦ *ps ∈ nlists k {0..<2∗2^n}*; *Ps ⊆ nlists k {0..<2^n}*; *bs ∈ bits k* ⟧ ⟹
ps ∈ mv bs ‘ Ps ⟷ map (λx. x div 2) ps ∈ Ps ∧ (bs = map even ps)
⟨*proof*⟩

lemma *subtree-union*: *subtree (union t1 t2) bs = union (subtree t1 bs) (subtree t2 bs)*
⟨*proof*⟩

lemma *points-union*:
⟦ *max (height t1) (height t2) ≤ k∗n* ⟧ ⟹
points k n (union t1 t2) = points k n t1 ∪ points k n t2
⟨*proof*⟩

lemma *compressed-union*: *compressed t1 ⟹ compressed t2 ⟹ compressed(union t1 t2)*
⟨*proof*⟩

6.6 Extracting a point from a tree

lemma *size-subtree*: *bs ≠ [] ⟹ (∀ b. t ≠ Box b) ⟹ size (subtree t bs) < size t*
⟨*proof*⟩

For termination of *get*:

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

bs ≠ [] ⟹ *size* (*subtree* (*Split l r*) *bs*) < *Suc* (*size l* + *size r*)
⟨*proof*⟩

fun *get* :: *'a kdt* ⇒ *nat list* ⇒ *'a* **where**
get (*Box b*) - = *b* |
get t ps = (*if ps*=[] *then undefined else get* (*subtree t* (*map even ps*)) (*map* (λ*i*. *i div 2*) *ps*))

lemma *points-get*: ⟦ *height t* ≤ *k*∗*n*; *ps* ∈ *nlists k* {*0*..<*2*^*n*} ⟧ ⟹
get t ps = (*ps* ∈ *points k n t*)
⟨*proof*⟩

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*)
⟨*proof*⟩
lemma *nlists-singleton*: *nlists n* {*a*} = {*replicate n a*}
⟨*proof*⟩

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* ≠ {}
⟨*proof*⟩

lemma *kdt-tree1-term*[*termination-simp*]: *x* ∈ *set-tree1 t* ⟹ *size-kdt f x* < *Suc* (*size-tree1* (*size-kdt f*) *t*)
⟨*proof*⟩

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*
⟨*proof*⟩
termination
⟨*proof*⟩

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*))
⟨*proof*⟩
termination
⟨*proof*⟩

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

lemma *bits-0*[*code*]: *bits 0* = {[]}
⟨*proof*⟩

lemma *bits-Suc*[*code*]: *bits* (*Suc n*) = (*let B* = *bits n in* (#) *True* ‘ *B* ∪ (#) *False* ‘ *B*)
⟨*proof*⟩

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* (λ*b 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* ≠ {}
⟨*proof*⟩

lemma *finite-bits*: *finite* (*bits n*)
⟨*proof*⟩

lemma *mv-in-nlists*:
⟦ *p* ∈ *nlists k* {*0*..<*2* ^ *n*}; *bs* ∈ *bits k* ⟧ ⟹ *mv bs p* ∈ *nlists k* {*0*..<*2* ∗ *2* ^ *n*}
⟨*proof*⟩

lemma *leaf-append*: *length bs* ≥ *h-tree1 t* ⟹ *leaf t* (*bs*@*bs'*) = *leaf t bs*
⟨*proof*⟩

lemma *leaf-take*: *length bs* $\geq$ *h-tree1 t* $\Longrightarrow$ *leaf t* (*bs*) = *leaf t* (*take* (*h-tree1 t*) *bs*)
⟨*proof*⟩

lemma *Union-bits-le*:
h-tree1 t $\leq$ *n* $\Longrightarrow$ ($\bigcup$ *bs*$\in$*bits n*. {*leaf t bs*}) = ($\bigcup$ *bs*$\in$*bits* (*h-tree1 t*). {*leaf t bs*})
⟨*proof*⟩

lemma *set-tree1-leafs*:
set-tree1 t = ($\bigcup$ *bs* $\in$ *bits* (*h-tree1 t*). {*leaf t bs*})
⟨*proof*⟩

lemma *points-subset*: *inv-kdt k t* $\Longrightarrow$ *h-kdt t* $\leq$ *n* $\Longrightarrow$ *points k n t* $\subseteq$ *nlists k* {*0..<2^n*}
⟨*proof*⟩

fun *comb1* :: ('*a* $\Rightarrow$ '*a* $\Rightarrow$ '*a*) $\Rightarrow$ '*a tree1* $\Rightarrow$ '*a tree1* $\Rightarrow$ '*a 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*]:
⟦*s1* = *t1*; *s2* = *t2*; $\bigwedge$*x y*. *x* $\in$ *set-tree1 t1* $\Longrightarrow$ *y* $\in$ *set-tree1 t2* $\Longrightarrow$ *f x y* = *g x y*⟧ $\Longrightarrow$ *comb1 f s1 s2* = *comb1 g t1 t2*
⟨*proof*⟩

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*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *leaf-comb1*:
⟦ *length bs* $\geq$ *max* (*h-tree1 t1*) (*h-tree1 t2*) ⟧ $\Longrightarrow$
leaf (*comb1 f t1 t2*) *bs* = *f* (*leaf t1 bs*) (*leaf t2 bs*)
⟨*proof*⟩

lemma *leaf-in-set-tree1*: ⟦ *length bs* $\geq$ *h-tree1 t* ⟧ $\Longrightarrow$ *leaf t bs* $\in$ *set-tree1 t*
⟨*proof*⟩

lemma *leaf-in-set-tree2*: $[\![x \in nlists\ k\ UNIV;\ h\text{-}tree1\ t1 \leq k]\!] \Longrightarrow leaf\ t1\ x \in set\text{-}tree1\ t1$
⟨*proof*⟩

lemma *points-union*:
$[\![\ inv\text{-}kdt\ k\ t1;\ inv\text{-}kdt\ k\ t2;\ n \geq max\ (h\text{-}kdt\ t1)\ (h\text{-}kdt\ t2)\ ]\!] \Longrightarrow$
$points\ k\ n\ (union\ t1\ t2) = points\ k\ n\ t1 \cup points\ k\ n\ t2$
⟨*proof*⟩

lemma *size-leaf*[*termination-simp*]: $size\ (leaf\ t\ (map\ f\ ps)) < Suc\ (size\text{-}tree1\ size\ t)$
⟨*proof*⟩

fun *get* :: *'a kdt* ⇒ *nat list* ⇒ *'a* **where**
get (*Cube b*) - = *b* |
get (*Dims t*) *ps* = *get* (*leaf t* (*map even ps*)) (*map* (λ*x*. *x div 2*) *ps*)

lemma *map-zip1*: $[\![\ length\ xs = length\ ys;\ \forall p \in set(zip\ xs\ ys).\ f\ p = fst\ p\ ]\!] \Longrightarrow map\ f\ (zip\ xs\ ys) = xs$
⟨*proof*⟩

lemma *map-mv1*: $[\![\ length\ ps = length\ bs\ ]\!] \Longrightarrow map\ even\ (mv\ bs\ ps) = bs$
⟨*proof*⟩

lemma *map-zip2*: $[\![\ length\ xs = length\ ys;\ \forall p \in set(zip\ xs\ ys).\ f\ p = snd\ p\ ]\!] \Longrightarrow map\ f\ (zip\ xs\ ys) = ys$
⟨*proof*⟩

lemma *map-mv2*: $[\![\ length\ ps = length\ bs\ ]\!] \Longrightarrow map\ (\lambda x.\ x\ div\ 2)\ (mv\ bs\ ps) = ps$
⟨*proof*⟩

lemma *mv-map-map*: $mv\ (map\ even\ ps)\ (map\ (\lambda x.\ x\ div\ 2)\ ps) = ps$
⟨*proof*⟩

lemma *in-mv-image*: $[\![\ ps \in nlists\ k\ \{0..<2*2\hat{}n\};\ Ps \subseteq nlists\ k\ \{0..<2\hat{}n\};\ bs \in bits\ k\ ]\!] \Longrightarrow$
$ps \in mv\ bs\ `\ Ps \longleftrightarrow map\ (\lambda x.\ x\ div\ 2)\ ps \in Ps \wedge (bs = map\ even\ ps)$
⟨*proof*⟩

lemma *get-points*: $[\![\ inv\text{-}kdt\ k\ t;\ h\text{-}kdt\ t \leq n;\ ps \in nlists\ k\ \{0..<2\hat{}n\}\ ]\!] \Longrightarrow$
$get\ t\ ps = (ps \in points\ k\ n\ t)$
⟨*proof*⟩

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* (λ*i*. *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'*)
⟨*proof*⟩

lemma *in-nlists2D*: *xs* ∈ *nlists k* {*0*..<*2* ∗ *2* ^ *n*} ⟹ ∃ *bs*∈*nlists k UNIV*. *xs* ∈ *mv bs* ‘ *nlists k* {*0*..<*2* ^ *n*}
⟨*proof*⟩

lemma *nlists2-simp*: *nlists k* {*0*..<*2* ∗ *2* ^ *n*} = (⋃ *bs*∈*nlists k UNIV*. *mv bs* ‘ *nlists k* {*0*..<*2* ^ *n*})
⟨*proof*⟩

lemma *mv-diff*:
  ⟦ *length qs* = *length bs*; ∀ *as* ∈ *A*. *length as* = *length bs* ⟧ ⟹ *mv bs* ‘ (*A* − {*qs*}) = *mv bs* ‘ *A* − {*mv bs qs*}
⟨*proof*⟩

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*⟩

end

References

[1] S. Aluru. Quadtrees 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.