

# Closest Pair of Points Algorithms

Martin Rau and Tobias Nipkow

April 14, 2026

## Abstract

This entry provides two related verified divide-and-conquer algorithms solving the fundamental *Closest Pair of Points* problem in Computational Geometry. Functional correctness and the optimal running time of  $\mathcal{O}(n \log n)$  are proved. Executable code is generated which is empirically competitive with handwritten reference implementations.

## Contents

<b>1</b>	<b>Common</b>	<b>3</b>
1.1	Auxiliary Functions and Lemmas . . . . .	3
1.1.1	Time Monad . . . . .	3
1.1.2	Landau Auxiliary . . . . .	3
1.1.3	Miscellaneous Lemmas . . . . .	4
1.1.4	<i>length</i> . . . . .	6
1.1.5	<i>rev</i> . . . . .	6
1.1.6	<i>take</i> . . . . .	7
1.1.7	<i>filter</i> . . . . .	7
1.1.8	<i>split-at</i> . . . . .	8
1.2	Mergesort . . . . .	9
1.2.1	Functional Correctness Proof . . . . .	9
1.2.2	Time Complexity Proof . . . . .	12
1.2.3	Code Export . . . . .	14
1.3	Minimal Distance . . . . .	15
1.4	Distance . . . . .	15
1.5	Brute Force Closest Pair Algorithm . . . . .	16
1.5.1	Functional Correctness Proof . . . . .	16
1.5.2	Time Complexity Proof . . . . .	19
1.5.3	Code Export . . . . .	20
1.6	Geometry . . . . .	22
1.6.1	Band Filter . . . . .	22
1.6.2	2D-Boxes and Points . . . . .	23
1.6.3	Pigeonhole Argument . . . . .	24
1.6.4	Delta Sparse Points within a Square . . . . .	26

<b>2</b>	<b>Closest Pair Algorithm</b>	<b>28</b>
2.1	Functional Correctness Proof . . . . .	28
2.1.1	Combine Step . . . . .	28
2.1.2	Divide and Conquer Algorithm . . . . .	36
2.2	Time Complexity Proof . . . . .	44
2.2.1	Core Argument . . . . .	44
2.2.2	Combine Step . . . . .	45
2.2.3	Divide and Conquer Algorithm . . . . .	49
2.3	Code Export . . . . .	54
2.3.1	Combine Step . . . . .	54
2.3.2	Divide and Conquer Algorithm . . . . .	59
<b>3</b>	<b>Closest Pair Algorithm 2</b>	<b>64</b>
3.1	Functional Correctness Proof . . . . .	64
3.1.1	Core Argument . . . . .	64
3.1.2	Combine step . . . . .	67
3.1.3	Divide and Conquer Algorithm . . . . .	75
3.2	Time Complexity Proof . . . . .	83
3.2.1	Combine Step . . . . .	83
3.2.2	Divide and Conquer Algorithm . . . . .	84
3.3	Code Export . . . . .	88
3.3.1	Combine Step . . . . .	88
3.3.2	Divide and Conquer Algorithm . . . . .	91

# 1 Common

```
theory Common
imports
  HOL-Library.Going-To-Filter
  Akra-Bazzi.Akra-Bazzi-Method
  Akra-Bazzi.Akra-Bazzi-Approximation
  HOL-Library.Code-Target-Numeral
  Root-Balanced-Tree.Time-Monad
begin
```

```
type-synonym point = int * int
```

## 1.1 Auxiliary Functions and Lemmas

### 1.1.1 Time Monad

```
lemma time-distrib-bind:
  time (bind-tm tm f) = time tm + time (f (val tm))
unfolding bind-tm-def by (simp split: tm.split)
```

```
lemmas time-simps = time-distrib-bind tick-def
```

```
lemma bind-tm-cong[fundef-cong]:
  assumes  $\bigwedge v. v = \text{val } n \implies f v = g v m = n$ 
  shows bind-tm m f = bind-tm n g
  using assms unfolding bind-tm-def by (auto split: tm.split)
```

### 1.1.2 Landau Auxiliary

The following lemma expresses a procedure for deriving complexity properties of the form  $t \in O[m \text{ going-to at-top within } A](f \circ m)$  where

- $t$  is a (timing) function on same data domain (e.g. lists),
- $m$  is a measure function on that data domain (e.g. length),
- $t'$  is a function on  $\text{nat}$ ,
- $A$  is the set of valid inputs for the data domain. One needs to show that
- $t$  is bounded by  $t' \circ m$  for valid inputs
- $t' \in O(f)$  to conclude the overall property  $t \in O[m \text{ going-to at-top within } A](f \circ m)$ .

```
lemma bigo-measure-trans:
  fixes  $t :: 'a \Rightarrow \text{real}$  and  $t' :: \text{nat} \Rightarrow \text{real}$  and  $m :: 'a \Rightarrow \text{nat}$  and  $f :: \text{nat} \Rightarrow \text{real}$ 
  assumes  $\bigwedge x. x \in A \implies t x \leq (t' \circ m) x$ 
```

```

    and  $t' \in O(f)$ 
    and  $\bigwedge x. x \in A \implies 0 \leq t x$ 
  shows  $t \in O[m \text{ going-to at-top within } A](f \circ m)$ 
proof -
  have 0:  $\bigwedge x. x \in A \implies 0 \leq (t' \circ m) x$  by (meson assms(1,3) order-trans)
  have 1:  $t \in O[m \text{ going-to at-top within } A](t' \circ m)$ 
    apply(rule bigoI[where c=1]) using assms 0
    by (simp add: eventually-inf-principal going-to-within-def)
  have 2:  $t' \circ m \in O[m \text{ going-to at-top}](f \circ m)$ 
    unfolding o-def going-to-def
    by(rule landau-o.big.filtercomap[OF assms(2)])
  have 3:  $t' \circ m \in O[m \text{ going-to at-top within } A](f \circ m)$ 
    using landau-o.big.filter-mono[OF -2] going-to-mono[OF -subset-UNIV] by
blast
  show ?thesis by(rule landau-o.big-trans[OF 1 3])
qed

```

**lemma** *const-1-bigo-n-ln-n:*

$(\lambda(n::nat). 1) \in O(\lambda n. n * \ln n)$

```

proof -
  have  $\exists N. \forall (n::nat) \geq N. (\lambda x. 1 \leq x * \ln x) n$ 
  proof -
    have  $\forall (n::nat) \geq 3. (\lambda x. 1 \leq x * \ln x) n$ 
    proof standard
      fix n
      show  $3 \leq n \longrightarrow 1 \leq \text{real } n * \ln (\text{real } n)$ 
      proof standard
        assume  $3 \leq n$ 
        hence A:  $1 \leq \text{real } n$ 
        by simp
        have B:  $1 \leq \ln (\text{real } n)$ 
        using ln-ln-nonneg' ‹ $3 \leq n$ › by simp
        show  $1 \leq \text{real } n * \ln (\text{real } n)$ 
        using mult-mono [OF A B] by simp
      qed
    qed
  qed
  thus ?thesis
  by blast
qed
  thus ?thesis
  by auto
qed

```

### 1.1.3 Miscellaneous Lemmas

**lemma** *set-take-drop-i-le-j:*

$i \leq j \implies \text{set } xs = \text{set } (\text{take } j \text{ } xs) \cup \text{set } (\text{drop } i \text{ } xs)$

**proof** (*induction xs arbitrary: i j*)

case (*Cons x xs*)

```

show ?case
proof (cases i = 0)
  case True
  thus ?thesis
  using set-take-subset by force
next
  case False
  hence set xs = set (take (j - 1) xs) ∪ set (drop (i - 1) xs)
  by (simp add: Cons diff-le-mono)
  moreover have set (take j (x # xs)) = insert x (set (take (j - 1) xs))
  using False Cons.prem1 by (auto simp: take-Cons')
  moreover have set (drop i (x # xs)) = set (drop (i - 1) xs)
  using False Cons.prem1 by (auto simp: drop-Cons')
  ultimately show ?thesis
  by auto
qed
qed simp

```

```

lemma set-take-drop:
  set xs = set (take n xs) ∪ set (drop n xs)
  using set-take-drop-i-le-j by fast

```

```

lemma sorted-wrt-take-drop:
  sorted-wrt f xs  $\implies \forall x \in \text{set } (take\ n\ xs). \forall y \in \text{set } (drop\ n\ xs). f\ x\ y$ 
  using sorted-wrt-append[of f take n xs drop n xs] by simp

```

```

lemma sorted-wrt-hd-less:
  assumes sorted-wrt f xs  $\wedge x. f\ x\ x$ 
  shows  $\forall x \in \text{set } xs. f\ (\text{hd } xs)\ x$ 
  using assms by (cases xs) auto

```

```

lemma sorted-wrt-hd-less-take:
  assumes sorted-wrt f (x # xs)  $\wedge x. f\ x\ x$ 
  shows  $\forall y \in \text{set } (take\ n\ (x\ \#\ xs)). f\ x\ y$ 
  using assms sorted-wrt-hd-less [of f <x # xs>] in-set-takeD [of - n <x # xs>]
  by auto

```

```

lemma sorted-wrt-take-less-hd-drop:
  assumes sorted-wrt f xs n < length xs
  shows  $\forall x \in \text{set } (take\ n\ xs). f\ x\ (\text{hd } (drop\ n\ xs))$ 
  using sorted-wrt-take-drop assms by fastforce

```

```

lemma sorted-wrt-hd-drop-less-drop:
  assumes sorted-wrt f xs  $\wedge x. f\ x\ x$ 
  shows  $\forall x \in \text{set } (drop\ n\ xs). f\ (\text{hd } (drop\ n\ xs))\ x$ 
  using assms sorted-wrt-drop sorted-wrt-hd-less by blast

```

```

lemma length-filter-P-impl-Q:
  ( $\wedge x. P\ x \implies Q\ x$ )  $\implies \text{length } (\text{filter } P\ xs) \leq \text{length } (\text{filter } Q\ xs)$ 

```

**by** (*induction xs*) *auto*

**lemma** *filter-Un*:

$set\ xs = A \cup B \implies set\ (filter\ P\ xs) = \{x \in A. P\ x\} \cup \{x \in B. P\ x\}$   
**by** (*induction xs*) (*auto, metis UnI1 insert-iff, metis UnI2 insert-iff*)

#### 1.1.4 *length*

**fun** *length-tm* :: 'a list  $\Rightarrow$  nat tm **where**

*length-tm* [] = 1 return 0  
| *length-tm* (x # xs) = 1  
  do {  
    l <- *length-tm* xs;  
    return (1 + l)  
  }

**lemma** *length-eq-val-length-tm*:

*val* (*length-tm* xs) = *length* xs  
**by** (*induction xs*) *auto*

**lemma** *time-length-tm*:

*time* (*length-tm* xs) = *length* xs + 1  
**by** (*induction xs*) (*auto simp: time-simps*)

**fun** *length-it'* :: nat  $\Rightarrow$  'a list  $\Rightarrow$  nat **where**

*length-it'* acc [] = acc  
| *length-it'* acc (x#xs) = *length-it'* (acc+1) xs

**definition** *length-it* :: 'a list  $\Rightarrow$  nat **where**

*length-it* xs = *length-it'* 0 xs

**lemma** *length-conv-length-it'*:

*length* xs + acc = *length-it'* acc xs  
**by** (*induction acc xs rule: length-it'.induct*) *auto*

**lemma** *length-conv-length-it*[*code-unfold*]:

*length* xs = *length-it* xs  
**unfolding** *length-it-def* **using** *length-conv-length-it' add-0-right* **by** *metis*

#### 1.1.5 *rev*

**fun** *rev-it'* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list **where**

*rev-it'* acc [] = acc  
| *rev-it'* acc (x#xs) = *rev-it'* (x#acc) xs

**definition** *rev-it* :: 'a list  $\Rightarrow$  'a list **where**

*rev-it* xs = *rev-it'* [] xs

**lemma** *rev-conv-rev-it'*:

*rev* xs @ acc = *rev-it'* acc xs

**by** (*induction acc xs rule: rev-it'.induct*) *auto*

**lemma** *rev-conv-rev-it*[*code-unfold*]:

*rev xs = rev-it xs*

**unfolding** *rev-it-def* **using** *rev-conv-rev-it' append-Nil2* **by** *metis*

### 1.1.6 *take*

**fun** *take-tm* :: *nat*  $\Rightarrow$  '*a list*  $\Rightarrow$  '*a list tm* **where**

*take-tm* *n* [] =1 *return* []

| *take-tm* *n* (*x* # *xs*) =1

(*case n of*

0  $\Rightarrow$  *return* []

| *Suc m*  $\Rightarrow$  *do* {

*ys* <- *take-tm m xs*;

*return* (*x* # *ys*)

}

)

**lemma** *take-eq-val-take-tm*:

*val* (*take-tm n xs*) = *take n xs*

**by** (*induction xs arbitrary: n*) (*auto split: nat.split*)

**lemma** *time-take-tm*:

*time* (*take-tm n xs*) = *min n (length xs) + 1*

**by** (*induction xs arbitrary: n*) (*auto simp: time-simps split: nat.split*)

### 1.1.7 *filter*

**fun** *filter-tm* :: ('*a*  $\Rightarrow$  *bool*)  $\Rightarrow$  '*a list*  $\Rightarrow$  '*a list tm* **where**

*filter-tm P* [] =1 *return* []

| *filter-tm P* (*x* # *xs*) =1

(*if P x then*

*do* {

*ys* <- *filter-tm P xs*;

*return* (*x* # *ys*)

}

*else*

*filter-tm P xs*

)

**lemma** *filter-eq-val-filter-tm*:

*val* (*filter-tm P xs*) = *filter P xs*

**by** (*induction xs*) *auto*

**lemma** *time-filter-tm*:

*time* (*filter-tm P xs*) = *length xs + 1*

**by** (*induction xs*) (*auto simp: time-simps*)

**fun** *filter-it'* :: '*a list*  $\Rightarrow$  ('*a*  $\Rightarrow$  *bool*)  $\Rightarrow$  '*a list*  $\Rightarrow$  '*a list* **where**

```

  filter-it' acc P [] = rev acc
| filter-it' acc P (x#xs) = (
  if P x then
    filter-it' (x#acc) P xs
  else
    filter-it' acc P xs
)

```

**definition** *filter-it* :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list **where**  
*filter-it* P xs = *filter-it'* [] P xs

**lemma** *filter-conv-filter-it'*:  
 rev acc @ filter P xs = *filter-it'* acc P xs  
**by** (induction acc P xs rule: *filter-it'.induct*) auto

**lemma** *filter-conv-filter-it[code-unfold]*:  
 filter P xs = *filter-it* P xs  
**unfolding** *filter-it-def* **using** *filter-conv-filter-it'* *append-Nil* *rev.simps(1)* **by**  
*metis*

### 1.1.8 split-at

**fun** *split-at-tm* :: nat ⇒ 'a list ⇒ ('a list × 'a list) tm **where**  
*split-at-tm* n [] =1 return ([], [])  
| *split-at-tm* n (x # xs) =1 (
 case n of
 0 ⇒ return ([], x # xs)
 | Suc m ⇒
 do {
 (xs', ys') <- *split-at-tm* m xs;
 return (x # xs', ys')
 }
)

**fun** *split-at* :: nat ⇒ 'a list ⇒ 'a list × 'a list **where**  
*split-at* n [] = ([], [])  
| *split-at* n (x # xs) = (
 case n of
 0 ⇒ ([], x # xs)
 | Suc m ⇒
 let (xs', ys') = *split-at* m xs in
 (x # xs', ys')
)

**lemma** *split-at-eq-val-split-at-tm*:  
 val (*split-at-tm* n xs) = *split-at* n xs  
**by** (induction xs arbitrary: n) (auto *split*: nat.*split* prod.*split*)

**lemma** *split-at-take-drop-conv*:

*split-at n xs = (take n xs, drop n xs)*  
**by** (*induction xs arbitrary: n*) (*auto simp: split: nat.split*)

**lemma** *time-split-at-tm*:  
*time (split-at-tm n xs) = min n (length xs) + 1*  
**by** (*induction xs arbitrary: n*) (*auto simp: time-simps split: nat.split prod.split*)

**fun** *split-at-it'* :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  'a list  $\Rightarrow$  ('a list \* 'a list) **where**  
*split-at-it' acc n [] = (rev acc, [])*  
| *split-at-it' acc n (x#xs) = (*  
  *case n of*  
  0  $\Rightarrow$  (*rev acc, x#xs*)  
  | *Suc m  $\Rightarrow$  split-at-it' (x#acc) m xs*  
*)*

**definition** *split-at-it* :: nat  $\Rightarrow$  'a list  $\Rightarrow$  ('a list \* 'a list) **where**  
*split-at-it n xs = split-at-it' [] n xs*

**lemma** *split-at-conv-split-at-it'*:  
**assumes** (*ts, ds*) = *split-at n xs (ts', ds') = split-at-it' acc n xs*  
**shows** *rev acc @ ts = ts'*  
  **and** *ds = ds'*  
**using** *assms*  
**by** (*induction acc n xs arbitrary: ts rule: split-at-it'.induct*)  
  (*auto simp: split: prod.splits nat.splits*)

**lemma** *split-at-conv-split-at-it-prod*:  
**assumes** (*ts, ds*) = *split-at n xs (ts', ds') = split-at-it n xs*  
**shows** (*ts, ds*) = (*ts', ds'*)  
**using** *assms unfolding split-at-it-def*  
**using** *split-at-conv-split-at-it' rev.simps(1) append-Nil by fast+*

**lemma** *split-at-conv-split-at-it[code-unfold]*:  
*split-at n xs = split-at-it n xs*  
**using** *split-at-conv-split-at-it-prod surj-pair by metis*

**declare** *split-at-tm.simps [simp del]*  
**declare** *split-at.simps [simp del]*

## 1.2 Mergesort

### 1.2.1 Functional Correctness Proof

**definition** *sorted-fst* :: point list  $\Rightarrow$  bool **where**  
*sorted-fst ps = sorted-wrt ( $\lambda p_0 p_1. fst p_0 \leq fst p_1$ ) ps*

**definition** *sorted-snd* :: point list  $\Rightarrow$  bool **where**  
*sorted-snd ps = sorted-wrt ( $\lambda p_0 p_1. snd p_0 \leq snd p_1$ ) ps*

**fun** *merge-tm* :: ('b  $\Rightarrow$  'a::linorder)  $\Rightarrow$  'b list  $\Rightarrow$  'b list  $\Rightarrow$  'b list tm **where**

```

merge-tm f (x # xs) (y # ys) = 1 (
  if f x ≤ f y then
    do {
      tl <- merge-tm f xs (y # ys);
      return (x # tl)
    }
  else
    do {
      tl <- merge-tm f (x # xs) ys;
      return (y # tl)
    }
)
| merge-tm f [] ys = 1 return ys
| merge-tm f xs [] = 1 return xs

fun merge :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list ⇒ 'b list where
  merge f (x # xs) (y # ys) = (
    if f x ≤ f y then
      x # merge f xs (y # ys)
    else
      y # merge f (x # xs) ys
  )
| merge f [] ys = ys
| merge f xs [] = xs

lemma merge-eq-val-merge-tm:
  val (merge-tm f xs ys) = merge f xs ys
  by (induction f xs ys rule: merge.induct) auto

lemma length-merge:
  length (merge f xs ys) = length xs + length ys
  by (induction f xs ys rule: merge.induct) auto

lemma set-merge:
  set (merge f xs ys) = set xs ∪ set ys
  by (induction f xs ys rule: merge.induct) auto

lemma distinct-merge:
  assumes set xs ∩ set ys = {} distinct xs distinct ys
  shows distinct (merge f xs ys)
  using assms by (induction f xs ys rule: merge.induct) (auto simp: set-merge)

lemma sorted-merge:
  assumes P = (λx y. f x ≤ f y)
  shows sorted-wrt P (merge f xs ys) ⟷ sorted-wrt P xs ∧ sorted-wrt P ys
  using assms by (induction f xs ys rule: merge.induct) (auto simp: set-merge)

declare split-at-take-drop-conv [simp]

```

```

function (sequential) mergesort-tm :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list tm
where
  mergesort-tm f [] =1 return []
| mergesort-tm f [x] =1 return [x]
| mergesort-tm f xs =1 (
  do {
    n <- length-tm xs;
    (xsl, xsr) <- split-at-tm (n div 2) xs;
    l <- mergesort-tm f xsl;
    r <- mergesort-tm f xsr;
    merge-tm f l r
  }
)
by pat-completeness auto
termination mergesort-tm
by (relation Wellfounded.measure (λ(-, xs). length xs))
  (auto simp add: length-eq-val-length-tm split-at-eq-val-split-at-tm)

```

```

fun mergesort :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list where
  mergesort f [] = []
| mergesort f [x] = [x]
| mergesort f xs = (
  let n = length xs div 2 in
  let (l, r) = split-at n xs in
  merge f (mergesort f l) (mergesort f r)
)

```

```

declare split-at-take-drop-conv [simp del]

```

```

lemma mergesort-eq-val-mergesort-tm:
  val (mergesort-tm f xs) = mergesort f xs
by (induction f xs rule: mergesort.induct)
  (auto simp add: length-eq-val-length-tm split-at-eq-val-split-at-tm merge-eq-val-merge-tm
split: prod.split)

```

```

lemma sorted-wrt-mergesort:
  sorted-wrt (λx y. f x ≤ f y) (mergesort f xs)
by (induction f xs rule: mergesort.induct) (auto simp: split-at-take-drop-conv
sorted-merge)

```

```

lemma set-mergesort:
  set (mergesort f xs) = set xs
by (induction f xs rule: mergesort.induct)
  (simp-all add: set-merge split-at-take-drop-conv, metis list.simps(15) set-take-drop)

```

```

lemma length-mergesort:
  length (mergesort f xs) = length xs
by (induction f xs rule: mergesort.induct) (auto simp: length-merge split-at-take-drop-conv)

```

**lemma** *distinct-mergesort*:  
*distinct xs  $\implies$  distinct (mergesort f xs)*  
**proof** (*induction f xs rule: mergesort.induct*)  
**case** ( $\exists f x y xs$ )  
**let**  $?xs' = x \# y \# xs$   
**obtain**  $l r$  **where** *lr-def*:  $(l, r) = \text{split-at } (\text{length } ?xs' \text{ div } 2) ?xs'$   
**by** (*metis surj-pair*)  
**have** *distinct l distinct r*  
**using**  $\exists$ .prems *split-at-take-drop-conv distinct-take distinct-drop lr-def* **by** (*metis prod.sel*)  
**hence** *distinct (mergesort f l) distinct (mergesort f r)*  
**using**  $\exists$ .IH *lr-def* **by** *auto*  
**moreover** **have** *set l  $\cap$  set r = {}*  
**using**  $\exists$ .prems *split-at-take-drop-conv lr-def* **by** (*metis append-take-drop-id distinct-append prod.sel*)  
**ultimately show** *?case*  
**using** *lr-def* **by** (*auto simp: distinct-merge set-mergesort split: prod.splits*)  
**qed** *auto*

**lemmas** *mergesort = sorted-wrt-mergesort set-mergesort length-mergesort distinct-mergesort*

**lemma** *sorted-fst-take-less-hd-drop*:  
**assumes** *sorted-fst ps n < length ps*  
**shows**  $\forall p \in \text{set } (\text{take } n \text{ ps}). \text{fst } p \leq \text{fst } (\text{hd } (\text{drop } n \text{ ps}))$   
**using** *assms sorted-wrt-take-less-hd-drop*[of  $\lambda p_0 p_1. \text{fst } p_0 \leq \text{fst } p_1$ ] *sorted-fst-def*  
**by** *fastforce*

**lemma** *sorted-fst-hd-drop-less-drop*:  
**assumes** *sorted-fst ps*  
**shows**  $\forall p \in \text{set } (\text{drop } n \text{ ps}). \text{fst } (\text{hd } (\text{drop } n \text{ ps})) \leq \text{fst } p$   
**using** *assms sorted-wrt-hd-drop-less-drop*[of  $\lambda p_0 p_1. \text{fst } p_0 \leq \text{fst } p_1$ ] *sorted-fst-def*  
**by** *fastforce*

## 1.2.2 Time Complexity Proof

**lemma** *time-merge-tm*:  
*time (merge-tm f xs ys)  $\leq$  length xs + length ys + 1*  
**by** (*induction f xs ys rule: merge-tm.induct*) (*auto simp: time-simps*)

**function** *mergesort-recurrence* :: *nat  $\Rightarrow$  real* **where**  
*mergesort-recurrence 0 = 1*  
| *mergesort-recurrence 1 = 1*  
|  $2 \leq n \implies \text{mergesort-recurrence } n = 4 + 3 * n + \text{mergesort-recurrence } (\text{nat } \lceil \text{real } n / 2 \rceil) +$   
*mergesort-recurrence (nat  $\lceil \text{real } n / 2 \rceil)$*   
**by** *force simp-all*  
**termination** **by** *akra-bazzi-termination simp-all*

**lemma** *mergesort-recurrence-nonneg*[*simp*]:

```

0 ≤ mergesort-recurrence n
by (induction n rule: mergesort-recurrence.induct) (auto simp del: One-nat-def)

lemma time-mergesort-conv-mergesort-recurrence:
  time (mergesort-tm f xs) ≤ mergesort-recurrence (length xs)
proof (induction f xs rule: mergesort-tm.induct)
  case (1 f)
  thus ?case by (auto simp: time-simps)
next
  case (2 f x)
  thus ?case using mergesort-recurrence.simps(2) by (auto simp: time-simps)
next
  case (3 f x y xs')

  define xs where xs = x # y # xs'
  define n where n = length xs
  obtain l r where lr-def: (l, r) = split-at (n div 2) xs
    using prod.collapse by blast
  define l' where l' = mergesort f l
  define r' where r' = mergesort f r
  note defs = xs-def n-def lr-def l'-def r'-def

  have IHL: time (mergesort-tm f l) ≤ mergesort-recurrence (length l)
    using defs 3.IH(1) by (auto simp: length-eq-val-length-tm split-at-eq-val-split-at-tm)
  have IHR: time (mergesort-tm f r) ≤ mergesort-recurrence (length r)
    using defs 3.IH(2) by (auto simp: length-eq-val-length-tm split-at-eq-val-split-at-tm)

  have *: length l = n div 2 length r = n - n div 2
    using defs by (auto simp: split-at-take-drop-conv)
  hence (nat ⌊real n / 2⌋) = length l (nat ⌈real n / 2⌉) = length r
    by linarith+
  hence IH: time (mergesort-tm f l) ≤ mergesort-recurrence (nat ⌊real n / 2⌋)
    time (mergesort-tm f r) ≤ mergesort-recurrence (nat ⌈real n / 2⌉)
    using IHL IHR by simp-all

  have n = length l + length r
    using * by linarith
  hence time (merge-tm f l' r') ≤ n + 1
    using time-merge-tm defs by (metis length-mergesort)

  have time (mergesort-tm f xs) = 1 + time (length-tm xs) + time (split-at-tm (n
div 2) xs) +
    time (mergesort-tm f l) + time (mergesort-tm f r) + time (merge-tm f l'
r')
    using defs by (auto simp add: time-simps length-eq-val-length-tm merge-
sort-eq-val-mergesort-tm
      split-at-eq-val-split-at-tm
      split: prod.split)
  also have ... ≤ 4 + 3 * n + time (mergesort-tm f l) + time (mergesort-tm f r)

```

```

using time-length-tm[of xs] time-split-at-tm[of n div 2 xs] n-def <time (merge-tm
f l' r') ≤ n + 1> by simp
also have ... ≤ 4 + 3 * n + mergesort-recurrence (nat [real n / 2]) + merge-
sort-recurrence (nat [real n / 2])
using IH by simp
also have ... = mergesort-recurrence n
using defs by simp
finally show ?case
using defs by simp
qed

```

**theorem** mergesort-recurrence:  
 $\text{mergesort-recurrence} \in \Theta(\lambda n. n * \ln n)$   
**by** (master-theorem) auto

**theorem** time-mergesort-tm-bigo:  
 $(\lambda xs. \text{time} (\text{mergesort-tm } f \text{ } xs)) \in O[\text{length going-to at-top}]((\lambda n. n * \ln n) \circ \text{length})$   
**proof** –  
**have** 0:  $\bigwedge xs. \text{time} (\text{mergesort-tm } f \text{ } xs) \leq (\text{mergesort-recurrence} \circ \text{length}) \text{ } xs$   
**unfolding** comp-def **using** time-mergesort-conv-mergesort-recurrence **by** blast  
**show** ?thesis  
**using** bigo-measure-trans[OF 0] **by** (simp add: bighetaD1 mergesort-recurrence)  
**qed**

### 1.2.3 Code Export

**lemma** merge-xs-Nil[simp]:  
 $\text{merge } f \text{ } xs \ [] = xs$   
**by** (cases xs) auto

**fun** merge-it' :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list ⇒ 'b list ⇒ 'b list **where**  
 $\text{merge-it}' f \text{ } acc \ [] \ [] = \text{rev } acc$   
 $| \text{merge-it}' f \text{ } acc \ (x\#xs) \ [] = \text{merge-it}' f \text{ } (x\#acc) \ xs \ []$   
 $| \text{merge-it}' f \text{ } acc \ [] \ (y\#ys) = \text{merge-it}' f \text{ } (y\#acc) \ ys \ []$   
 $| \text{merge-it}' f \text{ } acc \ (x\#xs) \ (y\#ys) = ($   
 $\quad \text{if } f \text{ } x \leq f \text{ } y \text{ then}$   
 $\quad \text{merge-it}' f \text{ } (x\#acc) \ xs \ (y\#ys)$   
 $\quad \text{else}$   
 $\quad \text{merge-it}' f \text{ } (y\#acc) \ (x\#xs) \ ys$   
 $)$

**definition** merge-it :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list ⇒ 'b list **where**  
 $\text{merge-it } f \text{ } xs \ ys = \text{merge-it}' f \ [] \ xs \ ys$

**lemma** merge-conv-merge-it':  
 $\text{rev } acc \ @ \ \text{merge } f \text{ } xs \ ys = \text{merge-it}' f \text{ } acc \ xs \ ys$   
**by** (induction f acc xs ys rule: merge-it'.induct) auto

**lemma** merge-conv-merge-it[code-unfold]:

*merge f xs ys = merge-it f xs ys*  
**unfolding** *merge-it-def* **using** *merge-conv-merge-it' rev.simps(1) append-Nil* **by**  
*metis*

### 1.3 Minimal Distance

**definition** *sparse* :: *real*  $\Rightarrow$  *point set*  $\Rightarrow$  *bool* **where**  
*sparse*  $\delta$  *ps*  $\longleftrightarrow (\forall p_0 \in ps. \forall p_1 \in ps. p_0 \neq p_1 \longrightarrow \delta \leq \text{dist } p_0 \ p_1)$

**lemma** *sparse-identity*:  
**assumes** *sparse*  $\delta$  (*set ps*)  $\forall p \in \text{set } ps. \delta \leq \text{dist } p_0 \ p$   
**shows** *sparse*  $\delta$  (*set* ( $p_0 \# ps$ ))  
**using** *assms* **by** (*simp add: dist-commute sparse-def*)

**lemma** *sparse-update*:  
**assumes** *sparse*  $\delta$  (*set ps*)  
**assumes**  $\text{dist } p_0 \ p_1 \leq \delta \ \forall p \in \text{set } ps. \text{dist } p_0 \ p_1 \leq \text{dist } p_0 \ p$   
**shows** *sparse* ( $\text{dist } p_0 \ p_1$ ) (*set* ( $p_0 \# ps$ ))  
**using** *assms* **by** (*auto simp: dist-commute sparse-def, force+*)

**lemma** *sparse-mono*:  
*sparse*  $\Delta \ P \Longrightarrow \delta \leq \Delta \Longrightarrow \text{sparse } \delta \ P$   
**unfolding** *sparse-def* **by** *fastforce*

### 1.4 Distance

**lemma** *dist-transform*:  
**fixes**  $p :: \text{point}$  **and**  $\delta :: \text{real}$  **and**  $l :: \text{int}$   
**shows**  $\text{dist } p \ (l, \text{snd } p) < \delta \longleftrightarrow l - \delta < \text{fst } p \wedge \text{fst } p < l + \delta$   
**proof** –  
**have**  $\text{dist } p \ (l, \text{snd } p) = \text{sqrt } ((\text{real-of-int } (\text{fst } p) - l)^2)$   
**by** (*auto simp add: dist-prod-def dist-real-def prod.case-eq-if*)  
**thus** *?thesis*  
**by** *auto*  
**qed**

**fun** *dist-code* :: *point*  $\Rightarrow$  *point*  $\Rightarrow$  *int* **where**  
*dist-code*  $p_0 \ p_1 = (\text{fst } p_0 - \text{fst } p_1)^2 + (\text{snd } p_0 - \text{snd } p_1)^2$

**lemma** *dist-eq-sqrt-dist-code*:  
**fixes**  $p_0 :: \text{point}$   
**shows**  $\text{dist } p_0 \ p_1 = \text{sqrt } (\text{dist-code } p_0 \ p_1)$   
**by** (*auto simp: dist-prod-def dist-real-def split: prod.splits*)

**lemma** *dist-eq-dist-code-lt*:  
**fixes**  $p_0 :: \text{point}$   
**shows**  $\text{dist } p_0 \ p_1 < \text{dist } p_2 \ p_3 \longleftrightarrow \text{dist-code } p_0 \ p_1 < \text{dist-code } p_2 \ p_3$   
**using** *dist-eq-sqrt-dist-code real-sqrt-less-iff* **by** *presburger*

**lemma** *dist-eq-dist-code-le*:

```

fixes  $p_0 :: \text{point}$ 
shows  $\text{dist } p_0 \ p_1 \leq \text{dist } p_2 \ p_3 \iff \text{dist-code } p_0 \ p_1 \leq \text{dist-code } p_2 \ p_3$ 
using dist-eq-sqrt-dist-code real-sqrt-le-iff by presburger

```

```

lemma dist-eq-dist-code-abs-lt:
  fixes  $p_0 :: \text{point}$ 
  shows  $|c| < \text{dist } p_0 \ p_1 \iff c^2 < \text{dist-code } p_0 \ p_1$ 
  using dist-eq-sqrt-dist-code
  by (metis of-int-less-of-int-power-cancel-iff real-sqrt-abs real-sqrt-less-iff)

```

```

lemma dist-eq-dist-code-abs-le:
  fixes  $p_0 :: \text{point}$ 
  shows  $\text{dist } p_0 \ p_1 \leq |c| \iff \text{dist-code } p_0 \ p_1 \leq c^2$ 
  using dist-eq-sqrt-dist-code
  by (metis of-int-power-le-of-int-cancel-iff real-sqrt-abs real-sqrt-le-iff)

```

```

lemma dist-fst-abs:
  fixes  $p :: \text{point}$  and  $l :: \text{int}$ 
  shows  $\text{dist } p \ (l, \text{snd } p) = |\text{fst } p - l|$ 
proof -
  have  $\text{dist } p \ (l, \text{snd } p) = \text{sqrt } ((\text{real-of-int } (\text{fst } p) - l)^2)$ 
    by (simp add: dist-prod-def dist-real-def prod.case-eq-if)
  thus ?thesis
    by simp
qed

```

```

declare dist-code.simps [simp del]

```

## 1.5 Brute Force Closest Pair Algorithm

### 1.5.1 Functional Correctness Proof

```

fun find-closest-bf-tm ::  $\text{point} \Rightarrow \text{point list} \Rightarrow \text{point tm}$  where
  find-closest-bf-tm - [] = 1 return undefined
| find-closest-bf-tm - [p] = 1 return p
| find-closest-bf-tm p (p0 # ps) = 1 (
  do {
    p1 <- find-closest-bf-tm p ps;
    if  $\text{dist } p \ p_0 < \text{dist } p \ p_1$  then
      return p0
    else
      return p1
  }
)

```

```

fun find-closest-bf ::  $\text{point} \Rightarrow \text{point list} \Rightarrow \text{point}$  where
  find-closest-bf - [] = undefined
| find-closest-bf - [p] = p
| find-closest-bf p (p0 # ps) = (
  let p1 = find-closest-bf p ps in

```

```

    if dist p p0 < dist p p1 then
      p0
    else
      p1
  )

```

**lemma** *find-closest-bf-eq-val-find-closest-bf-tm*:  
*val* (*find-closest-bf-tm* p ps) = *find-closest-bf* p ps  
**by** (*induction* p ps *rule*: *find-closest-bf.induct*) (*auto simp*: *Let-def*)

**lemma** *find-closest-bf-set*:  
 $0 < \text{length } ps \implies \text{find-closest-bf } p \ ps \in \text{set } ps$   
**by** (*induction* p ps *rule*: *find-closest-bf.induct*)  
(*auto simp*: *Let-def split*: *prod.splits if-splits*)

**lemma** *find-closest-bf-dist*:  
 $\forall q \in \text{set } ps. \text{dist } p \ (\text{find-closest-bf } p \ ps) \leq \text{dist } p \ q$   
**by** (*induction* p ps *rule*: *find-closest-bf.induct*)  
(*auto split*: *prod.splits*)

**fun** *closest-pair-bf-tm* :: *point list*  $\Rightarrow$  (*point*  $\times$  *point*) *tm* **where**  
*closest-pair-bf-tm* [] = 1 *return undefined*  
| *closest-pair-bf-tm* [-] = 1 *return undefined*  
| *closest-pair-bf-tm* [p0, p1] = 1 *return* (p0, p1)  
| *closest-pair-bf-tm* (p0 # ps) = 1 (  
  do {  
    (c0::point, c1::point) <- *closest-pair-bf-tm* ps;  
    p1 <- *find-closest-bf-tm* p0 ps;  
    if dist c0 c1  $\leq$  dist p0 p1 then  
      *return* (c0, c1)  
    else  
      *return* (p0, p1)  
  }  
)

**fun** *closest-pair-bf* :: *point list*  $\Rightarrow$  (*point* \* *point*) **where**  
*closest-pair-bf* [] = *undefined*  
| *closest-pair-bf* [-] = *undefined*  
| *closest-pair-bf* [p0, p1] = (p0, p1)  
| *closest-pair-bf* (p0 # ps) = (  
  let (c0, c1) = *closest-pair-bf* ps in  
  let p1 = *find-closest-bf* p0 ps in  
  if dist c0 c1  $\leq$  dist p0 p1 then  
    (c0, c1)  
  else  
    (p0, p1)  
)

**lemma** *closest-pair-bf-eq-val-closest-pair-bf-tm*:

$val$  (*closest-pair-bf-tm ps*) = *closest-pair-bf ps*  
**by** (*induction ps rule: closest-pair-bf.induct*)  
*(auto simp: Let-def find-closest-bf-eq-val-find-closest-bf-tm split: prod.split)*

**lemma** *closest-pair-bf-c0*:  
 $1 < length\ ps \implies (c_0, c_1) = closest-pair-bf\ ps \implies c_0 \in set\ ps$   
**by** (*induction ps arbitrary: c\_0 c\_1 rule: closest-pair-bf.induct*)  
*(auto simp: Let-def find-closest-bf-set split: if-splits prod.splits)*

**lemma** *closest-pair-bf-c1*:  
 $1 < length\ ps \implies (c_0, c_1) = closest-pair-bf\ ps \implies c_1 \in set\ ps$   
**proof** (*induction ps arbitrary: c\_0 c\_1 rule: closest-pair-bf.induct*)  
**case** ( $\lambda p_0 p_2 p_3 ps$ )  
**let**  $?ps = p_2 \# p_3 \# ps$   
**obtain**  $c_0 c_1$  **where**  $c_0-def: (c_0, c_1) = closest-pair-bf\ ?ps$   
**using** *prod.collapse by blast*  
**define**  $p_1$  **where**  $p_1-def: p_1 = find-closest-bf\ p_0\ ?ps$   
**note**  $defs = c_0-def\ p_1-def$   
**have**  $c_1 \in set\ ?ps$   
**using**  $\lambda.IH\ defs$  **by** *simp*  
**moreover** **have**  $p_1 \in set\ ?ps$   
**using** *find-closest-bf-set defs by blast*  
**ultimately show**  $?case$   
**using**  $\lambda.prem\ s(2)\ defs$  **by** (*auto simp: Let-def split: prod.splits if-splits*)  
**qed** *auto*

**lemma** *closest-pair-bf-c0-ne-c1*:  
 $1 < length\ ps \implies distinct\ ps \implies (c_0, c_1) = closest-pair-bf\ ps \implies c_0 \neq c_1$   
**proof** (*induction ps arbitrary: c\_0 c\_1 rule: closest-pair-bf.induct*)  
**case** ( $\lambda p_0 p_2 p_3 ps$ )  
**let**  $?ps = p_2 \# p_3 \# ps$   
**obtain**  $c_0 c_1$  **where**  $c_0-def: (c_0, c_1) = closest-pair-bf\ ?ps$   
**using** *prod.collapse by blast*  
**define**  $p_1$  **where**  $p_1-def: p_1 = find-closest-bf\ p_0\ ?ps$   
**note**  $defs = c_0-def\ p_1-def$   
**have**  $c_0 \neq c_1$   
**using**  $\lambda.IH\ \lambda.prem\ s(2)\ defs$  **by** *simp*  
**moreover** **have**  $p_0 \neq p_1$   
**using** *find-closest-bf-set \lambda.prem\ s(2)\ defs*  
**by** (*metis distinct.simps(2) length-pos-if-in-set list.set-intros(1)*)  
**ultimately show**  $?case$   
**using**  $\lambda.prem\ s(3)\ defs$  **by** (*auto simp: Let-def split: prod.splits if-splits*)  
**qed** *auto*

**lemmas** *closest-pair-bf-c0-c1 = closest-pair-bf-c0 closest-pair-bf-c1 closest-pair-bf-c0-ne-c1*

**lemma** *closest-pair-bf-dist*:  
**assumes**  $1 < length\ ps\ (c_0, c_1) = closest-pair-bf\ ps$   
**shows** *sparse (dist c\_0 c\_1) (set ps)*

```

using assms
proof (induction ps arbitrary: c0 c1 rule: closest-pair-bf.induct)
  case ( $\lambda p_0 p_2 p_3 ps$ )
  let  $?ps = p_2 \# p_3 \# ps$ 
  obtain  $c_0 c_1$  where  $c_0\text{-def}: (c_0, c_1) = \text{closest-pair-bf } ?ps$ 
    using prod.collapse by blast
  define  $p_1$  where  $p_1\text{-def}: p_1 = \text{find-closest-bf } p_0 \ ?ps$ 
  note  $\text{defs} = c_0\text{-def } p_1\text{-def}$ 
  hence  $IH: \text{sparse } (dist \ c_0 \ c_1) \ (set \ ?ps)$ 
    using  $\lambda c_0\text{-def}$  by simp
  have  $*$ :  $\forall p \in set \ ?ps. (dist \ p_0 \ p_1) \leq dist \ p_0 \ p$ 
    using find-closest-bf-dist  $\text{defs}$  by blast
  show  $?case$ 
  proof (cases dist c0 c1 ≤ dist p0 p1)
    case True
    hence  $\forall p \in set \ ?ps. dist \ c_0 \ c_1 \leq dist \ p_0 \ p$ 
      using  $*$  by auto
    hence  $\text{sparse } (dist \ c_0 \ c_1) \ (set \ (p_0 \# \ ?ps))$ 
      using sparse-identity IH by blast
    thus  $?thesis$ 
      using True  $\lambda.prem \ \text{defs}$  by (auto split: prod.splits)
    next
    case False
    hence  $\text{sparse } (dist \ p_0 \ p_1) \ (set \ (p_0 \# \ ?ps))$ 
      using sparse-update[of dist c0 c1 ?ps p0 p1] IH * defs by argo
    thus  $?thesis$ 
      using False  $\lambda.prem \ \text{defs}$  by (auto split: prod.splits)
  qed
qed (auto simp: dist-commute sparse-def)

```

## 1.5.2 Time Complexity Proof

**lemma** *time-find-closest-bf-tm:*

*time (find-closest-bf-tm p ps) ≤ length ps + 1*

**by** (*induction p ps rule: find-closest-bf-tm.induct*) (*auto simp: time-simps*)

**lemma** *time-closest-pair-bf-tm:*

*time (closest-pair-bf-tm ps) ≤ length ps \* length ps + 1*

**proof** (*induction ps rule: closest-pair-bf-tm.induct*)

**case** ( $\lambda p_0 p_2 p_3 ps$ )

**let**  $?ps = p_2 \# p_3 \# ps$

**have**  $\text{time } (\text{closest-pair-bf-tm } (p_0 \# \ ?ps)) = 1 + \text{time } (\text{find-closest-bf-tm } p_0 \ ?ps)$   
 $+ \text{time } (\text{closest-pair-bf-tm } ?ps)$

**by** (*auto simp: time-simps split: prod.split*)

**also have**  $\dots \leq 2 + \text{length } ?ps + \text{time } (\text{closest-pair-bf-tm } ?ps)$

**using** *time-find-closest-bf-tm*[*of p<sub>0</sub> ?ps*] **by** *simp*

**also have**  $\dots \leq 2 + \text{length } ?ps + \text{length } ?ps * \text{length } ?ps + 1$

**using**  $\lambda.IH$  **by** *simp*

**also have**  $\dots \leq \text{length } (p_0 \# \ ?ps) * \text{length } (p_0 \# \ ?ps) + 1$

```

    by auto
  finally show ?case
    by blast
qed (auto simp: time-simps)

```

### 1.5.3 Code Export

```

fun find-closest-bf-code :: point  $\Rightarrow$  point list  $\Rightarrow$  (int * point) where
  find-closest-bf-code p [] = undefined
| find-closest-bf-code p [p0] = (dist-code p p0, p0)
| find-closest-bf-code p (p0 # ps) = (
  let (δ1, p1) = find-closest-bf-code p ps in
  let δ0 = dist-code p p0 in
  if δ0 < δ1 then
    (δ0, p0)
  else
    (δ1, p1)
)

```

**lemma** find-closest-bf-code-dist-eq:

```

0 < length ps  $\implies$  (δ, c) = find-closest-bf-code p ps  $\implies$  δ = dist-code p c
by (induction p ps rule: find-closest-bf-code.induct)
  (auto simp: Let-def split: prod.splits if-splits)

```

**lemma** find-closest-bf-code-eq:

```

0 < length ps  $\implies$  c = find-closest-bf p ps  $\implies$  (δ', c') = find-closest-bf-code p ps
 $\implies$  c = c'

```

**proof** (induction p ps arbitrary: c δ' c' rule: find-closest-bf.induct)

**case** (3 p p<sub>0</sub> p<sub>2</sub> ps)

**define** δ<sub>0</sub> δ<sub>0</sub>' **where** δ<sub>0</sub>-def: δ<sub>0</sub> = dist p p<sub>0</sub> δ<sub>0</sub>' = dist-code p p<sub>0</sub>

**obtain** δ<sub>1</sub> p<sub>1</sub> δ<sub>1</sub>' p<sub>1</sub>' **where** δ<sub>1</sub>-def: δ<sub>1</sub> = dist p p<sub>1</sub> p<sub>1</sub> = find-closest-bf p (p<sub>2</sub> # ps)

(δ<sub>1</sub>', p<sub>1</sub>') = find-closest-bf-code p (p<sub>2</sub> # ps)

**using** prod.collapse **by** blast+

**note** defs = δ<sub>0</sub>-def δ<sub>1</sub>-def

**have** \*: p<sub>1</sub> = p<sub>1</sub>'

**using** 3.IH defs **by** simp

**hence** δ<sub>0</sub> < δ<sub>1</sub>  $\longleftrightarrow$  δ<sub>0</sub>' < δ<sub>1</sub>'

**using** find-closest-bf-code-dist-eq[of p<sub>2</sub> # ps δ<sub>1</sub>' p<sub>1</sub>' p]  
dist-eq-dist-code-lt defs

**by** simp

**thus** ?case

**using** 3.prem(2,3) \* defs **by** (auto split: prod.splits)

**qed** auto

**declare** find-closest-bf-code.simps [simp del]

```

fun closest-pair-bf-code :: point list  $\Rightarrow$  (int * point * point) where
  closest-pair-bf-code [] = undefined

```

```

| closest-pair-bf-code [p0] = undefined
| closest-pair-bf-code [p0, p1] = (dist-code p0 p1, p0, p1)
| closest-pair-bf-code (p0 # ps) = (
  let (δc, c0, c1) = closest-pair-bf-code ps in
  let (δp, p1) = find-closest-bf-code p0 ps in
  if δc ≤ δp then
    (δc, c0, c1)
  else
    (δp, p0, p1)
)

```

**lemma** *closest-pair-bf-code-dist-eq*:

$1 < \text{length } ps \implies (\delta, c_0, c_1) = \text{closest-pair-bf-code } ps \implies \delta = \text{dist-code } c_0 \ c_1$

**proof** (*induction ps arbitrary: δ c<sub>0</sub> c<sub>1</sub> rule: closest-pair-bf-code.induct*)

**case** ( $\lambda p_0 \ p_2 \ p_3 \ ps$ )

**let**  $?ps = p_2 \# p_3 \# ps$

**obtain**  $\delta_c \ c_0 \ c_1$  **where**  $\delta_c\text{-def}: (\delta_c, c_0, c_1) = \text{closest-pair-bf-code } ?ps$   
**by** (*metis prod-cases3*)

**obtain**  $\delta_p \ p_1$  **where**  $\delta_p\text{-def}: (\delta_p, p_1) = \text{find-closest-bf-code } p_0 \ ?ps$   
**using** *prod.collapse* **by** *blast*

**note**  $\text{defs} = \delta_c\text{-def } \delta_p\text{-def}$

**have**  $\delta_c = \text{dist-code } c_0 \ c_1$

**using** *4.IH* **defs** **by** *simp*

**moreover** **have**  $\delta_p = \text{dist-code } p_0 \ p_1$

**using** *find-closest-bf-code-dist-eq* **defs** **by** *blast*

**ultimately** **show** *?case*

**using** *4.prem2* **defs** **by** (*auto split: prod.splits if-splits*)

**qed** *auto*

**lemma** *closest-pair-bf-code-eq*:

**assumes**  $1 < \text{length } ps$

**assumes**  $(c_0, c_1) = \text{closest-pair-bf } ps \ (\delta', c_0', c_1') = \text{closest-pair-bf-code } ps$

**shows**  $c_0 = c_0' \wedge c_1 = c_1'$

**using** *assms*

**proof** (*induction ps arbitrary: c<sub>0</sub> c<sub>1</sub> δ' c<sub>0</sub>' c<sub>1</sub>' rule: closest-pair-bf-code.induct*)

**case** ( $\lambda p_0 \ p_2 \ p_3 \ ps$ )

**let**  $?ps = p_2 \# p_3 \# ps$

**obtain**  $c_0 \ c_1 \ \delta_c' \ c_0' \ c_1'$  **where**  $\delta_c\text{-def}: (c_0, c_1) = \text{closest-pair-bf } ?ps$   
 $(\delta_c', c_0', c_1') = \text{closest-pair-bf-code } ?ps$

**by** (*metis prod-cases3*)

**obtain**  $p_1 \ \delta_p' \ p_1'$  **where**  $\delta_p\text{-def}: p_1 = \text{find-closest-bf } p_0 \ ?ps$

$(\delta_p', p_1') = \text{find-closest-bf-code } p_0 \ ?ps$

**using** *prod.collapse* **by** *blast*

**note**  $\text{defs} = \delta_c\text{-def } \delta_p\text{-def}$

**have**  $A: c_0 = c_0' \wedge c_1 = c_1'$

**using** *4.IH* **defs** **by** *simp*

**moreover** **have**  $B: p_1 = p_1'$

**using** *find-closest-bf-code-eq* **defs** **by** *blast*

**moreover** **have**  $\delta_c' = \text{dist-code } c_0' \ c_1'$

using defs closest-pair-bf-code-dist-eq[of ?ps] by simp  
 moreover have  $\delta_p' = \text{dist-code } p_0 \ p_1'$   
 using defs find-closest-bf-code-dist-eq by blast  
 ultimately have  $\text{dist } c_0 \ c_1 \leq \text{dist } p_0 \ p_1 \iff \delta_c' \leq \delta_p'$   
 by (simp add: dist-eq-dist-code-le)  
 thus ?case  
 using 4.premis(2,3) defs A B by (auto simp: Let-def split: prod.splits)  
 qed auto

## 1.6 Geometry

### 1.6.1 Band Filter

lemma set-band-filter-aux:

fixes  $\delta :: \text{real}$  and  $ps :: \text{point list}$

assumes  $p_0 \in ps_L \ p_1 \in ps_R \ p_0 \neq p_1 \ \text{dist } p_0 \ p_1 < \delta \ \text{set } ps = ps_L \cup ps_R$

assumes  $\forall p \in ps_L. \text{fst } p \leq l \ \forall p \in ps_R. l \leq \text{fst } p$

assumes  $ps' = \text{filter } (\lambda p. l - \delta < \text{fst } p \wedge \text{fst } p < l + \delta) \ ps$

shows  $p_0 \in \text{set } ps' \wedge p_1 \in \text{set } ps'$

proof (rule ccontr)

assume  $\neg (p_0 \in \text{set } ps' \wedge p_1 \in \text{set } ps')$

then consider (A)  $p_0 \notin \text{set } ps' \wedge p_1 \notin \text{set } ps'$

| (B)  $p_0 \in \text{set } ps' \wedge p_1 \notin \text{set } ps'$

| (C)  $p_0 \notin \text{set } ps' \wedge p_1 \in \text{set } ps'$

by blast

thus False

proof cases

case A

hence  $\text{fst } p_0 \leq l - \delta \vee l + \delta \leq \text{fst } p_0 \ \text{fst } p_1 \leq l - \delta \vee l + \delta \leq \text{fst } p_1$

using assms(1,2,5,8) by auto

hence  $\text{fst } p_0 \leq l - \delta \ \vee \ l + \delta \leq \text{fst } p_1$

using assms(1,2,6,7) by force+

hence  $\delta \leq \text{dist } (\text{fst } p_0) \ (\text{fst } p_1)$

using dist-real-def by simp

hence  $\delta \leq \text{dist } p_0 \ p_1$

using dist-fst-le[of  $p_0 \ p_1$ ] by (auto split: prod.splits)

then show ?thesis

using assms(4) by fastforce

next

case B

hence  $\text{fst } p_1 \leq l - \delta \vee l + \delta \leq \text{fst } p_1$

using assms(2,5,8) by auto

hence  $l + \delta \leq \text{fst } p_1$

using assms(2,7) by auto

moreover have  $\text{fst } p_0 \leq l$

using assms(1,6) by simp

ultimately have  $\delta \leq \text{dist } (\text{fst } p_0) \ (\text{fst } p_1)$

using dist-real-def by simp

hence  $\delta \leq \text{dist } p_0 \ p_1$

using dist-fst-le[of  $p_0 \ p_1$ ] less-le-trans by (auto split: prod.splits)

**thus** *?thesis*  
**using** *assms(4)* **by** *simp*  
**next**  
**case** *C*  
**hence**  $\text{fst } p_0 \leq l - \delta \vee l + \delta \leq \text{fst } p_0$   
**using** *assms(1,2,5,8)* **by** *auto*  
**hence**  $\text{fst } p_0 \leq l - \delta$   
**using** *assms(1,6)* **by** *auto*  
**moreover have**  $l \leq \text{fst } p_1$   
**using** *assms(2,7)* **by** *simp*  
**ultimately have**  $\delta \leq \text{dist } (\text{fst } p_0) (\text{fst } p_1)$   
**using** *dist-real-def* **by** *simp*  
**hence**  $\delta \leq \text{dist } p_0 p_1$   
**using** *dist-fst-le[of p\_0 p\_1] less-le-trans* **by** (*auto split: prod.splits*)  
**thus** *?thesis*  
**using** *assms(4)* **by** *simp*  
**qed**  
**qed**

**lemma** *set-band-filter*:

**fixes**  $\delta :: \text{real}$  **and**  $ps :: \text{point list}$   
**assumes**  $p_0 \in \text{set } ps$   $p_1 \in \text{set } ps$   $p_0 \neq p_1$   $\text{dist } p_0 p_1 < \delta$   $\text{set } ps = ps_L \cup ps_R$   
**assumes** *sparse*  $\delta$   $ps_L$  *sparse*  $\delta$   $ps_R$   
**assumes**  $\forall p \in ps_L. \text{fst } p \leq l \vee p \in ps_R. l \leq \text{fst } p$   
**assumes**  $ps' = \text{filter } (\lambda p. l - \delta < \text{fst } p \wedge \text{fst } p < l + \delta)$   $ps$   
**shows**  $p_0 \in \text{set } ps' \wedge p_1 \in \text{set } ps'$

**proof** –

**have**  $p_0 \notin ps_L \vee p_1 \notin ps_L$   $p_0 \notin ps_R \vee p_1 \notin ps_R$   
**using** *assms(3,4,6,7)* *sparse-def* **by** *force+*  
**then consider** (A)  $p_0 \in ps_L \wedge p_1 \in ps_R$  | (B)  $p_0 \in ps_R \wedge p_1 \in ps_L$   
**using** *assms(1,2,5)* **by** *auto*  
**thus** *?thesis*  
**proof cases**  
**case** *A*  
**thus** *?thesis*  
**using** *set-band-filter-aux* *assms(3,4,5,8,9,10)* **by** (*auto split: prod.splits*)

**next**

**case** *B*  
**moreover have**  $\text{dist } p_1 p_0 < \delta$   
**using** *assms(4)* *dist-commute* **by** *metis*  
**ultimately show** *?thesis*  
**using** *set-band-filter-aux* *assms(3)[symmetric]* *assms(5,8,9,10)* **by** (*auto split:*  
*prod.splits*)

**qed**

**qed**

## 1.6.2 2D-Boxes and Points

**lemma** *cbox-2D*:

**fixes**  $x_0 :: \text{real}$  **and**  $y_0 :: \text{real}$   
**shows**  $\text{cbox } (x_0, y_0) (x_1, y_1) = \{ (x, y). x_0 \leq x \wedge x \leq x_1 \wedge y_0 \leq y \wedge y \leq y_1 \}$   
**by** *fastforce*

**lemma** *mem-cbox-2D*:

**fixes**  $x :: \text{real}$  **and**  $y :: \text{real}$   
**shows**  $x_0 \leq x \wedge x \leq x_1 \wedge y_0 \leq y \wedge y \leq y_1 \longleftrightarrow (x, y) \in \text{cbox } (x_0, y_0) (x_1, y_1)$   
**by** *fastforce*

**lemma** *cbox-top-un*:

**fixes**  $x_0 :: \text{real}$  **and**  $y_0 :: \text{real}$   
**assumes**  $y_0 \leq y_1$   $y_1 \leq y_2$   
**shows**  $\text{cbox } (x_0, y_0) (x_1, y_1) \cup \text{cbox } (x_0, y_1) (x_1, y_2) = \text{cbox } (x_0, y_0) (x_1, y_2)$   
**using** *assms* **by** *auto*

**lemma** *cbox-right-un*:

**fixes**  $x_0 :: \text{real}$  **and**  $y_0 :: \text{real}$   
**assumes**  $x_0 \leq x_1$   $x_1 \leq x_2$   
**shows**  $\text{cbox } (x_0, y_0) (x_1, y_1) \cup \text{cbox } (x_1, y_0) (x_2, y_1) = \text{cbox } (x_0, y_0) (x_2, y_1)$   
**using** *assms* **by** *auto*

**lemma** *cbox-max-dist*:

**assumes**  $p_0 = (x, y)$   $p_1 = (x + \delta, y + \delta)$   
**assumes**  $(x_0, y_0) \in \text{cbox } p_0 p_1$   $(x_1, y_1) \in \text{cbox } p_0 p_1$   $0 \leq \delta$   
**shows**  $\text{dist } (x_0, y_0) (x_1, y_1) \leq \text{sqrt } 2 * \delta$

**proof** –

**have**  $X$ :  $\text{dist } x_0 x_1 \leq \delta$   
**using** *assms* *dist-real-def* **by** *auto*  
**have**  $Y$ :  $\text{dist } y_0 y_1 \leq \delta$   
**using** *assms* *dist-real-def* **by** *auto*

**have**  $\text{dist } (x_0, y_0) (x_1, y_1) = \text{sqrt } ((\text{dist } x_0 x_1)^2 + (\text{dist } y_0 y_1)^2)$

**using** *dist-Pair-Pair* **by** *auto*

**also have**  $\dots \leq \text{sqrt } (\delta^2 + (\text{dist } y_0 y_1)^2)$

**using**  $X$  *power-mono* **by** *fastforce*

**also have**  $\dots \leq \text{sqrt } (\delta^2 + \delta^2)$

**using**  $Y$  *power-mono* **by** *fastforce*

**also have**  $\dots = \text{sqrt } 2 * \text{sqrt } (\delta^2)$

**using** *real-sqrt-mult* **by** *simp*

**also have**  $\dots = \text{sqrt } 2 * \delta$

**using** *assms*(5) **by** *simp*

**finally show** *?thesis* .

**qed**

### 1.6.3 Pigeonhole Argument

**lemma** *card-le-1-if-pairwise-eq*:

**assumes**  $\forall x \in S. \forall y \in S. x = y$

**shows**  $\text{card } S \leq 1$

**proof** (*rule ccontr*)  
**assume**  $\neg \text{card } S \leq 1$   
**hence**  $2 \leq \text{card } S$   
**by** *simp*  
**then obtain**  $T$  **where**  $*$ :  $T \subseteq S \wedge \text{card } T = 2$   
**using** *ex-card* **by** *metis*  
**then obtain**  $x y$  **where**  $x \in T \wedge y \in T \wedge x \neq y$   
**by** (*meson card-2-iff'*)  
**then show** *False*  
**using**  $*$  *assms* **by** *blast*  
**qed**

**lemma** *card-Int-if-either-in*:  
**assumes**  $\forall x \in S. \forall y \in S. x = y \vee x \notin T \vee y \notin T$   
**shows**  $\text{card } (S \cap T) \leq 1$   
**proof** (*rule ccontr*)  
**assume**  $\neg (\text{card } (S \cap T) \leq 1)$   
**then obtain**  $x y$  **where**  $*$ :  $x \in S \cap T \wedge y \in S \cap T \wedge x \neq y$   
**by** (*meson card-le-1-if-pairwise-eq*)  
**hence**  $x \in T \wedge y \in T$   
**by** *simp-all*  
**moreover have**  $x \notin T \vee y \notin T$   
**using** *assms*  $*$  **by** *auto*  
**ultimately show** *False*  
**by** *blast*  
**qed**

**lemma** *card-Int-Un-le-Sum-card-Int*:  
**assumes** *finite*  $S$   
**shows**  $\text{card } (A \cap \bigcup S) \leq (\sum B \in S. \text{card } (A \cap B))$   
**using** *assms*  
**proof** (*induction card*  $S$  *arbitrary*:  $S$ )  
**case** (*Suc*  $n$ )  
**then obtain**  $B T$  **where**  $*$ :  $S = \{ B \} \cup T$   $\text{card } T = n$   $B \notin T$   
**by** (*metis card-Suc-eq Suc-eq-plus1 insert-is-Un*)  
**hence**  $\text{card } (A \cap \bigcup S) = \text{card } (A \cap \bigcup (\{ B \} \cup T))$   
**by** *blast*  
**also have**  $\dots \leq \text{card } (A \cap B) + \text{card } (A \cap \bigcup T)$   
**by** (*simp add: card-Un-le inf-sup-distrib1*)  
**also have**  $\dots \leq \text{card } (A \cap B) + (\sum B \in T. \text{card } (A \cap B))$   
**using** *Suc*  $*$  **by** *simp*  
**also have**  $\dots \leq (\sum B \in S. \text{card } (A \cap B))$   
**using** *Suc.prem*s  $*$  **by** *simp*  
**finally show** *?case* .  
**qed** *simp*

**lemma** *pigeonhole*:  
**assumes** *finite*  $T$   $S \subseteq \bigcup T$   $\text{card } T < \text{card } S$   
**shows**  $\exists x \in S. \exists y \in S. \exists X \in T. x \neq y \wedge x \in X \wedge y \in X$

**proof** (*rule ccontr*)  
**assume**  $\neg (\exists x \in S. \exists y \in S. \exists X \in T. x \neq y \wedge x \in X \wedge y \in X)$   
**hence** \*:  $\forall X \in T. \text{card } (S \cap X) \leq 1$   
**using** *card-Int-if-either-in* **by** *metis*  
**have**  $\text{card } T < \text{card } (S \cap \bigcup T)$   
**using** *Int-absorb2* *assms* **by** *fastforce*  
**also have**  $\dots \leq (\sum X \in T. \text{card } (S \cap X))$   
**using** *assms(1)* *card-Int-Un-le-Sum-card-Int* **by** *blast*  
**also have**  $\dots \leq \text{card } T$   
**using** \* *sum-mono* **by** *fastforce*  
**finally show** *False* **by** *simp*  
**qed**

#### 1.6.4 Delta Sparse Points within a Square

**lemma** *max-points-square*:  
**assumes**  $\forall p \in ps. p \in \text{cbox } (x, y) (x + \delta, y + \delta)$  *sparse*  $\delta$   $ps$   $0 \leq \delta$   
**shows**  $\text{card } ps \leq 4$   
**proof** (*cases*  $\delta = 0$ )  
**case** *True*  
**hence**  $\{ (x, y) \} = \text{cbox } (x, y) (x + \delta, y + \delta)$   
**using** *cbox-def* **by** *simp*  
**hence**  $\forall p \in ps. p = (x, y)$   
**using** *assms(1)* **by** *blast*  
**hence**  $\forall p \in ps. \forall q \in ps. p = q$   
**apply** (*auto split: prod.splits*) **by** (*metis of-int-eq-iff*)+  
**thus** *?thesis*  
**using** *card-le-1-if-pairwise-eq* **by** *force*  
**next**  
**case** *False*  
**hence**  $\delta: 0 < \delta$   
**using** *assms(3)* **by** *simp*  
**show** *?thesis*  
**proof** (*rule ccontr*)  
**assume** *A*:  $\neg (\text{card } ps \leq 4)$   
**define** *PS* **where** *PS-def*:  $PS = (\lambda(x, y). (\text{real-of-int } x, \text{real-of-int } y))$  ‘ *ps*  
**have** *inj-on*  $(\lambda(x, y). (\text{real-of-int } x, \text{real-of-int } y))$  *ps*  
**using** *inj-on-def* **by** *fastforce*  
**hence** \*:  $\neg (\text{card } PS \leq 4)$   
**using** *A* *PS-def* **by** (*simp add: card-image*)  
  
**let**  $?x' = x + \delta / 2$   
**let**  $?y' = y + \delta / 2$   
  
**let**  $?ll = \text{cbox } (x, y) (?x', ?y')$   
**let**  $?lu = \text{cbox } (x, ?y') (?x', y + \delta)$   
**let**  $?rl = \text{cbox } (?x', y) (x + \delta, ?y')$   
**let**  $?ru = \text{cbox } (?x', ?y') (x + \delta, y + \delta)$

**let**  $?sq = \{ ?ll, ?lu, ?rl, ?ru \}$   
**have**  $card-le-4$ :  $card\ ?sq \leq 4$   
**by** (*simp add: card-insert-le-m1*)  
  
**have**  $cbox\ (x, y)\ (?x', y + \delta) = ?ll \cup ?lu$   
**using** *cbox-top-un assms(3)* **by** *auto*  
**moreover** **have**  $cbox\ (?x', y)\ (x + \delta, y + \delta) = ?rl \cup ?ru$   
**using** *cbox-top-un assms(3)* **by** *auto*  
**moreover** **have**  $cbox\ (x, y)\ (?x', y + \delta) \cup cbox\ (?x', y)\ (x + \delta, y + \delta) = cbox$   
 $(x, y)\ (x + \delta, y + \delta)$   
**using** *cbox-right-un assms(3)* **by** *simp*  
**ultimately** **have**  $?ll \cup ?lu \cup ?rl \cup ?ru = cbox\ (x, y)\ (x + \delta, y + \delta)$   
**by** *blast*  
  
**hence**  $PS \subseteq \bigcup(?sq)$   
**using** *assms(1) PS-def* **by** (*auto split: prod.splits*)  
**moreover** **have**  $card\ ?sq < card\ PS$   
**using**  $*$  *card-insert-le-m1 card-le-4* **by** *linarith*  
**moreover** **have**  $finite\ ?sq$   
**by** *simp*  
**ultimately** **have**  $\exists p_0 \in PS. \exists p_1 \in PS. \exists S \in ?sq. (p_0 \neq p_1 \wedge p_0 \in S \wedge p_1 \in$   
 $S)$   
**using** *pigeonhole[of ?sq PS]* **by** *blast*  
**then** **obtain**  $S\ p_0\ p_1$  **where**  $\#$ :  $p_0 \in PS\ p_1 \in PS\ S \in ?sq\ p_0 \neq p_1\ p_0 \in S\ p_1$   
 $\in S$   
**by** *blast*  
  
**have**  $D$ :  $0 \leq \delta / 2$   
**using** *assms(3)* **by** *simp*  
**have**  $LL$ :  $\forall p_0 \in ?ll. \forall p_1 \in ?ll. dist\ p_0\ p_1 \leq sqrt\ 2 * (\delta / 2)$   
**using** *cbox-max-dist[of (x, y) x y (?x', ?y')  $\delta / 2$ ]* **by** *auto*  
**have**  $LU$ :  $\forall p_0 \in ?lu. \forall p_1 \in ?lu. dist\ p_0\ p_1 \leq sqrt\ 2 * (\delta / 2)$   
**using** *cbox-max-dist[of (x, ?y') x ?y' (?x', y +  $\delta$ )  $\delta / 2$ ]* **by** *auto*  
**have**  $RL$ :  $\forall p_0 \in ?rl. \forall p_1 \in ?rl. dist\ p_0\ p_1 \leq sqrt\ 2 * (\delta / 2)$   
**using** *cbox-max-dist[of (?x', y) ?x' y (x +  $\delta$ , ?y')  $\delta / 2$ ]* **by** *auto*  
**have**  $RU$ :  $\forall p_0 \in ?ru. \forall p_1 \in ?ru. dist\ p_0\ p_1 \leq sqrt\ 2 * (\delta / 2)$   
**using** *cbox-max-dist[of (?x', ?y') ?x' ?y' (x +  $\delta$ , y +  $\delta$ )  $\delta / 2$ ]* **by** *auto*  
  
**have**  $\forall p_0 \in S. \forall p_1 \in S. dist\ p_0\ p_1 \leq sqrt\ 2 * (\delta / 2)$   
**using**  $\#$  *LL LU RL RU* **by** *blast*  
**hence**  $dist\ p_0\ p_1 \leq (sqrt\ 2 / 2) * \delta$   
**using**  $\#$  **by** *simp*  
**moreover** **have**  $(sqrt\ 2 / 2) * \delta < \delta$   
**using** *sqrt2-less-2  $\delta$*  **by** *simp*  
**ultimately** **have**  $dist\ p_0\ p_1 < \delta$   
**by** *simp*  
**moreover** **have**  $\delta \leq dist\ p_0\ p_1$   
**using** *assms(2) # sparse-def PS-def* **by** *auto*

```

    ultimately show False
      by simp
  qed
qed
end

```

## 2 Closest Pair Algorithm

```

theory Closest-Pair
  imports Common
begin

```

Formalization of a slightly optimized divide-and-conquer algorithm solving the Closest Pair Problem based on the presentation of Cormen *et al.* [1].

### 2.1 Functional Correctness Proof

#### 2.1.1 Combine Step

```

fun find-closest-tm :: point  $\Rightarrow$  real  $\Rightarrow$  point list  $\Rightarrow$  point tm where
  find-closest-tm - - [] =1 return undefined
| find-closest-tm - - [p] =1 return p
| find-closest-tm p  $\delta$  ( $p_0 \# ps$ ) =1 (
  if  $\delta \leq \text{snd } p_0 - \text{snd } p$  then
    return  $p_0$ 
  else
    do {
       $p_1 \leftarrow \text{find-closest-tm } p (\text{min } \delta (\text{dist } p \ p_0)) \ ps;$ 
      if  $\text{dist } p \ p_0 \leq \text{dist } p \ p_1$  then
        return  $p_0$ 
      else
        return  $p_1$ 
    }
  )

```

```

fun find-closest :: point  $\Rightarrow$  real  $\Rightarrow$  point list  $\Rightarrow$  point where
  find-closest - - [] = undefined
| find-closest - - [p] = p
| find-closest p  $\delta$  ( $p_0 \# ps$ ) = (
  if  $\delta \leq \text{snd } p_0 - \text{snd } p$  then
     $p_0$ 
  else
    let  $p_1 = \text{find-closest } p (\text{min } \delta (\text{dist } p \ p_0)) \ ps$  in
    if  $\text{dist } p \ p_0 \leq \text{dist } p \ p_1$  then
       $p_0$ 
    else
       $p_1$ 

```

)

**lemma** *find-closest-eq-val-find-closest-tm*:

*val* (*find-closest-tm*  $p$   $\delta$   $ps$ ) = *find-closest*  $p$   $\delta$   $ps$

**by** (*induction*  $p$   $\delta$   $ps$  *rule*: *find-closest.induct*) (*auto simp*: *Let-def*)

**lemma** *find-closest-set*:

$0 < \text{length } ps \implies \text{find-closest } p \ \delta \ ps \in \text{set } ps$

**by** (*induction*  $p$   $\delta$   $ps$  *rule*: *find-closest.induct*)

(*auto simp*: *Let-def*)

**lemma** *find-closest-dist*:

**assumes** *sorted-snd* ( $p \# ps$ )  $\exists q \in \text{set } ps. \text{dist } p \ q < \delta$

**shows**  $\forall q \in \text{set } ps. \text{dist } p \ (\text{find-closest } p \ \delta \ ps) \leq \text{dist } p \ q$

**using** *assms*

**proof** (*induction*  $p$   $\delta$   $ps$  *rule*: *find-closest.induct*)

**case** ( $\exists p \ \delta \ p_0 \ p_2 \ ps$ )

**let**  $?ps = p_0 \# p_2 \# ps$

**define**  $p_1$  **where**  $p_1$ -*def*:  $p_1 = \text{find-closest } p \ (\min \ \delta \ (\text{dist } p \ p_0)) \ (p_2 \# ps)$

**have**  $A: \neg \delta \leq \text{snd } p_0 - \text{snd } p$

**proof** (*rule ccontr*)

**assume**  $B: \neg \neg \delta \leq \text{snd } p_0 - \text{snd } p$

**have**  $\forall q \in \text{set } ?ps. \text{snd } p \leq \text{snd } q$

**using** *sorted-snd-def*  $\exists$ .*prems*(1) **by** *simp*

**moreover have**  $\forall q \in \text{set } ?ps. \delta \leq \text{snd } q - \text{snd } p$

**using** *sorted-snd-def*  $\exists$ .*prems*(1)  $B$  **by** *auto*

**ultimately have**  $\forall q \in \text{set } ?ps. \delta \leq \text{dist } (\text{snd } p) \ (\text{snd } q)$

**using** *dist-real-def* **by** *simp*

**hence**  $\forall q \in \text{set } ?ps. \delta \leq \text{dist } p \ q$

**using** *dist-snd-le* *order-trans*

**apply** (*auto split*: *prod.splits*) **by** *fastforce+*

**thus** *False*

**using**  $\exists$ .*prems*(2) **by** *fastforce*

**qed**

**show** *?case*

**proof** *cases*

**assume**  $\exists q \in \text{set } (p_2 \# ps). \text{dist } p \ q < \min \ \delta \ (\text{dist } p \ p_0)$

**hence**  $\forall q \in \text{set } (p_2 \# ps). \text{dist } p \ p_1 \leq \text{dist } p \ q$

**using**  $\exists$ .*IH*  $\exists$ .*prems*(1)  $A$   $p_1$ -*def* *sorted-snd-def* **by** *simp*

**thus** *?thesis*

**using**  $p_1$ -*def*  $A$  **by** (*auto split*: *prod.splits*)

**next**

**assume**  $B: \neg (\exists q \in \text{set } (p_2 \# ps). \text{dist } p \ q < \min \ \delta \ (\text{dist } p \ p_0))$

**hence**  $\text{dist } p \ p_0 < \delta$

**using**  $\exists$ .*prems*(2)  $p_1$ -*def* **by** *auto*

**hence**  $C: \forall q \in \text{set } ?ps. \text{dist } p \ p_0 \leq \text{dist } p \ q$

**using**  $p_1$ -*def*  $B$  **by** *auto*

**have**  $p_1 \in \text{set } (p_2 \# ps)$

**using**  $p_1$ -*def* *find-closest-set* **by** *blast*

```

    hence  $dist\ p\ p_0 \leq dist\ p\ p_1$ 
    using  $p_1$ -def  $C$  by auto
    thus ?thesis
    using  $p_1$ -def  $A\ C$  by (auto split: prod.splits)
  qed
qed auto

```

```

declare find-closest.simps [simp del]

```

```

fun find-closest-pair-tm :: (point * point)  $\Rightarrow$  point list  $\Rightarrow$  (point  $\times$  point) tm where
  find-closest-pair-tm (c0, c1) [] = 1 return (c0, c1)
| find-closest-pair-tm (c0, c1) [-] = 1 return (c0, c1)
| find-closest-pair-tm (c0, c1) (p0 # ps) = 1 (
  do {
    p1 <- find-closest-tm p0 (dist c0 c1) ps;
    if dist c0 c1  $\leq$  dist p0 p1 then
      find-closest-pair-tm (c0, c1) ps
    else
      find-closest-pair-tm (p0, p1) ps
  }
)

```

```

fun find-closest-pair :: (point * point)  $\Rightarrow$  point list  $\Rightarrow$  (point  $\times$  point) where
  find-closest-pair (c0, c1) [] = (c0, c1)
| find-closest-pair (c0, c1) [-] = (c0, c1)
| find-closest-pair (c0, c1) (p0 # ps) = (
  let p1 = find-closest p0 (dist c0 c1) ps in
  if dist c0 c1  $\leq$  dist p0 p1 then
    find-closest-pair (c0, c1) ps
  else
    find-closest-pair (p0, p1) ps
)

```

```

lemma find-closest-pair-eq-val-find-closest-pair-tm:
  val (find-closest-pair-tm (c0, c1) ps) = find-closest-pair (c0, c1) ps
by (induction (c0, c1) ps arbitrary: c0 c1 rule: find-closest-pair.induct)
  (auto simp: Let-def find-closest-eq-val-find-closest-tm)

```

```

lemma find-closest-pair-set:
  assumes (C0, C1) = find-closest-pair (c0, c1) ps
  shows (C0  $\in$  set ps  $\wedge$  C1  $\in$  set ps)  $\vee$  (C0 = c0  $\wedge$  C1 = c1)
  using assms
proof (induction (c0, c1) ps arbitrary: c0 c1 C0 C1 rule: find-closest-pair.induct)
  case ( $\exists$  c0 c1 p0 p2 ps)
  define p1 where p1-def: p1 = find-closest p0 (dist c0 c1) (p2 # ps)
  hence A: p1  $\in$  set (p2 # ps)
  using find-closest-set by blast
  show ?case
  proof (cases dist c0 c1  $\leq$  dist p0 p1)

```

```

case True
obtain  $C_0' C_1'$  where  $C'\text{-def}: (C_0', C_1') = \text{find-closest-pair } (c_0, c_1) (p_2 \#$ 
 $ps)$ 
  using prod.collapse by blast
  note  $\text{defs} = p_1\text{-def } C'\text{-def}$ 
  hence  $(C_0' \in \text{set } (p_2 \# ps) \wedge C_1' \in \text{set } (p_2 \# ps)) \vee (C_0' = c_0 \wedge C_1' = c_1)$ 
  using 3.hyps(1) True  $p_1\text{-def}$  by blast
  moreover have  $C_0 = C_0' C_1 = C_1'$ 
  using defs True 3.prems by (auto split: prod.splits, metis Pair-inject)+
  ultimately show ?thesis
  by auto
next
case False
obtain  $C_0' C_1'$  where  $C'\text{-def}: (C_0', C_1') = \text{find-closest-pair } (p_0, p_1) (p_2 \#$ 
 $ps)$ 
  using prod.collapse by blast
  note  $\text{defs} = p_1\text{-def } C'\text{-def}$ 
  hence  $(C_0' \in \text{set } (p_2 \# ps) \wedge C_1' \in \text{set } (p_2 \# ps)) \vee (C_0' = p_0 \wedge C_1' = p_1)$ 
  using 3.hyps(2)  $p_1\text{-def}$  False by blast
  moreover have  $C_0 = C_0' C_1 = C_1'$ 
  using defs False 3.prems by (auto split: prod.splits, metis Pair-inject)+
  ultimately show ?thesis
  using A by auto
qed
qed auto

lemma find-closest-pair-c0-ne-c1:
 $c_0 \neq c_1 \implies \text{distinct } ps \implies (C_0, C_1) = \text{find-closest-pair } (c_0, c_1) ps \implies C_0 \neq$ 
 $C_1$ 
proof (induction  $(c_0, c_1) ps$  arbitrary: c_0 c_1 C_0 C_1 rule: find-closest-pair.induct)
  case (3 c_0 c_1 p_0 p_2 ps)
  define  $p_1$  where  $p_1\text{-def}: p_1 = \text{find-closest } p_0 (\text{dist } c_0 c_1) (p_2 \# ps)$ 
  hence  $A: p_0 \neq p_1$ 
  using 3.prems(1,2)
  by (metis distinct.simps(2) find-closest-set length-pos-if-in-set list.set-intros(1))
  show ?case
  proof (cases  $\text{dist } c_0 c_1 \leq \text{dist } p_0 p_1$ )
    case True
    obtain  $C_0' C_1'$  where  $C'\text{-def}: (C_0', C_1') = \text{find-closest-pair } (c_0, c_1) (p_2 \#$ 
 $ps)$ 
    using prod.collapse by blast
    note  $\text{defs} = p_1\text{-def } C'\text{-def}$ 
    hence  $C_0' \neq C_1'$ 
    using 3.hyps(1) 3.prems(1,2) True  $p_1\text{-def}$  by simp
    moreover have  $C_0 = C_0' C_1 = C_1'$ 
    using defs True 3.prems(3) by (auto split: prod.splits, metis Pair-inject)+
    ultimately show ?thesis
    by simp
  next

```

```

case False
obtain  $C_0' C_1'$  where  $C'\text{-def}: (C_0', C_1') = \text{find-closest-pair } (p_0, p_1) (p_2 \#$ 
 $ps)$ 
  using prod.collapse by blast
  note  $\text{defs} = p_1\text{-def } C'\text{-def}$ 
  hence  $C_0' \neq C_1'$ 
  using  $\exists.\text{hyps}(2) \exists.\text{prems}(2) A \text{ False } p_1\text{-def}$  by simp
  moreover have  $C_0 = C_0' C_1 = C_1'$ 
  using  $\text{defs False } \exists.\text{prems}(3)$  by (auto split: prod.splits, metis Pair-inject)+
  ultimately show ?thesis
  by simp
qed
qed auto

```

**lemma** *find-closest-pair-dist-mono*:

```

assumes  $(C_0, C_1) = \text{find-closest-pair } (c_0, c_1) ps$ 
shows  $\text{dist } C_0 C_1 \leq \text{dist } c_0 c_1$ 
using assms
proof (induction (c_0, c_1) ps arbitrary: c_0 c_1 C_0 C_1 rule: find-closest-pair.induct)
  case  $(\exists c_0 c_1 p_0 p_2 ps)$ 
  define  $p_1$  where  $p_1\text{-def}: p_1 = \text{find-closest } p_0 (\text{dist } c_0 c_1) (p_2 \# ps)$ 
  show ?case
  proof (cases dist c_0 c_1 ≤ dist p_0 p_1)
    case True
    obtain  $C_0' C_1'$  where  $C'\text{-def}: (C_0', C_1') = \text{find-closest-pair } (c_0, c_1) (p_2 \#$ 
 $ps)$ 
    using prod.collapse by blast
    note  $\text{defs} = p_1\text{-def } C'\text{-def}$ 
    hence  $\text{dist } C_0' C_1' \leq \text{dist } c_0 c_1$ 
    using  $\exists.\text{hyps}(1) \text{ True } p_1\text{-def}$  by simp
    moreover have  $C_0 = C_0' C_1 = C_1'$ 
    using  $\text{defs True } \exists.\text{prems}$  by (auto split: prod.splits, metis Pair-inject)+
    ultimately show ?thesis
    by simp
  next
  case False
  obtain  $C_0' C_1'$  where  $C'\text{-def}: (C_0', C_1') = \text{find-closest-pair } (p_0, p_1) (p_2 \#$ 
 $ps)$ 
  using prod.collapse by blast
  note  $\text{defs} = p_1\text{-def } C'\text{-def}$ 
  hence  $\text{dist } C_0' C_1' \leq \text{dist } p_0 p_1$ 
  using  $\exists.\text{hyps}(2) \text{ False } p_1\text{-def}$  by blast
  moreover have  $C_0 = C_0' C_1 = C_1'$ 
  using  $\text{defs False } \exists.\text{prems}(1)$  by (auto split: prod.splits, metis Pair-inject)+
  ultimately show ?thesis
  using False by simp
qed
qed auto

```

**lemma** *find-closest-pair-dist*:  
**assumes** *sorted-snd ps*  $(C_0, C_1) = \text{find-closest-pair } (c_0, c_1) \text{ ps}$   
**shows** *sparse*  $(\text{dist } C_0 \ C_1) (\text{set } ps)$   
**using** *assms*  
**proof** (*induction*  $(c_0, c_1) \text{ ps}$  *arbitrary*:  $c_0 \ c_1 \ C_0 \ C_1$  *rule*: *find-closest-pair.induct*)  
**case**  $(\exists \ c_0 \ c_1 \ p_0 \ p_2 \ ps)$   
**define**  $p_1$  **where**  $p_1\text{-def}$ :  $p_1 = \text{find-closest } p_0 \ (\text{dist } c_0 \ c_1) \ (p_2 \ \# \ ps)$   
**show** *?case*  
**proof** *cases*  
**assume**  $\exists p \in \text{set } (p_2 \ \# \ ps)$ .  $\text{dist } p_0 \ p < \text{dist } c_0 \ c_1$   
**hence**  $A$ :  $\forall p \in \text{set } (p_2 \ \# \ ps)$ .  $\text{dist } p_0 \ p_1 \leq \text{dist } p_0 \ p$   $\text{dist } p_0 \ p_1 < \text{dist } c_0 \ c_1$   
**using**  $p_1\text{-def}$  *find-closest-dist*  $\exists$ .*prems*(1) *le-less-trans* **by** *blast+*  
**obtain**  $C_0' \ C_1'$  **where**  $C'\text{-def}$ :  $(C_0', C_1') = \text{find-closest-pair } (p_0, p_1) \ (p_2 \ \# \ ps)$   
**using** *prod.collapse* **by** *blast*  
**hence**  $B$ :  $(C_0', C_1') = \text{find-closest-pair } (c_0, c_1) \ (p_0 \ \# \ p_2 \ \# \ ps)$   
**using**  $A(2)$   $p_1\text{-def}$  **by** *simp*  
**have** *sparse*  $(\text{dist } C_0' \ C_1') (\text{set } (p_2 \ \# \ ps))$   
**using**  $\exists$ .*hyps*(2)[*of*  $p_1 \ C_0' \ C_1'$ ]  $p_1\text{-def}$   $C'\text{-def}$   $\exists$ .*prems*(1)  $A(2)$  *sorted-snd-def*  
**by** *fastforce*  
**moreover** **have**  $\text{dist } C_0' \ C_1' \leq \text{dist } p_0 \ p_1$   
**using**  $C'\text{-def}$  *find-closest-pair-dist-mono* **by** *blast*  
**ultimately** **have** *sparse*  $(\text{dist } C_0' \ C_1') (\text{set } (p_0 \ \# \ p_2 \ \# \ ps))$   
**using**  $A$  *sparse-identity* *order-trans* **by** *blast*  
**thus** *?thesis*  
**using**  $B$  **by** (*metis*  $\exists$ .*prems*(2)) *Pair-inject*  
**next**  
**assume**  $A$ :  $\neg (\exists p \in \text{set } (p_2 \ \# \ ps))$ .  $\text{dist } p_0 \ p < \text{dist } c_0 \ c_1$   
**hence**  $B$ :  $\text{dist } c_0 \ c_1 \leq \text{dist } p_0 \ p_1$   
**using** *find-closest-set*[*of*  $p_2 \ \# \ ps \ p_0 \ \text{dist } c_0 \ c_1$ ]  $p_1\text{-def}$  **by** *auto*  
**obtain**  $C_0' \ C_1'$  **where**  $C'\text{-def}$ :  $(C_0', C_1') = \text{find-closest-pair } (c_0, c_1) \ (p_2 \ \# \ ps)$   
**using** *prod.collapse* **by** *blast*  
**hence**  $C$ :  $(C_0', C_1') = \text{find-closest-pair } (c_0, c_1) \ (p_0 \ \# \ p_2 \ \# \ ps)$   
**using**  $B$   $p_1\text{-def}$  **by** *simp*  
**have** *sparse*  $(\text{dist } C_0' \ C_1') (\text{set } (p_2 \ \# \ ps))$   
**using**  $\exists$ .*hyps*(1)[*of*  $p_1 \ C_0' \ C_1'$ ]  $p_1\text{-def}$   $C'\text{-def}$   $B$   $\exists$ .*prems* *sorted-snd-def* **by**  
*simp*  
**moreover** **have**  $\text{dist } C_0' \ C_1' \leq \text{dist } c_0 \ c_1$   
**using**  $C'\text{-def}$  *find-closest-pair-dist-mono* **by** *blast*  
**ultimately** **have** *sparse*  $(\text{dist } C_0' \ C_1') (\text{set } (p_0 \ \# \ p_2 \ \# \ ps))$   
**using**  $A$  *sparse-identity*[*of*  $\text{dist } C_0' \ C_1' \ p_2 \ \# \ ps \ p_0$ ] *order-trans* **by** *force*  
**thus** *?thesis*  
**using**  $C$  **by** (*metis*  $\exists$ .*prems*(2)) *Pair-inject*  
**qed**  
**qed** (*auto simp: sparse-def*)  
**declare** *find-closest-pair.simps* [*simp del*]

```

fun combine-tm :: (point × point) ⇒ (point × point) ⇒ int ⇒ point list ⇒ (point
× point) tm where
  combine-tm (p0L, p1L) (p0R, p1R) l ps = 1 (
    let (c0, c1) = if dist p0L p1L < dist p0R p1R then (p0L, p1L) else (p0R, p1R) in
    do {
      ps' <- filter-tm (λp. dist p (l, snd p) < dist c0 c1) ps;
      find-closest-pair-tm (c0, c1) ps'
    }
  )

```

```

fun combine :: (point × point) ⇒ (point × point) ⇒ int ⇒ point list ⇒ (point ×
point) where
  combine (p0L, p1L) (p0R, p1R) l ps = (
    let (c0, c1) = if dist p0L p1L < dist p0R p1R then (p0L, p1L) else (p0R, p1R) in
    let ps' = filter (λp. dist p (l, snd p) < dist c0 c1) ps in
    find-closest-pair (c0, c1) ps'
  )

```

**lemma** combine-eq-val-combine-tm:

```

val (combine-tm (p0L, p1L) (p0R, p1R) l ps) = combine (p0L, p1L) (p0R, p1R) l
ps
by (auto simp: filter-eq-val-filter-tm find-closest-pair-eq-val-find-closest-pair-tm)

```

**lemma** combine-set:

```

assumes (c0, c1) = combine (p0L, p1L) (p0R, p1R) l ps
shows (c0 ∈ set ps ∧ c1 ∈ set ps) ∨ (c0 = p0L ∧ c1 = p1L) ∨ (c0 = p0R ∧ c1
= p1R)
proof –
  obtain C0' C1' where C'-def: (C0', C1') = (if dist p0L p1L < dist p0R p1R
then (p0L, p1L) else (p0R, p1R))
  by metis
  define ps' where ps'-def: ps' = filter (λp. dist p (l, snd p) < dist C0' C1') ps
  obtain C0 C1 where C-def: (C0, C1) = find-closest-pair (C0', C1') ps'
  using prod.collapse by blast
  note defs = C'-def ps'-def C-def
  have (C0 ∈ set ps' ∧ C1 ∈ set ps') ∨ (C0 = C0' ∧ C1 = C1')
  using C-def find-closest-pair-set by blast+
  hence (C0 ∈ set ps ∧ C1 ∈ set ps) ∨ (C0 = C0' ∧ C1 = C1')
  using ps'-def by auto
  moreover have C0 = c0 C1 = c1
  using assms defs apply (auto split: if-splits prod.splits) by (metis Pair-inject)+
  ultimately show ?thesis
  using C'-def by (auto split: if-splits)

```

qed

**lemma** combine-c0-ne-c1:

```

assumes p0L ≠ p1L p0R ≠ p1R distinct ps
assumes (c0, c1) = combine (p0L, p1L) (p0R, p1R) l ps
shows c0 ≠ c1

```

**proof** –  
**obtain**  $C_0' C_1'$  **where**  $C'\text{-def}: (C_0', C_1') = (\text{if } \text{dist } p_{0L} p_{1L} < \text{dist } p_{0R} p_{1R} \text{ then } (p_{0L}, p_{1L}) \text{ else } (p_{0R}, p_{1R}))$   
**by** *metis*  
**define**  $ps'$  **where**  $ps'\text{-def}: ps' = \text{filter } (\lambda p. \text{dist } p (l, \text{snd } p) < \text{dist } C_0' C_1') ps$   
**obtain**  $C_0 C_1$  **where**  $C\text{-def}: (C_0, C_1) = \text{find-closest-pair } (C_0', C_1') ps'$   
**using** *prod.collapse* **by** *blast*  
**note**  $\text{defs} = C'\text{-def } ps'\text{-def } C\text{-def}$   
**have**  $C_0 \neq C_1$   
**using**  $\text{defs } \text{find-closest-pair-c0-ne-c1}[\text{of } C_0' C_1' ps'] \text{ assms}$  **by** (*auto split: if-splits*)  
**moreover** **have**  $C_0 = c_0 C_1 = c_1$   
**using**  $\text{assms}(4) \text{ defs}$  **apply** (*auto split: if-splits prod.splits*) **by** (*metis Pair-inject*)+  
**ultimately show** *?thesis*  
**by** *blast*  
**qed**

**lemma** *combine-dist*:

**assumes** *sorted-snd ps set ps = ps<sub>L</sub> ∪ ps<sub>R</sub>*  
**assumes**  $\forall p \in ps_L. \text{fst } p \leq l \forall p \in ps_R. l \leq \text{fst } p$   
**assumes** *sparse (dist p<sub>0L</sub> p<sub>1L</sub>) ps<sub>L</sub> sparse (dist p<sub>0R</sub> p<sub>1R</sub>) ps<sub>R</sub>*  
**assumes**  $(c_0, c_1) = \text{combine } (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) l ps$   
**shows** *sparse (dist c<sub>0</sub> c<sub>1</sub>) (set ps)*  
**proof** –  
**obtain**  $C_0' C_1'$  **where**  $C'\text{-def}: (C_0', C_1') = (\text{if } \text{dist } p_{0L} p_{1L} < \text{dist } p_{0R} p_{1R} \text{ then } (p_{0L}, p_{1L}) \text{ else } (p_{0R}, p_{1R}))$   
**by** *metis*  
**define**  $ps'$  **where**  $ps'\text{-def}: ps' = \text{filter } (\lambda p. \text{dist } p (l, \text{snd } p) < \text{dist } C_0' C_1') ps$   
**obtain**  $C_0 C_1$  **where**  $C\text{-def}: (C_0, C_1) = \text{find-closest-pair } (C_0', C_1') ps'$   
**using** *prod.collapse* **by** *blast*  
**note**  $\text{defs} = C'\text{-def } ps'\text{-def } C\text{-def}$   
**have**  $EQ: C_0 = c_0 C_1 = c_1$   
**using**  $\text{defs } \text{assms}(7)$  **apply** (*auto split: if-splits prod.splits*) **by** (*metis Pair-inject*)+  
**have**  $ps': ps' = \text{filter } (\lambda p. l - \text{dist } C_0' C_1' < \text{fst } p \wedge \text{fst } p < l + \text{dist } C_0' C_1')$   
 $ps$   
**using**  $ps'\text{-def } \text{dist-transform}$  **by** *simp*  
**have**  $ps_L: \text{sparse } (\text{dist } C_0' C_1') ps_L$   
**using**  $\text{assms}(3,5) C'\text{-def } \text{sparse-def}$  **apply** (*auto split: if-splits*) **by** *force*+  
**have**  $ps_R: \text{sparse } (\text{dist } C_0' C_1') ps_R$   
**using**  $\text{assms}(4,6) C'\text{-def } \text{sparse-def}$  **apply** (*auto split: if-splits*) **by** *force*+  
**have** *sorted-snd ps'*  
**using**  $ps'\text{-def } \text{assms}(1) \text{ sorted-snd-def } \text{sorted-wrt-filter}$  **by** *blast*  
**hence**  $*$ : *sparse (dist C<sub>0</sub> C<sub>1</sub>) (set ps')*  
**using** *find-closest-pair-dist C-def* **by** *simp*  
**have**  $\forall p_0 \in \text{set } ps. \forall p_1 \in \text{set } ps. p_0 \neq p_1 \wedge \text{dist } p_0 p_1 < \text{dist } C_0' C_1' \longrightarrow p_0 \in \text{set } ps' \wedge p_1 \in \text{set } ps'$   
 $\text{set } ps' \wedge p_1 \in \text{set } ps'$   
**using** *set-band-filter ps' ps<sub>L</sub> ps<sub>R</sub> assms(2,3,4)* **by** *blast*  
**moreover** **have**  $\text{dist } C_0 C_1 \leq \text{dist } C_0' C_1'$   
**using** *C-def find-closest-pair-dist-mono* **by** *blast*

```

ultimately have  $\forall p_0 \in \text{set } ps. \forall p_1 \in \text{set } ps. p_0 \neq p_1 \wedge \text{dist } p_0 \ p_1 < \text{dist } C_0$ 
 $C_1 \longrightarrow p_0 \in \text{set } ps' \wedge p_1 \in \text{set } ps'$ 
  by simp
  hence sparse (dist  $C_0 \ C_1$ ) (set  $ps$ )
  using sparse-def * by (meson not-less)
  thus ?thesis
  using EQ by blast
qed

```

```

declare combine.simps [simp del]
declare combine-tm.simps[simp del]

```

### 2.1.2 Divide and Conquer Algorithm

```

declare split-at-take-drop-conv [simp add]

```

```

function closest-pair-rec-tm :: point list  $\Rightarrow$  (point list  $\times$  point  $\times$  point) tm where
  closest-pair-rec-tm xs = 1 (
    do {
      n <- length-tm xs;
      if n  $\leq$  3 then
        do {
          ys <- mergesort-tm snd xs;
          p <- closest-pair-bf-tm xs;
          return (ys, p)
        }
      else
        do {
          (xsL, xsR) <- split-at-tm (n div 2) xs;
          (ysL, p0L, p1L) <- closest-pair-rec-tm xsL;
          (ysR, p0R, p1R) <- closest-pair-rec-tm xsR;
          ys <- merge-tm snd ysL ysR;
          (p0, p1) <- combine-tm (p0L, p1L) (p0R, p1R) (fst (hd xsR)) ys;
          return (ys, p0, p1)
        }
      }
    )
  by pat-completeness auto
termination closest-pair-rec-tm
  by (relation Wellfounded.measure ( $\lambda xs. \text{length } xs$ ))
    (auto simp add: length-eq-val-length-tm split-at-eq-val-split-at-tm)

```

```

function closest-pair-rec :: point list  $\Rightarrow$  (point list * point * point) where
  closest-pair-rec xs = (
    let n = length xs in
    if n  $\leq$  3 then
      (mergesort snd xs, closest-pair-bf xs)
    else
      let (xsL, xsR) = split-at (n div 2) xs in

```

```

    let (ysL, p0L, p1L) = closest-pair-rec xsL in
    let (ysR, p0R, p1R) = closest-pair-rec xsR in
    let ys = merge snd ysL ysR in
    (ys, combine (p0L, p1L) (p0R, p1R) (fst (hd xsR)) ys)
  )
  by pat-completeness auto
termination closest-pair-rec
  by (relation Wellfounded.measure (λxs. length xs))
    (auto simp: Let-def)

declare split-at-take-drop-conv [simp del]

lemma closest-pair-rec-simps:
  assumes n = length xs ∧ (n ≤ 3)
  shows closest-pair-rec xs = (
    let (xsL, xsR) = split-at (n div 2) xs in
    let (ysL, p0L, p1L) = closest-pair-rec xsL in
    let (ysR, p0R, p1R) = closest-pair-rec xsR in
    let ys = merge snd ysL ysR in
    (ys, combine (p0L, p1L) (p0R, p1R) (fst (hd xsR)) ys)
  )
  using assms by (auto simp: Let-def)

declare closest-pair-rec.simps [simp del]

lemma closest-pair-rec-eq-val-closest-pair-rec-tm:
  val (closest-pair-rec-tm xs) = closest-pair-rec xs
proof (induction rule: length-induct)
  case (1 xs)
  define n where n = length xs
  obtain xsL xsR where xs-def: (xsL, xsR) = split-at (n div 2) xs
  by (metis surj-pair)
  note defs = n-def xs-def
  show ?case
  proof cases
  assume n ≤ 3
  then show ?thesis
  using defs
  by (auto simp: length-eq-val-length-tm mergesort-eq-val-mergesort-tm
    closest-pair-bf-eq-val-closest-pair-bf-tm closest-pair-rec.simps)
next
  assume asm: ¬ n ≤ 3
  have length xsL < length xs length xsR < length xs
  using asm defs by (auto simp: split-at-take-drop-conv)
  hence val (closest-pair-rec-tm xsL) = closest-pair-rec xsL
    val (closest-pair-rec-tm xsR) = closest-pair-rec xsR
  using 1.IH by blast+
  thus ?thesis
  using asm defs

```

**apply** (*subst closest-pair-rec.simps, subst closest-pair-rec-tm.simps*)  
**by** (*auto simp del: closest-pair-rec-tm.simps*  
*simp add: Let-def length-eq-val-length-tm merge-eq-val-merge-tm*  
*split-at-eq-val-split-at-tm combine-eq-val-combine-tm*  
*split: prod.split*)

**qed**  
**qed**

**lemma** *closest-pair-rec-set-length-sorted-snd:*

**assumes** (*ys, p*) = *closest-pair-rec xs*

**shows** *set ys = set xs  $\wedge$  length ys = length xs  $\wedge$  sorted-snd ys*

**using** *assms*

**proof** (*induction xs arbitrary: ys p rule: length-induct*)

**case** (*1 xs*)

**let** *?n = length xs*

**show** *?case*

**proof** (*cases ?n  $\leq$  3*)

**case** *True*

**thus** *?thesis using 1.prem sorted-snd-def*

**by** (*auto simp: mergesort closest-pair-rec.simps*)

**next**

**case** *False*

**obtain** *XS<sub>L</sub> XS<sub>R</sub>* **where** *XS<sub>LR</sub>-def: (XS<sub>L</sub>, XS<sub>R</sub>) = split-at (?n div 2) xs*

**using** *prod.collapse by blast*

**define** *L* **where** *L = fst (hd XS<sub>R</sub>)*

**obtain** *YS<sub>L</sub> P<sub>L</sub>* **where** *YSP<sub>L</sub>-def: (YS<sub>L</sub>, P<sub>L</sub>) = closest-pair-rec XS<sub>L</sub>*

**using** *prod.collapse by blast*

**obtain** *YS<sub>R</sub> P<sub>R</sub>* **where** *YSP<sub>R</sub>-def: (YS<sub>R</sub>, P<sub>R</sub>) = closest-pair-rec XS<sub>R</sub>*

**using** *prod.collapse by blast*

**define** *YS* **where** *YS = merge ( $\lambda p. snd p$ ) YS<sub>L</sub> YS<sub>R</sub>*

**define** *P* **where** *P = combine P<sub>L</sub> P<sub>R</sub> L YS*

**note** *defs = XS<sub>LR</sub>-def L-def YSP<sub>L</sub>-def YSP<sub>R</sub>-def YS-def P-def*

**have** *length XS<sub>L</sub> < length xs length XS<sub>R</sub> < length xs*

**using** *False defs by (auto simp: split-at-take-drop-conv)*

**hence** *IH: set XS<sub>L</sub> = set YS<sub>L</sub> set XS<sub>R</sub> = set YS<sub>R</sub>*

*length XS<sub>L</sub> = length YS<sub>L</sub> length XS<sub>R</sub> = length YS<sub>R</sub>*

*sorted-snd YS<sub>L</sub> sorted-snd YS<sub>R</sub>*

**using** *1.IH defs by metis+*

**have** *set xs = set XS<sub>L</sub>  $\cup$  set XS<sub>R</sub>*

**using** *defs by (auto simp: set-take-drop split-at-take-drop-conv)*

**hence** *SET: set xs = set YS*

**using** *set-merge IH(1,2) defs by fast*

**have** *length xs = length XS<sub>L</sub> + length XS<sub>R</sub>*

**using** *defs by (auto simp: split-at-take-drop-conv)*

**hence** *LENGTH: length xs = length YS*

```

using IH(3,4) length-merge defs by metis

have SORTED: sorted-snd YS
  using IH(5,6) by (simp add: defs sorted-snd-def sorted-merge)

have (YS, P) = closest-pair-rec xs
  using False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)
hence (ys, p) = (YS, P)
  using 1.prems by argo
thus ?thesis
  using SET LENGTH SORTED by simp
qed
qed

lemma closest-pair-rec-distinct:
  assumes distinct xs (ys, p) = closest-pair-rec xs
  shows distinct ys
  using assms
proof (induction xs arbitrary: ys p rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    thus ?thesis using 1.prems
    by (auto simp: mergesort closest-pair-rec.simps)
  next
  case False

  obtain XSL XSR where XSLR-def: (XSL, XSR) = split-at (?n div 2) xs
    using prod.collapse by blast
  define L where L = fst (hd XSR)
  obtain YSL PL where YSPL-def: (YSL, PL) = closest-pair-rec XSL
    using prod.collapse by blast
  obtain YSR PR where YSPR-def: (YSR, PR) = closest-pair-rec XSR
    using prod.collapse by blast
  define YS where YS = merge (λp. snd p) YSL YSR
  define P where P = combine PL PR L YS
  note defs = XSLR-def L-def YSPL-def YSPR-def YS-def P-def

  have length XSL < length xs length XSR < length xs
    using False defs by (auto simp: split-at-take-drop-conv)
  moreover have distinct XSL distinct XSR
    using 1.prems(1) defs by (auto simp: split-at-take-drop-conv)
  ultimately have IH: distinct YSL distinct YSR
    using 1.IH defs by blast+

  have set XSL ∩ set XSR = {}
    using 1.prems(1) defs by (auto simp: split-at-take-drop-conv set-take-disj-set-drop-if-distinct)

```

**moreover have**  $set\ XS_L = set\ YS_L$   $set\ XS_R = set\ YS_R$   
**using** *closest-pair-rec-set-length-sorted-snd* **defs by blast+**  
**ultimately have**  $set\ YS_L \cap set\ YS_R = \{\}$   
**by blast**  
**hence** *DISTINCT*: *distinct YS*  
**using** *distinct-merge IH* **defs by blast**

**have**  $(YS, P) = closest\ pair\ rec\ xs$   
**using** *False closest-pair-rec-simps* **defs by (auto simp: Let-def split: prod.split)**  
**hence**  $(ys, p) = (YS, P)$   
**using** *1.premis* **by argo**  
**thus** *?thesis*  
**using** *DISTINCT* **by blast**

**qed**  
**qed**

**lemma** *closest-pair-rec-c0-c1*:  
**assumes**  $1 < length\ xs$  *distinct xs*  $(ys, c_0, c_1) = closest\ pair\ rec\ xs$   
**shows**  $c_0 \in set\ xs \wedge c_1 \in set\ xs \wedge c_0 \neq c_1$   
**using** *assms*  
**proof** (*induction xs arbitrary: ys c\_0 c\_1 rule: length-induct*)  
**case**  $(1\ xs)$   
**let**  $?n = length\ xs$   
**show** *?case*  
**proof** (*cases ?n ≤ 3*)  
**case** *True*  
**hence**  $(c_0, c_1) = closest\ pair\ bf\ xs$   
**using** *1.premis(3) closest-pair-rec.simps* **by simp**  
**thus** *?thesis*  
**using** *1.premis(1,2) closest-pair-bf-c0-c1* **by simp**  
**next**  
**case** *False*

**obtain**  $XS_L\ XS_R$  **where**  $XS_{LR}\text{-def}: (XS_L, XS_R) = split\ at\ (?n\ div\ 2)\ xs$   
**using** *prod.collapse* **by blast**  
**define**  $L$  **where**  $L = fst\ (hd\ XS_R)$

**obtain**  $YS_L\ C_{0L}\ C_{1L}$  **where**  $YSC_{01L}\text{-def}: (YS_L, C_{0L}, C_{1L}) = closest\ pair\ rec\ XS_L$   
**using** *prod.collapse* **by metis**  
**obtain**  $YS_R\ C_{0R}\ C_{1R}$  **where**  $YSC_{01R}\text{-def}: (YS_R, C_{0R}, C_{1R}) = closest\ pair\ rec\ XS_R$   
**using** *prod.collapse* **by metis**

**define**  $YS$  **where**  $YS = merge\ (\lambda p.\ snd\ p)\ YS_L\ YS_R$   
**obtain**  $C_0\ C_1$  **where**  $C_{01}\text{-def}: (C_0, C_1) = combine\ (C_{0L}, C_{1L})\ (C_{0R}, C_{1R})$   
 $L\ YS$   
**using** *prod.collapse* **by metis**  
**note**  $defs = XS_{LR}\text{-def}\ L\text{-def}\ YSC_{01L}\text{-def}\ YSC_{01R}\text{-def}\ YS\text{-def}\ C_{01}\text{-def}$

**have**  $1 < \text{length } XS_L$   $\text{length } XS_L < \text{length } xs$  *distinct*  $XS_L$   
**using** *False 1.premis(2) defs by (auto simp: split-at-take-drop-conv)*  
**hence**  $C_{0L} \in \text{set } XS_L$   $C_{1L} \in \text{set } XS_L$  **and** *IHL1:  $C_{0L} \neq C_{1L}$*   
**using** *1.IH defs bymetis+*  
**hence** *IHL2:  $C_{0L} \in \text{set } xs$   $C_{1L} \in \text{set } xs$*   
**using** *split-at-take-drop-conv in-set-takeD fst-conv defs bymetis+*

**have**  $1 < \text{length } XS_R$   $\text{length } XS_R < \text{length } xs$  *distinct*  $XS_R$   
**using** *False 1.premis(2) defs by (auto simp: split-at-take-drop-conv)*  
**hence**  $C_{0R} \in \text{set } XS_R$   $C_{1R} \in \text{set } XS_R$  **and** *IHR1:  $C_{0R} \neq C_{1R}$*   
**using** *1.IH defs bymetis+*  
**hence** *IHR2:  $C_{0R} \in \text{set } xs$   $C_{1R} \in \text{set } xs$*   
**using** *split-at-take-drop-conv in-set-dropD snd-conv defs bymetis+*

**have**  $*$ :  $(YS, C_0, C_1) = \text{closest-pair-rec } xs$   
**using** *False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)*  
**have**  $YS$ : *set  $xs = \text{set } YS$  *distinct*  $YS$*   
**using** *1.premis(2) closest-pair-rec-set-length-sorted-snd closest-pair-rec-distinct*  
 $*$  **by** *blast+*

**have**  $C_0 \in \text{set } xs$   $C_1 \in \text{set } xs$   
**using** *combine-set IHL2 IHR2 YS defs by blast+*  
**moreover** **have**  $C_0 \neq C_1$   
**using** *combine-c0-ne-c1 IHL1(1) IHR1(1) YS defs by blast*  
**ultimately** **show** *?thesis*  
**using** *1.premis(3) \* by (metis Pair-inject)*

**qed**  
**qed**

**lemma** *closest-pair-rec-dist*:  
**assumes**  $1 < \text{length } xs$  *sorted-fst*  $(ys, c_0, c_1) = \text{closest-pair-rec } xs$   
**shows** *sparse (dist  $c_0$   $c_1$ ) (set  $xs$ )*  
**using** *assms*

**proof** (*induction xs arbitrary: ys c\_0 c\_1 rule: length-induct*)  
**case** ( $1\ xs$ )  
**let**  $?n = \text{length } xs$   
**show** *?case*  
**proof** (*cases ?n  $\leq$  3*)  
**case** *True*  
**hence**  $(c_0, c_1) = \text{closest-pair-bf } xs$   
**using** *1.premis(3) closest-pair-rec.simps by simp*  
**thus** *?thesis*  
**using** *1.premis(1,3) closest-pair-bf-dist by metis*

**next**  
**case** *False*

**obtain**  $XS_L$   $XS_R$  **where**  *$XS_{LR}$ -def:  $(XS_L, XS_R) = \text{split-at } (?n \text{ div } 2) xs$*   
**using** *prod.collapse by blast*

**define**  $L$  **where**  $L = fst (hd XS_R)$

**obtain**  $YS_L C_{0L} C_{1L}$  **where**  $YSC_{01L}$ -def:  $(YS_L, C_{0L}, C_{1L}) = closest\text{-pair}\text{-rec}$   
 $XS_L$   
**using**  $prod.collapse$  **by**  $metis$

**obtain**  $YS_R C_{0R} C_{1R}$  **where**  $YSC_{01R}$ -def:  $(YS_R, C_{0R}, C_{1R}) = closest\text{-pair}\text{-rec}$   
 $XS_R$   
**using**  $prod.collapse$  **by**  $metis$

**define**  $YS$  **where**  $YS = merge (\lambda p. snd p) YS_L YS_R$

**obtain**  $C_0 C_1$  **where**  $C_{01}$ -def:  $(C_0, C_1) = combine (C_{0L}, C_{1L}) (C_{0R}, C_{1R})$   
 $L YS$   
**using**  $prod.collapse$  **by**  $metis$

**note**  $defs = XS_{LR}$ -def  $L$ -def  $YSC_{01L}$ -def  $YSC_{01R}$ -def  $YS$ -def  $C_{01}$ -def

**have**  $XSLR$ :  $XS_L = take (?n div 2) xs$   $XS_R = drop (?n div 2) xs$   
**using**  $defs$  **by**  $(auto simp: split\text{-at}\text{-take}\text{-drop}\text{-conv})$

**have**  $1 < length XS_L$   $length XS_L < length xs$   
**using**  $False XSLR$  **by**  $simp\text{-all}$

**moreover** **have**  $sorted\text{-fst} XS_L$   
**using**  $1.premis(2) XSLR$  **by**  $(auto simp: sorted\text{-fst}\text{-def} sorted\text{-wrt}\text{-take})$

**ultimately** **have**  $L$ :  $sparse (dist C_{0L} C_{1L}) (set XS_L)$   
 $set XS_L = set YS_L$   
**using**  $1 closest\text{-pair}\text{-rec}\text{-set}\text{-length}\text{-sorted}\text{-snd} closest\text{-pair}\text{-rec}\text{-c0}\text{-c1}$   
 $YSC_{01L}$ -def **by**  $blast+$

**hence**  $IHL$ :  $sparse (dist C_{0L} C_{1L}) (set YS_L)$   
**by**  $argo$

**have**  $1 < length XS_R$   $length XS_R < length xs$   
**using**  $False XSLR$  **by**  $simp\text{-all}$

**moreover** **have**  $sorted\text{-fst} XS_R$   
**using**  $1.premis(2) XSLR$  **by**  $(auto simp: sorted\text{-fst}\text{-def} sorted\text{-wrt}\text{-drop})$

**ultimately** **have**  $R$ :  $sparse (dist C_{0R} C_{1R}) (set XS_R)$   
 $set XS_R = set YS_R$   
**using**  $1 closest\text{-pair}\text{-rec}\text{-set}\text{-length}\text{-sorted}\text{-snd} closest\text{-pair}\text{-rec}\text{-c0}\text{-c1}$   
 $YSC_{01R}$ -def **by**  $blast+$

**hence**  $IHR$ :  $sparse (dist C_{0R} C_{1R}) (set YS_R)$   
**by**  $argo$

**have**  $*$ :  $(YS, C_0, C_1) = closest\text{-pair}\text{-rec} xs$   
**using**  $False closest\text{-pair}\text{-rec}\text{-simps} defs$  **by**  $(auto simp: Let\text{-def} split: prod.split)$

**have**  $set xs = set YS$   $sorted\text{-snd} YS$   
**using**  $1.premis(2) closest\text{-pair}\text{-rec}\text{-set}\text{-length}\text{-sorted}\text{-snd} closest\text{-pair}\text{-rec}\text{-distinct}$   
 $*$  **by**  $blast+$

**moreover** **have**  $\forall p \in set YS_L. fst p \leq L$   
**using**  $False 1.premis(2) XSLR L$ -def  $L(2)$   $sorted\text{-fst}\text{-take}\text{-less}\text{-hd}\text{-drop}$  **by**  $simp$

**moreover** **have**  $\forall p \in set YS_R. L \leq fst p$

```

using False 1.premis(2) XSLR L-def R(2) sorted-fst-hd-drop-less-drop by simp
moreover have set YS = set YSL ∪ set YSR
using set-merge defs by fast
moreover have (C0, C1) = combine (C0L, C1L) (C0R, C1R) L YS
by (auto simp add: defs)
ultimately have sparse (dist C0 C1) (set xs)
using combine-dist IHL IHR by auto
moreover have (YS, C0, C1) = (ys, c0, c1)
using 1.premis(3) * by simp
ultimately show ?thesis
by blast
qed
qed

```

```

fun closest-pair-tm :: point list ⇒ (point * point) tm where
  closest-pair-tm [] = 1 return undefined
| closest-pair-tm [-] = 1 return undefined
| closest-pair-tm ps = 1 (
  do {
    xs <- mergesort-tm fst ps;
    (-, p) <- closest-pair-rec-tm xs;
    return p
  }
)

```

```

fun closest-pair :: point list ⇒ (point * point) where
  closest-pair [] = undefined
| closest-pair [-] = undefined
| closest-pair ps = (let (-, p) = closest-pair-rec (mergesort fst ps) in p)

```

```

lemma closest-pair-eq-val-closest-pair-tm:
  val (closest-pair-tm ps) = closest-pair ps
by (induction ps rule: induct-list012)
  (auto simp del: closest-pair-rec-tm.simps mergesort-tm.simps
    simp add: closest-pair-rec-eq-val-closest-pair-rec-tm mergesort-eq-val-mergesort-tm
    split: prod.split)

```

```

lemma closest-pair-simps:
  1 < length ps ⇒ closest-pair ps = (let (-, p) = closest-pair-rec (mergesort fst
ps) in p)
by (induction ps rule: induct-list012) auto

```

```

declare closest-pair.simps [simp del]

```

```

theorem closest-pair-c0-c1:
  assumes 1 < length ps distinct ps (c0, c1) = closest-pair ps
  shows c0 ∈ set ps c1 ∈ set ps c0 ≠ c1
  using assms closest-pair-rec-c0-c1 [of mergesort fst ps]
  by (auto simp: closest-pair-simps mergesort split: prod.splits)

```

**theorem** *closest-pair-dist*:  
**assumes**  $1 < \text{length } ps$   $(c_0, c_1) = \text{closest-pair } ps$   
**shows**  $\text{sparse } (\text{dist } c_0 \ c_1)$   $(\text{set } ps)$   
**using** *assms sorted-fst-def closest-pair-rec-dist*[of mergesort fst ps] *closest-pair-rec-c0-c1*[of mergesort fst ps]  
**by**  $(\text{auto simp: closest-pair-simps mergesort split: prod.splits})$

## 2.2 Time Complexity Proof

### 2.2.1 Core Argument

**lemma** *core-argument*:

**fixes**  $\delta :: \text{real}$  **and**  $p :: \text{point}$  **and**  $ps :: \text{point list}$   
**assumes**  $\text{distinct } (p \# ps)$   $\text{sorted-snd } (p \# ps)$   $0 \leq \delta$   $\text{set } (p \# ps) = ps_L \cup ps_R$   
**assumes**  $\forall q \in \text{set } (p \# ps). l - \delta < \text{fst } q \wedge \text{fst } q < l + \delta$   
**assumes**  $\forall q \in ps_L. \text{fst } q \leq l \ \forall q \in ps_R. l \leq \text{fst } q$   
**assumes**  $\text{sparse } \delta \ ps_L$   $\text{sparse } \delta \ ps_R$   
**shows**  $\text{length } (\text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) \ ps) \leq \gamma$

**proof** –

**define**  $PS$  **where**  $PS = p \# ps$   
**define**  $R$  **where**  $R = \text{cbox } (l - \delta, \text{snd } p) (l + \delta, \text{snd } p + \delta)$   
**define**  $RPS$  **where**  $RPS = \{ p \in \text{set } PS. p \in R \}$   
**define**  $LSQ$  **where**  $LSQ = \text{cbox } (l - \delta, \text{snd } p) (l, \text{snd } p + \delta)$   
**define**  $LSQPS$  **where**  $LSQPS = \{ p \in ps_L. p \in LSQ \}$   
**define**  $RSQ$  **where**  $RSQ = \text{cbox } (l, \text{snd } p) (l + \delta, \text{snd } p + \delta)$   
**define**  $RSQPS$  **where**  $RSQPS = \{ p \in ps_R. p \in RSQ \}$   
**note**  $\text{defs} = PS\text{-def } R\text{-def } RPS\text{-def } LSQ\text{-def } LSQPS\text{-def } RSQ\text{-def } RSQPS\text{-def}$

**have**  $R = LSQ \cup RSQ$   
**using**  $\text{defs cbox-right-un}$  **by** *auto*  
**moreover** **have**  $\forall p \in ps_L. p \in RSQ \longrightarrow p \in LSQ$   
**using**  $RSQ\text{-def } LSQ\text{-def assms}(6)$  **by** *auto*  
**moreover** **have**  $\forall p \in ps_R. p \in LSQ \longrightarrow p \in RSQ$   
**using**  $RSQ\text{-def } LSQ\text{-def assms}(7)$  **by** *auto*  
**ultimately** **have**  $RPS = LSQPS \cup RSQPS$   
**using**  $LSQPS\text{-def } RSQPS\text{-def } PS\text{-def } RPS\text{-def assms}(4)$  **by** *blast*

**have**  $\text{sparse } \delta \ LSQPS$   
**using**  $\text{assms}(8) \ LSQPS\text{-def } \text{sparse-def}$  **by** *simp*  
**hence**  $CLSQPS: \text{card } LSQPS \leq 4$   
**using**  $\text{max-points-square}$ [of  $LSQPS \ l - \delta \ \text{snd } p \ \delta$ ]  $\text{assms}(3) \ LSQ\text{-def } LSQPS\text{-def}$   
**by** *auto*

**have**  $\text{sparse } \delta \ RSQPS$   
**using**  $\text{assms}(9) \ RSQPS\text{-def } \text{sparse-def}$  **by** *simp*  
**hence**  $CRSQPS: \text{card } RSQPS \leq 4$   
**using**  $\text{max-points-square}$ [of  $RSQPS \ l \ \text{snd } p \ \delta$ ]  $\text{assms}(3) \ RSQ\text{-def } RSQPS\text{-def}$   
**by** *auto*

**have**  $CRPS$ :  $\text{card } RPS \leq 8$   
**using**  $CLSQPS$   $CRSQPS$   $\text{card-Un-le[of } LSQPS \text{ } RSQPS]$   $\langle RPS = LSQPS \cup RSQPS \rangle$  **by** *auto*

**have**  $\text{set } (p \# \text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) \text{ } ps) \subseteq RPS$   
**proof** *standard*  
**fix**  $q$   
**assume**  $*$ :  $q \in \text{set } (p \# \text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) \text{ } ps)$   
**hence**  $CPS$ :  $q \in \text{set } PS$   
**using**  $PS\text{-def}$  **by** *auto*  
**hence**  $\text{snd } p \leq \text{snd } q$   $\text{snd } q \leq \text{snd } p + \delta$   
**using**  $\text{assms}(2,3)$   $PS\text{-def}$   $\text{sorted-snd-def}$   $*$  **by**  $(\text{auto split: if-splits})$   
**moreover** **have**  $l - \delta < \text{fst } q$   $\text{fst } q < l + \delta$   
**using**  $CPS$   $\text{assms}(5)$   $PS\text{-def}$  **by**  $\text{blast+}$   
**ultimately** **have**  $q \in R$   
**using**  $R\text{-def}$   $\text{mem-cbox-2D[of } l - \delta \text{ } \text{fst } q \text{ } l + \delta \text{ } \text{snd } p \text{ } \text{snd } q \text{ } \text{snd } p + \delta]$   
**by**  $(\text{simp add: prod.case-eq-if})$   
**thus**  $q \in RPS$   
**using**  $CPS$   $RPS\text{-def}$  **by** *simp*  
**qed**  
**moreover** **have**  $\text{finite } RPS$   
**by**  $(\text{simp add: } RPS\text{-def})$   
**ultimately** **have**  $\text{card } (\text{set } (p \# \text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) \text{ } ps)) \leq 8$   
**using**  $CRPS$   $\text{card-mono[of } RPS \text{ } \text{set } (p \# \text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) \text{ } ps)]$   
**by** *simp*  
**moreover** **have**  $\text{distinct } (p \# \text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) \text{ } ps)$   
**using**  $\text{assms}(1)$  **by** *simp*  
**ultimately** **have**  $\text{length } (p \# \text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) \text{ } ps) \leq 8$   
**using**  $\text{assms}(1)$   $PS\text{-def}$   $\text{distinct-card}$  **by** *metis*  
**thus**  $?thesis$   
**by** *simp*  
**qed**

### 2.2.2 Combine Step

**fun**  $t\text{-find-closest} :: \text{point} \Rightarrow \text{real} \Rightarrow \text{point list} \Rightarrow \text{nat}$  **where**  
 $t\text{-find-closest} \text{ - - } [] = 1$   
 $| t\text{-find-closest} \text{ - - } [-] = 1$   
 $| t\text{-find-closest } p \delta (p_0 \# ps) = 1 + (  
\quad \text{if } \delta \leq \text{snd } p_0 - \text{snd } p \text{ then } 0  
\quad \text{else } t\text{-find-closest } p (\min \delta (\text{dist } p \text{ } p_0)) \text{ } ps  
)$

**lemma**  $t\text{-find-closest-eq-time-find-closest-tm}$ :  
 $t\text{-find-closest } p \delta ps = \text{time } (\text{find-closest-tm } p \delta ps)$   
**by**  $(\text{induction } p \delta ps \text{ rule: } t\text{-find-closest.induct})$   
 $(\text{auto simp: time-simps})$

**lemma**  $t\text{-find-closest-mono}$ :

$\delta' \leq \delta \implies t\text{-find-closest } p \ \delta' \ ps \leq t\text{-find-closest } p \ \delta \ ps$   
**by** (induction rule: *t-find-closest.induct*)  
(auto simp: *Let-def min-def*)

**lemma** *t-find-closest-cnt*:

*t-find-closest } p } \delta } ps \leq 1 + \text{length } (\text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) } ps)*

**proof** (induction *p } \delta } ps* rule: *t-find-closest.induct*)

**case** ( $\exists p \ \delta \ p_0 \ p_2 \ ps$ )

**show** *?case*

**proof** (cases  $\delta \leq \text{snd } p_0 - \text{snd } p$ )

**case** *True*

**thus** *?thesis*

**by** *simp*

**next**

**case** *False*

**hence**  $*$ :  $\text{snd } p_0 - \text{snd } p \leq \delta$

**by** *simp*

**have**  $t\text{-find-closest } p \ \delta \ (p_0 \ \# \ p_2 \ \# \ ps) = 1 + t\text{-find-closest } p \ (\text{min } \delta \ (\text{dist } p \ p_0)) \ (p_2 \ \# \ ps)$

**using** *False by simp*

**also have**  $\dots \leq 1 + 1 + \text{length } (\text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \text{min } \delta \ (\text{dist } p \ p_0)) \ (p_2 \ \# \ ps))$

**using** *False } by simp*

**also have**  $\dots \leq 1 + 1 + \text{length } (\text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) \ (p_2 \ \# \ ps))$

**using**  $*$  **by** (*meson add-le-cancel-left length-filter-P-impl-Q min.bounded-iff*)

**also have**  $\dots \leq 1 + \text{length } (\text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) \ (p_0 \ \# \ p_2 \ \# \ ps))$

**using** *False by simp*

**ultimately show** *?thesis*

**by** *simp*

**qed**

**qed** *auto*

**corollary** *t-find-closest-bound*:

**fixes**  $\delta :: \text{real}$  **and**  $p :: \text{point}$  **and**  $ps :: \text{point list}$  **and**  $l :: \text{int}$

**assumes** *distinct } (p } \# } ps) sorted-snd } (p } \# } ps) 0 \leq \delta \ \text{set } (p } \# } ps) = ps\_L \cup ps\_R*

**assumes**  $\forall p' \in \text{set } (p \ \# \ ps). \ l - \delta < \text{fst } p' \wedge \text{fst } p' < l + \delta$

**assumes**  $\forall p \in ps_L. \ \text{fst } p \leq l \ \forall p \in ps_R. \ l \leq \text{fst } p$

**assumes** *sparse } \delta } ps\_L sparse } \delta } ps\_R*

**shows**  $t\text{-find-closest } p \ \delta \ ps \leq \delta$

**using** *assms core-argument[of p ps } \delta } ps\_L ps\_R l] t-find-closest-cnt[of p } \delta } ps]* **by** *linarith*

**fun** *t-find-closest-pair*  $:: (\text{point} * \text{point}) \Rightarrow \text{point list} \Rightarrow \text{nat}$  **where**

*t-find-closest-pair - [] = 1*

| *t-find-closest-pair - [-] = 1*

|  $t\text{-find-closest-pair } (c_0, c_1) \ (p_0 \ \# \ ps) = 1 + ($

*let } p\_1 = \text{find-closest } p\_0 \ (\text{dist } c\_0 \ c\_1) \ ps \ \text{in}*

*t-find-closest } p\_0 \ (\text{dist } c\_0 \ c\_1) \ ps + (*

*if } \text{dist } c\_0 \ c\_1 \leq \text{dist } p\_0 \ p\_1 \ \text{then}*

```

    t-find-closest-pair (c0, c1) ps
  else
    t-find-closest-pair (p0, p1) ps
))

```

**lemma** *t-find-closest-pair-eq-time-find-closest-pair-tm*:

```

t-find-closest-pair (c0, c1) ps = time (find-closest-pair-tm (c0, c1) ps)
by (induction (c0, c1) ps arbitrary: c0 c1 rule: t-find-closest-pair.induct)
(auto simp: time-simps find-closest-eq-val-find-closest-tm t-find-closest-eq-time-find-closest-tm)

```

**lemma** *t-find-closest-pair-bound*:

```

assumes distinct ps sorted-snd ps δ = dist c0 c1 set ps = psL ∪ psR
assumes ∀ p ∈ set ps. l - Δ < fst p ∧ fst p < l + Δ
assumes ∀ p ∈ psL. fst p ≤ l ∀ p ∈ psR. l ≤ fst p
assumes sparse Δ psL sparse Δ psR δ ≤ Δ
shows t-find-closest-pair (c0, c1) ps ≤ 9 * length ps + 1
using assms

```

**proof** (induction (c<sub>0</sub>, c<sub>1</sub>) ps arbitrary: δ c<sub>0</sub> c<sub>1</sub> ps<sub>L</sub> ps<sub>R</sub> rule: t-find-closest-pair.induct)

```

case (∃ c0 c1 p0 p2 ps)

```

```

let ?ps = p2 # ps

```

```

define p1 where p1-def: p1 = find-closest p0 (dist c0 c1) ?ps

```

```

define PSL where PSL-def: PSL = psL - { p0 }

```

```

define PSR where PSR-def: PSR = psR - { p0 }

```

```

note defs = p1-def PSL-def PSR-def

```

```

have *: 0 ≤ Δ

```

```

using 3.premis(3,10) zero-le-dist[of c0 c1] by argo

```

```

hence t-find-closest p0 Δ ?ps ≤ 8

```

```

using t-find-closest-bound[of p0 ?ps Δ psL psR] 3.premis by blast

```

```

hence A: t-find-closest p0 (dist c0 c1) ?ps ≤ 8

```

```

by (metis 3.premis(3,10) order-trans t-find-closest-mono)

```

```

have B: distinct ?ps sorted-snd ?ps

```

```

using 3.premis(1,2) sorted-snd-def by simp-all

```

```

have C: set ?ps = PSL ∪ PSR

```

```

using defs 3.premis(1,4) by auto

```

```

have D: ∀ p ∈ set ?ps. l - Δ < fst p ∧ fst p < l + Δ

```

```

using 3.premis(5) by simp

```

```

have E: ∀ p ∈ PSL. fst p ≤ l ∀ p ∈ PSR. l ≤ fst p

```

```

using defs 3.premis(6,7) by simp-all

```

```

have F: sparse Δ PSL sparse Δ PSR

```

```

using defs 3.premis(8,9) sparse-def by simp-all

```

```

show ?case

```

```

proof (cases dist c0 c1 ≤ dist p0 p1)

```

```

case True

```

```

hence t-find-closest-pair (c0, c1) ?ps ≤ 9 * length ?ps + 1

```

```

using 3.hyps(1) 3.premis(3,10) defs(1) B C D E F by blast

```

```

moreover have t-find-closest-pair (c0, c1) (p0 # ?ps) =

```

```

    1 + t-find-closest p0 (dist c0 c1) ?ps + t-find-closest-pair (c0, c1)

```

```

?ps

```

```

using defs True by (auto split: prod.splits)

```

```

ultimately show ?thesis
  using A by auto
next
case False
moreover have  $0 \leq \text{dist } p_0 \ p_1$ 
  by auto
ultimately have t-find-closest-pair (p0, p1) ?ps ≤ 9 * length ?ps + 1
  using 3.hyps(2) 3.prem(3,10) defs(1) B C D E F by auto
moreover have t-find-closest-pair (c0, c1) (p0 # ?ps) =
  1 + t-find-closest p0 (dist c0 c1) ?ps + t-find-closest-pair (p0, p1)
?ps
  using defs False by (auto split: prod.splits)
ultimately show ?thesis
  using A by simp
qed
qed auto

fun t-combine :: (point * point) ⇒ (point * point) ⇒ int ⇒ point list ⇒ nat where
  t-combine (p0L, p1L) (p0R, p1R) l ps = 1 + (
    let (c0, c1) = if dist p0L p1L < dist p0R p1R then (p0L, p1L) else (p0R, p1R) in
    let ps' = filter (λp. dist p (l, snd p) < dist c0 c1) ps in
    time (filter-tm (λp. dist p (l, snd p) < dist c0 c1) ps) + t-find-closest-pair (c0,
c1) ps'
  )

lemma t-combine-eq-time-combine-tm:
  t-combine (p0L, p1L) (p0R, p1R) l ps = time (combine-tm (p0L, p1L) (p0R, p1R)
l ps)
  by (auto simp: combine-tm.simps time-simps t-find-closest-pair-eq-time-find-closest-pair-tm
filter-eq-val-filter-tm)

lemma t-combine-bound:
  fixes ps :: point list
  assumes distinct ps sorted_snd ps set ps = psL ∪ psR
  assumes ∀ p ∈ psL. fst p ≤ l ∀ p ∈ psR. l ≤ fst p
  assumes sparse (dist p0L p1L) psL sparse (dist p0R p1R) psR
  shows t-combine (p0L, p1L) (p0R, p1R) l ps ≤ 10 * length ps + 3
proof -
  obtain c0 c1 where c-def:
    (c0, c1) = (if dist p0L p1L < dist p0R p1R then (p0L, p1L) else (p0R, p1R)) by
metis
  let ?P = (λp. dist p (l, snd p) < dist c0 c1)
  define ps' where ps'-def: ps' = filter ?P ps
  define psL' where psL'-def: psL' = { p ∈ psL. ?P p }
  define psR' where psR'-def: psR' = { p ∈ psR. ?P p }
  note defs = c-def ps'-def psL'-def psR'-def
  have sparse (dist c0 c1) psL sparse (dist c0 c1) psR
  using asms(6,7) sparse-mono c-def by (auto split: if-splits)
  hence sparse (dist c0 c1) psL' sparse (dist c0 c1) psR'

```

**using**  $ps_L'$ -def  $ps_R'$ -def *sparse-def* **by** *auto*  
**moreover have** *distinct*  $ps'$   
**using**  $ps'$ -def *assms*(1) **by** *simp*  
**moreover have** *sorted-snd*  $ps'$   
**using**  $ps'$ -def *assms*(2) *sorted-snd-def sorted-wrt-filter* **by** *blast*  
**moreover have**  $0 \leq \text{dist } c_0 \ c_1$   
**by** *simp*  
**moreover have** *set*  $ps' = ps_L' \cup ps_R'$   
**using** *assms*(3) *defs*(2,3,4) *filter-Un* **by** *auto*  
**moreover have**  $\forall p \in \text{set } ps'. l - \text{dist } c_0 \ c_1 < \text{fst } p \wedge \text{fst } p < l + \text{dist } c_0 \ c_1$   
**using**  $ps'$ -def *dist-transform* **by** *force*  
**moreover have**  $\forall p \in ps_L'. \text{fst } p \leq l \vee p \in ps_R'. l \leq \text{fst } p$   
**using** *assms*(4,5)  $ps_L'$ -def  $ps_R'$ -def **by** *blast+*  
**ultimately have** *t-find-closest-pair*  $(c_0, c_1) \ ps' \leq 9 * \text{length } ps' + 1$   
**using** *t-find-closest-pair-bound* **by** *blast*  
**moreover have**  $\text{length } ps' \leq \text{length } ps$   
**using**  $ps'$ -def **by** *simp*  
**ultimately have** \*: *t-find-closest-pair*  $(c_0, c_1) \ ps' \leq 9 * \text{length } ps + 1$   
**by** *simp*  
**have** *t-combine*  $(p_{0L}, p_{1L}) \ (p_{0R}, p_{1R}) \ l \ ps =$   
 $1 + \text{time } (\text{filter-tm } ?P \ ps) + \text{t-find-closest-pair } (c_0, c_1) \ ps'$   
**using** *defs* **by** (*auto split: prod.splits*)  
**also have** ... =  $2 + \text{length } ps + \text{t-find-closest-pair } (c_0, c_1) \ ps'$   
**using** *time-filter-tm* **by** *auto*  
**finally show** *?thesis*  
**using** \* **by** *simp*  
**qed**

**declare** *t-combine.simps* [*simp del*]

### 2.2.3 Divide and Conquer Algorithm

**lemma** *time-closest-pair-rec-tm-simps-1*:

**assumes**  $\text{length } xs \leq 3$   
**shows**  $\text{time } (\text{closest-pair-rec-tm } xs) = 1 + \text{time } (\text{length-tm } xs) + \text{time } (\text{mergesort-tm } \text{snd } xs) + \text{time } (\text{closest-pair-bf-tm } xs)$   
**using** *assms* **by** (*auto simp: time-simps length-eq-val-length-tm*)

**lemma** *time-closest-pair-rec-tm-simps-2*:

**assumes**  $\neg (\text{length } xs \leq 3)$   
**shows**  $\text{time } (\text{closest-pair-rec-tm } xs) = 1 + ($   
 $\text{let } (xs_L, xs_R) = \text{val } (\text{split-at-tm } (\text{length } xs \ \text{div } 2) \ xs) \ \text{in}$   
 $\text{let } (ys_L, p_L) = \text{val } (\text{closest-pair-rec-tm } xs_L) \ \text{in}$   
 $\text{let } (ys_R, p_R) = \text{val } (\text{closest-pair-rec-tm } xs_R) \ \text{in}$   
 $\text{let } ys = \text{val } (\text{merge-tm } (\lambda p. \ \text{snd } p) \ ys_L \ ys_R) \ \text{in}$   
 $\text{time } (\text{length-tm } xs) + \text{time } (\text{split-at-tm } (\text{length } xs \ \text{div } 2) \ xs) + \text{time } (\text{closest-pair-rec-tm } xs_L) +$   
 $\text{time } (\text{closest-pair-rec-tm } xs_R) + \text{time } (\text{merge-tm } (\lambda p. \ \text{snd } p) \ ys_L \ ys_R) +$   
 $\text{t-combine } p_L \ p_R \ (\text{fst } (\text{hd } xs_R)) \ ys$

```

)
using assms
apply (subst closest-pair-rec-tm.simps)
by (auto simp del: closest-pair-rec-tm.simps
      simp add: time-simps length-eq-val-length-tm t-combine-eq-time-combine-tm
      split: prod.split)

function closest-pair-recurrence :: nat  $\Rightarrow$  real where
  n  $\leq 3 \implies$  closest-pair-recurrence n = 3 + n + mergesort-recurrence n + n * n
| 3 < n  $\implies$  closest-pair-recurrence n = 7 + 13 * n +
  closest-pair-recurrence (nat  $\lfloor$  real n / 2  $\rfloor$ ) + closest-pair-recurrence (nat  $\lceil$  real n
/ 2  $\rceil$ )
by force simp-all
termination by akra-bazzi-termination simp-all

lemma closest-pair-recurrence-nonneg[simp]:
  0  $\leq$  closest-pair-recurrence n
by (induction n rule: closest-pair-recurrence.induct) auto

lemma time-closest-pair-rec-conv-closest-pair-recurrence:
assumes distinct ps sorted-fst ps
shows time (closest-pair-rec-tm ps)  $\leq$  closest-pair-recurrence (length ps)
using assms
proof (induction ps rule: length-induct)
case (1 ps)
let ?n = length ps
show ?case
proof (cases ?n  $\leq$  3)
case True
hence time (closest-pair-rec-tm ps) = 1 + time (length-tm ps) + time (mergesort-tm
snd ps) + time (closest-pair-bf-tm ps)
using time-closest-pair-rec-tm-simps-1 by simp
moreover have closest-pair-recurrence ?n = 3 + ?n + mergesort-recurrence
?n + ?n * ?n
using True by simp
moreover have time (length-tm ps)  $\leq$  1 + ?n time (mergesort-tm snd ps)  $\leq$ 
mergesort-recurrence ?n
      time (closest-pair-bf-tm ps)  $\leq$  1 + ?n * ?n
using time-length-tm[of ps] time-mergesort-conv-mergesort-recurrence[of snd
ps] time-closest-pair-bf-tm[of ps] by auto
ultimately show ?thesis
by linarith
next
case False

obtain XSL XSR where XS-def: (XSL, XSR) = val (split-at-tm (?n div 2)
ps)
using prod.collapse by blast
obtain YSL C0L C1L where CPL-def: (YSL, C0L, C1L) = val (closest-pair-rec-tm

```

$XS_L$ )  
**using** *prod.collapse by metis*  
**obtain**  $YS_R C_{0R} C_{1R}$  **where**  $CP_R\text{-def}: (YS_R, C_{0R}, C_{1R}) = \text{val} (\text{closest-pair-rec-tm } XS_R)$   
**using** *prod.collapse by metis*  
**define**  $YS$  **where**  $YS = \text{val} (\text{merge-tm } (\lambda p. \text{snd } p) YS_L YS_R)$   
**obtain**  $C_0 C_1$  **where**  $C_{01}\text{-def}: (C_0, C_1) = \text{val} (\text{combine-tm } (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) (\text{fst } (\text{hd } XS_R)) YS)$   
**using** *prod.collapse by metis*  
**note**  $\text{defs} = XS\text{-def } CP_L\text{-def } CP_R\text{-def } YS\text{-def } C_{01}\text{-def}$   
  
**have**  $XSLR: XS_L = \text{take } (?n \text{ div } 2) \text{ ps } XS_R = \text{drop } (?n \text{ div } 2) \text{ ps}$   
**using**  $\text{defs}$  **by** *(auto simp: split-at-take-drop-conv split-at-eq-val-split-at-tm)*  
**hence**  $\text{length } XS_L = ?n \text{ div } 2 \text{ length } XS_R = ?n - ?n \text{ div } 2$   
**by** *simp-all*  
**hence**  $*$ :  $(\text{nat } \lfloor \text{real } ?n / 2 \rfloor) = \text{length } XS_L (\text{nat } \lceil \text{real } ?n / 2 \rceil) = \text{length } XS_R$   
**by** *linarith+*  
**have**  $\text{length } XS_L = \text{length } YS_L \text{ length } XS_R = \text{length } YS_R$   
**using**  $\text{defs}$  *closest-pair-rec-set-length-sorted-snd closest-pair-rec-eq-val-closest-pair-rec-tm*  
**by** *metis+*  
**hence**  $L: ?n = \text{length } YS_L + \text{length } YS_R$   
**using**  $\text{defs}$   $XSLR$  **by** *fastforce*  
  
**have**  $1 < \text{length } XS_L \text{ length } XS_L < \text{length } \text{ps}$   
**using** *False XSLR by simp-all*  
**moreover** **have** *distinct*  $XS_L$  *sorted-fst*  $XS_L$   
**using**  $XSLR$   $1.\text{prems}(1,2)$  *sorted-fst-def sorted-wrt-take* **by** *simp-all*  
**ultimately** **have**  $\text{time } (\text{closest-pair-rec-tm } XS_L) \leq \text{closest-pair-recurrence } (\text{length } XS_L)$   
**using**  $1.IH$  **by** *simp*  
**hence**  $IHL: \text{time } (\text{closest-pair-rec-tm } XS_L) \leq \text{closest-pair-recurrence } (\text{nat } \lfloor \text{real } ?n / 2 \rfloor)$   
**using**  $*$  **by** *simp*  
  
**have**  $1 < \text{length } XS_R \text{ length } XS_R < \text{length } \text{ps}$   
**using** *False XSLR by simp-all*  
**moreover** **have** *distinct*  $XS_R$  *sorted-fst*  $XS_R$   
**using**  $XSLR$   $1.\text{prems}(1,2)$  *sorted-fst-def sorted-wrt-drop* **by** *simp-all*  
**ultimately** **have**  $\text{time } (\text{closest-pair-rec-tm } XS_R) \leq \text{closest-pair-recurrence } (\text{length } XS_R)$   
**using**  $1.IH$  **by** *simp*  
**hence**  $IHR: \text{time } (\text{closest-pair-rec-tm } XS_R) \leq \text{closest-pair-recurrence } (\text{nat } \lceil \text{real } ?n / 2 \rceil)$   
**using**  $*$  **by** *simp*  
  
**have**  $(YS, C_0, C_1) = \text{val} (\text{closest-pair-rec-tm } \text{ps})$   
**using** *False closest-pair-rec-simps*  $\text{defs}$  **by** *(auto simp: Let-def length-eq-val-length-tm split!: prod.split)*  
**hence**  $\text{set } \text{ps} = \text{set } YS \text{ length } \text{ps} = \text{length } YS \text{ distinct } YS \text{ sorted-snd } YS$

**using** *1.prem*s *closest-pair-rec-set-length-sorted-snd* *closest-pair-rec-distinct*  
*closest-pair-rec-eq-val-closest-pair-rec-tm* **by** *auto*  
**moreover have**  $\forall p \in \text{set } YS_L. \text{fst } p \leq \text{fst } (\text{hd } XS_R)$   
**using** *False 1.prem*s(2) *XSLR*  $\langle \text{length } XS_L < \text{length } ps \rangle \langle \text{length } XS_L = \text{length } ps \text{ div } 2 \rangle$   
*CP<sub>L</sub>-def* *sorted-fst-take-less-hd-drop* *closest-pair-rec-set-length-sorted-snd*  
*closest-pair-rec-eq-val-closest-pair-rec-tm* **by** *metis*  
**moreover have**  $\forall p \in \text{set } YS_R. \text{fst } (\text{hd } XS_R) \leq \text{fst } p$   
**using** *False 1.prem*s(2) *XSLR* *CP<sub>R</sub>-def* *sorted-fst-hd-drop-less-drop*  
*closest-pair-rec-set-length-sorted-snd* *closest-pair-rec-eq-val-closest-pair-rec-tm*  
**by** *metis*  
**moreover have**  $\text{set } YS = \text{set } YS_L \cup \text{set } YS_R$   
**using** *set-merge* *defs* **by** (*metis* *merge-eq-val-merge-tm*)  
**moreover have** *sparse* (*dist* *C<sub>0L</sub>* *C<sub>1L</sub>*) (*set* *YS<sub>L</sub>*)  
**using** *CP<sub>L</sub>-def*  $\langle 1 < \text{length } XS_L \rangle \langle \text{distinct } XS_L \rangle \langle \text{sorted-fst } XS_L \rangle$   
*closest-pair-rec-dist* *closest-pair-rec-set-length-sorted-snd*  
*closest-pair-rec-eq-val-closest-pair-rec-tm* **by** *auto*  
**moreover have** *sparse* (*dist* *C<sub>0R</sub>* *C<sub>1R</sub>*) (*set* *YS<sub>R</sub>*)  
**using** *CP<sub>R</sub>-def*  $\langle 1 < \text{length } XS_R \rangle \langle \text{distinct } XS_R \rangle \langle \text{sorted-fst } XS_R \rangle$   
*closest-pair-rec-dist* *closest-pair-rec-set-length-sorted-snd*  
*closest-pair-rec-eq-val-closest-pair-rec-tm* **by** *auto*  
**ultimately have** *combine-bound*: *t-combine* (*C<sub>0L</sub>*, *C<sub>1L</sub>*) (*C<sub>0R</sub>*, *C<sub>1R</sub>*) (*fst* (*hd* *XS<sub>R</sub>*))  $YS \leq 3 + 10 * ?n$   
**using** *t-combine-bound*[*of* *YS* *set* *YS<sub>L</sub>* *set* *YS<sub>R</sub>* *fst* (*hd* *XS<sub>R</sub>*)] **by** (*simp* *add*:  
*add.commute*)  
**have**  $\text{time } (\text{closest-pair-rec-tm } ps) = 1 + \text{time } (\text{length-tm } ps) + \text{time } (\text{split-at-tm } (?n \text{ div } 2) ps) +$   
 $\text{time } (\text{closest-pair-rec-tm } XS_L) + \text{time } (\text{closest-pair-rec-tm } XS_R) + \text{time } (\text{merge-tm } (\lambda p. \text{snd } p) YS_L YS_R) +$   
 $\text{t-combine } (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) (\text{fst } (\text{hd } XS_R)) YS$   
**using** *time-closest-pair-rec-tm-simps-2*[*OF* *False*] *defs*  
**by** (*auto* *simp* *del*: *closest-pair-rec-tm.simps* *simp* *add*: *Let-def* *split*: *prod.split*)  
**also have**  $\dots \leq 7 + 13 * ?n + \text{time } (\text{closest-pair-rec-tm } XS_L) + \text{time } (\text{closest-pair-rec-tm } XS_R)$   
**using** *time-merge-tm*[*of* ( $\lambda p. \text{snd } p$ ) *YS<sub>L</sub>* *YS<sub>R</sub>*] *L* *combine-bound* **by** (*simp* *add*: *time-length-tm* *time-split-at-tm*)  
**also have**  $\dots \leq 7 + 13 * ?n + \text{closest-pair-recurrence } (\text{nat } \lfloor \text{real } ?n / 2 \rfloor) +$   
 $\text{closest-pair-recurrence } (\text{nat } \lceil \text{real } ?n / 2 \rceil)$   
**using** *IHL* *IHR* **by** *simp*  
**also have**  $\dots = \text{closest-pair-recurrence } (\text{length } ps)$   
**using** *False* **by** *simp*  
**finally show** *?thesis*  
**by** *simp*  
**qed**  
**qed**

**theorem** *closest-pair-recurrence*:  
 $\text{closest-pair-recurrence} \in \Theta(\lambda n. n * \ln n)$   
**by** (*master-theorem*) *auto*

**theorem** *time-closest-pair-rec-bigo*:  
 $(\lambda xs. \text{time } (\text{closest-pair-rec-tm } xs)) \in O[\text{length going-to at-top within } \{ ps. \text{distinct } ps \wedge \text{sorted-fst } ps \}][(\lambda n. n * \ln n) \text{ o length}]$   
**proof** –  
**have**  $0: \bigwedge ps. ps \in \{ ps. \text{distinct } ps \wedge \text{sorted-fst } ps \} \implies$   
 $\text{time } (\text{closest-pair-rec-tm } ps) \leq (\text{closest-pair-recurrence o length}) ps$   
**unfolding** *comp-def* **using** *time-closest-pair-rec-conv-closest-pair-recurrence* **by**  
*auto*  
**show** *?thesis*  
**using** *bigo-measure-trans[OF 0] bigthetaD1[OF closest-pair-recurrence] of-nat-0-le-iff*  
**by** *blast*  
**qed**

**definition** *closest-pair-time* ::  $\text{nat} \Rightarrow \text{real}$  **where**  
 $\text{closest-pair-time } n = 1 + \text{mergesort-recurrence } n + \text{closest-pair-recurrence } n$

**lemma** *time-closest-pair-conv-closest-pair-recurrence*:  
**assumes** *distinct ps*  
**shows**  $\text{time } (\text{closest-pair-tm } ps) \leq \text{closest-pair-time } (\text{length } ps)$   
**using** *assms*  
**unfolding** *closest-pair-time-def*  
**proof** (*induction rule: induct-list012*)  
**case**  $(\exists x y zs)$   
**let**  $?ps = x \# y \# zs$   
**define**  $xs$  **where**  $xs = \text{val } (\text{mergesort-tm fst } ?ps)$   
**have**  $*$ :  $\text{distinct } xs \text{ sorted-fst } xs \text{ length } xs = \text{length } ?ps$   
**using**  $xs\text{-def mergesort}(4)[\text{OF } 3.\text{prems}, \text{ of fst}] \text{ mergesort}(1)[\text{of fst } ?ps] \text{ merge-sort}(3)[\text{of fst } ?ps]$   
 $\text{sorted-fst-def mergesort-eq-val-mergesort-tm}$  **by** *metis+*  
**have**  $\text{time } (\text{closest-pair-tm } ?ps) = 1 + \text{time } (\text{mergesort-tm fst } ?ps) + \text{time } (\text{closest-pair-rec-tm } xs)$   
**using**  $xs\text{-def}$  **by** (*auto simp del: mergesort-tm.simps closest-pair-rec-tm.simps simp add: time-simps split: prod.split*)  
**also have**  $\dots \leq 1 + \text{mergesort-recurrence } (\text{length } ?ps) + \text{time } (\text{closest-pair-rec-tm } xs)$   
**using** *time-mergesort-conv-mergesort-recurrence[of fst ?ps]* **by** *simp*  
**also have**  $\dots \leq 1 + \text{mergesort-recurrence } (\text{length } ?ps) + \text{closest-pair-recurrence } (\text{length } ?ps)$   
**using** *time-closest-pair-rec-conv-closest-pair-recurrence[of xs]* **by** *auto*  
**finally show** *?case*  
**by** *blast*  
**qed** (*auto simp: time-simps*)

**corollary** *closest-pair-time*:  
 $\text{closest-pair-time} \in O(\lambda n. n * \ln n)$   
**unfolding** *closest-pair-time-def*  
**using** *mergesort-recurrence closest-pair-recurrence sum-in-bigo(1) const-1-bigo-n-ln-n*  
**by** *blast*

**corollary** *time-closest-pair-bigo*:

$(\lambda ps. \text{time } (\text{closest-pair-tm } ps)) \in O[\text{length going-to at-top within } \{ ps. \text{distinct } ps \}](\lambda n. n * \ln n) o \text{ length})$

**proof** –

**have**  $0: \bigwedge ps. ps \in \{ ps. \text{distinct } ps \} \implies$

$\text{time } (\text{closest-pair-tm } ps) \leq (\text{closest-pair-time } o \text{ length}) ps$

**unfolding** *comp-def* **using** *time-closest-pair-conv-closest-pair-recurrence* **by**

*auto*

**show** *?thesis*

**using** *bigo-measure-trans[OF 0]* *closest-pair-time* **by** *fastforce*

**qed**

## 2.3 Code Export

### 2.3.1 Combine Step

**fun** *find-closest-code* :: *point*  $\Rightarrow$  *int*  $\Rightarrow$  *point list*  $\Rightarrow$  (*int* \* *point*) **where**

*find-closest-code* - - [] = *undefined*

| *find-closest-code* *p* - [p<sub>0</sub>] = (*dist-code* *p* p<sub>0</sub>, p<sub>0</sub>)

| *find-closest-code* *p*  $\delta$  (p<sub>0</sub> # *ps*) = (

*let*  $\delta_0 = \text{dist-code } p \ p_0$  *in*

*if*  $\delta \leq (\text{snd } p_0 - \text{snd } p)^2$  *then*

( $\delta_0$ , p<sub>0</sub>)

*else*

*let* ( $\delta_1$ , p<sub>1</sub>) = *find-closest-code* *p* (*min*  $\delta$   $\delta_0$ ) *ps* *in*

*if*  $\delta_0 \leq \delta_1$  *then*

( $\delta_0$ , p<sub>0</sub>)

*else*

( $\delta_1$ , p<sub>1</sub>)

)

**lemma** *find-closest-code-dist-eq*:

$0 < \text{length } ps \implies (\delta_c, c) = \text{find-closest-code } p \ \delta \ ps \implies \delta_c = \text{dist-code } p \ c$

**proof** (*induction* *p*  $\delta$  *ps* *arbitrary*:  $\delta_c \ c$  *rule*: *find-closest-code.induct*)

**case** ( $\exists p \ \delta \ p_0 \ p_2 \ ps$ )

**show** *?case*

**proof** *cases*

**assume**  $\delta \leq (\text{snd } p_0 - \text{snd } p)^2$

**thus** *?thesis*

**using** *3.premis(2)* **by** *simp*

**next**

**assume** *A*:  $\neg \delta \leq (\text{snd } p_0 - \text{snd } p)^2$

**define**  $\delta_0$  **where**  $\delta_0\text{-def}$ :  $\delta_0 = \text{dist-code } p \ p_0$

**obtain**  $\delta_1 \ p_1$  **where**  $\delta_1\text{-def}$ : ( $\delta_1$ , p<sub>1</sub>) = *find-closest-code* *p* (*min*  $\delta$   $\delta_0$ ) (p<sub>2</sub> # *ps*)

**by** (*metis surj-pair*)

**note** *defs* =  $\delta_0\text{-def}$   $\delta_1\text{-def}$

**have**  $\delta_1 = \text{dist-code } p \ p_1$

**using** *3.IH[of  $\delta_0 \ \delta_1 \ p_1$ ]* *A* *defs* **by** *simp*

**thus** *?thesis*

```

    using defs 3.prem1 by (auto simp: Let-def split: if-splits prod.splits)
  qed
qed simp-all

declare find-closest.simps [simp add]

lemma find-closest-code-eq:
  assumes 0 < length ps  $\delta = \text{dist } c_0 \ c_1 \ \delta' = \text{dist-code } c_0 \ c_1 \ \text{sorted-snd } (p \ \# \ ps)$ 
  assumes  $c = \text{find-closest } p \ \delta \ ps \ (\delta_c', \ c') = \text{find-closest-code } p \ \delta' \ ps$ 
  shows  $c = c'$ 
  using assms
proof (induction p  $\delta \ ps$  arbitrary:  $\delta' \ c_0 \ c_1 \ c \ \delta_c' \ c'$  rule: find-closest.induct)
  case (3 p  $\delta \ p_0 \ p_2 \ ps$ )
  define  $\delta_0 \ \delta_0'$  where  $\delta_0$ -def:  $\delta_0 = \text{dist } p \ p_0 \ \delta_0' = \text{dist-code } p \ p_0$ 
  obtain  $p_1 \ \delta_1' \ p_1'$  where  $\delta_1$ -def:  $p_1 = \text{find-closest } p \ (\min \ \delta \ \delta_0) \ (p_2 \ \# \ ps)$ 
    ( $\delta_1', \ p_1'$ ) = find-closest-code  $p \ (\min \ \delta' \ \delta_0') \ (p_2 \ \# \ ps)$ 
    by (metis surj-pair)
  note defs =  $\delta_0$ -def  $\delta_1$ -def
  show ?case
  proof cases
    assume *:  $\delta \leq \text{snd } p_0 - \text{snd } p$ 
    hence  $\delta' \leq (\text{snd } p_0 - \text{snd } p)^2$ 
      using 3.prem1(2,3) dist-eq-dist-code-abs-le by fastforce
    thus ?thesis
      using * 3.prem1(5,6) by simp
  next
    assume *:  $\neg \delta \leq \text{snd } p_0 - \text{snd } p$ 
    moreover have  $0 \leq \text{snd } p_0 - \text{snd } p$ 
      using 3.prem1(4) sorted-snd-def by simp
    ultimately have A:  $\neg \delta' \leq (\text{snd } p_0 - \text{snd } p)^2$ 
      using 3.prem1(2,3) dist-eq-dist-code-abs-le[of  $c_0 \ c_1 \ \text{snd } p_0 - \text{snd } p$ ] by simp
    have  $\min \ \delta \ \delta_0 = \delta \iff \min \ \delta' \ \delta_0' = \delta' \ \min \ \delta \ \delta_0 = \delta_0 \iff \min \ \delta' \ \delta_0' = \delta_0'$ 
      by (metis 3.prem1(2,3) defs(1,2) dist-eq-dist-code-le min.commute min-def)+
    moreover have sorted-snd  $(p \ \# \ p_2 \ \# \ ps)$ 
      using 3.prem1(4) sorted-snd-def by simp
    ultimately have B:  $p_1 = p_1'$ 
      using 3.IH[of  $c_0 \ c_1 \ \delta' \ p_1 \ \delta_1' \ p_1'$ ] 3.IH[of  $p \ p_0 \ \delta_0' \ p_1 \ \delta_1' \ p_1'$ ] 3.prem1(2,3)
      defs * by auto
    have  $\delta_1' = \text{dist-code } p \ p_1'$ 
      using find-closest-code-dist-eq defs by blast
    hence  $\delta_0 \leq \text{dist } p \ p_1 \iff \delta_0' \leq \delta_1'$ 
      using defs(1,2) dist-eq-dist-code-le by (simp add: B)
    thus ?thesis
      using 3.prem1(5,6) * A B defs by (auto simp: Let-def split: prod.splits)
  qed
qed auto

fun find-closest-pair-code :: (int * point * point)  $\Rightarrow$  point list  $\Rightarrow$  (int * point * point) where

```

```

    find-closest-pair-code ( $\delta$ ,  $c_0$ ,  $c_1$ ) [] = ( $\delta$ ,  $c_0$ ,  $c_1$ )
  | find-closest-pair-code ( $\delta$ ,  $c_0$ ,  $c_1$ ) [p] = ( $\delta$ ,  $c_0$ ,  $c_1$ )
  | find-closest-pair-code ( $\delta$ ,  $c_0$ ,  $c_1$ ) (p0 # ps) = (
    let ( $\delta'$ , p1) = find-closest-code p0  $\delta$  ps in
    if  $\delta \leq \delta'$  then
      find-closest-pair-code ( $\delta$ ,  $c_0$ ,  $c_1$ ) ps
    else
      find-closest-pair-code ( $\delta'$ , p0, p1) ps
  )

```

**lemma** *find-closest-pair-code-dist-eq*:

**assumes**  $\delta = \text{dist-code } c_0 \ c_1 \ (\Delta, C_0, C_1) = \text{find-closest-pair-code } (\delta, c_0, c_1) \ ps$

**shows**  $\Delta = \text{dist-code } C_0 \ C_1$

**using** *assms*

**proof** (*induction* ( $\delta$ ,  $c_0$ ,  $c_1$ ) ps *arbitrary*:  $\delta \ c_0 \ c_1 \ \Delta \ C_0 \ C_1$  *rule*: *find-closest-pair-code.induct*)

**case** ( $\exists \ \delta \ c_0 \ c_1 \ p_0 \ p_2 \ ps$ )

**obtain**  $\delta' \ p_1$  **where**  $\delta'$ -def: ( $\delta'$ , p<sub>1</sub>) = *find-closest-code* p<sub>0</sub>  $\delta$  (p<sub>2</sub> # ps)

**by** (*metis surj-pair*)

**hence**  $A$ :  $\delta' = \text{dist-code } p_0 \ p_1$

**using** *find-closest-code-dist-eq* **by** *blast*

**show** *?case*

**proof** (*cases*  $\delta \leq \delta'$ )

**case** *True*

**obtain**  $\Delta' \ C_0' \ C_1'$  **where**  $\Delta'$ -def: ( $\Delta'$ , C<sub>0</sub>', C<sub>1</sub>') = *find-closest-pair-code* ( $\delta$ , c<sub>0</sub>, c<sub>1</sub>) (p<sub>2</sub> # ps)

**by** (*metis prod-cases4*)

**note** *defs* =  $\delta'$ -def  $\Delta'$ -def

**hence**  $\Delta' = \text{dist-code } C_0' \ C_1'$

**using**  $\exists$ .*hyps*(1)[of ( $\delta'$ , p<sub>1</sub>)  $\delta'$  p<sub>1</sub>]  $\exists$ .*prems*(1) *True*  $\delta'$ -def **by** *blast*

**moreover** **have**  $\Delta = \Delta' \ C_0 = C_0' \ C_1 = C_1'$

**using** *defs* *True*  $\exists$ .*prems*(2) **apply** (*auto split: prod.splits*) **by** (*metis Pair-inject*)+

**ultimately** **show** *?thesis*

**by** *simp*

**next**

**case** *False*

**obtain**  $\Delta' \ C_0' \ C_1'$  **where**  $\Delta'$ -def: ( $\Delta'$ , C<sub>0</sub>', C<sub>1</sub>') = *find-closest-pair-code* ( $\delta'$ , p<sub>0</sub>, p<sub>1</sub>) (p<sub>2</sub> # ps)

**by** (*metis prod-cases4*)

**note** *defs* =  $\delta'$ -def  $\Delta'$ -def

**hence**  $\Delta' = \text{dist-code } C_0' \ C_1'$

**using**  $\exists$ .*hyps*(2)[of ( $\delta'$ , p<sub>1</sub>)  $\delta'$  p<sub>1</sub>] *A* *False*  $\delta'$ -def **by** *blast*

**moreover** **have**  $\Delta = \Delta' \ C_0 = C_0' \ C_1 = C_1'$

**using** *defs* *False*  $\exists$ .*prems*(2) **apply** (*auto split: prod.splits*) **by** (*metis Pair-inject*)+

**ultimately** **show** *?thesis*

**by** *simp*

**qed**

**qed** *auto*

**declare** *find-closest-pair.simps* [*simp add*]

**lemma** *find-closest-pair-code-eq*:

**assumes**  $\delta = \text{dist } c_0 \ c_1 \ \delta' = \text{dist-code } c_0 \ c_1 \ \text{sorted-snd } ps$   
**assumes**  $(C_0, C_1) = \text{find-closest-pair } (c_0, c_1) \ ps$   
**assumes**  $(\Delta', C_0', C_1') = \text{find-closest-pair-code } (\delta', c_0, c_1) \ ps$   
**shows**  $C_0 = C_0' \wedge C_1 = C_1'$   
**using** *assms*

**proof** (*induction*  $(c_0, c_1) \ ps$  *arbitrary*:  $\delta \ \delta' \ c_0 \ c_1 \ C_0 \ C_1 \ \Delta' \ C_0' \ C_1'$  *rule*: *find-closest-pair.induct*)

**case**  $(\exists \ c_0 \ c_1 \ p_0 \ p_2 \ ps)$

**obtain**  $p_1 \ \delta_p' \ p_1'$  **where**  $\delta_p\text{-def}$ :  $p_1 = \text{find-closest } p_0 \ \delta \ (p_2 \ \# \ ps)$   
 $(\delta_p', p_1') = \text{find-closest-code } p_0 \ \delta' \ (p_2 \ \# \ ps)$   
**by** (*metis surj-pair*)

**hence**  $A$ :  $\delta_p' = \text{dist-code } p_0 \ p_1'$

**using** *find-closest-code-dist-eq* **by** *blast*

**have**  $B$ :  $p_1 = p_1'$

**using**  $\exists$ .*prems*(1,2,3)  $\delta_p\text{-def}$  *find-closest-code-eq* **by** *blast*

**show** *?case*

**proof** (*cases*  $\delta \leq \text{dist } p_0 \ p_1$ )

**case** *True*

**hence**  $C$ :  $\delta' \leq \delta_p'$

**by** (*simp add*:  $\exists$ .*prems*(1,2)  $A \ B$  *dist-eq-dist-code-le*)

**obtain**  $C_{0i} \ C_{1i} \ \Delta_i' \ C_{0i}' \ C_{1i}'$  **where**  $\Delta_i\text{-def}$ :

$(C_{0i}, C_{1i}) = \text{find-closest-pair } (c_0, c_1) \ (p_2 \ \# \ ps)$

$(\Delta_i', C_{0i}', C_{1i}') = \text{find-closest-pair-code } (\delta', c_0, c_1) \ (p_2 \ \# \ ps)$

**by** (*metis prod-cases3*)

**note**  $\text{defs} = \delta_p\text{-def } \Delta_i\text{-def}$

**have** *sorted-snd*  $(p_2 \ \# \ ps)$

**using**  $\exists$ .*prems*(3) *sorted-snd-def* **by** *simp*

**hence**  $C_{0i} = C_{0i}' \wedge C_{1i} = C_{1i}'$

**using**  $\exists$ .*hyps*(1)  $\exists$ .*prems*(1,2) *True*  $\text{defs}$  **by** *blast*

**moreover** **have**  $C_0 = C_{0i} \ C_1 = C_{1i}$

**using**  $\text{defs}$ (1,3) *True*  $\exists$ .*prems*(1,4) **apply** (*auto split: prod.splits*) **by** (*metis*

*Pair-inject*)+

**moreover** **have**  $\Delta' = \Delta_i' \ C_0' = C_{0i}' \ C_1' = C_{1i}'$

**using**  $\text{defs}$ (2,4)  $C$   $\exists$ .*prems*(5) **apply** (*auto split: prod.splits*) **by** (*metis*

*Pair-inject*)+

**ultimately** **show** *?thesis*

**by** *simp*

**next**

**case** *False*

**hence**  $C$ :  $\neg \delta' \leq \delta_p'$

**by** (*simp add*:  $\exists$ .*prems*(1,2)  $A \ B$  *dist-eq-dist-code-le*)

**obtain**  $C_{0i} \ C_{1i} \ \Delta_i' \ C_{0i}' \ C_{1i}'$  **where**  $\Delta_i\text{-def}$ :

$(C_{0i}, C_{1i}) = \text{find-closest-pair } (p_0, p_1) \ (p_2 \ \# \ ps)$

$(\Delta_i', C_{0i}', C_{1i}') = \text{find-closest-pair-code } (\delta_p', p_0, p_1') \ (p_2 \ \# \ ps)$

**by** (*metis prod-cases3*)

**note**  $\text{defs} = \delta_p\text{-def } \Delta_i\text{-def}$

**have** *sorted-snd*  $(p_2 \ \# \ ps)$

**using**  $\exists$ .*prems*(3) *sorted-snd-def* **by** *simp*

**hence**  $C_{0i} = C_{0i}' \wedge C_{1i} = C_{1i}'$   
**using**  $\mathcal{P}.prems(1)$   $\mathcal{P}.hyps(2)$   $A B$  *False* **defs** **by** *blast*  
**moreover** **have**  $C_0 = C_{0i}$   $C_1 = C_{1i}$   
**using**  $defs(1,3)$  *False*  $\mathcal{P}.prems(1,4)$  **apply** (*auto split: prod.splits*) **by** (*metis*  
*Pair-inject*)  
**moreover** **have**  $\Delta' = \Delta_i'$   $C_0' = C_{0i}'$   $C_1' = C_{1i}'$   
**using**  $defs(2,4)$   $C$   $\mathcal{P}.prems(5)$  **apply** (*auto split: prod.splits*) **by** (*metis*  
*Pair-inject*)  
**ultimately** **show** *?thesis*  
**by** *simp*  
**qed**  
**qed** *auto*

**fun** *combine-code* ::  $(int * point * point) \Rightarrow (int * point * point) \Rightarrow int \Rightarrow point$   
 $list \Rightarrow (int * point * point)$  **where**  
*combine-code*  $(\delta_L, p_{0L}, p_{1L}) (\delta_R, p_{0R}, p_{1R}) l ps =$   
 $let (\delta, c_0, c_1) = if \delta_L < \delta_R then (\delta_L, p_{0L}, p_{1L}) else (\delta_R, p_{0R}, p_{1R}) in$   
 $let ps' = filter (\lambda p. (fst p - l)^2 < \delta) ps in$   
 $find-closest-pair-code (\delta, c_0, c_1) ps'$   
 $)$

**lemma** *combine-code-dist-eq*:

**assumes**  $\delta_L = dist-code p_{0L} p_{1L}$   $\delta_R = dist-code p_{0R} p_{1R}$   
**assumes**  $(\delta, c_0, c_1) = combine-code (\delta_L, p_{0L}, p_{1L}) (\delta_R, p_{0R}, p_{1R}) l ps$   
**shows**  $\delta = dist-code c_0 c_1$   
**using** *assms* **by** (*auto simp: find-closest-pair-code-dist-eq split: if-splits*)

**lemma** *combine-code-eq*:

**assumes**  $\delta_L' = dist-code p_{0L} p_{1L}$   $\delta_R' = dist-code p_{0R} p_{1R}$  *sorted-snd*  $ps$   
**assumes**  $(c_0, c_1) = combine (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) l ps$   
**assumes**  $(\delta', c_0', c_1') = combine-code (\delta_L', p_{0L}, p_{1L}) (\delta_R', p_{0R}, p_{1R}) l ps$   
**shows**  $c_0 = c_0' \wedge c_1 = c_1'$

**proof** –

**obtain**  $C_{0i}$   $C_{1i}$   $\Delta_i'$   $C_{0i}'$   $C_{1i}'$  **where**  $\Delta_i$ -*def*:

$(C_{0i}, C_{1i}) = (if dist p_{0L} p_{1L} < dist p_{0R} p_{1R} then (p_{0L}, p_{1L}) else (p_{0R}, p_{1R}))$   
 $(\Delta_i', C_{0i}', C_{1i}') = (if \delta_L' < \delta_R' then (\delta_L', p_{0L}, p_{1L}) else (\delta_R', p_{0R}, p_{1R}))$

**by** *metis*

**define**  $ps'$   $ps''$  **where**  $ps'$ -*def*:

$ps' = filter (\lambda p. dist p (l, snd p) < dist C_{0i} C_{1i}) ps$   
 $ps'' = filter (\lambda p. (fst p - l)^2 < \Delta_i') ps$

**obtain**  $C_0$   $C_1$   $\Delta'$   $C_0'$   $C_1'$  **where**  $\Delta$ -*def*:

$(C_0, C_1) = find-closest-pair (C_{0i}, C_{1i}) ps'$   
 $(\Delta', C_0', C_1') = find-closest-pair-code (\Delta_i', C_{0i}', C_{1i}') ps''$

**by** (*metis prod-cases3*)

**note**  $defs = \Delta_i$ -*def*  $ps'$ -*def*  $\Delta$ -*def*

**have**  $*$ :  $C_{0i} = C_{0i}'$   $C_{1i} = C_{1i}'$   $\Delta_i' = dist-code C_{0i}' C_{1i}'$

**using**  $\Delta_i$ -*def* *assms(1,2,3,4)* *dist-eq-dist-code-lt* **by** (*auto split: if-splits*)

**hence**  $\bigwedge p. |fst p - l| < dist C_{0i} C_{1i} \iff (fst p - l)^2 < \Delta_i'$

**using** *dist-eq-dist-code-abs-lt* **by** (*metis (mono-tags) of-int-abs*)

hence  $ps' = ps''$   
 using  $ps'$ -def *dist-fst-abs* by *auto*  
 moreover have *sorted-snd*  $ps'$   
 using *assms*(3)  $ps'$ -def *sorted-snd-def sorted-wrt-filter* by *blast*  
 ultimately have  $C_0 = C_0' C_1 = C_1'$   
 using \* *find-closest-pair-code-eq*  $\Delta$ -def by *blast+*  
 moreover have  $C_0 = c_0 C_1 = c_1$   
 using *assms*(4) *defs*(1,3,5) *apply* (*auto simp: combine.simps split: prod.splits*)  
 by (*metis Pair-inject*)+  
 moreover have  $C_0' = c_0' C_1' = c_1'$   
 using *assms*(5) *defs*(2,4,6) *apply* (*auto split: prod.splits*) by (*metis prod.inject*)+  
 ultimately show *?thesis*  
 by *blast*  
 qed

### 2.3.2 Divide and Conquer Algorithm

**function** *closest-pair-rec-code* :: *point list*  $\Rightarrow$  (*point list* \* *int* \* *point* \* *point*)  
**where**

*closest-pair-rec-code*  $xs =$  (  
   *let*  $n = \text{length } xs$  *in*  
   *if*  $n \leq 3$  *then*  
     (*mergesort snd*  $xs$ , *closest-pair-bf-code*  $xs$ )  
   *else*  
     *let* ( $xs_L, xs_R$ ) = *split-at* ( $n \text{ div } 2$ )  $xs$  *in*  
     *let*  $l = \text{fst } (\text{hd } xs_R)$  *in*  
  
     *let* ( $ys_L, p_L$ ) = *closest-pair-rec-code*  $xs_L$  *in*  
     *let* ( $ys_R, p_R$ ) = *closest-pair-rec-code*  $xs_R$  *in*  
  
     *let*  $ys = \text{merge snd } ys_L \ ys_R$  *in*  
     ( $ys$ , *combine-code*  $p_L \ p_R \ l \ ys$ )  
 )  
 by *pat-completeness auto*

**termination** *closest-pair-rec-code*  
 by (*relation Wellfounded.measure* ( $\lambda xs. \text{length } xs$ ))  
 (*auto simp: split-at-take-drop-conv Let-def*)

**lemma** *closest-pair-rec-code-simps*:  
**assumes**  $n = \text{length } xs \neg (n \leq 3)$   
**shows** *closest-pair-rec-code*  $xs =$  (  
   *let* ( $xs_L, xs_R$ ) = *split-at* ( $n \text{ div } 2$ )  $xs$  *in*  
   *let*  $l = \text{fst } (\text{hd } xs_R)$  *in*  
   *let* ( $ys_L, p_L$ ) = *closest-pair-rec-code*  $xs_L$  *in*  
   *let* ( $ys_R, p_R$ ) = *closest-pair-rec-code*  $xs_R$  *in*  
   *let*  $ys = \text{merge snd } ys_L \ ys_R$  *in*  
   ( $ys$ , *combine-code*  $p_L \ p_R \ l \ ys$ )  
 )  
 using *assms* by (*auto simp: Let-def*)

```

declare combine.simps combine-code.simps closest-pair-rec-code.simps [simp del]

lemma closest-pair-rec-code-dist-eq:
  assumes  $1 < \text{length } xs$  ( $ys, \delta, c_0, c_1$ ) = closest-pair-rec-code xs
  shows  $\delta = \text{dist-code } c_0 c_1$ 
  using assms
proof (induction xs arbitrary: ys  $\delta$   $c_0$   $c_1$  rule: length-induct)
  case ( $1\ xs$ )
  let  $?n = \text{length } xs$ 
  show ?case
  proof (cases ?n  $\leq 3$ )
    case True
    hence  $(\delta, c_0, c_1) = \text{closest-pair-bf-code } xs$ 
    using 1.premis(2) closest-pair-rec-code.simps by simp
    thus ?thesis
    using 1.premis(1) closest-pair-bf-code-dist-eq by simp
  next
  case False

  obtain  $XS_L\ XS_R$  where  $XS_{LR}\text{-def}: (XS_L, XS_R) = \text{split-at } (?n \text{ div } 2)\ xs$ 
  using prod.collapse by blast
  define  $L$  where  $L = \text{fst } (\text{hd } XS_R)$ 

  obtain  $YS_L\ \Delta_L\ C_{0L}\ C_{1L}$  where  $YSC_{01L}\text{-def}: (YS_L, \Delta_L, C_{0L}, C_{1L}) =$ 
closest-pair-rec-code  $XS_L$ 
  using prod.collapse by metis
  obtain  $YS_R\ \Delta_R\ C_{0R}\ C_{1R}$  where  $YSC_{01R}\text{-def}: (YS_R, \Delta_R, C_{0R}, C_{1R}) =$ 
closest-pair-rec-code  $XS_R$ 
  using prod.collapse by metis

  define  $YS$  where  $YS = \text{merge } (\lambda p. \text{snd } p)\ YS_L\ YS_R$ 
  obtain  $\Delta\ C_0\ C_1$  where  $C_{01}\text{-def}: (\Delta, C_0, C_1) = \text{combine-code } (\Delta_L, C_{0L},$ 
 $C_{1L})\ (\Delta_R, C_{0R}, C_{1R})\ L\ YS$ 
  using prod.collapse by metis
  note  $\text{defs} = XS_{LR}\text{-def } L\text{-def } YSC_{01L}\text{-def } YSC_{01R}\text{-def } YS\text{-def } C_{01}\text{-def}$ 

  have  $1 < \text{length } XS_L$   $\text{length } XS_L < \text{length } xs$ 
  using False 1.premis(1) defs by (auto simp: split-at-take-drop-conv)
  hence IHL:  $\Delta_L = \text{dist-code } C_{0L}\ C_{1L}$ 
  using 1.IH defs by metis+

  have  $1 < \text{length } XS_R$   $\text{length } XS_R < \text{length } xs$ 
  using False 1.premis(1) defs by (auto simp: split-at-take-drop-conv)
  hence IHR:  $\Delta_R = \text{dist-code } C_{0R}\ C_{1R}$ 
  using 1.IH defs by metis+

  have  $*$ :  $(YS, \Delta, C_0, C_1) = \text{closest-pair-rec-code } xs$ 
  using False closest-pair-rec-code-simps defs by (auto simp: Let-def split:

```

```

prod.split)
  moreover have  $\Delta = \text{dist-code } C_0 C_1$ 
  using combine-code-dist-eq IHL IHR  $C_{01}\text{-def}$  by blast
  ultimately show ?thesis
  using 1.premis(2) * by (metis Pair-inject)
qed
qed

lemma closest-pair-rec-ys-eq:
  assumes  $1 < \text{length } xs$ 
  assumes  $(ys, c_0, c_1) = \text{closest-pair-rec } xs$ 
  assumes  $(ys', \delta', c_0', c_1') = \text{closest-pair-rec-code } xs$ 
  shows  $ys = ys'$ 
  using assms
proof (induction xs arbitrary: ys c_0 c_1 ys'  $\delta'$   $c_0'$   $c_1'$  rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n  $\leq 3$ )
    case True
    hence  $ys = \text{mergesort snd } xs$ 
    using 1.premis(2) closest-pair-rec.simps by simp
    moreover have  $ys' = \text{mergesort snd } xs$ 
    using 1.premis(3) closest-pair-rec-code.simps by (simp add: True)
    ultimately show ?thesis
    using 1.premis(1) by simp
  next
    case False
    obtain  $XS_L XS_R$  where  $XS_{LR}\text{-def}: (XS_L, XS_R) = \text{split-at } (?n \text{ div } 2) xs$ 
    using prod.collapse by blast
    define L where  $L = \text{fst } (\text{hd } XS_R)$ 

    obtain  $YS_L C_{0L} C_{1L} YS_L' \Delta_L' C_{0L}' C_{1L}'$  where  $YSC_{01L}\text{-def}$ :
       $(YS_L, C_{0L}, C_{1L}) = \text{closest-pair-rec } XS_L$ 
       $(YS_L', \Delta_L', C_{0L}', C_{1L}') = \text{closest-pair-rec-code } XS_L$ 
    using prod.collapse by metis
    obtain  $YS_R C_{0R} C_{1R} YS_R' \Delta_R' C_{0R}' C_{1R}'$  where  $YSC_{01R}\text{-def}$ :
       $(YS_R, C_{0R}, C_{1R}) = \text{closest-pair-rec } XS_R$ 
       $(YS_R', \Delta_R', C_{0R}', C_{1R}') = \text{closest-pair-rec-code } XS_R$ 
    using prod.collapse by metis

    define  $YS YS'$  where  $YS\text{-def}$ :
       $YS = \text{merge } (\lambda p. \text{snd } p) YS_L YS_R$ 
       $YS' = \text{merge } (\lambda p. \text{snd } p) YS_L' YS_R'$ 
    obtain  $C_0 C_1 \Delta' C_0' C_1'$  where  $C_{01}\text{-def}$ :
       $(C_0, C_1) = \text{combine } (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) L YS$ 
       $(\Delta', C_0', C_1') = \text{combine-code } (\Delta_L', C_{0L}', C_{1L}') (\Delta_R', C_{0R}', C_{1R}') L YS'$ 
    using prod.collapse by metis
  end
end

```

```

note  $defs = XS_{LR}$ -def L-def YSC01L-def YSC01R-def YS-def C01-def

have  $1 < \text{length } XS_L$   $\text{length } XS_L < \text{length } xs$ 
  using False 1.premis(1) defs by (auto simp: split-at-take-drop-conv)
hence IHL:  $YS_L = YS'_L$ 
  using 1.IH defs by metis

have  $1 < \text{length } XS_R$   $\text{length } XS_R < \text{length } xs$ 
  using False 1.premis(1) defs by (auto simp: split-at-take-drop-conv)
hence IHR:  $YS_R = YS'_R$ 
  using 1.IH defs by metis

have  $(YS, C_0, C_1) = \text{closest-pair-rec } xs$ 
  using False closest-pair-rec-simps defs(1,2,3,5,7,9)
  by (auto simp: Let-def split: prod.split)
moreover have  $(YS', \Delta', C'_0, C'_1) = \text{closest-pair-rec-code } xs$ 
  using False closest-pair-rec-code-simps defs(1,2,4,6,8,10)
  by (auto simp: Let-def split: prod.split)
moreover have  $YS = YS'$ 
  using IHL IHR YS-def by simp
ultimately show ?thesis
  by (metis 1.premis(2,3) Pair-inject)
qed
qed

lemma closest-pair-rec-code-eq:
  assumes  $1 < \text{length } xs$ 
  assumes  $(ys, c_0, c_1) = \text{closest-pair-rec } xs$ 
  assumes  $(ys', \delta', c'_0, c'_1) = \text{closest-pair-rec-code } xs$ 
  shows  $c_0 = c'_0 \wedge c_1 = c'_1$ 
  using assms
proof (induction xs arbitrary: ys c_0 c_1 ys' \delta' c'_0 c'_1 rule: length-induct)
  case (1 xs)
  let  $?n = \text{length } xs$ 
  show ?case
  proof (cases ?n ≤ 3)
    case True
    hence  $(c_0, c_1) = \text{closest-pair-bf } xs$ 
    using 1.premis(2) closest-pair-rec.simps by simp
    moreover have  $(\delta', c'_0, c'_1) = \text{closest-pair-bf-code } xs$ 
    using 1.premis(3) closest-pair-rec-code.simps by (simp add: True)
    ultimately show ?thesis
    using 1.premis(1) closest-pair-bf-code-eq by simp
  next
  case False

obtain  $XS_L XS_R$  where  $XS_{LR}$ -def:  $(XS_L, XS_R) = \text{split-at } (?n \text{ div } 2) xs$ 
  using prod.collapse by blast
define  $L$  where  $L = \text{fst } (\text{hd } XS_R)$ 

```

**obtain**  $YS_L C_{0L} C_{1L} YS_L' \Delta_L' C_{0L}' C_{1L}'$  **where**  $YSC_{01L}$ -def:  
 $(YS_L, C_{0L}, C_{1L}) = \text{closest-pair-rec } XS_L$   
 $(YS_L', \Delta_L', C_{0L}', C_{1L}') = \text{closest-pair-rec-code } XS_L$   
**using** *prod.collapse by metis*

**obtain**  $YS_R C_{0R} C_{1R} YS_R' \Delta_R' C_{0R}' C_{1R}'$  **where**  $YSC_{01R}$ -def:  
 $(YS_R, C_{0R}, C_{1R}) = \text{closest-pair-rec } XS_R$   
 $(YS_R', \Delta_R', C_{0R}', C_{1R}') = \text{closest-pair-rec-code } XS_R$   
**using** *prod.collapse by metis*

**define**  $YS YS'$  **where**  $YS$ -def:  
 $YS = \text{merge } (\lambda p. \text{snd } p) YS_L YS_R$   
 $YS' = \text{merge } (\lambda p. \text{snd } p) YS_L' YS_R'$

**obtain**  $C_0 C_1 \Delta' C_0' C_1'$  **where**  $C_{01}$ -def:  
 $(C_0, C_1) = \text{combine } (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) L YS$   
 $(\Delta', C_0', C_1') = \text{combine-code } (\Delta_L', C_{0L}', C_{1L}') (\Delta_R', C_{0R}', C_{1R}') L YS'$   
**using** *prod.collapse by metis*

**note**  $\text{defs} = XS_{LR}$ -def  $L$ -def  $YSC_{01L}$ -def  $YSC_{01R}$ -def  $YS$ -def  $C_{01}$ -def

**have**  $1 < \text{length } XS_L \text{ length } XS_L < \text{length } xs$   
**using** *False 1.premis(1) defs by (auto simp: split-at-take-drop-conv)*  
**hence**  $IHL: C_{0L} = C_{0L}' C_{1L} = C_{1L}'$   
**using** *1.IH defs by metis+*

**have**  $1 < \text{length } XS_R \text{ length } XS_R < \text{length } xs$   
**using** *False 1.premis(1) defs by (auto simp: split-at-take-drop-conv)*  
**hence**  $IHR: C_{0R} = C_{0R}' C_{1R} = C_{1R}'$   
**using** *1.IH defs by metis+*

**have** *sorted-snd*  $YS_L$  *sorted-snd*  $YS_R$   
**using** *closest-pair-rec-set-length-sorted-snd*  $YSC_{01L}$ -def(1)  $YSC_{01R}$ -def(1)

**by** *blast+*  
**hence** *sorted-snd*  $YS$   
**using** *sorted-merge sorted-snd-def*  $YS$ -def **by** *blast*

**moreover** **have**  $YS = YS'$   
**using**  $\text{defs } \langle 1 < \text{length } XS_L \rangle \langle 1 < \text{length } XS_R \rangle$  *closest-pair-rec-ys-eq* **by** *blast*

**moreover** **have**  $\Delta_L' = \text{dist-code } C_{0L}' C_{1L}' \Delta_R' = \text{dist-code } C_{0R}' C_{1R}'$   
**using**  $\text{defs } \langle 1 < \text{length } XS_L \rangle \langle 1 < \text{length } XS_R \rangle$  *closest-pair-rec-code-dist-eq*

**by** *blast+*  
**ultimately** **have**  $C_0 = C_0' C_1 = C_1'$   
**using** *combine-code-eq IHL IHR*  $C_{01}$ -def **by** *blast+*

**moreover** **have**  $(YS, C_0, C_1) = \text{closest-pair-rec } xs$   
**using** *False closest-pair-rec-simps*  $\text{defs}(1,2,3,5,7,9)$   
**by** *(auto simp: Let-def split: prod.split)*

**moreover** **have**  $(YS', \Delta', C_0', C_1') = \text{closest-pair-rec-code } xs$   
**using** *False closest-pair-rec-code-simps*  $\text{defs}(1,2,4,6,8,10)$   
**by** *(auto simp: Let-def split: prod.split)*

**ultimately** **show** *?thesis*  
**using**  $1.\text{prems}(2,3)$  **by** *(metis Pair-inject)*

```

qed
qed

declare closest-pair.simps [simp add]

fun closest-pair-code :: point list  $\Rightarrow$  (point * point) where
  closest-pair-code [] = undefined
| closest-pair-code [-] = undefined
| closest-pair-code ps = (let (-, -, c0, c1) = closest-pair-rec-code (mergesort fst ps)
  in (c0, c1))

lemma closest-pair-code-eq:
  closest-pair ps = closest-pair-code ps
proof (induction ps rule: induct-list012)
  case ( $\exists x y zs$ )
  obtain ys c0 c1 ys'  $\delta'$  c0' c1' where *:
    (ys, c0, c1) = closest-pair-rec (mergesort fst (x # y # zs))
    (ys',  $\delta'$ , c0', c1') = closest-pair-rec-code (mergesort fst (x # y # zs))
  by (metis prod-cases3)
  moreover have  $1 < \text{length} (\text{mergesort fst } (x \# y \# zs))$ 
    using length-mergesort[of fst x # y # zs] by simp
  ultimately have c0 = c0' c1 = c1'
    using closest-pair-rec-code-eq by blast+
  thus ?case
    using * by (auto split: prod.splits)
qed auto

export-code closest-pair-code in OCaml
  module-name Verified

end

```

## 3 Closest Pair Algorithm 2

```

theory Closest-Pair-Alternative
  imports Common
begin

```

Formalization of a divide-and-conquer algorithm solving the Closest Pair Problem based on the presentation of Cormen *et al.* [1].

### 3.1 Functional Correctness Proof

#### 3.1.1 Core Argument

```

lemma core-argument:
  assumes distinct (p0 # ps) sorted-snd (p0 # ps)  $0 \leq \delta$  set (p0 # ps) = psL  $\cup$ 
psR
  assumes  $\forall p \in \text{set } (p_0 \# ps). l - \delta \leq \text{fst } p \wedge \text{fst } p \leq l + \delta$ 

```

**assumes**  $\forall p \in ps_L. fst\ p \leq l \ \forall p \in ps_R. l \leq fst\ p$   
**assumes**  $sparse\ \delta\ ps_L\ sparse\ \delta\ ps_R$   
**assumes**  $p_1 \in set\ ps\ dist\ p_0\ p_1 < \delta$   
**shows**  $p_1 \in set\ (take\ 7\ ps)$   
**proof** –  
**define**  $PS$  **where**  $PS = p_0 \# ps$   
**define**  $R$  **where**  $R = cbox\ (l - \delta, snd\ p_0)\ (l + \delta, snd\ p_0 + \delta)$   
**define**  $RPS$  **where**  $RPS = \{ p \in set\ PS. p \in R \}$   
**define**  $LSQ$  **where**  $LSQ = cbox\ (l - \delta, snd\ p_0)\ (l, snd\ p_0 + \delta)$   
**define**  $LSQPS$  **where**  $LSQPS = \{ p \in ps_L. p \in LSQ \}$   
**define**  $RSQ$  **where**  $RSQ = cbox\ (l, snd\ p_0)\ (l + \delta, snd\ p_0 + \delta)$   
**define**  $RSQPS$  **where**  $RSQPS = \{ p \in ps_R. p \in RSQ \}$   
**note**  $defs = PS-def\ R-def\ RPS-def\ LSQ-def\ LSQPS-def\ RSQ-def\ RSQPS-def$   
  
**have**  $R = LSQ \cup RSQ$   
**using**  $defs\ cbox-right-un$  **by**  $auto$   
**moreover** **have**  $\forall p \in ps_L. p \in RSQ \longrightarrow p \in LSQ$   
**using**  $RSQ-def\ LSQ-def\ assms(6)$  **by**  $auto$   
**moreover** **have**  $\forall p \in ps_R. p \in LSQ \longrightarrow p \in RSQ$   
**using**  $RSQ-def\ LSQ-def\ assms(7)$  **by**  $auto$   
**ultimately** **have**  $RPS = LSQPS \cup RSQPS$   
**using**  $LSQPS-def\ RSQPS-def\ PS-def\ RPS-def\ assms(4)$  **by**  $blast$   
  
**have**  $sparse\ \delta\ LSQPS$   
**using**  $assms(8)\ LSQPS-def\ sparse-def$  **by**  $simp$   
**hence**  $CLSQPS: card\ LSQPS \leq 4$   
**using**  $max-points-square[of\ LSQPS\ l - \delta\ snd\ p_0\ \delta]$   $assms(3)\ LSQ-def\ LSQPS-def$   
**by**  $auto$   
  
**have**  $sparse\ \delta\ RSQPS$   
**using**  $assms(9)\ RSQPS-def\ sparse-def$  **by**  $simp$   
**hence**  $CRSQPS: card\ RSQPS \leq 4$   
**using**  $max-points-square[of\ RSQPS\ l\ snd\ p_0\ \delta]$   $assms(3)\ RSQ-def\ RSQPS-def$   
**by**  $auto$   
  
**have**  $CRPS: card\ RPS \leq 8$   
**using**  $CLSQPS\ CRSQPS\ card-Un-le[of\ LSQPS\ RSQPS]$   $\langle RPS = LSQPS \cup RSQPS \rangle$  **by**  $auto$   
  
**have**  $RPS \subseteq set\ (take\ 8\ PS)$   
**proof**  $(rule\ ccontr)$   
**assume**  $\neg (RPS \subseteq set\ (take\ 8\ PS))$   
**then** **obtain**  $p$  **where**  $*: p \in set\ PS\ p \in RPS\ p \notin set\ (take\ 8\ PS)\ p \in R$   
**using**  $RPS-def$  **by**  $auto$   
  
**have**  $\forall p_0 \in set\ (take\ 8\ PS). \forall p_1 \in set\ (drop\ 8\ PS). snd\ p_0 \leq snd\ p_1$   
**using**  $sorted-wrt-take-drop[of\ \lambda p_0\ p_1. snd\ p_0 \leq snd\ p_1\ PS\ 8]$   $assms(2)$   
 $sorted-snd-def\ PS-def$  **by**  $fastforce$   
**hence**  $\forall p' \in set\ (take\ 8\ PS). snd\ p' \leq snd\ p$

**using** *append-take-drop-id set-append Un-iff \*(1,3)* **by** *metis*  
**moreover have**  $\text{snd } p \leq \text{snd } p_0 + \delta$   
**using**  $\langle p \in R \rangle$  *R-def mem-cbox-2D* **by** (*metis (mono-tags, lifting) prod.case-eq-if*)  
**ultimately have**  $\forall p \in \text{set } (\text{take } 8 \text{ PS}). \text{snd } p \leq \text{snd } p_0 + \delta$   
**by** *fastforce*  
**moreover have**  $\forall p \in \text{set } (\text{take } 8 \text{ PS}). \text{snd } p_0 \leq \text{snd } p$   
**using** *sorted-wrt-hd-less-take[of  $\lambda p_0 p_1. \text{snd } p_0 \leq \text{snd } p_1$   $p_0$   $ps$  8] assms(2)*  
*sorted-snd-def PS-def* **by** *fastforce*  
**moreover have**  $\forall p \in \text{set } (\text{take } 8 \text{ PS}). l - \delta \leq \text{fst } p \wedge \text{fst } p \leq l + \delta$   
**using** *assms(5) PS-def* **by** (*meson in-set-takeD*)  
**ultimately have**  $\forall p \in \text{set } (\text{take } 8 \text{ PS}). p \in R$   
**using** *R-def mem-cbox-2D* **by** *fastforce*

**hence**  $\text{set } (\text{take } 8 \text{ PS}) \subseteq RPS$   
**using** *RPS-def set-take-subset* **by** *fastforce*  
**hence** *NINE*:  $\{ p \} \cup \text{set } (\text{take } 8 \text{ PS}) \subseteq RPS$   
**using** *\** **by** *simp*

**have**  $8 \leq \text{length } PS$   
**using** *\*(1,3) nat-le-linear* **by** *fastforce*  
**hence**  $\text{length } (\text{take } 8 \text{ PS}) = 8$   
**by** *simp*

**have** *finite*  $\{ p \}$  *finite*  $(\text{set } (\text{take } 8 \text{ PS}))$   
**by** *simp-all*  
**hence**  $\text{card } (\{ p \} \cup \text{set } (\text{take } 8 \text{ PS})) = \text{card } (\{ p \}) + \text{card } (\text{set } (\text{take } 8 \text{ PS}))$   
**using** *\*(3) card-Un-disjoint* **by** *blast*  
**hence**  $\text{card } (\{ p \} \cup \text{set } (\text{take } 8 \text{ PS})) = 9$   
**using** *assms(1)  $\langle \text{length } (\text{take } 8 \text{ PS}) = 8 \rangle$  distinct-card[of take 8 PS] distinct-take[of PS] PS-def* **by** *fastforce*  
**moreover have** *finite* *RPS*  
**using** *RPS-def* **by** *simp*  
**ultimately have**  $9 \leq \text{card } RPS$   
**using** *NINE card-mono* **by** *metis*  
**thus** *False*  
**using** *CRPS* **by** *simp*

**qed**

**have**  $\text{dist } (\text{snd } p_0) (\text{snd } p_1) < \delta$   
**using** *assms(11) dist-snd-le le-less-trans* **by** (*metis (no-types, lifting) prod.case-eq-if snd-conv*)  
**hence**  $\text{snd } p_1 \leq \text{snd } p_0 + \delta$   
**by** (*simp add: dist-real-def*)  
**moreover have**  $l - \delta \leq \text{fst } p_1 \wedge \text{fst } p_1 \leq l + \delta$   
**using** *assms(5,10)* **by** *auto*  
**moreover have**  $\text{snd } p_0 \leq \text{snd } p_1$   
**using** *sorted-snd-def assms(2,10)* **by** *auto*  
**ultimately have**  $p_1 \in R$   
**using** *mem-cbox-2D[of  $l - \delta$   $\text{fst } p_1$   $l + \delta$   $\text{snd } p_0$   $\text{snd } p_1$   $\text{snd } p_0 + \delta$ ]* *defs*

by (*simp add*:  $\langle l - \delta \leq \text{fst } p_1 \rangle \langle \text{snd } p_0 \leq \text{snd } p_1 \rangle \text{prod.case-eq-if}$ )  
 moreover have  $p_1 \in \text{set } PS$   
 using *PS-def assms(10)* by *simp*  
 ultimately have  $p_1 \in \text{set (take 8 PS)}$   
 using *RPS-def*  $\langle RPS \subseteq \text{set (take 8 PS)} \rangle$  by *auto*  
 thus ?thesis  
 using *assms(1,10) PS-def* by *auto*  
 qed

### 3.1.2 Combine step

**lemma** *find-closest-bf-dist-take-7*:

assumes  $\exists p_1 \in \text{set } ps. \text{dist } p_0 p_1 < \delta$   
 assumes *distinct*  $(p_0 \# ps)$  *sorted-snd*  $(p_0 \# ps)$   $0 < \text{length } ps$   $0 \leq \delta$  *set*  $(p_0 \# ps) = ps_L \cup ps_R$   
 assumes  $\forall p \in \text{set } (p_0 \# ps). l - \delta \leq \text{fst } p \wedge \text{fst } p \leq l + \delta$   
 assumes  $\forall p \in ps_L. \text{fst } p \leq l \forall p \in ps_R. l \leq \text{fst } p$   
 assumes *sparse*  $\delta ps_L$  *sparse*  $\delta ps_R$   
 shows  $\forall p_1 \in \text{set } ps. \text{dist } p_0 (\text{find-closest-bf } p_0 (\text{take } 7 ps)) \leq \text{dist } p_0 p_1$   
**proof** –  
 have  $\text{dist } p_0 (\text{find-closest-bf } p_0 ps) < \delta$   
 using *assms(1) dual-order.strict-trans2 find-closest-bf-dist* by *blast*  
 moreover have *find-closest-bf*  $p_0 ps \in \text{set } ps$   
 using *assms(4) find-closest-bf-set* by *blast*  
 ultimately have *find-closest-bf*  $p_0 ps \in \text{set (take 7 ps)}$   
 using *core-argument*[*of*  $p_0 ps \delta ps_L ps_R l$  *find-closest-bf*  $p_0 ps$ ] *assms* by *blast*  
 moreover have  $\forall p_1 \in \text{set (take 7 ps)}. \text{dist } p_0 (\text{find-closest-bf } p_0 (\text{take 7 ps})) \leq \text{dist } p_0 p_1$   
 using *find-closest-bf-dist* by *blast*  
 ultimately have  $\forall p_1 \in \text{set } ps. \text{dist } p_0 (\text{find-closest-bf } p_0 (\text{take 7 ps})) \leq \text{dist } p_0 p_1$   
 using *find-closest-bf-dist order.trans* by *blast*  
 thus ?thesis .  
 qed

**fun** *find-closest-pair-tm* ::  $(\text{point} * \text{point}) \Rightarrow \text{point list} \Rightarrow (\text{point} \times \text{point}) \text{tm}$  **where**  
*find-closest-pair-tm*  $(c_0, c_1) [] = 1$  *return*  $(c_0, c_1)$   
*find-closest-pair-tm*  $(c_0, c_1) [-] = 1$  *return*  $(c_0, c_1)$   
*find-closest-pair-tm*  $(c_0, c_1) (p_0 \# ps) = 1$  (  
   do {  
      $ps' <- \text{take-tm } 7 ps$ ;  
      $p_1 <- \text{find-closest-bf-tm } p_0 ps'$ ;  
     if  $\text{dist } c_0 c_1 \leq \text{dist } p_0 p_1$  then  
       *find-closest-pair-tm*  $(c_0, c_1) ps$   
     else  
       *find-closest-pair-tm*  $(p_0, p_1) ps$   
   }  
 )

```

fun find-closest-pair :: (point * point) ⇒ point list ⇒ (point * point) where
  find-closest-pair (c0, c1) [] = (c0, c1)
| find-closest-pair (c0, c1) [-] = (c0, c1)
| find-closest-pair (c0, c1) (p0 # ps) = (
  let p1 = find-closest-bf p0 (take 7 ps) in
  if dist c0 c1 ≤ dist p0 p1 then
    find-closest-pair (c0, c1) ps
  else
    find-closest-pair (p0, p1) ps
)

```

**lemma** find-closest-pair-eq-val-find-closest-pair-tm:

```

val (find-closest-pair-tm (c0, c1) ps) = find-closest-pair (c0, c1) ps
by (induction (c0, c1) ps arbitrary: c0 c1 rule: find-closest-pair.induct)
  (auto simp: Let-def find-closest-bf-eq-val-find-closest-bf-tm take-eq-val-take-tm)

```

**lemma** find-closest-pair-set:

```

assumes (C0, C1) = find-closest-pair (c0, c1) ps
shows (C0 ∈ set ps ∧ C1 ∈ set ps) ∨ (C0 = c0 ∧ C1 = c1)
using assms
proof (induction (c0, c1) ps arbitrary: c0 c1 C0 C1 rule: find-closest-pair.induct)
  case (∃ c0 c1 p0 p2 ps)
    define p1 where p1-def: p1 = find-closest-bf p0 (take 7 (p2 # ps))
    hence A: p1 ∈ set (p2 # ps)
      using find-closest-bf-set[of take 7 (p2 # ps)] in-set-takeD by fastforce
    show ?case
    proof (cases dist c0 c1 ≤ dist p0 p1)
      case True
        obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (c0, c1) (p2 # ps)
          using prod.collapse by blast
        note defs = p1-def C'-def
        hence (C0' ∈ set (p2 # ps) ∧ C1' ∈ set (p2 # ps)) ∨ (C0' = c0 ∧ C1' = c1)
          using 3.hyps(1) True p1-def by blast
        moreover have C0 = C0' C1 = C1'
          using defs True 3.prem1 apply (auto split: prod.splits) by (metis Pair-inject)+
        ultimately show ?thesis
          by auto
      next
        case False
          obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (p0, p1) (p2 # ps)
            using prod.collapse by blast
          note defs = p1-def C'-def
          hence (C0' ∈ set (p2 # ps) ∧ C1' ∈ set (p2 # ps)) ∨ (C0' = p0 ∧ C1' = p1)
            using 3.hyps(2) p1-def by blast
          moreover have C0 = C0' C1 = C1'
            using defs False 3.prem1 apply (auto split: prod.splits) by (metis Pair-inject)+
          ultimately show ?thesis

```

```

    using A by auto
  qed
qed auto

lemma find-closest-pair-c0-ne-c1:
  c0 ≠ c1 ⇒ distinct ps ⇒ (C0, C1) = find-closest-pair (c0, c1) ps ⇒ C0 ≠
  C1
proof (induction (c0, c1) ps arbitrary: c0 c1 C0 C1 rule: find-closest-pair.induct)
  case (∃ c0 c1 p0 p2 ps)
  define p1 where p1-def: p1 = find-closest-bf p0 (take 7 (p2 # ps))
  hence p1 ∈ set (p2 # ps)
    using find-closest-bf-set[of take 7 (p2 # ps)] in-set-takeD by fastforce
  hence A: p0 ≠ p1
    using ∃.prems(1,2) by auto
  show ?case
  proof (cases dist c0 c1 ≤ dist p0 p1)
    case True
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (c0, c1) (p2 #
  ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence C0' ≠ C1'
      using ∃.hyps(1) ∃.prems(1,2) True p1-def by simp
    moreover have C0 = C0' C1 = C1'
      using defs True ∃.prems(3) apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
      by simp
    next
    case False
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (p0, p1) (p2 #
  ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence C0' ≠ C1'
      using ∃.hyps(2) ∃.prems(2) A False p1-def by simp
    moreover have C0 = C0' C1 = C1'
      using defs False ∃.prems(3) apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
      by simp
  qed
qed auto

lemma find-closest-pair-dist-mono:
  assumes (C0, C1) = find-closest-pair (c0, c1) ps
  shows dist C0 C1 ≤ dist c0 c1
  using assms
proof (induction (c0, c1) ps arbitrary: c0 c1 C0 C1 rule: find-closest-pair.induct)
  case (∃ c0 c1 p0 p2 ps)
  define p1 where p1-def: p1 = find-closest-bf p0 (take 7 (p2 # ps))

```

```

show ?case
proof (cases dist c0 c1 ≤ dist p0 p1)
  case True
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (c0, c1) (p2 #
ps)
    using prod.collapse by blast
    note defs = p1-def C'-def
    hence dist C0' C1' ≤ dist c0 c1
    using 3.hyps(1) True p1-def by simp
    moreover have C0 = C0' C1 = C1'
    using defs True 3.prems apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
    by simp
  next
    case False
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (p0, p1) (p2 #
ps)
    using prod.collapse by blast
    note defs = p1-def C'-def
    hence dist C0' C1' ≤ dist p0 p1
    using 3.hyps(2) False p1-def by blast
    moreover have C0 = C0' C1 = C1'
    using defs False 3.prems(1) apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
    using False by simp
qed
qed auto

```

**lemma** find-closest-pair-dist:

```

assumes sorted-snd ps distinct ps set ps = psL ∪ psR 0 ≤ δ
assumes ∀ p ∈ set ps. l - δ ≤ fst p ∧ fst p ≤ l + δ
assumes ∀ p ∈ psL. fst p ≤ l ∀ p ∈ psR. l ≤ fst p
assumes sparse δ psL sparse δ psR dist c0 c1 ≤ δ
assumes (C0, C1) = find-closest-pair (c0, c1) ps
shows sparse (dist C0 C1) (set ps)
using assms
proof (induction (c0, c1) ps arbitrary: c0 c1 C0 C1 psL psR rule: find-closest-pair.induct)
  case (1 c0 c1)
    thus ?case unfolding sparse-def
    by simp
  next
    case (2 c0 c1 uu)
    thus ?case unfolding sparse-def
    by (metis length-greater-0-conv length-pos-if-in-set set-ConsD)
  next
    case (3 c0 c1 p0 p2 ps)
    define p1 where p1-def: p1 = find-closest-bf p0 (take 7 (p2 # ps))
    define PSL where PSL-def: PSL = psL - { p0 }
    define PSR where PSR-def: PSR = psR - { p0 }

```

**have** *assms*: *sorted-snd* ( $p_2 \# ps$ ) *distinct* ( $p_2 \# ps$ ) *set* ( $p_2 \# ps$ ) =  $PS_L \cup PS_R$   
 $\forall p \in \text{set } (p_2 \# ps). l - \delta \leq (\text{fst } p) \wedge (\text{fst } p) \leq l + \delta$   
 $\forall p \in PS_L. \text{fst } p \leq l \forall p \in PS_R. l \leq \text{fst } p$   
*sparse*  $\delta$   $PS_L$  *sparse*  $\delta$   $PS_R$   
**using**  $3.\text{prems}(1-9)$  *sparse-def sorted-snd-def*  $PS_L\text{-def}$   $PS_R\text{-def}$  **by** *auto*

**show** *?case*

**proof** *cases*

**assume**  $C1$ :  $\exists p \in \text{set } (p_2 \# ps). \text{dist } p_0 p < \delta$   
**hence**  $A$ :  $\forall p \in \text{set } (p_2 \# ps). \text{dist } p_0 p_1 \leq \text{dist } p_0 p$   
**using**  $p_1\text{-def}$  *find-closest-bf-dist-take-7*  $3.\text{prems}$  **by** *blast*  
**hence**  $B$ :  $\text{dist } p_0 p_1 < \delta$   
**using**  $C1$  **by** *auto*

**show** *?thesis*

**proof** *cases*

**assume**  $C2$ :  $\text{dist } c_0 c_1 \leq \text{dist } p_0 p_1$   
**obtain**  $C_0' C_1'$  **where** *def*:  $(C_0', C_1') = \text{find-closest-pair } (c_0, c_1) (p_2 \# ps)$   
**using** *prod.collapse* **by** *blast*  
**hence** *sparse*  $(\text{dist } C_0' C_1') (\text{set } (p_2 \# ps))$   
**using**  $3.\text{hyps}(1)$ [*of*  $p_1$   $PS_L$   $PS_R$ ]  $C2$   $p_1\text{-def}$   $3.\text{prems}(4,10)$  *assms* **by** *blast*  
**moreover** **have**  $\text{dist } C_0' C_1' \leq \text{dist } c_0 c_1$   
**using** *def find-closest-pair-dist-mono* **by** *blast*  
**ultimately** **have** *sparse*  $(\text{dist } C_0' C_1') (\text{set } (p_0 \# p_2 \# ps))$   
**using**  $A$   $C2$  *sparse-identity*[*of*  $\text{dist } C_0' C_1' p_2 \# ps p_0$ ] **by** *fastforce*  
**moreover** **have**  $C_0' = C_0$   $C_1' = C_1$   
**using** *def*  $3.\text{prems}(11)$   $C2$   $p_1\text{-def}$  **apply** (*auto*) **by** (*metis prod.inject*)+  
**ultimately** **show** *?thesis*  
**by** *simp*

**next**

**assume**  $C2$ :  $\neg \text{dist } c_0 c_1 \leq \text{dist } p_0 p_1$   
**obtain**  $C_0' C_1'$  **where** *def*:  $(C_0', C_1') = \text{find-closest-pair } (p_0, p_1) (p_2 \# ps)$   
**using** *prod.collapse* **by** *blast*  
**hence** *sparse*  $(\text{dist } C_0' C_1') (\text{set } (p_2 \# ps))$   
**using**  $3.\text{hyps}(2)$ [*of*  $p_1$   $PS_L$   $PS_R$ ]  $C2$   $p_1\text{-def}$   $3.\text{prems}(4)$  *assms*  $B$  **by** *auto*  
**moreover** **have**  $\text{dist } C_0' C_1' \leq \text{dist } p_0 p_1$   
**using** *def find-closest-pair-dist-mono* **by** *blast*  
**ultimately** **have** *sparse*  $(\text{dist } C_0' C_1') (\text{set } (p_0 \# p_2 \# ps))$   
**using**  $A$  *sparse-identity order-trans* **by** *blast*  
**moreover** **have**  $C_0' = C_0$   $C_1' = C_1$   
**using** *def*  $3.\text{prems}(11)$   $C2$   $p_1\text{-def}$  **apply** (*auto*) **by** (*metis prod.inject*)+  
**ultimately** **show** *?thesis*  
**by** *simp*

**qed**

**next**

**assume**  $C1$ :  $\neg (\exists p \in \text{set } (p_2 \# ps). \text{dist } p_0 p < \delta)$   
**show** *?thesis*  
**proof** *cases*

```

assume C2:  $\text{dist } c_0 \ c_1 \leq \text{dist } p_0 \ p_1$ 
obtain  $C_0' \ C_1'$  where  $\text{def: } (C_0', C_1') = \text{find-closest-pair } (c_0, c_1) \ (p_2 \# \text{ps})$ 
  using prod.collapse by blast
hence sparse ( $\text{dist } C_0' \ C_1'$ ) (set ( $p_2 \# \text{ps}$ ))
  using 3.hyps(1)[of  $p_1 \ PS_L \ PS_R$ ] C2  $p_1\text{-def}$  3.prem(4,10) assms by blast
moreover have  $\text{dist } C_0' \ C_1' \leq \text{dist } c_0 \ c_1$ 
  using def find-closest-pair-dist-mono by blast
ultimately have sparse ( $\text{dist } C_0' \ C_1'$ ) (set ( $p_0 \# \ p_2 \# \ \text{ps}$ ))
  using 3.prem(10) C1 sparse-identity[of  $\text{dist } C_0' \ C_1' \ p_2 \# \ \text{ps} \ p_0$ ] by force
moreover have  $C_0' = C_0 \ C_1' = C_1$ 
  using def 3.prem(11) C2  $p_1\text{-def}$  apply (auto) by (metis prod.inject)+
ultimately show ?thesis
  by simp
next
assume C2:  $\neg \text{dist } c_0 \ c_1 \leq \text{dist } p_0 \ p_1$ 
obtain  $C_0' \ C_1'$  where  $\text{def: } (C_0', C_1') = \text{find-closest-pair } (p_0, p_1) \ (p_2 \# \text{ps})$ 
  using prod.collapse by blast
hence sparse ( $\text{dist } C_0' \ C_1'$ ) (set ( $p_2 \# \text{ps}$ ))
  using 3.hyps(2)[of  $p_1 \ PS_L \ PS_R$ ] C2  $p_1\text{-def}$  3.prem(4,10) assms by auto
moreover have  $\text{dist } C_0' \ C_1' \leq \text{dist } p_0 \ p_1$ 
  using def find-closest-pair-dist-mono by blast
ultimately have sparse ( $\text{dist } C_0' \ C_1'$ ) (set ( $p_0 \# \ p_2 \# \ \text{ps}$ ))
  using 3.prem(10) C1 C2 sparse-identity[of  $\text{dist } C_0' \ C_1' \ p_2 \# \ \text{ps} \ p_0$ ] by
force
moreover have  $C_0' = C_0 \ C_1' = C_1$ 
  using def 3.prem(11) C2  $p_1\text{-def}$  apply (auto) by (metis prod.inject)+
ultimately show ?thesis
  by simp
qed
qed
qed

declare find-closest-pair.simps [simp del]

fun combine-tm :: (point  $\times$  point)  $\Rightarrow$  (point  $\times$  point)  $\Rightarrow$  int  $\Rightarrow$  point list  $\Rightarrow$  (point
 $\times$  point) tm where
  combine-tm ( $p_{0L}, p_{1L}$ ) ( $p_{0R}, p_{1R}$ ) l ps = 1 (
    let ( $c_0, c_1$ ) = if  $\text{dist } p_{0L} \ p_{1L} < \text{dist } p_{0R} \ p_{1R}$  then ( $p_{0L}, p_{1L}$ ) else ( $p_{0R}, p_{1R}$ ) in
    do {
       $ps' \leftarrow \text{filter-tm } (\lambda p. \text{dist } p \ (l, \text{snd } p) < \text{dist } c_0 \ c_1) \ ps;$ 
      find-closest-pair-tm ( $c_0, c_1$ )  $ps'$ 
    }
  )

fun combine :: (point  $*$  point)  $\Rightarrow$  (point  $*$  point)  $\Rightarrow$  int  $\Rightarrow$  point list  $\Rightarrow$  (point  $*$ 
point) where
  combine ( $p_{0L}, p_{1L}$ ) ( $p_{0R}, p_{1R}$ ) l ps = (
    let ( $c_0, c_1$ ) = if  $\text{dist } p_{0L} \ p_{1L} < \text{dist } p_{0R} \ p_{1R}$  then ( $p_{0L}, p_{1L}$ ) else ( $p_{0R}, p_{1R}$ ) in
    let  $ps' = \text{filter } (\lambda p. \text{dist } p \ (l, \text{snd } p) < \text{dist } c_0 \ c_1) \ ps$  in
  )

```

)  
*find-closest-pair* ( $c_0, c_1$ )  $ps'$ )

**lemma** *combine-eq-val-combine-tm*:

*val* (*combine-tm* ( $p_{0L}, p_{1L}$ ) ( $p_{0R}, p_{1R}$ )  $l$   $ps$ ) = *combine* ( $p_{0L}, p_{1L}$ ) ( $p_{0R}, p_{1R}$ )  $l$   $ps$   
**by** (*auto simp: filter-eq-val-filter-tm find-closest-pair-eq-val-find-closest-pair-tm*)

**lemma** *combine-set*:

**assumes** ( $c_0, c_1$ ) = *combine* ( $p_{0L}, p_{1L}$ ) ( $p_{0R}, p_{1R}$ )  $l$   $ps$   
**shows** ( $c_0 \in \text{set } ps \wedge c_1 \in \text{set } ps$ )  $\vee$  ( $c_0 = p_{0L} \wedge c_1 = p_{1L}$ )  $\vee$  ( $c_0 = p_{0R} \wedge c_1 = p_{1R}$ )

**proof** –

**obtain**  $C_0' C_1'$  **where**  $C'$ -def: ( $C_0', C_1'$ ) = (*if*  $\text{dist } p_{0L} p_{1L} < \text{dist } p_{0R} p_{1R}$  *then* ( $p_{0L}, p_{1L}$ ) *else* ( $p_{0R}, p_{1R}$ ))

**by** *metis*

**define**  $ps'$  **where**  $ps'$ -def:  $ps' = \text{filter } (\lambda p. \text{dist } p (l, \text{snd } p) < \text{dist } C_0' C_1') ps$

**obtain**  $C_0 C_1$  **where**  $C$ -def: ( $C_0, C_1$ ) = *find-closest-pair* ( $C_0', C_1'$ )  $ps'$

**using** *prod.collapse* **by** *blast*

**note**  $\text{defs} = C'$ -def  $ps'$ -def  $C$ -def

**have** ( $C_0 \in \text{set } ps' \wedge C_1 \in \text{set } ps'$ )  $\vee$  ( $C_0 = C_0' \wedge C_1 = C_1'$ )

**using**  $C$ -def *find-closest-pair-set* **by** *blast+*

**hence** ( $C_0 \in \text{set } ps \wedge C_1 \in \text{set } ps$ )  $\vee$  ( $C_0 = C_0' \wedge C_1 = C_1'$ )

**using**  $ps'$ -def **by** *auto*

**moreover** **have**  $C_0 = c_0$   $C_1 = c_1$

**using**  $\text{assms } \text{defs}$  **apply** (*auto split: if-splits prod.splits*) **by** (*metis Pair-inject*)+

**ultimately show** *?thesis*

**using**  $C'$ -def **by** (*auto split: if-splits*)

**qed**

**lemma** *combine-c0-ne-c1*:

**assumes**  $p_{0L} \neq p_{1L}$   $p_{0R} \neq p_{1R}$  *distinct*  $ps$

**assumes** ( $c_0, c_1$ ) = *combine* ( $p_{0L}, p_{1L}$ ) ( $p_{0R}, p_{1R}$ )  $l$   $ps$

**shows**  $c_0 \neq c_1$

**proof** –

**obtain**  $C_0' C_1'$  **where**  $C'$ -def: ( $C_0', C_1'$ ) = (*if*  $\text{dist } p_{0L} p_{1L} < \text{dist } p_{0R} p_{1R}$  *then* ( $p_{0L}, p_{1L}$ ) *else* ( $p_{0R}, p_{1R}$ ))

**by** *metis*

**define**  $ps'$  **where**  $ps'$ -def:  $ps' = \text{filter } (\lambda p. \text{dist } p (l, \text{snd } p) < \text{dist } C_0' C_1') ps$

**obtain**  $C_0 C_1$  **where**  $C$ -def: ( $C_0, C_1$ ) = *find-closest-pair* ( $C_0', C_1'$ )  $ps'$

**using** *prod.collapse* **by** *blast*

**note**  $\text{defs} = C'$ -def  $ps'$ -def  $C$ -def

**have**  $C_0 \neq C_1$

**using**  $\text{defs } \text{find-closest-pair-c0-ne-c1}$  [*of*  $C_0' C_1' ps'$ ]  $\text{assms}$  **by** (*auto split: if-splits*)

**moreover** **have**  $C_0 = c_0$   $C_1 = c_1$

**using**  $\text{assms } \text{defs}$  **apply** (*auto split: if-splits prod.splits*) **by** (*metis Pair-inject*)+

**ultimately show** *?thesis*

**by** *blast*

qed

**lemma** *combine-dist*:

**assumes** *distinct ps sorted-snd ps set ps = ps<sub>L</sub> ∪ ps<sub>R</sub>*  
**assumes**  $\forall p \in ps_L. fst\ p \leq l \ \forall p \in ps_R. l \leq fst\ p$   
**assumes** *sparse (dist p<sub>0L</sub> p<sub>1L</sub>) ps<sub>L</sub> sparse (dist p<sub>0R</sub> p<sub>1R</sub>) ps<sub>R</sub>*  
**assumes**  $(c_0, c_1) = combine\ (p_{0L}, p_{1L})\ (p_{0R}, p_{1R})\ l\ ps$   
**shows** *sparse (dist c<sub>0</sub> c<sub>1</sub>) (set ps)*  
**proof** –  
**obtain**  $C_0' C_1'$  **where**  $C'\text{-def}: (C_0', C_1') = (if\ dist\ p_{0L}\ p_{1L} < dist\ p_{0R}\ p_{1R}\ then\ (p_{0L}, p_{1L})\ else\ (p_{0R}, p_{1R}))$   
**by** *metis*  
**define**  $ps'$  **where**  $ps'\text{-def}: ps' = filter\ (\lambda p. dist\ p\ (l, snd\ p) < dist\ C_0'\ C_1')\ ps$   
**define**  $PS_L$  **where**  $PS_L\text{-def}: PS_L = \{ p \in ps_L. dist\ p\ (l, snd\ p) < dist\ C_0'\ C_1' \}$   
**define**  $PS_R$  **where**  $PS_R\text{-def}: PS_R = \{ p \in ps_R. dist\ p\ (l, snd\ p) < dist\ C_0'\ C_1' \}$   
**obtain**  $C_0 C_1$  **where**  $C\text{-def}: (C_0, C_1) = find\ closest\ pair\ (C_0', C_1')\ ps'$   
**using** *prod.collapse by blast*  
**note**  $defs = C'\text{-def}\ ps'\text{-def}\ PS_L\text{-def}\ PS_R\text{-def}\ C\text{-def}$   
**have**  $EQ: C_0 = c_0\ C_1 = c_1$   
**using**  $defs\ assms(8)$  **apply** *(auto split: if-splits prod.splits) by (metis Pair-inject)+*  
**have**  $ps': ps' = filter\ (\lambda p. l - dist\ C_0'\ C_1' < fst\ p \wedge fst\ p < l + dist\ C_0'\ C_1')$   
 $ps$   
**using**  $ps'\text{-def}\ dist\ transform\ by\ simp$   
**have**  $ps_L: sparse\ (dist\ C_0'\ C_1')\ ps_L$   
**using**  $assms(6,8)\ C'\text{-def}\ sparse\ def\ apply\ (auto\ split: if-splits)\ by\ force+$   
**hence**  $PS_L: sparse\ (dist\ C_0'\ C_1')\ PS_L$   
**using**  $PS_L\text{-def}\ by\ (simp\ add: sparse\ def)$   
**have**  $ps_R: sparse\ (dist\ C_0'\ C_1')\ ps_R$   
**using**  $assms(5,7)\ C'\text{-def}\ sparse\ def\ apply\ (auto\ split: if-splits)\ by\ force+$   
**hence**  $PS_R: sparse\ (dist\ C_0'\ C_1')\ PS_R$   
**using**  $PS_R\text{-def}\ by\ (simp\ add: sparse\ def)$   
**have** *sorted-snd ps'*  
**using**  $ps'\text{-def}\ assms(2)\ sorted\ snd\ def\ sorted\ wrt\ filter\ by\ blast$   
**moreover** **have** *distinct ps'*  
**using**  $ps'\text{-def}\ assms(1)\ distinct\ filter\ by\ blast$   
**moreover** **have**  $set\ ps' = PS_L \cup PS_R$   
**using**  $ps'\text{-def}\ PS_L\text{-def}\ PS_R\text{-def}\ assms(3)\ filter\ Un\ by\ auto$   
**moreover** **have**  $0 \leq dist\ C_0'\ C_1'$   
**by** *simp*  
**moreover** **have**  $\forall p \in set\ ps'. l - dist\ C_0'\ C_1' \leq fst\ p \wedge fst\ p \leq l + dist\ C_0'\ C_1'$   
 $C_1'$   
**using**  $ps'\ by\ simp$   
**ultimately** **have**  $*$ : *sparse (dist C<sub>0</sub> C<sub>1</sub>) (set ps')*  
**using**  $find\ closest\ pair\ dist[of\ ps'\ PS_L\ PS_R\ dist\ C_0'\ C_1'\ l\ C_0'\ C_1']\ C\text{-def}\ PS_L\ PS_R$   
**by** *(simp add: PS<sub>L</sub>-def PS<sub>R</sub>-def assms(4,5))*  
**have**  $\forall p_0 \in set\ ps. \forall p_1 \in set\ ps. p_0 \neq p_1 \wedge dist\ p_0\ p_1 < dist\ C_0'\ C_1' \longrightarrow p_0 \in$

```

set ps' ∧ p1 ∈ set ps'
  using set-band-filter ps' psL psR assms(3,4,5) by blast
  moreover have dist C0 C1 ≤ dist C0' C1'
  using C-def find-closest-pair-dist-mono by blast
  ultimately have ∀ p0 ∈ set ps. ∀ p1 ∈ set ps. p0 ≠ p1 ∧ dist p0 p1 < dist C0
C1 → p0 ∈ set ps' ∧ p1 ∈ set ps'
  by simp
  hence sparse (dist C0 C1) (set ps)
  using sparse-def * by (meson not-less)
  thus ?thesis
  using EQ by blast
qed

```

```

declare combine.simps [simp del]
declare combine-tm.simps [simp del]

```

### 3.1.3 Divide and Conquer Algorithm

```

declare split-at-take-drop-conv [simp add]

```

```

function closest-pair-rec-tm :: point list ⇒ (point list × point × point) tm where
  closest-pair-rec-tm xs = 1 (
    do {
      n <- length-tm xs;
      if n ≤ 3 then
        do {
          ys <- mergesort-tm snd xs;
          p <- closest-pair-bf-tm xs;
          return (ys, p)
        }
      else
        do {
          (xsL, xsR) <- split-at-tm (n div 2) xs;
          (ysL, p0L, p1L) <- closest-pair-rec-tm xsL;
          (ysR, p0R, p1R) <- closest-pair-rec-tm xsR;
          ys <- merge-tm snd ysL ysR;
          (p0, p1) <- combine-tm (p0L, p1L) (p0R, p1R) (fst (hd xsR)) ys;
          return (ys, p0, p1)
        }
    }
  )
by pat-completeness auto
termination closest-pair-rec-tm
by (relation Wellfounded.measure (λxs. length xs))
  (auto simp add: length-eq-val-length-tm split-at-eq-val-split-at-tm)

```

```

function closest-pair-rec :: point list ⇒ (point list * point * point) where
  closest-pair-rec xs = (
    let n = length xs in

```

```

    if  $n \leq 3$  then
      (mergesort snd  $xs$ , closest-pair-bf  $xs$ )
    else
      let ( $xs_L$ ,  $xs_R$ ) = split-at ( $n \text{ div } 2$ )  $xs$  in
      let ( $ys_L$ ,  $p_{0L}$ ,  $p_{1L}$ ) = closest-pair-rec  $xs_L$  in
      let ( $ys_R$ ,  $p_{0R}$ ,  $p_{1R}$ ) = closest-pair-rec  $xs_R$  in
      let  $ys$  = merge snd  $ys_L$   $ys_R$  in
      ( $ys$ , combine ( $p_{0L}$ ,  $p_{1L}$ ) ( $p_{0R}$ ,  $p_{1R}$ ) (fst (hd  $xs_R$ ))  $ys$ )
  )
  by pat-completeness auto
termination closest-pair-rec
  by (relation Wellfounded.measure ( $\lambda xs$ . length  $xs$ ))
  (auto simp: Let-def)

declare split-at-take-drop-conv [simp del]

lemma closest-pair-rec-simps:
  assumes  $n = \text{length } xs \wedge (n \leq 3)$ 
  shows closest-pair-rec  $xs =$  (
    let ( $xs_L$ ,  $xs_R$ ) = split-at ( $n \text{ div } 2$ )  $xs$  in
    let ( $ys_L$ ,  $p_{0L}$ ,  $p_{1L}$ ) = closest-pair-rec  $xs_L$  in
    let ( $ys_R$ ,  $p_{0R}$ ,  $p_{1R}$ ) = closest-pair-rec  $xs_R$  in
    let  $ys$  = merge snd  $ys_L$   $ys_R$  in
    ( $ys$ , combine ( $p_{0L}$ ,  $p_{1L}$ ) ( $p_{0R}$ ,  $p_{1R}$ ) (fst (hd  $xs_R$ ))  $ys$ )
  )
  using assms by (auto simp: Let-def)

declare closest-pair-rec.simps [simp del]

lemma closest-pair-rec-eq-val-closest-pair-rec-tm:
  val (closest-pair-rec-tm  $xs$ ) = closest-pair-rec  $xs$ 
proof (induction rule: length-induct)
  case (1  $xs$ )
  define  $n$  where  $n = \text{length } xs$ 
  obtain  $xs_L$   $xs_R$  where  $xs\text{-def}: (xs_L, xs_R) = \text{split-at } (n \text{ div } 2) \text{ } xs$ 
  by (metis surj-pair)
  note  $defs = n\text{-def } xs\text{-def}$ 
  show ?case
  proof cases
  assume  $n \leq 3$ 
  then show ?thesis
  using  $defs$ 
  by (auto simp: length-eq-val-length-tm mergesort-eq-val-mergesort-tm
    closest-pair-bf-eq-val-closest-pair-bf-tm closest-pair-rec.simps)
next
  assume  $asm: \neg n \leq 3$ 
  have length  $xs_L < \text{length } xs$  length  $xs_R < \text{length } xs$ 
  using  $asm$   $defs$  by (auto simp: split-at-take-drop-conv)
  hence val (closest-pair-rec-tm  $xs_L$ ) = closest-pair-rec  $xs_L$ 

```

```

    val (closest-pair-rec-tm xsR) = closest-pair-rec xsR
  using 1.IH by blast+
thus ?thesis
  using asm defs
  apply (subst closest-pair-rec.simps, subst closest-pair-rec-tm.simps)
  by (auto simp del: closest-pair-rec-tm.simps
      simp add: Let-def length-eq-val-length-tm merge-eq-val-merge-tm
              split-at-eq-val-split-at-tm combine-eq-val-combine-tm
              split: prod.split)
qed
qed

lemma closest-pair-rec-set-length-sorted-snd:
  assumes (ys, p) = closest-pair-rec xs
  shows set ys = set xs ∧ length ys = length xs ∧ sorted-snd ys
  using assms
proof (induction xs arbitrary: ys p rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    thus ?thesis using 1.prem1 sorted-snd-def
    by (auto simp: mergesort closest-pair-rec.simps)
  next
  case False
    obtain XSL XSR where XSLR-def: (XSL, XSR) = split-at (?n div 2) xs
    using prod.collapse by blast
    define L where L = fst (hd XSR)
    obtain YSL PL where YSPL-def: (YSL, PL) = closest-pair-rec XSL
    using prod.collapse by blast
    obtain YSR PR where YSPR-def: (YSR, PR) = closest-pair-rec XSR
    using prod.collapse by blast
    define YS where YS = merge (λp. snd p) YSL YSR
    define P where P = combine PL PR L YS
    note defs = XSLR-def L-def YSPL-def YSPR-def YS-def P-def

    have length XSL < length xs length XSR < length xs
    using False defs by (auto simp: split-at-take-drop-conv)
    hence IH: set XSL = set YSL set XSR = set YSR
              length XSL = length YSL length XSR = length YSR
              sorted-snd YSL sorted-snd YSR
    using 1.IH defs by metis+

    have set xs = set XSL ∪ set XSR
    using defs by (auto simp: set-take-drop split-at-take-drop-conv)
    hence SET: set xs = set YS
    using set-merge IH(1,2) defs by fast

```

```

have length xs = length XSL + length XSR
  using defs by (auto simp: split-at-take-drop-conv)
hence LENGTH: length xs = length YS
  using IH(3,4) length-merge defs by metis

have SORTED: sorted-snd YS
  using IH(5,6) by (simp add: defs sorted-snd-def sorted-merge)

have (YS, P) = closest-pair-rec xs
  using False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)
hence (ys, p) = (YS, P)
  using 1.prem by argo
thus ?thesis
  using SET LENGTH SORTED by simp
qed
qed

lemma closest-pair-rec-distinct:
  assumes distinct xs (ys, p) = closest-pair-rec xs
  shows distinct ys
  using assms
proof (induction xs arbitrary: ys p rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    thus ?thesis using 1.prem
      by (auto simp: mergesort closest-pair-rec.simps)
  next
    case False

    obtain XSL XSR where XSLR-def: (XSL, XSR) = split-at (?n div 2) xs
      using prod.collapse by blast
    define L where L = fst (hd XSR)
    obtain YSL PL where YSPL-def: (YSL, PL) = closest-pair-rec XSL
      using prod.collapse by blast
    obtain YSR PR where YSPR-def: (YSR, PR) = closest-pair-rec XSR
      using prod.collapse by blast
    define YS where YS = merge (λp. snd p) YSL YSR
    define P where P = combine PL PR L YS
    note defs = XSLR-def L-def YSPL-def YSPR-def YS-def P-def

    have length XSL < length xs length XSR < length xs
      using False defs by (auto simp: split-at-take-drop-conv)
    moreover have distinct XSL distinct XSR
      using 1.prem(1) defs by (auto simp: split-at-take-drop-conv)
    ultimately have IH: distinct YSL distinct YSR

```

```

using 1.IH defs by blast+

have set  $XS_L \cap set\ XS_R = \{\}$ 
using 1.prem1(1) defs by (auto simp: split-at-take-drop-conv set-take-disj-set-drop-if-distinct)
moreover have set  $XS_L = set\ YS_L$  set  $XS_R = set\ YS_R$ 
  using closest-pair-rec-set-length-sorted-snd defs by blast+
ultimately have set  $YS_L \cap set\ YS_R = \{\}$ 
  by blast
hence DISTINCT: distinct YS
  using distinct-merge IH defs by blast

have (YS, P) = closest-pair-rec xs
  using False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)
hence (ys, p) = (YS, P)
  using 1.prem1 by argo
thus ?thesis
  using DISTINCT by blast
qed
qed

lemma closest-pair-rec-c0-c1:
  assumes 1 < length xs distinct xs (ys, c0, c1) = closest-pair-rec xs
  shows c0 ∈ set xs ∧ c1 ∈ set xs ∧ c0 ≠ c1
  using assms
proof (induction xs arbitrary: ys c0 c1 rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    hence (c0, c1) = closest-pair-bf xs
    using 1.prem1(3) closest-pair-rec.simps by simp
    thus ?thesis
    using 1.prem1(1,2) closest-pair-bf-c0-c1 by simp
  next
  case False

  obtain  $XS_L\ XS_R$  where  $XS_{LR}$ -def:  $(XS_L, XS_R) = split\ at\ (?n\ div\ 2)\ xs$ 
    using prod.collapse by blast
  define L where L = fst (hd  $XS_R$ )

  obtain  $YS_L\ C_{0L}\ C_{1L}$  where  $YSC_{01L}$ -def:  $(YS_L, C_{0L}, C_{1L}) = closest\ pair\ rec\ XS_L$ 
    using prod.collapse by metis
  obtain  $YS_R\ C_{0R}\ C_{1R}$  where  $YSC_{01R}$ -def:  $(YS_R, C_{0R}, C_{1R}) = closest\ pair\ rec\ XS_R$ 
    using prod.collapse by metis

  define YS where YS = merge (λp. snd p)  $YS_L\ YS_R$ 

```

**obtain**  $C_0 C_1$  **where**  $C_{01}\text{-def}: (C_0, C_1) = \text{combine } (C_{0L}, C_{1L}) (C_{0R}, C_{1R})$   
 $L$   $YS$   
**using**  $\text{prod.collapse}$  **by**  $\text{metis}$   
**note**  $\text{defs} = XS_{LR}\text{-def } L\text{-def } YSC_{01L}\text{-def } YSC_{01R}\text{-def } YS\text{-def } C_{01}\text{-def}$

**have**  $1 < \text{length } XS_L$   $\text{length } XS_L < \text{length } xs$   $\text{distinct } XS_L$   
**using**  $\text{False } 1.\text{prems}(2)$  **defs by**  $(\text{auto simp: split-at-take-drop-conv})$   
**hence**  $C_{0L} \in \text{set } XS_L$   $C_{1L} \in \text{set } XS_L$  **and**  $IHL1: C_{0L} \neq C_{1L}$   
**using**  $1.IH$  **defs by**  $\text{metis+}$   
**hence**  $IHL2: C_{0L} \in \text{set } xs$   $C_{1L} \in \text{set } xs$   
**using**  $\text{split-at-take-drop-conv in-set-takeD fst-conv}$  **defs by**  $\text{metis+}$

**have**  $1 < \text{length } XS_R$   $\text{length } XS_R < \text{length } xs$   $\text{distinct } XS_R$   
**using**  $\text{False } 1.\text{prems}(2)$  **defs by**  $(\text{auto simp: split-at-take-drop-conv})$   
**hence**  $C_{0R} \in \text{set } XS_R$   $C_{1R} \in \text{set } XS_R$  **and**  $IHR1: C_{0R} \neq C_{1R}$   
**using**  $1.IH$  **defs by**  $\text{metis+}$   
**hence**  $IHR2: C_{0R} \in \text{set } xs$   $C_{1R} \in \text{set } xs$   
**using**  $\text{split-at-take-drop-conv in-set-dropD snd-conv}$  **defs by**  $\text{metis+}$

**have**  $*$ :  $(YS, C_0, C_1) = \text{closest-pair-rec } xs$   
**using**  $\text{False closest-pair-rec-simps}$  **defs by**  $(\text{auto simp: Let-def split: prod.split})$   
**have**  $YS$ :  $\text{set } xs = \text{set } YS$   $\text{distinct } YS$   
**using**  $1.\text{prems}(2)$   $\text{closest-pair-rec-set-length-sorted-snd closest-pair-rec-distinct}$   
 $*$  **by**  $\text{blast+}$

**have**  $C_0 \in \text{set } xs$   $C_1 \in \text{set } xs$   
**using**  $\text{combine-set IHL2 IHR2 } YS$  **defs by**  $\text{blast+}$   
**moreover have**  $C_0 \neq C_1$   
**using**  $\text{combine-c0-ne-c1 IHL1}(1)$   $IHR1(1)$   $YS$  **defs by**  $\text{blast}$   
**ultimately show**  $?thesis$   
**using**  $1.\text{prems}(3)$   $*$  **by**  $(\text{metis Pair-inject})$

**qed**  
**qed**

**lemma**  $\text{closest-pair-rec-dist}$ :  
**assumes**  $1 < \text{length } xs$   $\text{distinct } xs$   $\text{sorted-fst } xs$   $(ys, c_0, c_1) = \text{closest-pair-rec } xs$   
**shows**  $\text{sparse } (\text{dist } c_0 c_1) (\text{set } xs)$   
**using**  $\text{assms}$   
**proof**  $(\text{induction } xs \text{ arbitrary: } ys c_0 c_1 \text{ rule: length-induct})$   
**case**  $(1 xs)$   
**let**  $?n = \text{length } xs$   
**show**  $?case$   
**proof**  $(\text{cases } ?n \leq 3)$   
**case**  $\text{True}$   
**hence**  $(c_0, c_1) = \text{closest-pair-bf } xs$   
**using**  $1.\text{prems}(4)$   $\text{closest-pair-rec.simps}$  **by**  $\text{simp}$   
**thus**  $?thesis$   
**using**  $1.\text{prems}(1,4)$   $\text{closest-pair-bf-dist}$  **by**  $\text{metis}$   
**next**

**case** *False*

**obtain**  $XS_L XS_R$  **where**  $XS_{LR}$ -def:  $(XS_L, XS_R) = \text{split-at } (?n \text{ div } 2) \text{ } xs$   
**using** *prod.collapse by blast*  
**define**  $L$  **where**  $L = \text{fst } (\text{hd } XS_R)$

**obtain**  $YS_L C_{0L} C_{1L}$  **where**  $YSC_{01L}$ -def:  $(YS_L, C_{0L}, C_{1L}) = \text{closest-pair-rec } XS_L$   
**using** *prod.collapse by metis*  
**obtain**  $YS_R C_{0R} C_{1R}$  **where**  $YSC_{01R}$ -def:  $(YS_R, C_{0R}, C_{1R}) = \text{closest-pair-rec } XS_R$   
**using** *prod.collapse by metis*

**define**  $YS$  **where**  $YS = \text{merge } (\lambda p. \text{snd } p) \text{ } YS_L \text{ } YS_R$   
**obtain**  $C_0 C_1$  **where**  $C_{01}$ -def:  $(C_0, C_1) = \text{combine } (C_{0L}, C_{1L}) (C_{0R}, C_{1R})$   
 $L \text{ } YS$   
**using** *prod.collapse by metis*  
**note**  $\text{defs} = XS_{LR}$ -def  $L$ -def  $YSC_{01L}$ -def  $YSC_{01R}$ -def  $YS$ -def  $C_{01}$ -def

**have**  $XSLR$ :  $XS_L = \text{take } (?n \text{ div } 2) \text{ } xs$   $XS_R = \text{drop } (?n \text{ div } 2) \text{ } xs$   
**using**  $\text{defs}$  **by** *(auto simp: split-at-take-drop-conv)*

**have**  $1 < \text{length } XS_L$   $\text{length } XS_L < \text{length } xs$   
**using** *False XSLR by simp-all*  
**moreover** **have** *distinct*  $XS_L$  *sorted-fst*  $XS_L$   
**using**  $1.\text{prems}(2,3)$   $XSLR$  **by** *(auto simp: sorted-fst-def sorted-wrt-take)*  
**ultimately** **have**  $L$ : *sparse*  $(\text{dist } C_{0L} \text{ } C_{1L})$   $(\text{set } XS_L)$   
 $\text{set } XS_L = \text{set } YS_L$   
**using**  $1$  *closest-pair-rec-set-length-sorted-snd closest-pair-rec-c0-c1*  
 $YSC_{01L}$ -def **by** *blast+*

**hence**  $IHL$ : *sparse*  $(\text{dist } C_{0L} \text{ } C_{1L})$   $(\text{set } YS_L)$   
**by** *argo*

**have**  $1 < \text{length } XS_R$   $\text{length } XS_R < \text{length } xs$   
**using** *False XSLR by simp-all*  
**moreover** **have** *distinct*  $XS_R$  *sorted-fst*  $XS_R$   
**using**  $1.\text{prems}(2,3)$   $XSLR$  **by** *(auto simp: sorted-fst-def sorted-wrt-drop)*  
**ultimately** **have**  $R$ : *sparse*  $(\text{dist } C_{0R} \text{ } C_{1R})$   $(\text{set } XS_R)$   
 $\text{set } XS_R = \text{set } YS_R$   
**using**  $1$  *closest-pair-rec-set-length-sorted-snd closest-pair-rec-c0-c1*  
 $YSC_{01R}$ -def **by** *blast+*

**hence**  $IHR$ : *sparse*  $(\text{dist } C_{0R} \text{ } C_{1R})$   $(\text{set } YS_R)$   
**by** *argo*

**have**  $*$ :  $(YS, C_0, C_1) = \text{closest-pair-rec } xs$   
**using** *False closest-pair-rec-simps defs* **by** *(auto simp: Let-def split: prod.split)*

**have**  $\text{set } xs = \text{set } YS$  *distinct*  $YS$  *sorted-snd*  $YS$   
**using**  $1.\text{prems}(2)$  *closest-pair-rec-set-length-sorted-snd closest-pair-rec-distinct*

```

* by blast+
  moreover have  $\forall p \in \text{set } YS_L. \text{fst } p \leq L$ 
    using False 1.premis(3) XSLR L-def L(2) sorted-fst-take-less-hd-drop by simp
  moreover have  $\forall p \in \text{set } YS_R. L \leq \text{fst } p$ 
    using False 1.premis(3) XSLR L-def R(2) sorted-fst-hd-drop-less-drop by simp
  moreover have  $\text{set } YS = \text{set } YS_L \cup \text{set } YS_R$ 
    using set-merge defs by fast
  moreover have  $(C_0, C_1) = \text{combine } (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) L YS$ 
    by (auto simp add: defs)
  ultimately have  $\text{sparse } (\text{dist } C_0 C_1) (\text{set } xs)$ 
    using combine-dist IHL IHR by auto
  moreover have  $(YS, C_0, C_1) = (ys, c_0, c_1)$ 
    using 1.premis(4) * by simp
  ultimately show ?thesis
    by blast
qed

```

```

fun closest-pair-tm :: point list  $\Rightarrow$  (point * point) tm where
  closest-pair-tm [] = 1 return undefined
| closest-pair-tm [-] = 1 return undefined
| closest-pair-tm ps = 1 (
  do {
    xs  $\leftarrow$  mergesort-tm fst ps;
    (-, p)  $\leftarrow$  closest-pair-rec-tm xs;
    return p
  }
)

```

```

fun closest-pair :: point list  $\Rightarrow$  (point * point) where
  closest-pair [] = undefined
| closest-pair [-] = undefined
| closest-pair ps = (let (-, c0, c1) = closest-pair-rec (mergesort fst ps) in (c0, c1))

```

```

lemma closest-pair-eq-val-closest-pair-tm:
  val (closest-pair-tm ps) = closest-pair ps
  by (induction ps rule: induct-list012)
  (auto simp del: closest-pair-rec-tm.simps mergesort-tm.simps
    simp add: closest-pair-rec-eq-val-closest-pair-rec-tm mergesort-eq-val-mergesort-tm
    split: prod.split)

```

```

lemma closest-pair-simps:
  1 < length ps  $\implies$  closest-pair ps = (let (-, c0, c1) = closest-pair-rec (mergesort
fst ps) in (c0, c1))
  by (induction ps rule: induct-list012) auto

```

```

declare closest-pair.simps [simp del]

```

```

theorem closest-pair-c0-c1:

```

**assumes**  $1 < \text{length } ps$   $\text{distinct } ps$   $(c_0, c_1) = \text{closest-pair } ps$   
**shows**  $c_0 \in \text{set } ps$   $c_1 \in \text{set } ps$   $c_0 \neq c_1$   
**using** *assms closest-pair-rec-c0-c1*[of mergesort fst ps]  
**by** (auto simp: mergesort closest-pair-simps split: prod.splits)

**theorem** *closest-pair-dist*:

**assumes**  $1 < \text{length } ps$   $\text{distinct } ps$   $(c_0, c_1) = \text{closest-pair } ps$   
**shows** *sparse* (dist  $c_0$   $c_1$ ) (set ps)  
**using** *assms closest-pair-rec-dist*[of mergesort fst ps] *closest-pair-rec-c0-c1*[of mergesort fst ps]  
**by** (auto simp: sorted-fst-def mergesort closest-pair-simps split: prod.splits)

## 3.2 Time Complexity Proof

### 3.2.1 Combine Step

**lemma** *time-find-closest-pair-tm*:

*time* (find-closest-pair-tm ( $c_0, c_1$ ) ps)  $\leq 17 * \text{length } ps + 1$

**proof** (induction ps rule: find-closest-pair-tm.induct)

**case** ( $\exists c_0 c_1 p_0 p_2 ps$ )

**let**  $?ps = p_2 \# ps$

**let**  $?p_1 = \text{val } (\text{find-closest-bf-tm } p_0 (\text{val } (\text{take-tm } 7 ?ps)))$

**have**  $*$ :  $\text{length } (\text{val } (\text{take-tm } 7 ?ps)) \leq 7$

**by** (subst take-eq-val-take-tm, simp)

**show**  $?case$

**proof** *cases*

**assume** *C1*:  $\text{dist } c_0 c_1 \leq \text{dist } p_0 ?p_1$

**hence** *time* (find-closest-pair-tm ( $c_0, c_1$ ) ( $p_0 \# ?ps$ )) =  $1 + \text{time } (\text{take-tm } 7 ?ps)$  +

*time* (find-closest-bf-tm  $p_0 (\text{val } (\text{take-tm } 7 ?ps)))$  + *time* (find-closest-pair-tm ( $c_0, c_1$ )  $?ps$ )

**by** (auto simp: time-simps)

**also have**  $\dots \leq 17 + \text{time } (\text{find-closest-pair-tm } (c_0, c_1) ?ps)$

**using** *time-take-tm*[of 7  $?ps$ ] *time-find-closest-bf-tm*[of  $p_0$   $\text{val } (\text{take-tm } 7 ?ps)$ ]

$*$  **by** *auto*

**also have**  $\dots \leq 17 + 17 * (\text{length } ?ps) + 1$

**using** 3.IH(1) *C1* **by** *simp*

**also have**  $\dots = 17 * \text{length } (p_0 \# ?ps) + 1$

**by** *simp*

**finally show**  $?thesis$  .

**next**

**assume** *C1*:  $\neg \text{dist } c_0 c_1 \leq \text{dist } p_0 ?p_1$

**hence** *time* (find-closest-pair-tm ( $c_0, c_1$ ) ( $p_0 \# ?ps$ )) =  $1 + \text{time } (\text{take-tm } 7 ?ps)$  +

*time* (find-closest-bf-tm  $p_0 (\text{val } (\text{take-tm } 7 ?ps)))$  + *time* (find-closest-pair-tm ( $p_0, ?p_1$ )  $?ps$ )

**by** (auto simp: time-simps)

**also have**  $\dots \leq 17 + \text{time } (\text{find-closest-pair-tm } (p_0, ?p_1) ?ps)$

**using** *time-take-tm*[of 7  $?ps$ ] *time-find-closest-bf-tm*[of  $p_0$   $\text{val } (\text{take-tm } 7 ?ps)$ ]

$*$  **by** *auto*

```

also have ...  $\leq 17 + 17 * (\text{length } ?ps) + 1$ 
using 3.IH(2) C1 by simp
also have ...  $= 17 * \text{length } (p_0 \# ?ps) + 1$ 
by simp
finally show ?thesis .
qed
qed (auto simp: time-simps)

```

**lemma** *time-combine-tm*:

```

fixes ps :: point list
shows time (combine-tm (p0L, p1L) (p0R, p1R) l ps)  $\leq 3 + 18 * \text{length } ps$ 
proof –
obtain c0 c1 where c-def:
  (c0, c1) = (if dist p0L p1L < dist p0R p1R then (p0L, p1L) else (p0R, p1R)) by
metis
let ?P = ( $\lambda p. \text{dist } p (l, \text{snd } p) < \text{dist } c_0 c_1$ )
define ps' where ps' = val (filter-tm ?P ps)
note defs = c-def ps'-def
hence time (combine-tm (p0L, p1L) (p0R, p1R) l ps) = 1 + time (filter-tm ?P
ps) +
time (find-closest-pair-tm (c0, c1) ps')
by (auto simp: combine-tm.simps Let-def time-simps split: prod.split)
also have ... = 2 + length ps + time (find-closest-pair-tm (c0, c1) ps')
using time-filter-tm by auto
also have ...  $\leq 3 + \text{length } ps + 17 * \text{length } ps'$ 
using defs time-find-closest-pair-tm by simp
also have ...  $\leq 3 + 18 * \text{length } ps$ 
unfolding ps'-def by (subst filter-eq-val-filter-tm, simp)
finally show ?thesis
by blast
qed

```

### 3.2.2 Divide and Conquer Algorithm

**lemma** *time-closest-pair-rec-tm-simps-1*:

```

assumes length xs  $\leq 3$ 
shows time (closest-pair-rec-tm xs) = 1 + time (length-tm xs) + time (mergesort-tm
snd xs) + time (closest-pair-bf-tm xs)
using assms by (auto simp: time-simps length-eq-val-length-tm)

```

**lemma** *time-closest-pair-rec-tm-simps-2*:

```

assumes  $\neg (\text{length } xs \leq 3)$ 
shows time (closest-pair-rec-tm xs) = 1 + (
  let (xsL, xsR) = val (split-at-tm (length xs div 2) xs) in
  let (ysL, pL) = val (closest-pair-rec-tm xsL) in
  let (ysR, pR) = val (closest-pair-rec-tm xsR) in
  let ys = val (merge-tm ( $\lambda p. \text{snd } p$ ) ysL ysR) in
  time (length-tm xs) + time (split-at-tm (length xs div 2) xs) + time (closest-pair-rec-tm
xsL) +

```

```

    time (closest-pair-rec-tm xsR) + time (merge-tm (λp. snd p) ysL ysR) + time
    (combine-tm pL pR (fst (hd xsR)) ys)
  )
  using assms
  apply (subst closest-pair-rec-tm.simps)
  by (auto simp del: closest-pair-rec-tm.simps
      simp add: time-simps length-eq-val-length-tm
      split: prod.split)

```

```

function closest-pair-recurrence :: nat ⇒ real where
  n ≤ 3 ⇒ closest-pair-recurrence n = 3 + n + mergesort-recurrence n + n * n
| 3 < n ⇒ closest-pair-recurrence n = 7 + 21 * n + closest-pair-recurrence (nat
[real n / 2]) +
  closest-pair-recurrence (nat [real n / 2])
  by force simp-all
termination by akra-bazzi-termination simp-all

```

```

lemma closest-pair-recurrence-nonneg[simp]:
  0 ≤ closest-pair-recurrence n
  by (induction n rule: closest-pair-recurrence.induct) auto

```

```

lemma time-closest-pair-rec-conv-closest-pair-recurrence:
  time (closest-pair-rec-tm ps) ≤ closest-pair-recurrence (length ps)
proof (induction ps rule: length-induct)
  case (1 ps)
  let ?n = length ps
  show ?case
  proof (cases ?n ≤ 3)
    case True
    hence time (closest-pair-rec-tm ps) = 1 + time (length-tm ps) + time (mergesort-tm
snd ps) + time (closest-pair-bf-tm ps)
    using time-closest-pair-rec-tm-simps-1 by simp
    moreover have closest-pair-recurrence ?n = 3 + ?n + mergesort-recurrence
?n + ?n * ?n
    using True by simp
    moreover have time (length-tm ps) ≤ 1 + ?n time (mergesort-tm snd ps) ≤
mergesort-recurrence ?n
    time (closest-pair-bf-tm ps) ≤ 1 + ?n * ?n
    using time-length-tm[of ps] time-mergesort-conv-mergesort-recurrence[of snd
ps] time-closest-pair-bf-tm[of ps] by auto
    ultimately show ?thesis
    by linarith
  next
  case False
  obtain XSL XSR where XS-def: (XSL, XSR) = val (split-at-tm (?n div 2)
ps)
  using prod.collapse by blast
  obtain YSL C0L C1L where CPL-def: (YSL, C0L, C1L) = val (closest-pair-rec-tm

```

$XS_L$ )  
**using** *prod.collapse by metis*  
**obtain**  $YS_R C_{0R} C_{1R}$  **where**  $CP_R\text{-def}: (YS_R, C_{0R}, C_{1R}) = \text{val} (\text{closest-pair-rec-tm } XS_R)$   
**using** *prod.collapse by metis*  
**define**  $YS$  **where**  $YS = \text{val} (\text{merge-tm } (\lambda p. \text{snd } p) YS_L YS_R)$   
**obtain**  $C_0 C_1$  **where**  $C_{01}\text{-def}: (C_0, C_1) = \text{val} (\text{combine-tm } (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) (\text{fst } (\text{hd } XS_R)) YS)$   
**using** *prod.collapse by metis*  
**note**  $\text{defs} = XS\text{-def } CP_L\text{-def } CP_R\text{-def } YS\text{-def } C_{01}\text{-def}$   
  
**have**  $XSLR: XS_L = \text{take } (?n \text{ div } 2) \text{ ps } XS_R = \text{drop } (?n \text{ div } 2) \text{ ps}$   
**using**  $\text{defs}$  **by** *(auto simp: split-at-take-drop-conv split-at-eq-val-split-at-tm)*  
**hence**  $\text{length } XS_L = ?n \text{ div } 2 \text{ length } XS_R = ?n - ?n \text{ div } 2$   
**by** *simp-all*  
**hence**  $*$ :  $(\text{nat } \lfloor \text{real } ?n / 2 \rfloor) = \text{length } XS_L (\text{nat } \lceil \text{real } ?n / 2 \rceil) = \text{length } XS_R$   
**by** *linarith+*  
**have**  $\text{length } XS_L = \text{length } YS_L \text{ length } XS_R = \text{length } YS_R$   
**using**  $\text{defs}$  *closest-pair-rec-set-length-sorted-snd closest-pair-rec-eq-val-closest-pair-rec-tm*  
**by** *metis+*  
**hence**  $L: ?n = \text{length } YS_L + \text{length } YS_R$   
**using**  $\text{defs}$   $XSLR$  **by** *fastforce*  
  
**have**  $1 < \text{length } XS_L \text{ length } XS_L < \text{length } \text{ps}$   
**using** *False XSLR by simp-all*  
**hence**  $\text{time } (\text{closest-pair-rec-tm } XS_L) \leq \text{closest-pair-recurrence } (\text{length } XS_L)$   
**using** *1.IH by simp*  
**hence**  $IHL: \text{time } (\text{closest-pair-rec-tm } XS_L) \leq \text{closest-pair-recurrence } (\text{nat } \lfloor \text{real } ?n / 2 \rfloor)$   
**using**  $*$  **by** *simp*  
  
**have**  $1 < \text{length } XS_R \text{ length } XS_R < \text{length } \text{ps}$   
**using** *False XSLR by simp-all*  
**hence**  $\text{time } (\text{closest-pair-rec-tm } XS_R) \leq \text{closest-pair-recurrence } (\text{length } XS_R)$   
**using** *1.IH by simp*  
**hence**  $IHR: \text{time } (\text{closest-pair-rec-tm } XS_R) \leq \text{closest-pair-recurrence } (\text{nat } \lceil \text{real } ?n / 2 \rceil)$   
**using**  $*$  **by** *simp*  
  
**have**  $(YS, C_0, C_1) = \text{val} (\text{closest-pair-rec-tm } \text{ps})$   
**using** *False closest-pair-rec-simps defs* **by** *(auto simp: Let-def length-eq-val-length-tm split!: prod.split)*  
**hence**  $\text{length } \text{ps} = \text{length } YS$   
**using** *closest-pair-rec-set-length-sorted-snd closest-pair-rec-eq-val-closest-pair-rec-tm*  
**by** *auto*  
**hence** *combine-bound: time (combine-tm (C<sub>0L</sub>, C<sub>1L</sub>) (C<sub>0R</sub>, C<sub>1R</sub>) (fst (hd XS<sub>R</sub>)) YS) ≤ 3 + 18 \* ?n*  
**using** *time-combine-tm* **by** *simp*  
**have**  $\text{time } (\text{closest-pair-rec-tm } \text{ps}) = 1 + \text{time } (\text{length-tm } \text{ps}) + \text{time } (\text{split-at-tm } \text{ps})$

```

(?n div 2) ps) +
  time (closest-pair-rec-tm XSL) + time (closest-pair-rec-tm XSR) + time
(merge-tm (λp. snd p) YSL YSR) +
  time (combine-tm (C0L, C1L) (C0R, C1R) (fst (hd XSR)) YS)
  using time-closest-pair-rec-tm-simps-2[OF False] defs
  by (auto simp del: closest-pair-rec-tm.simps simp add: Let-def split: prod.split)
  also have ... ≤ 7 + 21 * ?n + time (closest-pair-rec-tm XSL) + time
(closest-pair-rec-tm XSR)
  using time-merge-tm[of (λp. snd p) YSL YSR] L combine-bound by (simp
add: time-length-tm time-split-at-tm)
  also have ... ≤ 7 + 21 * ?n + closest-pair-recurrence (nat ⌊real ?n / 2⌋) +
closest-pair-recurrence (nat ⌈real ?n / 2⌉)
  using IHL IHR by simp
  also have ... = closest-pair-recurrence (length ps)
  using False by simp
  finally show ?thesis
  by simp
qed
qed

```

**theorem** *closest-pair-recurrence*:  
 $closest-pair-recurrence \in \Theta(\lambda n. n * \ln n)$   
by (master-theorem) auto

**theorem** *time-closest-pair-rec-bigo*:  
 $(\lambda xs. time (closest-pair-rec-tm xs)) \in O[length \text{ going-to at-top}]((\lambda n. n * \ln n) o$   
 $length)$   
**proof** –  
have 0:  $\bigwedge ps. time (closest-pair-rec-tm ps) \leq (closest-pair-recurrence o length)$   
 $ps$   
**unfolding** *comp-def* **using** *time-closest-pair-rec-conv-closest-pair-recurrence* **by**  
*auto*  
**show** ?thesis  
**using** *bigo-measure-trans*[OF 0] *bighetaD1*[OF *closest-pair-recurrence*] *of-nat-0-le-iff*  
**by** *blast*  
**qed**

**definition** *closest-pair-time* ::  $nat \Rightarrow real$  **where**  
 $closest-pair-time\ n = 1 + mergesort-recurrence\ n + closest-pair-recurrence\ n$

**lemma** *time-closest-pair-conv-closest-pair-recurrence*:  
 $time (closest-pair-tm ps) \leq closest-pair-time (length ps)$   
**unfolding** *closest-pair-time-def*  
**proof** (*induction rule: induct-list012*)  
**case** (3 x y zs)  
**let** ?ps = x # y # zs  
**define** xs **where** xs = val (mergesort-tm fst ?ps)  
**have** \*:  $length\ xs = length\ ?ps$   
**using** *xs-def mergesort(3)*[of fst ?ps] *mergesort-eq-val-mergesort-tm* **by** *metis*

```

have time (closest-pair-tm ?ps) = 1 + time (mergesort-tm fst ?ps) + time
(closest-pair-rec-tm xs)
using xs-def by (auto simp del: mergesort-tm.simps closest-pair-rec-tm.simps
simp add: time-simps split: prod.split)
also have ... ≤ 1 + mergesort-recurrence (length ?ps) + time (closest-pair-rec-tm
xs)
using time-mergesort-conv-mergesort-recurrence[of fst ?ps] by simp
also have ... ≤ 1 + mergesort-recurrence (length ?ps) + closest-pair-recurrence
(length ?ps)
using time-closest-pair-rec-conv-closest-pair-recurrence[of xs] * by auto
finally show ?case
by blast
qed (auto simp: time-simps)

```

**corollary** *closest-pair-time*:

```

closest-pair-time ∈ O(λn. n * ln n)
unfolding closest-pair-time-def
using mergesort-recurrence closest-pair-recurrence sum-in-bigo(1) const-1-bigo-n-ln-n
by blast

```

**corollary** *time-closest-pair-bigo*:

```

(λps. time (closest-pair-tm ps)) ∈ O[length going-to at-top](λn. n * ln n) o
length)
proof –
have 0: ∧ps. time (closest-pair-tm ps) ≤ (closest-pair-time o length) ps
unfolding comp-def using time-closest-pair-conv-closest-pair-recurrence by
auto
show ?thesis
using bigo-measure-trans[OF 0] closest-pair-time by simp
qed

```

### 3.3 Code Export

#### 3.3.1 Combine Step

```

fun find-closest-pair-code :: (int * point * point) ⇒ point list ⇒ (int * point *
point) where
  find-closest-pair-code (δ, c0, c1) [] = (δ, c0, c1)
| find-closest-pair-code (δ, c0, c1) [p] = (δ, c0, c1)
| find-closest-pair-code (δ, c0, c1) (p0 # ps) = (
  let (δ', p1) = find-closest-bf-code p0 (take 7 ps) in
  if δ ≤ δ' then
    find-closest-pair-code (δ, c0, c1) ps
  else
    find-closest-pair-code (δ', p0, p1) ps
)

```

**lemma** *find-closest-pair-code-dist-eq*:

```

assumes δ = dist-code c0 c1 (Δ, C0, C1) = find-closest-pair-code (δ, c0, c1) ps
shows Δ = dist-code C0 C1

```

```

using assms
proof (induction  $(\delta, c_0, c_1)$  ps arbitrary:  $\delta \ c_0 \ c_1 \ \Delta \ C_0 \ C_1$  rule: find-closest-pair-code.induct)
  case ( $\exists \ \delta \ c_0 \ c_1 \ p_0 \ p_2 \ ps$ )
    obtain  $\delta' \ p_1$  where  $\delta'$ -def:  $(\delta', p_1) = \text{find-closest-bf-code } p_0 \ (\text{take } 7 \ (p_2 \ \# \ ps))$ 
      by (metis surj-pair)
    hence  $A: \delta' = \text{dist-code } p_0 \ p_1$ 
      using find-closest-bf-code-dist-eq[of take 7 (p2 # ps)] by simp
    show ?case
    proof (cases  $\delta \leq \delta'$ )
      case True
        obtain  $\Delta' \ C_0' \ C_1'$  where  $\Delta'$ -def:  $(\Delta', C_0', C_1') = \text{find-closest-pair-code } (\delta,$ 
 $c_0, c_1) \ (p_2 \ \# \ ps)$ 
          by (metis prod-cases4)
        note  $\text{defs} = \delta'$ -def  $\Delta'$ -def
        hence  $\Delta' = \text{dist-code } C_0' \ C_1'$ 
          using  $\exists$ .hyps(1)[of  $(\delta', p_1) \ \delta' \ p_1]$   $\exists$ .prems(1) True  $\delta'$ -def by blast
        moreover have  $\Delta = \Delta' \ C_0 = C_0' \ C_1 = C_1'$ 
          using  $\text{defs } \text{True } \exists$ .prems(2) apply (auto split: prod.splits) by (metis Pair-inject)+
        ultimately show ?thesis
          by simp
      next
        case False
          obtain  $\Delta' \ C_0' \ C_1'$  where  $\Delta'$ -def:  $(\Delta', C_0', C_1') = \text{find-closest-pair-code } (\delta',$ 
 $p_0, p_1) \ (p_2 \ \# \ ps)$ 
            by (metis prod-cases4)
          note  $\text{defs} = \delta'$ -def  $\Delta'$ -def
          hence  $\Delta' = \text{dist-code } C_0' \ C_1'$ 
            using  $\exists$ .hyps(2)[of  $(\delta', p_1) \ \delta' \ p_1]$  A False  $\delta'$ -def by blast
          moreover have  $\Delta = \Delta' \ C_0 = C_0' \ C_1 = C_1'$ 
            using  $\text{defs } \text{False } \exists$ .prems(2) apply (auto split: prod.splits) by (metis Pair-inject)+
          ultimately show ?thesis
            by simp
        qed
    qed auto

declare find-closest-pair.simps [simp add]

lemma find-closest-pair-code-eq:
  assumes  $\delta = \text{dist } c_0 \ c_1 \ \delta' = \text{dist-code } c_0 \ c_1$ 
  assumes  $(C_0, C_1) = \text{find-closest-pair } (c_0, c_1) \ ps$ 
  assumes  $(\Delta', C_0', C_1') = \text{find-closest-pair-code } (\delta', c_0, c_1) \ ps$ 
  shows  $C_0 = C_0' \wedge C_1 = C_1'$ 
  using assms
proof (induction  $(c_0, c_1)$  ps arbitrary:  $\delta \ \delta' \ c_0 \ c_1 \ C_0 \ C_1 \ \Delta' \ C_0' \ C_1'$  rule: find-closest-pair.induct)
  case ( $\exists \ c_0 \ c_1 \ p_0 \ p_2 \ ps$ )
    obtain  $p_1 \ \delta_p' \ p_1'$  where  $\delta_p'$ -def:  $p_1 = \text{find-closest-bf } p_0 \ (\text{take } 7 \ (p_2 \ \# \ ps))$ 
       $(\delta_p', p_1') = \text{find-closest-bf-code } p_0 \ (\text{take } 7 \ (p_2 \ \# \ ps))$ 
      by (metis surj-pair)
    hence  $A: \delta_p' = \text{dist-code } p_0 \ p_1'$ 

```

```

    using find-closest-bf-code-dist-eq[of take 7 (p2 # ps)] by simp
  have B: p1 = p1'
    using 3.prem(1,2,3)  $\delta_p$ -def find-closest-bf-code-eq by auto
  show ?case
  proof (cases  $\delta \leq \text{dist } p_0 \ p_1$ )
    case True
      hence C:  $\delta' \leq \delta_p'$ 
        by (simp add: 3.prem(1,2) A B dist-eq-dist-code-le)
      obtain C0i C1i  $\Delta_i'$  C0i' C1i' where  $\Delta_i$ -def:
        (C0i, C1i) = find-closest-pair (c0, c1) (p2 # ps)
        ( $\Delta_i'$ , C0i', C1i') = find-closest-pair-code ( $\delta'$ , c0, c1) (p2 # ps)
        by (metis prod-cases3)
      note defs =  $\delta_p$ -def  $\Delta_i$ -def
      have C0i = C0i'  $\wedge$  C1i = C1i'
        using 3.hyps(1)[of p1] 3.prem True defs by blast
      moreover have C0 = C0i C1 = C1i
        using defs(1,3) True 3.prem(1,3) apply (auto split: prod.splits) by (metis
Pair-inject)+
      moreover have  $\Delta' = \Delta_i' \ C_0' = C_{0i}' \ C_1' = C_{1i}'$ 
        using defs(2,4) C 3.prem(4) apply (auto split: prod.splits) by (metis
Pair-inject)+
      ultimately show ?thesis
        by simp
    next
      case False
      hence C:  $\neg \delta' \leq \delta_p'$ 
        by (simp add: 3.prem(1,2) A B dist-eq-dist-code-le)
      obtain C0i C1i  $\Delta_i'$  C0i' C1i' where  $\Delta_i$ -def:
        (C0i, C1i) = find-closest-pair (p0, p1) (p2 # ps)
        ( $\Delta_i'$ , C0i', C1i') = find-closest-pair-code ( $\delta_p'$ , p0, p1') (p2 # ps)
        by (metis prod-cases3)
      note defs =  $\delta_p$ -def  $\Delta_i$ -def
      have C0i = C0i'  $\wedge$  C1i = C1i'
        using 3.prem 3.hyps(2)[of p1] A B False defs by blast
      moreover have C0 = C0i C1 = C1i
        using defs(1,3) False 3.prem(1,3) apply (auto split: prod.splits) by (metis
Pair-inject)+
      moreover have  $\Delta' = \Delta_i' \ C_0' = C_{0i}' \ C_1' = C_{1i}'$ 
        using defs(2,4) C 3.prem(4) apply (auto split: prod.splits) by (metis
Pair-inject)+
      ultimately show ?thesis
        by simp
  qed
qed auto

fun combine-code :: (int * point * point)  $\Rightarrow$  (int * point * point)  $\Rightarrow$  int  $\Rightarrow$  point
list  $\Rightarrow$  (int * point * point) where
  combine-code ( $\delta_L$ , p0L, p1L) ( $\delta_R$ , p0R, p1R) l ps = (
    let ( $\delta$ , c0, c1) = if  $\delta_L < \delta_R$  then ( $\delta_L$ , p0L, p1L) else ( $\delta_R$ , p0R, p1R) in

```

let  $ps' = \text{filter } (\lambda p. (\text{fst } p - l)^2 < \delta) ps$  in  
*find-closest-pair-code*  $(\delta, c_0, c_1) ps'$   
 )

**lemma** *combine-code-dist-eq*:

**assumes**  $\delta_L = \text{dist-code } p_{0L} p_{1L}$   $\delta_R = \text{dist-code } p_{0R} p_{1R}$   
**assumes**  $(\delta, c_0, c_1) = \text{combine-code } (\delta_L, p_{0L}, p_{1L}) (\delta_R, p_{0R}, p_{1R}) \text{ l } ps$   
**shows**  $\delta = \text{dist-code } c_0 c_1$   
**using** *assms* **by** (*auto simp: find-closest-pair-code-dist-eq split: if-splits*)

**lemma** *combine-code-eq*:

**assumes**  $\delta_L' = \text{dist-code } p_{0L} p_{1L}$   $\delta_R' = \text{dist-code } p_{0R} p_{1R}$   
**assumes**  $(c_0, c_1) = \text{combine } (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) \text{ l } ps$   
**assumes**  $(\delta', c_0', c_1') = \text{combine-code } (\delta_L', p_{0L}, p_{1L}) (\delta_R', p_{0R}, p_{1R}) \text{ l } ps$   
**shows**  $c_0 = c_0' \wedge c_1 = c_1'$

**proof** –

**obtain**  $C_{0i} C_{1i} \Delta_i' C_{0i}' C_{1i}'$  **where**  $\Delta_i$ -def:

$(C_{0i}, C_{1i}) = (\text{if } \text{dist } p_{0L} p_{1L} < \text{dist } p_{0R} p_{1R} \text{ then } (p_{0L}, p_{1L}) \text{ else } (p_{0R}, p_{1R}))$   
 $(\Delta_i', C_{0i}', C_{1i}') = (\text{if } \delta_L' < \delta_R' \text{ then } (\delta_L', p_{0L}, p_{1L}) \text{ else } (\delta_R', p_{0R}, p_{1R}))$   
**by** *metis*

**define**  $ps' ps''$  **where**  $ps'$ -def:

$ps' = \text{filter } (\lambda p. \text{dist } p (l, \text{snd } p) < \text{dist } C_{0i} C_{1i}) ps$   
 $ps'' = \text{filter } (\lambda p. (\text{fst } p - l)^2 < \Delta_i') ps$

**obtain**  $C_0 C_1 \Delta' C_0' C_1'$  **where**  $\Delta$ -def:

$(C_0, C_1) = \text{find-closest-pair } (C_{0i}, C_{1i}) ps'$   
 $(\Delta', C_0', C_1') = \text{find-closest-pair-code } (\Delta_i', C_{0i}', C_{1i}') ps''$   
**by** (*metis prod-cases3*)

**note**  $\text{defs} = \Delta_i\text{-def } ps'\text{-def } \Delta\text{-def}$

**have** \*:  $C_{0i} = C_{0i}' C_{1i} = C_{1i}' \Delta_i' = \text{dist-code } C_{0i}' C_{1i}'$

**using**  $\Delta_i\text{-def } \text{assms}(1,2,3,4)$  *dist-eq-dist-code-lt* **by** (*auto split: if-splits*)

**hence**  $\bigwedge p. |\text{fst } p - l| < \text{dist } C_{0i} C_{1i} \iff (\text{fst } p - l)^2 < \Delta_i'$

**using** *dist-eq-dist-code-abs-lt* **by** (*metis (mono-tags) of-int-abs*)

**hence**  $ps' = ps''$

**using**  $ps'\text{-def } \text{dist-fst-abs}$  **by** *auto*

**hence**  $C_0 = C_0' C_1 = C_1'$

**using** \* *find-closest-pair-code-eq*  $\Delta\text{-def}$  **by** *blast+*

**moreover** **have**  $C_0 = c_0 C_1 = c_1$

**using** *assms*(3)  $\text{defs}(1,3,5)$  **apply** (*auto simp: combine.simps split: prod.splits*)

**by** (*metis Pair-inject*)+

**moreover** **have**  $C_0' = c_0' C_1' = c_1'$

**using** *assms*(4)  $\text{defs}(2,4,6)$  **apply** (*auto split: prod.splits*) **by** (*metis prod.inject*)+

**ultimately** **show** *?thesis*

**by** *blast*

**qed**

### 3.3.2 Divide and Conquer Algorithm

**function** *closest-pair-rec-code* :: *point list*  $\Rightarrow$  (*point list* \* *int* \* *point* \* *point*)  
**where**

```

closest-pair-rec-code xs = (
  let n = length xs in
  if n ≤ 3 then
    (mergesort snd xs, closest-pair-bf-code xs)
  else
    let (xsL, xsR) = split-at (n div 2) xs in
    let l = fst (hd xsR) in

    let (ysL, pL) = closest-pair-rec-code xsL in
    let (ysR, pR) = closest-pair-rec-code xsR in

    let ys = merge snd ysL ysR in
    (ys, combine-code pL pR l ys)
)
by pat-completeness auto
termination closest-pair-rec-code
by (relation Wellfounded.measure (λxs. length xs))
  (auto simp: split-at-take-drop-conv Let-def)

lemma closest-pair-rec-code-simps:
assumes n = length xs ∧ (n ≤ 3)
shows closest-pair-rec-code xs = (
  let (xsL, xsR) = split-at (n div 2) xs in
  let l = fst (hd xsR) in
  let (ysL, pL) = closest-pair-rec-code xsL in
  let (ysR, pR) = closest-pair-rec-code xsR in
  let ys = merge snd ysL ysR in
  (ys, combine-code pL pR l ys)
)
using assms by (auto simp: Let-def)

declare combine.simps combine-code.simps closest-pair-rec-code.simps [simp del]

lemma closest-pair-rec-code-dist-eq:
assumes 1 < length xs (ys, δ, c0, c1) = closest-pair-rec-code xs
shows δ = dist-code c0 c1
using assms
proof (induction xs arbitrary: ys δ c0 c1 rule: length-induct)
case (1 xs)
let ?n = length xs
show ?case
proof (cases ?n ≤ 3)
case True
hence (δ, c0, c1) = closest-pair-bf-code xs
using 1.prem(2) closest-pair-rec-code.simps by simp
thus ?thesis
using 1.prem(1) closest-pair-bf-code-dist-eq by simp
next
case False

```

**obtain**  $XS_L XS_R$  **where**  $XS_{LR}$ -def:  $(XS_L, XS_R) = \text{split-at } (?n \text{ div } 2) \text{ } xs$   
**using** *prod.collapse* **by** *blast*  
**define**  $L$  **where**  $L = \text{fst } (\text{hd } XS_R)$

**obtain**  $YS_L \Delta_L C_{0L} C_{1L}$  **where**  $YSC_{01L}$ -def:  $(YS_L, \Delta_L, C_{0L}, C_{1L}) =$   
*closest-pair-rec-code*  $XS_L$   
**using** *prod.collapse* **by** *metis*  
**obtain**  $YS_R \Delta_R C_{0R} C_{1R}$  **where**  $YSC_{01R}$ -def:  $(YS_R, \Delta_R, C_{0R}, C_{1R}) =$   
*closest-pair-rec-code*  $XS_R$   
**using** *prod.collapse* **by** *metis*

**define**  $YS$  **where**  $YS = \text{merge } (\lambda p. \text{snd } p) \text{ } YS_L \text{ } YS_R$   
**obtain**  $\Delta C_0 C_1$  **where**  $C_{01}$ -def:  $(\Delta, C_0, C_1) = \text{combine-code } (\Delta_L, C_{0L},$   
 $C_{1L}) (\Delta_R, C_{0R}, C_{1R}) \text{ } L \text{ } YS$   
**using** *prod.collapse* **by** *metis*  
**note**  $\text{defs} = XS_{LR}$ -def  $L$ -def  $YSC_{01L}$ -def  $YSC_{01R}$ -def  $YS$ -def  $C_{01}$ -def

**have**  $1 < \text{length } XS_L \text{ length } XS_L < \text{length } xs$   
**using** *False 1.premis(1) defs* **by** *(auto simp: split-at-take-drop-conv)*  
**hence**  $IHL: \Delta_L = \text{dist-code } C_{0L} \text{ } C_{1L}$   
**using** *1.IH defs* **by** *metis+*

**have**  $1 < \text{length } XS_R \text{ length } XS_R < \text{length } xs$   
**using** *False 1.premis(1) defs* **by** *(auto simp: split-at-take-drop-conv)*  
**hence**  $IHR: \Delta_R = \text{dist-code } C_{0R} \text{ } C_{1R}$   
**using** *1.IH defs* **by** *metis+*

**have**  $*$ :  $(YS, \Delta, C_0, C_1) = \text{closest-pair-rec-code } xs$   
**using** *False closest-pair-rec-code-simps defs* **by** *(auto simp: Let-def split: prod.split)*  
**moreover** **have**  $\Delta = \text{dist-code } C_0 \text{ } C_1$   
**using** *combine-code-dist-eq IHL IHR C\_{01}-def* **by** *blast*  
**ultimately** **show** *?thesis*  
**using** *1.premis(2) \** **by** *(metis Pair-inject)*

**qed**  
**qed**

**lemma** *closest-pair-rec-ys-eq*:  
**assumes**  $1 < \text{length } xs$   
**assumes**  $(ys, c_0, c_1) = \text{closest-pair-rec } xs$   
**assumes**  $(ys', \delta', c_0', c_1') = \text{closest-pair-rec-code } xs$   
**shows**  $ys = ys'$   
**using** *assms*

**proof** *(induction xs arbitrary: ys c\_0 c\_1 ys' delta' c\_0' c\_1' rule: length-induct)*  
**case** *(1 xs)*  
**let**  $?n = \text{length } xs$   
**show** *?case*  
**proof** *(cases ?n ≤ 3)*

**case** *True*  
**hence**  $ys = \text{mergesort } \text{snd } xs$   
**using**  $1.\text{prems}(2)$  *closest-pair-rec.simps* **by** *simp*  
**moreover have**  $ys' = \text{mergesort } \text{snd } xs$   
**using**  $1.\text{prems}(3)$  *closest-pair-rec-code.simps* **by** (*simp add: True*)  
**ultimately show** *?thesis*  
**using**  $1.\text{prems}(1)$  **by** *simp*  
**next**  
**case** *False*

**obtain**  $XS_L XS_R$  **where**  $XS_{LR}\text{-def}: (XS_L, XS_R) = \text{split-at } (?n \text{ div } 2) xs$   
**using** *prod.collapse* **by** *blast*  
**define**  $L$  **where**  $L = \text{fst } (\text{hd } XS_R)$

**obtain**  $YS_L C_{0L} C_{1L} YS_L' \Delta_L' C_{0L}' C_{1L}'$  **where**  $YSC_{01L}\text{-def}$ :  
 $(YS_L, C_{0L}, C_{1L}) = \text{closest-pair-rec } XS_L$   
 $(YS_L', \Delta_L', C_{0L}', C_{1L}') = \text{closest-pair-rec-code } XS_L$   
**using** *prod.collapse* **by** *metis*

**obtain**  $YS_R C_{0R} C_{1R} YS_R' \Delta_R' C_{0R}' C_{1R}'$  **where**  $YSC_{01R}\text{-def}$ :  
 $(YS_R, C_{0R}, C_{1R}) = \text{closest-pair-rec } XS_R$   
 $(YS_R', \Delta_R', C_{0R}', C_{1R}') = \text{closest-pair-rec-code } XS_R$   
**using** *prod.collapse* **by** *metis*

**define**  $YS YS'$  **where**  $YS\text{-def}$ :  
 $YS = \text{merge } (\lambda p. \text{snd } p) YS_L YS_R$   
 $YS' = \text{merge } (\lambda p. \text{snd } p) YS_L' YS_R'$

**obtain**  $C_0 C_1 \Delta' C_0' C_1'$  **where**  $C_{01}\text{-def}$ :  
 $(C_0, C_1) = \text{combine } (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) L YS$   
 $(\Delta', C_0', C_1') = \text{combine-code } (\Delta_L', C_{0L}', C_{1L}') (\Delta_R', C_{0R}', C_{1R}') L YS'$   
**using** *prod.collapse* **by** *metis*

**note**  $\text{defs} = XS_{LR}\text{-def } L\text{-def } YSC_{01L}\text{-def } YSC_{01R}\text{-def } YS\text{-def } C_{01}\text{-def}$

**have**  $1 < \text{length } XS_L \text{ length } XS_L < \text{length } xs$   
**using** *False 1.prems(1) defs* **by** (*auto simp: split-at-take-drop-conv*)  
**hence** *IHL*:  $YS_L = YS_L'$   
**using**  $1.IH$  *defs* **by** *metis*

**have**  $1 < \text{length } XS_R \text{ length } XS_R < \text{length } xs$   
**using** *False 1.prems(1) defs* **by** (*auto simp: split-at-take-drop-conv*)  
**hence** *IHR*:  $YS_R = YS_R'$   
**using**  $1.IH$  *defs* **by** *metis*

**have**  $(YS, C_0, C_1) = \text{closest-pair-rec } xs$   
**using** *False closest-pair-rec-simps defs(1,2,3,5,7,9)*  
**by** (*auto simp: Let-def split: prod.split*)

**moreover have**  $(YS', \Delta', C_0', C_1') = \text{closest-pair-rec-code } xs$   
**using** *False closest-pair-rec-code-simps defs(1,2,4,6,8,10)*  
**by** (*auto simp: Let-def split: prod.split*)

**moreover have**  $YS = YS'$

```

    using IHL IHR YS-def by simp
    ultimately show ?thesis
    by (metis 1.prem(2,3) Pair-inject)
qed
qed

```

**lemma** *closest-pair-rec-code-eq*:

```

assumes 1 < length xs
assumes (ys, c0, c1) = closest-pair-rec xs
assumes (ys', δ', c0', c1') = closest-pair-rec-code xs
shows c0 = c0' ∧ c1 = c1'
using assms
proof (induction xs arbitrary: ys c0 c1 ys' δ' c0' c1' rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    hence (c0, c1) = closest-pair-bf xs
    using 1.prem(2) closest-pair-rec.simps by simp
    moreover have (δ', c0', c1') = closest-pair-bf-code xs
    using 1.prem(3) closest-pair-rec-code.simps by (simp add: True)
    ultimately show ?thesis
    using 1.prem(1) closest-pair-bf-code-eq by simp
  next
  case False

```

```

obtain XSL XSR where XSLR-def: (XSL, XSR) = split-at (?n div 2) xs
  using prod.collapse by blast
define L where L = fst (hd XSR)

```

```

obtain YSL C0L C1L YS'L Δ'L C0L' C1L' where YSC01L-def:
  (YSL, C0L, C1L) = closest-pair-rec XSL
  (YS'L, Δ'L, C0L', C1L') = closest-pair-rec-code XSL
  using prod.collapse by metis

```

```

obtain YSR C0R C1R YS'R Δ'R C0R' C1R' where YSC01R-def:
  (YSR, C0R, C1R) = closest-pair-rec XSR
  (YS'R, Δ'R, C0R', C1R') = closest-pair-rec-code XSR
  using prod.collapse by metis

```

**define** YS YS' **where** YS-def:

```

YS = merge (λp. snd p) YSL YSR
YS' = merge (λp. snd p) YS'L YS'R

```

**obtain** C<sub>0</sub> C<sub>1</sub> Δ' C<sub>0</sub>' C<sub>1</sub>' **where** C<sub>01</sub>-def:

```

(C0, C1) = combine (C0L, C1L) (C0R, C1R) L YS
(Δ', C0', C1') = combine-code (Δ'L, C0L', C1L') (Δ'R, C0R', C1R') L YS'
  using prod.collapse by metis

```

**note** defs = XS<sub>LR</sub>-def L-def YSC<sub>01L</sub>-def YSC<sub>01R</sub>-def YS-def C<sub>01</sub>-def

```

have 1 < length XS_L length XS_L < length xs
  using False 1.premis(1) defs by (auto simp: split-at-take-drop-conv)
hence IHL: C0L = C0L' C1L = C1L'
  using 1.IH defs by metis+

have 1 < length XS_R length XS_R < length xs
  using False 1.premis(1) defs by (auto simp: split-at-take-drop-conv)
hence IHR: C0R = C0R' C1R = C1R'
  using 1.IH defs by metis+

have YS = YS'
  using defs ⟨1 < length XS_L⟩ ⟨1 < length XS_R⟩ closest-pair-rec-ys-eq by blast
moreover have ΔL' = dist-code C0L' C1L' ΔR' = dist-code C0R' C1R'
  using defs ⟨1 < length XS_L⟩ ⟨1 < length XS_R⟩ closest-pair-rec-code-dist-eq
by blast+
ultimately have C0 = C0' C1 = C1'
  using combine-code-eq IHL IHR C01-def by blast+
moreover have (YS, C0, C1) = closest-pair-rec xs
  using False closest-pair-rec-simps defs(1,2,3,5,7,9)
  by (auto simp: Let-def split: prod.split)
moreover have (YS', Δ', C0', C1') = closest-pair-rec-code xs
  using False closest-pair-rec-code-simps defs(1,2,4,6,8,10)
  by (auto simp: Let-def split: prod.split)
ultimately show ?thesis
  using 1.premis(2,3) by (metis Pair-inject)
qed
qed

declare closest-pair.simps [simp add]

fun closest-pair-code :: point list ⇒ (point * point) where
  closest-pair-code [] = undefined
| closest-pair-code [-] = undefined
| closest-pair-code ps = (let (-, -, c0, c1) = closest-pair-rec-code (mergesort fst ps)
  in (c0, c1))

lemma closest-pair-code-eq:
  closest-pair ps = closest-pair-code ps
proof (induction ps rule: induct-list012)
case (3 x y zs)
obtain ys c0 c1 ys' δ' c0' c1' where *:
  (ys, c0, c1) = closest-pair-rec (mergesort fst (x # y # zs))
  (ys', δ', c0', c1') = closest-pair-rec-code (mergesort fst (x # y # zs))
  by (metis prod-cases3)
moreover have 1 < length (mergesort fst (x # y # zs))
  using length-mergesort[of fst x # y # zs] by simp
ultimately have c0 = c0' c1 = c1'
  using closest-pair-rec-code-eq by blast+
thus ?case

```

```
    using * by (auto split: prod.splits)
qed auto

export-code closest-pair-code in OCaml
  module-name Verified

end
```

## References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.