

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

1	Common	3
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>	5
1.1.5	<i>rev</i>	5
1.1.6	<i>take</i>	6
1.1.7	<i>filter</i>	6
1.1.8	<i>split-at</i>	7
1.2	Mergesort	8
1.2.1	Functional Correctness Proof	8
1.2.2	Time Complexity Proof	11
1.2.3	Code Export	11
1.3	Minimal Distance	12
1.4	Distance	12
1.5	Brute Force Closest Pair Algorithm	13
1.5.1	Functional Correctness Proof	13
1.5.2	Time Complexity Proof	15
1.5.3	Code Export	15
1.6	Geometry	16
1.6.1	Band Filter	16
1.6.2	2D-Boxes and Points	16
1.6.3	Pigeonhole Argument	17
1.6.4	Delta Sparse Points within a Square	17

2	Closest Pair Algorithm	18
2.1	Functional Correctness Proof	18
2.1.1	Combine Step	18
2.1.2	Divide and Conquer Algorithm	21
2.2	Time Complexity Proof	23
2.2.1	Core Argument	23
2.2.2	Combine Step	24
2.2.3	Divide and Conquer Algorithm	25
2.3	Code Export	26
2.3.1	Combine Step	26
2.3.2	Divide and Conquer Algorithm	28
3	Closest Pair Algorithm 2	29
3.1	Functional Correctness Proof	30
3.1.1	Core Argument	30
3.1.2	Combine step	30
3.1.3	Divide and Conquer Algorithm	32
3.2	Time Complexity Proof	35
3.2.1	Combine Step	35
3.2.2	Divide and Conquer Algorithm	35
3.3	Code Export	36
3.3.1	Combine Step	36
3.3.2	Divide and Conquer Algorithm	37

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))
  <proof>
```

```
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
  <proof>
```

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 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 $t \in O[m \text{ going-to at-top within } A](f \text{ o } m)$
 <proof>

lemma *const-1-bigo-n-ln-n*:
 $(\lambda(n::nat). 1) \in O(\lambda n. n * \ln n)$
 <proof>

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>

lemma *set-take-drop*:
 $\text{set } xs = \text{set } (\text{take } n \text{ } xs) \cup \text{set } (\text{drop } n \text{ } xs)$
 <proof>

lemma *sorted-wrt-take-drop*:
 $\text{sorted-wrt } f \text{ } xs \implies \forall x \in \text{set } (\text{take } n \text{ } xs). \forall y \in \text{set } (\text{drop } n \text{ } xs). f \text{ } x \ y$
 <proof>

lemma *sorted-wrt-hd-less*:
assumes $\text{sorted-wrt } f \text{ } xs \bigwedge x. f \text{ } x \ x$
shows $\forall x \in \text{set } xs. f \text{ } (\text{hd } xs) \ x$
 <proof>

lemma *sorted-wrt-hd-less-take*:
assumes $\text{sorted-wrt } f \text{ } (x \# xs) \bigwedge x. f \text{ } x \ x$
shows $\forall y \in \text{set } (\text{take } n \text{ } (x \# xs)). f \text{ } x \ y$
 <proof>

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

lemma *sorted-wrt-hd-drop-less-drop*:
assumes $\text{sorted-wrt } f \text{ } xs \bigwedge x. f \text{ } x \ x$
shows $\forall x \in \text{set } (\text{drop } n \text{ } xs). f \text{ } (\text{hd } (\text{drop } n \text{ } xs)) \ x$
 <proof>

lemma *length-filter-P-impl-Q*:
 $(\bigwedge x. P \ x \implies Q \ x) \implies \text{length } (\text{filter } P \ xs) \leq \text{length } (\text{filter } Q \ xs)$
 <proof>

lemma *filter-Un*:
 $\text{set } xs = A \cup B \implies \text{set } (\text{filter } P \ xs) = \{ x \in A. P \ x \} \cup \{ x \in B. P \ x \}$

<proof>

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
<proof>

lemma *time-length-tm*:

time (*length-tm* xs) = *length* xs + 1
<proof>

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
<proof>

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

length xs = *length-it* xs
<proof>

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
<proof>

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

rev xs = *rev-it* xs

<proof>

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
<proof>
```

lemma time-take-tm:

```
time (take-tm n xs) = min n (length xs) + 1
<proof>
```

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
<proof>
```

lemma time-filter-tm:

```
time (filter-tm P xs) = length xs + 1
<proof>
```

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
 ⟨proof⟩

lemma *filter-conv-filter-it[code-unfold]*:
 filter P xs = *filter-it* P xs
 ⟨proof⟩

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
 ⟨proof⟩

lemma *split-at-take-drop-conv*:
split-at n xs = (take n xs, drop n xs)
 ⟨proof⟩

lemma *time-split-at-tm*:
 time (*split-at-tm* n xs) = min n (length xs) + 1

<proof>

```
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'
<proof>

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')
<proof>

lemma split-at-conv-split-at-it[code-unfold]:
split-at n xs = split-at-it n xs
<proof>

```
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 ($\lambda p_0 p_1. \text{fst } p_0 \leq \text{fst } p_1$) ps

definition sorted-snd :: point list \Rightarrow bool **where**
sorted-snd ps = sorted-wrt ($\lambda p_0 p_1. \text{snd } p_0 \leq \text{snd } p_1$) 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  $\leq$  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
<proof>

lemma *length-merge*:
length (merge f xs ys) = *length* xs + *length* ys
<proof>

lemma *set-merge*:
set (merge f xs ys) = *set* xs ∪ *set* ys
<proof>

lemma *distinct-merge*:
assumes *set* xs ∩ *set* ys = {} *distinct* xs *distinct* ys
shows *distinct* (merge f xs ys)
<proof>

lemma *sorted-merge*:
assumes $P = (\lambda x y. f x \leq f y)$
shows *sorted-wrt* P (merge f xs ys) \longleftrightarrow *sorted-wrt* P xs ∧ *sorted-wrt* P ys
<proof>

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;
 (xs_l, xs_r) <- *split-at-tm* (n div 2) xs;

```

    l <- mergesort-tm f xs_l;
    r <- mergesort-tm f xs_r;
    merge-tm f l r
  }
)
⟨proof⟩
termination mergesort-tm
⟨proof⟩

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
  ⟨proof⟩

lemma sorted-wrt-mergesort:
  sorted-wrt (λx y. f x ≤ f y) (mergesort f xs)
  ⟨proof⟩

lemma set-mergesort:
  set (mergesort f xs) = set xs
  ⟨proof⟩

lemma length-mergesort:
  length (mergesort f xs) = length xs
  ⟨proof⟩

lemma distinct-mergesort:
  distinct xs ⇒ distinct (mergesort f xs)
  ⟨proof⟩

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 ∀ p ∈ set (take n ps). fst p ≤ fst (hd (drop n ps))
  ⟨proof⟩

lemma sorted-fst-hd-drop-less-drop:
  assumes sorted-fst ps

```

shows $\forall p \in \text{set } (\text{drop } n \text{ ps}). \text{fst } (\text{hd } (\text{drop } n \text{ ps})) \leq \text{fst } p$
 ⟨proof⟩

1.2.2 Time Complexity Proof

lemma *time-merge-tm*:

$\text{time } (\text{merge-tm } f \text{ xs } \text{ ys}) \leq \text{length } \text{xs} + \text{length } \text{ys} + 1$
 ⟨proof⟩

function *mergesort-recurrence* :: *nat* \Rightarrow *real* **where**

mergesort-recurrence 0 = 1
 | *mergesort-recurrence* 1 = 1
 | $2 \leq n \implies \text{mergesort-recurrence } n = 4 + 3 * n + \text{mergesort-recurrence } (\text{nat } \lfloor \text{real } n / 2 \rfloor) +$
 mergesort-recurrence (nat $\lceil \text{real } n / 2 \rceil$)
 ⟨proof⟩

termination ⟨proof⟩

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

$0 \leq \text{mergesort-recurrence } n$
 ⟨proof⟩

lemma *time-mergesort-conv-mergesort-recurrence*:

$\text{time } (\text{mergesort-tm } f \text{ xs}) \leq \text{mergesort-recurrence } (\text{length } \text{xs})$
 ⟨proof⟩

theorem *mergesort-recurrence*:

$\text{mergesort-recurrence} \in \Theta(\lambda n. n * \ln n)$
 ⟨proof⟩

theorem *time-mergesort-tm-bigo*:

$(\lambda \text{xs}. \text{time } (\text{mergesort-tm } f \text{ xs})) \in O[\text{length going-to at-top}]((\lambda n. n * \ln n) \text{ o } \text{length})$
 ⟨proof⟩

1.2.3 Code Export

lemma *merge-xs-Nil[simp]*:

$\text{merge } f \text{ xs } [] = \text{xs}$
 ⟨proof⟩

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

merge-it' f acc [] [] = rev acc
 | *merge-it'* f acc (x#xs) [] = *merge-it'* f (x#acc) xs []
 | *merge-it'* f acc [] (y#ys) = *merge-it'* f (y#acc) ys []
 | *merge-it'* f acc (x#xs) (y#ys) = (
 if f x \leq f y then
 merge-it' f (x#acc) xs (y#ys)
 else
 merge-it' f (y#acc) (x#xs) ys
)

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

lemma *merge-conv-merge-it'*:
 rev acc @ merge f xs ys = *merge-it'* f acc xs ys
 ⟨proof⟩

lemma *merge-conv-merge-it*[code-unfold]:
 merge f xs ys = *merge-it* f xs ys
 ⟨proof⟩

1.3 Minimal Distance

definition *sparse* :: real ⇒ point set ⇒ bool **where**
sparse δ ps ↔ (∀ p₀ ∈ ps. ∀ p₁ ∈ ps. p₀ ≠ p₁ → δ ≤ dist p₀ p₁)

lemma *sparse-identity*:
assumes *sparse* δ (set ps) ∀ p ∈ set ps. δ ≤ dist p₀ p
shows *sparse* δ (set (p₀ # ps))
 ⟨proof⟩

lemma *sparse-update*:
assumes *sparse* δ (set ps)
assumes dist p₀ p₁ ≤ δ ∀ p ∈ set ps. dist p₀ p₁ ≤ dist p₀ p
shows *sparse* (dist p₀ p₁) (set (p₀ # ps))
 ⟨proof⟩

lemma *sparse-mono*:
sparse Δ P ⇒ δ ≤ Δ ⇒ *sparse* δ P
 ⟨proof⟩

1.4 Distance

lemma *dist-transform*:
fixes p :: point **and** δ :: real **and** l :: int
shows dist p (l, snd p) < δ ↔ l - δ < fst p ∧ fst p < l + δ
 ⟨proof⟩

fun *dist-code* :: point ⇒ point ⇒ int **where**
dist-code p₀ p₁ = (fst p₀ - fst p₁)² + (snd p₀ - snd p₁)²

lemma *dist-eq-sqrt-dist-code*:
fixes p₀ :: point
shows dist p₀ p₁ = sqrt (dist-code p₀ p₁)
 ⟨proof⟩

lemma *dist-eq-dist-code-lt*:
fixes p₀ :: point
shows dist p₀ p₁ < dist p₂ p₃ ↔ dist-code p₀ p₁ < dist-code p₂ p₃

<proof>

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

fixes $p_0 :: \text{point}$

shows $\text{dist } p_0 \ p_1 \leq \text{dist } p_2 \ p_3 \iff \text{dist-code } p_0 \ p_1 \leq \text{dist-code } p_2 \ p_3$

<proof>

lemma *dist-eq-dist-code-abs-lt:*

fixes $p_0 :: \text{point}$

shows $|c| < \text{dist } p_0 \ p_1 \iff c^2 < \text{dist-code } p_0 \ p_1$

<proof>

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

fixes $p_0 :: \text{point}$

shows $\text{dist } p_0 \ p_1 \leq |c| \iff \text{dist-code } p_0 \ p_1 \leq c^2$

<proof>

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>

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 $\text{dist } p \ p_0 < \text{dist } p \ p_1$ 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
<proof>

lemma *find-closest-bf-set*:
 $0 < \text{length } ps \implies \text{find-closest-bf } p \ ps \in \text{set } ps$
<proof>

lemma *find-closest-bf-dist*:
 $\forall q \in \text{set } ps. \text{dist } p \ (\text{find-closest-bf } p \ ps) \leq \text{dist } p \ q$
<proof>

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 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
<proof>

lemma *closest-pair-bf-c0*:
 $1 < \text{length } ps \implies (c_0, c_1) = \text{closest-pair-bf } ps \implies c_0 \in \text{set } ps$
<proof>

lemma *closest-pair-bf-c1*:

$1 < \text{length } ps \implies (c_0, c_1) = \text{closest-pair-bf } ps \implies c_1 \in \text{set } ps$
(proof)

lemma *closest-pair-bf-c0-ne-c1*:

$1 < \text{length } ps \implies \text{distinct } ps \implies (c_0, c_1) = \text{closest-pair-bf } ps \implies c_0 \neq c_1$
(proof)

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 $1 < \text{length } ps$ $(c_0, c_1) = \text{closest-pair-bf } ps$
shows *sparse* $(\text{dist } c_0 \ c_1)$ $(\text{set } ps)$
(proof)

1.5.2 Time Complexity Proof

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

$\text{time } (\text{find-closest-bf-tm } p \ ps) \leq \text{length } ps + 1$
(proof)

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

$\text{time } (\text{closest-pair-bf-tm } ps) \leq \text{length } ps * \text{length } ps + 1$
(proof)

1.5.3 Code Export

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

find-closest-bf-code $p \ [] = \text{undefined}$
| *find-closest-bf-code* $p \ [p_0] = (\text{dist-code } p \ p_0, p_0)$
| *find-closest-bf-code* $p \ (p_0 \ \# \ ps) = (
 \text{let } (\delta_1, p_1) = \text{find-closest-bf-code } p \ ps \ \text{in}$
 $\text{let } \delta_0 = \text{dist-code } p \ p_0 \ \text{in}$
 if $\delta_0 < \delta_1$ then
 (δ_0, p_0)
 else
 (δ_1, p_1)
)

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

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

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

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

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>

lemma closest-pair-bf-code-eq:

assumes $1 < \text{length } ps$
assumes $(c_0, c_1) = \text{closest-pair-bf-code } ps$ $(\delta', c_0', c_1') = \text{closest-pair-bf-code } ps$
shows $c_0 = c_0' \wedge c_1 = c_1'$
 <proof>

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>

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 $\text{sparse } \delta \ ps_L$ $\text{sparse } \delta \ ps_R$
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>

1.6.2 2D-Boxes and Points

lemma cbox-2D:

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

<proof>

lemma *mem-cbox-2D*:

fixes $x :: \text{real}$ **and** $y :: \text{real}$

shows $x_0 \leq x \wedge x \leq x_1 \wedge y_0 \leq y \wedge y \leq y_1 \longleftrightarrow (x, y) \in \text{cbox } (x_0, y_0) (x_1, y_1)$

<proof>

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 $\text{cbox } (x_0, y_0) (x_1, y_1) \cup \text{cbox } (x_0, y_1) (x_1, y_2) = \text{cbox } (x_0, y_0) (x_1, y_2)$

<proof>

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 $\text{cbox } (x_0, y_0) (x_1, y_1) \cup \text{cbox } (x_1, y_0) (x_2, y_1) = \text{cbox } (x_0, y_0) (x_2, y_1)$

<proof>

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>

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>

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>

lemma *card-Int-Un-le-Sum-card-Int*:

assumes *finite* S

shows $\text{card } (A \cap \bigcup S) \leq (\sum B \in S. \text{card } (A \cap B))$

<proof>

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>

1.6.4 Delta Sparse Points within a Square

lemma *max-points-square*:

```

assumes  $\forall p \in ps. p \in \text{cbox } (x, y) (x + \delta, y + \delta) \text{ sparse } \delta ps \ 0 \leq \delta$ 
shows  $\text{card } ps \leq 4$ 
<proof>

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 δ ps) = *find-closest* p δ ps
 ⟨*proof*⟩

lemma *find-closest-set*:
 $0 < \text{length } ps \implies \text{find-closest } p \delta ps \in \text{set } ps$
 ⟨*proof*⟩

lemma *find-closest-dist*:
assumes *sorted-snd* ($p \# ps$) $\exists q \in \text{set } ps. \text{dist } p q < \delta$
shows $\forall q \in \text{set } ps. \text{dist } p (\text{find-closest } p \delta ps) \leq \text{dist } p q$
 ⟨*proof*⟩

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 (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 {
 $p_1 <- \text{find-closest-tm } p_0 (\text{dist } c_0 c_1) ps$;
 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*) \Rightarrow *point list* \Rightarrow (*point* \times *point*) **where**
find-closest-pair (c_0, c_1) [] = (c_0, c_1)
 | *find-closest-pair* (c_0, c_1) [-] = (c_0, c_1)
 | *find-closest-pair* (c_0, c_1) ($p_0 \# ps$) = (
 let $p_1 = \text{find-closest } p_0 (\text{dist } c_0 c_1) ps$ *in*
 if $\text{dist } c_0 c_1 \leq \text{dist } p_0 p_1$ *then*
 find-closest-pair (c_0, c_1) ps
 else
 find-closest-pair (p_0, p_1) ps
)

lemma *find-closest-pair-eq-val-find-closest-pair-tm*:
val (*find-closest-pair-tm* (c_0, c_1) ps) = *find-closest-pair* (c_0, c_1) ps
 ⟨*proof*⟩

lemma *find-closest-pair-set*:
assumes (C_0, C_1) = *find-closest-pair* (c_0, c_1) ps
shows ($C_0 \in \text{set } ps \wedge C_1 \in \text{set } ps$) \vee ($C_0 = c_0 \wedge C_1 = c_1$)
 ⟨*proof*⟩

lemma *find-closest-pair-c0-ne-c1*:

$c_0 \neq c_1 \implies \text{distinct } ps \implies (C_0, C_1) = \text{find-closest-pair } (c_0, c_1) \text{ } ps \implies C_0 \neq C_1$
 <proof>

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

assumes $(C_0, C_1) = \text{find-closest-pair } (c_0, c_1) \text{ } ps$
shows $\text{dist } C_0 \ C_1 \leq \text{dist } c_0 \ c_1$
 <proof>

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* $(\text{dist } C_0 \ C_1)$ $(\text{set } ps)$
 <proof>

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

fun *combine-tm* :: $(\text{point} \times \text{point}) \Rightarrow (\text{point} \times \text{point}) \Rightarrow \text{int} \Rightarrow \text{point list} \Rightarrow (\text{point} \times \text{point}) \text{ tm}$ **where**

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

fun *combine* :: $(\text{point} \times \text{point}) \Rightarrow (\text{point} \times \text{point}) \Rightarrow \text{int} \Rightarrow \text{point list} \Rightarrow (\text{point} \times \text{point})$ **where**

combine $(p_{0L}, p_{1L}) (p_{0R}, p_{1R}) \ l \ ps =$ (
 let $(c_0, c_1) = \text{if } \text{dist } p_{0L} \ p_{1L} < \text{dist } p_{0R} \ p_{1R} \ \text{then } (p_{0L}, p_{1L}) \ \text{else } (p_{0R}, p_{1R})$ 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*:

$\text{val } (\text{combine-tm } (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) \ l \ ps) = \text{combine } (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) \ l \ ps$
 <proof>

lemma *combine-set*:

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

lemma *combine-c0-ne-c1*:

assumes $p_{0L} \neq p_{1L} \ p_{0R} \neq p_{1R} \ \text{distinct } ps$
assumes $(c_0, c_1) = \text{combine } (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) \ l \ ps$

shows $c_0 \neq c_1$
 ⟨proof⟩

lemma *combine-dist*:

assumes *sorted-snd* *ps set ps = ps_L ∪ ps_R*
assumes $\forall p \in ps_L. fst\ p \leq l \ \forall p \in ps_R. l \leq fst\ p$
assumes *sparse (dist p_{0L} p_{1L}) ps_L sparse (dist p_{0R} p_{1R}) ps_R*
assumes $(c_0, c_1) = combine\ (p_{0L}, p_{1L})\ (p_{0R}, p_{1R})\ l\ ps$
shows *sparse (dist c₀ c₁) (set ps)*
 ⟨proof⟩

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* ⇒ (*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 {
 (*xs_L*, *xs_R*) <- *split-at-tm (n div 2) xs*;
 (*ys_L*, *p_{0L}*, *p_{1L}*) <- *closest-pair-rec-tm xs_L*;
 (*ys_R*, *p_{0R}*, *p_{1R}*) <- *closest-pair-rec-tm xs_R*;
ys <- *merge-tm snd ys_L ys_R*;
 (*p₀*, *p₁*) <- *combine-tm (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) (fst (hd xs_R)) ys*;
 return (*ys*, *p₀*, *p₁*)
 }
 }
)
 ⟨proof⟩

termination *closest-pair-rec-tm*
 ⟨proof⟩

function *closest-pair-rec* :: *point list* ⇒ (*point list* * *point* * *point*) **where**
closest-pair-rec xs = (
 let *n = length xs* in
 if *n* ≤ 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)
  )
  ⟨proof⟩
termination closest-pair-rec
  ⟨proof⟩

```

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)
  )
  ⟨proof⟩

```

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⟩

```

```

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
  ⟨proof⟩

```

```

lemma closest-pair-rec-distinct:
  assumes distinct xs (ys, p) = closest-pair-rec xs
  shows distinct ys
  ⟨proof⟩

```

```

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
  ⟨proof⟩

```

```

lemma closest-pair-rec-dist:
  assumes 1 < length xs sorted-fst xs (ys, c0, c1) = closest-pair-rec xs
  shows sparse (dist c0 c1) (set xs)
  ⟨proof⟩

```

```

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 <- mergesort-tm fst ps;
    (-, p) <- 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 (-, 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
<proof>

lemma *closest-pair-simps*:
 $1 < \text{length } ps \implies \text{closest-pair } ps = (\text{let } (-, p) = \text{closest-pair-rec } (\text{mergesort fst } ps) \text{ in } p)$
<proof>

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

theorem *closest-pair-c0-c1*:
assumes $1 < \text{length } ps$ *distinct ps* (c_0, c_1) = *closest-pair ps*
shows $c_0 \in \text{set } ps$ $c_1 \in \text{set } ps$ $c_0 \neq c_1$
<proof>

theorem *closest-pair-dist*:
assumes $1 < \text{length } ps$ (c_0, c_1) = *closest-pair ps*
shows *sparse (dist c₀ c₁) (set ps)*
<proof>

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 *distinct (p # ps)* *sorted-snd (p # ps)* $0 \leq \delta$ *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 *sparse δ ps_L* *sparse δ ps_R*
shows $\text{length } (\text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) ps) \leq 7$
<proof>

2.2.2 Combine Step

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

lemma *t-find-closest-eq-time-find-closest-tm*:
t-find-closest *p* δ *ps* = *time* (*find-closest-tm* *p* δ *ps*)
⟨*proof*⟩

lemma *t-find-closest-mono*:
 $\delta' \leq \delta \implies \textit{t-find-closest } p \delta' ps \leq \textit{t-find-closest } p \delta ps$
⟨*proof*⟩

lemma *t-find-closest-cnt*:
t-find-closest *p* δ *ps* $\leq 1 + \text{length } (\text{filter } (\lambda q. \text{snd } q - \text{snd } p \leq \delta) ps)$
⟨*proof*⟩

corollary *t-find-closest-bound*:
fixes $\delta :: \textit{real}$ **and** *p* :: *point* **and** *ps* :: *point list* **and** *l* :: *int*
assumes *distinct* (*p* # *ps*) *sorted-snd* (*p* # *ps*) $0 \leq \delta$ *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* δ *ps*_L *sparse* δ *ps*_R
shows *t-find-closest* *p* δ *ps* $\leq \delta$
⟨*proof*⟩

fun *t-find-closest-pair* :: (*point* * *point*) \Rightarrow *point list* \Rightarrow *nat* **where**
t-find-closest-pair - [] = 1
| *t-find-closest-pair* - [-] = 1
| *t-find-closest-pair* (*c*₀, *c*₁) (*p*₀ # *ps*) = 1 + (
 let *p*₁ = *find-closest* *p*₀ (*dist* *c*₀ *c*₁) *ps* in
 t-find-closest *p*₀ (*dist* *c*₀ *c*₁) *ps* + (
 if *dist* *c*₀ *c*₁ \leq *dist* *p*₀ *p*₁ then
 t-find-closest-pair (*c*₀, *c*₁) *ps*
 else
 t-find-closest-pair (*p*₀, *p*₁) *ps*
)
)

lemma *t-find-closest-pair-eq-time-find-closest-pair-tm*:
t-find-closest-pair (*c*₀, *c*₁) *ps* = *time* (*find-closest-pair-tm* (*c*₀, *c*₁) *ps*)
⟨*proof*⟩

lemma *t-find-closest-pair-bound*:
assumes *distinct* *ps* *sorted-snd* *ps* $\delta = \text{dist } c_0 c_1$ *set* *ps* = *ps*_L \cup *ps*_R
assumes $\forall p \in \text{set } ps. l - \Delta < \text{fst } p \wedge \text{fst } p < l + \Delta$

assumes $\forall p \in ps_L. fst\ p \leq l \vee \forall p \in ps_R. l \leq fst\ p$
assumes $sparse\ \Delta\ ps_L\ sparse\ \Delta\ ps_R\ \delta \leq \Delta$
shows $t\text{-find-closest-pair}\ (c_0, c_1)\ ps \leq 9 * length\ ps + 1$
 <proof>

fun $t\text{-combine} :: (point * point) \Rightarrow (point * point) \Rightarrow int \Rightarrow point\ list \Rightarrow nat$ **where**
 $t\text{-combine}\ (p_{0L}, p_{1L})\ (p_{0R}, p_{1R})\ l\ ps = 1 +$
 $\quad let\ (c_0, c_1) = if\ dist\ p_{0L}\ p_{1L} < dist\ p_{0R}\ p_{1R}\ then\ (p_{0L}, p_{1L})\ else\ (p_{0R}, p_{1R})\ in$
 $\quad let\ ps' = filter\ (\lambda p. dist\ p\ (l, snd\ p) < dist\ c_0\ c_1)\ ps\ in$
 $\quad time\ (filter\text{-tm}\ (\lambda p. dist\ p\ (l, snd\ p) < dist\ c_0\ c_1)\ ps) + t\text{-find-closest-pair}\ (c_0,$
 $c_1)\ ps'$
 $\quad)$

lemma $t\text{-combine-eq-time-combine-tm}$:
 $t\text{-combine}\ (p_{0L}, p_{1L})\ (p_{0R}, p_{1R})\ l\ ps = time\ (combine\text{-tm}\ (p_{0L}, p_{1L})\ (p_{0R}, p_{1R})$
 $l\ ps)$
 <proof>

lemma $t\text{-combine-bound}$:
fixes $ps :: point\ list$
assumes $distinct\ ps\ sorted\text{-snd}\ ps\ set\ ps = ps_L \cup ps_R$
assumes $\forall p \in ps_L. fst\ p \leq l \vee \forall p \in ps_R. l \leq fst\ p$
assumes $sparse\ (dist\ p_{0L}\ p_{1L})\ ps_L\ sparse\ (dist\ p_{0R}\ p_{1R})\ ps_R$
shows $t\text{-combine}\ (p_{0L}, p_{1L})\ (p_{0R}, p_{1R})\ l\ ps \leq 10 * length\ ps + 3$
 <proof>

declare $t\text{-combine.simps}\ [simp\ del]$

2.2.3 Divide and Conquer Algorithm

lemma $time\text{-closest-pair-rec-tm-simps-1}$:
assumes $length\ xs \leq 3$
shows $time\ (closest\text{-pair-rec-tm}\ xs) = 1 + time\ (length\text{-tm}\ xs) + time\ (mergesort\text{-tm}$
 $snd\ xs) + time\ (closest\text{-pair-bf-tm}\ xs)$
 <proof>

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

function *closest-pair-recurrence* :: *nat* \Rightarrow *real* **where**
 $n \leq 3 \implies \text{closest-pair-recurrence } n = 3 + n + \text{mergesort-recurrence } n + n * n$
 $| 3 < n \implies \text{closest-pair-recurrence } n = 7 + 13 * n +$
 $\text{closest-pair-recurrence } (\text{nat } \lfloor \text{real } n / 2 \rfloor) + \text{closest-pair-recurrence } (\text{nat } \lceil \text{real } n / 2 \rceil)$
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

lemma *closest-pair-recurrence-nonneg[simp]*:
 $0 \leq \text{closest-pair-recurrence } n$
 $\langle \text{proof} \rangle$

lemma *time-closest-pair-rec-conv-closest-pair-recurrence*:
assumes *distinct ps sorted-fst ps*
shows $\text{time } (\text{closest-pair-rec-tm } ps) \leq \text{closest-pair-recurrence } (\text{length } ps)$
 $\langle \text{proof} \rangle$

theorem *closest-pair-recurrence*:
 $\text{closest-pair-recurrence} \in \Theta(\lambda n. n * \ln n)$
 $\langle \text{proof} \rangle$

theorem *time-closest-pair-rec-bigo*:
 $(\lambda xs. \text{time } (\text{closest-pair-rec-tm } xs)) \in O[\text{length going-to at-top within } \{ ps. \text{distinct } ps \wedge \text{sorted-fst } ps \}](\lambda n. n * \ln n) \circ \text{length}$
 $\langle \text{proof} \rangle$

definition *closest-pair-time* :: *nat* \Rightarrow *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 $\text{time } (\text{closest-pair-tm } ps) \leq \text{closest-pair-time } (\text{length } ps)$
 $\langle \text{proof} \rangle$

corollary *closest-pair-time*:
 $\text{closest-pair-time} \in O(\lambda n. n * \ln n)$
 $\langle \text{proof} \rangle$

corollary *time-closest-pair-bigo*:
 $(\lambda ps. \text{time } (\text{closest-pair-tm } ps)) \in O[\text{length going-to at-top within } \{ ps. \text{distinct } ps \}](\lambda n. n * \ln n) \circ \text{length}$
 $\langle \text{proof} \rangle$

2.3 Code Export

2.3.1 Combine Step

fun *find-closest-code* :: *point* \Rightarrow *int* \Rightarrow *point list* \Rightarrow (*int* * *point*) **where**
 $\text{find-closest-code } - - [] = \text{undefined}$

```

| find-closest-code p - [p0] = (dist-code p p0, p0)
| find-closest-code p δ (p0 # ps) = (
  let δ0 = dist-code p p0 in
  if δ ≤ (snd p0 - snd p)2 then
    (δ0, p0)
  else
    let (δ1, p1) = find-closest-code p (min δ δ0) ps in
    if δ0 ≤ δ1 then
      (δ0, p0)
    else
      (δ1, p1)
)

```

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>

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

lemma *find-closest-code-eq*:

assumes $0 < \text{length } ps \ \delta = \text{dist } c_0 \ c_1 \ \delta' = \text{dist-code } c_0 \ c_1 \ \text{sorted-snd } (p \# ps)$
assumes $c = \text{find-closest } p \ \delta \ ps \ (\delta_{c'}, c') = \text{find-closest-code } p \ \delta' \ ps$
shows $c = c'$
<proof>

fun *find-closest-pair-code* :: $(\text{int} * \text{point} * \text{point}) \Rightarrow \text{point list} \Rightarrow (\text{int} * \text{point} * \text{point})$ **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-code p0 δ 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 $\delta = \text{dist-code } c_0 \ c_1 \ (\Delta, C_0, C_1) = \text{find-closest-pair-code } (\delta, c_0, c_1) \ ps$
shows $\Delta = \text{dist-code } C_0 \ C_1$
<proof>

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 $C_0 = C_0' \wedge C_1 = C_1'$

<proof>

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

lemma *combine-code-eq*:

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

2.3.2 Divide and Conquer Algorithm

function *closest-pair-rec-code* :: point list ⇒ (point list * int * point * point) **where**

closest-pair-rec-code *xs* = (
 let *n* = 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* = fst (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* = merge *snd ys_L* *ys_R* in
 (*ys*, *combine-code* *p_L* *p_R* *l ys*)
)
<proof>

termination *closest-pair-rec-code*
<proof>

lemma *closest-pair-rec-code-simps*:

assumes $n = \text{length } xs \wedge (n \leq 3)$
shows *closest-pair-rec-code* *xs* = (
 let (*xs_L*, *xs_R*) = split-at ($n \text{ div } 2$) *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)
  )
  ⟨proof⟩

```

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

lemma *closest-pair-rec-code-dist-eq*:
assumes $1 < \text{length } xs$ $(ys, \delta, c_0, c_1) = \text{closest-pair-rec-code } xs$
shows $\delta = \text{dist-code } c_0 \ c_1$
 ⟨proof⟩

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 $ys = ys'$
 ⟨proof⟩

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 $c_0 = c_0' \wedge c_1 = c_1'$
 ⟨proof⟩

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⟩

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 ( $p_0 \# ps$ ) sorted-snd ( $p_0 \# ps$ )  $0 \leq \delta$  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  $\delta ps_R$ 
assumes  $p_1 \in \text{set } ps$  dist  $p_0 p_1 < \delta$ 
shows  $p_1 \in \text{set } (\text{take } 7 ps)$ 
<proof>

```

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>

```

fun *find-closest-pair-tm* :: (*point* * *point*) \Rightarrow *point list* \Rightarrow (*point* \times *point*) *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 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*) \Rightarrow *point list* \Rightarrow (*point* * *point*) **where**

```

find-closest-pair ( $c_0, c_1$ ) [] = ( $c_0, c_1$ )
| find-closest-pair ( $c_0, c_1$ ) [-] = ( $c_0, c_1$ )
| find-closest-pair ( $c_0, c_1$ ) ( $p_0 \# ps$ ) = (

```

```

    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
⟨proof⟩

```

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)
⟨proof⟩

```

lemma *find-closest-pair-c0-ne-c1*:

```

c0 ≠ c1 ⇒ distinct ps ⇒ (C0, C1) = find-closest-pair (c0, c1) ps ⇒ C0 ≠
C1
⟨proof⟩

```

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

```

assumes (C0, C1) = find-closest-pair (c0, c1) ps
shows dist C0 C1 ≤ dist c0 c1
⟨proof⟩

```

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)
⟨proof⟩

```

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
⟨proof⟩

```

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⟩

```

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⟩

```

lemma *combine-dist*:

```

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
assumes (c0, c1) = combine (p0L, p1L) (p0R, p1R) l ps
shows sparse (dist c0 c1) (set ps)
⟨proof⟩

```

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)
}
}
)
⟨proof⟩
termination closest-pair-rec-tm
⟨proof⟩

```

```

function closest-pair-rec :: point list ⇒ (point list * point * point) where
closest-pair-rec xs = (
  let n = length xs in
  if n ≤ 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)
)
⟨proof⟩
termination closest-pair-rec
⟨proof⟩

```

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)
)
⟨proof⟩

```

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⟩

```

```

lemma closest-pair-rec-set-length-sorted-snd:
  assumes  $(ys, p) = \text{closest-pair-rec } xs$ 
  shows  $\text{set } ys = \text{set } xs \wedge \text{length } ys = \text{length } xs \wedge \text{sorted-snd } ys$ 
   $\langle \text{proof} \rangle$ 

lemma closest-pair-rec-distinct:
  assumes  $\text{distinct } xs \ (ys, p) = \text{closest-pair-rec } xs$ 
  shows  $\text{distinct } ys$ 
   $\langle \text{proof} \rangle$ 

lemma closest-pair-rec-c0-c1:
  assumes  $1 < \text{length } xs \ \text{distinct } xs \ (ys, c_0, c_1) = \text{closest-pair-rec } xs$ 
  shows  $c_0 \in \text{set } xs \wedge c_1 \in \text{set } xs \wedge c_0 \neq c_1$ 
   $\langle \text{proof} \rangle$ 

lemma 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  $\text{sparse } (\text{dist } c_0 \ c_1) \ (\text{set } xs)$ 
   $\langle \text{proof} \rangle$ 

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

fun closest-pair ::  $\text{point list} \Rightarrow (\text{point} * \text{point})$  where
  closest-pair [] = undefined
| closest-pair [-] = undefined
| closest-pair ps = (let  $(-, c_0, c_1) = \text{closest-pair-rec } (\text{mergesort } \text{fst } ps)$  in  $(c_0, c_1)$ )

lemma closest-pair-eq-val-closest-pair-tm:
   $\text{val } (\text{closest-pair-tm } ps) = \text{closest-pair } ps$ 
   $\langle \text{proof} \rangle$ 

lemma closest-pair-simps:
   $1 < \text{length } ps \Longrightarrow \text{closest-pair } ps = (\text{let } (-, c_0, c_1) = \text{closest-pair-rec } (\text{mergesort } \text{fst } ps) \text{ in } (c_0, c_1))$ 
   $\langle \text{proof} \rangle$ 

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 $c_0 \in \text{set } ps \ c_1 \in \text{set } ps \ c_0 \neq c_1$
 ⟨proof⟩

theorem *closest-pair-dist*:

assumes $1 < \text{length } ps \ \text{distinct } ps \ (c_0, c_1) = \text{closest-pair } ps$
shows $\text{sparse } (\text{dist } c_0 \ c_1) \ (\text{set } ps)$
 ⟨proof⟩

3.2 Time Complexity Proof

3.2.1 Combine Step

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

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

lemma *time-combine-tm*:

fixes $ps :: \text{point list}$
shows $\text{time } (\text{combine-tm } (p_{0L}, p_{1L}) \ (p_{0R}, p_{1R}) \ l \ ps) \leq 3 + 18 * \text{length } ps$
 ⟨proof⟩

3.2.2 Divide and Conquer Algorithm

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

assumes $\text{length } xs \leq 3$
shows $\text{time } (\text{closest-pair-rec-tm } xs) = 1 + \text{time } (\text{length-tm } xs) + \text{time } (\text{mergesort-tm } \text{snd } xs) + \text{time } (\text{closest-pair-bf-tm } xs)$
 ⟨proof⟩

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

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

function *closest-pair-recurrence* $:: \text{nat} \Rightarrow \text{real}$ **where**

$n \leq 3 \implies \text{closest-pair-recurrence } n = 3 + n + \text{mergesort-recurrence } n + n * n$
 $| \ 3 < n \implies \text{closest-pair-recurrence } n = 7 + 21 * n + \text{closest-pair-recurrence } (\text{nat } \lfloor \text{real } n / 2 \rfloor) +$
 $\text{closest-pair-recurrence } (\text{nat } \lceil \text{real } n / 2 \rceil)$
 ⟨proof⟩

termination ⟨proof⟩

lemma *closest-pair-recurrence-nonneg[simp]*:

$0 \leq \text{closest-pair-recurrence } n$
{proof}

lemma *time-closest-pair-rec-conv-closest-pair-recurrence*:

$\text{time } (\text{closest-pair-rec-tm } ps) \leq \text{closest-pair-recurrence } (\text{length } ps)$
{proof}

theorem *closest-pair-recurrence*:

$\text{closest-pair-recurrence} \in \Theta(\lambda n. n * \ln n)$
{proof}

theorem *time-closest-pair-rec-bigo*:

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

definition *closest-pair-time* :: *nat* \Rightarrow *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*:

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

corollary *closest-pair-time*:

$\text{closest-pair-time} \in O(\lambda n. n * \ln n)$
{proof}

corollary *time-closest-pair-bigo*:

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

3.3 Code Export

3.3.1 Combine Step

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

$\text{find-closest-pair-code } (\delta, c_0, c_1) [] = (\delta, c_0, c_1)$
 $|\ \text{find-closest-pair-code } (\delta, c_0, c_1) [p] = (\delta, c_0, c_1)$
 $|\ \text{find-closest-pair-code } (\delta, c_0, c_1) (p_0 \# ps) = ($
 $\text{let } (\delta', p_1) = \text{find-closest-bf-code } p_0 (\text{take } 7 \text{ } ps) \text{ in}$
 $\text{if } \delta \leq \delta' \text{ then}$
 $\text{find-closest-pair-code } (\delta, c_0, c_1) ps$
 else
 $\text{find-closest-pair-code } (\delta', p_0, p_1) ps$
 $)$

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

<proof>

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 $C_0 = C_0' \wedge C_1 = C_1'$

<proof>

fun *combine-code* :: $(\text{int} * \text{point} * \text{point}) \Rightarrow (\text{int} * \text{point} * \text{point}) \Rightarrow \text{int} \Rightarrow \text{point}$

list $\Rightarrow (\text{int} * \text{point} * \text{point})$ **where**

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

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 $(\delta, c_0, c_1) = \text{combine-code } (\delta_L, p_{0L}, p_{1L}) \ (\delta_R, p_{0R}, p_{1R}) \ l \ ps$

shows $\delta = \text{dist-code } c_0 \ c_1$

<proof>

lemma *combine-code-eq*:

assumes $\delta_L' = \text{dist-code } p_{0L} \ p_{1L} \ \delta_R' = \text{dist-code } p_{0R} \ p_{1R}$

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

assumes $(\delta', c_0', c_1') = \text{combine-code } (\delta_L', p_{0L}, p_{1L}) \ (\delta_R', p_{0R}, p_{1R}) \ l \ ps$

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

<proof>

3.3.2 Divide and Conquer Algorithm

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

where

closest-pair-rec-code $xs = ($
 $\text{let } n = \text{length } xs \text{ in}$
 $\text{if } n \leq 3 \text{ then}$
 $(\text{mergesort snd } xs, \text{closest-pair-bf-code } xs)$
 else

$\text{let } (xs_L, xs_R) = \text{split-at } (n \text{ div } 2) \ xs \text{ in}$
 $\text{let } l = \text{fst } (\text{hd } xs_R) \text{ in}$

$\text{let } (ys_L, p_L) = \text{closest-pair-rec-code } xs_L \text{ in}$
 $\text{let } (ys_R, p_R) = \text{closest-pair-rec-code } xs_R \text{ in}$

```

    let ys = merge snd ys_L ys_R in
    (ys, combine-code p_L p_R l ys)
  )
  ⟨proof⟩
termination closest-pair-rec-code
  ⟨proof⟩

lemma closest-pair-rec-code-simps:
  assumes n = length xs  $\wedge$  (n ≤ 3)
  shows closest-pair-rec-code xs = (
    let (xs_L, xs_R) = split-at (n div 2) xs in
    let l = fst (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 = merge snd ys_L ys_R in
    (ys, combine-code p_L p_R l ys)
  )
  ⟨proof⟩

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, c_0, c_1) = closest-pair-rec-code xs
  shows δ = dist-code c_0 c_1
  ⟨proof⟩

lemma closest-pair-rec-ys-eq:
  assumes 1 < length xs
  assumes (ys, c_0, c_1) = closest-pair-rec xs
  assumes (ys', δ', c_0', c_1') = closest-pair-rec-code xs
  shows ys = ys'
  ⟨proof⟩

lemma closest-pair-rec-code-eq:
  assumes 1 < length xs
  assumes (ys, c_0, c_1) = closest-pair-rec xs
  assumes (ys', δ', c_0', c_1') = closest-pair-rec-code xs
  shows c_0 = c_0'  $\wedge$  c_1 = c_1'
  ⟨proof⟩

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 (-, -, c_0, c_1) = closest-pair-rec-code (mergesort fst ps)
  in (c_0, c_1))

```

```
lemma closest-pair-code-eq:  
  closest-pair ps = closest-pair-code ps  
  <proof>  
  
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.