Closest Pair of Points Algorithms

Martin Rau and Tobias Nipkow

December 14, 2021

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

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 $\bigwedge v.\ v = val\ n \Longrightarrow f\ v = g\ v\ m = n$
  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 $t \in O[m\ going\text{-}to\ at\text{-}top\ within\ A](f \circ m)$ 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 $t \in O[m\ going\text{-}to\ at\text{-}top\ within\ A](f \circ m)$.

lemma *bigo-measure-trans*:
  fixes $t :: 'a \Rightarrow real$ **and** $t' :: nat \Rightarrow real$ **and** $m :: 'a \Rightarrow nat$ **and** $f :: nat \Rightarrow real$
  assumes $\bigwedge x.\ x \in A \Longrightarrow t\ x \le (t'\ o\ m)\ x$

and $t' \in O(f)$
and $\bigwedge x.\ x \in A \Longrightarrow 0 \leq t\ x$
shows $t \in O[m\ \textit{going-to at-top within}\ A](f\ o\ m)$
$\langle \textit{proof} \rangle$

lemma *const-1-bigo-n-ln-n*:
$(\lambda(n::nat).\ 1) \in O(\lambda n.\ n * \ln n)$
$\langle \textit{proof} \rangle$

1.1.3 Miscellaneous Lemmas

lemma *set-take-drop-i-le-j*:
$i \leq j \Longrightarrow \textit{set xs} = \textit{set}\ (\textit{take j xs}) \cup \textit{set}\ (\textit{drop i xs})$
$\langle \textit{proof} \rangle$

lemma *set-take-drop*:
$\textit{set xs} = \textit{set}\ (\textit{take n xs}) \cup \textit{set}\ (\textit{drop n xs})$
$\langle \textit{proof} \rangle$

lemma *sorted-wrt-take-drop*:
$\textit{sorted-wrt f xs} \Longrightarrow \forall x \in \textit{set}\ (\textit{take n xs}).\ \forall y \in \textit{set}\ (\textit{drop n xs}).\ f\ x\ y$
$\langle \textit{proof} \rangle$

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

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

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

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

lemma *length-filter-P-impl-Q*:
$(\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \textit{length}\ (\textit{filter P xs}) \leq \textit{length}\ (\textit{filter Q xs})$
$\langle \textit{proof} \rangle$

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

⟨*proof*⟩

1.1.4 *length*

fun *length-tm* :: *'a list* ⇒ *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* ⇒ *'a list* ⇒ *nat* **where**
  length-it' acc [] = *acc*
| *length-it' acc* (*x*#*xs*) = *length-it'* (*acc*+*1*) *xs*

definition *length-it* :: *'a list* ⇒ *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* ⇒ *'a list* ⇒ *'a list* **where**
  rev-it' acc [] = *acc*
| *rev-it' acc* (*x*#*xs*) = *rev-it'* (*x*#*acc*) *xs*

definition *rev-it* :: *'a list* ⇒ *'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* ⇒ *'a list* ⇒ *'a list tm* **where**
  take-tm n [] =*1* *return* []
| *take-tm n* (*x* # *xs*) =*1*
    (*case n of*
      0 ⇒ *return* []
     | *Suc m* ⇒ *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* ⇒ *bool*) ⇒ *'a list* ⇒ *'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* ⇒ (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'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 ⇒ nat ⇒ 'a list ⇒ ('a list ∗ 'a list) where
  split-at-it' acc n [] = (rev acc, [])
| split-at-it' acc n (x#xs) = (
    case n of
      0 ⇒ (rev acc, x#xs)
    | Suc m ⇒ split-at-it' (x#acc) m xs
  )

definition split-at-it :: nat ⇒ 'a list ⇒ ('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 ⇒ bool where
  sorted-fst ps = sorted-wrt (λp0 p1. fst p0 ≤ fst p1) ps

definition sorted-snd :: point list ⇒ bool where
  sorted-snd ps = sorted-wrt (λp0 p1. snd p0 ≤ snd p1) ps

fun merge-tm :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list ⇒ '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
⟨*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* = (λ*x y. f x ≤ f y*)
shows *sorted-wrt P (merge f xs ys)* ⟷ *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 (λ*x 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 $\forall\, p \in set\ (drop\ n\ ps).\ fst\ (hd\ (drop\ n\ ps)) \leq fst\ p$
$\langle proof \rangle$

1.2.2 Time Complexity Proof

lemma *time-merge-tm*:
$time\ (merge\text{-}tm\ f\ xs\ ys) \leq length\ xs + length\ ys + 1$
$\langle proof \rangle$

function *mergesort-recurrence* :: $nat \Rightarrow real$ **where**
$mergesort\text{-}recurrence\ 0 = 1$
$|\ mergesort\text{-}recurrence\ 1 = 1$
$|\ 2 \leq n \Longrightarrow mergesort\text{-}recurrence\ n = 4 + 3 * n + mergesort\text{-}recurrence\ (nat\ \lfloor real\ n\ /\ 2 \rfloor) +$
$mergesort\text{-}recurrence\ (nat\ \lceil real\ n\ /\ 2 \rceil)$
$\langle proof \rangle$
termination $\langle proof \rangle$

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

lemma *time-mergesort-conv-mergesort-recurrence*:
$time\ (mergesort\text{-}tm\ f\ xs) \leq mergesort\text{-}recurrence\ (length\ xs)$
$\langle proof \rangle$

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

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

1.2.3 Code Export

lemma *merge-xs-Nil*[*simp*]:
$merge\ f\ xs\ [] = xs$
$\langle proof \rangle$

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

definition *merge-it* :: $('b \Rightarrow 'a{::}linorder) \Rightarrow 'b\ list \Rightarrow 'b\ list \Rightarrow 'b\ list$ **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* :: $real \Rightarrow point\ set \Rightarrow bool$ **where**
$sparse\ \delta\ ps \longleftrightarrow (\forall p_0 \in ps.\ \forall p_1 \in ps.\ p_0 \neq p_1 \longrightarrow \delta \leq dist\ p_0\ p_1)$

lemma *sparse-identity*:
assumes $sparse\ \delta\ (set\ ps)\ \forall p \in set\ ps.\ \delta \leq dist\ p_0\ p$
shows $sparse\ \delta\ (set\ (p_0\ \#\ ps))$
⟨*proof*⟩

lemma *sparse-update*:
assumes $sparse\ \delta\ (set\ ps)$
assumes $dist\ p_0\ p_1 \leq \delta\ \forall p \in set\ ps.\ dist\ p_0\ p_1 \leq dist\ p_0\ p$
shows $sparse\ (dist\ p_0\ p_1)\ (set\ (p_0\ \#\ ps))$
⟨*proof*⟩

lemma *sparse-mono*:
$sparse\ \Delta\ P \Longrightarrow \delta \leq \Delta \Longrightarrow sparse\ \delta\ P$
⟨*proof*⟩

1.4 Distance

lemma *dist-transform*:
fixes $p :: point$ **and** $\delta :: real$ **and** $l :: int$
shows $dist\ p\ (l,\ snd\ p) < \delta \longleftrightarrow l - \delta < fst\ p \wedge fst\ p < l + \delta$
⟨*proof*⟩

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

lemma *dist-eq-sqrt-dist-code*:
fixes $p_0 :: point$
shows $dist\ p_0\ p_1 = sqrt\ (dist\text{-}code\ p_0\ p_1)$
⟨*proof*⟩

lemma *dist-eq-dist-code-lt*:
fixes $p_0 :: point$
shows $dist\ p_0\ p_1 < dist\ p_2\ p_3 \longleftrightarrow dist\text{-}code\ p_0\ p_1 < dist\text{-}code\ p_2\ p_3$

⟨*proof*⟩

lemma *dist-eq-dist-code-le*:
  fixes p_0 :: *point*
  shows *dist* p_0 p_1 $\leq$ *dist* p_2 p_3 $\longleftrightarrow$ *dist-code* p_0 p_1 $\leq$ *dist-code* p_2 p_3
  ⟨*proof*⟩

lemma *dist-eq-dist-code-abs-lt*:
  fixes p_0 :: *point*
  shows $|c|$ < *dist* p_0 p_1 $\longleftrightarrow$ c^2 < *dist-code* p_0 p_1
  ⟨*proof*⟩

lemma *dist-eq-dist-code-abs-le*:
  fixes p_0 :: *point*
  shows *dist* p_0 p_1 $\leq$ $|c|$ $\longleftrightarrow$ *dist-code* p_0 p_1 $\leq$ c^2
  ⟨*proof*⟩

lemma *dist-fst-abs*:
  fixes *p* :: *point* **and** *l* :: *int*
  shows *dist p* (*l*, *snd p*) = |*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* :: *point* ⇒ *point list* ⇒ *point tm* **where**
  find-closest-bf-tm - [] =1 *return undefined*
| *find-closest-bf-tm* - [*p*] =1 *return p*
| *find-closest-bf-tm p* (p_0 # *ps*) =1 (
    do {
      p_1 <− *find-closest-bf-tm p ps*;
      if dist p p_0 < *dist p* p_1 *then*
        return p_0
      else
        return p_1
    }
)

fun *find-closest-bf* :: *point* ⇒ *point list* ⇒ *point* **where**
  find-closest-bf - [] = *undefined*
| *find-closest-bf* - [*p*] = *p*
| *find-closest-bf p* (p_0 # *ps*) = (
    let p_1 = *find-closest-bf p ps in*
    if dist p p_0 < *dist p* p_1 *then*
      p_0
    else
      p_1

)

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 < *length ps* ⟹ *find-closest-bf p ps* ∈ *set ps*
⟨*proof*⟩

lemma *find-closest-bf-dist*:
∀ *q* ∈ *set ps*. *dist p* (*find-closest-bf p ps*) ≤ *dist p q*
⟨*proof*⟩

fun *closest-pair-bf-tm* :: *point list* ⇒ (*point* × *point*) *tm* **where**
closest-pair-bf-tm [] =*1 return undefined*
| *closest-pair-bf-tm* [-] =*1 return undefined*
| *closest-pair-bf-tm* [p_0, p_1] =*1 return* (p_0, p_1)
| *closest-pair-bf-tm* (p_0 # *ps*) =*1* (
do {
(c_0::*point*, c_1::*point*) <− *closest-pair-bf-tm ps*;
p_1 <− *find-closest-bf-tm* p_0 *ps*;
if dist c_0 c_1 ≤ *dist* p_0 p_1 *then*
return (c_0, c_1)
else
return (p_0, p_1)
}
)

fun *closest-pair-bf* :: *point list* ⇒ (*point* ∗ *point*) **where**
closest-pair-bf [] = *undefined*
| *closest-pair-bf* [-] = *undefined*
| *closest-pair-bf* [p_0, p_1] = (p_0, p_1)
| *closest-pair-bf* (p_0 # *ps*) = (
let (c_0, c_1) = *closest-pair-bf ps in*
let p_1 = *find-closest-bf* p_0 *ps in*
if dist c_0 c_1 ≤ *dist* p_0 p_1 *then*
(c_0, c_1)
else
(p_0, p_1)
)

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 < *length ps* ⟹ (c_0, c_1) = *closest-pair-bf ps* ⟹ c_0 ∈ *set ps*
⟨*proof*⟩

lemma *closest-pair-bf-c1*:
$1 < length\ ps \Longrightarrow (c_0,\ c_1) = closest\text{-}pair\text{-}bf\ ps \Longrightarrow c_1 \in set\ ps$
⟨*proof*⟩

lemma *closest-pair-bf-c0-ne-c1*:
$1 < length\ ps \Longrightarrow distinct\ ps \Longrightarrow (c_0,\ c_1) = closest\text{-}pair\text{-}bf\ ps \Longrightarrow c_0 \neq c_1$
⟨*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 $1 < length\ ps\ (c_0,\ c_1) = closest\text{-}pair\text{-}bf\ ps$
shows $sparse\ (dist\ c_0\ c_1)\ (set\ ps)$
⟨*proof*⟩

1.5.2 Time Complexity Proof

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

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

1.5.3 Code Export

fun *find-closest-bf-code* :: *point* ⇒ *point list* ⇒ (*int* ∗ *point*) **where**
$find\text{-}closest\text{-}bf\text{-}code\ p\ [] = undefined$
$|\ find\text{-}closest\text{-}bf\text{-}code\ p\ [p_0] = (dist\text{-}code\ p\ p_0,\ p_0)$
$|\ find\text{-}closest\text{-}bf\text{-}code\ p\ (p_0\ \#\ ps) = ($
$let\ (\delta_1,\ p_1) = find\text{-}closest\text{-}bf\text{-}code\ p\ ps\ in$
$let\ \delta_0 = dist\text{-}code\ p\ p_0\ in$
$if\ \delta_0 < \delta_1\ then$
$(\delta_0,\ p_0)$
$else$
$(\delta_1,\ p_1)$
$)$

lemma *find-closest-bf-code-dist-eq*:
$0 < length\ ps \Longrightarrow (\delta,\ c) = find\text{-}closest\text{-}bf\text{-}code\ p\ ps \Longrightarrow \delta = dist\text{-}code\ p\ c$
⟨*proof*⟩

lemma *find-closest-bf-code-eq*:
$0 < length\ ps \Longrightarrow c = find\text{-}closest\text{-}bf\ p\ ps \Longrightarrow (\delta',\ c') = find\text{-}closest\text{-}bf\text{-}code\ p\ ps$
$\Longrightarrow c = c'$
⟨*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* [p_0] = *undefined*
| *closest-pair-bf-code* [p_0, p_1] = (*dist-code* p_0 p_1, p_0, p_1)
| *closest-pair-bf-code* (p_0 # *ps*) = (
    let (δ_c, c_0, c_1) = *closest-pair-bf-code ps in*
    let (δ_p, p_1) = *find-closest-bf-code* p_0 *ps in*
    if $\delta_c \leq \delta_p$ *then*
      (δ_c, c_0, c_1)
    else
      (δ_p, p_0, p_1)
)

lemma *closest-pair-bf-code-dist-eq*:
  1 < *length ps* $\Longrightarrow$ (δ, c_0, c_1) = *closest-pair-bf-code ps* $\Longrightarrow$ δ = *dist-code* c_0 c_1
⟨*proof*⟩

lemma *closest-pair-bf-code-eq*:
  assumes *1* < *length ps*
  assumes (c_0, c_1) = *closest-pair-bf ps* (δ', c_0', c_1') = *closest-pair-bf-code ps*
  shows $c_0 = c_0' \land c_1 = c_1'$
  ⟨*proof*⟩

1.6 Geometry

1.6.1 Band Filter

lemma *set-band-filter-aux*:
  fixes δ :: *real* **and** *ps* :: *point list*
  assumes $p_0 \in ps_L$ $p_1 \in ps_R$ $p_0 \neq p_1$ *dist* p_0 $p_1 < \delta$ *set ps* = $ps_L \cup ps_R$
  assumes $\forall p \in ps_L$. *fst p* $\leq l$ $\forall p \in ps_R$. $l \leq$ *fst p*
  assumes ps' = *filter* (λp. $l - \delta <$ *fst p* $\land$ *fst p* $< l + \delta$) *ps*
  shows $p_0 \in$ *set* $ps' \land p_1 \in$ *set* ps'
⟨*proof*⟩

lemma *set-band-filter*:
  fixes δ :: *real* **and** *ps* :: *point list*
  assumes $p_0 \in$ *set ps* $p_1 \in$ *set ps* $p_0 \neq p_1$ *dist* p_0 $p_1 < \delta$ *set ps* = $ps_L \cup ps_R$
  assumes *sparse* δ ps_L *sparse* δ ps_R
  assumes $\forall p \in ps_L$. *fst p* $\leq l$ $\forall p \in ps_R$. $l \leq$ *fst p*
  assumes ps' = *filter* (λp. $l - \delta <$ *fst p* $\land$ *fst p* $< l + \delta$) *ps*
  shows $p_0 \in$ *set* $ps' \land p_1 \in$ *set* ps'
⟨*proof*⟩

1.6.2 2D-Boxes and Points

lemma *cbox-2D*:
  fixes x_0 :: *real* **and** y_0 :: *real*
  shows *cbox* (x_0, y_0) (x_1, y_1) = { (x, y). $x_0 \leq x \land x \leq x_1 \land y_0 \leq y \land y \leq y_1$ }

$\langle proof \rangle$

lemma *mem-cbox-2D*:
fixes x :: *real* **and** y :: *real*
shows $x_0 \leq x \wedge x \leq x_1 \wedge y_0 \leq y \wedge y \leq y_1 \longleftrightarrow (x, y) \in cbox\ (x_0, y_0)\ (x_1, y_1)$
$\langle proof \rangle$

lemma *cbox-top-un*:
fixes x_0 :: *real* **and** y_0 :: *real*
assumes $y_0 \leq y_1$ $y_1 \leq y_2$
shows $cbox\ (x_0, y_0)\ (x_1, y_1) \cup cbox\ (x_0, y_1)\ (x_1, y_2) = cbox\ (x_0, y_0)\ (x_1, y_2)$
$\langle proof \rangle$

lemma *cbox-right-un*:
fixes x_0 :: *real* **and** y_0 :: *real*
assumes $x_0 \leq x_1$ $x_1 \leq x_2$
shows $cbox\ (x_0, y_0)\ (x_1, y_1) \cup cbox\ (x_1, y_0)\ (x_2, y_1) = cbox\ (x_0, y_0)\ (x_2, y_1)$
$\langle proof \rangle$

lemma *cbox-max-dist*:
assumes $p_0 = (x, y)$ $p_1 = (x + \delta, y + \delta)$
assumes $(x_0, y_0) \in cbox\ p_0\ p_1$ $(x_1, y_1) \in cbox\ p_0\ p_1$ $0 \leq \delta$
shows $dist\ (x_0, y_0)\ (x_1, y_1) \leq sqrt\ 2 * \delta$
$\langle proof \rangle$

1.6.3 Pigeonhole Argument

lemma *card-le-1-if-pairwise-eq*:
assumes $\forall x \in S.\ \forall y \in S.\ x = y$
shows $card\ S \leq 1$
$\langle proof \rangle$

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 $card\ (S \cap T) \leq 1$
$\langle proof \rangle$

lemma *card-Int-Un-le-Sum-card-Int*:
assumes *finite S*
shows $card\ (A \cap \bigcup S) \leq (\sum B \in S.\ card\ (A \cap B))$
$\langle proof \rangle$

lemma *pigeonhole*:
assumes $finite\ T$ $S \subseteq \bigcup T$ $card\ T < card\ S$
shows $\exists x \in S.\ \exists y \in S.\ \exists X \in T.\ x \neq y \wedge x \in X \wedge y \in X$
$\langle proof \rangle$

1.6.4 Delta Sparse Points within a Square

lemma *max-points-square*:

assumes $\forall\, p \in ps.\ p \in cbox\ (x,\ y)\ (x + \delta,\ y + \delta)$ *sparse* δ *ps* $0 \leq \delta$
shows *card ps* ≤ 4
$\langle proof \rangle$

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 ⇒ real ⇒ point list ⇒ point tm where
  find-closest-tm - - [] =1 return undefined
| find-closest-tm - - [p] =1 return p
| find-closest-tm p δ (p0 # ps) =1 (
    if δ ≤ snd p0 − snd p then
      return p0
    else
      do {
        p1 <− find-closest-tm p (min δ (dist p p0)) ps;
        if dist p p0 ≤ dist p p1 then
          return p0
        else
          return p1
      }
  )
```

```
fun find-closest :: point ⇒ real ⇒ point list ⇒ point where
  find-closest - - [] = undefined
| find-closest - - [p] = p
| find-closest p δ (p0 # ps) = (
    if δ ≤ snd p0 − snd p then
      p0
    else
      let p1 = find-closest p (min δ (dist p p0)) ps in
      if dist p p0 ≤ dist p p1 then
        p0
      else
        p1
  )
```

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 < *length ps* $\Longrightarrow$ *find-closest* p δ *ps* $\in$ *set ps*
  ⟨*proof*⟩

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

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

fun *find-closest-pair-tm* :: (*point* ∗ *point*) ⇒ *point list* ⇒ (*point* × *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 <− *find-closest-tm* p_0 (*dist* c_0 c_1) *ps*;
      if dist c_0 c_1 $\leq$ *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 (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 = *find-closest* p_0 (*dist* c_0 c_1) *ps in*
    if dist c_0 c_1 $\leq$ *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 ($C_0 \in$ *set ps* $\wedge$ $C_1 \in$ *set ps*) $\vee$ ($C_0 = c_0 \wedge C_1 = c_1$)
  ⟨*proof*⟩

lemma *find-closest-pair-c0-ne-c1*:
$c_0 \neq c_1 \Longrightarrow$ *distinct ps* $\Longrightarrow (C_0, C_1) =$ *find-closest-pair* (c_0, c_1) *ps* $\Longrightarrow C_0 \neq C_1$
$\langle proof \rangle$

lemma *find-closest-pair-dist-mono*:
assumes $(C_0, C_1) =$ *find-closest-pair* (c_0, c_1) *ps*
shows *dist* C_0 $C_1 \leq$ *dist* c_0 c_1
$\langle proof \rangle$

lemma *find-closest-pair-dist*:
assumes *sorted-snd ps* $(C_0, C_1) =$ *find-closest-pair* (c_0, c_1) *ps*
shows *sparse* (*dist* C_0 C_1) (*set ps*)
$\langle proof \rangle$

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

fun *combine* :: (*point* $\times$ *point*) $\Rightarrow$ (*point* $\times$ *point*) $\Rightarrow$ *int* $\Rightarrow$ *point list* $\Rightarrow$ (*point* $\times$ *point*) **where**
combine (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps* = (
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*
let $ps' =$ *filter* ($\lambda p.$ *dist p* (*l*, *snd p*) $<$ *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*
$\langle proof \rangle$

lemma *combine-set*:
assumes $(c_0, c_1) =$ *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
shows $(c_0 \in$ *set ps* $\wedge$ $c_1 \in$ *set ps*$) \vee (c_0 = p_{0L} \wedge c_1 = p_{1L}) \vee (c_0 = p_{0R} \wedge c_1 = p_{1R})$
$\langle proof \rangle$

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*⟩

lemma *combine-dist*:
assumes *sorted-snd ps set ps* $= ps_L \cup ps_R$
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* $(dist\ c_0\ c_1)\ (set\ ps)$
⟨*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 $n \leq 3$ *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* $ys_L\ ys_R$;
        $(p_0,\ p_1)$ <− *combine-tm* $(p_{0L},\ p_{1L})\ (p_{0R},\ p_{1R})$ (*fst* (*hd* xs_R)) *ys*;
        return $(ys,\ p_0,\ p_1)$
      }
  }
)
⟨*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 $n \leq 3$ *then*
    (*mergesort snd xs*, *closest-pair-bf xs*)
  else

$let\ (xs_L,\ xs_R) = split\text{-}at\ (n\ div\ 2)\ xs\ in$
$let\ (ys_L,\ p_{0L},\ p_{1L}) = closest\text{-}pair\text{-}rec\ xs_L\ in$
$let\ (ys_R,\ p_{0R},\ p_{1R}) = closest\text{-}pair\text{-}rec\ xs_R\ in$
$let\ ys = merge\ snd\ ys_L\ ys_R\ in$
$(ys,\ combine\ (p_{0L},\ p_{1L})\ (p_{0R},\ p_{1R})\ (fst\ (hd\ xs_R))\ ys)$
)
⟨*proof*⟩
termination *closest-pair-rec*
⟨*proof*⟩

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

lemma *closest-pair-rec-simps*:
assumes $n = length\ xs$ $\neg\ (n \leq 3)$
shows $closest\text{-}pair\text{-}rec\ xs = ($
$let\ (xs_L,\ xs_R) = split\text{-}at\ (n\ div\ 2)\ xs\ in$
$let\ (ys_L,\ p_{0L},\ p_{1L}) = closest\text{-}pair\text{-}rec\ xs_L\ in$
$let\ (ys_R,\ p_{0R},\ p_{1R}) = closest\text{-}pair\text{-}rec\ xs_R\ in$
$let\ ys = merge\ snd\ ys_L\ ys_R\ in$
$(ys,\ combine\ (p_{0L},\ p_{1L})\ (p_{0R},\ p_{1R})\ (fst\ (hd\ xs_R))\ ys)$
)
⟨*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 $set\ ys = set\ xs \land length\ ys = length\ xs \land sorted\text{-}snd\ ys$
⟨*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* $(ys,\ c_0,\ c_1) = closest\text{-}pair\text{-}rec\ xs$
shows $c_0 \in set\ xs \land c_1 \in set\ xs \land c_0 \neq c_1$
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *closest-pair-simps*:
1 < *length ps* ⟹ *closest-pair ps* = (*let* (-, *p*) = *closest-pair-rec* (*mergesort fst ps*) *in p*)
⟨*proof*⟩

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

theorem *closest-pair-c0-c1*:
assumes *1* < *length ps distinct ps* (c_0, c_1) = *closest-pair ps*
shows c_0 ∈ *set ps* c_1 ∈ *set ps* $c_0 \neq c_1$
⟨*proof*⟩

theorem *closest-pair-dist*:
assumes *1* < *length ps* (c_0, c_1) = *closest-pair ps*
shows *sparse* (*dist* c_0 c_1) (*set ps*)
⟨*proof*⟩

2.2 Time Complexity Proof

2.2.1 Core Argument

lemma *core-argument*:
fixes δ :: *real* **and** *p* :: *point* **and** *ps* :: *point list*
assumes *distinct* (*p* # *ps*) *sorted-snd* (*p* # *ps*) $0 \leq \delta$ *set* (*p* # *ps*) = $ps_L \cup ps_R$
assumes ∀ *q* ∈ *set* (*p* # *ps*). $l - \delta <$ *fst q* ∧ *fst q* $< l + \delta$
assumes ∀ *q* ∈ ps_L. *fst q* ≤ *l* ∀ *q* ∈ ps_R. *l* ≤ *fst q*
assumes *sparse* δ ps_L *sparse* δ ps_R
shows *length* (*filter* (λ*q*. *snd q* − *snd p* ≤ δ) *ps*) ≤ *7*
⟨*proof*⟩

2.2.2 Combine Step

fun *t-find-closest* :: *point* ⇒ *real* ⇒ *point list* ⇒ *nat* **where**
  t-find-closest - - [] = *1*
| *t-find-closest* - - [-] = *1*
| *t-find-closest* p δ (p_0 # *ps*) = *1* + (
    if $\delta \leq$ *snd* p_0 − *snd* p *then 0*
    else t-find-closest p (*min* δ (*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*:
  $\delta' \leq \delta \Longrightarrow$ *t-find-closest* p δ' *ps* ≤ *t-find-closest* p δ *ps*
  ⟨*proof*⟩

lemma *t-find-closest-cnt*:
  t-find-closest p δ *ps* ≤ *1* + *length* (*filter* (λq. *snd* q − *snd* $p \leq \delta$) *ps*)
⟨*proof*⟩

corollary *t-find-closest-bound*:
  fixes δ :: *real* **and** p :: *point* **and** *ps* :: *point list* **and** l :: *int*
  assumes *distinct* (p # *ps*) *sorted-snd* (p # *ps*) *0* ≤ δ *set* (p # *ps*) = $ps_L \cup ps_R$
  assumes $\forall p' \in$ *set* (p # *ps*). $l - \delta <$ *fst* $p' \wedge$ *fst* $p' < l + \delta$
  assumes $\forall p \in ps_L$. *fst* $p \leq l$ $\forall p \in ps_R$. $l \leq$ *fst* p
  assumes *sparse* δ ps_L *sparse* δ ps_R
  shows *t-find-closest* p δ *ps* ≤ *8*
  ⟨*proof*⟩

fun *t-find-closest-pair* :: (*point* ∗ *point*) ⇒ *point list* ⇒ *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 ≤ *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* δ = *dist* c_0 c_1 *set ps* = $ps_L \cup ps_R$
  assumes $\forall p \in$ *set ps*. $l - \Delta <$ *fst* $p \wedge$ *fst* $p < l + \Delta$

assumes $\forall\, p \in ps_L.\ fst\ p \leq l$ $\forall\, p \in ps_R.\ l \leq fst\ p$
assumes *sparse* Δ ps_L *sparse* Δ ps_R $\delta \leq \Delta$
shows *t-find-closest-pair* $(c_0,\ c_1)$ *ps* $\leq$ *9* $*$ *length ps* + *1*
⟨*proof*⟩

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

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

lemma *t-combine-bound*:
fixes *ps* :: *point list*
assumes *distinct ps sorted-snd ps set ps* = $ps_L \cup ps_R$
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
shows *t-combine* $(p_{0L},\ p_{1L})$ $(p_{0R},\ p_{1R})$ *l ps* $\leq$ *10* $*$ *length ps* + *3*
⟨*proof*⟩

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

2.2.3 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*)
⟨*proof*⟩

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

function *closest-pair-recurrence* :: *nat* ⇒ *real* **where**
$n \leq 3 \Longrightarrow$ *closest-pair-recurrence* $n = 3 + n +$ *mergesort-recurrence* $n + n * n$
$|\ 3 < n \Longrightarrow$ *closest-pair-recurrence* $n = 7 + 13 * n +$
closest-pair-recurrence (*nat* $\lfloor$*real* n / $2\rfloor$) + *closest-pair-recurrence* (*nat* $\lceil$*real* n / $2\rceil$)
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *closest-pair-recurrence-nonneg*[*simp*]:
$0 \leq$ *closest-pair-recurrence* n
⟨*proof*⟩

lemma *time-closest-pair-rec-conv-closest-pair-recurrence*:
assumes *distinct ps sorted-fst ps*
shows *time* (*closest-pair-rec-tm ps*) ≤ *closest-pair-recurrence* (*length ps*)
⟨*proof*⟩

theorem *closest-pair-recurrence*:
closest-pair-recurrence ∈ $\Theta(\lambda n.\ n * \ln n)$
⟨*proof*⟩

theorem *time-closest-pair-rec-bigo*:
(λ*xs. time* (*closest-pair-rec-tm xs*)) ∈ *O*[*length going-to at-top within* { *ps. distinct ps* ∧ *sorted-fst ps* }](($\lambda n.\ n * \ln n$) *o length*)
⟨*proof*⟩

definition *closest-pair-time* :: *nat* ⇒ *real* **where**
closest-pair-time $n = 1 +$ *mergesort-recurrence* $n +$ *closest-pair-recurrence* n

lemma *time-closest-pair-conv-closest-pair-recurrence*:
assumes *distinct ps*
shows *time* (*closest-pair-tm ps*) ≤ *closest-pair-time* (*length ps*)
⟨*proof*⟩

corollary *closest-pair-time*:
closest-pair-time ∈ $O(\lambda n.\ n * \ln n)$
⟨*proof*⟩

corollary *time-closest-pair-bigo*:
(λ*ps. time* (*closest-pair-tm ps*)) ∈ *O*[*length going-to at-top within* { *ps. distinct ps* }](($\lambda n.\ n * \ln n$) *o length*)
⟨*proof*⟩

2.3 Code Export

2.3.1 Combine Step

fun *find-closest-code* :: *point* ⇒ *int* ⇒ *point list* ⇒ (*int* ∗ *point*) **where**
find-closest-code - - [] = *undefined*

| *find-closest-code* p - $[p_0]$ = (*dist-code* p p_0, p_0)
| *find-closest-code* p δ (p_0 # *ps*) = (
    let δ_0 = *dist-code* p p_0 *in*
    if $\delta \leq (snd\ p_0 - snd\ p)^2$ *then*
      (δ_0, p_0)
    else
      let (δ_1, p_1) = *find-closest-code* p (*min* δ δ_0) *ps in*
      if $\delta_0 \leq \delta_1$ *then*
        (δ_0, p_0)
      else
        (δ_1, p_1)
)

lemma *find-closest-code-dist-eq*:
  $0 <$ *length ps* $\Longrightarrow$ (δ_c, c) = *find-closest-code* p δ *ps* $\Longrightarrow$ δ_c = *dist-code* p c
⟨*proof*⟩

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

lemma *find-closest-code-eq*:
  assumes $0 <$ *length ps* δ = *dist* c_0 c_1 δ' = *dist-code* c_0 c_1 *sorted-snd* (p # *ps*)
  assumes c = *find-closest* p δ *ps* (${\delta_c}'$, c') = *find-closest-code* p δ' *ps*
  shows $c = c'$
  ⟨*proof*⟩

fun *find-closest-pair-code* :: (*int* ∗ *point* ∗ *point*) ⇒ *point list* ⇒ (*int* ∗ *point* ∗ *point*) **where**
  find-closest-pair-code (δ, c_0, c_1) [] = (δ, c_0, c_1)
| *find-closest-pair-code* (δ, c_0, c_1) [p] = (δ, c_0, c_1)
| *find-closest-pair-code* (δ, c_0, c_1) (p_0 # *ps*) = (
    let (δ', p_1) = *find-closest-code* p_0 δ *ps in*
    if $\delta \leq \delta'$ *then*
      find-closest-pair-code (δ, c_0, c_1) *ps*
    else
      find-closest-pair-code (δ', p_0, p_1) *ps*
)

lemma *find-closest-pair-code-dist-eq*:
  assumes δ = *dist-code* c_0 c_1 (Δ, C_0, C_1) = *find-closest-pair-code* (δ, c_0, c_1) *ps*
  shows Δ = *dist-code* C_0 C_1
  ⟨*proof*⟩

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

lemma *find-closest-pair-code-eq*:
  assumes δ = *dist* c_0 c_1 δ' = *dist-code* c_0 c_1 *sorted-snd ps*
  assumes (C_0, C_1) = *find-closest-pair* (c_0, c_1) *ps*
  assumes (Δ', ${C_0}'$, ${C_1}'$) = *find-closest-pair-code* (δ', 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 $(\delta_L, p_{0L}, p_{1L})$ $(\delta_R, p_{0R}, p_{1R})$ *l ps* = (
let (δ, 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 (δ, c_0, c_1) ps'
)

lemma *combine-code-dist-eq*:
assumes δ_L = *dist-code* p_{0L} p_{1L} δ_R = *dist-code* p_{0R} p_{1R}
assumes (δ, c_0, c_1) = *combine-code* $(\delta_L, p_{0L}, p_{1L})$ $(\delta_R, p_{0R}, p_{1R})$ *l ps*
shows δ = *dist-code* c_0 c_1
⟨*proof*⟩

lemma *combine-code-eq*:
assumes δ_L' = *dist-code* p_{0L} p_{1L} δ_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 (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 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 = *length xs* $\neg$ $(n \leq 3)$
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* ⇒ (*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

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 $distinct\ (p_0\ \#\ ps)\ sorted\text{-}snd\ (p_0\ \#\ ps)\ 0 \leq \delta\ set\ (p_0\ \#\ ps) = ps_L \cup ps_R$
assumes $\forall p \in set\ (p_0\ \#\ ps).\ l - \delta \leq fst\ p \wedge fst\ p \leq l + \delta$
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*⟩

3.1.2 Combine step

lemma *find-closest-bf-dist-take-7*:
assumes $\exists p_1 \in set\ ps.\ dist\ p_0\ p_1 < \delta$
assumes $distinct\ (p_0\ \#\ ps)\ sorted\text{-}snd\ (p_0\ \#\ ps)\ 0 < length\ ps\ 0 \leq \delta\ set\ (p_0\ \#\ ps) = ps_L \cup ps_R$
assumes $\forall p \in set\ (p_0\ \#\ ps).\ l - \delta \leq fst\ p \wedge fst\ p \leq l + \delta$
assumes $\forall p \in ps_L.\ fst\ p \leq l\ \forall p \in ps_R.\ l \leq fst\ p$
assumes $sparse\ \delta\ ps_L\ sparse\ \delta\ ps_R$
shows $\forall p_1 \in set\ ps.\ dist\ p_0\ (find\text{-}closest\text{-}bf\ p_0\ (take\ 7\ ps)) \leq dist\ p_0\ p_1$
⟨*proof*⟩

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

fun *find-closest-pair* :: $(point * point) \Rightarrow point\ list \Rightarrow (point * point)$ **where**
  $find\text{-}closest\text{-}pair\ (c_0,\ c_1)\ [] = (c_0,\ c_1)$
$|\ find\text{-}closest\text{-}pair\ (c_0,\ c_1)\ [\text{-}] = (c_0,\ c_1)$
$|\ find\text{-}closest\text{-}pair\ (c_0,\ c_1)\ (p_0\ \#\ ps) =$ (

  let p_1 = *find-closest-bf* p_0 (*take 7 ps*) *in*
  if dist c_0 c_1 $\leq$ *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 ($C_0 \in$ *set ps* $\wedge$ $C_1 \in$ *set ps*) $\vee$ ($C_0 = c_0 \wedge C_1 = c_1$)
 ⟨*proof*⟩

lemma *find-closest-pair-c0-ne-c1*:
 $c_0 \neq c_1 \Longrightarrow$ *distinct ps* $\Longrightarrow$ (C_0, C_1) = *find-closest-pair* (c_0, c_1) *ps* $\Longrightarrow C_0 \neq C_1$
⟨*proof*⟩

lemma *find-closest-pair-dist-mono*:
 assumes (C_0, C_1) = *find-closest-pair* (c_0, c_1) *ps*
 shows *dist* C_0 C_1 $\leq$ *dist* c_0 c_1
 ⟨*proof*⟩

lemma *find-closest-pair-dist*:
 assumes *sorted-snd ps distinct ps set ps* = $ps_L \cup ps_R$ *0* $\leq \delta$
 assumes $\forall p \in$ *set ps*. $l - \delta \leq$ *fst p* $\wedge$ *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* δ ps_L *sparse* δ ps_R *dist* c_0 c_1 $\leq \delta$
 assumes (C_0, C_1) = *find-closest-pair* (c_0, c_1) *ps*
 shows *sparse* (*dist* C_0 C_1) (*set ps*)
 ⟨*proof*⟩

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 dist* p_{0L} p_{1L} < *dist* p_{0R} p_{1R} *then* (p_{0L}, p_{1L}) *else* (p_{0R}, p_{1R}) *in*
  do {
   ps' <− *filter-tm* (λp. *dist p* (*l*, *snd p*) < *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 dist* p_{0L} p_{1L} < *dist* p_{0R} p_{1R} *then* (p_{0L}, p_{1L}) *else* (p_{0R}, p_{1R}) *in*
  let ps' = *filter* (λp. *dist p* (*l*, *snd p*) < *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*
⟨*proof*⟩

lemma *combine-set*:
assumes (c_0, c_1) = *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
shows ($c_0 \in$ *set ps* $\wedge$ $c_1 \in$ *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 $p_{0L} \neq p_{1L}$ $p_{0R} \neq p_{1R}$ *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*⟩

lemma *combine-dist*:
assumes *distinct ps sorted-snd ps set ps* = $ps_L \cup ps_R$
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* c_0 c_1) (*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 {
        (xs_L, xs_R) <- split-at-tm (n div 2) xs;
        (ys_L, p0_L, p1_L) <- closest-pair-rec-tm xs_L;
        (ys_R, p0_R, p1_R) <- closest-pair-rec-tm xs_R;
        ys <- merge-tm snd ys_L ys_R;
        (p0, p1) <- combine-tm (p0_L, p1_L) (p0_R, p1_R) (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 (xs_L, xs_R) = split-at (n div 2) xs in
      let (ys_L, p0_L, p1_L) = closest-pair-rec xs_L in
      let (ys_R, p0_R, p1_R) = closest-pair-rec xs_R in
      let ys = merge snd ys_L ys_R in
      (ys, combine (p0_L, p1_L) (p0_R, p1_R) (fst (hd xs_R)) 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 (xs_L, xs_R) = split-at (n div 2) xs in
    let (ys_L, p0_L, p1_L) = closest-pair-rec xs_L in
    let (ys_R, p0_R, p1_R) = closest-pair-rec xs_R in
    let ys = merge snd ys_L ys_R in
    (ys, combine (p0_L, p1_L) (p0_R, p1_R) (fst (hd xs_R)) 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 $(ys,\ p) = closest\text{-}pair\text{-}rec\ xs$
shows $set\ ys = set\ xs \wedge length\ ys = length\ xs \wedge sorted\text{-}snd\ ys$
⟨*proof*⟩

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

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

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

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*
| $closest\text{-}pair\ ps = (let\ (\text{-},\ c_0,\ c_1) = closest\text{-}pair\text{-}rec\ (mergesort\ fst\ ps)\ in\ (c_0,\ c_1))$

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

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

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

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

shows $c_0 \in$ *set ps* $c_1 \in$ *set ps* $c_0 \neq c_1$
⟨*proof*⟩

theorem *closest-pair-dist*:
assumes *1* < *length ps distinct ps* (c_0, c_1) = *closest-pair ps*
shows *sparse* (*dist* c_0 c_1) (*set ps*)
⟨*proof*⟩

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*) ≤ *17* ∗ *length ps* + *1*
⟨*proof*⟩

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

3.2.2 Divide and Conquer Algorithm

lemma *time-closest-pair-rec-tm-simps-1*:
assumes *length xs* ≤ *3*
shows *time* (*closest-pair-rec-tm xs*) = *1* + *time* (*length-tm xs*) + *time* (*mergesort-tm snd xs*) + *time* (*closest-pair-bf-tm xs*)
⟨*proof*⟩

lemma *time-closest-pair-rec-tm-simps-2*:
assumes ¬ (*length xs* ≤ *3*)
shows *time* (*closest-pair-rec-tm xs*) = *1* + (
let (xs_L, xs_R) = *val* (*split-at-tm* (*length xs div 2*) *xs*) *in*
let (ys_L, p_L) = *val* (*closest-pair-rec-tm* xs_L) *in*
let (ys_R, p_R) = *val* (*closest-pair-rec-tm* xs_R) *in*
let ys = *val* (*merge-tm* (λ*p. snd p*) ys_L ys_R) *in*
time (*length-tm xs*) + *time* (*split-at-tm* (*length xs div 2*) *xs*) + *time* (*closest-pair-rec-tm* xs_L) +
time (*closest-pair-rec-tm* xs_R) + *time* (*merge-tm* (λ*p. snd p*) ys_L ys_R) + *time* (*combine-tm* p_L p_R (*fst* (*hd* xs_R)) *ys*)
)
⟨*proof*⟩

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*⌉)
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *closest-pair-recurrence-nonneg*[*simp*]:
$0 \le$ *closest-pair-recurrence n*
⟨*proof*⟩

lemma *time-closest-pair-rec-conv-closest-pair-recurrence*:
time (*closest-pair-rec-tm ps*) $\le$ *closest-pair-recurrence* (*length ps*)
⟨*proof*⟩

theorem *closest-pair-recurrence*:
closest-pair-recurrence $\in \Theta(\lambda n.\ n * ln\ n)$
⟨*proof*⟩

theorem *time-closest-pair-rec-bigo*:
(λ*xs. time* (*closest-pair-rec-tm xs*)) $\in$ *O*[*length going-to at-top*](($\lambda n.\ n * ln\ n$) *o length*)
⟨*proof*⟩

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*) $\le$ *closest-pair-time* (*length ps*)
⟨*proof*⟩

corollary *closest-pair-time*:
closest-pair-time $\in O(\lambda n.\ n * ln\ n)$
⟨*proof*⟩

corollary *time-closest-pair-bigo*:
(λ*ps. time* (*closest-pair-tm ps*)) $\in$ *O*[*length going-to at-top*](($\lambda n.\ n * ln\ n$) *o 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**
find-closest-pair-code (δ, c_0, c_1) [] = (δ, c_0, c_1)
| *find-closest-pair-code* (δ, c_0, c_1) [*p*] = (δ, c_0, c_1)
| *find-closest-pair-code* (δ, c_0, c_1) (p_0 # *ps*) = (
let (δ', p_1) = *find-closest-bf-code* p_0 (*take 7 ps*) *in*
if $\delta \le \delta'$ *then*
find-closest-pair-code (δ, c_0, c_1) *ps*
else
find-closest-pair-code (δ', p_0, p_1) *ps*
)

lemma *find-closest-pair-code-dist-eq*:
  assumes $\delta = \textit{dist-code}\ c_0\ c_1\ (\Delta,\ C_0,\ C_1) = \textit{find-closest-pair-code}\ (\delta,\ c_0,\ c_1)\ ps$
  shows $\Delta = \textit{dist-code}\ C_0\ C_1$
  ⟨*proof*⟩

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

lemma *find-closest-pair-code-eq*:
  assumes $\delta = \textit{dist}\ c_0\ c_1\ \delta' = \textit{dist-code}\ c_0\ c_1$
  assumes $(C_0,\ C_1) = \textit{find-closest-pair}\ (c_0,\ c_1)\ ps$
  assumes $(\Delta',\ C_0',\ C_1') = \textit{find-closest-pair-code}\ (\delta',\ c_0,\ c_1)\ ps$
  shows $C_0 = C_0' \wedge C_1 = C_1'$
  ⟨*proof*⟩

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

lemma *combine-code-dist-eq*:
  assumes $\delta_L = \textit{dist-code}\ p_{0L}\ p_{1L}\ \delta_R = \textit{dist-code}\ p_{0R}\ p_{1R}$
  assumes $(\delta,\ c_0,\ c_1) = \textit{combine-code}\ (\delta_L,\ p_{0L},\ p_{1L})\ (\delta_R,\ p_{0R},\ p_{1R})\ l\ ps$
  shows $\delta = \textit{dist-code}\ c_0\ c_1$
  ⟨*proof*⟩

lemma *combine-code-eq*:
  assumes $\delta_L' = \textit{dist-code}\ p_{0L}\ p_{1L}\ \delta_R' = \textit{dist-code}\ p_{0R}\ p_{1R}$
  assumes $(c_0,\ c_1) = \textit{combine}\ (p_{0L},\ p_{1L})\ (p_{0R},\ p_{1R})\ l\ ps$
  assumes $(\delta',\ c_0',\ c_1') = \textit{combine-code}\ (\delta_L',\ p_{0L},\ p_{1L})\ (\delta_R',\ p_{0R},\ p_{1R})\ l\ ps$
  shows $c_0 = c_0' \wedge c_1 = c_1'$
⟨*proof*⟩

3.3.2 Divide and Conquer Algorithm

function *closest-pair-rec-code* :: $\textit{point list} \Rightarrow (\textit{point list} * \textit{int} * \textit{point} * \textit{point})$ **where**
  $\textit{closest-pair-rec-code}\ xs = ($
    $\textit{let}\ n = \textit{length}\ xs\ \textit{in}$
    $\textit{if}\ n \leq 3\ \textit{then}$
      $(\textit{mergesort snd}\ xs,\ \textit{closest-pair-bf-code}\ xs)$
    else
      $\textit{let}\ (xs_L,\ xs_R) = \textit{split-at}\ (n\ \textit{div}\ 2)\ xs\ \textit{in}$
      $\textit{let}\ l = \textit{fst}\ (\textit{hd}\ xs_R)\ \textit{in}$

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

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 $n = \mathit{length}\ \mathit{xs}\ \neg\ (n \leq 3)$
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 $1 < \mathit{length}\ \mathit{xs}\ (\mathit{ys}, \delta, c_0, c_1) = \mathit{closest\text{-}pair\text{-}rec\text{-}code}\ \mathit{xs}$
shows $\delta = \mathit{dist\text{-}code}\ c_0\ c_1$
⟨*proof*⟩

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

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

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* (-, -, 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.