Concentrated Liquidity Market Making Operations

Mnacho Echenim

February 10, 2026

Abstract

Automated Market Makers (AMMs) are one of the cornerstones of decentralized finance. They enable users to exchange tokens without the need for order books as would be the case in traditional finance. They involve *liquidity providers*, whose tokens, usually called the *quote* and *base* tokens, can be used in the swap process in exchange for a fee, and *liquidity takers* who swap their tokens. The rules specifying the quantities of tokens that can swapped and those that act as fees are predefined and lead to several categories of AMMs.

Uniswap v3 introduced a new market-making design that improves capital efficiency by allowing liquidity providers to allocate their assets within selected price intervals. By concentrating liquidity over narrower ranges, providers may earn higher fee income than in earlier AMMs, where liquidity is generally distributed uniformly across all prices. Owing to its success, this design was adopted by several decentralized exchanges on various blockchains, including Trader Joe, PancakeSwap v3, Sunswap v3, and Sushiswap v3. These protocols are collectively referred to as *Concentrated Liquidity Market Makers* (CLMMs). Despite differences in implementation details, such as fee structures, tick spacing, or incentive mechanisms, they all rely on the same underlying principles.

In practice, liquidity takers can thus interact with multiple CLMM pools involving the same pair of tokens but different liquidity profiles or fee structures. A crucial task for them is to understand how these pools can be combined, both conceptually and computationally.

Based on the work in [1], we formalize several notions related to CLMMs, and introduce several operations on such pools that permit to derive an optimality result: if two pools admit the same fees, then the defined transformations permit to determine the optimal quantities of quote tokens to trade in each pool in order to recover as many base tokens as possible.

Contents

theory *CLMM-Misc* **imports** *HOL−Analysis.Analysis*

begin

1 Preliminary definitions and results

1.1 Misc

lemma *diff-min-le*:
assumes (*a*::*real*) ≤ *b*
and *x* ≤ *y*
shows *min x b* − *min x a* ≤ *min y b* − *min y a*
using *assms* **by** *linarith*

lemma *sum-ex-strict-pos*:
fixes *f g* :: *'i* ⇒ *'a*::*ordered-cancel-comm-monoid-add*
assumes *finite A*
and ∀ *x*∈*A*. *0* ≤ *f x*
and ∃ *a*∈*A*. *0* < *f a*
shows *0* < *sum f A*
proof −
obtain *a* **where** *0* < *f a* **and** *a*∈ *A* **using** *assms* **by** *auto* **note** *aprop* = *this*
define *B* **where** *B* = *A* − {*a*}
hence *A* = *insert a B* **using** *aprop* **by** *auto*
have *0* < *f a* **using** *aprop* **by** *simp*
also have ... ≤ *f a* + *sum f B*
proof −
have *0* ≤ *sum f B*
proof (*rule sum-nonneg*)
fix *x*
assume *x*∈ *B*
thus *0* ≤ *f x* **using** *B-def assms* **by** *simp*
qed
thus *?thesis* **by** (*simp add*: *add-increasing2*)
qed
also have ... = *sum f* (*insert a B*)
proof (*rule sum.insert*[*symmetric*])
show *finite B* **using** *assms B-def* **by** *simp*

```
    show a∉ B using B-def by simp
  qed
  also have ... = sum f A using ‹A = insert a B› by simp
  finally show ?thesis .
qed

lemma diff-inv-max-le:
  assumes 0 < a
  and (a::real) ≤ b
  and x ≤ y
shows inverse (max y a) − inverse (max y b) ≤
  inverse (max x a) − inverse (max x b)
proof (cases b ≤ y)
  case True
  thus ?thesis using assms by auto
next
  case False
  hence max y b = b by simp
  have max x b = b using False assms by auto
  show ?thesis using ‹max y b = b› assms by fastforce
qed

lemma int-interval-insert:
  fixes a::int
  assumes a ≤ b
  shows {a..< (b+1)} = insert b {a..< b}
proof
  show {a..<b + 1} ⊆ insert b {a..<b}
  proof
    fix x
    assume x ∈ {a..<b + 1}
    show x ∈ insert b {a..<b}
    proof (cases x = b)
      case True
      then show ?thesis by simp
    next
      case False
      hence x < b using ‹x ∈ {a..<b + 1}› by simp
      then show ?thesis using ‹x ∈ {a..<b + 1}› by simp
    qed
  qed
next
  show insert b {a..<b} ⊆ {a..<b + 1} by (simp add: assms)
qed

lemma int-telescoping-sum:
  fixes f::int ⇒ 'a::ab-group-add
  assumes a ≤ b
  shows (∑ i ∈{a..<b}. (f i − f (i+1))) = f a − (f b) using ‹a ≤ b›
```

proof (*induct rule*: *int-ge-induct*)
  case *base*
  then show *?case* **by** *simp*
next
  case (*step i*)
  have $\{a..<i + 1\} = insert\ i\ \{a..<i\}$
    using *int-interval-insert* ‹$a \leq i$› **by** *simp*
  hence $(\sum i \in \{a..<i + 1\}.\ f\ i - f\ (i + 1)) =$
    $(\sum i \in (insert\ i\ \{a..<i\}).\ f\ i - f\ (i + 1))$ **by** *simp*
  also have ... $= f\ i - f\ (i+1) + (\sum i = a..<i.\ f\ i - f\ (i + 1))$
    by (*rule sum.insert*, *auto*)
  also have ... $= f\ i - f\ (i+1) + f\ a - f\ i$ **using** *step* **by** *simp*
  also have ... $= f\ a - f\ (i+1)$ **by** *simp*
  finally show *?case* **.**
qed

lemma *int-telescoping-sum′*:
  fixes *f*::*int* ⇒ *′a*::*ab-group-add*
  assumes $a \leq b$
  shows $(\sum i \in \{a..<b\}.\ (f\ (i+1) - f\ i)) = f\ b - (f\ a)$
proof −
  define *g* **where** $g = (\lambda x.\ - f\ x)$
  have $(\sum i \in \{a..<b\}.\ (f\ (i+1) - f\ i)) = (\sum i \in \{a..<b\}.\ (g\ i - g\ (i+1)))$
    by (*rule sum.cong*, *auto simp add*: *g-def*)
  also have ... $= g\ a - g\ b$ **using** *assms int-telescoping-sum*[*of a b*] **by** *simp*
  also have ... $= f\ b - f\ a$ **using** *g-def* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *int-telescoping-sum-le′*:
  fixes *f*::*int* ⇒ *′a*::*ab-group-add*
  assumes $a \leq b$
  shows $(\sum i \in \{a..b\}.\ (f\ (i+1) - f\ i)) = f\ (b+1) - (f\ a)$
proof −
  have $\{a..b\} = \{a..<\ b+1\}$ **by** *auto*
  thus *?thesis* **using** *assms int-telescoping-sum′*[*of a b+1*] **by** *simp*
qed

lemma *diff-sum-dcomp*:
  fixes *f*::*′a* ⇒ *real*
  assumes *finite A*
  and $A = B \cup C$
  and $B \cap C = \{\}$
shows $x + sum\ f\ A - (y + sum\ f\ B) = x + sum\ f\ C - y$
proof −
  have $sum\ f\ A = sum\ f\ (B \cup C)$ **using** *assms* **by** *simp*
  also have ... $= sum\ f\ B + sum\ f\ C$
  proof (*rule sum.union-inter-neutral*)
    show *finite B* **using** *assms* **by** *simp*

```
    show finite C using assms by simp
    show ∀ x∈B ∩ C. f x = 0 using assms by simp
  qed
  finally have sum f A = sum f B + sum f C .
  thus ?thesis by simp
qed

lemma sum-remove-el:
  assumes finite A
  and x∈ A
  and B = A − {x}
  shows sum f A = f x + sum f B
proof −
  have A = insert x B using assms by auto
  hence sum f A = sum f (insert x B) by simp
  also have ... = f x + sum f B
  proof (rule sum.insert)
    show finite B using assms by simp
    show x∉ B using assms by simp
  qed
  finally show ?thesis .
qed

lemma int-set-bdd-above:
  fixes X::int set
  assumes X ≠ {}
    and bdd-above X
  shows Sup X ∈ X ∀ x ∈ X. x ≤ Sup X
proof −
  from assms obtain x y where x ∈ X and X ⊆ {..y}
    by (auto simp: bdd-above-def)
  hence ∗: finite (X ∩ {x..y}) X ∩ {x..y} ≠ {} and x ≤ y
    by (auto simp: subset-eq)
  have ∃!x∈X. (∀ y∈X. y ≤ x)
  proof
    { fix z assume z ∈ X
      have z ≤ Max (X ∩ {x..y})
      proof cases
        assume x ≤ z with ‹z ∈ X› ‹X ⊆ {..y}› ∗(1) show ?thesis
          by (auto intro!: Max-ge)
      next
        assume ¬ x ≤ z
        then have z < x by simp
        also have x ≤ Max (X ∩ {x..y})
          using ‹x ∈ X› ∗(1) ‹x ≤ y› by (intro Max-ge) auto
        finally show ?thesis by simp
      qed }
    note le = this
    with Max-in[OF ∗] show
```

```
    ex: Max (X ∩ {x..y}) ∈ X ∧ (∀ z∈X. z ≤ Max (X ∩ {x..y}))
   by auto
   fix z assume *: z ∈ X ∧ (∀ y∈X. y ≤ z)
   with le have z ≤ Max (X ∩ {x..y})
    by auto
   moreover have Max (X ∩ {x..y}) ≤ z
    using * ex by auto
   ultimately show z = Max (X ∩ {x..y})
    by auto
  qed
  hence Sup X ∈ X ∧ (∀ y∈X. y ≤ Sup X)
  unfolding Sup-int-def by (rule theI')
 thus Sup X ∈ X ∀ x ∈ X. x ≤ Sup X by auto
qed

definition wedge where
wedge f (i::int) sqp = (λn. if n ≤ i then f n else f (n−1))(i+1:=sqp)

lemma wedge-arg-lt[simp]:
 assumes n ≤ i
 shows wedge f i sqp n = f n using assms unfolding wedge-def by simp

lemma wedge-arg-gt[simp]:
 assumes i+1 < n
 shows wedge f i sqp n = f (n−1) using assms unfolding wedge-def by simp

lemma wedge-arg-eq[simp]:
 shows wedge f i sqp (i+1) = sqp unfolding wedge-def by simp

lemma wedge-strict-mono:
 assumes strict-mono f
 and f i < sqp
 and sqp < f (i+1)
 and g = wedge f i sqp
shows strict-mono g unfolding strict-mono-def
proof (intro allI impI)
 fix x y
 assume (x::int) < y
 show g x < g y
 proof (cases y < i+1)
  case True
  then show ?thesis
   using ‹x < y› assms strict-mono-less by fastforce
 next
  case False
  show ?thesis
  proof (cases y = i+1)
   case True
   hence wedge f i sqp y = sqp by simp
```

```
      have x ≤ i using True ‹x < y› by simp
      hence wedge f i sqp x = f x by simp
      then show ?thesis using ‹wedge f i sqp y = sqp› assms
        by (metis ‹x ≤ i› monoE order-le-less-trans strict-mono-mono)
    next
      case False
      hence i+1 < y using ‹¬ y < i+1› by simp
      hence wedge f i sqp y = f (y − 1) by simp
      show ?thesis
      proof (cases x = i+1)
        case True
        then show ?thesis
          by (metis (mono-tags, lifting) ‹wedge f i sqp y = f (y − 1)›
              ‹x < y› assms(1) assms(3) assms(4) monoD order-less-le-subst1
              strict-mono-on-imp-mono-on wedge-arg-eq zle-diff1-eq)
      next
        case False
        then show ?thesis
          by (metis ‹i + 1 < y› ‹x < y› assms(1) assms(4) diff-strict-right-mono
              linorder-le-less-linear order-le-imp-less-or-eq strict-monoD
              wedge-arg-gt wedge-arg-lt zle-diff1-eq zless-imp-add1-zle)
      qed
    qed
  qed
qed

lemma wedge-gt:
  assumes ∀ i. x < f i
  and x < sqp
  shows ∀ i. x < wedge f j sqp i
  by (smt (verit) assms wedge-arg-eq wedge-arg-gt wedge-arg-lt)

lemma wedge-ge:
  assumes ∀ i. x ≤ f i
  and x ≤ sqp
  shows ∀ i. x ≤ wedge f j sqp i
  by (smt (verit) assms wedge-arg-eq wedge-arg-gt wedge-arg-lt)

lemma wedge-lt:
  assumes ∀ i. f i < x
  and sqp < x
  shows ∀ i. wedge f j sqp i < x
  by (smt (verit) assms wedge-arg-eq wedge-arg-gt wedge-arg-lt)

lemma wedge-le:
  assumes ∀ i. f i ≤ x
  and sqp ≤ x
  shows ∀ i. wedge f j sqp i ≤ x
  by (smt (verit) assms wedge-arg-eq wedge-arg-gt wedge-arg-lt)
```

lemma *wedge-images*:
shows $\forall j.\ \exists k.\ f\ j = wedge\ f\ i\ sqp\ k$
proof
fix j
show $\exists k.\ f\ j = wedge\ f\ i\ sqp\ k$
proof (*cases* $j \leq i$)
case *True*
hence $wedge\ f\ i\ sqp\ j = f\ j$ **by** *simp*
then show *?thesis* **by** *metis*
next
case *False*
hence $i{+}1 \leq j$ **by** *simp*
hence $wedge\ f\ i\ sqp\ (j{+}1) = f\ j$ **by** *simp*
then show *?thesis* **by** *metis*
qed
qed

lemma *wedge-images'*:
assumes $A = \{j.\ j \leq i\}$
and $B = \{j.\ i{+}1 < j\}$
shows $wedge\ f\ i\ sqp\ k \in f`A \cup (f`((\lambda i.\ i{-}1)`B)) \cup \{sqp\}$
proof (*cases* $k \leq i$)
case *True*
hence $wedge\ f\ i\ sqp\ k = f\ k$ **by** *simp*
hence $wedge\ f\ i\ sqp\ k \in f`A$ **using** *assms True* **by** *simp*
then show *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof (*cases* $k = i{+}1$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
hence $i{+}1 < k$ **using** $\langle \neg\ k \leq i \rangle$ **by** *simp*
then show *?thesis* **by** (*simp add*: $\langle i + 1 < k \rangle$ *assms*(*2*))
qed
qed

lemma *wedge-as-ubound*:
assumes $\forall (r{::}real).\ \exists (i{::}int).\ r < f\ i$
shows $\forall r.\ \exists k.\ r < wedge\ f\ i\ sqp\ k$ **using** *assms*
by (*metis wedge-images*)

lemma *wedge-as-lbound*:
assumes $\forall (r{::}real) > 0.\ \exists (i{::}int).\ f\ i < r$
shows $\forall r > 0.\ \exists k.\ wedge\ f\ i\ sqp\ k < r$ **using** *assms*
by (*metis wedge-images*)

lemma *wedge-arg-prop*:
shows $\{j.\ P\ (wedge\ f\ i\ sqp\ j)\} \subseteq \{j.\ j \leq i \wedge P\ (f\ j)\} \cup$
$\{j.\ i{+}1 < j \wedge P\ (f\ (j{-}1))\} \cup \{i{+}1\}$
proof
fix *j*
assume $j \in \{j.\ P\ (wedge\ f\ i\ sqp\ j)\}$
hence *pr*: $P\ (wedge\ f\ i\ sqp\ j)$ **by** *auto*
show $j \in \{j.\ j \leq i \wedge P\ (f\ j)\} \cup \{j.\ i{+}1 < j \wedge P\ (f\ (j{-}1))\} \cup \{i{+}1\}$
proof (*cases* $j \leq i$)
case *True*
hence $wedge\ f\ i\ sqp\ j = f\ j$ **by** *simp*
then show *?thesis* **using** *pr True* **by** *simp*
next
case *False*
show *?thesis*
proof (*cases* $j = i{+}1$)
case *True*
then show *?thesis* **using** *pr* **by** *simp*
next
case *False*
hence $i{+}1 < j$ **using** ‹$\neg\ j \leq i$› **by** *simp*
hence $wedge\ f\ i\ sqp\ j = f\ (j{-}1)$ **by** *simp*
then show *?thesis* **using** *pr* ‹$i{+}1 < j$› **by** *simp*
qed
qed
qed

definition *one-cpl* **where**
$one\text{-}cpl\ phi = (\lambda(i{::}int).\ (1{::}real) - (phi\ i))$

definition *gross-fct* **where**
$gross\text{-}fct\ f\ phi = (\lambda i.\ f\ i\ /\ (one\text{-}cpl\ phi\ i))$

lemma *gross-fct-sgn*:
assumes $phi\ i < (1{::}real)$
shows $((0{::}real) \leq f\ i) \longleftrightarrow (0 \leq gross\text{-}fct\ f\ phi\ i)$ **unfolding** *gross-fct-def*
by (*metis assms diff-ge-0-iff-ge eucl-less-le-not-le le-iff-diff-le-0*
one-cpl-def zero-le-divide-iff)

lemma *gross-fct-nz-eq*:
assumes $phi\ i \neq (1{::}real)$
shows $f\ i = 0 \longleftrightarrow gross\text{-}fct\ f\ phi\ i = 0$ **using** *assms* **unfolding** *gross-fct-def*
by (*simp add*: *one-cpl-def*)

lemma *gross-fct-cong*:
assumes $f\ a = f'\ b$
and $phi\ a = phi'\ b$
shows $gross\text{-}fct\ f\ phi\ a = gross\text{-}fct\ f'\ phi'\ b$ **using** *assms*

unfolding *gross-fct-def* **by** (*simp add*: *one-cpl-def*)

lemma *gross-fct-zero-if*:
assumes *f a = 0*
shows *gross-fct f phi a = 0* **using** *assms* **unfolding** *gross-fct-def* **by** *simp*

definition *fee-union* **where**
fee-union (*l1*::*real*) *l2 f1 f2* = (*l1***f1***(1*−*f2*) + *l2***f2***(1*−*f1*))/
(*l1***(1*−*f2*) + *l2***(1*−*f1*))

lemma *fee-union-pos*:
assumes *0* ≤ *l1*
and *0* ≤ *l2*
and *0* ≤ *f1*
and *0* ≤ *f2*
and *f1* < *1*
and *f2* < *1*
shows *0* ≤ *fee-union l1 l2 f1 f2* **using** *assms* **unfolding** *fee-union-def* **by** *simp*

lemma *fee-union-lt-1*:
assumes *0* ≤ *l1*
and *0* ≤ *l2*
and *0* ≤ *f1*
and *0* ≤ *f2*
and *f1* < *1*
and *f2* < *1*
shows *fee-union l1 l2 f1 f2* < *1*
proof (*cases l1 = 0*)
case *True*
thus *?thesis* **unfolding** *fee-union-def* **by** (*simp add*: *assms*(*6*))
next
case *False*
show *?thesis*
proof (*cases l2 = 0*)
case *True*
then show *?thesis* **unfolding** *fee-union-def* **by** (*simp add*: *assms*(*5*))
next
case *False*
hence *0* < *l1***(1*−*f2*) + *l2***(1*−*f1*) **using** *assms* ‹¬ *l1 = 0*›
by (*simp add*: *less-eq-real-def pos-add-strict*)
moreover have *l1***f1***(1*−*f2*) + *l2***f2***(1*−*f1*) < *l1***(1*−*f2*) + *l2***(1*−*f1*)
using *assms False* ‹¬ *l1 = 0*›
by (*smt* (*verit*, *best*) *mult-less-cancel-left2 mult-less-cancel-right*)
ultimately show *?thesis* **unfolding** *fee-union-def* **by** *simp*
qed
qed

lemma *diff-inv-le*:
assumes *0* < (*x*::*real*)

and $x \leq y$
and $y \leq z$
shows $(y - x)/(z*z) \leq inverse\ x - inverse\ y$
proof −
have $0 < y$ **using** *assms* **by** *simp*
hence $0 < z$ **using** *assms* **by** *simp*
have $(y - x)/(z*z) \leq (y - x)\ /\ (x * y)$ **using** *assms*
by (*simp add*: *frac-le mult-mono*)
also have ... = $inverse\ x - inverse\ y$
using ‹$0 < x$› ‹$0 < y$›
by (*simp add*: *divide-real-def division-ring-inverse-diff*)
finally show *?thesis* **.**
qed

lemma *diff-inv-le′*:
assumes $0 < (x::real)$
and $x \leq y$
and $y \leq z$
and $0 \leq a$
shows $a * (y - x)/(z*z) \leq a * (inverse\ x - inverse\ y)$
proof −
have $0 < y$ **using** *assms* **by** *simp*
hence $0 < z$ **using** *assms* **by** *simp*
have $(y - x)/(z*z) \leq (y - x)\ /\ (x * y)$ **using** *assms*
by (*simp add*: *frac-le mult-mono*)
also have ... = $inverse\ x - inverse\ y$
using ‹$0 < x$› ‹$0 < y$›
by (*simp add*: *divide-real-def division-ring-inverse-diff*)
finally show *?thesis*
by (*metis assms*(*4*) *mult-left-mono times-divide-eq-right*)
qed

lemma *diff-inv-sum-le′*:
assumes $\forall\, i \in I.\ (0::real) < f\ i$
and $\forall\, i \in I.\ f\ i \leq f\ (i+1)$
and $\forall\, i \in I.\ f\ (i+1) \leq z$
and $\forall\, i \in I.\ 0 \leq g\ i$
shows $sum\ (\lambda i.\ g\ i * (f\ (i+1) - f\ i))\ I\ /\ (z * z) \leq$
$sum\ (\lambda i.\ g\ i * (inverse\ (f\ i) - inverse\ (f\ (i+1))))\ I$
proof −
have $sum\ (\lambda i.\ g\ i * (f\ (i+1) - f\ i))\ I\ /\ (z * z) =$
$sum\ (\lambda i.\ g\ i * (f\ (i+1) - f\ i)/\ (z * z))\ I$
by (*rule sum-divide-distrib*)
also have ... ≤ $sum\ (\lambda i.\ g\ i * (inverse\ (f\ i) - inverse\ (f\ (i+1))))\ I$
proof (*rule sum-mono*)
fix i
assume $i \in I$
show $g\ i * (f\ (i + 1) - f\ i)\ /\ (z * z) \leq$
$g\ i * (inverse\ (f\ i) - inverse\ (f\ (i + 1)))$

```
    by (rule diff-inv-le', (auto simp add: assms ‹i ∈ I›))
  qed
  finally show ?thesis .
qed

lemma diff-inv-ge:
  assumes 0 < (x::real)
  and x ≤ y
  and y ≤ z
shows inverse y − inverse z ≤ (z − y)/(x*x)
proof −
  have 0 < y using assms by simp
  hence 0 < z using assms by simp
  hence inverse y − inverse z = (z − y) / (y * z)
    using ‹0 < y› by (simp add: divide-real-def division-ring-inverse-diff)
  also have ... ≤ (z − y)/(x*x) using assms
    by (simp add: frac-le mult-mono)
  finally show ?thesis .
qed

lemma diff-inv-ge':
  assumes 0 < (x::real)
  and x ≤ y
  and y ≤ z
  and 0 ≤ a
shows a * (inverse y − inverse z) ≤ a * (z − y)/(x*x)
proof −
  have 0 < y using assms by simp
  hence 0 < z using assms by simp
  hence inverse y − inverse z = (z − y) / (y * z)
    using ‹0 < y› by (simp add: divide-real-def division-ring-inverse-diff)
  also have ... ≤ (z − y)/(x*x) using assms
    by (simp add: frac-le mult-mono)
  finally show ?thesis
    by (metis assms(4) mult-left-mono times-divide-eq-right)
qed

lemma diff-inv-sum-ge':
  assumes (0::real) < z
  and ∀ i ∈ I. f i ≤ f (i+1)
  and ∀ i∈ I. z ≤ f i
  and ∀ i ∈ I. 0 ≤ g i
shows  sum (λi. g i * (inverse (f i) − inverse (f (i+1)))) I ≤
  sum (λi. g i * (f (i+1) − f i)) I / (z * z)
proof −
  have sum (λi. g i * (inverse (f i) − inverse (f (i+1)))) I ≤
      sum (λi. g i * (f (i+1) − f i)/ (z * z)) I
  proof (rule sum-mono)
    fix i
```

```
    assume i ∈ I
    show g i * (inverse (f i) − inverse (f (i + 1))) ≤
        g i * (f (i + 1) − f i) / (z * z)
      by (rule diff-inv-ge', (auto simp add: assms ‹i ∈ I›))
  qed
  also have ... = sum (λi. g i * (f (i+1) − f i)) I / (z * z)
    by (rule sum-divide-distrib[symmetric])
  finally show ?thesis .
qed
```

1.2 Support of a discrete function

```
definition nz-support where
nz-support f = {i. f i ≠ 0}

lemma nz-supportD:
  assumes i∈ nz-support f
  shows f i ≠ 0 using assms unfolding nz-support-def by simp

lemma wedge-finite-nz-support:
  assumes finite (nz-support f)
  shows finite (nz-support (wedge f i sqp))
proof −
  define A where A = {j. j ≤ i ∧ f j ≠ 0}
  define B where  B = {j. i+1 < j ∧ f (j−1) ≠ 0}
  have inc: nz-support (wedge f i sqp) ⊆ A ∪ B ∪ {i+1}
    using wedge-arg-prop[of λx. x ≠ 0]
    unfolding nz-support-def A-def B-def by auto
  have finite A using assms unfolding nz-support-def A-def by auto
  have B⊆ (λj. j+1)‘{j. f j ≠ 0}
  proof
    fix j
    assume j∈ B
    hence i+1 < j and f (j−1) ≠ 0 unfolding B-def by auto note asm = this
    define k where k = j−1
    hence f k ≠ 0 using asm by simp
    hence k ∈ {j. f j≠ 0} by simp
    thus j ∈ (λj. j+1)‘{j. f j ≠ 0} using ‹k = j−1› by force
  qed
  hence finite B using assms finite-surj unfolding nz-support-def by auto
  thus ?thesis using assms ‹finite A› inc
    by (meson finite.simps finite-UnI finite-subset)
qed

lemma gross-nz-support-eq:
  assumes ∀ i ∈ nz-support f. phi i ≠ 1
  shows nz-support f = nz-support (gross-fct f phi)
  using assms gross-fct-nz-eq gross-fct-zero-if unfolding nz-support-def
  by blast
```

definition *idx-min* **where**
idx-min f = *Inf* (*nz-support f*)

definition *idx-max* **where**
idx-max f = *Sup* (*nz-support f*)

lemma *idx-max-bdd-above-ge*:
fixes *f*::*'a*::*conditionally-complete-linorder* ⇒ *'b*::*zero*
assumes *f i* ≠ *0*
and *bdd-above* (*nz-support f*)
shows *i* ≤ *idx-max f*
proof −
have *i* ∈ *nz-support f* **unfolding** *nz-support-def* **using** *assms* **by** *simp*
thus *?thesis* **unfolding** *idx-max-def*
by (*simp add*: *assms cSup-upper*)
qed

lemma *idx-min-bdd-below-le*:
fixes *f*::*'a*::*conditionally-complete-linorder* ⇒ *'b*::*zero*
assumes *f i* ≠ *0*
and *bdd-below* (*nz-support f*)
shows *idx-min f* ≤ *i*
proof −
have *i* ∈ *nz-support f* **unfolding** *nz-support-def* **using** *assms* **by** *simp*
thus *?thesis* **unfolding** *idx-min-def*
by (*simp add*: *assms cInf-lower*)
qed

lemma *idx-max-finite-ge*:
fixes *f*::*'a*::*conditionally-complete-linorder* ⇒ *'b*::*zero*
assumes *f i* ≠ *0*
and *finite* (*nz-support f*)
shows *i* ≤ *idx-max f* **using** *assms*
by (*simp add*: *idx-max-bdd-above-ge*)

lemma *idx-min-finite-le*:
fixes *f*::*'a*::*conditionally-complete-linorder* ⇒ *'b*::*zero*
assumes *f i* ≠ *0*
and *finite* (*nz-support f*)
shows *idx-min f* ≤ *i* **using** *assms*
by (*simp add*: *idx-min-bdd-below-le*)

lemma *idx-max-finite*:
fixes *f*::*'a*::*conditionally-complete-linorder* ⇒ *'b*::*zero*
assumes *nz-support f* ≠ {}
and *finite* (*nz-support f*)
shows *idx-max f* = *Max* (*nz-support f*) **using** *assms* **unfolding** *idx-max-def*
by (*simp add*: *cSup-eq-Max*)

lemma *idx-min-finite*:
fixes f::$'a$::*conditionally-complete-linorder* $\Rightarrow$ $'b$::*zero*
assumes *nz-support* $f \neq \{\}$
and *finite* (*nz-support* f)
shows *idx-min* f = *Min* (*nz-support* f) **using** *assms* **unfolding** *idx-min-def*
by (*simp add*: *cInf-eq-Min*)

lemma *idx-max-finite-in*:
fixes f::$'a$::*conditionally-complete-linorder* $\Rightarrow$ $'b$::*zero*
assumes *nz-support* $f \neq \{\}$
and *finite* (*nz-support* f)
shows f (*idx-max* f) $\neq$ *0* **using** *assms idx-max-finite*
by (*metis Max-in nz-supportD*)

lemma *idx-min-finite-in*:
fixes f::$'a$::*conditionally-complete-linorder* $\Rightarrow$ $'b$::*zero*
assumes *nz-support* $f \neq \{\}$
and *finite* (*nz-support* f)
shows f (*idx-min* f) $\neq$ *0* **using** *assms idx-min-finite*
by (*metis Min-in nz-supportD*)

lemma *idx-max-finite-gt*:
fixes f::$'a$::*conditionally-complete-linorder* $\Rightarrow$ $'b$::*zero*
assumes *finite* (*nz-support* f)
and *idx-max* $f < i$
shows $f\ i$ = *0*
proof −
have $i \notin$ *nz-support* f **using** *assms idx-max-finite*
by (*metis Max-ge emptyE linorder-not-less*)
thus *?thesis* **by** (*simp add*: *nz-support-def*)
qed

lemma *idx-min-finite-lt*:
fixes f::$'a$::*conditionally-complete-linorder* $\Rightarrow$ $'b$::*zero*
assumes *finite* (*nz-support* f)
and $i <$ *idx-min* f
shows $f\ i$ = *0*
proof −
have $i \notin$ *nz-support* f **using** *assms idx-min-finite*
by (*metis Min-le emptyE linorder-not-less*)
thus *?thesis* **by** (*simp add*: *nz-support-def*)
qed

lemma *idx-min-finite-max*:
fixes f::$'a$::*conditionally-complete-linorder* $\Rightarrow$ $'b$::*zero*
assumes *nz-support* $f \neq \{\}$
and *finite* (*nz-support* f)
and $\bigwedge j.\ j < i \Longrightarrow f\ j$ = *0*

shows $i \leq$ *idx-min f*
proof (*rule ccontr*)
  assume $\neg$ $i \leq$ *idx-min f*
  hence *idx-min f* $<$ *i* **by** *simp*
  hence *f* (*idx-min f*) $=$ *0* **using** *assms* **by** *simp*
  thus *False* **using** *idx-min-finite-in assms* **by** *metis*
qed

lemma *idx-min-max-finite*:
  fixes *f*::'*a*::*conditionally-complete-linorder* $\Rightarrow$ '*b*::*zero*
  assumes *nz-support f* $\neq$ {}
  and *finite* (*nz-support f*)
shows *idx-min f*$\leq$ *idx-max f*
proof −
  have *idx-max f* $\in$ *nz-support f*
    using *idx-max-finite-in assms* **unfolding** *nz-support-def* **by** *simp*
  thus *idx-min f* $\leq$ *idx-max f*
    using *idx-min-finite-le assms* **unfolding** *nz-support-def* **by** *simp*
qed

lemma *idx-min-finiteI*:
  fixes *f*::'*a*::*conditionally-complete-linorder* $\Rightarrow$ '*b*::*zero*
  assumes *finite* (*nz-support f*)
  and *f i* $\neq$ *0*
  and $\bigwedge$*j*. *j* $<$ *i*$\Longrightarrow$ *f j* $=$ *0*
shows *i* $=$ *idx-min f*
proof −
  have *nz-support f* $\neq$ {} **using** *assms* **unfolding** *nz-support-def* **by** *auto*
  thus *?thesis*
    using *assms idx-min-finite-le nless-le* **by** (*metis idx-min-finite-in*)
qed

lemma *idx-min-finite-ge*:
  fixes *f*::'*a*::*conditionally-complete-linorder* $\Rightarrow$ '*b*::*zero*
  assumes *finite* (*nz-support f*)
  and *nz-support f* $\neq$ {}
  and $\bigwedge$*j*. *j* $\leq$ *i*$\Longrightarrow$ *f j* $=$ *0*
shows *i* $\leq$ *idx-min f*
  by (*meson assms idx-min-finite-in nle-le*)

lemma *idx-max-finiteI*:
  fixes *f*::'*a*::*conditionally-complete-linorder* $\Rightarrow$ '*b*::*zero*
  assumes *finite* (*nz-support f*)
  and *f i* $\neq$ *0*
  and $\bigwedge$*j*. *j* $>$ *i* $\Longrightarrow$ *f j* $=$ *0*
shows *i* $=$ *idx-max f*
proof −
  have *nz-support f* $\neq$ {} **using** *assms* **unfolding** *nz-support-def* **by** *auto*
  thus *?thesis* **using** *assms*

by (*meson idx-max-finite-gt idx-max-finite-in linorder-less-linear*)
qed

lemma *idx-max-finite-le*:
fixes f::$'a$::*conditionally-complete-linorder* $\Rightarrow$ $'b$::*zero*
assumes *finite* (*nz-support f*)
and *nz-support f* $\neq$ {}
and $\bigwedge j$. $i \leq j \Longrightarrow f\, j = 0$
shows *idx-max f* $\leq i$
using *assms idx-max-finite-in linorder-linear* **by** *auto*

definition *idx-min-img* **where**
idx-min-img g f = *g* (*idx-min f*)

lemma *idx-min-img-eq*:
assumes $\forall x.\ f\, x = 0 \longleftrightarrow f'\, x = 0$
shows *idx-min-img g f* = *idx-min-img g f'* **unfolding** *idx-min-img-def* **using** *assms*
by (*simp add*: *idx-min-def nz-support-def*)

definition *idx-max-img* **where**
idx-max-img g f = *g* (*idx-max f* + *1*)

lemma *idx-max-img-eq*:
assumes $\forall x.\ f\, x = 0 \longleftrightarrow f'\, x = 0$
shows *idx-max-img g f* = *idx-max-img g f'* **unfolding** *idx-max-img-def* **using** *assms*
by (*simp add*: *idx-max-def nz-support-def*)

lemma *non-zero-liq-interv*:
fixes *i*::$'a$::*conditionally-complete-linorder*
assumes *finite* (*nz-support L*)
and *L i* $\neq$ *0*
shows *i* $\in$ {*idx-min L* .. *idx-max L*}
using *assms idx-max-finite-ge idx-min-finite-le* **by** *auto*

end
theory *Grid-Information* **imports** *CLMM-Misc*

begin

2 Grid information

2.1 Definitions

A grid information consists of three functions defining the way a grid is associated to (square root) prices, the liquidity on each price range and the fees on each price range.

type-synonym *grid-info* = (*int*⇒ *real*) × (*int*⇒ *real*) × (*int*⇒*real*)

definition *grd*::*grid-info* ⇒ (*int*⇒ *real*) **where**
grd P = *fst P*

definition *lq*::*grid-info* ⇒ (*int*⇒ *real*) **where**
lq P = *fst* (*snd P*)

definition *fee*::*grid-info* ⇒ (*int*⇒ *real*) **where**
fee P = *snd* (*snd P*)

Although several results are formalized in a generalized setting, the pools of interest are those admitting a finite range with nonzero liquidity.

definition *finite-liq* **where**
finite-liq P ⟷ *finite* (*nz-support* (*lq P*))

lemma *finite-liqI*[*intro*]:
assumes *finite* {*i*. *lq P i* ≠ *0*}
shows *finite-liq P* **using** *assms* **unfolding** *finite-liq-def nz-support-def*
by *simp*

lemma *finite-liqD*:
assumes *finite-liq P*
shows *finite* {*i*. *lq P i* ≠ *0*} **using** *assms*
unfolding *finite-liq-def nz-support-def*
by *simp*

definition *grd-min* **where**
grd-min P = *idx-min-img* (*grd P*) (*lq P*)

definition *grd-max* **where**
grd-max P = *idx-max-img* (*grd P*) (*lq P*)

lemma *grd-min-pos*:
assumes *nz-support* (*lq P*) ≠ {}
and ⋀*i*. *0* < *grd P i*
shows *0* < *grd-min P*
by (*simp add*: *assms*(*2*) *idx-min-img-def grd-min-def*)

lemma *grd-max-gt*:
assumes *nz-support* (*lq P*) ≠ {}
and ⋀*i*. *0* < *grd P i*

shows *0 < grd-max P*
by (*simp add*: *assms*(*2*) *idx-max-img-def grd-max-def*)

locale *finite-nz-support* =
fixes *L*::*int* ⇒ *real*
assumes *fin-nz-sup*: *finite* (*nz-support L*)

locale *finite-liq-pool* =
fixes *P*
assumes *fin-liq*: *finite-liq P*

sublocale *finite-liq-pool* ⊆ *finite-nz-support lq P*
using *fin-liq finite-liq-def finite-nz-support.intro* **by** *auto*

context *finite-liq-pool*
begin

lemma *idx-max-mem*:
assumes *nz-support* (*lq P*) ≠ {}
shows *idx-max* (*lq P*) ∈ *nz-support* (*lq P*)
proof −
have *finite* (*nz-support* (*lq P*))
by (*simp add*: *fin-liq finite-liqD nz-support-def*)
thus *?thesis* **using** *assms* **unfolding** *idx-max-def* **by** (*metis Max-in cSup-eq-Max*)
qed

lemma *idx-min-mem*:
assumes *nz-support* (*lq P*) ≠ {}
shows *idx-min* (*lq P*) ∈ *nz-support* (*lq P*)
proof −
have *finite* (*nz-support* (*lq P*))
by (*simp add*: *fin-liq finite-liqD nz-support-def*)
thus *?thesis* **using** *assms* **unfolding** *idx-min-def*
by (*metis finite-less-Inf-iff nless-le not-le-imp-less*)
qed

lemma *grd-min-max*:
assumes *nz-support* (*lq P*) ≠ {}
and *mono* (*grd P*)
shows *grd-min P* ≤ *grd-max P*
unfolding *grd-min-def grd-max-def idx-min-img-def idx-max-img-def*
idx-max-def
by (*metis add.commute add-increasing assms fin-nz-sup idx-min-def*
idx-min-mem le-cSup-finite zero-less-one-class.zero-le-one
monoD)

lemma *finite-liq-gross-fct*:
shows *finite* {*i*. *gross-fct* (*lq P*) (*fee P*) *i* ≠ *0*}
using *finite-liqD fin-nz-sup* **unfolding** *gross-fct-def nz-support-def* **by** *auto*

end

2.2 Gross and net token quantities

2.2.1 General definitions

We define generic functions that are afterwards instantiated to represent the gross (resp. net) quantities of base (resp. quote) tokens in a pool.

definition *rng-token* **where**
rng-token = (λ*dff L* (*pi*::*real*) *i*. ((*L i*)::*real*) ∗ (*dff pi* (*i*::*int*)))

lemma *rng-token-pos*:
assumes *0* ≤ *L i*
and *0* ≤ *dff x i*
shows *0* ≤ *rng-token dff L x i* **unfolding** *rng-token-def*
using *zero-le-mult-iff assms* **by** *auto*

lemma *rng-token-continuous-on*:
assumes *continuous-on A* (λ*pi*. *dff pi i*)
shows *continuous-on A* (λ*pi*. *rng-token dff L pi i*) **unfolding** *rng-token-def*
by (*rule continuous-on-mult-left*, *simp add*: *assms*)

(Anti)-monotonicity is preserved by the generic function *rng-token*.

lemma *rng-token-mono*:
assumes *0* ≤ *L i*
and *mono* (λ*pi*. *dff pi i*)
shows *mono* (λ*pi*. *rng-token dff L pi i*)
proof
fix *x y*::*real*
assume *x* ≤ *y*
show *rng-token dff L x i* ≤ *rng-token dff L y i*
unfolding *rng-token-def*
proof (*rule ordered-comm-semiring-class.comm-mult-left-mono*)
show *0* ≤ *L i* **using** *assms* **by** *simp*
show *dff x i* ≤ *dff y i* **using** *assms monoD* ‹*x* ≤ *y*› **by** *auto*
qed
qed

lemma *rng-token-strict-mono*:
assumes (*0*::*real*) < *L i*
and *strict-mono* (λ*pi*. *dff pi i*)
shows *strict-mono* (λ*pi*. *rng-token dff L pi i*)
proof
fix *x y*::*real*
assume *x* < *y*
hence *dff x i* < *dff y i* **using** *assms strict-monoD* **by** *auto*
thus *rng-token dff L x i* < *rng-token dff L y i*
using *assms* **unfolding** *rng-token-def* **by** *simp*

qed

lemma *rng-token-antimono*:
assumes $0 \leq L\ i$
and *antimono* ($\lambda pi.$ *dff pi i*)
shows *antimono* ($\lambda pi.$ *rng-token dff L pi i*)
proof
fix *x y*::*real*
assume $x \leq y$
show *rng-token dff L y i* $\leq$ *rng-token dff L x i*
unfolding *rng-token-def*
proof (*rule ordered-comm-semiring-class.comm-mult-left-mono*)
show $0 \leq L\ i$ **using** *assms* **by** *simp*
show *dff y i* $\leq$ *dff x i* **using** *assms antimonoD* ‹$x \leq y$› **by** *auto*
qed
qed

lemma *rng-token-add*:
assumes $\forall\, i.\ L\ i = L1\ i + L2\ i$
shows *rng-token dff L x i* = *rng-token dff L1 x i* +
rng-token dff L2 x i
using *assms* **unfolding** *rng-token-def*
by (*simp add*: *ring-class.ring-distribs*(*2*))

The generic function for the gross or net token quantities on the entire pool is obtained by summation of *rng-token* on all ranges.

definition *gen-token* **where**
gen-token = (λ*dff L pi*. (*infsum* (*rng-token dff L pi*) *UNIV*))

lemma *gen-token-pos*:
assumes $\forall\, i.\ 0 \leq L\ i$
and $\forall\, i.\ 0 \leq dff\ x\ i$
shows $0 \leq$ *gen-token dff L x* **unfolding** *gen-token-def*
proof (*rule infsum-nonneg*)
show $\bigwedge y.\ y \in UNIV \Longrightarrow 0 \leq$ *rng-token dff L x y*
using *assms* **unfolding** *rng-token-def* **by** *simp*
qed

lemma *gen-token-mono*:
assumes $\forall\, i.\ 0 \leq L\ i$
and $\forall\, x.$ *rng-token dff L x summable-on UNIV*
and $\forall\, i.$ *mono* ($\lambda pi.$ *dff pi i*)
shows *mono* ($\lambda pi.$ *gen-token dff L pi*)
proof
fix *x y*::*real*
assume $x \leq y$
show *gen-token dff L x* $\leq$ *gen-token dff L y* **unfolding** *gen-token-def*
proof (*rule infsum-mono*)
show *rng-token dff L x summable-on UNIV* **using** *assms* **by** *simp*

show *rng-token dff L y summable-on UNIV* **using** *assms* **by** *simp*
show $\bigwedge i.\ i \in UNIV \Longrightarrow$ *rng-token dff L x i* $\leq$ *rng-token dff L y i*
using *rng-token-mono assms*
by (*meson* ‹$x \leq y$› *monotoneD*)
qed
qed

lemma *gen-token-antimono*:
assumes $\forall\, i.\ 0 \leq L\ i$
and $\forall\, x.$ *rng-token dff L x summable-on UNIV*
and $\forall\, i.$ *antimono* ($\lambda pi.$ *dff pi i*)
shows *antimono* ($\lambda pi.$ *gen-token dff L pi*)
proof
fix *x y*::*real*
assume $x \leq y$
show *gen-token dff L y* $\leq$ *gen-token dff L x* **unfolding** *gen-token-def*
proof (*rule infsum-mono*)
show *rng-token dff L x summable-on UNIV* **using** *assms* **by** *simp*
show *rng-token dff L y summable-on UNIV* **using** *assms* **by** *simp*
show $\bigwedge i.\ i \in UNIV \Longrightarrow$ *rng-token dff L y i* $\leq$ *rng-token dff L x i*
proof −
fix *i*::*int*
assume *i*$\in$ *UNIV*
have *antimono* ($\lambda pi.$ *rng-token dff L pi i*)
using *rng-token-antimono assms* **by** *simp*
thus *rng-token dff L y i* $\leq$ *rng-token dff L x i*
using ‹$x \leq y$› *antimono-def* **by** *metis*
qed
qed
qed

2.2.2 Finite support restriction

context *finite-nz-support*

begin

lemma *finite-nonzero-summable*:
shows *rng-token dff L x summable-on UNIV*
proof (*rule finite-nonzero-values-imp-summable-on*)
define *rg* **where** *rg* = *rng-token dff L x*
define *Lnz* **where** *Lnz* = $\{i.\ L\ i \neq 0\}$
have *finite Lnz* **using** *fin-nz-sup* **unfolding** *Lnz-def*
by (*simp add*: *nz-support-def*)
define *Lz* **where** *Lz* = *UNIV* − *Lnz*
have $\forall\, i \in$ *Lz*. *L i* = *0* **using** *Lnz-def Lz-def* **by** *simp*
hence $\forall\, i \in$ *Lz*. *rg i* = *0* **unfolding** *rg-def rng-token-def* **by** *simp*
hence $\forall\, i.$ *rg i* $\neq$ *0* $\longrightarrow i \in$ *Lnz* **using** *Lz-def Lnz-def* **by** *blast*
show *finite* $\{x \in UNIV.$ *rg x* $\neq$ *0*$\}$

using ‹∀ *i*. *rg i* ≠ *0* ⟶ *i*∈ *Lnz*› ‹*finite Lnz*›
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq rev-finite-subset subsetI*)
qed

lemma *gen-token-antimono-finite*:
assumes ∀ *i*. *0* ≤ *L i*
and ∀ *i*. *antimono* (λ*pi*. *dff pi i*)
shows *antimono* (λ*pi*. *gen-token dff L pi*)
proof (*rule gen-token-antimono*)
show ∀ *x*. *rng-token dff L x summable-on UNIV*
using *finite-nonzero-summable assms* **by** *simp*
qed (*simp add*: *assms*)+

lemma *gen-token-sum*:
shows *gen-token dff L x* =
sum (*rng-token dff L x*) {*i*. *L i* ≠ *0*}
proof −
define *rg* **where** *rg* = *rng-token dff L x*
define *Lnz* **where** *Lnz* = {*i*. *L i* ≠ *0*}
have *finite Lnz* **using** *fin-nz-sup*
unfolding *Lnz-def nz-support-def* **by** *simp*
define *Lz* **where** *Lz* = *UNIV* − *Lnz*
have ∀ *i*∈ *Lz*. *L i* = *0* **using** *Lnz-def Lz-def* **by** *simp*
hence ∀ *i*∈ *Lz*. *rg i* = *0* **unfolding** *rg-def rng-token-def* **by** *simp*
have *infsum rg UNIV* = *infsum rg* (*Lnz* ∪ *Lz*) **unfolding** *Lz-def Lnz-def*
by *simp*
also have ... = *infsum rg Lnz* + *infsum rg Lz*
proof (*rule infsum-Un-disjoint*)
show *rg summable-on Lz*
using ‹∀ *i*∈ *Lz*. *rg i* = *0*› *summable-on-0*[*of Lz rg*] **by** *simp*
show *rg summable-on Lnz* **using** ‹*finite Lnz*› **by** *simp*
show *Lnz* ∩ *Lz* = {} **unfolding** *Lz-def* **by** *simp*
qed
also have ... = *infsum rg Lnz* **using** *infsum-0* ‹∀ *i*∈ *Lz*. *rg i* = *0*›
by *fastforce*
also have ... = *sum rg Lnz* **using** ‹*finite Lnz*› **by** *simp*
finally show *?thesis* **using** *rg-def Lnz-def* **unfolding** *gen-token-def* **by** *simp*
qed

lemma *gen-token-continuous*:
assumes ∀ *i*. *continuous-on A* (λ*pi*. *dff pi i*)
shows *continuous-on A* (*gen-token dff L*)
proof −
have *gen-token dff L* =
(λ*pi*. *sum* (*rng-token dff L pi*) {*i*. *L i* ≠ *0*})
using *gen-token-sum assms* **by** *auto*
moreover have *continuous-on A*
(λ*pi*. *sum* (*rng-token dff L pi*) {*i*. *L i* ≠ *0*})
proof (*rule continuous-on-sum*)

```
    fix i::int
    assume i ∈ {i. L i ≠ 0}
    show continuous-on A (λx. rng-token dff L x i)
      by (rule rng-token-continuous-on, simp add: assms)
  qed
  ultimately show ?thesis by simp
qed

lemma gen-token-strict-mono:
  assumes ∀ i. 0 ≤ L i
  and nz-support L ≠ {}
  and ∀ i. strict-mono (λpi. dff pi i)
  shows strict-mono (λpi. gen-token dff L pi)
proof
  fix x y::real
  assume x < y
  define M where M = {i. L i ≠ 0}
  have gen-token dff L x = sum (rng-token dff L x) M
    using gen-token-sum unfolding M-def by simp
  also have ... < sum (rng-token dff L y) M
  proof (rule sum-strict-mono)
    show finite M
      unfolding M-def by (metis fin-nz-sup nz-support-def)
    show M ≠ {} using assms unfolding nz-support-def M-def by simp
    fix j
    assume j ∈ M
    hence 0 < L j using assms less-eq-real-def unfolding M-def by auto
    hence strict-mono (λpi. rng-token dff L pi j)
      using assms rng-token-strict-mono by simp
    thus rng-token dff L x j < rng-token dff L y j
      using ‹x < y› strict-monoD by auto
  qed
  also have ... = gen-token dff L y
    using gen-token-sum unfolding M-def by simp
  finally show gen-token dff L x < gen-token dff L y .
qed

lemma gen-token-add:
  assumes ∀ i. L i = L1 i + L2 i
  and ∀ i. 0 ≤ L1 i
  and ∀ i. 0 ≤ L2 i
  shows gen-token dff L x = gen-token dff L1 x + gen-token dff L2 x
proof −
  have sub1: {i. L1 i ≠ 0} ⊆ {i. L i ≠ 0}
     by (simp add: Collect-mono add-nonneg-eq-0-iff assms)
  have sub2: {i. L2 i ≠ 0} ⊆ {i. L i ≠ 0}
    by (simp add: Collect-mono add-nonneg-eq-0-iff assms)
  have gen-token dff L x = sum (rng-token dff L x) {i. L i ≠ 0}
    using gen-token-sum by simp
```

```
  also have ... = sum (λi. rng-token dff L1 x i + rng-token dff L2 x i)
      {i. L i ≠ 0}
    by (rule sum.cong, (simp add: assms rng-token-add)+)
  also have ... = sum (rng-token dff L1 x) {i. L i ≠ 0} +
    sum (rng-token dff L2 x) {i. L i ≠ 0}
    by (rule sum.distrib)
  also have ... = gen-token dff L1 x + gen-token dff L2 x
  proof −
    have gen-token dff L1 x = sum (rng-token dff L1 x) {i. L1 i ≠ 0}
    proof (rule finite-nz-support.gen-token-sum)
      show finite-nz-support L1 using sub1 fin-nz-sup
        by (metis finite-nz-support.intro nz-support-def rev-finite-subset)
    qed
    also have ... =
        sum (rng-token dff L1 x) {i. L i ≠ 0}
    proof (rule sum.mono-neutral-left)
      show finite {i. L i ≠ 0}
        using fin-nz-sup unfolding nz-support-def by simp
      show {i. L1 i ≠ 0} ⊆ {i. L i ≠ 0} using sub1 by simp
      show ∀ i∈{i. L i ≠ 0} − {i. L1 i ≠ 0}. rng-token dff L1 x i = 0
        unfolding rng-token-def by simp
    qed
    finally have 1: gen-token dff L1 x =
      sum (rng-token dff L1 x) {i. L i ≠ 0} .
    have gen-token dff L2 x = sum (rng-token dff L2 x) {i. L2 i ≠ 0}
    proof (rule finite-nz-support.gen-token-sum)
      show finite-nz-support L2 using sub2 fin-nz-sup
        by (metis finite-nz-support.intro nz-support-def rev-finite-subset)
    qed
    also have ... = sum (rng-token dff L2 x) {i. L i ≠ 0}
    proof (rule sum.mono-neutral-left)
      show finite {i. L i ≠ 0}
        using fin-nz-sup unfolding nz-support-def by simp
      show {i. L2 i ≠ 0} ⊆ {i. L i ≠ 0} using sub2 by simp
      show ∀ i∈{i. L i ≠ 0} − {i. L2 i ≠ 0}. rng-token dff L2 x i = 0
        unfolding rng-token-def by simp
    qed
    finally have gen-token dff L2 x =
        sum (rng-token dff L2 x) {i. L i ≠ 0} .
    thus ?thesis using 1 by simp
  qed
  finally show ?thesis .
qed

end
```

2.3 Gross and net quantities of quote tokens

2.3.1 Generic functions for quote tokens

definition *gamma-min-diff* **where**
gamma-min-diff gamma =
(λ(*pi*::*real*) *i*. (*min pi* (*gamma* (*i*+(*1*::*int*)))) − (*min pi* (*gamma i*)))

lemma *gamma-min-diff-pos*:
assumes *gamma i* ≤ *gamma* (*i*+*1*)
shows *0* ≤ *gamma-min-diff gamma x i*
proof −
show *?thesis*
proof (*cases x* ≤ *gamma i*)
case *True*
hence *min x* (*gamma i*) = *x* **by** *simp*
have *x* ≤ *gamma* (*i* + *1*) **using** *True assms* **by** *simp*
hence *min x* (*gamma* (*i* + *1*)) = *x* **by** *simp*
then show *?thesis* **using** ‹*min x* (*gamma i*) = *x*›
unfolding *gamma-min-diff-def* **by** *simp*
next
case *False*
hence *min x* (*gamma i*) = *gamma i* **by** *simp*
show *?thesis*
proof (*cases x* ≤ *gamma* (*i* + *1*))
case *True*
hence *min x* (*gamma* (*i* + *1*)) = *x* **by** *simp*
then show *?thesis* **using** ‹*min x* (*gamma i*) = *gamma i*› *False*
unfolding *gamma-min-diff-def* **by** *simp*
next
case *False*
hence *min x* (*gamma* (*i* + *1*)) = *gamma* (*i*+*1*) **by** *simp*
then show *?thesis* **using** *assms* **unfolding** *gamma-min-diff-def* **by** *simp*
qed
qed
qed

lemma *gamma-min-diff-continuous*:
shows *continuous-on A* (λ(*pi*::*real*). *gamma-min-diff gamma pi i*)
unfolding *gamma-min-diff-def*
proof (*rule continuous-on-diff*)
show *continuous-on A* (λ*x*. *min x* (*gamma* (*i* + *1*))) **using** *continuous-on-min*
continuous-on-const continuous-on-id **by** *blast*
show *continuous-on A* (λ*x*. *min x* (*gamma i*)) **using** *continuous-on-min*
continuous-on-const continuous-on-id **by** *blast*
qed

lemma *gamma-min-diff-mono*:
fixes *gamma*::*int* ⇒ *real*
assumes *gamma i* ≤ *gamma* (*i*+*1*)

shows *mono* (*λpi. gamma-min-diff gamma pi i*)
unfolding *gamma-min-diff-def*
proof
fix *x y*::*real*
assume *asm*: $x \leq y$
show *min x* (*gamma* (*i* + *1*)) − *min x* (*gamma i*) ≤
min y (*gamma* (*i* + *1*)) − *min y* (*gamma i*)
proof (*rule diff-min-le*)
show $x \leq y$ **using** *asm* **.**
show *gamma i* ≤ *gamma* (*i* + *1*) **using** *assms* **by** *simp*
qed
qed

definition *rng-gen-quote* **where**
rng-gen-quote gamma = (*λL pi i. rng-token* (*gamma-min-diff gamma*) *L pi i*)

lemma *rng-gen-quote-pos*:
assumes *0* ≤ *L i*
and *gamma i* ≤ *gamma* (*i+1*)
shows *0* ≤ *rng-gen-quote gamma L x i* **unfolding** *rng-gen-quote-def*
by (*rule rng-token-pos*, *auto simp add*: *assms gamma-min-diff-pos*)

lemma *rng-gen-quote-continuous-on*:
shows *continuous-on A* (*λpi. rng-gen-quote gamma L pi i*)
unfolding *rng-gen-quote-def*
by (*rule rng-token-continuous-on*, *rule gamma-min-diff-continuous*)

lemma *rng-gen-quote-mono*:
assumes *0* ≤ *L i*
and *gamma i* ≤ *gamma* (*i+1*)
shows *mono* (*λpi. rng-gen-quote gamma L pi i*)
proof
fix *x y*::*real*
assume *asm*: $x \leq y$
show *rng-gen-quote gamma L x i* ≤ *rng-gen-quote gamma L y i*
unfolding *rng-gen-quote-def rng-token-def*
proof (*rule ordered-comm-semiring-class.comm-mult-left-mono*)
show *0* ≤ *L i* **using** *assms* **by** *simp*
show *gamma-min-diff gamma x i* ≤ *gamma-min-diff gamma y i*
using *gamma-min-diff-mono asm monoD assms* **by** *blast*
qed
qed

definition *gen-quote* **where**
gen-quote = (*λ gamma L pi. gen-token* (*gamma-min-diff gamma*) *L pi*)

lemma *gen-quote-zero*:
assumes *mono gamma*
and ⋀*i. gamma i* < *sqp* ⟹ *L i* = *0*

shows *gen-quote gamma L sqp = 0* **unfolding** *gen-quote-def gen-token-def*
proof (*rule infsum-0*)
fix *i*
show *rng-token* (*gamma-min-diff gamma*) *L sqp i = 0*
proof (*cases sqp* ≤ *gamma i*)
case *True*
hence *sqp* ≤ *gamma* (*i+1*) **using** *assms monoD*
by (*metis dual-order.trans zle-add1-eq-le zless-add1-eq*)
hence *gamma-min-diff gamma sqp i = 0*
using *True* **unfolding** *gamma-min-diff-def* **by** *simp*
then show *?thesis* **unfolding** *rng-token-def* **by** *simp*
next
case *False*
hence *L i = 0* **using** *assms* **by** *simp*
then show *?thesis* **unfolding** *rng-token-def* **by** *simp*
qed
qed

lemma *gen-quote-pos*:
assumes ∀ *i. 0* ≤ *L i*
and ∀ *i. gamma i* ≤ *gamma* (*i+1*)
shows *0* ≤ *gen-quote gamma L x* **unfolding** *gen-quote-def*
using *gen-token-pos gamma-min-diff-pos assms* **by** *simp*

lemma *gen-quote-mono*:
assumes ∀ *i. 0* ≤ *L i*
and ∀ *x. rng-token* (*gamma-min-diff gamma*) *L x summable-on UNIV*
and ∀ *i. gamma i* ≤ *gamma* (*i+1*)
shows *mono* (*gen-quote gamma L*) **unfolding** *gen-quote-def*
proof (*rule gen-token-mono*)
show ∀ *i. mono* (λ*pi. gamma-min-diff gamma pi i*)
using *gamma-min-diff-mono assms* **by** *simp*
qed (*simp add*: *assms*)+

2.3.2 Finite support restriction

context *finite-nz-support*
begin

lemma *gen-quote-mono-finite*:
assumes ∀ *i. 0* ≤ *L i*
and ∀ *i. gamma i* ≤ *gamma* (*i+1*)
shows *mono* (*gen-quote gamma L*)
proof (*rule gen-quote-mono*)
show ∀ *x. rng-token* (*gamma-min-diff gamma*) *L x summable-on UNIV*
using *finite-nonzero-summable assms* **by** *simp*
qed (*simp add*: *assms*)+

lemma *gen-quote-continuous*:

shows *continuous-on A* (*gen-quote gamma L*) **unfolding** *gen-quote-def*
proof (*rule gen-token-continuous*)
show ∀ *i*. *continuous-on A* (λ*pi*. *gamma-min-diff gamma pi i*) **using**
gamma-min-diff-continuous **by** *simp*
qed

lemma *gen-quote-IVT*:
assumes (*idx-min-img gamma L*) ≤ (*idx-max-img gamma L*)
and *gen-quote gamma L* (*idx-min-img gamma L*) ≤ *y*
and *y* ≤ *gen-quote gamma L* (*idx-max-img gamma L*)
shows ∃ *pi* ≥ (*idx-min-img gamma L*). *pi* ≤ *idx-max-img gamma L* ∧
gen-quote gamma L pi = *y*
proof (*rule IVT*)
show ∀ *pi*. *idx-min-img gamma L* ≤ *pi* ∧ *pi* ≤ *idx-max-img gamma L* ⟶
isCont (*gen-quote gamma L*) *pi*
proof (*intro allI impI*)
fix *pi*
assume (*idx-min-img gamma L*) ≤ *pi* ∧ *pi* ≤ (*idx-max-img gamma L*)
show *isCont* (*gen-quote gamma L*) *pi* **using** *gen-quote-continuous*
by (*simp add*: *continuous-on-eq-continuous-within assms*)
qed
qed (*simp add*: *assms*)+

lemma *gen-quote-ne*:
assumes (*idx-min-img gamma L*) ≤ (*idx-max-img gamma L*)
and *gen-quote gamma L* (*idx-min-img gamma L*) ≤ *y*
and *y* ≤ *gen-quote gamma L* (*idx-max-img gamma L*)
shows (*gen-quote gamma L*)−‘ {*y*} ≠ {} **using** *gen-quote-IVT assms* **by** *blast*

lemma *finite-support-sum*:
assumes ⋀*i*. *L i* = *0* ⟹ *f L i* = *0*
shows *infsum* (*rng-token dff* (*f L*) *x*) *UNIV* =
sum (*rng-token dff* (*f L*) *x*) {*i*. *L i* ≠ *0*}
proof −
define *rg* **where** *rg* = *rng-token dff* (*f L*) *x*
define *Lnz* **where** *Lnz* = {*i*. *L i* ≠ *0*}
have *finite Lnz* **using** *assms fin-nz-sup* **unfolding** *Lnz-def*
by (*simp add*: *nz-support-def*)
define *Lz* **where** *Lz* = *UNIV* − *Lnz*
have ∀ *i*∈ *Lz*. (*f L*) *i* = *0* **using** *assms Lnz-def Lz-def* **by** *simp*
hence ∀ *i*∈ *Lz*. *rg i* = *0* **unfolding** *rg-def rng-token-def* **by** *simp*
have *infsum rg UNIV* = *infsum rg* (*Lnz* ∪ *Lz*) **unfolding** *Lz-def Lnz-def*
by *simp*
also have ... = *infsum rg Lnz* + *infsum rg Lz*
proof (*rule infsum-Un-disjoint*)
show *rg summable-on Lz*
using ‹∀ *i*∈ *Lz*. *rg i* = *0*› *summable-on-0*[*of Lz rg*] **by** *simp*
show *rg summable-on Lnz* **using** ‹*finite Lnz*› **by** *simp*
show *Lnz* ∩ *Lz* = {} **unfolding** *Lz-def* **by** *simp*

qed
also have ... = *infsum rg Lnz* **using** *infsum-0* ‹∀ *i*∈ *Lz*. *rg i* = *0*›
by *fastforce*
also have ... = *sum rg Lnz* **using** ‹*finite Lnz*› **by** *simp*
finally show *?thesis* **using** *rg-def Lnz-def* **by** *simp*
qed

lemma *gen-quote-plus*:
assumes ∀ *i*. *L i* = *L1 i* + *L2 i*
and ∀ *i*. *0* ≤ *L1 i*
and ∀ *i*. *0* ≤ *L2 i*
shows *gen-quote gam L x* = *gen-quote gam L1 x* + *gen-quote gam L2 x*
using *assms gen-token-add* **unfolding** *gen-quote-def* **by** *simp*

end

2.4 Gross quote token quantity into a pool

2.4.1 Function specialization

When the quote tokens are added to a pool, fees are to be taken into account: if a user adds a quantity q of tokens into a pool, the computation of the amount of base tokens received is based in $q \cdot (1 - \varphi)$.

definition *rng-quote-gross* **where**
rng-quote-gross P = *rng-gen-quote* (*grd P*) (*gross-fct* (*lq P*) (*fee P*))

lemma *rng-quote-gross-pos*:
assumes *0* ≤ *gross-fct* (*lq P*) (*fee P*) *i*
and *grd P i* ≤ *grd P* (*i*+*1*)
shows *0* ≤ *rng-quote-gross P pi i* **unfolding** *rng-quote-gross-def*
using *rng-gen-quote-pos assms* **by** *simp*

lemma *rng-quote-gross-continuous-on*:
shows *continuous-on A* (λ*pi*. *rng-quote-gross P pi i*)
unfolding *rng-quote-gross-def* **using** *rng-gen-quote-continuous-on* **by** *simp*

lemma *rng-quote-gross-mono*:
assumes *0* ≤ *gross-fct* (*lq P*) (*fee P*) *i*
and *grd P i* ≤ *grd P* (*i*+*1*)
shows *mono* (λ*pi*. *rng-quote-gross P pi i*) **unfolding** *rng-quote-gross-def*
using *rng-gen-quote-mono assms* **by** *simp*

definition *quote-gross* **where**
quote-gross P = *gen-quote* (*grd P*) (*gross-fct* (*lq P*) (*fee P*))

lemma *quote-gross-pos*:
assumes ∀ *i*. *0* ≤ *gross-fct* (*lq P*) (*fee P*) *i*
and ∀ *i*. *grd P i* ≤ *grd P* (*i*+*1*)
shows *0* ≤ *quote-gross P x* **unfolding** *quote-gross-def*

using *gen-quote-pos assms* **by** *simp*

lemma *quote-gross-mono*:
assumes $\forall\, i.\ 0 \leq (lq\ P)\ i$
and $\forall\, i.\ (fee\ P)\ i < 1$
and $\forall\, i.\ (grd\ P)\ i \leq (grd\ P)\ (i+1)$
and $\forall\, x.$ *rng-token* (*gamma-min-diff* (*grd P*)) (*gross-fct* (*lq P*) (*fee P*)) *x*
summable-on UNIV
shows *mono* (*quote-gross P*) **unfolding** *quote-gross-def*
proof (*rule gen-quote-mono*)
show $\forall\, i.\ 0 \leq$ *gross-fct* (*lq P*) (*fee P*) *i* **using** *gross-fct-sgn assms* **by** *blast*
qed (*simp add*: *assms*)+

lemma *gen-quote-grd-min*:
assumes *mono* (*grd P*)
and *finite* (*nz-support L*)
and *nz-support L* $\neq \{\}$
and *nz-support L* = *nz-support* (*lq P*)
shows *gen-quote* (*grd P*) *L* (*grd-min P*) = *0*
proof (*rule gen-quote-zero*)
fix *i*
assume *grd P i* < *grd-min P*
hence *i* < *idx-min* (*lq P*) **unfolding** *grd-min-def idx-min-img-def*
using *assms*(*1*) *mono-strict-invE* **by** *blast*
hence *lq P i* = *0* **using** *assms idx-min-finite-lt* **by** *auto*
hence *i* $\notin$ *nz-support* (*lq P*) **unfolding** *nz-support-def* **by** *auto*
thus *L i* = *0* **using** *assms nz-support-def* **by** *fastforce*
qed (*simp add*: *assms*)

Definition of the grid point that is reached in a pool for a given gross quantity of quote tokens.

definition *quote-reach* **where**
quote-reach = ($\lambda P\ y.$
if y = *0 then* (*grd-min P*)
else Inf ((*quote-gross P*)−‘ {*y*}))

2.4.2 Restriction to pools with a finite liquidity

context *finite-liq-pool*

begin

lemma *quote-gross-mono-finite*:
assumes $\forall\, i.\ 0 \leq (lq\ P)\ i$
and $\forall\, i.\ (fee\ P)\ i < 1$
and $\forall\, i.\ (grd\ P)\ i \leq (grd\ P)\ (i+1)$
shows *mono* (*quote-gross P*)
proof (*rule quote-gross-mono*)
show $\forall\, x.$ *rng-token* (*gamma-min-diff* (*grd P*)) (*gross-fct* (*lq P*) (*fee P*)) *x*

```
    summable-on UNIV
  proof
    fix x
    show rng-token (gamma-min-diff (grd P)) (gross-fct (lq P) (fee P)) x
      summable-on UNIV
    proof (rule finite-nz-support.finite-nonzero-summable)
      show finite-nz-support (gross-fct (lq P) (fee P))
        using assms finite-liq-gross-fct
        by (simp add: finite-nz-support.intro nz-support-def)
    qed
  qed
qed (simp add: assms)+

lemma quote-gross-mono-finite':
  assumes ∀ i. 0 ≤ (lq P) i
  and ∀ i. (fee P) i < 1
  and mono (grd P)
shows mono (quote-gross P)
proof (rule quote-gross-mono-finite)
  show ∀ i. grd P i ≤ grd P (i+1) using assms monoD by fastforce
qed (simp add: assms)+

lemma quote-gross-continuous:
  shows continuous-on A (quote-gross P) unfolding quote-gross-def
  using gen-quote-continuous finite-liq-gross-fct fin-liq finite-liqD
  by (simp add: finite-nz-support.gen-quote-continuous finite-nz-support.intro
      nz-support-def)

lemma quote-gross-IVT:
  assumes ∀ i. fee P i ≠ 1
  and grd-min P ≤ grd-max P
  and quote-gross P (grd-min P) ≤ y
  and y ≤ quote-gross P (grd-max P)
shows ∃ pi ≥ (grd-min P). pi ≤ (grd-max P) ∧
  quote-gross P pi = y
proof −
  have grd-min P = idx-min-img (grd P) (gross-fct (lq P) (fee P))
    by (simp add: assms gross-fct-nz-eq idx-min-img-eq grd-min-def)
  moreover have grd-max P = idx-max-img (grd P) (gross-fct (lq P) (fee P))
    by (simp add: assms gross-fct-nz-eq idx-max-img-eq grd-max-def)
  ultimately show ?thesis
    using gen-quote-IVT finite-liq-gross-fct assms unfolding quote-gross-def
    by (metis finite-nz-support.gen-quote-IVT finite-nz-support.intro
        nz-support-def)
qed

lemma quote-gross-ne:
  assumes ∀ i. fee P i ≠ 1
  and grd-min P ≤ grd-max P
```

and *quote-gross P* (*grd-min P*) ≤ *y*
and *y* ≤ *quote-gross P* (*grd-max P*)
shows *quote-gross P*−‘ {*y*} ≠ {} **using** *quote-gross-IVT assms* **by** *blast*

lemma *quote-gross-grd-min*:
assumes *mono* (*grd P*)
shows *quote-gross P* (*grd-min P*) = *0*
using *gen-quote-grd-min* **unfolding** *quote-gross-def*
by (*smt* (*verit*) *assms*(*1*) *fin-nz-sup gen-quote-zero gross-fct-zero-if*
idx-min-finite-le idx-min-img-def monoD grd-min-def)

lemma *quote-reach-mem*:
assumes ∀ *i*. *0* ≤ *lq P i*
and ∀ *i*. *fee P i* < *1*
and *mono* (*grd P*)
and *0* ≤ *y*
and *y* ≤ *quote-gross P* (*grd-max P*)
shows *quote-reach P y* ∈ *quote-gross P*−‘ {*y*}
proof (*cases y* = *0*)
case *True*
then show *?thesis*
using *quote-gross-grd-min assms* **unfolding** *quote-reach-def* **by** *simp*
next
case *False*
hence *quote-reach P y* = *Inf* ((*quote-gross P*)−‘ {*y*})
unfolding *quote-reach-def* **by** *simp*
also have ... ∈ (*quote-gross P*)−‘ {*y*}
proof (*rule closed-contains-Inf*)
define *X* **where** *X* = (*quote-gross P*)−‘ {*y*}
show *X* ≠ {}
using *quote-gross-grd-min quote-gross-ne assms* **unfolding** *X-def*
by (*smt* (*verit*) *False mono-invE quote-gross-mono-finite′*)
show *closed* ((*quote-gross P*) −‘ {*y*})
proof (*rule continuous-closed-vimage*)
show *closed* {*y*} **by** *simp*
show ⋀*x*. *isCont* (*quote-gross P*) *x* **using** *quote-gross-continuous assms*
by (*simp add*: *continuous-on-eq-continuous-within*)
qed
show *bdd-below* ((*quote-gross P*) −‘ {*y*})
proof
fix *x*
assume *x* ∈ (*quote-gross P*) −‘ {*y*}
hence *quote-gross P x* = *y* **by** *simp*
hence *quote-gross P* (*grd-min P*) < *quote-gross P x*
using *quote-gross-grd-min assms False* **by** *simp*
moreover have *mono* (*quote-gross P*)
proof (*rule quote-gross-mono-finite*)
show ∀ *i*. *grd P i* ≤ *grd P* (*i* + *1*) **using** *assms*(*3*) *monoD* **by** *fastforce*
qed (*simp add*: *assms*)+

```
    ultimately show (grd-min P) ≤ x using mono-invE assms by auto
   qed
  qed
  finally show ?thesis .
qed

lemma quote-gross-inv-strict-mono:
  assumes mono (quote-gross P)
  and quote-gross P sqp' < y
  and sqp ∈ quote-gross P -` {y}
shows sqp' < sqp
proof (rule ccontr)
  assume asm: ¬ sqp' < sqp
  have quote-gross P sqp' < y using assms by simp
  also have ... = quote-gross P sqp using assms by simp
  also have ... ≤ quote-gross P sqp' using asm assms mono-strict-invE by auto
  finally show False using assms by simp
qed

lemma quote-gross-inv-bounded:
  assumes mono (quote-gross P)
  and quote-gross P (grd-min P) < y
  and y < quote-gross P (grd-max P)
shows ∀ sqp' ∈ quote-gross P -` {y}.
  dist (grd-min P) sqp' ≤ grd-max P - grd-min P
proof
  fix sqp'
  assume sqp' ∈ quote-gross P -` {y}
  hence grd-min P ≤ sqp' using quote-gross-inv-strict-mono assms by fastforce
  have sqp' ≤ grd-max P
    using quote-gross-inv-strict-mono assms ‹sqp' ∈ quote-gross P -` {y}›
    by fastforce
  have dist (grd-min P) sqp' = sqp' - (grd-min P) using ‹grd-min P ≤ sqp'›
    by (simp add: dist-real-def)
  thus dist (grd-min P) sqp' ≤ grd-max P - grd-min P
    by (simp add: ‹sqp' ≤ grd-max P›)
qed

lemma quote-gross-bdd-below:
  assumes mono (quote-gross P)
  and quote-gross P (grd-min P) < y
  shows bdd-below (quote-gross P -`{y}) using assms
  by (metis bdd-below.I mono-strict-invE order-less-imp-le vimage-singleton-eq)

lemma quote-reach-le:
  assumes ∀ i. 0 ≤ lq P i
  and ∀ i. fee P i < 1
  and mono (grd P)
  and 0 < y
```

and *sqp* ∈ *quote-gross P* −‘{*y*}
shows *quote-reach P y* ≤ *sqp*
proof −
define *sqp′* **where** *sqp′* = *quote-reach P y*
define *X* **where** *X* = *quote-gross P* −‘ {*y*}
hence *sqp′* = *Inf X*
using *assms* **unfolding** *sqp′-def quote-reach-def* **by** *simp*
have ∀ *x*∈ *X*. *Inf X* ≤ *x*
proof
fix *x*
assume *x*∈ *X*
show *Inf X* ≤ *x*
proof (*rule cInf-lower*)
show *x*∈ *X* **using** ‹*x*∈ *X*› **.**
show *bdd-below X* **using** *assms quote-gross-bdd-below quote-gross-grd-min X-def*
by (*simp add*: *quote-gross-mono-finite′*)
qed
qed
thus *?thesis* **using** *assms X-def* ‹*sqp′* = *Inf X*› *sqp′-def* **by** *auto*
qed

lemma *quote-reach-gross-le*:
assumes ∀ *i*. 0 ≤ *lq P i*
and ∀ *i*. *fee P i* < 1
and *mono* (*grd P*)
and *grd-min P* ≤ *sqp*
shows *quote-reach P* (*quote-gross P sqp*) ≤ *sqp*
proof (*cases quote-gross P sqp* = 0)
case *True*
then show *?thesis* **using** *assms*(4) *quote-reach-def* **by** *presburger*
next
case *False*
then show *?thesis* **using** *quote-reach-le assms*
by (*metis mono-invE nle-le order-le-imp-less-or-eq quote-gross-grd-min quote-gross-mono-finite′ vimage-singleton-eq*)
qed

lemma *quote-gross-reach-eq*:
assumes ∀ *i*. 0 ≤ *lq P i*
and ∀ *i*. *fee P i* < 1
and *mono* (*grd P*)
and 0 ≤ *y*
and *y* ≤ *quote-gross P* (*grd-max P*)
shows *quote-gross P* (*quote-reach P y*) = *y*
using *assms quote-reach-mem* **by** *simp*

lemma *quote-reach-ge*:

assumes $\forall\, i.\ 0 \leq lq\ P\ i$
and $\forall\, i.\ fee\ P\ i < 1$
and *mono* (*grd P*)
and *grd-min P* $\leq$ *grd-max P*
and $0 < y$
and $y \leq$ *quote-gross P* (*grd-max P*)
shows *grd-min P* $\leq$ *quote-reach P y*
proof (*rule ccontr*)
assume $\neg$ *grd-min P* $\leq$ *quote-reach P y*
hence *quote-gross P* (*quote-reach P y*) $\leq$ *quote-gross P* (*grd-min P*)
by (*smt* (*verit*) *assms quote-gross-inv-strict-mono quote-gross-mono-finite′ quote-gross-grd-min quote-reach-mem*)
hence $y \leq$ *quote-gross P* (*grd-min P*) **using** *assms quote-gross-reach-eq* **by** *simp*
thus *False* **using** *assms*
by (*simp add*: *quote-gross-grd-min*)
qed

end

2.5 Net quote token quantity in a pool

2.5.1 Function specialization

There are no fees to take into account when tokens are withdrawn from a pool.

definition *rng-quote-net* **where**
rng-quote-net P = *rng-gen-quote* (*grd P*) (*lq P*)

lemma *rng-quote-net-pos*:
assumes $0 \leq$ (*lq P*) *i*
and *grd P i* $\leq$ *grd P* (*i*+1)
shows $0 \leq$ *rng-quote-net P x i* **unfolding** *rng-quote-net-def*
using *rng-gen-quote-pos assms* **by** *simp*

lemma *rng-quote-net-continuous-on*:
shows *continuous-on A* (λ*pi*. *rng-quote-net P pi i*)
unfolding *rng-quote-net-def* **using** *rng-gen-quote-continuous-on* **by** *simp*

lemma *rng-quote-net-mono*:
assumes $0 \leq$ (*lq P*) *i*
and *grd P i* $\leq$ *grd P* (*i*+1)
shows *mono* (λ*pi*. *rng-quote-net P pi i*) **unfolding** *rng-quote-net-def*
using *rng-gen-quote-mono assms* **by** *simp*

definition *quote-net* **where**
quote-net P = *gen-quote* (*grd P*) (*lq P*)

lemma *quote-net-pos*:
assumes $\forall\, i.\ 0 \leq$ (*lq P*) *i*

and $\forall i.\ grd\ P\ i \leq grd\ P\ (i+1)$
shows $0 \leq quote\text{-}net\ P\ x$ **unfolding** *quote-net-def*
using *gen-quote-pos assms* **by** *simp*

lemma *quote-net-mono*:
assumes $\forall i.\ 0 \leq (lq\ P)\ i$
and $\forall i.\ (fee\ P)\ i < 1$
and $\forall i.\ (grd\ P)\ i \leq (grd\ P)\ (i+1)$
and $\forall x.$ *rng-token* $(gamma\text{-}min\text{-}diff\ (grd\ P))\ (lq\ P)\ x$ *summable-on UNIV*
shows *mono* $(quote\text{-}net\ P)$ **unfolding** *quote-net-def*
using *gen-quote-mono assms* **by** *simp*

2.5.2 Restriction to pools with a finite liquidity

context *finite-liq-pool*
begin

lemma *quote-net-continuous*:
shows *continuous-on A* $(quote\text{-}net\ P)$ **unfolding** *quote-net-def*
using *gen-quote-continuous finite-liqD* **by** *simp*

lemma *quote-net-IVT*:
assumes $\forall i.$ *fee P i* $\neq 1$
and *grd-min P* $\leq$ *grd-max P*
and *quote-net P* $(grd\text{-}min\ P) \leq y$
and $y \leq quote\text{-}net\ P\ (grd\text{-}max\ P)$
shows $\exists pi \geq (grd\text{-}min\ P).\ pi \leq (grd\text{-}max\ P) \wedge$
quote-net P pi $= y$
using *gen-quote-IVT assms finite-liqD*
unfolding *quote-net-def grd-min-def grd-max-def* **by** *simp*

lemma *quote-net-ne*:
assumes $\forall i.$ *fee P i* $\neq 1$
and *grd-min P* $\leq$ *grd-max P*
and *quote-net P* $(grd\text{-}min\ P) \leq y$
and $y \leq quote\text{-}net\ P\ (grd\text{-}max\ P)$
shows *quote-net P* $-` \{y\} \neq \{\}$ **using** *quote-net-IVT assms* **by** *blast*

lemma *quote-net-mono-finite-liq*:
assumes $\forall i.\ 0 \leq (lq\ P)\ i$
and $\forall i.\ (fee\ P)\ i < 1$
and $\forall i.\ (grd\ P)\ i \leq (grd\ P)\ (i+1)$
shows *mono* $(quote\text{-}net\ P)$ **unfolding** *quote-net-def*
using *gen-quote-mono-finite finite-liqD assms* **by** *simp*

end

2.6 Gross and net quantities of base tokens

2.6.1 Generic functions for base tokens

definition *inv-gamma-max-diff* **where**
inv-gamma-max-diff = (λ*gamma* (*pi*::*real*) *i*. *inverse* (*max pi* (*gamma i*)) −
inverse (*max pi* (*gamma* (*i*+(*1*::*int*)))))

lemma *inv-max-pos*:
assumes *0* < *a*
and (*a*::*real*) ≤ *b*
shows *0* ≤ *inverse* (*max x a*) − *inverse* (*max x b*)
proof (*cases b* ≤ *x*)
case *True*
thus *?thesis* **using** *assms* **by** *auto*
next
case *False*
hence *max x b* = *b* **by** *simp*
show *?thesis*
proof (*cases a* ≤ *x*)
case *True*
hence *max x a* = *x* **by** *simp*
then show *?thesis* **using** ‹*max x b* = *b*› *False* **using** *assms* **by** *fastforce*
next
case *False*
hence *max x a* = *a* **by** *simp*
then show *?thesis*
by (*simp add*: ‹*max x b* = *b*› *assms le-imp-inverse-le*)
qed
qed

lemma *inv-gamma-max-diff-pos*:
assumes *gamma i* ≤ *gamma* (*i* +(*1*::*int*))
and *0* < *gamma i*
shows *0* ≤ *inv-gamma-max-diff gamma x i* **unfolding** *inv-gamma-max-diff-def*
by (*rule inv-max-pos*, (*simp add*: *assms*)+)

lemma *inv-gamma-max-diff-continuous*:
assumes *gamma i* ≤ *gamma* (*i* +(*1*::*int*))
and *0* < *gamma i*
shows *continuous-on A* (λ*pi*. *inv-gamma-max-diff gamma pi i*)
unfolding *inv-gamma-max-diff-def*
proof (*rule continuous-on-diff*)
show *continuous-on A* (λ*x*. *inverse* (*max x* (*gamma i*)))
proof (*rule continuous-on-inverse*)
show *continuous-on A* (λ*x*. *max x* (*gamma i*)) **using** *continuous-on-max*
continuous-on-const continuous-on-id **by** *blast*
show ∀ *x*∈*A*. *max x* (*gamma i*) ≠ *0* **using** *assms*
by (*metis leD max.cobounded2*)
qed

show *continuous-on A* (*λx. inverse* (*max x* (*gamma* (*i+1*))))
proof (*rule continuous-on-inverse*)
show *continuous-on A* (*λx. max x* (*gamma* (*i+1*))) **using** *continuous-on-max continuous-on-const continuous-on-id* **by** *blast*
show *∀ x∈A. max x* (*gamma* (*i+1*)) *≠ 0* **using** *assms* **by** *force*
qed
qed

lemma *inv-gamma-max-diff-antimono*:
assumes *gamma i ≤ gamma* (*i +*(*1*::*int*))
and *0 < gamma i*
shows *antimono* (*λpi. inv-gamma-max-diff gamma pi i*)
unfolding *inv-gamma-max-diff-def*
proof
fix *x y*::*real*
assume *asm*: *x ≤ y*
show *inverse* (*max y* (*gamma i*)) *− inverse* (*max y* (*gamma* (*i + 1*))) *≤*
inverse (*max x* (*gamma i*)) *− inverse* (*max x* (*gamma* (*i + 1*)))
proof (*rule diff-inv-max-le*)
show *x ≤ y* **using** *asm* **.**
show *gamma i ≤ gamma* (*i + 1*) **using** *assms* **by** *simp*
show *0 < gamma i* **using** *assms* **by** *simp*
qed
qed

definition *rng-gen-base* **where**
rng-gen-base =
(*λgamma L pi i. rng-token* (*inv-gamma-max-diff gamma*) *L pi i*)

lemma *rng-gen-base-pos*:
assumes *gamma i ≤ gamma* (*i +*(*1*::*int*))
and *0 < gamma i*
and *0 ≤ L i*
shows *0 ≤ rng-gen-base gamma L x i* **unfolding** *rng-gen-base-def*
by (*rule rng-token-pos*, *auto simp add*: *assms inv-gamma-max-diff-pos*)

lemma *rng-gen-base-continuous-on*:
assumes *gamma i ≤ gamma* (*i +*(*1*::*int*))
and *0 < gamma i*
shows *continuous-on A* (*λpi. rng-gen-base gamma L pi i*) **unfolding** *rng-gen-base-def*
by (*rule rng-token-continuous-on*,
simp add: *inv-gamma-max-diff-continuous assms*)

lemma *rng-gen-base-antimono*:
assumes *gamma i ≤ gamma* (*i +*(*1*::*int*))
and *0 < gamma i*
and *0 ≤ L i*
shows *antimono* (*λpi. rng-gen-base gamma L pi i*)
proof

```
  fix x y::real
  assume asm: x ≤ y
  show rng-gen-base gamma L y i ≤ rng-gen-base gamma L x i
    unfolding rng-gen-base-def rng-token-def
  proof (rule ordered-comm-semiring-class.comm-mult-left-mono)
    show 0 ≤ L i using assms by simp
    show inv-gamma-max-diff gamma y i ≤ inv-gamma-max-diff gamma x i
   using inv-gamma-max-diff-antimono[of gamma] asm antimonoD assms by auto

  qed
qed

definition gen-base where
gen-base = (λgamma L pi. gen-token (inv-gamma-max-diff gamma) L pi)

lemma gen-base-pos:
  assumes ∀ i. gamma i ≤ gamma (i +(1::int))
  and ∀ i. 0 < gamma i
  and ∀ i. 0 ≤ L i
  shows 0 ≤ gen-base gamma L x unfolding gen-base-def
  using gen-token-pos inv-gamma-max-diff-pos assms by simp

lemma gen-base-antimono:
  assumes ∀ x. rng-token (inv-gamma-max-diff gamma) L x summable-on UNIV
  and ∀ i. gamma i ≤ gamma (i +(1::int))
  and ∀ i. 0 < gamma i
  and ∀ i. 0 ≤ L i
  shows antimono (gen-base gamma L) using gen-token-antimono assms
    inv-gamma-max-diff-antimono
  by (simp add: gen-base-def)

lemma gen-base-zero:
  assumes mono gamma
  and ⋀i. sqp < gamma (i+1) ⟹ L i = 0
  shows gen-base gamma L sqp = 0 unfolding gen-base-def gen-token-def
proof (rule infsum-0)
  fix i
  show rng-token (inv-gamma-max-diff gamma) L sqp i = 0
  proof (cases gamma (i+1) ≤ sqp)
    case True
    hence gamma i ≤ sqp by (smt (verit) assms(1) mono-invE)
    hence inv-gamma-max-diff gamma sqp i = 0
      using True unfolding inv-gamma-max-diff-def by simp
    then show ?thesis unfolding rng-token-def by simp
  next
    case False
    hence L i = 0 using assms by simp
    then show ?thesis unfolding rng-token-def by simp
  qed
```

qed

lemma *gen-base-grd-max*:
assumes *mono* (*grd P*)
and *finite* (*nz-support L*)
and *nz-support L* ≠ {}
and *nz-support L* = *nz-support* (*lq P*)
shows *gen-base* (*grd P*) *L* (*grd-max P*) = *0*
proof (*rule gen-base-zero*)
fix *i*
assume *grd-max P* < *grd P* (*i* + *1*)
hence *idx-max* (*lq P*) < *i* **unfolding** *grd-max-def idx-max-img-def*
using *assms*(*1*)*mono-strict-invE* **by** *fastforce*
hence *lq P i* = *0* **using** *assms idx-max-finite-gt* **by** *auto*
hence *i* ∉ *nz-support* (*lq P*) **unfolding** *nz-support-def* **by** *auto*
thus *L i* = *0* **using** *assms nz-support-def* **by** *fastforce*
qed (*simp add*: *assms*)

2.6.2 Finite support restriction

context *finite-nz-support*
begin

lemma *gen-base-continuous*:
assumes ∀ *i*. *gamma i* ≤ *gamma* (*i* +(*1*::*int*))
and ∀ *i*. *0* < *gamma i*
shows *continuous-on A* (*gen-base gamma L*) **unfolding** *gen-base-def*
using *gen-token-continuous inv-gamma-max-diff-continuous assms* **by** *simp*

lemma *gen-base-IVT*:
assumes ∀ *i*. *gamma i* ≤ *gamma* (*i* +(*1*::*int*))
and ∀ *i*. *0* < *gamma i*
and (*idx-min-img gamma L*) ≤ (*idx-max-img gamma L*)
and *gen-base gamma L* (*idx-max-img gamma L*) ≤ *y*
and *y* ≤ *gen-base gamma L* (*idx-min-img gamma L*)
shows ∃ *pi* ≥ (*idx-min-img gamma L*). *pi* ≤ (*idx-max-img gamma L*) ∧
gen-base gamma L pi = *y*
proof (*rule IVT2*)
show ∀ *pi*. *idx-min-img gamma L* ≤ *pi* ∧ *pi* ≤ *idx-max-img gamma L* ⟶
isCont (*gen-base gamma L*) *pi*
proof (*intro allI impI*)
fix *pi*
assume (*idx-min-img gamma L*) ≤ *pi* ∧ *pi* ≤ (*idx-max-img gamma L*)
show *isCont* (*gen-base gamma L*) *pi* **using** *gen-base-continuous assms*
by (*simp add*: *continuous-on-eq-continuous-within*)
qed
qed (*simp add*: *assms*)+

lemma *gen-base-ne*:

assumes $\forall i.\ gamma\ i \leq gamma\ (i + (1::int))$
and $\forall i.\ 0 < gamma\ i$
and $(idx\text{-}min\text{-}img\ gamma\ L) \leq (idx\text{-}max\text{-}img\ gamma\ L)$
and $gen\text{-}base\ gamma\ L\ (idx\text{-}max\text{-}img\ gamma\ L) \leq y$
and $y \leq gen\text{-}base\ gamma\ L\ (idx\text{-}min\text{-}img\ gamma\ L)$
shows $(gen\text{-}base\ gamma\ L) - `\ \{y\} \neq \{\}$ **using** *gen-base-IVT assms* **by** *blast*

lemma *gen-base-antimono-finite*:
assumes $\forall i.\ gamma\ i \leq gamma\ (i + (1::int))$
and $\forall i.\ 0 < gamma\ i$
and $\forall i.\ 0 \leq L\ i$
shows *antimono* (*gen-base gamma L*)
proof (*rule gen-base-antimono*)
show $\forall x.$ *rng-token* (*inv-gamma-max-diff gamma*) *L x summable-on UNIV*
using *finite-nonzero-summable assms* **by** *simp*
qed (*simp add*: *assms*)+

lemma *gen-base-gross*:
assumes $\forall i.\ L\ i = L1\ i + L2\ i$
and $\forall i.\ 0 \leq L1\ i$
and $\forall i.\ 0 \leq L2\ i$
shows *gen-base gam L x = gen-base gam L1 x + gen-base gam L2 x*
using *assms gen-token-add* **unfolding** *gen-base-def* **by** *simp*

end

2.7 Gross base token quantity in a pool

2.7.1 Function specialization

definition *rng-base-gross* **where**
rng-base-gross P = rng-gen-base (*grd P*) (*gross-fct* (*lq P*) (*fee P*))

lemma *rng-base-gross-pos*:
assumes $0 \leq gross\text{-}fct\ (lq\ P)\ (fee\ P)\ i$
and $grd\ P\ i \leq grd\ P\ (i+1)$
and $0 < grd\ P\ i$
shows $0 \leq rng\text{-}base\text{-}gross\ P\ x\ i$ **unfolding** *rng-base-gross-def*
using *rng-gen-base-pos assms* **by** *simp*

lemma *rng-base-gross-continuous-on*:
assumes $grd\ P\ i \leq grd\ P\ (i+1)$
and $0 < grd\ P\ i$
shows *continuous-on A* ($\lambda pi.$ *rng-base-gross P pi i*)
unfolding *rng-base-gross-def*
using *rng-gen-base-continuous-on assms* **by** *simp*

lemma *rng-base-gross-mono*:
assumes $0 \leq gross\text{-}fct\ (lq\ P)\ (fee\ P)\ i$
and $grd\ P\ i \leq grd\ P\ (i+1)$

and *0 < grd P i*
shows *antimono* (*λpi. rng-base-gross P pi i*) **unfolding** *rng-base-gross-def*
using *rng-gen-base-antimono assms* **by** *simp*

definition *base-gross* **where**
base-gross P = gen-base (*grd P*) (*gross-fct* (*lq P*) (*fee P*))

lemma *base-gross-pos*:
assumes ∀ *i. 0 ≤ gross-fct* (*lq P*) (*fee P*) *i*
and ∀ *i. grd P i ≤ grd P* (*i+1*)
and ∀ *i. 0 < grd P i*
shows *0 ≤ base-gross P x* **unfolding** *base-gross-def*
using *gen-base-pos assms* **by** *simp*

lemma *base-gross-antimono*:
assumes ∀ *i. 0 ≤* (*lq P*) *i*
and ∀ *i.* (*fee P*) *i < 1*
and ∀ *i.* (*grd P*) *i ≤* (*grd P*) (*i+1*)
and ∀ *i. 0 < grd P i*
and ∀ *x. rng-token* (*inv-gamma-max-diff* (*grd P*)) (*gross-fct* (*lq P*) (*fee P*)) *x*
summable-on UNIV
shows *antimono* (*base-gross P*) **unfolding** *base-gross-def*
proof (*rule gen-base-antimono*)
show ∀ *i. 0 ≤ gross-fct* (*lq P*) (*fee P*) *i* **using** *gross-fct-sgn assms* **by** *blast*
qed (*simp add*: *assms*)+

lemma *base-gross-grd-max*:
assumes *mono* (*grd P*)
and *finite* (*nz-support* (*lq P*))
shows *base-gross P* (*grd-max P*) = *0*
using *gen-base-grd-max assms gen-base-zero gross-fct-zero-if*
idx-max-finite-ge idx-max-img-def monoD grd-max-def
unfolding *quote-gross-def*
by (*smt* (*z3*) *base-gross-def*)

definition *base-reach* **where**
base-reach = (*λP y.*
if y = 0
then (*grd-max P*)
else Sup ((*base-gross P*)−‘ {*y*}))

2.7.2 Restriction to pools with a finite liquidity

context *finite-liq-pool*
begin

lemma *base-gross-continuous*:
assumes ∀ *i. grd P i ≤ grd P* (*i+1*)
and ∀ *i. 0 < grd P i*

shows *continuous-on A* (*base-gross P*) **unfolding** *base-gross-def*
proof (*rule finite-nz-support.gen-base-continuous*)
show *finite-nz-support* (*gross-fct* (*lq P*) (*fee P*))
using *finite-liq-gross-fct*
by (*simp add*: *finite-nz-support.intro nz-support-def*)
qed (*simp add*: *assms*)+

lemma *base-gross-IVT*:
assumes ∀ *i*. *grd P i* ≤ *grd P* (*i*+*1*)
and ∀ *i*. *0* < *grd P i*
and ∀ *i*. *fee P i* ≠ *1*
and *grd-min P* ≤ *grd-max P*
and *base-gross P* (*grd-max P*) ≤ *y*
and *y* ≤ *base-gross P* (*grd-min P*)
shows ∃ *pi* ≥ (*grd-min P*). *pi* ≤ (*grd-max P*) ∧
base-gross P pi = *y*
proof −
define *L′* **where** *L′* = *gross-fct* (*lq P*) (*fee P*)
have *1*: *grd-min P* = *idx-min-img* (*grd P*) *L′*
by (*simp add*: *assms gross-fct-nz-eq idx-min-img-eq grd-min-def L′-def*)
have *2*: *grd-max P* = *idx-max-img* (*grd P*) *L′*
by (*simp add*: *assms gross-fct-nz-eq idx-max-img-eq grd-max-def L′-def*)
have ∃ *pi*≥*idx-min-img* (*grd P*) *L′*.
pi ≤ *idx-max-img* (*grd P*) *L′* ∧ *gen-base* (*grd P*) *L′ pi* = *y*
proof (*rule finite-nz-support.gen-base-IVT*)
show *idx-min-img* (*grd P*) *L′* ≤ *idx-max-img* (*grd P*) *L′* **using** *1 2 assms* **by** *simp*
show *gen-base* (*grd P*) *L′* (*idx-max-img* (*grd P*) *L′*) ≤ *y* **using** *2 assms*
by (*simp add*: *L′-def base-gross-def*)
show *y* ≤ *gen-base* (*grd P*) *L′* (*idx-min-img* (*grd P*) *L′*) **using** *1 assms*
by (*simp add*: *L′-def base-gross-def*)
show *finite-nz-support L′* **using** *L′-def finite-liq-gross-fct*
by (*simp add*: *finite-nz-support.intro nz-support-def*)
qed (*simp add*: *assms*)+
thus *?thesis* **by** (*simp add*: *1 2 L′-def base-gross-def*)
qed

lemma *base-gross-ne*:
assumes ∀ *i*. *grd P i* ≤ *grd P* (*i*+*1*)
and ∀ *i*. *0* < *grd P i*
and ∀ *i*. *fee P i* ≠ *1*
and *grd-min P* ≤ *grd-max P*
and *base-gross P* (*grd-max P*) ≤ *y*
and *y* ≤ *base-gross P* (*grd-min P*)
shows *base-gross P*−‘ {*y*} ≠ {} **using** *base-gross-IVT assms* **by** *blast*

lemma *base-gross-antimono-finite*:
assumes ∀ *i*. *0* ≤ (*lq P*) *i*
and ∀ *i*. *grd P i* ≤ *grd P* (*i*+*1*)

```
  and ∀ i. 0 < grd P i
  and ∀ i. (fee P) i < 1
shows antimono (base-gross P) unfolding base-gross-def
proof (rule finite-nz-support.gen-base-antimono-finite)
  show ∀ i. 0 ≤ gross-fct (lq P) (fee P) i
    using gross-fct-sgn assms by blast
  show finite-nz-support (gross-fct (lq P) (fee P))
    by (simp add: finite-liq-gross-fct finite-nz-support-def nz-support-def)
qed (simp add: assms)+

lemma base-reach-mem:
  assumes ∀ i. grd P i ≤ grd P (i+1)
  and ∀ i. 0 < grd P i
  and ∀ i. fee P i < 1
  and ∀ i. 0 ≤ lq P i
  and mono (grd P)
  and grd-min P ≤ grd-max P
  and 0 ≤ y
  and y ≤ base-gross P (grd-min P)
shows base-reach P y ∈ base-gross P−‘ {y}
proof (cases y = 0)
  case True
  then show ?thesis unfolding base-reach-def
    by (simp add: assms(5) base-gross-grd-max fin-nz-sup)
next
  case False
  hence base-reach P y = Sup ((base-gross P)−‘ {y})
    unfolding base-reach-def by simp
  also have ... ∈ (base-gross P)−‘ {y}
  proof (rule closed-contains-Sup)
    have antimono (base-gross P) using base-gross-antimono-finite assms by simp
    define X where X = (base-gross P)−‘ {y}
    show X ≠ {} using base-gross-ne assms unfolding X-def
      by (metis add-cancel-left-right add-less-same-cancel1 base-gross-grd-max
          fin-nz-sup less-int-code(1) mono-strict-invE)
    show closed ((base-gross P) −‘ {y})
    proof (rule continuous-closed-vimage)
      show closed {y} by simp
      show ⋀x. isCont (base-gross P) x using base-gross-continuous
        by (simp add: continuous-on-eq-continuous-within assms)
    qed
    show bdd-above ((base-gross P) −‘ {y})
    proof
      fix x
      assume x ∈ (base-gross P) −‘ {y}
      hence base-gross P x = y by simp
      hence base-gross P (grd-max P) < base-gross P x
        using assms False
        by (simp add: base-gross-grd-max fin-nz-sup)
```

thus $x \leq (grd\text{-}max\ P)$ **using** ‹*antimono* (*base-gross P*)›
by (*metis antimonoD incseq-const less-eq-real-def less-irrefl-nat mono-strict-invE nle-le*)
qed
qed
finally show *?thesis* **.**
qed

lemma *base-gross-dwn*:
assumes $\forall\, i.\ grd\ P\ i \leq grd\ P\ (i+1)$
and $\forall\, i.\ 0 < grd\ P\ i$
and $\forall\, i.\ fee\ P\ i < 1$
and $\forall\, i.\ 0 \leq lq\ P\ i$
and *mono* (*grd P*)
and $grd\text{-}min\ P \leq grd\text{-}max\ P$
and $0 \leq y$
and $y \leq base\text{-}gross\ P\ (grd\text{-}min\ P)$
shows $base\text{-}gross\ P\ (base\text{-}reach\ P\ y) = y$
using *assms base-reach-mem* **by** *simp*

end

2.8 Net base token quantity in a pool

2.8.1 Function specialization

definition *rng-base-net* **where**
$rng\text{-}base\text{-}net\ P = rng\text{-}gen\text{-}base\ (grd\ P)\ (lq\ P)$

lemma *rng-base-net-pos*:
assumes $grd\ P\ i \leq grd\ P\ (i + (1::int))$
and $0 < grd\ P\ i$
and $0 \leq lq\ P\ i$
shows $0 \leq rng\text{-}base\text{-}net\ P\ x\ i$ **unfolding** *rng-base-net-def*
using *rng-gen-base-pos assms* **by** *simp*

lemma *rng-base-net-continuous-on*:
assumes $grd\ P\ i \leq grd\ P\ (i + (1::int))$
and $0 < grd\ P\ i$
shows *continuous-on A* $(\lambda pi.\ rng\text{-}base\text{-}net\ P\ pi\ i)$
unfolding *rng-base-net-def* **using** *rng-gen-base-continuous-on assms* **by** *simp*

lemma *rng-base-net-mono*:
assumes $grd\ P\ i \leq grd\ P\ (i + (1::int))$
and $0 < grd\ P\ i$
and $0 \leq lq\ P\ i$
shows *antimono* $(\lambda pi.\ rng\text{-}base\text{-}net\ P\ pi\ i)$ **unfolding** *rng-base-net-def*
using *rng-gen-base-antimono assms* **by** *simp*

definition *base-net* **where**

base-net P = gen-base (grd P) (lq P)

lemma *base-net-pos*:
assumes $\forall\, i.$ *grd P i* $\leq$ *grd P* $(i + (1::int))$
and $\forall\, i.$ $0 <$ *grd P i*
and $\forall\, i.$ $0 \leq$ *lq P i*
shows $0 \leq$ *base-net P x* **unfolding** *base-net-def*
using *gen-base-pos assms* **by** *simp*

2.8.2 Restriction to pools with a finite liquidity

context *finite-liq-pool*
begin

lemma *base-net-continuous*:
assumes $\forall\, i.$ *grd P i* $\leq$ *grd P* $(i + (1::int))$
and $\forall\, i.$ $0 <$ *grd P i*
shows *continuous-on A* (*base-net P*) **unfolding** *base-net-def*
using *gen-base-continuous assms finite-liqD* **by** *simp*

lemma *base-net-IVT*:
assumes $\forall\, i.$ *grd P i* $\leq$ *grd P* $(i + (1::int))$
and $\forall\, i.$ $0 <$ *grd P i*
and *grd-min P* $\leq$ *grd-max P*
and *base-net P* (*grd-max P*) $\leq y$
and $y \leq$ *base-net P* (*grd-min P*)
shows $\exists\, pi \geq$ (*grd-min P*). *pi* $\leq$ (*grd-max P*) $\wedge$
base-net P pi $= y$
using *gen-base-IVT assms finite-liqD*
unfolding *base-net-def grd-min-def grd-max-def* **by** *simp*

lemma *base-net-ne*:
assumes $\forall\, i.$ *grd P i* $\leq$ *grd P* $(i + (1::int))$
and $\forall\, i.$ $0 <$ *grd P i*
and *grd-min P* $\leq$ *grd-max P*
and *base-net P* (*grd-max P*) $\leq y$
and $y \leq$ *base-net P* (*grd-min P*)
shows *base-net P* $-$ ‘ $\{y\} \neq \{\}$ **using** *base-net-IVT assms* **by** *blast*

lemma *base-net-antimono-finite*:
assumes $\forall\, i.$ *grd P i* $\leq$ *grd P* $(i + (1::int))$
and $\forall\, i.$ $0 <$ *grd P i*
and $\forall\, i.$ $0 \leq$ *lq P i*
shows *antimono* (*base-net P*) **unfolding** *base-net-def*
using *gen-base-antimono-finite finite-liqD assms* **by** *simp*

end

2.9 Swapping tokens, market depth and slippage

Given a grid point π and a quantity y of quote tokens to add to the pool, this function computes the amount of base tokens that are retrieved from the pool.

definition *quote-swap* **where**
quote-swap P = (λ*pi y*.
  base-net P pi − *base-net P* (*quote-reach P* (*y* + *quote-gross P pi*)))

Given a grid point π and a quantity x of base tokens to add to the pool, this function computes the amount of quote tokens that are retrieved from the pool.

definition *base-swap* **where**
base-swap P = (λ*pi x*.
  quote-net P pi − *quote-net P* (*base-reach P* (*x* + *base-gross P pi*)))

The market depth in a pool takes as arguments two grid points π and π', and returns the amounts of base or quote tokens that have to be added to the pool for its state to get from π to π'.

definition *mkt-depth* **where**
mkt-depth P = (λ *pi pi'*. *if pi* < *pi' then* (*base-net P pi* − *base-net P pi'*)
  else (*quote-net P pi* − *quote-net P pi'*))

Base and quote slippages relate the amount of tokens withdrawn from the pool from those given by an infinitesimally small amount of tokens and that can be deduced from the grid point.

definition *quote-slippage* **where**
quote-slippage P = (λ*pi y*. *y*/(*quote-swap P pi y* ∗ *pi* ∗ *pi*) − *1*)

definition *base-slippage* **where**
base-slippage P = (λ*pi x*. *base-swap P pi x*/(*x* ∗ *pi* ∗ *pi*) − *1*)

2.10 Identical profiles

definition *id-grid-on* **where**
id-grid-on P P' I ⟷ (∀ *i*∈ *I*. *grd P i* = *grd P' i*)

lemma *id-grid-onI*[*intro*]:
  assumes ⋀*i*. *i*∈ *I* ⟹ *grd P i* = *grd P' i*
  shows *id-grid-on P P' I* **using** *assms* **unfolding** *id-grid-on-def* **by** *simp*

lemma *id-grid-onD*[*dest*]:
  assumes *id-grid-on P P' I*
  and *i*∈ *I*
shows *grd P i* = *grd P' i* **using** *assms* **unfolding** *id-grid-on-def* **by** *simp*

lemma *id-grid-on-comm*:

assumes *id-grid-on P P′ I*
shows *id-grid-on P′ P I*
using *assms* **unfolding** *id-grid-on-def* **by** *simp*

lemma *id-grid-on-mono*:
assumes *id-grid-on P P′ I*
and $I' \subseteq I$
shows *id-grid-on P P′ I′* **using** *assms* **unfolding** *id-grid-on-def* **by** *auto*

definition *same-nz-liq-on* **where**
same-nz-liq-on P P′ I $\longleftrightarrow$ *id-grid-on P P′ I* $\wedge$
$(\forall\, i \in I.\ (lq\ P\ i = 0) \longleftrightarrow (lq\ P'\ i = 0))$

lemma *same-nz-liq-onI*[*intro*]:
assumes *id-grid-on P P′ I*
and $\bigwedge i.\ i \in I \Longrightarrow ((lq\ P\ i = 0) \longleftrightarrow (lq\ P'\ i = 0))$
shows *same-nz-liq-on P P′ I* **using** *assms* **unfolding** *same-nz-liq-on-def* **by** *simp*

lemma *same-nz-liq-onD*[*dest*]:
assumes *same-nz-liq-on P P′ I*
and $i \in I$
shows *grd P i* = *grd P′ i* $(lq\ P\ i = 0) \longleftrightarrow (lq\ P'\ i = 0)$
using *assms* **unfolding** *same-nz-liq-on-def* **by** *auto*

lemma *same-nz-liq-on-comm*:
assumes *same-nz-liq-on P P′ I*
shows *same-nz-liq-on P′ P I*
using *assms id-grid-on-comm* **unfolding** *same-nz-liq-on-def* **by** *simp*

lemma *same-nz-liq-on-mono*:
assumes *same-nz-liq-on P P′ I*
and $I' \subseteq I$
shows *same-nz-liq-on P P′ I′*
using *assms id-grid-on-mono* **unfolding** *same-nz-liq-on-def*
by (*meson id-grid-on-comm in-mono*)

definition *fee-diff-on* **where**
fee-diff-on P P′ I $\longleftrightarrow$ *id-grid-on P P′ I* $\wedge$ $(\forall\, i \in I.\ lq\ P\ i = lq\ P'\ i)$

lemma *fee-diff-onI*[*intro*]:
assumes *id-grid-on P P′ I*
and $\bigwedge i.\ i \in I \Longrightarrow lq\ P\ i = lq\ P'\ i$
shows *fee-diff-on P P′ I*
using *assms* **unfolding** *fee-diff-on-def* **by** *simp*

lemma *fee-diff-onD*[*dest*]:
assumes *fee-diff-on P P′ I*
shows *id-grid-on P P′ I* $\forall\, i \in I.\ lq\ P\ i = lq\ P'\ i$
proof−

show *id-grid-on P P′ I*
using *assms* **unfolding** *fee-diff-on-def* **by** *simp*
show $\forall\, i{\in}I.\ lq\ P\ i = lq\ P'\ i$
using *assms* **unfolding** *fee-diff-on-def* **by** *simp*
qed

lemma *fee-diff-on-nz-liq*:
assumes *fee-diff-on P P′ I*
shows *same-nz-liq-on P P′ I* **unfolding** *same-nz-liq-on-def*
proof
show *id-grid-on P P′ I* **using** *assms fee-diff-onD*(*1*) **by** *simp*
show $\forall\, i{\in}I.\ (lq\ P\ i = 0) = (lq\ P'\ i = 0)$ **using** *assms fee-diff-onD*(*2*) **by** *simp*
qed

lemma *fee-diff-on-comm*:
assumes *fee-diff-on P P′ I*
shows *fee-diff-on P′ P I*
using *assms fee-diff-on-def id-grid-on-comm* **by** *simp*

lemma *fee-diff-on-mono*:
assumes *fee-diff-on P P′ I*
and $I'{\subseteq}\ I$
shows *fee-diff-on P P′ I′*
using *assms id-grid-on-mono* **unfolding** *fee-diff-on-def* **by** *blast*

3 Grid refinement

We define the notion of pool refinement, that characterizes when a pool admits a finer price grid than another one but exhibits the same behavior.

3.1 Encompassement properties

definition *encomp-at* **where**
encomp-at gamma1 gamma2 i k ≡ *gamma2 k* ≤ *gamma1 i* ∧
gamma1 (*i*+*1*) ≤ *gamma2* (*k*+*1*)

lemma *encomp-atD1*:
assumes *encomp-at gamma1 gamma2 i k*
shows *gamma2 k* ≤ *gamma1 i*
using *assms* **unfolding** *encomp-at-def* **by** *simp*

lemma *encomp-atD2*:
assumes *encomp-at gamma1 gamma2 i k*
shows *gamma1* (*i*+*1*) ≤ *gamma2* (*k*+*1*)
using *assms* **unfolding** *encomp-at-def* **by** *simp*

lemma *encomp-atI*[*intro*]:
assumes *gamma2 k* ≤ *gamma1 i*

and *gamma1 (i+1) ≤ gamma2 (k+1)*
shows *encomp-at gamma1 gamma2 i k* **using** *assms* **unfolding** *encomp-at-def* **by** *simp*

definition *encompassed* **where**
encompassed gamma1 gamma2 k = {i::int. encomp-at gamma1 gamma2 i k}

lemma *encompassed-convex*:
assumes *(i::int) ∈ encompassed gamma1 gamma2 k*
and *j ∈ encompassed gamma1 gamma2 k*
and *i ≤ l*
and *l ≤ j*
and *mono gamma1*
shows *l ∈ encompassed gamma1 gamma2 k* **unfolding** *encompassed-def encomp-at-def*
proof
have *gamma2 k ≤ gamma1 i* **using** *assms encompassed-def encomp-at-def* **by** *blast*
hence *gamma2 k ≤ gamma1 l* **using** *assms*
by *(meson dual-order.trans monotoneD)*
have *gamma1 (j+1) ≤ gamma2 (k+1)*
using *assms encompassed-def encomp-at-def* **by** *blast*
hence *gamma1 (l+1) ≤ gamma2 (k+1)*
using *assms dual-order.trans monoD* **by** *fastforce*
thus *gamma2 k ≤ gamma1 l ∧ gamma1 (l + 1) ≤ gamma2 (k + 1)*
using ‹*gamma2 k ≤ gamma1 l*› **by** *simp*
qed

lemma *encompassed-interval*:
assumes *mono gamma1*
and *finite (encompassed gamma1 gamma2 k)*
and *encompassed gamma1 gamma2 k ≠ {}*
shows *encompassed gamma1 gamma2 k =*
{Min (encompassed gamma1 gamma2 k).. Max (encompassed gamma1 gamma2 k)}
proof
define *E* **where** *E = (encompassed gamma1 gamma2 k)*
define *m* **where** *m = Min E*
define *M* **where** *M = Max E*
have *m∈ E* **using** *m-def E-def assms* **by** *simp*
have *M ∈ E* **using** *M-def E-def assms* **by** *simp*
show *{m..M} ⊆ E*
proof
fix *x*
assume *x∈ {m..M}*
hence *m ≤ x ∧ x ≤ M* **by** *simp*
show *x∈ E* **using** *assms encompassed-convex*
E-def ‹*M ∈ E*› ‹*m ∈ E*› ‹*m ≤ x ∧ x ≤ M*› **by** *blast*
qed
show *E ⊆ {m..M}* **using** *m-def M-def assms*

by (*simp add*: *E-def subset-eq*)
qed

lemma *encomp-at-idx-leq*:
fixes *gamma1*::*int* ⇒ *real* **and** *gamma2*::*int* ⇒ *real*
assumes *strict-mono* (*gamma1*::*int* ⇒ *real*)
and *mono* (*gamma2*::*int* ⇒ *real*)
and *encomp-at gamma1 gamma2 i k*
and *gamma2 k′* ≤ *gamma1 i*
shows *k′* ≤ *k*
proof (*rule ccontr*)
assume ¬ *k′* ≤ *k*
hence *k* < *k′* **by** *simp*
hence *k*+*1* ≤ *k′* **by** *simp*
hence *gamma2* (*k*+*1*) ≤ *gamma2 k′* **using** *assms*
by (*simp add*: *monotoneD*)
hence *gamma1* (*i*+*1*) ≤ *gamma2 k′* **using** *assms encomp-atD2* **by** *fastforce*
hence *gamma1* (*i*+*1*) ≤ *gamma1 i* **using** *assms* **by** *simp*
thus *False* **using** *assms*(*1*) **by** (*simp add*: *strict-mono-less-eq*)
qed

lemma *encomp-at-unique*:
assumes *strict-mono* (*gamma1*::*int* ⇒ *real*)
and *mono* (*gamma2*::*int* ⇒ *real*)
and *encomp-at gamma1 gamma2 i k*
and *encomp-at gamma1 gamma2 i k′*
shows *k* = *k′*
proof −
have *k* ≤ *k′* **using** *assms encomp-at-idx-leq*
by (*simp add*: *encomp-atD1*)
moreover have *k′* ≤ *k* **using** *assms encomp-at-idx-leq*
by (*simp add*: *encomp-atD1*)
ultimately show *?thesis* **by** *simp*
qed

lemma *encomp-at-unique′*:
assumes *strict-mono* (*gamma1*::*int* ⇒ *real*)
and *mono* (*gamma2*::*int* ⇒ *real*)
and *encomp-at gamma1 gamma2 i k*
and *gamma2 k′* ≤ *gamma1 i*
and *gamma1 i* < *gamma2* (*k′*+*1*)
shows *k* = *k′*
proof (*rule ccontr*)
assume *k*≠ *k′*
have *k′* ≤ *k* **using** *assms encomp-at-idx-leq* **by** *simp*
hence *k′* < *k* **using** ‹*k*≠ *k′*› **by** *simp*
hence *k′*+*1* ≤ *k* **by** *simp*
hence *gamma2* (*k′*+*1*) ≤ *gamma2 k* **using** *assms monoE* **by** *blast*
moreover have *gamma2 k* < *gamma2* (*k′*+*1*)

using *assms encomp-atD1* **by** *fastforce*
ultimately show *False* **by** *simp*
qed

lemma *encomp-at-refl*:
fixes *gamma*::*'a*::{*one*, *plus*}⇒ *real*
shows *encomp-at gamma gamma i i*
proof
show *gamma i* ≤ *gamma i* **by** *simp*
show *gamma* (*i*+*1*) ≤ *gamma* (*i*+*1*) **by** *simp*
qed

3.2 Finer price grids

definition *finer-range*:: (*int* ⇒ *real*) ⇒ (*int* ⇒ *real*) ⇒ *bool* **where**
finer-range gamma1 gamma2 ≡ (∀ *i*. ∃ *k*. *encomp-at gamma1 gamma2 i k*)

definition *finer-grid* **where**
finer-grid P1 P2 ≡ *finer-range* (*grd P1*) (*grd P2*)

lemma *finer-grid-range*[*simp*]:
assumes *finer-grid P1 P2*
shows *finer-range* (*grd P1*) (*grd P2*)
using *assms* **unfolding** *finer-grid-def* **by** *simp*

definition *coarse-idx* **where**
coarse-idx gamma1 gamma2 i =
(*THE k*. *encomp-at gamma1 gamma2 i k*)

definition *finer-idx-bound* **where**
finer-idx-bound gamma1 gamma2 i =
(*THE k*. *gamma1 k* = *gamma2* (*coarse-idx gamma1 gamma2 i*))

lemma *finer-range-refl*:
shows *finer-range gamma gamma* **using** *encomp-at-refl*
unfolding *finer-range-def* **by** *auto*

locale *finer-ranges* =
fixes *gamma1*::*int* ⇒ *real* **and** *gamma2*::*int* ⇒ *real*
assumes *stm*: *strict-mono gamma1*
and *mon*: *mono gamma2*
and *fin*: *finer-range gamma1 gamma2*

begin

lemma *encomp-idx-unique*:
shows ∃!*k*. *encomp-at gamma1 gamma2 i k*
proof −
have *ex*: ∃ *k*. *encomp-at gamma1 gamma2 i k*

```
  using stm mon fin unfolding finer-range-def by simp
 {
  fix k k'
  assume encomp-at gamma1 gamma2 i k'
   and encomp-at gamma1 gamma2 i k
  hence k = k' using encomp-at-unique stm mon fin by auto
 }
 thus ?thesis using ex by auto
qed

lemma coarse-idx-bounds:
shows encomp-at gamma1 gamma2 i (coarse-idx gamma1 gamma2 i)
proof −
 define P where P = (λk. encomp-at gamma1 gamma2 i k)
 have P (coarse-idx gamma1 gamma2 i) unfolding P-def coarse-idx-def
  by (metis (no-types, lifting) encomp-idx-unique the-equality)
 thus ?thesis using P-def by simp
qed

lemma encompassed-bounds:
 shows i ∈ encompassed gamma1 gamma2 (coarse-idx gamma1 gamma2 i)
 using fin coarse-idx-bounds unfolding encompassed-def by auto

lemma encompassed-unique:
 assumes i ∈ encompassed gamma1 gamma2 k
 shows k = coarse-idx gamma1 gamma2 i
 using assms coarse-idx-bounds encompassed-def encomp-idx-unique by blast

lemma encompassed-inj:
 assumes k≠ k'
 shows encompassed gamma1 gamma2 k ∩ encompassed gamma1 gamma2 k' =
{}
proof (rule ccontr)
 assume encompassed gamma1 gamma2 k ∩ encompassed gamma1 gamma2 k' ≠
{}
 hence ∃ i. i ∈ encompassed gamma1 gamma2 k ∩ encompassed gamma1 gamma2
k'
  by auto
 from this obtain i where i ∈ encompassed gamma1 gamma2 k and
  i ∈ encompassed gamma1 gamma2 k' by auto
 hence k = k' using encompassed-unique by auto
 thus False using assms by simp
qed

lemma coarse-idx-eq:
 assumes gamma2 k' ≤ gamma1 i
 and gamma1 i < gamma2 (k'+1)
shows k' = coarse-idx gamma1 gamma2 i
proof (rule ccontr)
```

```
  assume k' ≠ coarse-idx gamma1 gamma2 i
  define k where k = coarse-idx gamma1 gamma2 i
  have gam: encomp-at gamma1 gamma2 i k
    using k-def assms by (simp add: coarse-idx-bounds)
  hence k' ≤ k
    using stm mon fin assms encomp-at-idx-leq encomp-idx-unique by blast
  hence k' < k using ‹k'≠ coarse-idx gamma1 gamma2 i› k-def by simp
  hence k'+1 ≤ k by simp
  hence gamma2 (k'+1) ≤ gamma2 k using assms stm mon monoE by blast
  moreover have gamma2 k < gamma2 (k'+1)
    using assms gam ‹k' ≠ coarse-idx gamma1 gamma2 i› encomp-idx-unique
      k-def encomp-atD1 by fastforce
  ultimately show False by simp
qed

lemma coarse-idx-reached:
  assumes gamma1 m ≤ gamma2 k
  and gamma2 k < gamma1 M
  and k = coarse-idx gamma1 gamma2 i
shows ∃j. gamma1 j = gamma2 k
proof (rule ccontr)
  assume ¬(∃j. gamma1 j = gamma2 k)
  hence ∀j. gamma1 j ≠ gamma2 k by simp
  have gamma2 k ≤ gamma1 i using coarse-idx-bounds assms
    by (simp add: encomp-atD1)
  define X where X = gamma1‘{j. m ≤ j ∧ gamma1 j ≤ gamma2 k}
  have gamma1 m∈ X using assms X-def by simp
  hence X ≠ {} by auto
  have X ⊆ gamma1‘{m..M}
  proof
    fix x
    assume x∈ X
    hence ∃l. l∈ {j. m ≤ j ∧ gamma1 j ≤ gamma2 k} ∧ x  = gamma1 l
      unfolding X-def by auto
    from this obtain l where m ≤ l and gamma1 l ≤ gamma2 k
      and x = gamma1 l by auto
    hence l ≤ M using assms stm
      by (meson linorder-not-less nle-le order-trans strict-mono-less-eq)
    hence l ∈ {m..M} using ‹m ≤ l› by simp
    thus x ∈ gamma1 ‘ {m..M} using ‹x = gamma1 l› by simp
  qed
  hence finite X using finite-surj by blast
  hence Sup X ∈ X
    by (metis ‹X ≠ {}› infinite-growing le-cSup-finite less-cSupD nless-le)
  hence ∃l. l∈ {j. m ≤ j ∧ gamma1 j ≤ gamma2 k} ∧ Sup X  = gamma1 l
    unfolding X-def by auto
  from this obtain l where m ≤ l and gamma1 l ≤ gamma2 k
    and Sup X = gamma1 l by auto
  hence gamma1 l < gamma2 k using ‹∀j. gamma1 j ≠ gamma2 k›
```

by (*simp add*: *less-eq-real-def*)
have *bdd-above X* **unfolding** *X-def* **using** *assms* **by** *auto*
have *gamma1 l* < *gamma1* (*l*+*1*) **using** *assms stm* **by** (*simp add*: *monotoneD*)
hence *gamma1* (*l*+*1*) ∉ *X* **using** ‹*Sup X* = *gamma1 l*› *cSup-upper* ‹*bdd-above X*›
by *fastforce*
hence *gamma2 k* < *gamma1* (*l*+*1*) **using** ‹*m*≤ *l*› **unfolding** *X-def*
by *fastforce*
show *False*
proof (*cases gamma2* (*k*−*1*) ≤ *gamma1 l*)
case *True*
hence *k*−*1* = *coarse-idx gamma1 gamma2 l*
using ‹*gamma1 l* < *gamma2 k*› *coarse-idx-eq assms encomp-atI* **by** *simp*
hence *gamma1* (*l*+*1*) ≤ *gamma2 k* **using** *coarse-idx-bounds assms*
by (*metis* (*mono-tags*, *opaque-lifting*) *add-diff-cancel diff-add-eq encomp-atD2*)
then show *?thesis* **using** ‹*gamma2 k* < *gamma1* (*l*+*1*)› **by** *simp*
next
case *False*
hence *gamma1 l* < *gamma2* (*k*−*1*) **by** *simp*
define *k′* **where** *k′* = *coarse-idx gamma1 gamma2 l*
have *gam2*: *gamma2 k′* ≤ *gamma1 l* ∧ *gamma1* (*l*+*1*) ≤ *gamma2* (*k′*+*1*)
using *assms k′-def encomp-atD1 encomp-atD2 coarse-idx-bounds*
by *metis*
hence *gamma2 k′* < *gamma2* (*k*−*1*) **using** ‹*gamma1 l* < *gamma2* (*k*−*1*)› **by** *simp*
hence *k′* < *k*−*1* **using** *assms stm mon mono-strict-invE* **by** *blast*
have *gamma2 k* < *gamma2* (*k′*+*1*)
using *gam2* ‹*gamma2 k* < *gamma1* (*l*+*1*)› **by** *simp*
hence *k* < *k′*+*1* **using** *assms stm mon mono-strict-invE* **by** *blast*
then show *?thesis* **using** ‹*k′* < *k*−*1*› **by** *simp*
qed
qed

lemma *coarse-idx-reached-unique*:
assumes *gamma1 m* ≤ *gamma2 k*
and *gamma2 k* < *gamma1 M*
and *k* = *coarse-idx gamma1 gamma2 i*
shows ∃!*j*. *gamma1 j* = *gamma2 k*
proof −
have ∃*j*. *gamma1 j* = *gamma2 k* **using** *assms coarse-idx-reached* **by** *simp*
from *this* **obtain** *j* **where** *gamma1 j* = *gamma2 k* **by** *auto*
{
fix *i*
assume *gamma1 i* = *gamma2 k*
hence *gamma1 i* = *gamma1 j* **using** ‹*gamma1 j* = *gamma2 k*› **by** *simp*
hence *i* = *j* **using** *assms stm* **by** (*simp add*: *strict-mono-eq*)
}
thus *?thesis* **using** ‹*gamma1 j* = *gamma2 k*› **by** *blast*

qed

lemma *encomp-idx-mono*:
assumes $i < j$
and *encomp-at gamma1 gamma2 i k*
and *encomp-at gamma1 gamma2 j l*
and $k \neq l$
shows $k < l$
proof (*rule ccontr*)
assume $\neg\ k < l$
hence $l \leq k$ **by** *simp*
hence $l < k$ **using** *assms* **by** *simp*
hence $l+1 \leq k$ **by** *simp*
hence *gamma2* $(l+1) \leq$ *gamma2* k **using** *mon*
by (*meson leD linorder-le-less-linear mono-strict-invE*)
also have ... $\leq$ *gamma1* i **using** *encomp-atD1*[*of gamma1 gamma2*] *assms*
by *simp*
also have ... $<$ *gamma1* $(i+1)$ **using** *stm*
by (*simp add*: *strict-mono-less*)
also have ... $\leq$ *gamma1* j **using** *assms stm strict-mono-less-eq*
zless-imp-add1-zle **by** *blast*
also have ... $<$ *gamma1* $(j+1)$ **using** *stm*
by (*simp add*: *strict-mono-less*)
also have ... $\leq$ *gamma2* $(l+1)$ **using** *assms encomp-atD2*[*of gamma1 gamma2*]

by *simp*
finally show *False* **by** *simp*
qed

lemma *encomp-idx-mono'*:
assumes $i \leq j$
and *encomp-at gamma1 gamma2 i k*
and *encomp-at gamma1 gamma2 j l*
shows $k \leq l$
proof (*cases* $i = j$)
case *True*
then show *?thesis*
using *assms encomp-idx-unique* **by** *auto*
next
case *False*
hence $i < j$ **using** *assms* **by** *simp*
show *?thesis*
proof (*cases* $k = l$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then show *?thesis* **using** ‹$i < j$› *assms encomp-idx-mono*[*of i j k l*]
by *simp*

qed
qed

lemma *encomp-idx-mono-conv*:
assumes $k < l$
and *encomp-at gamma1 gamma2 i k*
and *encomp-at gamma1 gamma2 j l*
shows $i < j$
proof (*rule ccontr*)
assume $\neg\ i < j$
hence $j < i$ **using** *assms* ‹$\neg\ i < j$› *encomp-at-unique*
linorder-less-linear mon stm **by** *blast*
hence $l < k$ **using** *encomp-idx-mono assms* **by** *simp*
thus *False* **using** *assms* **by** *simp*
qed

lemma *finer-idx-bound-eq*:
assumes *gamma1 m* $\leq$ *gamma2* (*coarse-idx gamma1 gamma2 i*)
and *gamma2* (*coarse-idx gamma1 gamma2 i*) $<$ *gamma1 M*
shows *gamma1* (*finer-idx-bound gamma1 gamma2 i*) =
gamma2 (*coarse-idx gamma1 gamma2 i*)
proof −
define *P* **where** $P = (\lambda i.$ *gamma1* (*finer-idx-bound gamma1 gamma2 i*) =
gamma2 (*coarse-idx gamma1 gamma2 i*))
have *P i* **unfolding** *P-def finer-idx-bound-def*
proof (*rule theI'*, *rule coarse-idx-reached-unique*)
show *gamma1 m* $\leq$ *gamma2* (*coarse-idx gamma1 gamma2 i*) **using** *assms* **by** *simp*
show *gamma2* (*coarse-idx gamma1 gamma2 i*) $<$ *gamma1 M* **using** *assms* **by** *simp*
qed (*simp add*: *assms*)
thus *?thesis* **using** *assms P-def* **by** *simp*
qed

lemma *finer-idx-bound-exists-eq*:
assumes $\exists\, m.$ *gamma1 m* $\leq$ *gamma2* (*coarse-idx gamma1 gamma2 i*)
and $\exists\, M.$ *gamma2* (*coarse-idx gamma1 gamma2 i*) $<$ *gamma1 M*
shows *gamma1* (*finer-idx-bound gamma1 gamma2 i*) =
gamma2 (*coarse-idx gamma1 gamma2 i*) **using** *assms finer-idx-bound-eq* **by** *auto*

lemma *finer-idx-bound-eq'*:
assumes $i \in$ *encompassed gamma1 gamma2 k*
and *gamma1 m* $\leq$ *gamma2 k*
and *gamma2 k* $<$ *gamma1 M*
shows *gamma1* (*finer-idx-bound gamma1 gamma2 i*) = *gamma2 k*
proof −
have $k =$ *coarse-idx gamma1 gamma2 i* **using** *encompassed-unique assms* **by** *simp*
thus *?thesis* **using** *finer-idx-bound-eq assms* **by** *simp*

qed

lemma *finer-idx-bound-exists-eq'*:
assumes $i \in$ *encompassed gamma1 gamma2 k*
and $\exists\, m.$ *gamma1 m* $\leq$ *gamma2 k*
and $\exists\, M.$ *gamma2 k* $<$ *gamma1 M*
shows *gamma1* (*finer-idx-bound gamma1 gamma2 i*) $=$ *gamma2 k*
using *assms finer-idx-bound-eq'* **by** *auto*

lemma *finer-idx-bound-mem*:
assumes *gamma1 m* $\leq$ *gamma2* (*coarse-idx gamma1 gamma2 i*)
and *gamma2* (*coarse-idx gamma1 gamma2 i* + *1*) $\leq$ *gamma1 M*
and *gamma2* (*coarse-idx gamma1 gamma2 i*) $\neq$
gamma2 (*coarse-idx gamma1 gamma2 i* + *1*)
shows *finer-idx-bound gamma1 gamma2 i* $\in$
encompassed gamma1 gamma2 (*coarse-idx gamma1 gamma2 i*)
proof $-$
define *k* **where** *k* $=$ *coarse-idx gamma1 gamma2 i*
have *gamma2 k* $<$ *gamma2* (*k*+*1*) **using** *mon assms*
by (*metis k-def less-eq-real-def monotoneD zle-add1-eq-le zless-add1-eq*)
hence *gamma2 k* $<$ *gamma1 M* **using** *assms k-def* **by** *simp*
define *idx* **where** *idx* $=$ *finer-idx-bound gamma1 gamma2 i*
have *gamma1 idx* $=$ *gamma2 k*
using *assms finer-idx-bound-eq idx-def k-def* ‹*gamma2 k* $<$ *gamma1 M*›
by *simp*
hence *gamma1* (*idx* + *1*) $\leq$ *gamma2* (*k*+*1*)
by (*metis* ‹*gamma2 k* $<$ *gamma2* (*k* + *1*)› *coarse-idx-bounds coarse-idx-eq*
encomp-at-def less-eq-real-def)
thus *?thesis*
using ‹*gamma1 idx* $=$ *gamma2 k*› *coarse-idx-eq encompassed-bounds*
idx-def k-def
by (*metis* ‹*gamma2 k* $<$ *gamma2* (*k* + *1*)› *order-refl*)
qed

lemma *finer-idx-bound-reached*:
assumes *gamma1 m* $\leq$ *gamma2* (*coarse-idx gamma1 gamma2 i*)
and *gamma2* (*coarse-idx gamma1 gamma2 i*) $<$ *gamma1 M*
and *gamma1 i* $=$ *gamma2* (*coarse-idx gamma1 gamma2 i*)
shows *i* $=$ *finer-idx-bound gamma1 gamma2 i*
using *assms coarse-idx-reached-unique finer-idx-bound-eq* **by** *blast*

lemma *finer-idx-bound-leq*:
assumes *gamma1 m* $\leq$ *gamma2* (*coarse-idx gamma1 gamma2 i*)
and *gamma2* (*coarse-idx gamma1 gamma2 i*) $<$ *gamma1 M*
shows *finer-idx-bound gamma1 gamma2 i* $\leq$ *i*
proof$-$
have *gamma1* (*finer-idx-bound gamma1 gamma2 i*) $=$
gamma2 (*coarse-idx gamma1 gamma2 i*)
using *assms finer-idx-bound-eq* **by** *simp*

```
  also have ... ≤ gamma1 i using assms coarse-idx-bounds
    by (simp add: encomp-atD1)
  finally have gamma1 (finer-idx-bound gamma1 gamma2 i) ≤ gamma1 i .
  thus ?thesis using assms stm by (simp add: strict-mono-less-eq)
qed

lemma finer-idx-bound-proj:
  assumes i ∈ encompassed gamma1 gamma2 k
  and j ∈ encompassed gamma1 gamma2 k
  and gamma1 m ≤ gamma2 k
  and gamma2 k < gamma1 M
shows finer-idx-bound gamma1 gamma2 i = finer-idx-bound gamma1 gamma2 j
proof (rule ccontr)
  define fi where fi = finer-idx-bound gamma1 gamma2 i
  define fj where fj = finer-idx-bound gamma1 gamma2 j
  assume fi ≠ fj
  have gamma1 fi = gamma2 k using finer-idx-bound-eq' assms fi-def by simp
  moreover have gamma1 fj = gamma2 k
    using finer-idx-bound-eq' assms fj-def by simp
  ultimately show False using stm by (metis ‹fi ≠ fj› strict-mono-eq)
qed

lemma finer-idx-bound-min:
  assumes i ∈ encompassed gamma1 gamma2 k
  and j ∈ encompassed gamma1 gamma2 k
  and gamma1 m ≤ gamma2 k
  and gamma2 k < gamma1 M
shows finer-idx-bound gamma1 gamma2 i ≤ j
  using assms finer-idx-bound-proj finer-idx-bound-leq
  by (metis encompassed-unique)

lemma coarse-idx-finer-bound:
  assumes gamma1 m ≤ gamma2 (coarse-idx gamma1 gamma2 i)
  and gamma2 (coarse-idx gamma1 gamma2 i) < gamma1 M
shows coarse-idx gamma1 gamma2 (finer-idx-bound gamma1 gamma2 i) =
  coarse-idx gamma1 gamma2 i
proof −
  define j where j = finer-idx-bound gamma1 gamma2 i
  define k where k = coarse-idx gamma1 gamma2 i
  have j ≤ i
    using j-def assms finer-ranges.finer-idx-bound-leq finer-ranges-axioms
    by blast
  hence gamma1 (j+1) ≤ gamma1 (i+1) using stm
    by (simp add: strict-mono-less-eq)
  hence gamma1 (j+1) ≤ gamma2 (k+1)
    using k-def encomp-atD2 coarse-idx-bounds order.trans by metis
  moreover have gamma2 k ≤ gamma1 j using assms k-def j-def
    by (simp add: finer-idx-bound-eq)
  ultimately show ?thesis using k-def j-def encomp-at-unique
```

using *assms stm mon coarse-idx-bounds encomp-atI* **by** *blast*
qed

lemma *finer-idx-bound-invol*:
assumes *gamma1 m* ≤ *gamma2* (*coarse-idx gamma1 gamma2 i*)
and *gamma2* (*coarse-idx gamma1 gamma2 i*) < *gamma1 M*
shows *finer-idx-bound gamma1 gamma2* (*finer-idx-bound gamma1 gamma2 i*) =
finer-idx-bound gamma1 gamma2 i
using *assms coarse-idx-finer-bound finer-idx-bound-eq finer-idx-bound-reached*
by *auto*

lemma *reached-imp-coarse*:
assumes *gamma1 i* = *gamma2 k*
and *gamma2 k* ≠ *gamma2* (*k*+1)
shows *gamma1* (*i*+1) ≤ *gamma2* (*k*+1)
proof (*rule ccontr*)
assume ¬ *gamma1* (*i* + 1) ≤ *gamma2* (*k* + 1)
hence *asm*: *gamma2* (*k*+1) < *gamma1* (*i*+1) **by** *simp*
have *gamma2 k* < *gamma2* (*k*+1) **using** *assms mon*
by (*metis linorder-neqE-linordered-idom mono-strict-invE*
order.asym zless-add1-eq)
have ∃ *j*. *encomp-at gamma1 gamma2 i j*
using *fin finer-range-def* **by** *simp*
hence ∃ *j*. *gamma2 j* ≤ *gamma1 i* ∧ *gamma1* (*i*+1) ≤ *gamma2* (*j*+1)
using *encomp-atD1 encomp-atD2* **by** *blast*
from *this* **obtain** *j* **where** *gamma2 j* ≤ *gamma1 i*
and *gamma1* (*i*+1) ≤ *gamma2* (*j*+1)
by *auto* **note** *jpr* = *this*
have *gamma2 j* ≤ *gamma2 k* **using** *jpr assms* **by** *simp*
moreover have *gamma2* (*k*+1) < *gamma2* (*j*+1) **using** *jpr asm* **by** *simp*
ultimately show *False* **using** *mon*
by (*metis assms*(2) *dual-order.trans mono-strict-invE monotoneD*
order-antisym-conv zle-add1-eq-le zless-add1-eq)
qed

lemma *less-imp-coarse*:
assumes *gamma1 m* < *gamma2 k*
and *gamma2 k* ≤ *gamma1 M*
and *gamma2 k* ≠ *gamma2* (*k*+1)
shows ∃ *i*. *encomp-at gamma1 gamma2 i k*
proof (*rule ccontr*)
assume ¬(∃ *i*. *encomp-at gamma1 gamma2 i k*)
hence *asm*: ∀ *i*. *gamma1 i* < *gamma2 k* ∨ *gamma2* (*k*+1) < *gamma1* (*i*+1)
using *not-le-imp-less* **unfolding** *encomp-at-def* **by** *auto*
define *B* **where** *B* = {*i*. *m* ≤ *i* ∧ *gamma1 i* < *gamma2 k*}
define *A* **where** *A* = {*i*. *gamma2* (*k*+1) < *gamma1* (*i*+1)}
have *m* ∈ *B* **using** *assms B-def* **by** *simp*
define *j1* **where** *j1* = *Sup B* + 1
have ∀ *j*∈ *B*. *m*≤ *j* **using** *B-def stm* **by** *simp*

moreover have $\forall j{\in}\ B.\ j \leq M$
using *assms stm B-def linorder-not-less strict-monoD* **by** *fastforce*
ultimately have $B{\subseteq}\ \{m..M\}$ **by** *auto*
hence *finite B* **using** *finite-subset* **by** *auto*
hence *Sup B* $\in$ *B*
by (*metis* ‹*m* $\in$ *B*› *dual-order.strict-iff-order finite-imp-Sup-less le-cSup-finite*)
hence *j1* $\notin$ *B*
using ‹*finite B*› *j1-def le-cSup-finite zle-add1-eq-le* **by** *blast*
hence *j1* $\in$ *A* **using** *asm A-def B-def* ‹*Sup B* $\in$ *B*› *j1-def* **by** *force*
hence *gamma2* (*k*+*1*) < *gamma1* (*j1* + *1*) **using** *A-def* **by** *simp*
have $\exists$ *l*. *gamma2 l* $\leq$ *gamma1 j1* $\wedge$ *gamma1* (*j1*+*1*) $\leq$ *gamma2* (*l*+*1*)
using *fin finer-range-def encomp-atD1 encomp-atD2* **by** *metis*
from *this* **obtain** *l1* **where** *gamma2 l1* $\leq$ *gamma1 j1*
and *gamma1* (*j1* + *1*) $\leq$ *gamma2* (*l1*+*1*)
by *auto* **note** *lppr* = *this*
show *False*
proof (*cases gamma2* (*k*+*1*) $\leq$ *gamma1 j1*)
case *True*
define *j* **where** *j* = *Sup B*
have *j1* = *j*+*1* **using** *j-def j1-def* **by** *simp*
have $\exists$ *l*. *gamma2 l* $\leq$ *gamma1 j* $\wedge$ *gamma1* (*j*+*1*) $\leq$ *gamma2* (*l*+*1*)
using *fin finer-range-def encomp-atD1 encomp-atD2* **by** *metis*
from *this* **obtain** *l* **where** *gamma2 l* $\leq$ *gamma1 j*
and *gamma1* (*j* + *1*) $\leq$ *gamma2* (*l*+*1*)
by *auto* **note** *lpr* = *this*
have *gamma2 l* < *gamma2 k* **using** ‹*Sup B* $\in$ *B*› *j-def lpr*
by (*simp add*: *B-def*)
hence *l* < *k* **using** *mon mono-strict-invE* **by** *blast*
hence *l*+*1* $\leq$ *k* **by** *simp*
hence *gamma2* (*l*+*1*) $\leq$ *gamma2 k* **using** *mon*
by (*meson leI less-le-not-le mono-strict-invE*)
moreover have *gamma2* (*k*+*1*) $\leq$ *gamma2* (*l*+*1*)
using *lpr True* ‹*j1* = *j* + *1*› *dual-order.trans* **by** *blast*
ultimately have *gamma2* (*k*+*1*) $\leq$ *gamma2 k* **by** *simp*
hence *gamma2* (*k*+*1*) = *gamma2 k*
using *mon*
by (*meson assms*(*3*) *dual-order.order-iff-strict mono-strict-invE order-less-imp-not-less zless-add1-eq*)
thus *?thesis* **using** *assms* **by** *simp*
next
case *False*
hence *gamma1 j1* < *gamma2* (*k*+*1*) **by** *simp*
have *gamma2* (*k*+*1*) < *gamma2* (*l1* + *1*) **using** *lppr* ‹*j1*$\in$ *A*› *A-def* **by** *auto*
hence *k* +*1* < *l1*+ *1* **using** *mon mono-strict-invE* **by** *blast*
hence *k*+*1* $\leq$ *l1* **by** *simp*
hence *gamma2* (*k*+*1*) $\leq$ *gamma2 l1* **using** *mon* **by** (*simp add*: *monotoneD*)
hence *gamma1 j1* < *gamma2 l1* **using** ‹*gamma1 j1* < *gamma2* (*k*+*1*)› **by** *simp*

```
    thus ?thesis using lppr by simp
  qed
qed

lemma ex-coarse-rep:
  assumes gamma1 m ≤ gamma2 k
  and gamma2 k ≤ gamma1 M
  and gamma2 k ≠ gamma2 (k+1)
shows ∃ i. encomp-at gamma1 gamma2 i k
proof (cases gamma1 m = gamma2 k)
  case True
  then show ?thesis using assms reached-imp-coarse
    by (metis encomp-at-def)
next
  case False
  hence gamma1 m < gamma2 k using assms by simp
  then show ?thesis using less-imp-coarse assms by simp
qed

lemma encompassed-ne:
  assumes gamma1 m ≤ gamma2 k
  and gamma2 k ≤ gamma1 M
  and gamma2 k ≠ gamma2 (k+1)
shows encompassed gamma1 gamma2 k ≠ {}
  using assms ex-coarse-rep unfolding encompassed-def by simp

lemma encompassed-ne':
  assumes ∃ m. gamma1 m ≤ gamma2 k
  and ∃ M. gamma2 k ≤ gamma1 M
  and gamma2 k ≠ gamma2 (k+1)
shows encompassed gamma1 gamma2 k ≠ {}
  using assms encompassed-ne by auto

lemma encompassed-finite:
  assumes gamma1 m ≤ gamma2 k
  and gamma2 (k+1) ≤ gamma1 M
  and gamma2 k ≠ gamma2 (k+1)
shows finite (encompassed gamma1 gamma2 k)
proof −
  have gamma2 k < gamma2 (k+1) using mon assms
    by (metis linorder-neqE-linordered-idom mono-strict-invE
        order.asym zless-add1-eq)
  hence lt: gamma2 k < gamma1 M using  assms by simp
  have encompassed gamma1 gamma2 k ≠ {} using assms encompassed-ne lt
    by (meson nless-le)
  hence ∃ i. i ∈ encompassed gamma1 gamma2 k by auto
  from this obtain i where i ∈ encompassed gamma1 gamma2 k by auto
  hence k = coarse-idx gamma1 gamma2 i using encompassed-unique by simp
  define j where j = finer-idx-bound gamma1 gamma2 i
```

```
  hence gamma1 j = gamma2 k
    using finer-idx-bound-eq assms ‹k = coarse-idx gamma1 gamma2 i› lt
    by simp
  have ∀ l∈ encompassed gamma1 gamma2 k. j ≤ l
   using finer-idx-bound-min j-def assms ‹i ∈ encompassed gamma1 gamma2 k› lt
    by auto
  moreover have ∀ l∈ encompassed gamma1 gamma2 k. l < M
  proof
    fix l
    assume l∈ encompassed gamma1 gamma2 k
    hence gamma1 (l+1) ≤ gamma1 M using encomp-at-def assms
      by (metis (mono-tags, opaque-lifting) coarse-idx-bounds
          dual-order.strict-trans2 encompassed-unique linorder-not-le)
    thus l < M using stm
      by (simp add: strict-mono-less-eq)
  qed
  ultimately have encompassed gamma1 gamma2 k ⊆ {j..< M} by auto
  thus ?thesis by (simp add: finite-subset)
qed

lemma encompassed-finite':
  assumes ∃ m. gamma1 m ≤ gamma2 k
  and ∃ M. gamma2 (k+1) ≤ gamma1 M
  and gamma2 k ≠ gamma2 (k+1)
shows finite (encompassed gamma1 gamma2 k) using assms encompassed-finite
  by auto

lemma encompassed-Min-in:
  assumes gamma1 m ≤ gamma2 k
  and gamma2 (k+1) ≤ gamma1 M
  and gamma2 k ≠ gamma2 (k+1)
shows Min (encompassed gamma1 gamma2 k) ∈ encompassed gamma1 gamma2 k
proof −
  define j where j = Min (encompassed gamma1 gamma2 k)
  have gamma2 k ≤ gamma2 (k+1) using mon by (simp add: monoD)
  hence gamma2 k ≤ gamma1 M using assms by simp
   hence encompassed gamma1 gamma2 k ≠ {} using assms encompassed-ne by
simp
  thus j∈ encompassed gamma1 gamma2 k
    using encompassed-finite encompassed-ne j-def assms by simp
qed

lemma encompassed-Max-in:
  assumes gamma1 m ≤ gamma2 k
  and gamma2 (k+1) ≤ gamma1 M
  and gamma2 k ≠ gamma2 (k+1)
shows Max (encompassed gamma1 gamma2 k) ∈ encompassed gamma1 gamma2 k
proof −
  define j where j = Max (encompassed gamma1 gamma2 k)
```

have *gamma2 k* ≤ *gamma2* (*k*+*1*) **using** *mon* **by** (*simp add*: *monoD*)
hence *gamma2 k* ≤ *gamma1 M* **using** *assms* **by** *simp*
hence *encompassed gamma1 gamma2 k* ≠ {} **using** *assms encompassed-ne* **by** *simp*
thus *j*∈ *encompassed gamma1 gamma2 k*
using *encompassed-finite encompassed-ne j-def assms* **by** *simp*
qed

lemma *encompassed-min-gamma-eq*:
assumes *gamma1 m* ≤ *gamma2 k*
and *gamma2* (*k*+*1*) ≤ *gamma1 M*
and *gamma2 k* ≠ *gamma2* (*k*+*1*)
shows *gamma1* (*Min* (*encompassed gamma1 gamma2 k*)) = *gamma2 k*
proof −
have *gamma2 k* < *gamma2* (*k*+*1*) **using** *mon assms*
by (*metis less-eq-real-def monotoneD zle-add1-eq-le zless-add1-eq*)
hence *gamma2 k* < *gamma1 M* **using** *assms* **by** *simp*
define *me* **where** *me* = *Min* (*encompassed gamma1 gamma2 k*)
define *fb* **where** *fb* = *finer-idx-bound gamma1 gamma2 me*
have *fb* ∈ *encompassed gamma1 gamma2 k* **using** *fb-def finer-idx-bound-mem*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *encompassed-Min-in encompassed-unique*
me-def)
hence *me* ≤ *fb* **using** *me-def*
using *assms finer-ranges.encompassed-finite finer-ranges-axioms* **by** *auto*
have *me* ∈ *encompassed gamma1 gamma2 k*
using *encompassed-Min-in*[*of m k M*] *assms me-def* **by** *simp*
hence *fb* ≤ *me* **using** *finer-idx-bound-min assms* ‹*gamma2 k* < *gamma1 M*› *fb-def*
by *blast*
hence *fb* = *me* **using** ‹*me* ≤ *fb*› **by** *simp*
thus *?thesis* **using** *assms fb-def* ‹*gamma2 k* < *gamma1 M*›
‹*fb* ∈ *encompassed gamma1 gamma2 k*› *me-def finer-idx-bound-eq*′[*of fb*]
by *simp*
qed

lemma *encompassed-min-gamma-eq*′:
assumes ∃ *m*. *gamma1 m* ≤ *gamma2 k*
and ∃ *M*. *gamma2* (*k*+*1*) ≤ *gamma1 M*
and *gamma2 k* ≠ *gamma2* (*k*+*1*)
shows *gamma1* (*Min* (*encompassed gamma1 gamma2 k*)) = *gamma2 k*
using *assms encompassed-min-gamma-eq* **by** *auto*

lemma *coarse-idx-upper*:
assumes *gamma2 k* < *gamma1 j*
and *j*∉ *encompassed gamma1 gamma2 k*
shows *k* < *coarse-idx gamma1 gamma2 j*
proof (*rule ccontr*)
define *k*′ **where** *k*′ = *coarse-idx gamma1 gamma2 j*

assume $\neg\ k < \textit{coarse-idx gamma1 gamma2 } j$
hence $k' \leq k$ **using** k'*-def* **by** *simp*
have $j\in$ *encompassed gamma1 gamma2* k'
by (*simp add*: *encompassed-bounds* k'*-def*)
hence $k'\neq k$ **using** *assms* **by** *auto*
hence $k' < k$ **using** $\langle k' \leq k\rangle$ **by** *simp*
hence $k'+1 < k+1$ **by** *simp*
have $\neg$ *encomp-at gamma1 gamma2 j k* **using** *assms encompassed-def* **by** *auto*
hence $\neg$ *gamma2 k* $\leq$ *gamma1 j* $\vee\ \neg$ *gamma1* $(j + 1) \leq$ *gamma2* $(k + 1)$
using *encomp-atI* **by** *auto*
hence $\neg$ *gamma1* $(j + 1) \leq$ *gamma2* $(k + 1)$ **using** *assms* **by** *simp*
hence *gamma2* $(k+1) <$ *gamma1* $(j+1)$ **by** *simp*
moreover have *gamma1* $(j+1) \leq$ *gamma2* $(k' + 1)$
using $\langle j\in$ *encompassed gamma1 gamma2* $k'\rangle$ *coarse-idx-bounds*
encomp-at-def k'*-def*
by *blast*
ultimately have *gamma2* $(k+1) <$ *gamma2* $(k'+1)$ **by** *simp*
thus *False* **using** $\langle k'+1 < k+1\rangle$ *mon mono-strict-invE* **by** *fastforce*
qed

lemma *encompassed-max-Suc-eq*:
assumes *gamma1 m* $\leq$ *gamma2 k*
and *gamma2* $(k+1) \leq$ *gamma1 M*
and *gamma2 k* $\neq$ *gamma2* $(k+1)$
and *gamma2* $(k+1) \neq$ *gamma2* $(k+2)$
shows *Max* (*encompassed gamma1 gamma2 k*) + *1* $\in$
encompassed gamma1 gamma2 $(k+1)$
proof −
define *j* **where** $j =$ *Max* (*encompassed gamma1 gamma2 k*)
have $j\in$ *encompassed gamma1 gamma2 k*
using *encompassed-Max-in j-def assms* **by** *simp*
hence *gamma1* $(j+1) \leq$ *gamma2* $(k+1)$ **using** *encompassed-def encomp-at-def*
by *blast*
have *gamma1 j* $<$ *gamma1* $(j+1)$ **using** *stm*
by (*simp add*: *strict-mono-less*)
hence *gamma2 k* $<$ *gamma1* $(j+1)$
using $\langle j\in$ *encompassed gamma1 gamma2 k*$\rangle$ *encomp-at-def*
by (*metis coarse-idx-bounds dual-order.trans encompassed-unique*
less-eq-real-def nle-le)
have *gamma2* $(k+1) \leq$ *gamma2* $(k+2)$ **using** *mon*
by (*simp add*: *monotoneD*)
hence *gamma2* $(k+1) <$ *gamma2* $(k+2)$ **using** *assms* **by** *simp*
define k' **where** $k' =$ *coarse-idx gamma1 gamma2* $(j+1)$
have *gamma2* $k' \leq$ *gamma1* $(j+1)$ **using** k'*-def*
by (*simp add*: *coarse-idx-bounds encomp-atD1*)
hence *gamma2* $k' \leq$ *gamma2* $(k+1)$ **using** $\langle$*gamma1* $(j+1) \leq$ *gamma2* $(k+1)\rangle$
by *simp*
have $j+1 \notin$ *encompassed gamma1 gamma2 k* **using** *j-def*
by (*meson Max-ge assms encompassed-finite linorder-not-less zless-add1-eq*)

hence $k < k'$ **using** *coarse-idx-upper k'-def* ‹*gamma2 k* < *gamma1* (*j*+*1*)› **by** *simp*
hence *k*+*1* ≤ k' **by** *simp*
hence *gamma2* (*k*+*1*) ≤ *gamma2 k'* **using** *mon* **by** (*simp add*: *monoD*)
hence *gamma2 k'* = *gamma2* (*k*+*1*) **using** ‹*gamma2 k'* ≤ *gamma2* (*k*+*1*)› **by** *simp*
hence *gamma2 k'* < *gamma2* (*k*+*2*) **using** ‹*gamma2* (*k*+*1*) < *gamma2* (*k*+*2*)› **by** *simp*
hence k' ≤ *k*+*1* **using** *mon mono-strict-invE* **by** *fastforce*
hence k' = *k*+*1* **using** ‹*k*+*1* ≤ k'› **by** *simp*
thus *?thesis* **using** *j-def encompassed-bounds k'-def* **by** *fastforce*
qed

lemma *encompassed-max-Suc-gamma-eq*:
assumes *gamma1 m* ≤ *gamma2 k*
and *gamma2* (*k*+*2*) ≤ *gamma1 M*
and *gamma2 k* ≠ *gamma2* (*k*+*1*)
and *gamma2* (*k*+*1*) ≠ *gamma2* (*k*+*2*)
shows *gamma1* (*Max* (*encompassed gamma1 gamma2 k*) + *1*) = *gamma2* (*k*+*1*)
proof −
have *gamma2* (*k*+*1*) ≤ *gamma2* (*k*+*2*) **using** *assms mon*
by (*simp add*: *monotoneD*)
hence *gamma2* (*k*+*1*) ≤ *gamma1 M* **using** *assms* **by** *simp*
have *gamma2 k* ≤ *gamma2* (*k*+*1*) **using** *assms mon*
by (*simp add*: *monotoneD*)
hence *gamma1 m* ≤ *gamma2* (*k*+*1*) **using** *assms* **by** *simp*
have *maxin*: *Max* (*encompassed gamma1 gamma2 k*) ∈
encompassed gamma1 gamma2 k
using *encompassed-Max-in assms* ‹*gamma2* (*k*+*1*) ≤ *gamma1 M*› **by** *simp*
define *sm* **where** *sm* = *Max* (*encompassed gamma1 gamma2 k*)+*1*
have *sm* ∈ *encompassed gamma1 gamma2* (*k*+*1*)
using *encompassed-max-Suc-eq sm-def assms* ‹*gamma2* (*k*+*1*) ≤ *gamma1 M*›
by *simp*
have *sm* = *Min* (*encompassed gamma1 gamma2* (*k*+*1*))
proof (*rule Min-eqI*[*symmetric*])
show *finite* (*encompassed gamma1 gamma2* (*k* + *1*))
proof (*rule encompassed-finite*)
show *gamma1 m* ≤ *gamma2* (*k*+*1*) **using** ‹*gamma1 m* ≤ *gamma2* (*k*+*1*)› **.**
show *gamma2* (*k* + *1*) ≠ *gamma2* (*k* + *1* + *1*) **using** *assms*
by (*simp add*: *add.assoc*)
show *gamma2* (*k* + *1* + *1*) ≤ *gamma1 M* **using** ‹*gamma2* (*k*+*2*) ≤ *gamma1 M*›
by (*simp add*: *add.assoc*)
qed
show *sm* ∈ *encompassed gamma1 gamma2* (*k* + *1*)
using ‹*sm* ∈ *encompassed gamma1 gamma2* (*k* + *1*)› **.**
fix *j*
assume *j*∈ *encompassed gamma1 gamma2* (*k*+*1*)
hence *coarse-idx gamma1 gamma2 j* = *k*+*1*

```
      using encompassed-unique by auto
    show sm ≤ j
    proof (rule ccontr)
      assume ¬ sm ≤ j
      hence j < sm by simp
      hence j ≤ Max (encompassed gamma1 gamma2 k) using sm-def by simp
      hence k+1 ≤ k
      proof (rule encomp-idx-mono')
        show encomp-at gamma1 gamma2 j (k + 1)
            using ‹j∈ encompassed gamma1 gamma2 (k+1)› unfolding encom-
passed-def
          by auto
        show encomp-at gamma1 gamma2 (Max (encompassed gamma1 gamma2
k)) k
          using maxin unfolding encompassed-def by auto
      qed
      thus False by simp
    qed
  qed
  thus ?thesis using sm-def
    by (metis ‹sm ∈ encompassed gamma1 gamma2 (k + 1)› coarse-idx-bounds
        dual-order.order-iff-strict encomp-atD1 encomp-atD2 encompassed-unique
        less-le-not-le maxin)
qed

lemma encompassed-max-Suc-gamma-eq':
  assumes ∃ m. gamma1 m ≤ gamma2 k
  and ∃ M. gamma2 (k+2) ≤ gamma1 M
  and gamma2 k ≠ gamma2 (k+1)
  and gamma2 (k+1) ≠ gamma2 (k+2)
shows gamma1 (Max (encompassed gamma1 gamma2 k) + 1) = gamma2 (k+1)
  using assms encompassed-max-Suc-gamma-eq by auto

end

lemma coarse-idx-refl:
  fixes gamma::int ⇒ real
  assumes strict-mono gamma
  shows i = coarse-idx gamma gamma i
proof (rule finer-ranges.coarse-idx-eq)
  show finer-ranges gamma gamma unfolding finer-ranges-def
  proof (intro conjI)
    show strict-mono gamma using assms by simp
    thus mono gamma by (simp add: strict-mono-mono)
    show finer-range gamma gamma using finer-range-refl by simp
  qed
  show gamma i ≤ gamma i by simp
   show gamma i < gamma (i+1) using assms unfolding strict-mono-def by
simp
```

qed

3.3 Pools with finer grids and coinciding profiles

definition *pool-coarse-idx* **where**
pool-coarse-idx = (λ*P1 P2 i*. *coarse-idx* (*grd P1*) (*grd P2*) *i*)

lemma *pool-coarse-idxD*:
assumes *k* = *pool-coarse-idx P1 P2 i*
shows *k* = *coarse-idx* (*grd P1*) (*grd P2*) *i*
using *assms* **unfolding** *pool-coarse-idx-def* **by** *simp*

definition *pool-finer-idx-bound* **where**
pool-finer-idx-bound = (λ*P1 P2 i*. *finer-idx-bound* (*grd P1*) (*grd P2*) *i*)

lemma *pool-finer-idx-boundD*:
assumes *l* = *pool-finer-idx-bound P1 P2 i*
shows *l* = *finer-idx-bound* (*grd P1*) (*grd P2*) *i*
using *assms* **unfolding** *pool-finer-idx-bound-def* **by** *simp*

definition *finer-pool* **where**
finer-pool P1 P2 ≡ *finer-grid P1 P2* ∧
(∀ *i*. *lq P1 i* = *lq P2* (*pool-coarse-idx P1 P2 i*)) ∧
(∀ *i*. *fee P1 i* = *fee P2* (*pool-coarse-idx P1 P2 i*))

lemma *finer-poolI*[*intro*]:
assumes *finer-range* (*grd P1*) (*grd P2*)
and (∀ *i*. *lq P1 i* = *lq P2* (*pool-coarse-idx P1 P2 i*))
and (∀ *i*. *fee P1 i* = *fee P2* (*pool-coarse-idx P1 P2 i*))
shows *finer-pool P1 P2*
using *assms* **unfolding** *finer-pool-def finer-grid-def* **by** *simp*

lemma *finer-poolD*:
assumes *finer-pool P1 P2* **shows**
finer-range (*grd P1*) (*grd P2*)
∀ *i*. *lq P1 i* = *lq P2* (*pool-coarse-idx P1 P2 i*)
∀ *i*. *fee P1 i* = *fee P2* (*pool-coarse-idx P1 P2 i*)
using *assms* **unfolding** *finer-pool-def* **by** *auto*

lemma *finer-pool-refl*:
assumes *strict-mono* (*grd P*)
shows *finer-pool P P*
proof
show *finer-range* (*grd P*) (*grd P*) **using** *finer-range-refl* **by** *simp*
have *i*: ∀ *i*. *pool-coarse-idx P P i* = *i*
using *coarse-idx-refl assms* **unfolding** *pool-coarse-idx-def* **by** *simp*
thus ∀ *i*. *lq P i* = *lq P* (*pool-coarse-idx P P i*) **by** *simp*
show ∀ *i*. *fee P i* = *fee P* (*pool-coarse-idx P P i*) **using** *i* **by** *simp*
qed

locale *finer-pools* =
  fixes *P1 P2*
  assumes *fin-pool*: *finer-pool P1 P2*

begin

lemma *finer-pool-grid*:
  shows *finer-range* (*grd P1*) (*grd P2*) **using** *fin-pool* **unfolding** *finer-pool-def*
  by *simp*

lemma *finer-pool-liq*:
  shows ∀ *i*. *lq P1 i* = *lq P2* (*pool-coarse-idx P1 P2 i*)
  using *fin-pool* **unfolding** *finer-pool-def*
  by *simp*

lemma *finer-pool-fee*:
  shows ∀ *i*. *fee P1 i* = *fee P2* (*pool-coarse-idx P1 P2 i*)
  using *fin-pool* **unfolding** *finer-pool-def*
  by *simp*

lemma *encompassed-liq-eq*:
  assumes *strict-mono* (*grd P1*)
  and *mono* (*grd P2*)
and *i* ∈ *encompassed* (*grd P1*) (*grd P2*) *k*
shows *lq P1 i* = *lq P2 k*
proof −
  have *k* = *coarse-idx* (*grd P1*) (*grd P2*) *i*
    using *assms finer-ranges.encompassed-unique finer-pool-grid*
    by (*simp add*: *finer-ranges.intro*)
  thus *?thesis* **using** *finer-pool-liq assms pool-coarse-idx-def* **by** *metis*
qed

lemma *encompassed-fee-eq*:
  assumes *strict-mono* (*grd P1*)
  and *mono* (*grd P2*)
and *i* ∈ *encompassed* (*grd P1*) (*grd P2*) *k*
shows *fee P1 i* = *fee P2 k*
proof −
  have *k* = *coarse-idx* (*grd P1*) (*grd P2*) *i*
    using *assms finer-ranges.encompassed-unique finer-pool-grid*
    by (*simp add*: *finer-ranges.intro*)
  thus *?thesis* **using** *finer-pool-fee assms pool-coarse-idx-def* **by** *metis*
qed

lemma *sum-rng-token*:
  assumes *strict-mono* (*grd P1*)
  and *mono* (*grd P2*)
  and *grd P1 m1* ≤ *grd P2 k*

```
  and grd P2 (k+1) ≤ grd P1 M1
  and grd P2 k ≠ grd P2 (k + 1)
  and ⋀ a b.  a ∈ encompassed (grd P1) (grd P2) b ⟹
    g (lq P1) a = g' (lq P2) b
  and ∀ i ∈ encompassed (grd P1) (grd P2) k. dff x i = f (i+1) − f i
shows sum (rng-token dff (g (lq P1)) x)
    (encompassed (grd P1) (grd P2) k) =
    (g' (lq P2)) k * (f (Max (encompassed (grd P1) (grd P2) k) + 1) −
    f (Min (encompassed (grd P1) (grd P2) k)))
proof −
  interpret finer-ranges grd P1 grd P2
    using assms finer-pool-grid by (simp add: finer-ranges-def)
  define Ek where Ek = encompassed (grd P1) (grd P2) k
  define m where m = Min Ek
  define M where M = Max Ek
  have m ≤ M using m-def M-def encompassed-Min-in encompassed-Max-in assms
    by (metis Ek-def Min.coboundedI encompassed-finite)
  have Ek = {m..M} unfolding Ek-def m-def M-def
    proof (rule encompassed-interval)
      show mono (grd P1)
        by (simp add: assms strict-mono-on-imp-mono-on)
      show finite (encompassed (grd P1) (grd P2) k)
        using encompassed-finite assms by blast
      show encompassed (grd P1) (grd P2) k ≠ {}
        using encompassed-ne assms encompassed-Max-in by fastforce
    qed
  have (∑ i∈Ek. (g (lq P1)) i * dff x i) = (∑ i∈Ek. (g' (lq P2)) k * dff x i)
  proof (rule sum.cong)
    fix i
    assume i ∈ Ek
    hence g (lq P1) i = g' (lq P2) k using assms Ek-def by simp
    thus (g (lq P1)) i * dff x i = (g' (lq P2)) k * dff x i by simp
  qed simp
  also have ... = (g' (lq P2)) k * (∑ i∈Ek. dff x i)
    by (simp add: sum-distrib-left)
  also have ... = (g' (lq P2)) k * (∑ i∈{m..M}. dff x i) using ‹Ek = {m..M}›
    by simp
  also have ... = (g' (lq P2)) k * (f (M+1) − f m)
  proof −
    have (∑ i∈{m..M}. dff x i) = (∑ i∈{m..M}. (f (i+1) − f i))
    proof (rule sum.cong)
      fix y
      assume y∈ {m..M}
      thus dff x y = f (y+1) − f y using assms ‹Ek = {m..M}› Ek-def by simp
    qed simp
    also have ... = f (M+1) − f m using int-telescoping-sum-le' ‹m ≤ M›
      by auto
    finally show ?thesis by simp
  qed
```

```
  finally have (∑ i∈Ek. (g (lq P1)) i * dff x i) =
    (g' (lq P2)) k * (f (M+1) - f m) .
  thus ?thesis unfolding Ek-def M-def m-def rng-token-def by simp
qed

lemma sum-rng-gen-quote:
  assumes strict-mono (grd P1)
  and mono (grd P2)
  and grd P1 m1 ≤ grd P2 k
  and grd P2 (k+2) ≤ grd P1 M1
  and grd P2 k ≠ grd P2 (k + 1)
  and grd P2 (k+1) ≠ grd P2 (k + 2)
  and ⋀ a b.  a ∈ encompassed (grd P1) (grd P2) b ⟹
    f (lq P1) a = f' (lq P2) b
shows sum (rng-gen-quote (grd P1) (f (lq P1)) x)
  (encompassed (grd P1) (grd P2) k) =
  rng-gen-quote (grd P2) (f' (lq P2)) x k
proof −
  interpret finer-ranges grd P1 grd P2
    using assms finer-pool-grid by (simp add: finer-ranges-def)
  have grd P2 (k+1) ≤ grd P2 (k+2) using mon
    by (simp add: monotoneD)
  hence grd P2 (k + 1) ≤ grd P1 M1 using assms by simp
  have sum (rng-gen-quote (grd P1) (f (lq P1)) x)
      (encompassed (grd P1) (grd P2) k) =
    (f' (lq P2)) k *
    (min x ((grd P1) (Max (encompassed (grd P1) (grd P2) k) + 1)) −
     min x ((grd P1) (Min (encompassed (grd P1) (grd P2) k))))
    unfolding rng-gen-quote-def
  proof (rule sum-rng-token)
    show grd P1 m1 ≤ grd P2 k using assms by simp
    show grd P2 (k + 1) ≤ grd P1 M1 using ‹grd P2 (k + 1) ≤ grd P1 M1› .
    show grd P2 k ≠ grd P2 (k + 1) using assms by simp
    show ∀ i∈encompassed (grd P1) (grd P2) k.
      gamma-min-diff (grd P1) x i = min x (grd P1 (i + 1))− min x (grd P1 i)
      unfolding gamma-min-diff-def by simp
    show ⋀ a b.  a ∈ encompassed (grd P1) (grd P2) b ⟹
      f (lq P1) a = f' (lq P2) b using assms
      by simp
  qed (simp add: assms)+
  also have ... = (f' (lq P2)) k * (min x (grd P2 (k+1)) − min x (grd P2 k))
  proof −
    have (grd P1) (Min (encompassed (grd P1) (grd P2) k)) = grd P2 k
     by (meson assms encompassed-min-gamma-eq ‹grd P2 (k + 1) ≤ grd P1 M1›)
    moreover have (grd P1) (Max (encompassed (grd P1) (grd P2) k) + 1) =
      grd P2 (k+1)
      using assms encompassed-max-Suc-gamma-eq by auto
    ultimately show ?thesis by simp
  qed
```

```
  finally show ?thesis
    unfolding rng-token-def gamma-min-diff-def rng-gen-quote-def .
qed

lemma sum-rng-gen-base:
  assumes strict-mono (grd P1)
  and mono (grd P2)
  and grd P1 m1 ≤ grd P2 k
  and grd P2 (k+2) ≤ grd P1 M1
  and grd P2 k ≠ grd P2 (k + 1)
  and grd P2 (k+1) ≠ grd P2 (k + 2)
  and ⋀ a b.  a ∈ encompassed (grd P1) (grd P2) b ⟹
    f (lq P1) a = f' (lq P2) b
shows sum (rng-gen-base (grd P1) (f (lq P1)) x)
  (encompassed (grd P1) (grd P2) k) =
  rng-gen-base (grd P2) (f' (lq P2)) x k
proof −
  interpret finer-ranges grd P1 grd P2
    using assms finer-pool-grid by (simp add: finer-ranges-def)
  have grd P2 (k+1) ≤ grd P2 (k+2) using mon
    by (simp add: monotoneD)
  hence grd P2 (k + 1) ≤ grd P1 M1 using assms by simp
  have sum (rng-gen-base (grd P1) (f (lq P1)) x)
      (encompassed (grd P1) (grd P2) k) =
    (f' (lq P2)) k ∗
    (−inverse (max x ((grd P1) (Max (encompassed (grd P1) (grd P2) k) + 1)))
−
     (−inverse (max x ((grd P1) (Min (encompassed (grd P1) (grd P2) k))))))
    unfolding rng-gen-base-def
  proof (rule sum-rng-token)
    show grd P1 m1 ≤ grd P2 k using assms by simp
    show grd P2 (k + 1) ≤ grd P1 M1 using ‹grd P2 (k + 1) ≤ grd P1 M1› .
    show grd P2 k ≠ grd P2 (k + 1) using assms by simp
    show ∀ i∈encompassed (grd P1) (grd P2) k.
      inv-gamma-max-diff (grd P1) x i =
      − inverse (max x (grd P1 (i + 1))) − − inverse (max x (grd P1 i))
      unfolding inv-gamma-max-diff-def by simp
    show ⋀ a b.  a ∈ encompassed (grd P1) (grd P2) b ⟹
      f (lq P1) a = f' (lq P2) b using assms by simp
  qed (simp add: assms)+
  also have ... = (f' (lq P2)) k ∗
    (inverse (max x ((grd P1) (Min (encompassed (grd P1) (grd P2) k)))) −
    inverse (max x ((grd P1) (Max (encompassed (grd P1) (grd P2) k) + 1))))
    by simp
  also have ... = (f' (lq P2)) k ∗
    (inverse (max x (grd P2 k)) − inverse (max x (grd P2 (k+1))))
  proof −
    have (grd P1) (Min (encompassed (grd P1) (grd P2) k)) = grd P2 k
     by (meson assms encompassed-min-gamma-eq ‹grd P2 (k + 1) ≤ grd P1 M1›)
```

```
    moreover have (grd P1) (Max (encompassed (grd P1) (grd P2) k) + 1) =
      grd P2 (k+1)
      using assms encompassed-max-Suc-gamma-eq by auto
    ultimately show ?thesis by simp
  qed
  finally show ?thesis
    unfolding rng-token-def inv-gamma-max-diff-def rng-gen-base-def .
qed

lemma finer-imp-finite-liq:
  assumes strict-mono (grd P1)
  and mono (grd P2)
  and finite-liq P2
  and ⋀k. lq P2 k ≠ 0 ⟹ finite (encompassed (grd P1) (grd P2) k)
  shows finite-liq P1
proof −
  interpret finer-ranges grd P1 grd P2
    using assms finer-pool-grid by (simp add: finer-ranges-def)
  have {i. lq P1 i ≠ 0} ⊆
    (⋃ (encompassed (grd P1) (grd P2) ‘ {k. lq P2 k ≠ 0}))
  proof
    fix i
    assume i∈ {i. lq P1 i ≠ 0}
    hence lq P1 i ≠ 0 by simp
    define k where k = coarse-idx (grd P1) (grd P2) i
    have i ∈ encompassed (grd P1) (grd P2) k using k-def encompassed-bounds
      by simp
    moreover have lq P2 k ≠ 0 using ‹lq P1 i ≠ 0› k-def finer-pool-liq
      by (metis pool-coarse-idx-def)
    ultimately show i ∈ (⋃ (encompassed (grd P1) (grd P2)‘ {k. lq P2 k ≠ 0}))
      by auto
  qed
  thus ?thesis
    by (metis (mono-tags, lifting) assms(3) assms(4) finite-UN-I
        finite-liqD finite-liqI finite-subset mem-Collect-eq)
qed

lemma finer-imp-finite-liq':
  assumes finer-pool P1 P2
  and strict-mono (grd P1)
  and mono (grd P2)
  and finite-liq P1
  and finite {k. encompassed (grd P1) (grd P2) k = {}}
  shows finite-liq P2
proof −
  interpret finer-ranges grd P1 grd P2
    using assms finer-pool-grid by (simp add: finer-ranges-def)
  have {k. lq P2 k ≠ 0} ⊆
    {k. encompassed (grd P1) (grd P2) k = {}} ∪
```

```
    coarse-idx (grd P1) (grd P2) ‘{i. lq P1 i ≠ 0}
  proof
    fix k
    assume k∈ {i. lq P2 i ≠ 0}
    hence lq P2 k ≠ 0 by simp
    show k ∈ {k. encompassed (grd P1) (grd P2) k = {}} ∪
      coarse-idx (grd P1) (grd P2) ‘{i. lq P1 i ≠ 0}
    proof
      assume asm: k ∉ coarse-idx (grd P1) (grd P2) ‘ {i. lq P1 i ≠ 0}
      show k ∈ {k. encompassed (grd P1) (grd P2) k = {}}
      proof (rule ccontr)
        assume k ∉ {k. encompassed (grd P1) (grd P2) k = {}}
        hence encompassed (grd P1) (grd P2) k ≠ {} by simp
        hence ∃ i. i ∈ encompassed (grd P1) (grd P2) k by auto
        from this obtain i where i ∈ encompassed (grd P1) (grd P2) k by auto
        hence k = coarse-idx (grd P1) (grd P2) i
          by (simp add: encompassed-unique)
        hence lq P1 i ≠ 0
          using assms ‹lq P2 k ≠ 0› finer-pool-liq pool-coarse-idx-def
          by presburger
        hence k ∈ coarse-idx (grd P1) (grd P2) ‘ {i. lq P1 i ≠ 0}
          using ‹k = coarse-idx (grd P1) (grd P2) i› by blast
        thus False using asm by simp
      qed
    qed
  qed
  moreover have finite (coarse-idx (grd P1) (grd P2) ‘{i. lq P1 i ≠ 0})
    using assms finite-liqD by auto
  ultimately show ?thesis using assms
    by (metis finite-UnI finite-liqI rev-finite-subset)
qed

end
```

3.4 Spanning grids

```
definition span-grid where
span-grid P ⟷ strict-mono (grd P) ∧ (∀ i. 0 < grd P i) ∧
    (∀ r>0. ∃ i. grd P i < r) ∧ (∀  r. ∃ i. r < grd P i)

lemma span-gridD:
  assumes span-grid P
  shows strict-mono (grd P) ∀ i. 0 < grd P i
    ∀ r>0. ∃ i. grd P i < r ∀  r. ∃ i. r < grd P i
  using assms unfolding span-grid-def by simp+

lemma span-gridI[intro]:
  assumes strict-mono (grd P)
  and ∀ i. 0 < grd P i
```

and $\forall\, r{>}0.\ \exists\, i.\ grd\ P\ i < r$
and $\forall\ \ r.\ \exists\, i.\ r < grd\ P\ i$
shows *span-grid P* **using** *assms* **unfolding** *span-grid-def* **by** *simp*

lemma *span-grid-eq*:
assumes *span-grid P*
and *grd P = grd P'*
shows *span-grid P'* **using** *assms* **unfolding** *span-grid-def* **by** *simp*

locale *finer-spanning-pool = finer-pools +*
assumes *span*: *span-grid P1*

begin

lemma *finer-spanning-gt*:
shows $\exists\, i.\ r < grd\ P2\ i$
proof −
have $\exists\, i.\ r < grd\ P1\ i$ **using** *span span-gridD* **by** *simp*
from *this* **obtain** *i* **where** $r < grd\ P1\ i$ **by** *auto*
hence $r < grd\ P1\ (i{+}1)$ **using** *span*
by (*metis dual-order.strict-trans less-add-one monotoneD*
span-gridD(*1*))
have $\exists\, k.$ *encomp-at* (*grd P1*) (*grd P2*) *i k* **using** *span finer-range-def*
finer-pool-grid **by** *simp*
from *this* **obtain** *k* **where** *encomp-at* (*grd P1*) (*grd P2*) *i k* **by** *auto*
hence *grd P1* $(i{+}1) \leq$ *grd P2* $(k{+}1)$ **using** *encomp-atD2*[*of grd P1 - i k*]
by *simp*
hence $r <$ *grd P2* $(k{+}1)$ **using** ‹$r <$ *grd P1* $(i + 1)$› **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *finer-spanning-lt*:
assumes $0 < r$
shows $\exists\, i.$ *grd P2* $i < r$
proof −
have $\exists\, i.$ *grd P1* $i < r$ **using** *assms finer-pool-grid span-gridD*
by (*simp add*: *span*)
from *this* **obtain** *i* **where** *grd P1* $i < r$ **by** *auto*
have $\exists\, k.$ *encomp-at* (*grd P1*) (*grd P2*) *i k* **using** *assms finer-pool-grid span*
by (*simp add*: *finer-range-def*)
from *this* **obtain** *k* **where** *encomp-at* (*grd P1*) (*grd P2*) *i k* **by** *auto*
hence *grd P2 k* $\leq$ *grd P1 i* **using** *encomp-atD1*[*of grd P1 - i k*]
by *simp*
hence *grd P2 k* $< r$ **using** ‹*grd P1 i* $< r$› **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *finer-span-grid*:
assumes $\forall\, i.\ 0 <$ *grd P2 i*

and *strict-mono* (*grd P2*)
shows *span-grid P2*
proof
show *strict-mono* (*grd P2*) **using** *assms* **by** *simp*
show $\forall\, i.\ 0 < grd\ P2\ i$ **using** *assms* **by** *simp*
show $\forall\, r.\ \exists\, i.\ r < grd\ P2\ i$ **using** *finer-spanning-gt assms* **by** *simp*
show $\forall\, r{>}0.\ \exists\, i.\ grd\ P2\ i < r$ **using** *finer-spanning-lt assms* **by** *simp*
qed

end

locale *finer-two-spanning-pools* = *finer-spanning-pool* +
assumes *span2*: *span-grid P2*

sublocale *finer-two-spanning-pools* ⊆ *finer-ranges grd P1 grd P2*
proof (*rule finer-ranges.intro*)
show *strict-mono* (*grd P1*) **using** *span span-gridD* **by** *simp*
show *mono* (*grd P2*) **using** *span span2*
by (*simp add*: *span-gridD*(*1*) *strict-mono-on-imp-mono-on*)
show *finer-range* (*grd P1*) (*grd P2*) **using** *finer-pool-grid* **by** *simp*
qed

context *finer-two-spanning-pools*
begin

lemma *spanning-sum-rng-gen-quote*:
assumes $\bigwedge\ a\ b.\ \ a \in encompassed\ (grd\ P1)\ (grd\ P2)\ b \Longrightarrow$
$f\ (lq\ P1)\ a = f'\ (lq\ P2)\ b$
shows *sum* (*rng-gen-quote* (*grd P1*) (*f* (*lq P1*)) *x*)
(*encompassed* (*grd P1*) (*grd P2*) *k*) =
rng-gen-quote (*grd P2*) (*f'* (*lq P2*)) *x k*
proof −
have *b*: *strict-mono* (*grd P1*) **using** *assms span-gridD span* **by** *simp*
have *c*: *mono* (*grd P2*) **using** *span2 span-gridD*
by (*simp add*: *strict-mono-mono*)
have *d*: $grd\ P2\ k \neq grd\ P2\ (k + 1)$ **using** *span2 span-gridD*
by (*simp add*: *strict-mono-eq*)
have *e*: $grd\ P2\ (k + 1) \neq grd\ P2\ (k + 2)$ **using** *span2 span-gridD*
by (*simp add*: *strict-mono-eq*)
have $\exists\, m.\ grd\ P1\ m \leq grd\ P2\ k$ **using** *span2 span-gridD span*
by (*meson order-less-imp-le*)
moreover have $\exists\, M.\ grd\ P2\ k \leq grd\ P1\ M$ **using** *assms span-gridD span*
by (*meson order-less-imp-le*)
ultimately show *?thesis* **using** *sum-rng-gen-quote*[*OF b c - - d e*]
by (*meson assms less-eq-real-def span span-gridD*(*4*))
qed

lemma *spanning-sum-rng-gen-base*:
assumes $\bigwedge\ a\ b.\ \ a \in encompassed\ (grd\ P1)\ (grd\ P2)\ b \Longrightarrow$

```
    f (lq P1) a = f' (lq P2) b
shows sum (rng-gen-base (grd P1) (f (lq P1)) x)
  (encompassed (grd P1) (grd P2) k) =
  rng-gen-base (grd P2) (f' (lq P2)) x k
proof −
  have b: strict-mono (grd P1) using assms span span-gridD by simp
  have c: mono (grd P2) using span2 span-gridD
    by (simp add: strict-mono-mono)
  have d: grd P2 k ≠ grd P2 (k + 1) using span2 span-gridD
    by (simp add: strict-mono-eq)
  have e: grd P2 (k + 1) ≠ grd P2 (k + 2) using span2 span-gridD
    by (simp add: strict-mono-eq)
  have ∃ m. grd P1 m ≤ grd P2 k using span2 span span-gridD
    by (meson order-less-imp-le)
  moreover have ∃ M. grd P2 k ≤ grd P1 M using assms span span-gridD
    by (meson order-less-imp-le)
  ultimately show ?thesis using sum-rng-gen-base[OF b c - - d e]
    by (meson assms span less-eq-real-def span-grid-def)
qed

lemma span-grid-encompassed:
shows finite (encompassed (grd P1) (grd P2) k)
proof (rule finer-ranges.encompassed-finite')
  show ∃ m. grd P1 m ≤ grd P2 k using span2 span span-gridD
    by (meson order-less-imp-le)
  show ∃ M. grd P2 (k+1) ≤ grd P1 M using span2 span span-gridD
    by (meson order-less-imp-le)
  show grd P2 k ≠ grd P2 (k + 1) using span2 span-gridD(1)
    by (simp add: strict-mono-eq)
  show finer-ranges (grd P1) (grd P2) unfolding finer-ranges-def
    by (simp add: finer-pool-grid span span2 span-gridD(1)
        strict-mono-mono)
qed

lemma span-grids-finite-liq:
  assumes finite-liq P2
shows finite-liq P1
proof (rule finer-imp-finite-liq)
  show strict-mono (grd P1) using assms span span-gridD by simp
  show finite-liq P2 using assms by simp
  show mono (grd P2) using assms span2 span-gridD
    by (simp add: strict-mono-on-imp-mono-on)
  show ⋀k. lq P2 k ≠ 0 ⟹ finite (encompassed (grd P1) (grd P2) k)
    using assms span-grid-encompassed finer-pool-grid by simp
qed

lemma span-grids-ex-le:
shows ∃ m. grd P1 m ≤ grd P2 k
by (meson span span2 linorder-le-less-linear order.asym
```

span-gridD(*2*) *span-gridD*(*3*))

lemma *span-grids-ex-ge*:
shows $\exists M.$ *grd P2 k* $\leq$ *grd P1 M*
by (*meson span nless-le span-gridD*(*4*))

lemma *span-grids-encompassed-ne*:
shows *encompassed* (*grd P1*) (*grd P2*) *k* $\neq$ {}
proof (*rule encompassed-ne'*)
show $\exists m.$ *grd P1 m* $\leq$ *grd P2 k* **using** *span-grids-ex-le span* **by** *simp*
show $\exists M.$ *grd P2 k* $\leq$ *grd P1 M* **using** *span-grids-ex-ge span* **by** *simp*
show *grd P2 k* $\neq$ *grd P2* (*k* + *1*) **using** *span2 span-gridD*
by (*simp add*: *strict-mono-eq*)
qed

end

3.5 Spanning grids and finite liquidity

locale *finer-two-span-finite-liq* = *finer-two-spanning-pools* +
assumes *fin-liq*: *finite-liq P1*

sublocale *finer-two-span-finite-liq* $\subseteq$ *finite-liq-pool P1*
by (*unfold-locales*, (*simp add*: *fin-liq*))

lemma (**in** *finer-two-span-finite-liq*) *span-grids-finite-liq'*:
shows *finite-liq P2*
proof (*rule finer-imp-finite-liq'*)
show *finer-pool P1 P2* **using** *fin-pool fin-pool* **by** *simp*
show *strict-mono* (*grd P1*) **using** *span span-gridD* **by** *simp*
show *finite-liq P1* **using** *fin-liq* **by** *simp*
show *mono* (*grd P2*) **using** *span2 span-gridD*
by (*simp add*: *strict-mono-on-imp-mono-on*)
have $\forall k.$ *encompassed* (*grd P1*) (*grd P2*) *k* $\neq$ {}
using *span-grids-encompassed-ne*
by (*simp add*: *finer-pool-grid*)
thus *finite* {*k*. *encompassed* (*grd P1*) (*grd P2*) *k* = {}} **by** *simp*
qed

sublocale *finer-two-span-finite-liq* $\subseteq$ *finite-liq-pool P2*
by (*unfold-locales*, (*simp add*: *span-grids-finite-liq'*))

context *finer-two-span-finite-liq*
begin

lemma *finer-pool-encompassed-Union*:
shows ($\bigcup$ (*encompassed* (*grd P1*) (*grd P2*) '{*i*. *lq P2 i* $\neq$ *0*})) =
{*i*. *lq P1 i* $\neq$ *0*}
proof

```
  show ⋃ (encompassed (grd P1) (grd P2) `{i. lq P2 i ≠ 0}) ⊆ {i. lq P1 i ≠ 0}
  proof
    fix j
    assume j ∈ ⋃ (encompassed (grd P1) (grd P2) ` {i. lq P2 i ≠ 0})
    hence ∃ k. lq P2 k ≠ 0 ∧ j ∈ encompassed (grd P1) (grd P2) k by auto
    from this obtain k where lq P2 k ≠ 0 and
      j ∈ encompassed (grd P1) (grd P2) k by auto
    hence k = pool-coarse-idx P1 P2 j
      using pool-coarse-idx-def encompassed-unique by metis
    hence lq P1 j = lq P2 k using span-grids-finite-liq' finer-pool-liq
      by simp
    hence lq P1 j ≠ 0 using ‹lq P2 k ≠ 0› by simp
    thus j∈ {i. lq P1 i ≠ 0} by simp
  qed
  show {i. lq P1 i ≠ 0} ⊆ ⋃ (encompassed (grd P1) (grd P2)` {i. lq P2 i ≠ 0})
  proof
    fix j
    assume j∈ {i. lq P1 i≠ 0}
    hence lq P1 j ≠ 0 by simp
    hence lq P2 (pool-coarse-idx P1 P2 j) ≠ 0
      using pool-coarse-idx-def finer-pool-liq by simp
    hence pool-coarse-idx P1 P2 j ∈ {i. lq P2 i ≠ 0} by simp
    moreover have j ∈ encompassed (grd P1) (grd P2) (pool-coarse-idx P1 P2 j)
      using encompassed-bounds unfolding pool-coarse-idx-def by auto
    ultimately show j ∈ ⋃ (encompassed (grd P1) (grd P2)` {i. lq P2 i ≠ 0})
      by auto
  qed
qed

lemma spanning-finer-gen-quote-eq:
  assumes ⋀ a b.  a ∈ encompassed (grd P1) (grd P2) b ⟹
    f (lq P1) a = f' (lq P2) b
  and ⋀i. lq P2 i = 0 ⟹ f' (lq P2) i = 0
  and ⋀i. lq P1 i = 0 ⟹ f (lq P1) i = 0
  shows gen-quote (grd P1) (f (lq P1)) x = gen-quote (grd P2) (f' (lq P2)) x
proof −
  define rg2 where rg2 = rng-token (gamma-min-diff (grd P2)) (f' (lq P2)) x
  define Lnz2 where Lnz2 = {i. lq P2 i ≠ 0}
  define Lnz1 where Lnz1 = {i. lq P1 i ≠ 0}
  have finite-liq P1 using fin-liq by simp
  have sm: ⋀ x k. sum (rng-gen-quote (grd P1) (f (lq P1)) x)
    (encompassed (grd P1) (grd P2) k) =
    rng-gen-quote (grd P2) (f' (lq P2)) x k
    using spanning-sum-rng-gen-quote assms by simp
  have gen-quote (grd P2) (f' (lq P2)) x = sum rg2 Lnz2
    unfolding gen-quote-def gen-token-def rg2-def Lnz2-def
    by (rule finite-support-sum, (simp add: assms)+)
  also have ... = sum (λk. sum (rng-gen-quote (grd P1) (f (lq P1)) x)
    (encompassed (grd P1) (grd P2) k)) Lnz2
```

```
  proof (rule sum.cong)
    show ⋀xa. xa ∈ Lnz2 ⟹ rg2 xa = sum (rng-gen-quote (grd P1) (f (lq P1))
x)
      (encompassed (grd P1) (grd P2) xa)
    proof −
      fix k
      assume k∈ Lnz2
      show rg2 k = sum (rng-gen-quote (grd P1) (f (lq P1)) x)
      (encompassed (grd P1) (grd P2) k)
        using sm unfolding rg2-def rng-gen-quote-def by simp
    qed
  qed simp
  also have ... = sum (rng-gen-quote (grd P1) (f (lq P1)) x)
    (⋃ (encompassed (grd P1) (grd P2) ‘ Lnz2))
  proof (rule sum.UNION-disjoint[symmetric])
    show finite Lnz2 using Lnz2-def span-grids-finite-liq′ finite-liqD
      by simp
    show ∀ i∈Lnz2. finite (encompassed (grd P1) (grd P2) i)
      using assms span-grid-encompassed
      by (simp add: finer-pool-grid)
    show ∀ i∈Lnz2. ∀ j∈Lnz2. i ≠ j ⟶
      encompassed (grd P1) (grd P2) i ∩ encompassed (grd P1) (grd P2) j = {}
      using encompassed-inj by simp
  qed
  also have ... = sum (rng-gen-quote (grd P1) (f (lq P1)) x) Lnz1
  proof −
    have (⋃ (encompassed (grd P1) (grd P2) ‘ Lnz2)) = Lnz1
      using finer-pool-encompassed-Union Lnz1-def Lnz2-def assms by simp
    thus ?thesis by simp
  qed
  also have ... = infsum (rng-gen-quote (grd P1) (f (lq P1)) x) UNIV
    unfolding Lnz1-def rng-gen-quote-def
  proof (rule finite-nz-support.finite-support-sum[symmetric])
    show finite-nz-support (lq P1)
      using fin-liq finite-liq-def finite-nz-support.intro by auto
  qed (simp add: assms)
  finally show ?thesis unfolding gen-quote-def gen-token-def rng-gen-quote-def
    by simp
qed

lemma spanning-finer-gen-base-eq:
  assumes ⋀ a b.  a ∈ encompassed (grd P1) (grd P2) b ⟹
    f (lq P1) a = f′ (lq P2) b
  and ⋀i. lq P2 i = 0 ⟹ f′ (lq P2) i = 0
  and ⋀i. lq P1 i = 0 ⟹ f (lq P1) i = 0
  shows gen-base (grd P1) (f (lq P1)) x = gen-base (grd P2) (f′ (lq P2)) x
proof −
  define rg2 where rg2 =rng-token (inv-gamma-max-diff (grd P2)) (f′ (lq P2))
x
```

define *Lnz2* **where** *Lnz2* = {*i*. *lq P2 i* ≠ *0*}
define *Lnz1* **where** *Lnz1* = {*i*. *lq P1 i* ≠ *0*}
have *finite-liq P1* **using** *fin-liq* **by** *simp*
have *sm*: ⋀ *x k*. *sum* (*rng-gen-base* (*grd P1*) (*f* (*lq P1*)) *x*)
(*encompassed* (*grd P1*) (*grd P2*) *k*) =
rng-gen-base (*grd P2*) (*f'* (*lq P2*)) *x k*
using *spanning-sum-rng-gen-base assms* **by** *simp*
have *gen-base* (*grd P2*) (*f'* (*lq P2*)) *x* = *sum rg2 Lnz2*
unfolding *gen-base-def gen-token-def rg2-def Lnz2-def*
by (*rule finite-support-sum*, (*simp add*: *assms*)+)
also have ... = *sum* (λ*k*. *sum* (*rng-gen-base* (*grd P1*) (*f* (*lq P1*)) *x*)
(*encompassed* (*grd P1*) (*grd P2*) *k*)) *Lnz2*
proof (*rule sum.cong*)
show ⋀*xa*. *xa* ∈ *Lnz2* ⟹ *rg2 xa* = *sum* (*rng-gen-base* (*grd P1*) (*f* (*lq P1*)) *x*)

(*encompassed* (*grd P1*) (*grd P2*) *xa*)
proof −
fix *k*
assume *k*∈ *Lnz2*
show *rg2 k* = *sum* (*rng-gen-base* (*grd P1*) (*f* (*lq P1*)) *x*)
(*encompassed* (*grd P1*) (*grd P2*) *k*)
using *sm* **unfolding** *rg2-def rng-gen-base-def* **by** *simp*
qed
qed *simp*
also have ... = *sum* (*rng-gen-base* (*grd P1*) (*f* (*lq P1*)) *x*)
(⋃ (*encompassed* (*grd P1*) (*grd P2*) ' *Lnz2*))
proof (*rule sum.UNION-disjoint*[*symmetric*])
show *finite Lnz2* **using** *Lnz2-def assms finite-liqD*
span-grids-finite-liq' **by** *auto*
show ∀ *i*∈*Lnz2*. *finite* (*encompassed* (*grd P1*) (*grd P2*) *i*)
using *assms span-grid-encompassed*
by (*simp add*: *finer-pool-grid*)
show ∀ *i*∈*Lnz2*. ∀ *j*∈*Lnz2*. *i* ≠ *j* ⟶
encompassed (*grd P1*) (*grd P2*) *i* ∩ *encompassed* (*grd P1*) (*grd P2*) *j* = {}
using *encompassed-inj* **by** *simp*
qed
also have ... = *sum* (*rng-gen-base* (*grd P1*) (*f* (*lq P1*)) *x*) *Lnz1*
proof −
have (⋃ (*encompassed* (*grd P1*) (*grd P2*) ' *Lnz2*)) = *Lnz1*
using *finer-pool-encompassed-Union Lnz1-def Lnz2-def assms* **by** *simp*
thus *?thesis* **by** *simp*
qed
also have ... = *infsum* (*rng-gen-base* (*grd P1*) (*f* (*lq P1*)) *x*) *UNIV*
unfolding *Lnz1-def rng-gen-base-def*
proof (*rule finite-nz-support.finite-support-sum*[*symmetric*])
show *finite-nz-support* (*lq P1*)
using *fin-liq finite-liq-def finite-nz-support.intro* **by** *auto*
qed (*simp add*: *assms*)
finally show *?thesis* **unfolding** *gen-base-def gen-token-def rng-gen-base-def*

by *simp*
qed

end

end
theory *CLMM-Description* **imports** *Grid-Information*

begin

4 CLMM description

Definition of CLMMs (Concentrated Liquidity Market Makers)

4.1 Preliminary results

definition *clmm-dsc* **where**
clmm-dsc P $\longleftrightarrow$ (*span-grid P*) $\wedge$ (*finite-liq P*) $\wedge$ ($\forall$ *i*. *0* $\leq$ *lq P i*) $\wedge$
($\forall$ *i*. *0* $\leq$ *fee P i*) $\wedge$ ($\forall$ *i*. *fee P i* $<$ *1*)

lemma *clmm-dscI*[*intro*]:
assumes *span-grid P*
and *finite-liq P*
and $\forall$ *i*. *0* $\leq$ *lq P i*
and $\forall$ *i*. *0* $\leq$ *fee P i*
and $\forall$ *i*. *fee P i* $<$ *1*
shows *clmm-dsc P* **using** *assms* **unfolding** *clmm-dsc-def* **by** *simp*

lemma *clmm-dsc-span-grid*:
assumes *clmm-dsc P*
shows *span-grid P* **using** *assms* **unfolding** *clmm-dsc-def* **by** *simp*

lemma *clmm-dsc-grid*[*simp*]:
assumes *clmm-dsc P*
shows *strict-mono* (*grd P*) ($\forall$ *i*. *0* $<$ *grd P i*)
($\forall$ *r*$>$*0*. $\exists$ *i*. *grd P i* $<$ *r*) ($\forall$ *r*. $\exists$ *i*. *r* $<$ *grd P i*)
using *assms* **unfolding** *clmm-dsc-def span-grid-def* **by** *simp*+

lemma *clmm-dsc-grd-Suc*:
assumes *clmm-dsc P*
shows *grd P i* $<$ *grd P* (*i*+*1*) **using** *assms clmm-dsc-grid*(*1*) *strict-mono-def*
by (*simp add*: *strict-mono-less*)

lemma *clmm-dsc-grd-smono*:

assumes *clmm-dsc P*
and $i < j$
shows *grd P i* $<$ *grd P j* **using** *assms clmm-dsc-grid*(*1*)
by (*simp add*: *strict-monoD*)

lemma *clmm-dsc-grd-mono*:
assumes *clmm-dsc P*
and $i \leq j$
shows *grd P i* $\leq$ *grd P j* **using** *assms clmm-dsc-grd-smono*
by (*metis linorder-not-less nle-le*)

lemma *clmm-dsc-liq*:
assumes *clmm-dsc P*
shows *finite-liq P 0* $\leq$ *lq P i* **using** *assms* **unfolding** *clmm-dsc-def* **by** *simp+*

lemma *clmm-dsc-fees*:
assumes *clmm-dsc P*
shows $(\forall\, i.\ 0 \leq fee\ P\ i) \wedge (\forall\, i.\ fee\ P\ i < 1)$ **using** *assms*
unfolding *clmm-dsc-def* **by** *simp*

lemma *clmm-dsc-fees-neq-1*:
assumes *clmm-dsc P*
shows $\forall\, i.\ fee\ P\ i \neq 1$
by (*metis assms clmm-dsc-def less-numeral-extra*(*4*))

lemma *clmm-dsc-gross-liq*:
assumes *clmm-dsc P*
shows *nz-support* (*gross-fct* (*lq P*) (*fee P*)) = *nz-support* (*lq P*)
using *gross-nz-support-eq clmm-dsc-fees assms*
by (*metis less-numeral-extra*(*4*))

lemma *clmm-dsc-gross-liq-zero-iff*:
assumes *clmm-dsc P*
shows (*lq P i* = *0*) $\longleftrightarrow$ (*gross-fct* (*lq P*) (*fee P*) *i* = *0*)
by (*simp add*: *assms clmm-dsc-fees-neq-1 gross-fct-nz-eq*)

lemma *gross-liq-gt*:
assumes *clmm-dsc P*
and *lq P i* $\neq$ *0*
and *L* = *gross-fct* (*lq P*) (*fee P*)
shows *0* $<$ *L i* **using** *assms*
by (*metis clmm-dsc-fees clmm-dsc-liq*(*2*) *dual-order.irrefl*
gross-fct-nz-eq gross-fct-sgn order.order-iff-strict)

lemma *gross-liq-ge*:
assumes *clmm-dsc P*
and *L* = *gross-fct* (*lq P*) (*fee P*)
shows *0* $\leq$ *L i* **using** *assms*
by (*meson clmm-dsc-fees clmm-dsc-liq*(*2*) *gross-fct-sgn*)

lemma *rng-quote-net-ge*:
assumes *clmm-dsc P*
shows *0* ≤ *lq P i* ∗ (*grd P* (*i*+*1*) − *grd P i*)
by (*simp add*: *assms clmm-dsc-grd-mono clmm-dsc-liq*(*2*))

lemma *rng-quote-gross-ge*:
assumes *clmm-dsc P*
and *L* = *gross-fct* (*lq P*) (*fee P*)
shows *0* ≤ *L i* ∗ (*grd P* (*i*+*1*) − *grd P i*)
using *assms clmm-dsc-grd-mono gross-liq-ge* **by** *auto*

lemma *clmm-quote-gross-pos*:
assumes *clmm-dsc P*
shows *0* ≤ *quote-gross P sqp* **using** *quote-gross-pos assms*
by (*meson clmm-dsc-fees clmm-dsc-grd-mono clmm-dsc-liq*(*2*) *gross-fct-sgn*
zle-add1-eq-le zless-add1-eq)

lemma *clmm-quote-gross-mono*:
assumes *clmm-dsc P*
shows *mono* (*quote-gross P*)
proof −
interpret *finite-liq-pool P*
by (*simp add*: *assms clmm-dsc-liq*(*1*) *finite-liq-pool-def*)
show *?thesis*
proof (*rule quote-gross-mono-finite*)
show ∀ *i*. *0* ≤ *lq P i* **using** *assms clmm-dsc-liq* **by** *simp*
show ∀ *i*. *fee P i* < *1* **using** *assms clmm-dsc-fees* **by** *simp*
show ∀ *i*. *grd P i* ≤ *grd P* (*i* + *1*) **using** *assms clmm-dsc-grid span-gridD*
by (*simp add*: *strict-mono-leD*)
qed
qed

lemma *quote-gross-imp-sqp-lt*:
assumes *clmm-dsc P*
and *quote-gross P sqp* < *quote-gross P sqp′*
shows *sqp* < *sqp′*
using *assms clmm-quote-gross-mono mono-strict-invE* **by** *blast*

lemma *clmm-quote-net-mono*:
assumes *clmm-dsc P*
shows *mono* (*quote-net P*)
proof −
interpret *finite-liq-pool P*
by (*simp add*: *assms clmm-dsc-liq*(*1*) *finite-liq-pool-def*)
show *?thesis*
proof (*rule quote-net-mono-finite-liq*)
show ∀ *i*. *0* ≤ *lq P i* **using** *assms clmm-dsc-liq* **by** *simp*
show ∀ *i*. *fee P i* < *1* **using** *assms clmm-dsc-fees* **by** *simp*

```
    show ∀ i. grd P i ≤ grd P (i + 1) using assms clmm-dsc-grid span-gridD
      by (simp add: strict-mono-leD)
  qed
qed

lemma clmm-base-gross-antimono:
  assumes clmm-dsc P
  shows antimono (base-gross P)
proof −
  interpret finite-liq-pool P
    by (simp add: assms clmm-dsc-liq(1) finite-liq-pool-def)
  show ?thesis
  proof (rule base-gross-antimono-finite)
    show ∀ i. 0 ≤ lq P i using assms clmm-dsc-liq by simp
    show ∀ i. fee P i < 1 using assms clmm-dsc-fees by simp
    show ∀ i. grd P i ≤ grd P (i + 1) using assms clmm-dsc-grid span-gridD
      by (simp add: strict-mono-leD)
    show ∀ i. 0 < grd P i using assms clmm-dsc-grid span-gridD by simp
  qed
qed

lemma clmm-base-net-antimono:
  assumes clmm-dsc P
  shows antimono (base-net P)
proof −
  interpret finite-liq-pool P
    by (simp add: assms clmm-dsc-liq(1) finite-liq-pool-def)
  show ?thesis
  proof (rule base-net-antimono-finite)
    show ∀ i. 0 ≤ lq P i using assms clmm-dsc-liq by simp
    show ∀ i. grd P i ≤ grd P (i + 1) using assms clmm-dsc-grid span-gridD
      by (simp add: strict-mono-leD)
    show ∀ i. 0 < grd P i using assms clmm-dsc-grid span-gridD by simp
  qed
qed

lemma liq-grd-min:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
shows 0 < grd-min P using grd-min-pos assms by simp

lemma liq-grd-min-max:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
shows grd-min P < grd-max P
proof −
  have finite-liq-pool P
    by (simp add: assms(1) clmm-dsc-liq(1) finite-liq-pool.intro)
  hence idx-min (lq P) ≤ idx-max (lq P)
```

using *idx-min-max-finite assms clmm-dsc-def finite-liq-def* **by** *auto*
thus *?thesis* **using** *assms*
unfolding *grd-min-def grd-max-def idx-min-img-def idx-max-img-def*
by (*simp add*: *clmm-dsc-grd-smono*)
qed

definition *rng-blw* **where**
rng-blw P prc = {*i*. *grd P i* ≤ *prc*}

lemma *rng-blw-mem*[*simp*]:
assumes *i* ∈ *rng-blw P prc*
shows *grd P i* ≤ *prc* **using** *assms* **unfolding** *rng-blw-def* **by** *simp*

lemma *rng-blw-bdd-above*:
assumes *clmm-dsc P*
shows *bdd-above* (*rng-blw P prc*) **unfolding** *rng-blw-def*
proof −
have *span-grid P* **using** *assms clmm-dsc-def* **by** *simp*
thus *bdd-above* {*i*. *grd P i* ≤ *prc*} **unfolding** *bdd-above-def span-grid-def*
by (*metis dual-order.trans less-eq-real-def mem-Collect-eq*
strict-mono-less-eq)
qed

lemma *rng-blw-ne*:
assumes *clmm-dsc P*
and *0* < *prc*
shows *rng-blw P prc* ≠ {}
proof −
have ∃ *i*. *grd P i* < *prc* **using** *assms clmm-dsc-grid span-grid-def* **by** *simp*
thus *?thesis* **unfolding** *rng-blw-def* **using** *less-eq-real-def* **by** *auto*
qed

definition *lower-tick* **where**
lower-tick P prc = *Sup* (*rng-blw P prc*)

lemma *grd-lower-tick-cong*:
assumes *grd P1* = *grd P2*
shows *lower-tick P1 sqp* = *lower-tick P2 sqp*
using *assms* **unfolding** *lower-tick-def rng-blw-def* **by** *simp*

lemma *lower-tick-mem*:
assumes *clmm-dsc P*
and *0* < *prc*
shows *lower-tick P prc* ∈ *rng-blw P prc* **unfolding** *lower-tick-def*
proof (*rule int-set-bdd-above*)
show *rng-blw P prc* ≠ {} **using** *rng-blw-ne assms* **by** *simp*
show *bdd-above* (*rng-blw P prc*) **using** *rng-blw-bdd-above assms* **by** *simp*
qed

lemma *lower-tick-geq*:
assumes *clmm-dsc P*
and *0 < prc*
shows *grd P (lower-tick P prc) ≤ prc*
using *assms lower-tick-mem* **unfolding** *rng-blw-def* **by** *simp*

lemma *lower-tick-geq′*:
assumes *clmm-dsc P*
and *i ∈ rng-blw P prc*
shows *i ≤ lower-tick P prc* **unfolding** *lower-tick-def*
proof (*rule cSup-upper*)
show *i ∈ rng-blw P prc* **using** *assms* **by** *simp*
show *bdd-above* (*rng-blw P prc*) **using** *assms rng-blw-bdd-above* **by** *simp*
qed

lemma *lower-tick-ubound*:
assumes *clmm-dsc P*
and *i = lower-tick P prc*
shows *prc < grd P (i+1)*
proof (*rule ccontr*)
assume *¬ prc < grd P (i + 1)*
hence *grd P (i + 1) ≤ prc* **by** *simp*
hence *i+1 ∈* (*rng-blw P prc*) **unfolding** *rng-blw-def* **by** *auto*
hence *i+1 ≤ i* **using** *assms lower-tick-geq′* **by** *blast*
thus *False* **by** *simp*
qed

lemma *lower-tick-lbound*:
assumes *clmm-dsc P*
and *0 < prc*
and *i = lower-tick P prc*
shows *grd P i ≤ prc* **unfolding** *lower-tick-def*
proof −
have *lower-tick P prc ∈ rng-blw P prc* **unfolding** *lower-tick-def*
proof (*rule int-set-bdd-above*(*1*))
show *bdd-above* (*rng-blw P prc*) **using** *assms rng-blw-bdd-above* **by** *simp*
show *rng-blw P prc ≠ {}* **using** *assms rng-blw-ne* **by** *simp*
qed
thus *grd P i ≤ prc* **using** *assms* **by** *simp*
qed

lemma *lower-tick-lt*:
assumes *clmm-dsc P*
and *0 < sqp′*
and *i = lower-tick P sqp*
and *j = lower-tick P sqp′*
and *i < j*
shows *sqp < sqp′*
proof −

```
  have i+1 ≤ j using assms by simp
  have sqp < grd P (i+1) using assms lower-tick-ubound by simp
  also have ... ≤ grd P j using assms clmm-dsc-grid span-gridD(1)
    by (simp add: strict-mono-less-eq)
  also have ... ≤ sqp' using assms lower-tick-lbound by simp
  finally show ?thesis .
qed

lemma lower-tick-lt':
  assumes clmm-dsc P
  and 0 < sqp'
  and i = lower-tick P sqp
  and j = lower-tick P sqp'
  and sqp' < sqp
  and grd P i = sqp
shows j < i
proof −
  have j ≤ i using assms lower-tick-lt by fastforce
  moreover have j≠ i
  proof (rule ccontr)
    assume ¬ j ≠ i
    hence sqp ≤ sqp' using assms lower-tick-lbound by blast
    thus False using assms by simp
  qed
  ultimately show ?thesis by simp
qed

lemma lower-tick-mono:
  assumes clmm-dsc P
  and 0 < sqp
  and i = lower-tick P sqp
  and j = lower-tick P sqp'
  and sqp ≤ sqp'
shows i ≤ j
  using assms lower-tick-lt by fastforce

lemma lower-tick-eq:
  assumes clmm-dsc P
  and grd P i = sqp
shows lower-tick P sqp = i
proof −
  define j where j = lower-tick P sqp
  have i ∈ rng-blw P sqp using assms unfolding rng-blw-def by auto
  hence i ≤ j using assms lower-tick-geq' unfolding j-def by simp
  moreover have j ≤ i
  proof (rule ccontr)
    assume ¬ j≤ i
    hence i+1 ≤ j by simp
    have sqp = grd P i using assms by simp
```

```
    also have ... < grd P (i+1) using assms clmm-dsc-grd-Suc by blast
    also have ... ≤ grd P j
      using ‹i+1 ≤ j› by (simp add: assms(1) clmm-dsc-grd-mono)
    also have ... ≤ sqp
      using assms clmm-dsc-grid(2) j-def lower-tick-lbound by blast
    finally have sqp < sqp .
    thus False by simp
  qed
  ultimately show j = i by simp
qed

lemma lower-tick-charact:
  assumes clmm-dsc P
  and grd P i ≤ sqp
  and sqp < grd P (i+1)
shows lower-tick P sqp = i
proof (rule ccontr)
  assume lower-tick P sqp ≠ i
  hence i < lower-tick P sqp
    by (metis assms(1,2) clmm-dsc-grid(2) lower-tick-eq lower-tick-mono
        order-le-imp-less-or-eq)
  hence i+1 ≤ lower-tick P sqp by simp
  hence grd P (i+1) ≤ grd P (lower-tick P sqp)
    by (simp add: assms(1) clmm-dsc-grd-mono)
  also have ... ≤ sqp
    by (meson assms(1,2) clmm-dsc-grid(2) lower-tick-lbound order-less-le-trans)
  finally have grd P (i+1) ≤ sqp .
  thus False using assms by simp
qed

lemma lower-tick-grd-min:
  assumes strict-mono (grd P)
shows idx-min (lq P) = lower-tick P (grd-min P)
proof −
  define A where A = {i. grd P i ≤ grd P (idx-min (lq P))}
  have idx-min (lq P) ∈ A using A-def by simp
  moreover have ∀ i∈ A. i ≤ idx-min (lq P) unfolding A-def using assms
    by (simp add: strict-mono-less-eq)
  ultimately show ?thesis
    unfolding A-def lower-tick-def rng-blw-def grd-min-def idx-min-img-def
    by (metis cSup-eq-maximum)
qed

lemma lower-tick-grd-max:
  assumes strict-mono (grd P)
  shows idx-max (lq P) + 1 = lower-tick P (grd-max P)
proof −
  define A where A = {i. grd P i ≤ grd P (idx-max (lq P) + 1)}
  have idx-max (lq P) + 1 ∈ A using A-def by simp
```

```
  moreover have ∀ i∈ A. i ≤ idx-max (lq P) + 1 unfolding A-def using assms
    by (simp add: strict-mono-less-eq)
  ultimately show ?thesis
    unfolding A-def lower-tick-def rng-blw-def grd-max-def idx-max-img-def
    by (metis cSup-eq-maximum)
qed

lemma grd-max-gt-if:
  assumes clmm-dsc P
  and i = lower-tick P sqp
  and lq P i ≠ 0
shows sqp < grd-max P
proof −
  have fin: finite (nz-support (lq P))
    by (meson assms(1) clmm-dsc-liq(1) finite-liq-def)
  have sqp < grd P (i+1) using assms lower-tick-ubound by auto
  also have ... ≤ grd-max P
  proof −
    have i ≤ idx-max (lq P) using assms
      by (simp add: fin idx-max-finite-ge)
    thus ?thesis unfolding grd-max-def idx-max-img-def
      by (simp add: assms(1) clmm-dsc-grd-mono)
  qed
  finally show ?thesis .
qed
```

4.2 Quote token addition and withdrawal in a CLMM

```
lemma (in finite-nz-support) clmm-gen-quote-sum:
  assumes clmm-dsc P
  and 0 < sqp
  and j = lower-tick P sqp
shows gen-quote (grd P) L sqp =
    L j * (sqp − grd P j) +
    sum (λ i. L i * (grd P (i+1) − grd P i)) {i. L i ≠ 0 ∧ i < j}
proof −
  define df where df = gamma-min-diff (grd P)
  define A where A = {i. L i ≠ 0 ∧ i ≤ j}
  define B where B = {i. L i ≠ 0 ∧ j < i}
  define C where C = {i. L i ≠ 0 ∧ i < j}
  have un: {i. L i ≠ 0} = A ∪ B unfolding A-def B-def by auto
  have inter: A ∩ B = {} unfolding A-def B-def by auto
  have fin: finite {i. L i ≠ 0} by (metis fin-nz-sup nz-support-def)
  have gen-quote (grd P) L sqp =
      sum (rng-token df L sqp) {i. L i ≠ 0}
    unfolding gen-quote-def df-def using gen-token-sum by simp
  also have ... = sum (rng-token df L sqp) A + sum (rng-token df L sqp) B
    by (metis empty-iff fin finite-Un inter sum.union-inter-neutral un)
  also have ... = sum (rng-token df L sqp) A
```

proof −
have ∀ *i*∈ *B*. *rng-token df L sqp i* = *0*
proof
fix *i*
assume *i* ∈ *B*
have *grd P i* < *grd P* (*i*+*1*) **using** *assms clmm-dsc-grid span-gridD*
by (*simp add*: *strict-mono-less*)
have *j* + *1* ≤ *i* **using** ‹*i* ∈ *B*› *assms* **unfolding** *B-def* **by** *simp*
hence *grd P* (*j* +*1*) ≤ *grd P i*
using *assms clmm-dsc-grid span-gridD*
by (*simp add*: *strict-mono-leD*)
hence *sqp* < *grd P i* **using** *lower-tick-ubound*[*of P*] *assms*
by (*metis dual-order.strict-trans nless-le*)
hence *sqp* < *grd P* (*i*+*1*) **using** ‹*grd P i* < *grd P* (*i*+*1*)› **by** *simp*
hence *df sqp i* = *0*
using ‹*sqp* < *grd P i*› **unfolding** *df-def gamma-min-diff-def* **by** *simp*
thus *rng-token df L sqp i* = *0* **unfolding** *rng-token-def* **by** *simp*
qed
thus *?thesis* **by** *simp*
qed
also have ... = *rng-token df L sqp j* + *sum* (*rng-token df L sqp*) *C*
proof (*cases j* ∈ *A*)
case *True*
hence *A* = {*j*} ∪ *C* **unfolding** *A-def C-def* **by** *auto*
hence *sum* (*rng-token df L sqp*) *A* = *sum* (*rng-token df L sqp*) ({*j*} ∪ *C*)
by *simp*
also have ... = *sum* (*rng-token df L sqp*) {*j*} + *sum* (*rng-token df L sqp*) *C*
proof (*rule sum.union-inter-neutral*)
show *finite* {*j*} **by** *simp*
show *finite C* **using** *fin C-def* **by** *simp*
show ∀ *x*∈{*j*} ∩ *C*. *rng-token df L sqp x* = *0* **using** *C-def* **by** *simp*
qed
also have ... = *rng-token df L sqp j* + *sum* (*rng-token df L sqp*) *C* **by** *simp*
finally show *?thesis* **.**
next
case *False*
hence *A* = *C* **unfolding** *A-def C-def* **using** *Collect-cong* **by** *fastforce*
have *L j* = *0* **using** *False A-def* **by** *auto*
hence *rng-token df L sqp j* = *0* **unfolding** *rng-token-def* **by** *simp*
then show *?thesis* **using** ‹*A* = *C*› **by** *simp*
qed
also have ... = *L j* ∗ (*sqp* − *grd P j*) + *sum* (*rng-token df L sqp*) *C*
proof −
have *min sqp* (*grd P* (*j* + *1*)) = *sqp* **using** *assms lower-tick-ubound* **by** *simp*
moreover have *min sqp* (*grd P j*) = *grd P j*
using *assms lower-tick-lbound* **by** *simp*
ultimately have *rng-token df L sqp j* = *L j* ∗ (*sqp* − *grd P j*)
unfolding *rng-token-def df-def gamma-min-diff-def* **by** *simp*
thus *?thesis* **by** *simp*

```
  qed
  also have ... = L j * (sqp - grd P j) + sum (λ i. L i * (grd P (i+1) - grd P
i)) C
  proof -
    have ∀ i ∈ C. rng-token df L sqp i = L i * (grd P (i+1) - grd P i)
    proof
      fix i
      assume i ∈ C
      hence i+1 ≤ j unfolding C-def by auto
      hence grd P (i+1) ≤ grd P j using assms clmm-dsc-grid(1) strict-monoD
        by (metis linorder-le-less-linear nless-le)
      hence grd P (i+1) ≤ sqp using lower-tick-lbound assms by fastforce
      have grd P i < grd P (i+1) using assms clmm-dsc-grd-Suc by simp
      hence grd P i ≤ sqp using ‹grd P (i+1) ≤ sqp› by simp
      thus rng-token df L sqp i = L i * (grd P (i+1) - grd P i)
        unfolding df-def rng-token-def gamma-min-diff-def
        using ‹grd P (i+1) ≤ sqp› by simp
    qed
    hence sum (rng-token df L sqp) C =
        sum (λ i. L i * (grd P (i+1) - grd P i)) C  by simp
    thus ?thesis unfolding df-def gamma-min-diff-def rng-token-def by simp
  qed
  finally show gen-quote (grd P) L sqp =
     L j * (sqp - grd P j) + sum (λ i. L i * (grd P (i+1) - grd P i)) C .
qed

lemma clmm-gen-quote-grd-min:
  assumes clmm-dsc P
  and nz-support L ≠ {}
  and finite (nz-support L)
  and nz-support L = nz-support (lq P)
shows gen-quote (grd P) L (grd-min P) = 0 using gen-quote-grd-min
  by (meson assms clmm-dsc-grd-mono mono-onI)

lemma (in finite-nz-support) clmm-gen-quote-grd-min-le:
  assumes clmm-dsc P
  and nz-support L = nz-support (lq P)
  and sqp ≤ grd-min P
  and 0 < sqp
shows gen-quote (grd P) L sqp = 0
proof -
  define j where j = lower-tick P sqp
  hence j ≤ idx-min (lq P) using lower-tick-grd-min[of P] assms
    by (simp add: lower-tick-mono)
  have gen-quote (grd P) L sqp =
     L j * (sqp - grd P j) +
     (∑ i | L i ≠ 0 ∧ i < j. L i * (grd P (i + 1) - grd P i))
    by (rule clmm-gen-quote-sum, (auto simp add: assms j-def))
  also have ... = L j * (sqp - grd P j)
```

```
  proof −
    have {i. L i ≠ 0 ∧ i < j} = {}
    proof −
      have ∀ i < j. L i = 0 using ‹j <= idx-min (lq P)› idx-min-def
        by (metis (mono-tags, opaque-lifting) assms(2)
            dual-order.strict-trans fin-nz-sup idx-min-finite-le nless-le)
      thus ?thesis by auto
    qed
    hence (∑ i | L i ≠ 0 ∧ i < j. L i * (grd P (i + 1) − grd P i)) = 0
      by (metis sum-clauses(1))
    thus ?thesis by simp
  qed
  also have ... = 0
  proof (cases sqp = grd-min P)
    case True
    hence grd P j = grd-min P
      using ‹j ≤ idx-min (lq P)› assms unfolding grd-min-def idx-min-img-def
      by (simp add: j-def lower-tick-eq)
    thus ?thesis using True by simp
  next
    case False
    hence j < idx-min (lq P) using lower-tick-lt'
      by (metis ‹j ≤ idx-min (lq P)› assms(1) assms(4) assms(3)
          idx-min-img-def j-def leI lower-tick-lbound grd-min-def
          verit-la-disequality)
    hence L j = 0 unfolding idx-min-def nz-support-def
      by (metis ‹j < idx-min (lq P)› assms(2) fin-nz-sup idx-min-def
          idx-min-finite-le leD)
    then show ?thesis by simp
  qed
  finally show ?thesis .
qed

lemma clmm-quote-gross-sum:
  assumes clmm-dsc P
  and 0 < sqp
  and L = gross-fct (lq P) (fee P)
  and j = lower-tick P sqp
shows quote-gross P sqp =
    L j * (sqp − grd P j) +
    sum (λ i. L i * (grd P (i+1) − grd P i)) {i. L i ≠ 0 ∧ i < j}
proof −
  interpret finite-nz-support L
    using finite-liq-pool.finite-liq-gross-fct assms
    by (metis clmm-dsc-liq(1) finite-liq-pool.intro finite-nz-support-def
        nz-support-def)
  show ?thesis unfolding quote-gross-def using clmm-gen-quote-sum assms by
simp
qed
```

lemma *clmm-quote-gross-grd-min*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
shows *quote-gross P* (*grd-min P*) = *0* **unfolding** *quote-gross-def*
proof (*rule clmm-gen-quote-grd-min*)
show *finite* (*nz-support* (*gross-fct* (*lq P*) (*fee P*)))
using *finite-liq-pool.finite-liq-gross-fct assms*
by (*metis clmm-dsc-liq*(*1*) *finite-liq-pool.intro nz-support-def*)
show *clmm-dsc P* **using** *assms* **by** *simp*
show *nz-support* (*gross-fct* (*lq P*) (*fee P*)) = *nz-support* (*lq P*)
using *clmm-dsc-gross-liq assms* **by** *simp*
thus *nz-support* (*gross-fct* (*lq P*) (*fee P*)) ≠ {} **using** *assms* **by** *simp*
qed

lemma *clmm-quote-gross-grd-min-le*:
assumes *clmm-dsc P*
and *sqp* ≤ *grd-min P*
and *0* < *sqp*
shows *quote-gross P sqp* = *0* **unfolding** *quote-gross-def*
proof (*rule finite-nz-support.clmm-gen-quote-grd-min-le*)
show *finite-nz-support* (*gross-fct* (*lq P*) (*fee P*))
using *finite-liq-pool.finite-liq-gross-fct assms*
by (*metis clmm-dsc-liq*(*1*) *finite-liq-pool.intro finite-nz-support-def*
nz-support-def)
show *clmm-dsc P* **using** *assms* **by** *simp*
show *nz-support* (*gross-fct* (*lq P*) (*fee P*)) = *nz-support* (*lq P*)
using *clmm-dsc-gross-liq assms* **by** *simp*
qed (*simp add*: *assms*)+

lemma *clmm-quote-reach-zero*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
shows *quote-reach P 0* = *grd-min P*
using *assms clmm-quote-gross-grd-min* **unfolding** *quote-reach-def* **by** *simp*

lemma *clmm-quote-reach-ge*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *0* ≤ *y*
and *y* ≤ *quote-gross P* (*grd-max P*)
shows *grd-min P* ≤ (*quote-reach P y*)
proof −
interpret *finite-liq-pool*
by (*simp add*: *assms*(*1*) *clmm-dsc-liq*(*1*) *finite-liq-pool.intro*)
show *?thesis*
proof (*cases y* = *0*)
case *True*
then show *?thesis* **using** *assms clmm-quote-reach-zero* **by** *simp*

```
  next
    case False
    hence 0 < y using assms by simp
    then show ?thesis using assms quote-reach-ge
     by (simp add: clmm-dsc-fees clmm-dsc-liq(2) grd-min-max strict-mono-mono)
  qed
qed

lemma clmm-quote-reach-pos:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and 0 ≤ y
  and y ≤ quote-gross P (grd-max P)
  and sqp = quote-reach P y
shows 0 < sqp
proof −
  have 0 < grd-min P by (simp add: assms liq-grd-min)
  thus 0 < sqp using assms clmm-quote-reach-ge by fastforce
qed

lemma clmm-quote-reach-mem:
  assumes clmm-dsc P
  and 0 ≤ y
  and y ≤ quote-gross P (grd-max P)
  and nz-support (lq P) ≠ {}
shows quote-reach P y ∈ quote-gross P−‘ {y}
proof −
  interpret finite-liq-pool
    by (simp add: assms(1) clmm-dsc-liq(1) finite-liq-pool.intro)
  show ?thesis
  proof (rule quote-reach-mem)
    show ∀ i. 0 ≤ lq P i using assms(1) clmm-dsc-def by simp
    show ∀ i. fee P i < 1 using clmm-dsc-fees assms by simp
    show mono (grd P)
      by (simp add: assms(1) clmm-dsc-grd-mono monoI)
    show 0 ≤ y using assms by simp
    show y ≤ quote-gross P (grd-max P) using assms by simp
  qed
qed

lemma clmm-quote-reach-le:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and 0 < y
  and sqp ∈ quote-gross P −‘{y}
  and sqp′ = quote-reach P y
shows sqp′ ≤ sqp
proof −
  interpret finite-liq-pool
```

by (*simp add*: *assms*(*1*) *clmm-dsc-liq*(*1*) *finite-liq-pool.intro*)
have *qs*: *quote-gross P* (*grd-min P*) = *0*
using *assms clmm-quote-gross-grd-min* **by** *simp*
define *sqp′* **where** *sqp′* = *quote-reach P y*
define *X* **where** *X* = *quote-gross P* −‘ {*y*}
hence *sqp′* = *Inf X*
using *assms qs* **unfolding** *sqp′-def quote-reach-def* **by** *simp*
have ∀ *x*∈ *X*. *Inf X* ≤ *x*
proof
fix *x*
assume *x* ∈ *X*
show *Inf X* ≤ *x*
proof (*rule cInf-lower*)
show *x*∈ *X* **using** ‹*x*∈ *X*› **.**
show *bdd-below X*
using *assms quote-gross-bdd-below X-def clmm-quote-gross-mono qs* **by** *simp*
qed
qed
thus *?thesis* **using** *assms X-def* ‹*sqp′* = *Inf X*› *sqp′-def* **by** *auto*
qed

lemma *clmm-quote-net-sum*:
assumes *clmm-dsc P*
and *0* < *sqp*
and *L* = *lq P*
and *j* = *lower-tick P sqp*
shows *quote-net P sqp* =
L j ∗ (*sqp* − *grd P j*) +
sum (λ *i*. *L i* ∗ (*grd P* (*i*+*1*) − *grd P i*)) {*i*. *L i* ≠ *0* ∧ *i* < *j*}
proof −
interpret *finite-nz-support L*
using *finite-liq-pool.finite-liq-gross-fct assms clmm-dsc-def*
finite-liq-def finite-nz-support-def
by *blast*
show *?thesis* **unfolding** *quote-net-def* **using** *clmm-gen-quote-sum assms* **by** *simp*
qed

lemma *clmm-quote-net-grd-min*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
shows *quote-net P* (*grd-min P*) = *0* **unfolding** *quote-net-def*
proof (*rule clmm-gen-quote-grd-min*)
show *finite* (*nz-support* (*lq P*))
using *clmm-dsc-liq finite-liqD assms*
unfolding *finite-nz-support-def nz-support-def* **by** *simp*
qed (*auto simp add*: *assms*)

lemma *clmm-quote-gross-reach-eq*:
assumes *clmm-dsc P*

and *nz-support* (*lq P*) ≠ {}
and *0* ≤ *y*
and *y* ≤ *quote-gross P* (*grd-max P*)
shows *quote-gross P* (*quote-reach P y*) = *y*
proof −
interpret *finite-liq-pool*
by (*simp add*: *assms*(*1*) *clmm-dsc-liq*(*1*) *finite-liq-pool.intro*)
show *?thesis*
proof (*rule quote-gross-reach-eq*)
show ∀ *i*. *0* ≤ *lq P i* **by** (*simp add*: *assms*(*1*) *clmm-dsc-liq*(*2*))
show ∀ *i*. *fee P i* < *1* **by** (*simp add*: *assms*(*1*) *clmm-dsc-fees*)
show *mono* (*grd P*)
by (*simp add*: *assms*(*1*) *clmm-dsc-grd-mono monoI*)
show *0* ≤ *y* **using** *assms* **by** *simp*
show *y* ≤ *quote-gross P* (*grd-max P*) **using** *assms* **by** *simp*
qed
qed

definition *gen-quote-diff* **where**
gen-quote-diff P L sqp sqp′ = *gen-quote* (*grd P*) *L sqp′* − *gen-quote* (*grd P*) *L sqp*

lemma (**in** *finite-nz-support*) *clmm-gen-quote-diff-eq*:
assumes *clmm-dsc P*
and *j* = *lower-tick P sqp*
and *k* = *lower-tick P sqp′*
and *0* < *sqp*
and *sqp* ≤ *sqp′*
and *j* < *k*
shows *gen-quote-diff P L sqp sqp′* = *L k* ∗ (*sqp′* − *grd P k*) +
sum (λ *i*. *L i* ∗ (*grd P* (*i*+*1*) − *grd P i*)) {*i*. *L i* ≠ *0* ∧ *j* <*i* ∧ *i* < *k*} +
L j ∗ (*grd P* (*j*+*1*) − *sqp*)
proof −
have *0* < *sqp* **using** *assms* **by** *simp*
hence *sqp* < *sqp′* **using** *lower-tick-lt assms* **by** *simp*
hence *0* < *sqp′* **using** ‹*0* < *sqp*› **by** *simp*
define *A* **where** *A* = {*i*. *L i* ≠ *0* ∧ *i* < *k*}
define *B* **where** *B* = {*i*. *L i* ≠ *0* ∧ *i* < *j*}
define *C* **where** *C* = {*i*. *L i* ≠ *0* ∧ *j* ≤*i* ∧ *i* < *k*}
define *Cj* **where** *Cj* = {*i*. *L i* ≠ *0* ∧ *j* <*i* ∧ *i* < *k*}
define *df* **where** *df* = (λ *i*. *L i* ∗ (*grd P* (*i*+*1*) − *grd P i*))
have *finite A*
unfolding *A-def* **by** (*metis fin-nz-sup finite-Collect-conjI nz-support-def*)
have *A* = *B* ∪ *C* **using** *assms* **unfolding** *A-def B-def C-def* **by** *auto*
have *Cj* = *C* − {*j*} **unfolding** *C-def Cj-def* **by** *auto*
have *gen-quote-diff P L sqp sqp′* = *L k* ∗ (*sqp′* − *grd P k*) +
sum df A − (*L j* ∗ (*sqp* − *grd P j*) + *sum df B*)
using *assms* ‹*0* < *sqp*› *clmm-gen-quote-sum* ‹*0* < *sqp′*›
unfolding *gen-quote-diff-def df-def A-def B-def* **by** *simp*
also have ... = *L k* ∗ (*sqp′* − *grd P k*) + *sum df C* − *L j* ∗ (*sqp* − *grd P j*)

```
  proof (rule diff-sum-dcomp)
    show finite A using ‹finite A› .
    show A = B ∪ C using ‹A = B ∪ C› .
    show B ∩ C = {} unfolding B-def C-def by auto
  qed
  also have ... =  L k * (sqp' − grd P k) + sum df Cj +
      df j − L j * (sqp − grd P j)
  proof (cases j ∈ C)
    case True
    have sum df C = df j + sum df Cj
    proof (rule sum-remove-el)
      show finite C using ‹A = B ∪ C› ‹finite A› by simp
      show j ∈ C using True .
      show Cj = C − {j} using ‹Cj = C − {j}› .
    qed
    then show ?thesis by simp
  next
    case False
    hence L j = 0 using assms unfolding C-def by auto
    hence df j = 0 unfolding df-def by simp
    moreover have Cj = C using False ‹Cj = C − {j}› by auto
    ultimately show ?thesis by simp
  qed
  also have ... = L k * (sqp' − grd P k) + sum df Cj +
      L j * (grd P (j+1) − sqp)
  proof −
    have df j − L j * (sqp − grd P j) = L j * (grd P (j+1) − sqp)
      unfolding df-def by (simp add: right-diff-distrib)
    thus ?thesis by simp
  qed
  finally show gen-quote-diff P L sqp sqp' = L k * (sqp' − grd P k) + sum df Cj
+
      L j * (grd P (j+1) − sqp) .
qed

lemma (in finite-nz-support) clmm-gen-quote-diff-eq':
  assumes clmm-dsc P
  and j = lower-tick P sqp
  and j = lower-tick P sqp'
  and 0 < sqp
  and sqp ≤ sqp'
  and L' j = L j
shows gen-quote-diff P L sqp sqp' = L' j * (sqp' − sqp)
proof −
  have 0 < sqp using assms liq-grd-min[of P] by simp
  hence 0 < sqp' using assms by simp
  define A where A = {i. L i ≠ 0 ∧ i < j}
  define df where df = (λ i. L i * (grd P (i+1) − grd P i))
  have finite A
```

```
    unfolding A-def by (metis fin-nz-sup finite-Collect-conjI nz-support-def)
  have gen-quote-diff P L sqp sqp' = L j * (sqp' - grd P j) +
      sum df A - (L j * (sqp - grd P j) + sum df A)
    using assms ‹0 < sqp› clmm-gen-quote-sum ‹0 < sqp'›
    unfolding gen-quote-diff-def df-def A-def by simp
  also have ... = L j * (sqp' - grd P j) - L j * (sqp - grd P j) by simp
  also have ... = L j * (sqp' - sqp)
    by (simp add: right-diff-distrib)
  finally show gen-quote-diff P L sqp sqp' = L' j * (sqp' - sqp)
    using assms by simp
qed

lemma clmm-quote-gross-diff-eq:
  assumes clmm-dsc P
  and L = gross-fct (lq P) (fee P)
  and j = lower-tick P sqp
  and k = lower-tick P sqp'
  and 0 < sqp
  and sqp ≤ sqp'
  and j < k
shows quote-gross P sqp' - quote-gross P sqp = L k * (sqp' - grd P k) +
  sum (λ i. L i * (grd P (i+1) - grd P i)) {i. L i ≠ 0 ∧ j <i ∧ i < k} +
  L j * (grd P (j+1) - sqp)
proof -
  interpret finite-nz-support L
    using finite-liq-pool.finite-liq-gross-fct assms
    by (metis clmm-dsc-liq(1) finite-liq-pool.intro finite-nz-support-def
        nz-support-def)
  show ?thesis
    using clmm-gen-quote-diff-eq assms
    unfolding quote-gross-def gen-quote-diff-def by simp
qed

lemma clmm-rng-quote-strict-pos:
  assumes clmm-dsc P
  and L = gross-fct (lq P) (fee P)
  and L i ≠ 0
shows 0 < L i * (grd P (i+1) - grd P i) using assms
  by (metis add-0 clmm-dsc-grd-smono gross-liq-ge less-add-one less-diff-eq
      less-eq-real-def zero-less-mult-iff)

lemma clmm-sum-rng-quote-pos:
  assumes clmm-dsc P
  and L = gross-fct (lq P) (fee P)
shows 0 ≤ sum (λ i. L i * (grd P (i+1) - grd P i)) M
  using sum-nonneg rng-quote-gross-ge assms
  by (metis (mono-tags, lifting))

lemma clmm-sum-rng-quote-strict-pos:
```

assumes *clmm-dsc P*
and $L = gross\text{-}fct\ (lq\ P)\ (fee\ P)$
and $L\ i \neq 0$
and $i \in M$
and *finite M*
shows $0 < sum\ (\lambda\ i.\ L\ i * (grd\ P\ (i+1) - grd\ P\ i))\ M$
proof (*rule sum-pos2*)
show *finite M* **using** *assms* **by** *simp*
show $i \in M$ **using** *assms* **by** *simp*
show $0 < L\ i * (grd\ P\ (i + 1) - grd\ P\ i)$
using *assms clmm-rng-quote-strict-pos* **by** *simp*
show $\bigwedge i.\ i \in M \Longrightarrow 0 \leq L\ i * (grd\ P\ (i + 1) - grd\ P\ i)$
using *assms rng-quote-gross-ge* **by** *simp*
qed

lemma *clmm-quote-gross-eq-sum-only-if*:
assumes *clmm-dsc P*
and $L = gross\text{-}fct\ (lq\ P)\ (fee\ P)$
and $j = lower\text{-}tick\ P\ sqp$
and $k = lower\text{-}tick\ P\ sqp'$
and $0 < sqp$
and $sqp \leq sqp'$
and $j < k$
and $quote\text{-}gross\ P\ sqp' = quote\text{-}gross\ P\ sqp$
and $j < i$
and $i < k$
shows $L\ i = 0$
proof (*rule ccontr*)
assume $L\ i \neq 0$
define S **where** $S = L\ k * (sqp' - grd\ P\ k) +$
$sum\ (\lambda\ i.\ L\ i * (grd\ P\ (i+1) - grd\ P\ i))\ \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} +$
$L\ j * (grd\ P\ (j+1) - sqp)$
have $S = quote\text{-}gross\ P\ sqp' - quote\text{-}gross\ P\ sqp$
unfolding *S-def* **using** *clmm-quote-gross-diff-eq*[*OF assms*(1−7)] **by** *simp*
also have $... = 0$ **using** *assms* **by** *simp*
finally have $S = 0$ **.**
have $grd\ P\ k \leq sqp'$
using *assms lower-tick-mem* **by** *auto*
have $sqp < grd\ P\ (j+1)$
by (*metis assms*(1) *assms*(3) *lower-tick-ubound*)
have *a*: $0 \leq L\ k * (sqp' - grd\ P\ k)$
using ‹$grd\ P\ k \leq sqp'$› *clmm-dsc-liq*(2) *assms*
by (*simp add*: *gross-liq-ge*)
have *b*: $0 \leq L\ j * (grd\ P\ (j+1) - sqp)$
using ‹$sqp < grd\ P\ (j+1)$›
by (*metis assms*(1) *assms*(2) *diff-ge-0-iff-ge gross-liq-ge less-eq-real-def*
split-mult-pos-le)
have *c*: $0 \leq sum\ (\lambda\ i.\ L\ i * (grd\ P\ (i+1) - grd\ P\ i))$
$\{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\}$

using *assms clmm-sum-rng-quote-pos* **by** *simp*
hence *sum* (λ *i*. *L i* ∗ (*grd P* (*i*+1) − *grd P i*))
{*i*. *L i* ≠ *0* ∧ *j* <*i* ∧ *i* < *k*} = *0*
using *a b c* ‹*S* = *0*› *S-def* **by** *simp*
moreover have *0* < *sum* (λ *i*. *L i* ∗ (*grd P* (*i*+1) − *grd P i*))
{*i*. *L i* ≠ *0* ∧ *j* <*i* ∧ *i* < *k*}
proof (*rule clmm-sum-rng-quote-strict-pos*)
show *clmm-dsc P* **using** *assms* **by** *simp*
show *L* = *gross-fct* (*lq P*) (*fee P*) **using** *assms* **by** *simp*
show *L i* ≠ *0* **using** ‹*L i* ≠ *0*› **.**
thus *i* ∈ {*i*. *L i* ≠ *0* ∧ *j* < *i* ∧ *i* < *k*} **using** *assms* **by** *simp*
show *finite* {*i*. *L i* ≠ *0* ∧ *j* < *i* ∧ *i* < *k*} **by** *simp*
qed
ultimately show *False* **by** *simp*
qed

lemma *clmm-quote-gross-eq-sum-emp*:
assumes *clmm-dsc P*
and *L* = *gross-fct* (*lq P*) (*fee P*)
and *j* = *lower-tick P sqp*
and *k* = *lower-tick P sqp′*
and *0* < *sqp*
and *sqp* ≤ *sqp′*
and *j* < *k*
and *quote-gross P sqp′* = *quote-gross P sqp*
shows {*i*. *L i* ≠ *0* ∧ *j* <*i* ∧ *i* < *k*} = {}
proof (*rule ccontr*)
assume {*i*. *L i* ≠ *0* ∧ *j* <*i* ∧ *i* < *k*} ≠ {}
hence ∃ *i*. *L i* ≠ *0* ∧ *j* < *i* ∧ *i* < *k* **by** *auto*
from *this* **obtain** *i* **where** *L i* ≠ *0* **and** *j* < *i* **and** *i* < *k* **by** *auto*
hence *L i* = *0* **using** *assms clmm-quote-gross-eq-sum-only-if* **by** *simp*
thus *False* **using** ‹*L i* ≠ *0*› **by** *simp*
qed

lemma *clmm-quote-gross-eq-lower-only-if*:
assumes *clmm-dsc P*
and *L* = *gross-fct* (*lq P*) (*fee P*)
and *j* = *lower-tick P sqp*
and *k* = *lower-tick P sqp′*
and *0* < *sqp*
and *sqp* ≤ *sqp′*
and *j* < *k*
and *quote-gross P sqp′* = *quote-gross P sqp*
shows *L j* = *0*
proof (*rule ccontr*)
assume *L j* ≠ *0*
define *S* **where** *S* = *L k* ∗ (*sqp′* − *grd P k*) +
sum (λ *i*. *L i* ∗ (*grd P* (*i*+1) − *grd P i*)) {*i*. *L i* ≠ *0* ∧ *j* <*i* ∧ *i* < *k*} +
L j ∗ (*grd P* (*j*+1) − *sqp*)

```
  have S = quote-gross P sqp' − quote-gross P sqp
    unfolding S-def using clmm-quote-gross-diff-eq[OF assms(1−7)] by simp
  also have ... = 0 using assms by simp
  finally have S = 0 .
  have grd P k ≤ sqp'
    using assms lower-tick-mem by auto
  have sqp < grd P (j+1)
    by (metis assms(1) assms(3) lower-tick-ubound)
  have a: 0 ≤ L k ∗ (sqp' − grd P k)
    using ‹grd P k ≤ sqp'› clmm-dsc-liq(2) assms
    by (simp add: gross-liq-ge)
  have b: 0 ≤ L j ∗ (grd P (j+1) − sqp)
    using ‹sqp < grd P (j+1)›
    by (metis assms(1) assms(2) diff-ge-0-iff-ge gross-liq-ge less-eq-real-def
        split-mult-pos-le)
  have c: 0 ≤ sum (λ i. L i ∗ (grd P (i+1) − grd P i))
    {i. L i ≠ 0 ∧ j <i ∧ i < k}
    using assms clmm-sum-rng-quote-pos by simp
  hence L j ∗ (grd P (j+1) − sqp) = 0
    using a b c ‹S = 0› S-def by linarith
  moreover have 0 < L j ∗ (grd P (j+1) − sqp)
    using ‹L j ≠ 0› ‹sqp < grd P (j+1)› calculation by auto
  ultimately show False by linarith
qed

lemma clmm-quote-gross-eq-upper-only-if:
  assumes clmm-dsc P
  and L = gross-fct (lq P) (fee P)
  and j = lower-tick P sqp
  and k = lower-tick P sqp'
  and 0 < sqp
  and sqp ≤ sqp'
  and j < k
  and quote-gross P sqp' = quote-gross P sqp
shows L k = 0 ∨ grd P k = sqp'
proof (rule ccontr)
  assume asm: ¬ (L k = 0 ∨ grd P k = sqp')
  hence L k ≠ 0 by simp
  have grd P k ≠ sqp using asm
    using assms lower-tick-eq by fastforce
  define S where S = L k ∗ (sqp' − grd P k) +
  sum (λ i. L i ∗ (grd P (i+1) − grd P i)) {i. L i ≠ 0 ∧ j <i ∧ i < k} +
  L j ∗ (grd P (j+1) − sqp)
  have S = quote-gross P sqp' − quote-gross P sqp
    unfolding S-def using clmm-quote-gross-diff-eq[OF assms(1−7)] by simp
  also have ... = 0 using assms by simp
  finally have S = 0 .
  have grd P k < sqp'
    using assms lower-tick-mem ‹grd P k ≠ sqp›
```

by (*metis asm linorder-not-less nle-le order.trans rng-blw-mem*)
have *sqp* < *grd P* (*j*+*1*)
by (*metis assms*(*1*) *assms*(*3*) *lower-tick-ubound*)
have *a*: *0* ≤ *L k* ∗ (*sqp*′ − *grd P k*)
using ‹*grd P k* < *sqp*′› *clmm-dsc-liq*(*2*) *assms*
by (*simp add*: *gross-liq-ge*)
have *b*: *0* ≤ *L j* ∗ (*grd P* (*j*+*1*) − *sqp*)
using ‹*sqp* < *grd P* (*j*+*1*)›
by (*metis assms*(*1*) *assms*(*2*) *diff-ge-0-iff-ge gross-liq-ge less-eq-real-def split-mult-pos-le*)
have *c*: *0* ≤ *sum* (λ *i*. *L i* ∗ (*grd P* (*i*+*1*) − *grd P i*)) {*i*. *L i* ≠ *0* ∧ *j* <*i* ∧ *i* < *k*}
using *assms clmm-sum-rng-quote-pos* **by** *simp*
hence *L k* ∗ (*sqp*′ − *grd P k*) = *0*
using *a b c* ‹*S* = *0*› *S-def* **by** *linarith*
moreover have *0* < *L k* ∗ (*sqp*′ − *grd P k*)
using ‹*L k* ≠ *0*› ‹*sqp* < *grd P* (*j*+*1*)› *calculation asm* **by** *force*
ultimately show *False* **by** *linarith*
qed

lemma *clmm-quote-gross-diff-eq*′:
assumes *clmm-dsc P*
and *L* = *gross-fct* (*lq P*) (*fee P*)
and *j* = *lower-tick P sqp*
and *j* = *lower-tick P sqp*′
and *0* < *sqp*
and *sqp* ≤ *sqp*′
shows *quote-gross P sqp*′ − *quote-gross P sqp* = *L j* ∗ (*sqp*′ − *sqp*)
proof −
interpret *finite-nz-support L*
using *finite-liq-pool.finite-liq-gross-fct assms*
by (*metis clmm-dsc-liq*(*1*) *finite-liq-pool.intro finite-nz-support-def nz-support-def*)
show *?thesis* **using** *clmm-gen-quote-diff-eq*′ *assms*
unfolding *quote-gross-def gen-quote-diff-def* **by** *simp*
qed

lemma *clmm-quote-gross-eq-lower-only-if*′:
assumes *clmm-dsc P*
and *L* = *gross-fct* (*lq P*) (*fee P*)
and *j* = *lower-tick P sqp*
and *j* = *lower-tick P sqp*′
and *0* < *sqp*
and *sqp* < *sqp*′
and *quote-gross P sqp*′ = *quote-gross P sqp*
shows *L j* = *0*
proof −
have *L j* ∗ (*sqp*′ − *sqp*) = *quote-gross P sqp*′ − *quote-gross P sqp*
using *assms clmm-quote-gross-diff-eq*′[*OF assms*(*1*−*4*)] **by** *simp*

```
  also have ... = 0 using assms by simp
  finally have L j * (sqp' - sqp) = 0 .
  thus ?thesis using assms by simp
qed

lemma clmm-quote-reach-grd-liq:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and 0 < y
  and y ≤ quote-gross P (grd-max P)
  and j = lower-tick P sqp
  and grd P j = sqp
  and sqp = quote-reach P y
shows lq P (j - 1) ≠ 0
proof (rule ccontr)
  assume ¬ lq P (j - 1) ≠ 0
  define L where L = gross-fct (lq P) (fee P)
  have quote-gross P sqp = y using assms clmm-quote-gross-reach-eq by simp
  have quote-gross P sqp - quote-gross P (grd P (j-1)) =
      L j * (sqp - grd P j) +
      (∑ i | L i ≠ 0 ∧ j - 1 < i ∧ i < j. L i * (grd P (i + 1) - grd P i)) +
      L (j - 1) * (grd P (j - 1 + 1) - grd P (j - 1))
  proof (rule clmm-quote-gross-diff-eq)
    show clmm-dsc P using assms(1) by simp
    show L = gross-fct (lq P) (fee P) using L-def by simp
    show j - 1 = lower-tick P (grd P (j - 1))
      using assms lower-tick-eq by presburger
    show j = lower-tick P sqp using assms by simp
    show 0 < grd P (j - 1) using assms by simp
    show j - 1 < j by simp
    show grd P (j - 1) ≤ sqp
      using ‹j - 1 < j› assms(1) assms(6) clmm-dsc-grd-mono order-less-imp-le
      by blast
  qed
  also have ... = 0
  proof -
    have L j * (sqp - grd P j) = 0 using assms by simp
    moreover have L (j - 1) * (grd P (j - 1 + 1) - grd P (j - 1)) = 0
      using ‹¬ lq P (j - 1) ≠ 0› by (simp add: L-def gross-fct-zero-if)
    moreover have (∑ i | L i ≠ 0 ∧ j - 1 < i ∧ i < j.
        L i * (grd P (i + 1) - grd P i)) = 0
    proof -
      have {i. L i ≠ 0 ∧ j - 1 < i ∧ i < j} = {} by auto
      thus ?thesis by (metis sum-clauses(1))
    qed
    ultimately show ?thesis by (simp add: assms(6))
  qed
  finally have quote-gross P sqp - quote-gross P (grd P (j-1)) = 0 .
  hence grd P (j-1) ∈ quote-gross P -`{y}
```

```
    by (simp add: ‹quote-gross P sqp = y›)
  hence sqp ≤ grd P (j−1)
    using assms clmm-quote-reach-le by simp
  moreover have grd P (j−1) < sqp using assms
    by (metis clmm-dsc-grd-smono order-refl zle-diff1-eq)
  ultimately show False by simp
qed

lemma quote-gross-gt-grd-min:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and grd-min P < sqp
shows 0 < quote-gross P sqp
proof −
  define L where L = gross-fct (lq P) (fee P)
  define sqpm where sqpm = grd-min P
  define j where j = lower-tick P (grd-min P)
  define k where k = lower-tick P sqp
  have j = idx-min (lq P) using lower-tick-grd-min
    by (simp add: assms(1) j-def)
  have lq P (idx-min (lq P)) ≠ 0
  proof (rule idx-min-finite-in)
    show finite (nz-support (lq P))
      using assms clmm-dsc-liq(1) finite-liq-def by auto
  qed (simp add: assms)
  hence lq P j ≠ 0 using ‹j = idx-min (lq P)› by simp
  hence 0 < L j using gross-liq-gt L-def assms by simp
  show ?thesis
  proof (cases k = j)
    case True
    have 0 < L j * (sqp − sqpm) using ‹0 < L j› assms sqpm-def by simp
    also have ... = quote-gross P sqp − quote-gross P sqpm
    proof (rule clmm-quote-gross-diff-eq'[symmetric])
      show L = gross-fct (lq P) (fee P) using L-def by simp
      show j = lower-tick P sqpm using j-def sqpm-def by simp
      show j = lower-tick P sqp using True k-def by simp
      show 0 < sqpm using assms unfolding sqpm-def
        by (simp add: liq-grd-min)
      show sqpm ≤ sqp using assms unfolding sqpm-def by simp
    qed (simp add: assms)
    also have ... = quote-gross P sqp
    proof −
      have quote-gross P sqpm = 0 unfolding sqpm-def
        by (simp add: assms clmm-quote-gross-grd-min)
      thus ?thesis by simp
    qed
    finally show 0 < quote-gross P sqp .
  next
    case False
```

```
    hence j < k unfolding j-def k-def
      by (metis assms(1) assms(3) clmm-dsc-grid(2) idx-min-img-def
          lower-tick-mono nless-le grd-min-def)
    have 0 < L j * (grd P (j+1) − sqpm)
      using ‹0 < L j› assms(1) j-def lower-tick-ubound sqpm-def by auto
    also have ... ≤ L k * (sqp − grd P k) +
        sum (λ i. L i * (grd P (i+1) − grd P i)) {i. L i ≠ 0 ∧ j <i ∧ i < k} +
        L j * (grd P (j+1) − sqpm)
    proof −
      have 0 ≤ L k * (sqp − grd P k) using gross-liq-ge k-def
        by (metis L-def assms(1) assms(2) assms(3) liq-grd-min
            diff-ge-0-iff-ge  k-def lower-tick-lbound order-less-trans
            zero-le-mult-iff)
      moreover have 0 ≤ sum (λ i. L i * (grd P (i+1) − grd P i))
          {i. L i ≠ 0 ∧ j <i ∧ i < k}
      proof (rule sum-nonneg)
        fix n
        assume n ∈ {i. L i ≠ 0 ∧ j < i ∧ i < k}
        thus 0 ≤ L n * (grd P (n + 1) − grd P n)
          by (simp add: L-def assms(1) rng-quote-gross-ge)
      qed
      ultimately show ?thesis by simp
    qed
    also have ... = quote-gross P sqp − quote-gross P sqpm
    proof (rule clmm-quote-gross-diff-eq[symmetric])
      show clmm-dsc P using assms by simp
      show L = gross-fct (lq P) (fee P) using L-def by simp
      show j < k using ‹j < k› .
      show 0 < sqpm using assms sqpm-def by (simp add: liq-grd-min)
      show sqpm ≤ sqp using sqpm-def assms by simp
    qed (simp add: j-def k-def sqpm-def)+
    also have ... = quote-gross P sqp
    proof −
      have quote-gross P sqpm = 0 unfolding sqpm-def
        by (simp add: assms clmm-quote-gross-grd-min)
      thus ?thesis by simp
    qed
    finally show 0 < quote-gross P sqp .
  qed
qed

lemma quote-gross-pos-gt-grd-min:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and 0 < quote-gross P sqp
shows grd-min P < sqp
proof (rule ccontr)
  assume ¬ grd-min P < sqp
  hence quote-gross P sqp = 0 using assms
```

```
    by (metis quote-gross-imp-sqp-lt clmm-quote-gross-grd-min)
  thus False using assms by simp
qed

lemma quote-gross-disj-gt:
  assumes clmm-dsc P
  and i = lower-tick P sqp
  and j = lower-tick P sqp'
  and i ≤ k
  and k < j
  and lq P k ≠ 0
  and 0 < sqp
  and 0 < sqp'
shows quote-gross P sqp < quote-gross P sqp'
proof −
  define L where L = gross-fct (lq P) (fee P)
  hence L k ≠ 0 using assms gross-liq-gt by fastforce
  have grd P j ≤ sqp' using assms by (simp add: lower-tick-lbound)
  have sqp < grd P (i+1) using assms lower-tick-ubound by simp
  have sqp < sqp' using lower-tick-lt assms by simp
  hence eq: quote-gross P sqp' − quote-gross P sqp = L j ∗ (sqp' − grd P j) +
      (∑ n | L n ≠ 0 ∧ i < n ∧ n < j. L n ∗ (grd P (n + 1) − grd P n)) +
      L i ∗ (grd P (i + 1) − sqp)
    using clmm-quote-gross-diff-eq assms L-def by simp
  show ?thesis
  proof (cases k = i)
    case True
    hence 0 < L i ∗ (grd P (i + 1) − sqp)
      using L-def assms gross-liq-gt lower-tick-ubound by auto
    also have ... ≤ L j ∗ (sqp' − grd P j) +
        (∑ n | L n ≠ 0 ∧ i < n ∧ n < j. L n ∗ (grd P (n + 1) − grd P n)) +
        L i ∗ (grd P (i + 1) − sqp)
    proof −
      have 0 ≤ L j ∗ (sqp' − grd P j)
        using assms L-def gross-liq-ge lower-tick-lbound by auto
      moreover have 0 ≤
          (∑ n | L n ≠ 0 ∧ i < n ∧ n < j. L n ∗ (grd P (n + 1) − grd P n))
      proof (rule sum-nonneg)
        fix n
        assume n ∈ {n. L n ≠ 0 ∧ i < n ∧ n < j}
        show 0 ≤ L n ∗ (grd P (n + 1) − grd P n)
          by (simp add: L-def assms(1) rng-quote-gross-ge)
      qed
      ultimately show ?thesis by simp
    qed
    also have ... = quote-gross P sqp' − quote-gross P sqp using eq by simp
    finally have 0 < quote-gross P sqp' − quote-gross P sqp .
    then show ?thesis by simp
  next
```

```
    case False
    have 0 < (∑ n | L n ≠ 0 ∧ i < n ∧ n < j. L n * (grd P (n + 1) − grd P n))
    proof (rule sum-ex-strict-pos)
      define M where M = {n. L n ≠ 0 ∧ i < n ∧ n < j}
      show finite M using M-def by simp
      have k∈ M using assms False M-def ‹L k ≠ 0› by simp
      moreover have 0 < L k * (grd P (k+1) − grd P k)
        using L-def assms clmm-dsc-grd-Suc gross-liq-gt by auto
      ultimately show ∃ a∈M. 0 < L a * (grd P (a + 1) − grd P a) by auto
      show ∀ x∈M. 0 ≤ L x * (grd P (x + 1) − grd P x)
        by (simp add: L-def assms(1) rng-quote-gross-ge)
    qed
    also have ... ≤ L j * (sqp' − grd P j) +
       (∑ n | L n ≠ 0 ∧ i < n ∧ n < j. L n * (grd P (n + 1) − grd P n)) +
       L i * (grd P (i + 1) − sqp)
    proof −
      have 0 ≤ L j * (sqp' − grd P j)
        using ‹grd P j ≤ sqp'› L-def assms(1) gross-liq-ge by auto
      moreover have 0 ≤ L i * (grd P (i + 1) − sqp)
        using ‹sqp < grd P (i+1)› L-def assms(1) gross-liq-ge by auto
      ultimately show ?thesis by simp
    qed
    also have ... = quote-gross P sqp' − quote-gross P sqp using eq by simp
    finally have 0 < quote-gross P sqp' − quote-gross P sqp .
    thus ?thesis by simp
  qed
qed

lemma quote-gross-disj-gt':
  assumes clmm-dsc P
  and i = lower-tick P sqp
  and j = lower-tick P sqp'
  and i < j
  and lq P j ≠ 0
  and grd P j < sqp'
  and 0 < sqp
  and 0 < sqp'
shows quote-gross P sqp < quote-gross P sqp'
proof −
  define L where L = gross-fct (lq P) (fee P)
  hence 0 < L j using assms gross-liq-gt by fastforce
  have sqp < grd P (i+1) using assms lower-tick-ubound by simp
  have sqp < sqp' using lower-tick-lt assms by simp
  have 0 < L j * (sqp' − grd P j) using assms ‹0 < L j› by simp
  also have ... ≤ L j * (sqp' − grd P j) +
     (∑ n | L n ≠ 0 ∧ i < n ∧ n < j. L n * (grd P (n + 1) − grd P n)) +
     L i * (grd P (i + 1) − sqp)
  proof −
    have 0 ≤ L i * (grd P (i + 1) − sqp)
```

```
    using ‹sqp < grd P (i+1)› L-def assms(1) gross-liq-ge by auto
  moreover have 0 ≤
      (∑ n | L n ≠ 0 ∧ i < n ∧ n < j. L n * (grd P (n + 1) - grd P n))
  proof (rule sum-nonneg)
    fix n
    assume n ∈ {n. L n ≠ 0 ∧ i < n ∧ n < j}
    show 0 ≤ L n * (grd P (n + 1) - grd P n)
      by (simp add: L-def assms(1) rng-quote-gross-ge)
  qed
  ultimately show ?thesis by simp
 qed
 also have ... = quote-gross P sqp' - quote-gross P sqp
  using clmm-quote-gross-diff-eq ‹sqp < sqp'› assms L-def by simp
 finally have 0 < quote-gross P sqp' - quote-gross P sqp .
 thus ?thesis by simp
qed

lemma quote-gross-lower-eq-gt:
 assumes clmm-dsc P
 and j = lower-tick P sqp
 and j = lower-tick P sqp'
 and lq P j ≠ 0
 and 0 < sqp
 and sqp < sqp'
shows quote-gross P sqp < quote-gross P sqp'
proof -
 define L where L = gross-fct (lq P) (fee P)
 hence 0 < L j using assms gross-liq-gt by fastforce
 have sqp < sqp' using lower-tick-lt assms by simp
 hence 0 < L j * (sqp' - sqp) using assms ‹0 < L j› by simp
 also have ... = quote-gross P sqp' - quote-gross P sqp
  using clmm-quote-gross-diff-eq' assms L-def by simp
 finally have 0 < quote-gross P sqp' - quote-gross P sqp .
 thus ?thesis by simp
qed

lemma quote-reach-gt-grd-min:
 assumes clmm-dsc P
 and nz-support (lq P) ≠ {}
 and 0 < y
 and y ≤ quote-gross P (grd-max P)
 and sqp = quote-reach P y
shows grd-min P < sqp
proof (rule ccontr)
 assume ¬ grd-min P < sqp
 hence quote-gross P sqp ≤ quote-gross P (grd-min P)
  using assms clmm-quote-gross-grd-min clmm-quote-gross-grd-min-le
   clmm-quote-reach-pos
  by auto
```

hence $y \leq 0$
by (*metis assms*(1) *assms*(2) *assms*(4) *assms*(5) *clmm-quote-gross-grd-min clmm-quote-gross-reach-eq linorder-le-cases*)
thus *False* **using** *assms* **by** *simp*
qed

lemma *sqp-lt-grd-max-imp-idx*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) $\neq \{\}$
and $0 < sqp$
and $sqp <$ *grd-max P*
and $i =$ *lower-tick P sqp*
shows $i \leq$ *idx-max* (*lq P*)
proof −
have *lid*: *lower-tick P* (*grd-max P*) = *idx-max* (*lq P*) + 1
by (*simp add*: *assms*(1) *idx-max-img-def lower-tick-eq grd-max-def*)
have $i <$ *lower-tick P* (*grd-max P*)
proof (*rule lower-tick-lt'*)
show *clmm-dsc P* **using** *assms* **by** *simp*
show $0 < sqp$ **using** *assms* **by** *simp*
show $sqp <$ *grd-max P* **using** *assms* **by** *simp*
show *grd P* (*lower-tick P* (*grd-max P*)) = *grd-max P*
using *lid* **by** (*simp add*: *idx-max-img-def grd-max-def*)
show $i =$ *lower-tick P sqp* **using** *assms* **by** *simp*
qed *simp*
also have ... = *idx-max* (*lq P*) + 1 **using** *lid* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *quote-gross-lt-grd-max*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) $\neq \{\}$
and $0 < sqp$
and $sqp <$ *grd-max P*
shows *quote-gross P sqp* < *quote-gross P* (*grd-max P*)
proof (*rule quote-gross-disj-gt*)
define i **where** $i =$ *lower-tick P sqp*
thus $i =$ *lower-tick P sqp* **.**
define j **where** $j =$ *lower-tick P* (*grd-max P*)
thus $j =$ *lower-tick P* (*grd-max P*) **.**
define k **where** $k = j - 1$
show *clmm-dsc P* **using** *assms* **by** *simp*
show $0 < sqp$ **using** *assms* **by** *simp*
show $0 <$ *grd-max P* **using** *assms* **by** *simp*
show $k < j$ **unfolding** *k-def* **by** *simp*
have $j =$ *idx-max* (*lq P*) +1 **using** *lower-tick-grd-max*
by (*simp add*: *assms*(1) *j-def*)
have *lq P* (*idx-max* (*lq P*)) $\neq 0$
proof (*rule idx-max-finite-in*)

```
    show finite (nz-support (lq P))
      using assms clmm-dsc-liq(1) finite-liq-def by auto
  qed (simp add: assms)
  hence lq P k ≠ 0 using ‹j = idx-max (lq P) + 1› k-def by simp
  thus lq P (lower-tick P (grd-max P) − 1) ≠ 0
    by (simp add: j-def k-def)
  have i < j
  proof (rule lower-tick-lt'[of P sqp])
    show j = lower-tick P (grd P j) using j-def
      by (simp add: assms(1) lower-tick-eq)
    have grd-max P = grd P j
      using ‹j = idx-max (lq P) +1› grd-max-def idx-max-img-def by metis
    thus sqp < grd P j using assms by simp
    show i = lower-tick P sqp using i-def by simp
  qed (simp add: assms k-def)+
  thus lower-tick P sqp ≤ lower-tick P (grd-max P) − 1
    using i-def j-def by simp
qed

lemma idx-max-gt-liq:
  assumes clmm-dsc P
  and j = idx-max (lq P)
shows ∀ k > j. lq P k = 0
proof (intro allI impI)
  fix k
  assume j < k
  show lq P k = 0
  proof (rule idx-max-finite-gt[of lq P])
    show finite (nz-support (lq P))
      using assms clmm-dsc-liq(1) finite-liq-def by simp
    show idx-max (lq P) < k using assms ‹j < k› by simp
  qed
qed

lemma idx-min-lt-liq:
  assumes clmm-dsc P
  and j = idx-min (lq P)
shows ∀ k < j. lq P k = 0
proof (intro allI impI)
  fix k
  assume k < j
  show lq P k = 0
  proof (rule idx-min-finite-lt[of lq P])
    show finite (nz-support (lq P))
      using assms clmm-dsc-liq(1) finite-liq-def by simp
    show k< idx-min (lq P) using assms ‹k < j› by simp
  qed
qed
```

lemma *quote-reach-le'*:
assumes *clmm-dsc P*
and *grd-min P < sqp*
and *i = lower-tick P sqp*
and *lq P i ≠ 0*
and *y = quote-gross P sqp*
shows *quote-reach P y ≤ sqp*
proof −
interpret *finite-liq-pool P*
by (*simp add*: *assms clmm-dsc-liq*(*1*) *finite-liq-pool-def*)
show *?thesis*
proof (*rule quote-reach-le*)
show ∀ *i*. *0 ≤ lq P i*
by (*simp add*: *assms*(*1*) *clmm-dsc-liq*(*2*))
show *sqp ∈ quote-gross P −' {y}* **using** *assms* **by** *simp*
have *0 < quote-gross P sqp*
proof (*rule quote-gross-gt-grd-min*)
show *grd-min P < sqp* **using** *assms* **by** *simp*
show *nz-support* (*lq P*) ≠ {}
using *assms* **unfolding** *nz-support-def* **by** *auto*
qed (*simp add*: *assms*)
then show *0 < y* **using** *assms* **by** *simp*
have *quote-gross P sqp < quote-gross P* (*grd-max P*)
proof (*rule quote-gross-lt-grd-max*)
have *nz-support* (*lq P*) ≠ {}
using *assms* **unfolding** *nz-support-def* **by** *auto*
hence *0 < grd-min P* **using** *assms*
by (*simp add*: *liq-grd-min*)
thus *0 < sqp* **using** *assms* **by** *simp*
show *sqp < grd-max P* **using** *assms grd-max-gt-if* **by** *simp*
show *nz-support* (*lq P*) ≠ {}
using *assms* **unfolding** *nz-support-def* **by** *auto*
qed (*simp add*: *assms*)+
show ∀ *i*. *fee P i < 1* **by** (*simp add*: *assms*(*1*) *clmm-dsc-fees*)
show *mono* (*grd P*) **by** (*simp add*: *assms*(*1*) *strict-mono-mono*)
qed
qed

lemma *quote-reach-gross-le*:
assumes *clmm-dsc P*
and *grd-min P ≤ sqp*
shows *quote-reach P* (*quote-gross P sqp*) ≤ *sqp*
proof (*rule finite-liq-pool.quote-reach-gross-le*)
show *finite-liq-pool P*
by (*simp add*: *assms*(*1*) *clmm-dsc-liq*(*1*) *finite-liq-pool.intro*)
show ∀ *i*. *0 ≤ lq P i* **by** (*simp add*: *assms*(*1*) *clmm-dsc-liq*(*2*))
show ∀ *i*. *fee P i < 1* **by** (*simp add*: *assms*(*1*) *clmm-dsc-fees*)
show *mono* (*grd P*) **by** (*simp add*: *assms*(*1*) *clmm-dsc-grd-mono monoI*)
show *grd-min P ≤ sqp* **using** *assms* **by** *simp*

qed

lemma *quote-reach-strict-mono*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *0* ≤ *y1*
and *y1* < *y2*
and *y2* ≤ *quote-gross P* (*grd-max P*)
and *sqp* = *quote-reach P y1*
and *sqp′* = *quote-reach P y2*
shows *sqp* < *sqp′*
proof (*rule ccontr*)
assume ¬ *sqp* < *sqp′*
hence *quote-gross P sqp′* ≤ *quote-gross P sqp*
using *assms clmm-quote-gross-mono*[*of P*] **by** (*simp add*: *monoD*)
hence *y2* ≤ *y1*
using *assms clmm-quote-gross-reach-eq* **by** *auto*
thus *False* **using** *assms* **by** *simp*
qed

lemma *quote-reach-mono*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *0* ≤ *y1*
and *y1* ≤ *y2*
and *y2* ≤ *quote-gross P* (*grd-max P*)
and *sqp* = *quote-reach P y1*
and *sqp′* = *quote-reach P y2*
shows *sqp* ≤ *sqp′*
proof (*cases y1* = *y2*)
case *True*
then show *?thesis* **using** *assms* **by** *simp*
next
case *False*
hence *y1* < *y2* **using** *assms* **by** *simp*
then show *?thesis* **using** *assms quote-reach-strict-mono* **by** *fastforce*
qed

lemma *grd-max-quote-reach*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
shows *quote-reach P* (*quote-gross P* (*grd-max P*)) = *grd-max P*
proof (*rule ccontr*)
define *sqp* **where** *sqp* = *quote-reach P* (*quote-gross P* (*grd-max P*))
hence *eq*: *quote-gross P sqp* = *quote-gross P* (*grd-max P*)
by (*meson assms*(*1*) *assms*(*2*) *clmm-quote-gross-reach-eq liq-grd-min-max*
dual-order.strict-trans linorder-le-less-linear order-less-irrefl
quote-gross-gt-grd-min)
define *i* **where** *i* = *lower-tick P sqp*

```
  assume sqp ≠ grd-max P
  hence sqp < grd-max P
    using clmm-quote-reach-le
    by (metis assms liq-grd-min-max order-le-imp-less-or-eq
        quote-gross-gt-grd-min sqp-def vimage-singleton-eq)
  have quote-gross P sqp < quote-gross P (grd-max P)
  proof (rule quote-gross-lt-grd-max)
    show 0 < sqp using clmm-quote-reach-pos[OF assms(1−2)]
      by (metis assms liq-grd-min-max linorder-le-less-linear order.asym
          quote-gross-gt-grd-min sqp-def)
    show sqp < grd-max P using ‹sqp < grd-max P› .
  qed (simp add: assms)+
  thus False using eq by simp
qed

lemma quote-reach-gt:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and 0 < y
  and y + quote-gross P sqp ≤ quote-gross P (grd-max P)
  and sqp′ = quote-reach P (y + quote-gross P sqp)
shows sqp < sqp′
proof (rule ccontr)
  assume ¬ sqp < sqp′
  have y + quote-gross P sqp = quote-gross P sqp′
    using assms clmm-quote-gross-reach-eq
    by (simp add: clmm-quote-gross-pos)
  also have ... ≤ quote-gross P sqp
    using assms ‹¬ sqp < sqp′› quote-gross-imp-sqp-lt by fastforce
  finally show False using assms by simp
qed

lemma lt-quote-gross-imp-up-price:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and 0 < y
  and y ≤ quote-gross P (grd-max P)
  and quote-gross P sqp < y
  and sqp′ = quote-reach P y
shows sqp < sqp′
proof (rule ccontr)
  assume ¬ sqp < sqp′
  have y  = quote-gross P sqp′
    using assms clmm-quote-gross-reach-eq
    by (simp add: clmm-quote-gross-pos)
  also have ... ≤ quote-gross P sqp
    using assms ‹¬ sqp < sqp′› quote-gross-imp-sqp-lt by fastforce
  finally show False using assms by simp
qed
```

lemma *quote-reach-add-gt*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *0* < *y*
and *y* + *quote-gross P sqp* ≤ *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp*)
shows *quote-gross P sqp* < *quote-gross P sqp′*
proof −
have *0* ≤ *y* + *quote-gross P sqp*
using *assms clmm-quote-gross-pos* **by** *simp*
hence *quote-gross P sqp′* = *y* + *quote-gross P sqp*
using *assms clmm-quote-gross-reach-eq* **by** *simp*
thus *?thesis* **using** *assms* **by** *simp*
qed

lemma *quote-reach-leq-grd-max*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *0* ≤ *y*
and *y* ≤ *quote-gross P* (*grd-max P*)
and *sqp* = *quote-reach P y*
shows *sqp* ≤ *grd-max P*
using *assms*
by (*metis quote-reach-mono grd-max-quote-reach*
order-refl)

lemma *quote-gross-grd-max-ge*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *grd-max P* < *sqp*
shows *quote-gross P sqp* = *quote-gross P* (*grd-max P*)
proof −
define *k* **where** *k* = *lower-tick P sqp*
define *j* **where** *j* = *idx-max* (*lq P*) + *1*
define *L* **where** *L* = *gross-fct* (*lq P*) (*fee P*)
have *zer*: ∀ *k* ≥ *j*. *lq P k* = *0* **using** *assms idx-max-gt-liq j-def* **by** *simp*
hence *eq*: ∀ *k* ≥ *j*. *L k* = *0*
by (*simp add*: *L-def gross-fct-zero-if*)
have *lower-tick P* (*grd-max P*) = *j*
by (*simp add*: *assms*(*1*) *lower-tick-grd-max j-def*)
hence *j* ≤ *k* **using** *assms*
by (*metis clmm-dsc-grid*(*2*) *k-def lower-tick-mono order-less-imp-le*
grd-max-gt)
show *?thesis*
proof (*cases k* = *j*)
case *True*
have *quote-gross P sqp* − *quote-gross P* (*grd-max P*) = *L k* ∗ (*sqp* − *grd-max*
P)

```
    proof (rule clmm-quote-gross-diff-eq')
      show  k = lower-tick P (grd-max P)
        using ‹lower-tick P (grd-max P) = j›
        by (simp add: assms(1) lower-tick-eq True)
      show L = gross-fct (lq P) (fee P) using L-def by simp
      show clmm-dsc P using assms by simp
      show grd-max P ≤ sqp using assms by simp
      show k = lower-tick P sqp using assms k-def  by simp
      show 0 < grd-max P using assms grd-max-gt by simp
    qed
    also have ... = 0
      using zer L-def True by (simp add: gross-fct-zero-if)
    finally have quote-gross P sqp − quote-gross P (grd-max P) = 0 .
    then show ?thesis using assms by simp
  next
    case False
    hence j < k using ‹j ≤ k› by simp
    have quote-gross P sqp − quote-gross P (grd-max P) = L k ∗ (sqp − grd P k)
+
       (∑ i | L i ≠ 0 ∧ j < i ∧ i < k. L i ∗ (grd P (i + 1) − grd P i)) +
       L j ∗ (grd P (j + 1) − grd-max P)
    proof (rule clmm-quote-gross-diff-eq)
      show  j = lower-tick P (grd-max P)
        using ‹lower-tick P (grd-max P) = j› by simp
      show L = gross-fct (lq P) (fee P) using L-def by simp
      show clmm-dsc P using assms by simp
      show grd-max P ≤ sqp using assms by simp
      show k = lower-tick P sqp using assms k-def  by simp
      show 0 < grd-max P using assms grd-max-gt by simp
      show j < k using ‹j < k› .
    qed
    also have ... = 0
    proof −
      have {i. L i ≠ 0 ∧ j < i ∧ i < k} = {} using eq by auto
      hence (∑ i | L i ≠ 0 ∧ j < i ∧ i < k. L i ∗ (grd P (i + 1) − grd P i)) = 0
        by (metis (full-types) sum.empty)
      moreover have L k = 0 using eq ‹j < k› by simp
      moreover have L j ∗ (grd P (j + 1) − grd-max P) = 0 using eq by simp
      ultimately show ?thesis by simp
    qed
    finally show ?thesis by simp
  qed
qed

lemma quote-gross-grd-max-max:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
shows quote-gross P sqp ≤ quote-gross P (grd-max P)
proof (cases sqp ≤ grd-max P)
```

case *True*
then show *?thesis*
using *assms*(*1*) *quote-gross-imp-sqp-lt verit-comp-simplify1*(*3*) **by** *blast*
next
case *False*
then show *?thesis* **using** *assms quote-gross-grd-max-ge* **by** *simp*
qed

lemma *gross-grd-max-max'*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *sqp* < *grd-max P*
shows *quote-gross P sqp* < *quote-gross P* (*grd-max P*)
using *assms quote-gross-grd-max-ge*
by (*metis antisym-conv3 liq-grd-min liq-grd-min-max order.strict-trans*
quote-gross-gt-grd-min quote-gross-lt-grd-max quote-gross-pos-gt-grd-min)

lemma *quote-reach-le-gross*:
assumes *clmm-dsc P*
and *0* < *y*
and *0* < *sqp*
and *y* ≤ *quote-gross P sqp*
and *sqp* ≤ *grd-max P*
and *sqp'* = *quote-reach P y*
and *nz-support* (*lq P*) ≠ {}
shows *sqp'* ≤ *sqp*
proof −
interpret *finite-liq-pool P*
by (*simp add*: *assms clmm-dsc-liq*(*1*) *finite-liq-pool-def*)
have *lt*: *quote-gross P sqp* ≤ *quote-gross P* (*grd-max P*)
by (*simp add*: *assms*(*1*) *assms*(*7*) *quote-gross-grd-max-max*)
show *?thesis*
proof (*cases y* = *quote-gross P sqp*)
case *True*
then show *?thesis* **using** *clmm-quote-reach-le lt assms* **by** *simp*
next
case *False*
hence *y* < *quote-gross P sqp* **using** *assms* **by** *simp*
hence *quote-gross P sqp'* < *quote-gross P sqp*
using *assms clmm-quote-gross-reach-eq lt* **by** *auto*
then show *?thesis*
using *assms*(*1*) *clmm-quote-gross-mono mono-invE* **by** *blast*
qed
qed

lemma *quote-net-diff-eq*:
assumes *clmm-dsc P*
and *L* = *lq P*
and *j* = *lower-tick P sqp*

and $k = \textit{lower-tick}\ P\ \textit{sqp}'$
and $0 < \textit{sqp}$
and $\textit{sqp} \leq \textit{sqp}'$
and $j < k$
shows $\textit{quote-net}\ P\ \textit{sqp}' - \textit{quote-net}\ P\ \textit{sqp} = L\ k * (\textit{sqp}' - \textit{grd}\ P\ k) +$
$\textit{sum}\ (\lambda\ i.\ L\ i * (\textit{grd}\ P\ (i{+}1) - \textit{grd}\ P\ i))\ \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} +$
$L\ j * (\textit{grd}\ P\ (j{+}1) - \textit{sqp})$
proof −
interpret *finite-nz-support* L
using *finite-liq-pool.finite-liq-gross-fct assms clmm-dsc-def*
finite-liq-def finite-nz-support-def
by *blast*
show *?thesis*
using *clmm-gen-quote-diff-eq assms*
unfolding *quote-net-def gen-quote-diff-def* **by** *simp*
qed

lemma *quote-net-diff-eq′*:
assumes *clmm-dsc P*
and $L = \textit{lq}\ P$
and $j = \textit{lower-tick}\ P\ \textit{sqp}$
and $j = \textit{lower-tick}\ P\ \textit{sqp}'$
and $0 < \textit{sqp}$
and $\textit{sqp} \leq \textit{sqp}'$
shows $\textit{quote-net}\ P\ \textit{sqp}' - \textit{quote-net}\ P\ \textit{sqp} = L\ j * (\textit{sqp}' - \textit{sqp})$
proof −
interpret *finite-nz-support* L
using *finite-liq-pool.finite-liq-gross-fct assms clmm-dsc-def*
finite-liq-def finite-nz-support-def
by *blast*
show *?thesis* **using** *clmm-gen-quote-diff-eq′ assms*
unfolding *quote-net-def gen-quote-diff-def* **by** *simp*
qed

4.3 Base token addition and withdrawal in a CLMM

lemma (**in** *finite-nz-support*) *gen-base-sum*:
assumes *clmm-dsc P*
and $0 < \textit{sqp}$
and $j = \textit{lower-tick}\ P\ \textit{sqp}$
shows $\textit{gen-base}\ (\textit{grd}\ P)\ L\ \textit{sqp} =$
$L\ j * (\textit{inverse}\ \textit{sqp} - \textit{inverse}\ (\textit{grd}\ P\ (j{+}1))) +$
$\textit{sum}\ (\lambda i.\ L\ i * (\textit{inverse}\ (\textit{grd}\ P\ i) - \textit{inverse}\ (\textit{grd}\ P\ (i{+}1))))$
$\{i.\ L\ i \neq 0 \wedge j < i\}$
proof −
define *df* **where** $\textit{df} = \textit{inv-gamma-max-diff}\ (\textit{grd}\ P)$
define *A* **where** $A = \{i.\ L\ i \neq 0 \wedge j \leq i\}$
define *B* **where** $B = \{i.\ L\ i \neq 0 \wedge j > i\}$
define *C* **where** $C = \{i.\ L\ i \neq 0 \wedge j < i\}$

have *un*: $\{i.\ L\ i \neq 0\} = A \cup B$ **unfolding** *A-def B-def* **by** *auto*
have *inter*: $A \cap B = \{\}$ **unfolding** *A-def B-def* **by** *auto*
have *fin*: *finite* $\{i.\ L\ i \neq 0\}$ **by** (*metis fin-nz-sup nz-support-def*)
have *gen-base* (*grd P*) *L sqp* =
sum (*rng-token df L sqp*) $\{i.\ L\ i \neq 0\}$
unfolding *gen-base-def df-def* **using** *gen-token-sum* **by** *simp*
also have ... = *sum* (*rng-token df L sqp*) *A* + *sum* (*rng-token df L sqp*) *B*
by (*metis empty-iff fin finite-Un inter sum.union-inter-neutral un*)
also have ... = *sum* (*rng-token df L sqp*) *A*
proof −
have ∀ *i*∈ *B*. *rng-token df L sqp i* = *0*
proof
fix *i*
assume *i* ∈ *B*
have *grd P i* < *grd P* (*i*+*1*) **using** *assms clmm-dsc-grid span-gridD*
by (*simp add*: *strict-mono-less*)
have *i* + *1* ≤ *j* **using** ‹*i* ∈ *B*› *assms* **unfolding** *B-def* **by** *simp*
hence *grd P* (*i* +*1*) ≤ *grd P j*
using *assms clmm-dsc-grid span-gridD* **by** (*simp add*: *strict-mono-leD*)
hence *grd P* (*i*+*1*) ≤ *sqp* **using** *assms*
by (*meson dual-order.trans lower-tick-lbound*)
hence *grd P i* < *sqp* **using** ‹*grd P i* < *grd P* (*i*+*1*)› **by** *simp*
hence *df sqp i* = *0*
using ‹*grd P* (*i*+*1*) ≤ *sqp*› **unfolding** *df-def inv-gamma-max-diff-def* **by** *simp*
thus *rng-token df L sqp i* = *0* **unfolding** *rng-token-def* **by** *simp*
qed
thus *?thesis* **by** *simp*
qed
also have ... = *rng-token df L sqp j* + *sum* (*rng-token df L sqp*) *C*
proof (*cases j* ∈ *A*)
case *True*
hence *A* = {*j*} ∪ *C* **unfolding** *A-def C-def* **by** *auto*
hence *sum* (*rng-token df L sqp*) *A* = *sum* (*rng-token df L sqp*) ({*j*} ∪ *C*)
by *simp*
also have ... = *sum* (*rng-token df L sqp*) {*j*} + *sum* (*rng-token df L sqp*) *C*
proof (*rule sum.union-inter-neutral*)
show *finite* {*j*} **by** *simp*
show *finite C* **using** *fin C-def* **by** *simp*
show ∀ *x*∈{*j*} ∩ *C*. *rng-token df L sqp x* = *0* **using** *C-def* **by** *simp*
qed
also have ... = *rng-token df L sqp j* + *sum* (*rng-token df L sqp*) *C* **by** *simp*
finally show *?thesis* **.**
next
case *False*
hence *A* = *C* **unfolding** *A-def C-def* **using** *Collect-cong* **by** *fastforce*
have *L j* = *0* **using** *False A-def* **by** *auto*
hence *rng-token df L sqp j* = *0* **unfolding** *rng-token-def* **by** *simp*
then show *?thesis* **using** ‹*A* = *C*› **by** *simp*

```
  qed
  also have ... = L j * (inverse sqp - inverse (grd P (j+1))) +
      sum (rng-token df L sqp) C
  proof -
    have max sqp (grd P (j + 1)) = grd P (j+1)
      using assms lower-tick-ubound by simp
    moreover have max sqp (grd P j) = sqp
      using assms lower-tick-lbound by simp
    ultimately have rng-token df L sqp j =
        L j * (inverse sqp - inverse (grd P (j+1)))
      unfolding rng-token-def df-def inv-gamma-max-diff-def by simp
    thus ?thesis by simp
  qed
  also have ... = L j * (inverse sqp - inverse (grd P (j+1))) +
      sum (λi. L i * (inverse (grd P i) - inverse (grd P (i+1)))) C
  proof -
    have ∀ i∈ C. rng-token df L sqp i =
        L i * (inverse (grd P i) - inverse (grd P (i+1)))
    proof
      fix i
      assume i∈ C
      hence j+1 ≤ i using C-def by auto
      hence grd P (j+1) ≤ grd P i using assms clmm-dsc-grid(1) strict-monoD
        by (metis linorder-le-less-linear nless-le)
      hence sqp ≤ grd P i using lower-tick-ubound assms by fastforce
      hence sqp ≤ grd P (i+1)
        using clmm-dsc-grd-Suc assms dual-order.strict-trans2 order-less-imp-le
        by blast
      thus rng-token df L sqp i =
          L i * (inverse (grd P i) - inverse (grd P (i+1)))
        unfolding rng-token-def df-def inv-gamma-max-diff-def
        using ‹sqp ≤ grd P i›
        by simp
    qed
    hence sum (rng-token df L sqp) C =
        sum (λi. L i * (inverse (grd P i) - inverse (grd P (i+1)))) C
      by simp
    thus ?thesis by simp
  qed
  finally show gen-base (grd P) L sqp =
      L j * (inverse sqp - inverse (grd P (j+1))) +
      sum (λi. L i * (inverse (grd P i) - inverse (grd P (i+1)))) C .
qed

lemma (in finite-nz-support) gen-base-grd-max:
  assumes clmm-dsc P
  and nz-support L ≠ {}
  and nz-support L = nz-support (lq P)
shows gen-base (grd P) L (grd-max P) = 0
```

```
proof −
  define j where j = lower-tick P (grd-max P)
  hence j = idx-max (lq P) + 1 using lower-tick-grd-max[of P] assms by simp
  have gen-base (grd P) L (grd-max P) =
      L j ∗ (inverse (grd-max P) − inverse (grd P (j + 1))) +
      (∑ i | L i ≠ 0 ∧ j < i.
      L i ∗ (inverse (grd P i) − inverse (grd P (i + 1))))
  proof (rule gen-base-sum)
    show 0 < grd-max P
    proof (rule grd-max-gt)
      show nz-support (lq P) ≠ {} using assms by simp
      show ⋀i. 0 < grd P i using assms by simp
    qed
  qed (simp add: assms j-def)+
  also have ... =0
  proof −
    have emp: {i. L i ≠ 0 ∧ j ≤ i} = {}
    proof −
      have ∀ i ≥ j. L i = 0
      proof (intro allI impI)
        fix i
        assume j ≤ i
        hence idx-max (lq P) < i using ‹j = idx-max (lq P) + 1› by simp
        hence lq P i = 0
          using idx-max-finite-ge[of lq P i] fin-nz-sup assms(3) by auto
        thus L i = 0 using assms unfolding nz-support-def by auto

      qed
      thus ?thesis by auto
    qed
    hence a: L j ∗ (inverse (grd-max P) − inverse (grd P (j + 1))) = 0 by auto
    have {i. L i ≠ 0 ∧ j < i} = {} using emp by auto
    hence (∑ i | L i ≠ 0 ∧ j < i.
        L i ∗ (inverse (grd P i) − inverse (grd P (i + 1)))) = 0
      by (metis (full-types) sum-clauses(1))
    thus ?thesis using a by simp
  qed
  finally show ?thesis .
qed

lemma base-gross-sum:
  assumes clmm-dsc P
  and 0 < sqp
  and L = gross-fct (lq P) (fee P)
  and j = lower-tick P sqp
shows base-gross P sqp =
    L j ∗ (inverse sqp − inverse (grd P (j+1))) +
    sum (λi. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
    {i. L i ≠ 0 ∧ j < i}
```

proof −
interpret *finite-nz-support L*
using *finite-liq-pool.finite-liq-gross-fct assms*
by (*metis clmm-dsc-liq*(*1*) *finite-liq-pool.intro finite-nz-support-def nz-support-def*)
show *?thesis* **unfolding** *base-gross-def* **using** *gen-base-sum assms* **by** *simp*
qed

lemma *clmm-base-gross-grd-max*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
shows *base-gross P* (*grd-max P*) = *0* **unfolding** *base-gross-def*
proof (*rule finite-nz-support.gen-base-grd-max*)
show *finite-nz-support* (*gross-fct* (*lq P*) (*fee P*))
using *finite-liq-pool.finite-liq-gross-fct assms*
by (*metis clmm-dsc-liq*(*1*) *finite-liq-pool.intro finite-nz-support-def nz-support-def*)
show *clmm-dsc P* **using** *assms* **by** *simp*
show *nz-support* (*gross-fct* (*lq P*) (*fee P*)) = *nz-support* (*lq P*)
using *clmm-dsc-gross-liq assms* **by** *simp*
thus *nz-support* (*gross-fct* (*lq P*) (*fee P*)) ≠ {} **using** *assms* **by** *simp*
qed

lemma *liq-base-reach-max*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
shows *base-reach P 0* = *grd-max P*
using *assms clmm-base-gross-grd-max* **unfolding** *base-reach-def* **by** *simp*

lemma *base-net-sum*:
assumes *clmm-dsc P*
and *0* < *sqp*
and *L* = *lq P*
and *j* = *lower-tick P sqp*
shows *base-net P sqp* =
L j ∗ (*inverse sqp* − *inverse* (*grd P* (*j*+*1*))) +
sum (λ*i*. *L i* ∗ (*inverse* (*grd P i*) − *inverse* (*grd P* (*i*+*1*))))
{*i*. *L i* ≠ *0* ∧ *j* < *i*}
proof −
interpret *finite-nz-support L*
using *finite-liq-pool.finite-liq-gross-fct assms clmm-dsc-def finite-liq-def finite-nz-support-def*
by *blast*
show *?thesis* **unfolding** *base-net-def* **using** *gen-base-sum assms* **by** *simp*
qed

definition *gen-base-diff* **where**
gen-base-diff P L sqp sqp′ = *gen-base* (*grd P*) *L sqp* − *gen-base* (*grd P*) *L sqp′*

lemma (**in** *finite-nz-support*) *gen-base-diff-eq*:
assumes *clmm-dsc P*
and $j = lower\text{-}tick\ P\ sqp$
and $k = lower\text{-}tick\ P\ sqp'$
and $0 < sqp$
and $sqp \leq sqp'$
and $j < k$
shows $gen\text{-}base\text{-}diff\ P\ L\ sqp\ sqp' =$
$L\ j * (inverse\ sqp - inverse\ (grd\ P\ (j{+}1))) +$
$sum\ (\lambda\ i.\ L\ i * (inverse\ (grd\ P\ i) - inverse\ (grd\ P\ (i{+}1))))$
$\{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} +$
$L\ k * (inverse\ (grd\ P\ k) - inverse\ sqp')$
proof −
have $0 < sqp$ **using** *assms* **by** *simp*
hence $sqp < sqp'$ **using** *lower-tick-lt assms* **by** *simp*
hence $0 < sqp'$ **using** ‹$0 < sqp$› **by** *simp*
define A **where** $A = \{i.\ L\ i \neq 0 \wedge j < i\}$
define B **where** $B = \{i.\ L\ i \neq 0 \wedge k < i\}$
define C **where** $C = \{i.\ L\ i \neq 0 \wedge j < i \wedge i \leq k\}$
define Ck **where** $Ck = \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\}$
define df **where** $df = (\lambda\ i.\ L\ i * (inverse\ (grd\ P\ i) - inverse\ (grd\ P\ (i{+}1))))$
have *finite A*
unfolding *A-def* **by** (*metis fin-nz-sup finite-Collect-conjI nz-support-def*)
have $A = B \cup C$ **using** *assms* **unfolding** *A-def B-def C-def* **by** *auto*
have $Ck = C - \{k\}$ **unfolding** *C-def Ck-def* **by** *auto*
have $gen\text{-}base\text{-}diff\ P\ L\ sqp\ sqp' =$
$L\ j * (inverse\ sqp - inverse\ (grd\ P\ (j{+}1))) +$
$sum\ df\ A - (L\ k * (inverse\ sqp' - inverse\ (grd\ P\ (k{+}1))) + sum\ df\ B)$
using *assms* ‹$0 < sqp$› *gen-base-sum* ‹$0 < sqp'$›
unfolding *gen-base-diff-def df-def A-def B-def* **by** *simp*
also have $... = L\ j * (inverse\ sqp - inverse\ (grd\ P\ (j{+}1))) +$
$sum\ df\ C - L\ k * (inverse\ sqp' - inverse\ (grd\ P\ (k{+}1)))$
proof (*rule diff-sum-dcomp*)
show *finite A* **using** ‹*finite A*› **.**
show $A = B \cup C$ **using** ‹$A = B \cup C$› **.**
show $B \cap C = \{\}$ **unfolding** *B-def C-def* **by** *auto*
qed
also have $... = L\ j * (inverse\ sqp - inverse\ (grd\ P\ (j{+}1))) +$
$df\ k + sum\ df\ Ck - L\ k * (inverse\ sqp' - inverse\ (grd\ P\ (k{+}1)))$
proof (*cases* $k \in C$)
case *True*
have $sum\ df\ C = df\ k + sum\ df\ Ck$
proof (*rule sum-remove-el*)
show *finite C* **using** ‹$A = B \cup C$› ‹*finite A*› **by** *simp*
show $k \in C$ **using** *True* **.**
show $Ck = C - \{k\}$ **using** ‹$Ck = C - \{k\}$› **.**
qed
then show *?thesis* **by** *simp*
next

```
    case False
    hence L k = 0 using assms unfolding C-def by auto
    hence df k = 0 unfolding df-def by simp
    moreover have Ck = C using False ‹Ck = C − {k}› by auto
    ultimately show ?thesis by simp
  qed
  also have ... = L j ∗ (inverse sqp − inverse (grd P (j+1))) + sum df Ck +
      L k ∗ (inverse (grd P k) − inverse sqp′)
  proof −
    have df k − L k ∗ (inverse sqp′ − inverse (grd P (k+1))) =
        L k ∗ (inverse (grd P k) − inverse sqp′)
      unfolding df-def by (simp add: right-diff-distrib)
    thus ?thesis by simp
  qed
  finally show gen-base-diff P L sqp sqp′ =
      L j ∗ (inverse sqp − inverse (grd P (j+1))) + sum df Ck +
      L k ∗ (inverse (grd P k) − inverse sqp′) .
qed

lemma (in finite-nz-support) gen-base-diff-eq′:
  assumes clmm-dsc P
  and j = lower-tick P sqp
  and j = lower-tick P sqp′
  and 0 < sqp
  and sqp ≤ sqp′
shows gen-base-diff P L sqp sqp′ = L j ∗ (inverse sqp − inverse sqp′)
proof −
  have 0 < sqp using assms by simp
  hence 0 < sqp′ using assms by simp
  define A where A = {i. L i ≠ 0 ∧ j < i}
  define df where
      df = (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
  have finite A
    unfolding A-def by (metis fin-nz-sup finite-Collect-conjI nz-support-def)
  have gen-base-diff P L sqp sqp′ =
      L j ∗ (inverse sqp − inverse (grd P (j+1))) + sum df A −
      (L j ∗ (inverse sqp′ − inverse (grd P (j+1))) + sum df A)
    using assms ‹0 < sqp› gen-base-sum ‹0 < sqp′›
    unfolding gen-base-diff-def df-def A-def by simp
  also have ... = L j ∗ (inverse sqp − inverse (grd P (j+1))) −
      L j ∗ (inverse sqp′ − inverse (grd P (j+1)))
    by simp
  also have ... = L j ∗ (inverse sqp − inverse sqp′)
    by (simp add: right-diff-distrib)
  finally show gen-base-diff P L sqp sqp′ =
      L j ∗ (inverse sqp − inverse sqp′) .
qed

lemma lower-tick-lt-grd-min:
```

```
  assumes clmm-dsc P
  and sqp < grd-min P
  and 0 < sqp
  and j = lower-tick P sqp
  shows j < idx-min (lq P)
proof (rule ccontr)
  have j ≤ idx-min (lq P) using lower-tick-grd-min[of P] assms
    by (simp add: lower-tick-mono)
  assume ¬ j < idx-min (lq P)
  hence j = idx-min (lq P) using assms ‹j ≤ idx-min (lq P)› by simp
  hence grd-min P = grd P j unfolding grd-min-def idx-min-img-def by simp
  also have ... ≤ sqp using ‹j = lower-tick P sqp›
    by (simp add: assms lower-tick-mem)
  finally show False using assms by simp
qed

lemma (in finite-nz-support) gen-base-grd-min-le:
  assumes clmm-dsc P
  and nz-support L = nz-support (lq P)
  and sqp < grd-min P
  and 0 < sqp
shows gen-base (grd P) L sqp = gen-base (grd P) L (grd-min P)
proof −
  define k where k = idx-min (lq P)
  hence k = lower-tick P (grd-min P)
    by (simp add: assms(1) idx-min-img-def lower-tick-eq grd-min-def)
  have grd-min P = grd P k
    using k-def grd-min-def idx-min-img-def by simp
  define j where j = lower-tick P sqp
  hence j < k using lower-tick-lt-grd-min[of P] assms k-def
    by simp
  have eq: ∀ i < k. L i = 0
    using assms unfolding k-def idx-min-def
    by (metis fin-nz-sup idx-min-def idx-min-finite-le leD)
  have gen-base-diff P L sqp (grd-min P) =
      L j * (inverse sqp − inverse (grd P (j+1))) +
      sum (λ i. L i * (inverse (grd P i) − inverse (grd P (i+1))))
      {i. L i ≠ 0 ∧ j <i ∧ i < k} +
      L k * (inverse (grd P k) − inverse (grd-min P))
  proof (rule gen-base-diff-eq)
    show j = lower-tick P sqp using j-def by simp
    show k = lower-tick P (grd-min P) using ‹k = lower-tick P (grd-min P)› .
    show sqp ≤ grd-min P using assms by simp
  qed (simp add: assms ‹j < k›)+
  also have ... = 0
  proof −
    have {i. L i ≠ 0 ∧ j <i ∧ i < k} = {} using eq by auto
    hence sum (λ i. L i * (inverse (grd P i) − inverse (grd P (i+1))))
        {i. L i ≠ 0 ∧ j <i ∧ i < k} = 0
```

```
    by (metis (full-types) sum.empty)
   moreover have L j * (inverse sqp - inverse (grd P (j+1))) = 0
    using ‹j < k› eq by simp
   moreover have  L k * (inverse (grd P k) - inverse (grd-min P)) = 0
    using ‹grd-min P = grd P k› by simp
   ultimately show ?thesis by linarith
  qed
  finally show ?thesis unfolding gen-base-diff-def by simp
qed

lemma base-net-grd-min-lt:
  assumes clmm-dsc P
  and sqp < grd-min P
  and 0 < sqp
shows base-net P sqp = base-net P (grd-min P)
proof -
  interpret finite-nz-support lq P
    using assms(1) clmm-dsc-liq(1) finite-liq-def finite-nz-support.intro
    by simp
  show ?thesis
    using assms gen-base-grd-min-le unfolding base-net-def by simp
qed

lemma base-net-grd-min-le:
  assumes clmm-dsc P
  and sqp ≤ grd-min P
  and 0 < sqp
shows base-net P sqp = base-net P (grd-min P)
proof -
  interpret finite-nz-support lq P
    using assms(1) clmm-dsc-liq(1) finite-liq-def finite-nz-support.intro
    by simp
  show ?thesis
  proof (cases sqp = grd-min P)
    case True
    then show ?thesis by simp
  next
    case False
    then show ?thesis
      using assms gen-base-grd-min-le unfolding base-net-def by simp
  qed
qed

lemma base-gross-diff-eq:
  assumes clmm-dsc P
  and L = gross-fct (lq P) (fee P)
  and j = lower-tick P sqp
  and k = lower-tick P sqp'
  and 0 < sqp
```

and $sqp \leq sqp'$
and $j < k$
shows *base-gross P sqp − base-gross P sqp′ =*
$L\ j * (inverse\ sqp - inverse\ (grd\ P\ (j+1))) +$
$sum\ (\lambda\ i.\ L\ i * (inverse\ (grd\ P\ i) - inverse\ (grd\ P\ (i+1))))$
$\{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} +$
$L\ k * (inverse\ (grd\ P\ k) - inverse\ sqp')$
proof −
interpret *finite-nz-support L*
using *finite-liq-pool.finite-liq-gross-fct assms*
by (*metis clmm-dsc-liq*(*1*) *finite-liq-pool.intro finite-nz-support-def nz-support-def*)
show *?thesis*
using *gen-base-diff-eq assms*
unfolding *base-gross-def gen-base-diff-def* **by** *simp*
qed

lemma *base-gross-diff-eq′*:
assumes *clmm-dsc P*
and $L = gross\text{-}fct\ (lq\ P)\ (fee\ P)$
and $j = lower\text{-}tick\ P\ sqp$
and $j = lower\text{-}tick\ P\ sqp'$
and $0 < sqp$
and $sqp \leq sqp'$
shows $base\text{-}gross\ P\ sqp - base\text{-}gross\ P\ sqp' = L\ j * (inverse\ sqp - inverse\ sqp')$
proof −
interpret *finite-nz-support L*
using *finite-liq-pool.finite-liq-gross-fct assms*
by (*metis clmm-dsc-liq*(*1*) *finite-liq-pool.intro finite-nz-support-def nz-support-def*)
show *?thesis* **using** *gen-base-diff-eq′ assms*
unfolding *base-gross-def gen-base-diff-def* **by** *simp*
qed

lemma *base-net-diff-eq*:
assumes *clmm-dsc P*
and $L = lq\ P$
and $j = lower\text{-}tick\ P\ sqp$
and $k = lower\text{-}tick\ P\ sqp'$
and $0 < sqp$
and $sqp \leq sqp'$
and $j < k$
shows *base-net P sqp − base-net P sqp′ =*
$L\ j * (inverse\ sqp - inverse\ (grd\ P\ (j+1))) +$
$sum\ (\lambda\ i.\ L\ i * (inverse\ (grd\ P\ i) - inverse\ (grd\ P\ (i+1))))$
$\{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} +$
$L\ k * (inverse\ (grd\ P\ k) - inverse\ sqp')$
proof −
interpret *finite-nz-support L*

using *finite-liq-pool.finite-liq-gross-fct assms clmm-dsc-def*
finite-liq-def finite-nz-support-def
by *blast*
show *?thesis*
using *gen-base-diff-eq assms*
unfolding *base-net-def gen-base-diff-def* **by** *simp*
qed

lemma *base-net-diff-eq'*:
assumes *clmm-dsc P*
and $L = lq\ P$
and $j = lower\text{-}tick\ P\ sqp$
and $j = lower\text{-}tick\ P\ sqp'$
and $0 < sqp$
and $sqp \leq sqp'$
shows $base\text{-}net\ P\ sqp - base\text{-}net\ P\ sqp' = L\ j * (inverse\ sqp - inverse\ sqp')$
proof −
interpret *finite-nz-support L*
using *finite-liq-pool.finite-liq-gross-fct assms clmm-dsc-def*
finite-liq-def finite-nz-support-def
by *blast*
show *?thesis* **using** *gen-base-diff-eq' assms*
unfolding *base-net-def gen-base-diff-def* **by** *simp*
qed

lemma *cst-fee-base-gross-net-tick-eq*:
assumes *clmm-dsc P*
and $\bigwedge i.\ fee\ P\ i = phi$
and $j = lower\text{-}tick\ P\ sqp$
and $j = lower\text{-}tick\ P\ sqp'$
and $0 < sqp$
and $sqp \leq sqp'$
shows $base\text{-}net\ P\ sqp - base\text{-}net\ P\ sqp' =$
$(1 - phi) * (base\text{-}gross\ P\ sqp - base\text{-}gross\ P\ sqp')$
proof −
define L **where** $L = gross\text{-}fct\ (lq\ P)\ (fee\ P)$
have $phi \neq 1$ **using** *assms clmm-dsc-fees* **by** *fastforce*
have $\bigwedge i.\ L\ i = lq\ P\ i\ /\ (1 - phi)$ **using** *L-def*
by (*simp add*: *assms*(*2*) *gross-fct-def one-cpl-def*)
hence *leq*: $\bigwedge i.\ lq\ P\ i = (1 - phi) * L\ i$ **using** *assms* ‹$phi \neq 1$› **by** *simp*
have $base\text{-}net\ P\ sqp - base\text{-}net\ P\ sqp' =$
$lq\ P\ j * (inverse\ sqp - inverse\ sqp')$
using *assms base-net-diff-eq'* **by** *simp*
also have $... = (1 - phi) * L\ j * (inverse\ sqp - inverse\ sqp')$
using *leq* **by** *simp*
also have $... = (1 - phi) * (L\ j * (inverse\ sqp - inverse\ sqp'))$ **by** *simp*
also have $... = (1 - phi) * (base\text{-}gross\ P\ sqp - base\text{-}gross\ P\ sqp')$
using *L-def assms base-gross-diff-eq'* **by** *auto*
finally show *?thesis* **.**

qed

lemma *cst-fee-base-gross-net-tick-lt*:
assumes *clmm-dsc P*
and $\bigwedge i.$ *fee P i = phi*
and *j = lower-tick P sqp*
and *k = lower-tick P sqp'*
and *0 < sqp*
and *sqp ≤ sqp'*
and *j < k*
shows *base-net P sqp − base-net P sqp' =*
(1 − phi) ∗ (base-gross P sqp − base-gross P sqp')
proof −
have *phi ≠ 1* **using** *assms clmm-dsc-fees* **by** *fastforce*
define *L* **where** *L = gross-fct (lq P) (fee P)*
have $\bigwedge i.$ *L i = lq P i / (1 − phi)* **using** *L-def*
by (*simp add*: *assms*(*2*) *gross-fct-def one-cpl-def*)
hence *leq*: $\bigwedge i.$ *lq P i = (1 − phi) ∗ L i* **using** *assms ‹phi ≠ 1›* **by** *simp*
have *base-net P sqp − base-net P sqp' =*
(lq P) j ∗ (inverse sqp − inverse (grd P (j+1))) +
sum (λ i. (lq P) i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
{i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} +
(lq P) k ∗ (inverse (grd P k) − inverse sqp')
using *assms base-net-diff-eq* **by** *simp*
also have ... =
(1 − phi) ∗ L j ∗ (inverse sqp − inverse (grd P (j+1))) +
sum (λ i. (1 − phi) ∗ L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
{i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} +
(1 − phi) ∗ L k ∗ (inverse (grd P k) − inverse sqp')
using *leq* **by** *simp*
also have ... =
(1 − phi) ∗ L j ∗ (inverse sqp − inverse (grd P (j+1))) +
(1 − phi) ∗ sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
{i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} +
(1 − phi) ∗ L k ∗ (inverse (grd P k) − inverse sqp')
proof −
have *sum (λ i. (1 − phi) ∗ L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))*
{i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} =
sum (λ i. (1 − phi) ∗(L i ∗ (inverse (grd P i) − inverse (grd P (i+1)))))
{i. (lq P) i ≠ 0 ∧ j <i ∧ i < k}
by (*meson ab-semigroup-mult-class.mult-ac*(*1*))
also have ... =
(1 − phi) ∗ sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
{i. (lq P) i ≠ 0 ∧ j <i ∧ i < k}
by (*simp add*: *sum-distrib-left*)
finally have *sum (λ i. (1 − phi) ∗ L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))*
{i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} =
(1 − phi) ∗ sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))

```
      {i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} .
    thus ?thesis by simp
  qed
  also have ... =
      (1 − phi) ∗ (L j ∗ (inverse sqp − inverse (grd P (j+1)))) +
      (1 − phi) ∗ sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
      {i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} +
      (1 − phi) ∗ (L k ∗ (inverse (grd P k) − inverse sqp′))
    by simp
  also have ... = (1 − phi) ∗
      (L j ∗ (inverse sqp − inverse (grd P (j+1))) +
      sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
      {i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} +
      L k ∗ (inverse (grd P k) − inverse sqp′))
    by (simp add: ring-class.ring-distribs(1))
  also have ... = (1 − phi) ∗ (base-gross P sqp − base-gross P sqp′)
  proof −
    have L j ∗ (inverse sqp − inverse (grd P (j+1))) +
        sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
        {i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} +
        L k ∗ (inverse (grd P k) − inverse sqp′) =
        L j ∗ (inverse sqp − inverse (grd P (j+1))) +
        sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
        {i. L i ≠ 0 ∧ j <i ∧ i < k} +
        L k ∗ (inverse (grd P k) − inverse sqp′)
    proof −
      have {i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} = {i. L i ≠ 0 ∧ j <i ∧ i < k}
        using L-def assms(1) clmm-dsc-gross-liq-zero-iff by presburger
      thus ?thesis by simp
    qed
    also have ... = base-gross P sqp − base-gross P sqp′
      using assms base-gross-diff-eq L-def by simp
    finally have L j ∗ (inverse sqp − inverse (grd P (j+1))) +
        sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
        {i. (lq P) i ≠ 0 ∧ j <i ∧ i < k} +
        L k ∗ (inverse (grd P k) − inverse sqp′) =
        base-gross P sqp − base-gross P sqp′ .
    thus ?thesis by simp
  qed
  finally show ?thesis .
qed

lemma cst-fee-base-gross-net:
  assumes clmm-dsc P
  and ⋀i. fee P i = phi
  and 0 < sqp
  and sqp ≤ sqp′
shows base-net P sqp − base-net P sqp′ =
    (1 − phi) ∗ (base-gross P sqp − base-gross P sqp′)
```

proof (*cases lower-tick P sqp = lower-tick P sqp'*)
case *True*
then show *?thesis* **using** *assms cst-fee-base-gross-net-tick-eq* **by** *simp*
next
case *False*
hence *lower-tick P sqp < lower-tick P sqp'*
using *assms lower-tick-mono* **by** *fastforce*
then show *?thesis* **using** *assms cst-fee-base-gross-net-tick-lt* **by** *simp*
qed

lemma *base-net-eq-only-if*:
assumes *clmm-dsc P*
and *j = lower-tick P sqp*
and *k = lower-tick P sqp'*
and *0 < sqp*
and *sqp ≤ sqp'*
and *j < k*
and *quote-gross P sqp' = quote-gross P sqp*
shows *base-net P sqp' = base-net P sqp*
proof −
define *L* **where** *L = lq P*
define *L'* **where** *L' = gross-fct (lq P) (fee P)*
have *base-net P sqp − base-net P sqp' =*
L j ∗ (inverse sqp − inverse (grd P (j+1))) +
sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))
{i. L i ≠ 0 ∧ j <i ∧ i < k} +
L k ∗ (inverse (grd P k) − inverse sqp')
using *assms base-net-diff-eq L-def* **by** *simp*
also have *... = L j ∗ (inverse sqp − inverse (grd P (j+1))) +*
L k ∗ (inverse (grd P k) − inverse sqp')
proof −
have *{i. L i ≠ 0 ∧ j <i ∧ i < k} = {}*
using *clmm-quote-gross-eq-sum-emp assms L-def clmm-dsc-gross-liq-zero-iff*
by *presburger*
hence *sum (λ i. L i ∗ (inverse (grd P i) − inverse (grd P (i+1))))*
{i. L i ≠ 0 ∧ j <i ∧ i < k} = 0
by (*metis* (*full-types*) *sum-clauses*(*1*))
thus *?thesis* **by** *simp*
qed
also have *... = L k ∗ (inverse (grd P k) − inverse sqp')*
proof −
have *L' j = 0* **using** *assms L'-def clmm-quote-gross-eq-lower-only-if* **by** *simp*
hence *L j = 0*
using *L-def L'-def clmm-dsc-gross-liq-zero-iff assms* **by** *simp*
thus *?thesis* **by** *simp*
qed
also have *... = 0*
proof −
have *L k ∗ (inverse (grd P k) − inverse sqp') = 0*

```
      using clmm-quote-gross-eq-upper-only-if assms L-def
        clmm-dsc-gross-liq-zero-iff
      by auto
    thus ?thesis by simp
  qed
  finally show ?thesis by simp
qed

lemma base-net-eq-only-if':
  assumes clmm-dsc P
  and L = lq P
  and j = lower-tick P sqp
  and j = lower-tick P sqp'
  and 0 < sqp
  and sqp ≤ sqp'
  and quote-gross P sqp = quote-gross P sqp'
shows base-net P sqp = base-net P sqp'
proof −
  have base-net P sqp − base-net P sqp' = L j ∗ (inverse sqp − inverse sqp')
    using base-net-diff-eq' assms by simp
  also have ... = 0
  proof (cases sqp = sqp')
    case True
    then show ?thesis by simp
  next
    case False
    hence gross-fct (lq P) (fee P) j = 0
      using assms clmm-quote-gross-eq-lower-only-if' by simp
    hence L j = 0 using assms
      by (simp add: clmm-dsc-gross-liq-zero-iff)
    then show ?thesis by simp
  qed
  finally show ?thesis by simp
qed

lemma quote-gross-equiv-base-net:
  assumes clmm-dsc P
  and 0 < sqp
  and sqp ≤ sqp'
  and quote-gross P sqp = quote-gross P sqp'
shows base-net P sqp = base-net P sqp'
proof (cases lower-tick P sqp = lower-tick P sqp')
  case True
  then show ?thesis
    using assms base-net-eq-only-if'[of P lq P - sqp  sqp'] by simp
next
  case False
  hence lower-tick P sqp < lower-tick P sqp'
    by (meson antisym-conv3 assms(1−3) leD lower-tick-lt)
```

then show *?thesis* **using** *assms base-net-eq-only-if* **by** *simp*
qed

lemma *quote-gross-equiv-base-net'*:
assumes *clmm-dsc P*
and $0 < sqp$
and $0 < sqp'$
and *quote-gross P sqp = quote-gross P sqp'*
shows *base-net P sqp = base-net P sqp'*
proof (*cases* $sqp \leq sqp'$)
case *True*
thus *?thesis* **using** *quote-gross-equiv-base-net assms* **by** *simp*
next
case *False*
then show *?thesis* **using** *quote-gross-equiv-base-net assms*
by (*metis linorder-le-cases*)
qed

lemma (**in** *finite-nz-support*) *gen-quote-le-badd*:
assumes *clmm-dsc P*
and $\bigwedge i.\ 0 \leq L\ i$
and $0 < sqp$
and $sqp \leq sqp'$
shows *gen-quote-diff P L sqp sqp'*$/(sqp' * sqp') \leq$ *gen-base-diff P L sqp sqp'*
proof −
have $0 < sqp'$ **using** *assms* **by** *simp*
define j **where** $j =$ *lower-tick P sqp*
define k **where** $k =$ *lower-tick P sqp'*
show *?thesis*
proof (*cases* $j = k$)
case *True*
hence *gen-quote-diff P L sqp sqp' / (sqp' * sqp') =*
*L j * (sqp' − sqp)/ (sqp' * sqp')*
using *assms j-def k-def clmm-gen-quote-diff-eq'* **by** *simp*
also have ... $\leq$ *L j * (inverse sqp − inverse sqp')*
by (*rule diff-inv-le'*, (*auto simp add*: *assms*))
also have ... = *gen-base-diff P L sqp sqp'*
using *assms j-def k-def gen-base-diff-eq' True* **by** *simp*
finally show *?thesis* **.**
next
case *False*
hence $j < k$ **using** *j-def k-def lower-tick-mono assms* **by** *fastforce*
hence *gen-quote-diff P L sqp sqp' / (sqp' * sqp') =*
*(L k * (sqp' − grd P k) +*
($\sum i \mid L\ i \neq 0 \wedge j < i \wedge i < k.$ *L i * (grd P (i + 1) − grd P i)) +*
*L j * (grd P (j + 1) − sqp))/ (sqp' * sqp')*
using *assms j-def k-def clmm-gen-quote-diff-eq* **by** *simp*
also have ... = *L k * (sqp' − grd P k) / (sqp' * sqp') +*
($\sum i \mid L\ i \neq 0 \wedge j < i \wedge i < k.$

```
    L i * (grd P (i + 1) - grd P i)) / (sqp' * sqp') +
    L j * (grd P (j + 1) - sqp) / (sqp' * sqp')
  by (simp add: add-divide-distrib)
also have ... ≤ L k * (inverse (grd P k) - inverse sqp')  +
    (∑ i | L i ≠ 0 ∧ j < i ∧ i < k.
    L i * (inverse (grd P i) - inverse (grd P (i + 1)))) +
    L j * (inverse sqp - inverse (grd P (j + 1)))
proof -
  have (∑ i | L i ≠ 0 ∧ j < i ∧ i < k.
    L i * (grd P (i + 1) - grd P i)) / (sqp' * sqp') ≤
    (∑ i | L i ≠ 0 ∧ j < i ∧ i < k.
      L i * (inverse (grd P i) - inverse (grd P (i + 1))))
  proof (rule diff-inv-sum-le')
    show ∀ i∈{i. L i ≠ 0 ∧ j < i ∧ i < k}. 0 < grd P i using assms by simp
    show ∀ i∈{i. L i ≠ 0 ∧ j < i ∧ i < k}. 0 ≤ L i using assms by simp
    show ∀ i∈{i. L i ≠ 0 ∧ j < i ∧ i < k}. grd P i ≤ grd P (i + 1)
      using assms clmm-dsc-grd-Suc[of P] less-eq-real-def by blast
    {
      fix i
      assume i∈{i. L i ≠ 0 ∧ j < i ∧ i < k}
      hence i+1 ≤ k by simp
      hence grd P (i+1) ≤ grd P k
        using assms clmm-dsc-grid(1) strict-mono-leD by blast
      hence grd P (i + 1) ≤ sqp'
        using k-def lower-tick-lbound assms ‹0 < sqp'› by fastforce
    }
    thus ∀ i∈{i. L i ≠ 0 ∧ j < i ∧ i < k}. grd P (i + 1) ≤ sqp' by simp
  qed
  moreover have L k * (sqp' - grd P k) / (sqp' * sqp') ≤
      L k * (inverse (grd P k) - inverse sqp')
  proof (rule diff-inv-le')
    show grd P k ≤ sqp' using k-def lower-tick-lbound assms by simp
  qed (simp add: assms)+
  moreover have L j * (grd P (j + 1) - sqp) / (sqp' * sqp') ≤
      L j * (inverse sqp - inverse (grd P (j + 1)))
  proof (rule diff-inv-le')
    show sqp ≤ grd P (j + 1)
      using j-def lower-tick-ubound assms by fastforce
    have j+1 ≤ k using ‹j < k› by simp
    hence grd P (j+1) ≤ grd P k using assms
      by (simp add: strict-mono-leD)
    thus grd P (j + 1) ≤ sqp'
      using ‹j < k› k-def lower-tick-lbound assms ‹0 < sqp'› by fastforce
  qed (simp add: assms)+
  ultimately show ?thesis by simp
qed
also have ... =  L j * (inverse sqp - inverse (grd P (j + 1))) +
    (∑ i | L i ≠ 0 ∧ j < i ∧ i < k.
    L i * (inverse (grd P i) - inverse (grd P (i + 1)))) +
```

```
      L k * (inverse (grd P k) - inverse sqp')
    by simp
  also have ... = gen-base-diff P L sqp sqp'
    using assms ‹j < k› j-def k-def gen-base-diff-eq by simp
  finally show ?thesis .
 qed
qed

lemma (in finite-nz-support) gen-base-le-qadd:
 assumes clmm-dsc P
 and ⋀i. 0 ≤ L i
 and 0 < sqp
 and sqp ≤ sqp'
shows gen-base-diff P L sqp sqp' ≤ gen-quote-diff P L sqp sqp'/(sqp * sqp)
proof -
 define j where j = lower-tick P sqp
 define k where k = lower-tick P sqp'
 show ?thesis
 proof (cases j = k)
  case True
  hence gen-base-diff P L sqp sqp' = L j * (inverse sqp - inverse sqp')
    using assms j-def k-def gen-base-diff-eq' True by simp
  also have ... ≤  L j * (sqp' - sqp)/ (sqp * sqp)
    by (rule diff-inv-ge', (auto simp add: assms))
  also have ... = gen-quote-diff P L sqp sqp' / (sqp * sqp)
    using assms j-def k-def clmm-gen-quote-diff-eq' True by simp
  finally show ?thesis .
 next
  case False
  hence j < k using j-def k-def lower-tick-mono assms by fastforce
  hence gen-base-diff P L sqp sqp' =
      L j * (inverse sqp - inverse (grd P (j + 1))) +
      (∑ i | L i ≠ 0 ∧ j < i ∧ i < k.
      L i * (inverse (grd P i) - inverse (grd P (i + 1)))) +
      L k * (inverse (grd P k) - inverse sqp')
    using assms j-def k-def gen-base-diff-eq by simp
  also have ... = L k * (inverse (grd P k) - inverse sqp')  +
      (∑ i | L i ≠ 0 ∧ j < i ∧ i < k.
      L i * (inverse (grd P i) - inverse (grd P (i + 1)))) +
      L j * (inverse sqp - inverse (grd P (j + 1)))
    by simp
  also have ... ≤ L k * (sqp' - grd P k) / (sqp * sqp) +
      (∑ i | L i ≠ 0 ∧ j < i ∧ i < k.
      L i * (grd P (i + 1) - grd P i)) / (sqp * sqp) +
      L j * (grd P (j + 1) - sqp) / (sqp * sqp)
  proof -
   have (∑ i | L i ≠ 0 ∧ j < i ∧ i < k.
       L i * (inverse (grd P i) - inverse (grd P (i + 1)))) ≤
       (∑ i | L i ≠ 0 ∧ j < i ∧ i < k.
```

```
      L i * (grd P (i + 1) - grd P i)) / (sqp * sqp)
    proof (rule diff-inv-sum-ge')
      show 0 < sqp using assms by simp
      show ∀ i∈{i. L i ≠ 0 ∧ j < i ∧ i < k}. 0 ≤ L i using assms by simp
      show ∀ i∈{i. L i ≠ 0 ∧ j < i ∧ i < k}. grd P i ≤ grd P (i + 1)
        using assms clmm-dsc-grd-Suc[of P] less-eq-real-def by blast
      {
        fix i
        assume i∈{i. L i ≠ 0 ∧ j < i ∧ i < k}
        hence j + 1 ≤ i by simp
        hence grd P (j+1) ≤ grd P i using assms clmm-dsc-grid(1)
          by (simp add: strict-mono-leD)
        hence sqp ≤ grd P i
          using assms j-def lower-tick-ubound by fastforce
      }
      thus ∀ i∈{i. L i ≠ 0 ∧ j < i ∧ i < k}. sqp ≤ grd P i by simp
    qed
    moreover have L k * (inverse (grd P k) - inverse sqp') ≤
      L k * (sqp' - grd P k) / (sqp * sqp)
    proof (rule diff-inv-ge')
      show grd P k ≤ sqp' using k-def lower-tick-lbound assms by simp
      have j+1 ≤ k using ‹j < k› by simp
      hence grd P (j+1) ≤ grd P k using clmm-dsc-grd-mono assms by simp
      thus sqp ≤ grd P k using j-def lower-tick-ubound assms by fastforce
    qed (simp add: assms)+
    moreover have L j * (inverse sqp - inverse (grd P (j + 1))) ≤
      L j * (grd P (j + 1) - sqp) / (sqp * sqp)
    proof (rule diff-inv-ge')
      show sqp ≤ grd P (j + 1)
        using j-def lower-tick-ubound assms by fastforce
    qed (simp add: assms)+
    ultimately show ?thesis by simp
  qed
  also have ... = (L k * (sqp' - grd P k) +
    (∑ i | L i ≠ 0 ∧ j < i ∧ i < k. L i * (grd P (i + 1) - grd P i)) +
    L j * (grd P (j + 1) - sqp))/ (sqp * sqp)
    by (simp add: add-divide-distrib)
  also have ... = gen-quote-diff P L sqp sqp' / (sqp * sqp)
    using assms j-def k-def clmm-gen-quote-diff-eq ‹j < k› by simp
  finally show ?thesis .
 qed
qed

lemma quote-swap-grd-min-zero:
 assumes clmm-dsc P
 and nz-support (lq P) ≠ {}
 and grd-min P ≤ sqp
 and sqp ≤ grd-max P
 shows quote-swap P sqp 0 = 0
```

proof −
define *sqp′* **where** *sqp′* = *quote-reach P* (*0* + *quote-gross P sqp*)
have *base-net P sqp* = *base-net P sqp′*
proof (*rule quote-gross-equiv-base-net*[*symmetric*])
show *clmm-dsc P* **using** *assms* **by** *simp*
show *0* < *sqp′*
proof (*rule clmm-quote-reach-pos*)
show *clmm-dsc P* **using** *assms* **by** *simp*
show *nz-support* (*lq P*) ≠ {} **using** *assms* **by** *simp*
show *0* ≤ *0* + *quote-gross P sqp*
by (*simp add*: *assms*(*1*) *clmm-quote-gross-pos*)
show *sqp′* = *quote-reach P* (*0* + *quote-gross P sqp*)
using *sqp′-def* **by** *simp*
show *0* + *quote-gross P sqp* ≤ *quote-gross P* (*grd-max P*)
by (*metis add-0 assms*(*1*) *assms*(*4*) *quote-gross-imp-sqp-lt*
less-eq-real-def nle-le)
qed
show *sqp′* ≤ *sqp* **using** *sqp′-def clmm-quote-reach-le assms*
by (*metis add-0 clmm-quote-gross-pos clmm-quote-reach-zero*
order-le-imp-less-or-eq vimage-singleton-eq)
show *quote-gross P sqp′* = *quote-gross P sqp*
by (*simp add*: *assms*(*1*) *assms*(*2*) *clmm-quote-gross-pos*
quote-gross-grd-max-max clmm-quote-gross-reach-eq sqp′-def)
qed
thus *?thesis* **unfolding** *quote-swap-def sqp′-def* **by** *simp*
qed

lemma *quote-swap-zero*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *0*< *sqp*
and *sqp* ≤ *grd-max P*
shows *quote-swap P sqp 0* = *0*
proof (*cases sqp* < *grd-min P*)
case *True*
hence *a*: *base-net P sqp* = *base-net P* (*grd-min P*)
by (*simp add*: *assms*(*1*) *assms*(*3*) *base-net-grd-min-lt*)
have *quote-gross P sqp* = *0* **using** *True*
by (*simp add*: *assms*(*1*) *assms*(*3*) *clmm-quote-gross-grd-min-le*)
hence *quote-reach P* (*0* + *quote-gross P sqp*) = *grd-min P*
using *clmm-quote-reach-zero assms* **by** *simp*
then show *?thesis* **using** *a* **unfolding** *quote-swap-def* **by** *simp*
next
case *False*
then show *?thesis* **using** *assms quote-swap-grd-min-zero* **by** *simp*
qed

lemma *quote-swap-grd-min-zero′*:
assumes *clmm-dsc P*

```
  and nz-support (lq P) ≠ {}
  and grd-min P ≤ sqp
  and quote-gross P sqp ≤ quote-gross P (grd-max P)
  shows quote-swap P sqp 0 = 0
proof −
  define sqp′ where sqp′ = quote-reach P (0 + quote-gross P sqp)
  have base-net P sqp = base-net P sqp′
  proof (rule quote-gross-equiv-base-net[symmetric])
    show clmm-dsc P using assms by simp
    show 0 < sqp′
    proof (rule clmm-quote-reach-pos)
      show clmm-dsc P using assms by simp
      show nz-support (lq P) ≠ {} using assms by simp
      show 0 ≤ 0 + quote-gross P sqp
        by (simp add: assms(1) clmm-quote-gross-pos)
      show sqp′ = quote-reach P (0 + quote-gross P sqp)
        using sqp′-def by simp
      show 0 + quote-gross P sqp ≤ quote-gross P (grd-max P)
        using assms by simp
    qed
    show sqp′ ≤ sqp using sqp′-def clmm-quote-reach-le assms
      by (metis add-0 clmm-quote-gross-pos clmm-quote-reach-zero
          order-le-imp-less-or-eq vimage-singleton-eq)
    show quote-gross P sqp′ = quote-gross P sqp
      by (simp add: assms(1) assms(2) clmm-quote-gross-pos
          quote-gross-grd-max-max clmm-quote-gross-reach-eq sqp′-def)
  qed
  thus ?thesis unfolding quote-swap-def sqp′-def by simp
qed

lemma quote-swap-zero′:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and 0< sqp
  and quote-gross P sqp ≤ quote-gross P (grd-max P)
shows quote-swap P sqp 0 = 0
proof (cases sqp < grd-min P)
  case True
  hence a: base-net P sqp = base-net P (grd-min P)
    by (simp add: assms(1) assms(3) base-net-grd-min-lt)
  have quote-gross P sqp = 0 using True
    by (simp add: assms(1) assms(3) clmm-quote-gross-grd-min-le)
  hence quote-reach P (0 + quote-gross P sqp) = grd-min P
    using clmm-quote-reach-zero assms by simp
  then show ?thesis using a unfolding quote-swap-def by simp
next
  case False
  then show ?thesis using assms quote-swap-grd-min-zero′ by simp
qed
```

lemma *quote-swap-grd-min*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *sqp* < *grd-min P*
and *0* < *sqp*
shows *quote-swap P sqp y* = *quote-swap P* (*grd-min P*) *y*
proof −
have *quote-gross P sqp* = *quote-gross P* (*grd-min P*) **using** *assms*
by (*simp add*: *clmm-quote-gross-grd-min-le*)
moreover have *base-net P sqp* = *base-net P* (*grd-min P*)
using *base-net-grd-min-lt assms* **by** *simp*
ultimately show *?thesis* **unfolding** *quote-swap-def* **by** *simp*
qed

lemma *quote-reach-gross-base-net*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *0* < *quote-gross P sqp*
and *sqp′* = *quote-reach P* (*quote-gross P sqp*)
shows *base-net P sqp′* = *base-net P sqp*
proof (*rule quote-gross-equiv-base-net*)
have *quote-gross P sqp* ≤ *quote-gross P* (*grd-max P*)
using *quote-gross-grd-max-max assms* **by** *simp*
thus *quote-gross P sqp′* = *quote-gross P sqp*
using *clmm-quote-gross-reach-eq assms* **by** *simp*
show *clmm-dsc P* **using** *assms* **by** *simp*
show *0* < *sqp′*
proof (*rule clmm-quote-reach-pos*)
show *clmm-dsc P* **using** *assms* **by** *simp*
show *nz-support* (*lq P*) ≠ {} **using** *assms* **by** *simp*
show *sqp′* = *quote-reach P* (*quote-gross P sqp*)
using *assms* **by** *simp*
show *0* ≤ *quote-gross P sqp*
by (*simp add*: *assms*(*1*) *clmm-quote-gross-pos*)
show *quote-gross P sqp* ≤ *quote-gross P* (*grd-max P*)
using *assms quote-gross-grd-max-max* **by** *simp*
qed
show *sqp′* ≤ *sqp* **using** *assms clmm-quote-reach-le* **by** *simp*
qed

lemma *quote-reach-base-net*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *0* < *sqp*
and *sqp′* = *quote-reach P* (*quote-gross P sqp*)
shows *base-net P sqp′* = *base-net P sqp*
proof (*cases quote-gross P sqp* = *0*)
case *True*

```
  hence sqp' = grd-min P
    by (simp add: assms clmm-quote-reach-zero)
  have base-net P sqp = base-net P sqp'
  proof (rule quote-gross-equiv-base-net)
    show sqp ≤ sqp' using ‹sqp' = grd-min P› True
      by (metis assms(1) assms(2) linorder-not-less quote-gross-gt-grd-min
          verit-comp-simplify1(1))
    show clmm-dsc P using assms by simp
    show 0 < sqp using assms by simp
    show quote-gross P sqp = quote-gross P sqp' using True
     by (simp add: ‹sqp' = grd-min P› assms(1) assms(2) clmm-quote-gross-grd-min)
  qed
  thus ?thesis by simp
next
  case False
  hence 0 < quote-gross P sqp
    by (meson assms(1) clmm-quote-gross-pos leI order-antisym-conv)
  then show ?thesis using assms quote-reach-gross-base-net by simp
qed

lemma base-le-quote-gross:
  assumes clmm-dsc P'
  and 0 < sqp
  and sqp ≤ sqp'
  shows base-gross P' sqp − base-gross P' sqp' ≤
    (quote-gross P' sqp' − quote-gross P' sqp)/(sqp ∗ sqp)
proof −
  define L where L = gross-fct (lq P') (fee P')
  have base-gross P' sqp − base-gross P' sqp' = gen-base-diff P' L sqp sqp'
    using gen-base-diff-def base-gross-def L-def by simp
  also have ... ≤ gen-quote-diff P' L sqp sqp' / (sqp ∗ sqp)
  proof (rule finite-nz-support.gen-base-le-qadd)
    show finite-nz-support L
      using L-def assms(1) clmm-dsc-gross-liq clmm-dsc-liq(1) finite-liq-def
        finite-nz-support.intro
      by auto
    show ⋀i. 0 ≤ L i using L-def assms(1) gross-liq-ge by simp
  qed (simp add: assms)+
  also have ... = (quote-gross P' sqp' − quote-gross P' sqp)/(sqp ∗ sqp)
    using gen-quote-diff-def quote-gross-def L-def by simp
  finally show ?thesis .
qed

lemma quote-le-base-gross:
  assumes clmm-dsc P'
  and 0 < sqp
  and sqp ≤ sqp'
shows (quote-gross P' sqp' − quote-gross P' sqp)/(sqp' ∗ sqp') ≤
    base-gross P' sqp − base-gross P' sqp'
```

proof −
define L **where** $L = \textit{gross-fct}\ (\textit{lq}\ P')\ (\textit{fee}\ P')$
have $(\textit{quote-gross}\ P'\ \textit{sqp}' - \textit{quote-gross}\ P'\ \textit{sqp})/(\textit{sqp}' * \textit{sqp}') =$
$\textit{gen-quote-diff}\ P'\ L\ \textit{sqp}\ \textit{sqp}'\ /\ (\textit{sqp}' * \textit{sqp}')$
using *gen-quote-diff-def quote-gross-def L-def* **by** *simp*
also have $\ldots \leq \textit{gen-base-diff}\ P'\ L\ \textit{sqp}\ \textit{sqp}'$
proof (*rule finite-nz-support.gen-quote-le-badd*)
show *finite-nz-support L*
using *L-def assms*(*1*) *clmm-dsc-gross-liq clmm-dsc-liq*(*1*) *finite-liq-def*
finite-nz-support.intro
by *auto*
show $\bigwedge i.\ 0 \leq L\ i$ **using** *L-def assms*(*1*) *gross-liq-ge* **by** *simp*
qed (*simp add*: *assms*)+
also have $\ldots = \textit{base-gross}\ P'\ \textit{sqp} - \textit{base-gross}\ P'\ \textit{sqp}'$
using *gen-base-diff-def base-gross-def L-def* **by** *simp*
finally show *?thesis* **.**
qed

lemma *base-net-quote-ubound*:
assumes *clmm-dsc P'*
and $\bigwedge i.\ \textit{fee}\ P'\ i = \textit{phi}$
and $\textit{phi} < 1$
and $0 < \textit{sqp}$
and $\textit{sqp} \leq \textit{sqp}'$
shows $\textit{base-net}\ P'\ \textit{sqp} - \textit{base-net}\ P'\ \textit{sqp}' \leq$
$(1 - \textit{phi}) * (\textit{quote-gross}\ P'\ \textit{sqp}' - \textit{quote-gross}\ P'\ \textit{sqp})/(\textit{sqp} * \textit{sqp})$
proof −
have $\textit{base-net}\ P'\ \textit{sqp} - \textit{base-net}\ P'\ \textit{sqp}' =$
$(1 - \textit{phi}) * (\textit{base-gross}\ P'\ \textit{sqp} - \textit{base-gross}\ P'\ \textit{sqp}')$
by (*rule cst-fee-base-gross-net*, (*auto simp add*: *assms*))
also have $\ldots \leq (1 - \textit{phi}) * (\textit{quote-gross}\ P'\ \textit{sqp}' - \textit{quote-gross}\ P'\ \textit{sqp})/(\textit{sqp} * \textit{sqp})$
using *base-le-quote-gross assms*
by (*metis ge-iff-diff-ge-0 less-eq-real-def mult-left-mono*
times-divide-eq-right)
finally show *?thesis* **.**
qed

lemma *base-net-quote-lbound*:
assumes *clmm-dsc P'*
and $\bigwedge i.\ \textit{fee}\ P'\ i = \textit{phi}$
and $0 < \textit{sqp}$
and $\textit{sqp} \leq \textit{sqp}'$
shows $(1 - \textit{phi}) * (\textit{quote-gross}\ P'\ \textit{sqp}' - \textit{quote-gross}\ P'\ \textit{sqp})/(\textit{sqp}' * \textit{sqp}') \leq$
$\textit{base-net}\ P'\ \textit{sqp} - \textit{base-net}\ P'\ \textit{sqp}'$
proof −
have $\textit{phi} < 1$ **using** *assms* **by** (*metis clmm-dsc-fees*)
hence $(1 - \textit{phi}) * (\textit{quote-gross}\ P'\ \textit{sqp}' - \textit{quote-gross}\ P'\ \textit{sqp})/(\textit{sqp}' * \textit{sqp}') \leq$
$(1 - \textit{phi}) * (\textit{base-gross}\ P'\ \textit{sqp} - \textit{base-gross}\ P'\ \textit{sqp}')$

using *quote-le-base-gross assms*
by (*metis diff-gt-0-iff-gt mult-le-cancel-left-pos times-divide-eq-right*)
also have ... = *base-net P′ sqp − base-net P′ sqp′*
by (*rule cst-fee-base-gross-net*[*symmetric*], (*auto simp add*: *assms*))
finally show *?thesis* **.**
qed

4.4 Market depth and slippage for finer CLMMs

4.4.1 Finer pools

locale *finer-clmm* =
fixes *P1 P2*
assumes *abs1*: *clmm-dsc P1* **and** *abs2*: *clmm-dsc P2*
and *finer*: *finer-pool P1 P2*

sublocale *finer-clmm* ⊆ *finer-two-span-finite-liq*
by (*meson abs1 abs2 clmm-dsc-def finer finer-pools.intro*
finer-spanning-pool.intro finer-spanning-pool-axioms.intro
finer-two-span-finite-liq.intro finer-two-span-finite-liq-axioms.intro
finer-two-spanning-pools.intro finer-two-spanning-pools-axioms.intro)

context *finer-clmm*
begin

lemma *finer-base-net-eq*:
shows *base-net P1* = *base-net P2*
proof −
have ⋀*a b*. *a* ∈ *encompassed* (*grd P1*) (*grd P2*) *b* ⟹ *lq P1 a* = *lq P2 b*
using *encompassed-liq-eq*
by (*simp add*: *mon stm*)
thus *?thesis* **unfolding** *base-net-def*
using *spanning-finer-gen-base-eq*[*of* λ*x*. *x* λ*x*. *x*]
clmm-dsc-grid clmm-dsc-liq **by** *blast*
qed

lemma *finer-quote-net-eq*:
shows *quote-net P1* = *quote-net P2*
proof −
have ⋀*a b*. *a* ∈ *encompassed* (*grd P1*) (*grd P2*) *b* ⟹ *lq P1 a* = *lq P2 b*
using *encompassed-liq-eq*
by (*simp add*: *mon stm*)
thus *?thesis* **unfolding** *quote-net-def*
using *spanning-finer-gen-quote-eq*[*of* λ*x*. *x* λ*x*. *x*]
clmm-dsc-grid clmm-dsc-liq **by** *blast*
qed

lemma *finer-base-gross-eq*:
shows *base-gross P1* = *base-gross P2*
proof −

```
  have base-gross P1 = gen-base (grd P2) ((λx. gross-fct x (fee P2)) (lq P2))
    unfolding base-gross-def
  proof
    fix x
    show gen-base (grd P1) ((λx. gross-fct x (fee P1)) (lq P1)) x =
      gen-base (grd P2) ((λx. gross-fct x (fee P2)) (lq P2)) x
    proof (rule spanning-finer-gen-base-eq)
      show ⋀i. lq P2 i = 0 ⟹ gross-fct (lq P2) (fee P2) i = 0
        using gross-fct-zero-if by simp
      show ⋀i. lq P1 i = 0 ⟹ gross-fct (lq P1) (fee P1) i = 0
        using gross-fct-zero-if by simp
      show ⋀a b. a ∈ encompassed (grd P1) (grd P2) b ⟹
        gross-fct (lq P1) (fee P1) a = gross-fct (lq P2) (fee P2) b
      proof −
        fix a b
        assume asm: a ∈ encompassed (grd P1) (grd P2) b
        hence lq P1 a = lq P2 b
          by (simp add: span span2 encompassed-liq-eq
              span-gridD(1) strict-mono-mono)
        moreover have fee P1 a = fee P2 b
          by (simp add: asm span span2 encompassed-fee-eq span-gridD(1)
              strict-mono-mono)
        ultimately show gross-fct (lq P1) (fee P1) a =
          gross-fct (lq P2) (fee P2) b
          using gross-fct-cong by metis
      qed
    qed
  qed
  also have ... = base-gross P2 unfolding base-gross-def by simp
  finally show ?thesis .
qed

lemma finer-quote-gross-eq:
shows quote-gross P1 = quote-gross P2
proof −
  have quote-gross P1 = gen-quote (grd P2) ((λx. gross-fct x (fee P2)) (lq P2))
    unfolding quote-gross-def
  proof
    fix x
    show gen-quote (grd P1) ((λx. gross-fct x (fee P1)) (lq P1)) x =
      gen-quote (grd P2) ((λx. gross-fct x (fee P2)) (lq P2)) x
    proof (rule spanning-finer-gen-quote-eq)
      show ⋀i. lq P2 i = 0 ⟹ gross-fct (lq P2) (fee P2) i = 0
        using gross-fct-zero-if by simp
      show ⋀i. lq P1 i = 0 ⟹ gross-fct (lq P1) (fee P1) i = 0
        using gross-fct-zero-if by simp
      show ⋀a b. a ∈ encompassed (grd P1) (grd P2) b ⟹
        gross-fct (lq P1) (fee P1) a = gross-fct (lq P2) (fee P2) b
      proof −
```

fix *a b*
assume *asm*: *a* ∈ *encompassed* (*grd P1*) (*grd P2*) *b*
hence *lq P1 a* = *lq P2 b*
by (*simp add*: *span span2 encompassed-liq-eq span-gridD*(*1*) *strict-mono-mono*)
moreover have *fee P1 a* = *fee P2 b*
by (*simp add*: *asm span span2 encompassed-fee-eq span-gridD*(*1*) *strict-mono-mono*)
ultimately show *gross-fct* (*lq P1*) (*fee P1*) *a* =
gross-fct (*lq P2*) (*fee P2*) *b*
using *gross-fct-cong* **by** *metis*
qed
qed
qed
also have ... = *quote-gross P2* **unfolding** *quote-gross-def* **by** *simp*
finally show *?thesis* **.**
qed

lemma *finer-mkt-depth*:
shows *mkt-depth P1* = *mkt-depth P2*
using *finer-base-net-eq finer-quote-net-eq* **unfolding** *mkt-depth-def* **by** *presburger*

end

4.4.2 Finer CLMMs with nonzero liquidity

locale *finer-clmm-ne* = *finer-clmm* +
assumes *nonempty-liq*: *nz-support* (*lq P1*) ≠ {}

context *finer-clmm-ne*
begin

lemma *id-max-Max-eq*:
assumes *i1* = *idx-max* (*lq P1*)
and *k2* = *pool-coarse-idx P1 P2 i1*
shows *i1* = *Max* (*encompassed* (*grd P1*) (*grd P2*) *k2*)
proof (*rule ccontr*)
assume *asm*: *i1* ≠ *Max* (*encompassed* (*grd P1*) (*grd P2*) *k2*)
interpret *finer-ranges grd P1 grd P2*
proof (*rule finer-ranges.intro*)
show *strict-mono* (*grd P1*) **using** *span span-gridD* **by** *simp*
show *mono* (*grd P2*) **using** *span2*
by (*simp add*: *span-gridD strict-mono-on-imp-mono-on*)
show *finer-range* (*grd P1*) (*grd P2*) **using** *assms*
by (*simp add*: *finer-pool-grid*)
qed
have *lq P1* (*i1* + *1*) = *0* **using** *idx-max-finite-gt assms clmm-dsc-liq finite-liqD nz-support-def nonempty-liq*

```
    by (metis less-add-one fin-liq)
  have i1 ∈ encompassed (grd P1) (grd P2) k2
    using encompassed-bounds assms pool-coarse-idxD by auto
  hence i1 < Max (encompassed (grd P1) (grd P2) k2) using asm
    by (meson Max.coboundedI fin linorder-less-linear linorder-not-less
        span-grid-encompassed)
  hence i1 + 1 ≤ Max (encompassed (grd P1) (grd P2) k2) by simp
  have Max (encompassed (grd P1) (grd P2) k2) ∈
    encompassed (grd P1) (grd P2) k2
    by (metis Max-in ‹i1 ∈ encompassed (grd P1) (grd P2) k2›
        emptyE span-grid-encompassed)
  hence i1 + 1 ∈ encompassed (grd P1) (grd P2) k2
    using ‹i1 ∈ encompassed (grd P1) (grd P2) k2› encompassed-convex
    ‹i1 + 1 ≤ Max (encompassed (grd P1) (grd P2) k2)› stm strict-mono-mono
    by fastforce
  hence lq P1 (i1 + 1) = lq P2 k2
    by (simp add: assms(2) encompassed-liq-eq mon stm)
  also have ... = lq P1 i1
    using ‹i1 ∈ encompassed (grd P1) (grd P2) k2› assms finer-pool-liq
    by auto
  also have ... ≠ 0 using span assms nonempty-liq fin-liq finite-liq-def
      idx-max-finite-in by blast
  finally show False using ‹lq P1 (i1 + 1) = 0› by simp
qed

lemma id-min-Min-eq:
  assumes i1 =  idx-min (lq P1)
  and k2 = pool-coarse-idx P1 P2 i1
shows i1 = Min (encompassed (grd P1) (grd P2) k2)
proof (rule ccontr)
  assume asm: i1 ≠ Min (encompassed (grd P1) (grd P2) k2)
  have lq P1 (i1 − 1) = 0 using  idx-min-finite-lt assms clmm-dsc-liq
      finite-liqD nz-support-def nonempty-liq
    by (metis order-refl fin-liq zle-diff1-eq)
  have i1 ∈ encompassed (grd P1) (grd P2) k2
    using encompassed-bounds assms pool-coarse-idxD by auto
  hence Min (encompassed (grd P1) (grd P2) k2) < i1 using asm
    by (meson Min.coboundedI fin linorder-less-linear linorder-not-less
        span-grid-encompassed)
  hence Min (encompassed (grd P1) (grd P2) k2) ≤ i1−1 by simp
  have Min (encompassed (grd P1) (grd P2) k2) ∈
    encompassed (grd P1) (grd P2) k2
    by (metis Min-in ‹i1 ∈ encompassed (grd P1) (grd P2) k2› emptyE
        span-grid-encompassed)
  hence i1 − 1 ∈ encompassed (grd P1) (grd P2) k2
    using ‹i1 ∈ encompassed (grd P1) (grd P2) k2› encompassed-convex
    ‹Min (encompassed (grd P1) (grd P2) k2) ≤ i1 − 1› stm strict-mono-mono
    by fastforce
  hence lq P1 (i1 − 1) = lq P2 k2
```

```
    by (simp add: assms(2) encompassed-liq-eq mon stm)
  also have ... = lq P1 i1
    using ‹i1 ∈ encompassed (grd P1) (grd P2) k2› assms finer-pool-liq
    by auto
  also have ... ≠ 0 using assms fin-liq finite-liq-def idx-min-finite-in
      nonempty-liq by blast
  finally show False using ‹lq P1 (i1 − 1) = 0› by simp
qed

lemma idx-max-Suc-grd-eq:
  assumes i1 =  idx-max (lq P1)
  and k2 = pool-coarse-idx P1 P2 i1
shows grd P1 (i1 + 1) = grd P2 (k2 + 1)
proof −
  have i1 = Max (encompassed (grd P1) (grd P2) k2)
    using id-max-Max-eq assms clmm-dsc-grid by simp
  hence grd P1 (i1 + 1) = grd P1 (Max (encompassed (grd P1) (grd P2) k2) +
1)
    by simp
  also have ... = grd P2 (k2+1)
  proof (rule encompassed-max-Suc-gamma-eq′)
    show ∃ m. grd P1 m ≤ grd P2 k2
      by (simp add: span-grids-ex-le)
    show ∃ M. grd P2 (k2 + 2) ≤ grd P1 M
      by (simp add: span-grids-ex-ge)
    show grd P2 k2 ≠ grd P2 (k2 + 1) using span2 span-gridD
      by (simp add: strict-mono-eq)
    show grd P2 (k2 + 1) ≠ grd P2 (k2 + 2) using span2 span-gridD
      by (simp add: strict-mono-eq)
  qed
  finally show ?thesis .
qed

lemma idx-min-grd-eq:
  assumes i1 =  idx-min (lq P1)
  and k2 = pool-coarse-idx P1 P2 i1
shows grd P1 i1 = grd P2 k2
  unfolding grd-max-def idx-max-img-def
proof −
  have i1 = Min (encompassed (grd P1) (grd P2) k2)
    using id-min-Min-eq assms clmm-dsc-grid by simp
  hence grd P1 i1 = grd P1 (Min (encompassed (grd P1) (grd P2) k2))
    by simp
  also have ... = grd P2 k2
  proof (rule encompassed-min-gamma-eq′)
    show ∃ m. grd P1 m ≤ grd P2 k2 by (simp add: span-grids-ex-le)
    show ∃ M. grd P2 (k2 + 1) ≤ grd P1 M by (simp add: span-grids-ex-ge)
    show grd P2 k2 ≠ grd P2 (k2 + 1) using span2 span-gridD
      by (simp add: strict-mono-eq)
```

```
  qed
  finally show ?thesis .
qed

lemma abs-finer-idx-max-coarse:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and finer-pool P1 P2
  and nz-support (lq P1) ≠ {}
  and i1 =  idx-max (lq P1)
  and k2 = pool-coarse-idx P1 P2 i1
shows k2 = idx-max (lq P2)
proof −
  define m2 where m2 = idx-max (lq P2)
  have lq P1 i1 ≠ 0
    using assms(5) fin-liq finite-liq-def idx-max-finite-in nonempty-liq by blast
  hence nz-support (lq P1) ≠ {} unfolding nz-support-def by auto
  have i1 ∈ encompassed (grd P1) (grd P2) k2
    using encompassed-bounds assms pool-coarse-idxD by auto
  hence lq P1 i1 = lq P2 k2
    by (simp add: assms finer-pool-liq)
  hence lq P2 k2 ≠ 0 using ‹lq P1 i1 ≠ 0› by simp
  hence nz-support (lq P2) ≠ {} unfolding nz-support-def by auto
  have finite-liq P1 using assms clmm-dsc-liq by simp
  have finite-liq P2 using assms clmm-dsc-liq by simp
  hence k2 ≤ m2 unfolding m2-def
    using ‹lq P2 k2 ≠ 0› idx-max-finite-ge finite-liq-def
    by metis
  have lq P2 m2 ≠ 0 using idx-max-finite-in m2-def
    by (simp add: ‹finite-liq P2› ‹nz-support (lq P2) ≠ {}› idx-max-mem
        nz-supportD)
  show k2 = m2
  proof (rule ccontr)
    assume k2 ≠ m2
    hence k2 < m2 using ‹k2 ≤m2› by auto
    have ∃j1. encomp-at (grd P1) (grd P2) j1 m2 using ex-coarse-rep
      by (metis Max-in encompassed-unique finer-ranges.coarse-idx-bounds
          finer-ranges-axioms span-grid-encompassed
          span-grids-encompassed-ne)
    from this obtain j1 where encomp-at (grd P1) (grd P2) j1 m2 by auto
    hence lq P1 j1 = lq P2 m2
      by (metis coarse-idx-bounds encomp-idx-unique finer-pool-liq
          pool-coarse-idxD)
    hence lq P1 j1 ≠ 0 using ‹lq P2 m2 ≠ 0› by simp
    hence j1 ∈ nz-support (lq P1) unfolding nz-support-def by simp
    hence j1 ≤ i1 using assms ‹lq P1 j1 ≠ 0› idx-max-finite-ge finite-liq-def
      by (metis ‹finite-liq P1›)
    moreover have i1 < j1
      using encomp-idx-mono-conv ‹encomp-at (grd P1) (grd P2) j1 m2›
```

```
      ‹k2 < m2› assms(6) coarse-idx-bounds pool-coarse-idxD by presburger
    ultimately show False by simp
  qed
qed

lemma abs-finer-idx-min-coarse:
  assumes i1 =  idx-min (lq P1)
  and k2 = pool-coarse-idx P1 P2 i1
shows k2 = idx-min (lq P2)
proof −
  define m2 where m2 = idx-min (lq P2)
  have lq P1 i1 ≠ 0
    using assms(1) fin-liq finite-liq-def idx-min-finite-in nonempty-liq by blast
  hence nz-support (lq P1) ≠ {} unfolding nz-support-def by auto
  have i1 ∈ encompassed (grd P1) (grd P2) k2
    using encompassed-bounds assms pool-coarse-idxD by auto
  hence lq P1 i1 = lq P2 k2
    by (simp add: assms finer-pool-liq)
  hence lq P2 k2 ≠ 0 using ‹lq P1 i1 ≠ 0› by simp
  hence nz-support (lq P2) ≠ {} unfolding nz-support-def by auto
  have finite-liq P1 using fin-liq by simp
  have finite-liq P2 using fin-liq by (simp add: span-grids-finite-liq')
  hence m2 ≤ k2 unfolding m2-def
    using ‹lq P2 k2 ≠ 0› idx-min-finite-le finite-liq-def
    by metis
  have lq P2 m2 ≠ 0 using idx-max-finite-in m2-def
    by (simp add: ‹finite-liq P2› ‹nz-support (lq P2) ≠ {}› idx-min-mem
        nz-supportD)
  show k2 = m2
  proof (rule ccontr)
    assume k2 ≠ m2
    hence m2 < k2 using ‹m2 ≤k2› by auto
    have ∃j1. encomp-at (grd P1) (grd P2) j1 m2 using ex-coarse-rep
      by (metis Max-in encompassed-unique finer-ranges.coarse-idx-bounds
          finer-ranges-axioms span-grid-encompassed
          span-grids-encompassed-ne)
    from this obtain j1 where encomp-at (grd P1) (grd P2) j1 m2 by auto
    hence lq P1 j1 = lq P2 m2
      using coarse-idx-bounds encomp-idx-unique finer-pool-liq pool-coarse-idxD
      by auto
    hence lq P1 j1 ≠ 0 using ‹lq P2 m2 ≠ 0› by simp
    hence j1 ∈ nz-support (lq P1) unfolding nz-support-def by simp
   hence i1 ≤ j1 using assms ‹lq P1 j1 ≠ 0› idx-min-finite-le finite-liq-def
      by (metis ‹finite-liq P1›)
    moreover have j1 < i1
      using encomp-idx-mono-conv ‹encomp-at (grd P1) (grd P2) j1 m2›
        ‹m2 < k2› assms coarse-idx-bounds pool-coarse-idxD by presburger
    ultimately show False by simp
  qed
```

qed

lemma *abs-finer-idx-max-img-eq*:
shows *grd-max P1 = grd-max P2*
proof −
  define *i1* **where** *i1 = idx-max* (*lq P1*)
  define *k2* **where** *k2 = pool-coarse-idx P1 P2 i1*
  have *k2 = idx-max* (*lq P2*)
    by (*simp add*: *abs1 abs2 abs-finer-idx-max-coarse finer i1-def k2-def*
        nonempty-liq)
  have *grd-max P1 = grd P2* (*k2 + 1*)
    unfolding *grd-max-def idx-max-img-def*
    using *idx-max-Suc-grd-eq i1-def k2-def span* **by** *simp*
  also have *... = grd-max P2* **using** ‹*k2 = idx-max* (*lq P2*)›
    unfolding *grd-max-def idx-max-img-def* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *abs-finer-idx-min-img-eq*:
shows *grd-min P1 = grd-min P2*
proof −
  define *i1* **where** *i1 = idx-min* (*lq P1*)
  define *k2* **where** *k2 = pool-coarse-idx P1 P2 i1*
  have *k2 = idx-min* (*lq P2*)
    by (*simp add*: *abs-finer-idx-min-coarse i1-def k2-def*)
  have *grd-min P1 = grd P2 k2*
    unfolding *grd-min-def idx-min-img-def*
    using *idx-min-grd-eq i1-def k2-def* **by** *simp*
  also have *... = grd-min P2* **using** ‹*k2 = idx-min* (*lq P2*)›
    unfolding *grd-min-def idx-min-img-def* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *finer-base-reach-eq*:
shows *base-reach P1 = base-reach P2* **unfolding** *base-reach-def*
  using *clmm-dsc-grid finer-base-gross-eq abs-finer-idx-max-img-eq* **by** *presburger*

lemma *finer-quote-reach-eq*:
shows *quote-reach P1 = quote-reach P2* **unfolding** *quote-reach-def*
  using *clmm-dsc-grid finer-quote-gross-eq abs-finer-idx-min-img-eq* **by** *presburger*

lemma *finer-base-slippage*:
shows *base-slippage P1 = base-slippage P2*
  unfolding *base-slippage-def base-swap-def*
  using *finer-quote-net-eq finer-base-reach-eq finer-base-gross-eq*
  by *simp*

lemma *finer-quote-slippage*:
shows *quote-slippage P1 = quote-slippage P2*

unfolding *quote-slippage-def quote-swap-def*
using *finer-base-net-eq finer-quote-reach-eq finer-quote-gross-eq*
by *simp*

end

5 Inequalities related to fees

context *finite-liq-pool*
begin

lemma *gross-fct-le*:
assumes $0 \leq f\ i$
and $phi\ i \leq phi'\ i$
and $phi'\ i < 1$
shows $gross\text{-}fct\ f\ phi\ i \leq gross\text{-}fct\ f\ phi'\ i$
unfolding *gross-fct-def one-cpl-def*
by (*metis assms diff-gt-0-iff-gt diff-left-mono frac-le linorder-not-less order.asym*)

lemma *gross-fct-lt*:
assumes $0 < f\ i$
and $phi\ i < phi'\ i$
and $phi'\ i < 1$
shows $gross\text{-}fct\ f\ phi\ i < gross\text{-}fct\ f\ phi'\ i$
unfolding *gross-fct-def one-cpl-def* **by** (*simp add*: *assms frac-less2*)

lemma *fee-diff-same-base-net*:
assumes *clmm-dsc P*
and *clmm-dsc P'*
and $I = \{k.\ L\ k \neq 0 \wedge j \leq k\}$
and *fee-diff-on P P' I*
and *same-nz-liq-on P P' {k. j ≤ k}*
and $0 < sqp$
and $j = lower\text{-}tick\ P\ sqp$
and $L = lq\ P$
and $lower\text{-}tick\ P\ sqp = lower\text{-}tick\ P'\ sqp$
shows $base\text{-}net\ P\ sqp = base\text{-}net\ P'\ sqp$
proof −
define L' **where** $L' = lq\ P'$
have *eq*: $\forall i \in I.\ L\ i = L'\ i$ **using** *assms L'-def* **by** *auto*
have $base\text{-}net\ P\ sqp = L\ j * (inverse\ sqp - inverse\ (grd\ P\ (j + 1))) + (\sum i \mid L\ i \neq 0 \wedge j < i.\ L\ i * (inverse\ (grd\ P\ i) - inverse\ (grd\ P\ (i + 1))))$
using *assms base-net-sum* **by** *simp*
also have ... $= L'\ j * (inverse\ sqp - inverse\ (grd\ P\ (j + 1))) + (\sum i \mid L\ i \neq 0 \wedge j < i.\ L\ i * (inverse\ (grd\ P\ i) - inverse\ (grd\ P\ (i + 1))))$
proof (*cases j∈ I*)
case *True*
then show *?thesis* **using** *eq* **by** *simp*

```
next
  case False
  hence L j = 0 using assms by simp
  then show ?thesis using L'-def assms(5,8) by auto
qed
also have ... = L' j * (inverse sqp − inverse (grd P (j + 1))) +
    (∑ i | L' i ≠ 0 ∧ j < i. L' i * (inverse (grd P i) − inverse (grd P (i + 1))))
proof −
  have (∑ i | L i ≠ 0 ∧ j< i.
      L i * (inverse (grd P i) − inverse (grd P (i + 1)))) =
      (∑ i | L' i ≠ 0 ∧ j < i.
      L' i * (inverse (grd P i) − inverse (grd P (i + 1))))
  proof (rule sum.cong)
    fix k
    assume k ∈ {i. L' i ≠ 0 ∧ j < i}
    hence k∈ I using assms L'-def same-nz-liq-on-def by auto
    thus L k * (inverse (grd P k) − inverse (grd P (k + 1))) =
        L' k * (inverse (grd P k) − inverse (grd P (k + 1)))
      using eq by simp
  next
    show {i. L i ≠ 0 ∧ j < i} = {i. L' i ≠ 0 ∧ j < i}
    proof
      show {i. L i ≠ 0 ∧ j < i} ⊆ {i. L' i ≠ 0 ∧ j < i}
        using assms(3) eq by auto
    next
      show {i. L' i ≠ 0 ∧ j < i} ⊆ {i. L i ≠ 0 ∧ j < i}
      proof
        fix k
        assume k ∈ {i. L' i ≠ 0 ∧ j < i}
        hence k∈ I using assms L'-def same-nz-liq-on-def by auto
        thus k ∈ {i. L i ≠ 0 ∧ j < i}
          using ‹k ∈ {i. L' i ≠ 0 ∧ j < i}› eq by auto
      qed
    qed
  qed
  thus ?thesis by simp
qed
also have ... = L' j * (inverse sqp − inverse (grd P' (j + 1))) +
    (∑ i | L' i ≠ 0 ∧ j < i.
    L' i * (inverse (grd P' i) − inverse (grd P' (i + 1))))
proof −
  have (∑ i | L' i ≠ 0 ∧ j < i.
      L' i * (inverse (grd P i) − inverse (grd P (i + 1)))) =
      (∑ i | L' i ≠ 0 ∧ j < i.
      L' i * (inverse (grd P' i) − inverse (grd P' (i + 1))))
  proof (rule sum.cong)
    fix x
    assume x ∈ {i. L' i ≠ 0 ∧ j < i}
    hence x ∈ {k. j ≤ k} by auto
```

hence *grd P x = grd P' x* **using** *assms* **by** (*simp add*: *same-nz-liq-onD*(*1*))
have *x+1* ∈ {*k*. *j* ≤ *k*} **using** ‹*x* ∈ {*i*. *L' i* ≠ *0* ∧ *j* < *i*}› **by** *auto*
hence *grd P* (*x* + *1*) = *grd P'* (*x* + *1*)
using *assms* **by** (*simp add*: *same-nz-liq-onD*(*1*))
thus *L' x* ∗ (*inverse* (*grd P x*) − *inverse* (*grd P* (*x* + *1*))) =
L' x ∗ (*inverse* (*grd P' x*) − *inverse* (*grd P'* (*x* + *1*)))
using ‹*grd P x = grd P' x*› **by** *simp*
qed *simp*
moreover have *grd P* (*j* + *1*) = *grd P'* (*j* + *1*)
using *same-nz-liq-onD*(*1*) *assms*(*5*) **by** *auto*
ultimately show *?thesis* **by** *simp*
qed
also have ... = *base-net P' sqp*
by (*rule base-net-sum*[*symmetric*], (*auto simp add*: *assms L'-def*))
finally show *?thesis* **.**
qed

lemma *fee-diff-le-imp-quote-gross*:
assumes *clmm-dsc P*
and *clmm-dsc P'*
and {*k*. *L k* ≠ *0* ∧ *k* ≤ *j*} ⊆ *I*
and *fee-diff-on P P' I*
and *same-nz-liq-on P P'* {*k*. *k* ≤ *j*}
and *0* < *sqp*
and *j* = *lower-tick P sqp*
and *L* = *gross-fct* (*lq P*) (*fee P*)
and ⋀*i*. *i* ∈ *I* ⟹ *fee P i* ≤ *fee P' i*
and *lower-tick P sqp* = *lower-tick P' sqp*
shows *quote-gross P sqp* ≤ *quote-gross P' sqp*
proof −
define *L'* **where** *L'* = *gross-fct* (*lq P'*) (*fee P'*)
have *pos*: ∀ *i*∈ *I*. *0* ≤ *L i* **using** *assms gross-liq-ge* **by** *simp*
have *le*: ∀ *i*∈ *I*. *L i* ≤ *L' i*
proof
fix *i*
assume *i* ∈ *I*
hence *lq P i* = *lq P' i* **using** *assms fee-diff-onD*(*2*) **by** *simp*
hence *L i* = *gross-fct* (*lq P'*) (*fee P*) *i*
using *assms*(*8*) *gross-fct-cong* **by** *blast*
also have ... ≤ *L' i* **unfolding** *L'-def*
proof (*rule gross-fct-le*)
show *0* ≤ *lq P' i* **by** (*simp add*: *assms*(*2*) *clmm-dsc-liq*(*2*))
show *fee P i* ≤ *fee P' i* **using** ‹*i* ∈ *I*› *assms* **by** *simp*
show *fee P' i* < *1* **by** (*simp add*: *assms*(*2*) *clmm-dsc-fees*)
qed
finally show *L i* ≤ *L' i* **.**
qed
have *quote-gross P sqp* = *L j* ∗ (*sqp* − *grd P j*) +
(∑ *i* | *L i* ≠ *0* ∧ *i* < *j*. *L i* ∗ (*grd P* (*i* + *1*) − *grd P i*))

```
  using assms clmm-quote-gross-sum by simp
also have ... ≤ L' j * (sqp - grd P j) +
    (∑ i | L i ≠ 0 ∧ i < j. L i * (grd P (i + 1) - grd P i))
proof (cases j∈ I)
  case True
  then show ?thesis using lower-tick-lbound  le pos
    by (simp add: assms(1) assms(6) assms(7) mult-right-mono)
next
  case False
  hence L j = 0 using assms by auto
  then show ?thesis
    using L'-def lower-tick-geq gross-liq-ge
    by (simp add: assms(1,2,6,7))
qed
also have ... ≤ L' j * (sqp - grd P j) +
    (∑ i | L i ≠ 0 ∧ i < j. L' i * (grd P (i + 1) - grd P i))
proof -
  have (∑ i | L i ≠ 0 ∧ i < j. L i * (grd P (i + 1) - grd P i)) ≤
      (∑ i | L i ≠ 0 ∧ i < j. L' i * (grd P (i + 1) - grd P i))
  proof (rule sum-mono)
    fix k
    assume k ∈ {i. L i ≠ 0 ∧ i < j}
    hence k∈ I using assms by auto
    thus  L k * (grd P (k + 1) - grd P k) ≤ L' k * (grd P (k + 1) - grd P k)
      using le
      by (simp add: assms(1) clmm-dsc-grd-mono mult-right-mono)
  qed
  thus ?thesis by simp
qed
also have ... = L' j * (sqp - grd P' j) +
    (∑ i | L' i ≠ 0 ∧ i < j. L' i * (grd P (i + 1) - grd P i))
proof -
  have ziff: ∀ i∈ I. (L i = 0 ⟷ L' i = 0) using assms le pos
    by (metis L'-def clmm-dsc-gross-liq-zero-iff fee-diff-onD(2))
  have {i. L i ≠ 0 ∧ i < j} = {i. L' i ≠ 0 ∧ i < j}
  proof
    show {i. L i ≠ 0 ∧ i < j} ⊆ {i. L' i ≠ 0 ∧ i < j}
      using ziff assms(3) by auto
    show {i. L' i ≠ 0 ∧ i < j} ⊆ {i. L i ≠ 0 ∧ i < j}
    proof
      fix i
      assume i ∈ {i. L' i ≠ 0 ∧ i < j}
      hence L' i ≠ 0 and i < j by auto
      hence L i ≠ 0
        using L'-def same-nz-liq-onD(2) assms clmm-dsc-gross-liq-zero-iff
        by (smt (verit, ccfv-threshold) mem-Collect-eq)
      thus i ∈ {i. L i ≠ 0 ∧ i < j} using ‹i < j› by auto
    qed
  qed
```

```
    moreover have grd P j = grd P' j using assms by auto
    ultimately show ?thesis by simp
  qed
  also have ... = L' j * (sqp - grd P' j) +
      (∑ i | L' i ≠ 0 ∧ i < j. L' i * (grd P' (i + 1) - grd P' i))
  proof -
    have (∑ i | L' i ≠ 0 ∧ i < j. L' i * (grd P (i + 1) - grd P i)) =
        (∑ i | L' i ≠ 0 ∧ i < j. L' i * (grd P' (i + 1) - grd P' i))
    proof (rule sum.cong)
      fix x
      assume x ∈ {i. L' i ≠ 0 ∧ i < j}
      hence x ∈ {k. k ≤ j} by auto
      hence grd P x = grd P' x using assms by force
      have x+1 ∈ {k. k ≤ j} using ‹x ∈ {i. L' i ≠ 0 ∧ i < j}› by auto
      hence grd P (x + 1) = grd P' (x + 1) using assms by force
      thus L' x * (grd P (x + 1) - grd P x) = L' x * (grd P' (x + 1) - grd P' x)
        using ‹grd P x = grd P' x› by simp
    qed simp
    thus ?thesis by simp
  qed
  also have ... = quote-gross P' sqp
    by (rule clmm-quote-gross-sum[symmetric], (auto simp add: assms L'-def))
  finally show ?thesis .
qed

lemma fee-diff-le-imp-quote-gross-mono:
  assumes clmm-dsc P
  and clmm-dsc P'
  and {k. L k ≠ 0 ∧ k ≤ j} ⊆ I
  and fee-diff-on P P' I
  and same-nz-liq-on P P' {k. k ≤ j}
  and 0 < sqp
  and j = lower-tick P sqp
  and L = gross-fct (lq P) (fee P)
  and ⋀i. i ∈ I ⟹ fee P i ≤ fee P' i
  and lower-tick P sqp = lower-tick P' sqp
  and sqp ≤ sqp'
shows quote-gross P sqp ≤ quote-gross P' sqp'
proof -
  have quote-gross P sqp ≤ quote-gross P' sqp
    using assms fee-diff-le-imp-quote-gross by simp
  also have ... ≤ quote-gross P' sqp'
    using clmm-quote-gross-mono[of P'] monoD assms(2,11) by simp
  finally show ?thesis .
qed

lemma fee-diff-quote-diff-expand:
  assumes clmm-dsc P
  and clmm-dsc P'
```

and $L = \textit{gross-fct}\ (\textit{lq}\ P)\ (\textit{fee}\ P)$
and $j = \textit{lower-tick}\ P\ \textit{sqp}$
and $k = \textit{lower-tick}\ P\ \textit{sqp}'$
and $0 < \textit{sqp}$
and $\textit{sqp} \leq \textit{sqp}'$
and $j < k$
and $\{m.\ L\ m \neq 0 \wedge j \leq m \wedge m \leq k\} \subseteq I$
and $\textit{fee-diff-on}\ P\ P'\ I$
and $\textit{same-nz-liq-on}\ P\ P'\ \{m.\ j \leq m \wedge m \leq k{+}1\}$
and $\bigwedge i.\ i \in I \Longrightarrow \textit{fee}\ P\ i \leq \textit{fee}\ P'\ i$
and $\textit{lower-tick}\ P\ \textit{sqp} = \textit{lower-tick}\ P'\ \textit{sqp}$
and $\textit{lower-tick}\ P\ \textit{sqp}' = \textit{lower-tick}\ P'\ \textit{sqp}'$
shows $\textit{quote-gross}\ P\ \textit{sqp}' - \textit{quote-gross}\ P\ \textit{sqp} \leq \textit{quote-gross}\ P'\ \textit{sqp}' - \textit{quote-gross}\ P'\ \textit{sqp}$
proof −
define L' **where** $L' = \textit{gross-fct}\ (\textit{lq}\ P')\ (\textit{fee}\ P')$
have *pos*: $\forall\, i{\in}\ I.\ 0 \leq L\ i$ **using** *assms gross-liq-ge* **by** *simp*
have *le*: $\forall\, i{\in}\ I.\ L\ i \leq L'\ i$
proof
fix i
assume $i \in I$
hence $\textit{lq}\ P\ i = \textit{lq}\ P'\ i$ **using** $\textit{assms fee-diff-onD}(2)$ **by** *simp*
hence $L\ i = \textit{gross-fct}\ (\textit{lq}\ P')\ (\textit{fee}\ P)\ i$
using *assms gross-fct-cong* **by** *blast*
also have $\ldots \leq L'\ i$ **unfolding** $L'\textit{-def}$
proof (*rule gross-fct-le*)
show $0 \leq \textit{lq}\ P'\ i$ **by** (*simp add*: $\textit{assms}(2)\ \textit{clmm-dsc-liq}(2)$)
show $\textit{fee}\ P\ i \leq \textit{fee}\ P'\ i$ **using** ‹$i \in I$› *assms* **by** *simp*
show $\textit{fee}\ P'\ i < 1$ **by** (*simp add*: $\textit{assms}(2)\ \textit{clmm-dsc-fees}$)
qed
finally show $L\ i \leq L'\ i$ **.**
qed
have $\textit{quote-gross}\ P\ \textit{sqp}' - \textit{quote-gross}\ P\ \textit{sqp} = L\ k * (\textit{sqp}' - \textit{grd}\ P\ k) + \textit{sum}\ (\lambda\ i.\ L\ i * (\textit{grd}\ P\ (i{+}1) - \textit{grd}\ P\ i))\ \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} + L\ j * (\textit{grd}\ P\ (j{+}1) - \textit{sqp})$
using *assms clmm-quote-gross-diff-eq* **by** *simp*
also have $\ldots \leq L'\ k * (\textit{sqp}' - \textit{grd}\ P\ k) + \textit{sum}\ (\lambda\ i.\ L\ i * (\textit{grd}\ P\ (i{+}1) - \textit{grd}\ P\ i))\ \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} + L\ j * (\textit{grd}\ P\ (j{+}1) - \textit{sqp})$
proof (*cases* $k{\in}\ I$)
case *True*
then show *?thesis* **using** *lower-tick-lbound le pos*
by (*smt* (*verit*, *best*) $\textit{assms}(1)\ \textit{assms}(5)\ \textit{assms}(6)\ \textit{assms}(7)$ *mult-right-mono*)
next
case *False*
hence $L\ k = 0$ **using** *assms* **by** *auto*
then show *?thesis*
using $L'\textit{-def lower-tick-geq gross-liq-ge}\ \textit{assms}(1,2,5{-}7)$ **by** *auto*
qed

also have ... $\leq L'\ k * (sqp' - grd\ P\ k) +$
$sum\ (\lambda\ i.\ L\ i * (grd\ P\ (i{+}1) - grd\ P\ i))\ \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} +$
$L'\ j * (grd\ P\ (j{+}1) - sqp)$
proof (*cases* $j \in I$)
case *True*
then show *?thesis* **using** *lower-tick-lbound le pos*
by (*simp add*: *assms*(1) *assms*(4) *lower-tick-ubound*)
next
case *False*
hence $L\ j = 0$ **using** *assms* **by** *auto*
then show *?thesis*
using *L'-def lower-tick-geq gross-liq-ge*
by (*smt* (*verit*, *ccfv-SIG*) *assms*(1,2,4) *lower-tick-ubound mult-right-mono*)
qed
also have ... $\leq L'\ k * (sqp' - grd\ P\ k) +$
$sum\ (\lambda\ i.\ L'\ i * (grd\ P\ (i{+}1) - grd\ P\ i))\ \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} +$
$L'\ j * (grd\ P\ (j{+}1) - sqp)$
proof −
have $sum\ (\lambda\ i.\ L\ i * (grd\ P\ (i{+}1) - grd\ P\ i))\ \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} \leq$
$sum\ (\lambda\ i.\ L'\ i * (grd\ P\ (i{+}1) - grd\ P\ i))\ \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\}$
proof (*rule sum-mono*)
fix i
assume $i \in \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\}$
hence $i \in I$ **using** *assms* **by** *auto*
thus $L\ i * (grd\ P\ (i + 1) - grd\ P\ i) \leq L'\ i * (grd\ P\ (i + 1) - grd\ P\ i)$
using *le*
by (*simp add*: *assms*(1) *clmm-dsc-grd-mono mult-right-mono*)
qed
thus *?thesis* **by** *simp*
qed
also have ... $= L'\ k * (sqp' - grd\ P'\ k) +$
$sum\ (\lambda\ i.\ L'\ i * (grd\ P\ (i{+}1) - grd\ P\ i))\ \{i.\ L'\ i \neq 0 \wedge j < i \wedge i < k\} +$
$L'\ j * (grd\ P'\ (j{+}1) - sqp)$
proof −
have *ziff*: $\forall\ i \in I.\ (L\ i = 0 \longleftrightarrow L'\ i = 0)$ **using** *assms le pos*
by (*metis L'-def clmm-dsc-gross-liq-zero-iff fee-diff-onD*(2))
have $\{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} = \{i.\ L'\ i \neq 0 \wedge j < i \wedge i < k\}$
proof
show $\{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\} \subseteq \{i.\ L'\ i \neq 0 \wedge j < i \wedge i < k\}$
using *ziff assms*(9) **by** *auto*
show $\{i.\ L'\ i \neq 0 \wedge j < i \wedge i < k\} \subseteq \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\}$
proof
fix i
assume $i \in \{i.\ L'\ i \neq 0 \wedge j < i \wedge i < k\}$
hence $L'\ i \neq 0$ **and** $i < k$ **and** $j < i$ **by** *auto*
hence $L\ i \neq 0$
using *L'-def same-nz-liq-onD*(2) *assms clmm-dsc-gross-liq-zero-iff*
by (*smt* (*verit*, *ccfv-threshold*) *mem-Collect-eq*)
thus $i \in \{i.\ L\ i \neq 0 \wedge j < i \wedge i < k\}$

```
        using ‹i < k› ‹j < i› by simp
      qed
    qed
    moreover have grd P k = grd P' k
      using assms(11) assms(8) same-nz-liq-onD(1) by auto
    moreover have grd P (j+1) = grd P' (j+1)
      using add1-zle-eq assms(11) assms(8) same-nz-liq-onD(1) by auto
    ultimately show ?thesis by simp
  qed
  also have ... = L' k * (sqp' - grd P' k) +
      sum (λ i. L' i * (grd P' (i+1) - grd P' i)) {i. L' i ≠ 0 ∧ j <i ∧ i < k} +
      L' j * (grd P' (j+1) - sqp)
  proof -
    have sum (λ i. L' i * (grd P (i+1) - grd P i)) {i. L' i ≠ 0 ∧ j <i ∧ i < k}
=
        sum (λ i. L' i * (grd P' (i+1) - grd P' i)) {i. L' i ≠ 0 ∧ j <i ∧ i < k}
    proof (rule sum.cong)
      fix x
      assume x ∈ {i. L' i ≠ 0 ∧ j < i ∧ i < k}
      hence x ∈ {i. j ≤ i ∧ i ≤ k} by auto
      hence grd P x = grd P' x using assms by force
      have x+1 ∈ {i. j ≤ i ∧ i ≤ k}
        using ‹x ∈ {i. L' i ≠ 0 ∧ j < i ∧ i < k}› by auto
      hence grd P (x + 1) = grd P' (x + 1) using assms by force
      thus L' x * (grd P (x + 1) - grd P x) = L' x * (grd P' (x + 1) - grd P' x)
        using ‹grd P x = grd P' x› by simp
    qed simp
    thus ?thesis by simp
  qed
  also have ... = quote-gross P' sqp' - quote-gross P' sqp
  proof (rule clmm-quote-gross-diff-eq[symmetric])
    show j < k using assms by simp
  qed (simp add: assms L'-def)+
  finally show ?thesis .
qed

lemma fee-diff-quote-diff-expand':
  assumes clmm-dsc P
  and clmm-dsc P'
  and L = gross-fct (lq P) (fee P)
  and j = lower-tick P sqp
  and j = lower-tick P sqp'
  and 0 < sqp
  and sqp ≤ sqp'
  and L j ≠ 0 ⟶ j∈ I
  and fee-diff-on P P' I
  and ⋀i. i ∈ I ⟹ fee P i ≤ fee P' i
  and lower-tick P sqp = lower-tick P' sqp
  and lower-tick P sqp' = lower-tick P' sqp'
```

shows *quote-gross P sqp′ − quote-gross P sqp ≤ quote-gross P′ sqp′ − quote-gross P′ sqp*
proof −
define L' **where** $L' = \textit{gross-fct}\ (\textit{lq}\ P')\ (\textit{fee}\ P')$
have *le*: $\forall\, i \in I.\ L\ i \leq L'\ i$
proof
fix i
assume $i \in I$
hence $\textit{lq}\ P\ i = \textit{lq}\ P'\ i$ **using** *assms fee-diff-onD*(2) **by** *simp*
hence $L\ i = \textit{gross-fct}\ (\textit{lq}\ P')\ (\textit{fee}\ P)\ i$
using *assms gross-fct-cong* **by** *blast*
also have $\ldots \leq L'\ i$ **unfolding** *L′-def*
proof (*rule gross-fct-le*)
show $0 \leq \textit{lq}\ P'\ i$ **by** (*simp add*: *assms*(2) *clmm-dsc-liq*(2))
show $\textit{fee}\ P\ i \leq \textit{fee}\ P'\ i$ **using** ‹$i \in I$› *assms* **by** *simp*
show $\textit{fee}\ P'\ i < 1$ **by** (*simp add*: *assms*(2) *clmm-dsc-fees*)
qed
finally show $L\ i \leq L'\ i$ **.**
qed
have $\textit{quote-gross}\ P\ \textit{sqp}' - \textit{quote-gross}\ P\ \textit{sqp} = L\ j * (\textit{sqp}' - \textit{sqp})$
using *assms clmm-quote-gross-diff-eq′* **by** *simp*
also have $\ldots \leq L'\ j * (\textit{sqp}' - \textit{sqp})$ **using** *lower-tick-lbound le*
by (*metis L′-def assms*(2,7,8) *diff-ge-0-iff-ge gross-liq-ge*
mult.commute ordered-comm-semiring-class.comm-mult-left-mono)
also have $\ldots = \textit{quote-gross}\ P'\ \textit{sqp}' - \textit{quote-gross}\ P'\ \textit{sqp}$
proof (*rule clmm-quote-gross-diff-eq′*[*symmetric*])
show *clmm-dsc P′* **using** *assms* **by** *simp*
show $L' = \textit{gross-fct}\ (\textit{lq}\ P')\ (\textit{fee}\ P')$ **using** *L′-def* **by** *simp*
show $j = \textit{lower-tick}\ P'\ \textit{sqp}$ **using** *assms* **by** *simp*
show $j = \textit{lower-tick}\ P'\ \textit{sqp}'$ **using** *assms* **by** *simp*
show $0 < \textit{sqp}$ **using** *assms* **by** *simp*
show $\textit{sqp} \leq \textit{sqp}'$ **using** *assms* **by** *simp*
qed
finally show *?thesis* **.**
qed

lemma *fee-diff-quote-diff-le*:
assumes *clmm-dsc P*
and *clmm-dsc P′*
and $L = \textit{gross-fct}\ (\textit{lq}\ P)\ (\textit{fee}\ P)$
and $j = \textit{lower-tick}\ P\ \textit{sqp}$
and $k = \textit{lower-tick}\ P\ \textit{sqp}'$
and $0 < \textit{sqp}$
and $\textit{sqp} \leq \textit{sqp}'$
and $\{m.\ L\ m \neq 0 \wedge j \leq m \wedge m \leq k\} \subseteq I$
and *fee-diff-on P P′ I*
and *same-nz-liq-on P P′* $\{m.\ j \leq m \wedge m \leq k{+}1\}$
and $\bigwedge i.\ i \in I \Longrightarrow \textit{fee}\ P\ i \leq \textit{fee}\ P'\ i$
and $\textit{lower-tick}\ P\ \textit{sqp} = \textit{lower-tick}\ P'\ \textit{sqp}$

and *lower-tick P sqp′* = *lower-tick P′ sqp′*
shows *quote-gross P sqp′* − *quote-gross P sqp* ≤ *quote-gross P′ sqp′* − *quote-gross P′ sqp*
proof (*cases j* = *k*)
case *True*
have *L j*≠ *0* ⟶ *j*∈ *I* **using** *True assms*(*8*) **by** *blast*
then show *?thesis* **using** *assms True fee-diff-quote-diff-expand′* **by** *simp*
next
case *False*
hence *j* < *k* **using** *lower-tick-mono assms*(*1*,*4*−*7*) **by** *fastforce*
then show *?thesis* **using** *fee-diff-quote-diff-expand assms* **by** *simp*
qed

lemma *same-nz-liq-on-nz-support*:
assumes *i* ∈ *I*
and *lq P i* ≠ *0*
and *same-nz-liq-on P P′ I*
shows *nz-support* (*lq P′*) ≠ {}
proof −
have *lq P′ i*≠ *0* **using** *assms* **by** *blast*
thus *?thesis* **unfolding** *nz-support-def* **by** *auto*
qed

lemma *same-nz-liq-on-idx-max*:
assumes *finite-liq P′*
and *nz-support* (*lq P*) ≠ {}
and *I* = {*idx-min* (*lq P*) .. *idx-max* (*lq P*) + *1*}
and *same-nz-liq-on P P′ I*
shows *idx-max* (*lq P*) ≤ *idx-max* (*lq P′*)
proof −
define *i* **where** *i* = *idx-max* (*lq P*)
have *i*∈ *I* **using** *i-def assms*
by (*simp add*: *fin-nz-sup idx-min-max-finite*)
have *lq P i* ≠ *0* **using** *i-def* **by** (*simp add*: *assms*(*2*) *idx-max-mem nz-supportD*)
hence *lq P′ i* ≠ *0* **using** *same-nz-liq-onD*(*2*) ‹*i*∈ *I*› *assms* **by** *simp*
thus *i* ≤ *idx-max* (*lq P′*)
using *idx-max-finite-ge assms*(*1*) *finite-liq-def* **by** *simp*
qed

lemma *same-nz-liq-on-grd-max*:
assumes *finite-liq P′*
and *mono* (*grd P′*)
and *nz-support* (*lq P*) ≠ {}
and *I* = {*idx-min* (*lq P*) .. *idx-max* (*lq P*) + *1*}
and *same-nz-liq-on P P′ I*
shows *grd-max P* ≤ *grd-max P′*
proof −
have *grd-max P* = *grd P* (*idx-max* (*lq P*) + *1*)
using *grd-max-def idx-max-img-def* **by** *simp*

```
  also have ... = grd P' (idx-max (lq P) + 1)
  proof −
    have idx-max (lq P) + 1 ∈ I
      by (simp add: add.commute add-increasing assms(3,4) fin-nz-sup
          idx-min-max-finite)
    thus ?thesis using same-nz-liq-onD(1) assms by auto
  qed
  also have ... ≤ grd P' (idx-max (lq P') + 1)
  proof −
    have idx-max (lq P) ≤ idx-max (lq P')
      using assms same-nz-liq-on-idx-max by simp
    thus ?thesis by (simp add: assms(2) monoD)
  qed
  finally show ?thesis unfolding grd-max-def idx-max-img-def by simp
qed

lemma same-nz-liq-on-lower-tick:
  assumes clmm-dsc P
  and clmm-dsc P'
  and same-nz-liq-on P P' {i. i ≤ j+1}
  and 0 < sqp
  and lower-tick P sqp ≤ j
shows lower-tick P' sqp = lower-tick P sqp
proof (rule lower-tick-charact)
  define i where i = lower-tick P sqp
  show clmm-dsc P' using assms by simp
  have grd P' i = grd P i
    using assms i-def by (simp add: same-nz-liq-onD(1))
  also have ... ≤ sqp
    by (simp add: assms(1,4) lower-tick-mem i-def)
  finally show grd P' i ≤ sqp .
  have sqp < grd P (i+1)
    by (simp add: assms(1) i-def lower-tick-ubound)
  also have ... = grd P' (i+1)
    using assms i-def by (simp add: same-nz-liq-onD(1))
  finally show sqp < grd P' (i+1) .
qed

lemma same-nz-liq-on-lower-tick':
  assumes clmm-dsc P'
  and same-nz-liq-on P P' {i. i ≤ j}
  and grd P j = sqp
shows lower-tick P' sqp = j
  using assms lower-tick-eq same-nz-liq-onD(1) by auto

lemma fee-diff-le-grd-max:
  assumes clmm-dsc P
  and clmm-dsc P'
  and nz-support (lq P) ≠ {}
```

```
  and {idx-min (lq P) .. idx-max (lq P) + 1} ⊆ I
  and fee-diff-on P P' I
  and same-nz-liq-on P P' {k. k ≤ idx-max (lq P) + 1}
  and ⋀i. i ∈ I ⟹ fee P i ≤ fee P' i
shows quote-gross P (grd-max P) ≤ quote-gross P' (grd-max P)
proof −
  define j where j = lower-tick P (grd-max P)
  have j = idx-max (lq P) +1
    by (simp add: assms(1) j-def lower-tick-grd-max)
  hence grd P j = grd-max P unfolding grd-max-def idx-max-img-def by simp
  define L where L = gross-fct (lq P) (fee P)
  show quote-gross P (grd-max P) ≤ quote-gross P' (grd-max P)
  proof (rule fee-diff-le-imp-quote-gross)
    show j = lower-tick P (grd-max P) using j-def by simp
    show L = gross-fct (lq P) (fee P) using L-def by simp
    show 0 < grd-max P by (simp add: assms(1) assms(3) grd-max-gt)
    have {k. L k ≠ 0 ∧ k ≤ j} ⊆ {idx-min (lq P) .. idx-max (lq P)+1}
    proof
      fix k
      assume k∈ {k. L k ≠ 0 ∧ k ≤ j}
      hence L k ≠ 0 and k ≤ j by auto
      hence k∈ {idx-min (lq P) .. idx-max (lq P)}
      using non-zero-liq-interv L-def assms(1) clmm-dsc-gross-liq-zero-iff fin-nz-sup

        by blast
      thus k∈ {idx-min (lq P) .. idx-max (lq P)+1} by auto
    qed
    thus {k. L k ≠ 0 ∧ k ≤ j} ⊆ I using assms by simp
    show fee-diff-on P P' I using assms by simp
    show same-nz-liq-on P P' {k. k ≤ j}
      using assms ‹j = idx-max (lq P) +1› by simp
    show ⋀i. i ∈ I ⟹ fee P i ≤ fee P' i using assms by simp
    show lower-tick P (grd-max P) = lower-tick P' (grd-max P)
     using ‹grd P j = grd-max P› ‹same-nz-liq-on P P' {k. k ≤ j}› assms(2) j-def

       same-nz-liq-on-lower-tick'
      by auto
  qed (simp add: assms)+
qed

lemma fee-diff-le-grd-max':
  assumes clmm-dsc P
  and clmm-dsc P'
  and nz-support (lq P) ≠ {}
  and {idx-min (lq P) .. idx-max (lq P) + 1} ⊆ I
  and fee-diff-on P P' I
  and same-nz-liq-on P P' {k. k ≤ idx-max (lq P) + 1}
  and ⋀i. i ∈ I ⟹ fee P i ≤ fee P' i
shows quote-gross P (grd-max P) ≤ quote-gross P' (grd-max P')
```

proof −
have *quote-gross P* (*grd-max P*) ≤ *quote-gross P′* (*grd-max P*)
using *assms fee-diff-le-grd-max* **by** *simp*
also have ... ≤ *quote-gross P′* (*grd-max P′*)
proof (*rule clmm-quote-gross-mono*[*THEN monoD*])
show *clmm-dsc P′* **using** *assms* **by** *simp*
show *grd-max P* ≤ *grd-max P′* **using** *same-nz-liq-on-grd-max*
by (*meson assms*(2−5) *clmm-dsc-grd-mono clmm-dsc-liq*(1) *fee-diff-on-mono*

fee-diff-on-nz-liq mono-onI)
qed
finally show *?thesis* **.**
qed

lemma *fee-diff-le-imp-quote-reach*:
assumes *clmm-dsc P*
and *clmm-dsc P′*
and *nz-support* (*lq P*) ≠ {}
and {*idx-min* (*lq P*) .. *idx-max* (*lq P*) + 1} ⊆ *I*
and *fee-diff-on P P′ I*
and *same-nz-liq-on P P′* {*i*. *i* ≤ *idx-max* (*lq P*) + 1}
and ⋀*i*. *i* ∈ *I* ⟹ *fee P i* ≤ *fee P′ i*
and 0 < *y*
and *y* ≤ *quote-gross P* (*grd-max P*)
shows *quote-reach P′ y* ≤ *quote-reach P y*
proof −
define *sqp′* **where** *sqp′* = *quote-reach P′ y*
define *sqp* **where** *sqp* = *quote-reach P y*
define *L* **where** *L* = *gross-fct* (*lq P*) (*fee P*)
have *nz-support* (*lq P′*) ≠ {}
proof (*rule same-nz-liq-on-nz-support*)
have *idx-min* (*lq P*) ∈ {*idx-min* (*lq P*) .. *idx-max* (*lq P*) + 1}
using *assms*(3) *fin-nz-sup idx-min-max-finite* **by** *fastforce*
thus *idx-min* (*lq P*) ∈ *I* **using** *assms* **by** *auto*
show *lq P* (*idx-min* (*lq P*)) ≠ 0
by (*simp add*: *assms*(3) *idx-min-mem nz-supportD*)
show *same-nz-liq-on P P′ I* **using** *assms fee-diff-on-nz-liq* **by** *simp*
qed
have *grd-max P* ≤ *grd-max P′*
by (*meson assms*(2−5) *clmm-dsc-grid*(1) *clmm-dsc-liq*(1) *fee-diff-on-mono*
fee-diff-on-nz-liq same-nz-liq-on-grd-max strict-mono-mono)
have 0 < *grd-max P*
by (*meson assms*(1) *assms*(3) *liq-grd-min liq-grd-min-max*
dual-order.strict-trans)
have *quote-gross P′ sqp′* = *y* **unfolding** *sqp′-def*
proof (*rule clmm-quote-gross-reach-eq*)
show *clmm-dsc P′* **using** *assms* **by** *simp*
show 0 ≤ *y* **using** *assms* **by** *simp*
show *nz-support* (*lq P′*) ≠ {} **using** ‹*nz-support* (*lq P′*) ≠ {}› **.**

```
  have y ≤ quote-gross P (grd-max P) using assms by simp
  also have ... ≤ quote-gross P' (grd-max P')
  proof (rule fee-diff-le-imp-quote-gross-mono[OF assms(1−2)])
    define j where j = lower-tick P (grd-max P)
    hence j = idx-max (lq P) + 1
      by (simp add: assms(1) idx-max-img-def lower-tick-eq grd-max-def)
    show same-nz-liq-on P P' {k. k ≤ j}
      using assms ‹j = idx-max (lq P) + 1› by simp
    show {k. L k ≠ 0 ∧ k ≤ j} ⊆ I
    proof
      fix x
      assume x ∈ {k. L k ≠ 0 ∧ k ≤ j}
      hence L x ≠ 0 and x ≤ j by auto
      hence idx-min (lq P) ≤ x
        by (metis L-def assms(1) clmm-dsc-gross-liq-zero-iff
            idx-min-lt-liq leI)
      moreover have x ≤ idx-max (lq P)
        using L-def ‹L x ≠ 0› fin-nz-sup gross-fct-zero-if idx-max-finite-ge
        by blast
      ultimately have x ∈ {idx-min (lq P) .. idx-max (lq P) + 1} by auto
      thus x∈ I using assms by auto
    qed
    show fee-diff-on P P' I using assms by simp
    show  0 < grd-max P using ‹0 < grd-max P› .
    show grd-max P ≤ grd-max P' using ‹grd-max P ≤ grd-max P'› .
    show j = lower-tick P' (grd-max P) unfolding j-def
    proof (rule same-nz-liq-on-lower-tick'[symmetric])
      show grd P (lower-tick P (grd-max P)) = grd-max P
     using ‹j = idx-max (lq P) + 1› unfolding j-def grd-max-def idx-max-img-def

      by simp
      show same-nz-liq-on P P' {i. i ≤lower-tick P (grd-max P)}
        using ‹same-nz-liq-on P P' {k. k ≤ j}› j-def by auto
    qed (simp add: assms)
    show L = gross-fct (lq P) (fee P) unfolding L-def by simp
    show ⋀i. i ∈ I ⟹ fee P i ≤ fee P' i using assms by simp
  qed simp
  finally show y ≤ quote-gross P' (grd-max P') .
qed
also have ... = quote-gross P sqp
  using assms clmm-quote-gross-reach-eq sqp-def by simp
also have ... ≤ quote-gross P' sqp
proof (rule fee-diff-le-imp-quote-gross)
  define k where k = lower-tick P sqp
  thus k = lower-tick P sqp by simp
  show 0 < sqp using clmm-quote-reach-pos assms sqp-def by simp
  show fee-diff-on P P' I using assms by simp
  show L = gross-fct (lq P) (fee P) using L-def by simp
  show ⋀i. i ∈ I ⟹ fee P i ≤ fee P' i using assms by simp
```

show $\{l.\ L\ l \neq 0 \wedge l \leq k\} \subseteq I$
proof
  fix x
  assume $x \in \{l.\ L\ l \neq 0 \wedge l \leq k\}$
  hence $L\ x \neq 0$ **and** $x \leq k$ **by** *auto*
  hence *idx-min* $(lq\ P) \leq x$
    by (*metis L-def assms*(*1*) *clmm-dsc-gross-liq-zero-iff idx-min-lt-liq leI*)
  moreover have $x \leq$ *idx-max* $(lq\ P)$ **using** ‹$L\ x \neq 0$›
    by (*metis L-def assms*(*1*) *idx-max-gt-liq gross-fct-zero-if leI*)
  ultimately have $x \in \{$*idx-min* $(lq\ P)$ .. *idx-max* $(lq\ P) + 1\}$ **by** *auto*
  thus $x \in I$ **using** *assms* **by** *auto*
qed
have *sqp* $\leq$ *grd-max P*
  by (*metis* ‹*y* = *quote-gross P sqp*› *assms*(*1*,*3*) *quote-gross-grd-max-ge grd-max-quote-reach order-less-irrefl sqp-def verit-comp-simplify1*(*3*))
moreover have *lower-tick P* (*grd-max P*) = *idx-max* (*lq P*) + *1*
  by (*simp add*: *assms*(*1*) *lower-tick-grd-max*)
ultimately have $k \leq$ *idx-max* (*lq P*) *+1* **using** *k-def*
  by (*metis* ‹*0* < *sqp*› *assms*(*1*) *lower-tick-mono*)
show *same-nz-liq-on P P'* $\{l.\ l \leq k\}$
proof (*rule same-nz-liq-on-mono*)
  show *same-nz-liq-on P P'* $\{i.\ i \leq$ *idx-max* $(lq\ P) + 1\}$
    using *assms* **by** *simp*
  show $\{l.\ l \leq k\} \subseteq \{i.\ i \leq$ *idx-max* $(lq\ P) + 1\}$
    using ‹$k \leq$ *idx-max* $(lq\ P)$ *+1*› **by** *auto*
qed
show $k =$ *lower-tick P' sqp*
proof (*cases sqp* = *grd-max P*)
  case *True*
  hence $k =$ *idx-max* $(lq\ P) + 1$ **using** *k-def*
    by (*simp add*: ‹*lower-tick P* (*grd-max P*) = *idx-max* (*lq P*) + *1*›)
  show *?thesis*
  proof (*rule same-nz-liq-on-lower-tick'*[*symmetric*])
    show *clmm-dsc P'* **using** *assms* **by** *simp*
    show *same-nz-liq-on P P'* $\{i.\ i \leq k\}$
      using *assms* ‹$k =$ *idx-max* $(lq\ P) + 1$› **by** *simp*
    show *grd P k* = *sqp*
      using *k-def True* ‹$k =$ *idx-max* $(lq\ P) + 1$›
      unfolding *grd-max-def idx-max-img-def* **by** *simp*
  qed
next
  case *False*
  hence *sqp* < *grd-max P* **using** ‹*sqp* ≤ *grd-max P*› **by** *simp*
  hence $k <$ *lower-tick P* (*grd-max P*)
    using ‹*0* < *sqp*› ‹*lower-tick P* (*grd-max P*) = *idx-max* (*lq P*) + *1*› *assms*(*1*,*3*) *sqp-lt-grd-max-imp-idx k-def*
    by *auto*

```
      hence k ≤ idx-max (lq P)
        by (simp add: ‹lower-tick P (grd-max P) = idx-max (lq P) + 1›)
      show ?thesis unfolding k-def
      proof (rule same-nz-liq-on-lower-tick[symmetric])
        show lower-tick P sqp ≤ idx-max (lq P)
          using ‹k ≤ idx-max (lq P)› k-def by simp
        show 0 < sqp using ‹0 < sqp› .
         show same-nz-liq-on P P' {i. i ≤ idx-max (lq P) + 1} using assms by
simp
      qed (simp add: assms)+
    qed
  qed (simp add: assms)+
  finally have quote-gross P' sqp' ≤ quote-gross P' sqp .
  define sqp2 where sqp2 = quote-reach P' (quote-gross P' sqp)
  have sqp' ≤ sqp2
  proof (rule quote-reach-mono)
    show clmm-dsc P' using assms by simp
    show nz-support (lq P') ≠ {} using ‹nz-support (lq P') ≠ {}› .
    show 0 ≤ y using assms by simp
    show y ≤ quote-gross P' sqp
      using ‹y = quote-gross P sqp› ‹quote-gross P sqp ≤ quote-gross P' sqp› by
simp
    show sqp' = quote-reach P' y using sqp'-def by simp
    show sqp2 = quote-reach P' (quote-gross P' sqp) using sqp2-def by simp
    show quote-gross P' sqp ≤ quote-gross P' (grd-max P')
    proof −
      have sqp ≤ grd-max P
        using sqp-def quote-reach-leq-grd-max
        by (simp add: ‹0 ≤ y› assms(1,3,9))
      also have ... ≤ grd-max P' using ‹grd-max P ≤ grd-max P'› .
      finally have sqp ≤ grd-max P' .
      thus ?thesis
        by (simp add: ‹nz-support (lq P') ≠ {}› assms(2)
            quote-gross-grd-max-max)
    qed
  qed
  also have ... ≤ sqp using clmm-quote-reach-le sqp2-def
    using ‹nz-support (lq P') ≠ {}› ‹quote-gross P' sqp' = y›
      ‹quote-gross P' sqp' ≤ quote-gross P' sqp› assms(2,8)
    by auto
  finally show ?thesis unfolding sqp'-def sqp-def .
qed

lemma same-nz-liq-on-if-simil:
  assumes grd P = grd P'
  and nz-support (lq P) = nz-support (lq P')
shows same-nz-liq-on P P' I
proof
  show id-grid-on P P' I using id-grid-onI assms by simp
```

```
  show ⋀i. i ∈ I ⟹ (lq P i = 0) = (lq P' i = 0)
  proof −
    fix i
    assume i ∈ I
    have (i∈ nz-support (lq P)) ⟷ (i ∈ nz-support (lq P')) using assms by simp
    thus (lq P i = 0) = (lq P' i = 0) using nz-support-def by fastforce
  qed
qed

lemma fee-diff-simil-base-net:
  assumes clmm-dsc P
  and clmm-dsc P'
  and nz-support (lq P) ≠ {}
  and {idx-min (lq P) .. idx-max (lq P) + 1} ⊆ I
  and fee-diff-on P P' I
  and nz-support (lq P) = nz-support (lq P')
  and grd P = grd P'
  and grd-min P ≤ sqp
  and sqp ≤ grd-max P
shows base-net P sqp = base-net P' sqp
proof (rule fee-diff-same-base-net)
  define j where j = lower-tick P sqp
  define L where L = lq P
  show j = lower-tick P sqp using j-def by simp
  show 0 < sqp using grd-min-pos
    using assms(1) assms(3) assms(8) liq-grd-min order-less-le-trans by blast
  show L = lq P using L-def by simp
  show same-nz-liq-on P P' {k. lower-tick P sqp ≤ k}
    using assms same-nz-liq-on-if-simil by simp
  show fee-diff-on P P' {k. lq P k ≠ 0 ∧ lower-tick P sqp ≤ k}
  proof (rule fee-diff-on-mono)
    show fee-diff-on P P' I using assms by simp
    show {k. lq P k ≠ 0 ∧ lower-tick P sqp ≤ k} ⊆ I
    proof
      fix k
      assume k ∈ {k. lq P k ≠ 0 ∧ lower-tick P sqp ≤ k}
      hence L k ≠ 0 and lower-tick P sqp ≤ k using L-def by auto
      hence idx-min L ≤ k using L-def
        by (metis assms(1) idx-min-lt-liq linorder-le-cases
            order-le-imp-less-or-eq)
      moreover have k ≤ idx-max L
        using L-def ‹L k ≠ 0› fin-nz-sup idx-max-finite-ge by auto
      ultimately have k ∈ {idx-min (lq P) .. idx-max (lq P) + 1}
        using L-def by auto
      thus k∈ I using assms by auto
    qed
  qed
  show lower-tick P sqp = lower-tick P' sqp
    using assms(7) grd-lower-tick-cong by blast
```

qed (*simp add*: *assms*)+

lemma *fee-diff-le-price-cmp*:
  assumes *clmm-dsc P*
  and *clmm-dsc P′*
  and *nz-support* (*lq P*) ≠ {}
  and {*idx-min* (*lq P*) .. *idx-max* (*lq P*) + *1*} ⊆ *I*
  and *fee-diff-on P P′ I*
  and *nz-support* (*lq P*) = *nz-support* (*lq P′*)
  and *grd P* = *grd P′*
  and ⋀*i*. *i* ∈ *I* ⟹ *fee P i* ≤ *fee P′ i*
  and *0* < *y*
  and *y* + *quote-gross P sqp* ≤ *quote-gross P* (*grd-max P*)
  and *grd-min P* ≤ *sqp*
  and *sqp1* = *quote-reach P* (*y* + *quote-gross P sqp*)
  and *sqp2* = *quote-reach P′* (*y* + *quote-gross P′ sqp*)
shows *sqp2* ≤ *sqp1*
proof (*rule quote-reach-le-gross*)
  define *L* **where** *L* = *gross-fct* (*lq P*) (*fee P*)
  define *j* **where** *j* = *lower-tick P sqp*
  define *k* **where** *k* = *lower-tick P sqp1*
  have *sqp* < *sqp1*
    using *quote-reach-gt*
    by (*simp add*: *assms*(*1*) *assms*(*10*) *assms*(*12*) *assms*(*3*) *assms*(*9*))
  have *0* < *sqp*
      using *assms*
      by (*metis liq-grd-min less-add-same-cancel1 less-eq-real-def*
          pos-add-strict)
  show *sqp2* = *quote-reach P′* (*y* + *quote-gross P′ sqp*) **using** *assms* **by** *simp*
  have *y* + *quote-gross P sqp* = *quote-gross P sqp1*
    using *assms*(*1*,*3*,*9*,*10*,*12*) *clmm-quote-gross-pos clmm-quote-gross-reach-eq*
    by *auto*
  hence *y* = *quote-gross P sqp1* − *quote-gross P sqp* **by** *simp*
  also have ... ≤ *quote-gross P′ sqp1* − *quote-gross P′ sqp*
  proof (*rule fee-diff-quote-diff-le*)
    show *clmm-dsc P* **using** *assms* **by** *simp*
    show *clmm-dsc P′* **using** *assms* **by** *simp*
    show *L* = *gross-fct* (*lq P*) (*fee P*) **using** *L-def* **by** *simp*
    show *j* = *lower-tick P sqp* **using** *j-def* **by** *simp*
    show *k* = *lower-tick P sqp1* **using** *k-def* **by** *simp*
    show *fee-diff-on P P′ I* **using** *assms* **by** *simp*
    show *same-nz-liq-on P P′* {*m*. *j* ≤ *m* ∧ *m* ≤ *k* + *1*}
      by (*simp add*: *assms*(*6*) *assms*(*7*) *same-nz-liq-on-if-simil*)
    show *lower-tick P sqp* = *lower-tick P′ sqp*
      by (*meson assms*(*7*) *grd-lower-tick-cong*)
    show *lower-tick P sqp1* = *lower-tick P′ sqp1*
      by (*meson assms*(*7*) *grd-lower-tick-cong*)
    show ⋀*i*. *i* ∈ *I* ⟹ *fee P i* ≤ *fee P′ i* **using** *assms* **by** *simp*
    show *0* < *sqp* **using** ‹*0* < *sqp*› **.**

```
    show {m. L m ≠ 0 ∧ j ≤ m ∧ m ≤ k} ⊆ I
     proof
      fix m
      assume m ∈ {m. L m ≠ 0 ∧ j ≤ m ∧ m ≤ k}
      hence L m ≠ 0  using L-def by auto
      hence idx-min (lq P) ≤ m using L-def
        by (meson assms(1) clmm-dsc-gross-liq-zero-iff
            idx-min-lt-liq leI)
      moreover have m ≤ idx-max (lq P)
        using L-def ‹L m ≠ 0› fin-nz-sup idx-max-finite-ge assms(1)
          clmm-dsc-gross-liq-zero-iff
          by blast
      ultimately have m ∈ {idx-min (lq P) .. idx-max (lq P) + 1}
        using L-def by auto
      thus m∈ I using assms by auto
    qed
    show sqp ≤ sqp1 using ‹sqp < sqp1› by simp
  qed
  finally have y ≤ quote-gross P' sqp1 − quote-gross P' sqp .
  thus y + quote-gross P' sqp ≤ quote-gross P' sqp1 by simp
  show 0 < sqp1 using ‹0 < sqp› ‹sqp < sqp1› by simp
  show nz-support (lq P') ≠ {} using assms by simp
  show 0 < y + quote-gross P' sqp
    by (simp add: add-strict-increasing assms(2) assms(9)
        clmm-quote-gross-pos)
  show clmm-dsc P' using assms by simp
  have sqp1 ≤ grd-max P
    using quote-reach-leq-grd-max assms(1,3,9,10,12)
      clmm-quote-gross-pos
    by auto
  also have ... ≤ grd-max P' using same-nz-liq-on-grd-max
    by (meson assms(2) assms(3) assms(6) assms(7) clmm-dsc-grd-mono
        clmm-dsc-liq(1) mono-onI same-nz-liq-on-if-simil)
  finally show sqp1 ≤ grd-max P' .
qed

lemma fee-diff-le-imp-quote-swap:
  assumes clmm-dsc P
  and clmm-dsc P'
  and nz-support (lq P) ≠ {}
  and {idx-min (lq P) .. idx-max (lq P) + 1} ⊆ I
  and fee-diff-on P P' I
  and nz-support (lq P) = nz-support (lq P')
  and grd P = grd P'
  and ⋀i. i ∈ I ⟹ fee P i ≤ fee P' i
  and 0 < y
  and y + quote-gross P sqp ≤ quote-gross P (grd-max P)
  and grd-min P ≤ sqp
shows quote-swap P' sqp y ≤ quote-swap P sqp y
```

proof −
have *leq*: *quote-reach P′ (y + quote-gross P sqp) ≤*
quote-reach P (y + quote-gross P sqp)
proof (*rule fee-diff-le-imp-quote-reach*[*OF assms*(*1*−*5*)])
show *0 < y + quote-gross P sqp*
by (*simp add*: *add-pos-nonneg assms*(*1*) *assms*(*9*) *clmm-quote-gross-pos*)
show *y + quote-gross P sqp ≤ quote-gross P* (*grd-max P*) **using** *assms* **by** *simp*
show *⋀i. i ∈ I ⟹ fee P i ≤ fee P′ i* **using** *assms* **by** *simp*
show *same-nz-liq-on P P′ {i. i ≤ idx-max* (*lq P*) + *1}*
using *assms same-nz-liq-on-if-simil assms* **by** *simp*
qed
have *leq′*: *quote-reach P′ (y + quote-gross P′ sqp) ≤*
quote-reach P (y + quote-gross P sqp)
proof (*rule fee-diff-le-price-cmp*[*OF assms*(*1*−*5*)])
show *0 < y* **using** *assms*(*9*) **.**
show *⋀i. i ∈ I ⟹ fee P i ≤ fee P′ i* **using** *assms* **by** *simp*
show *nz-support* (*lq P*) = *nz-support* (*lq P′*) **using** *assms* **by** *simp*
show *grd P = grd P′* **using** *assms* **by** *simp*
show *grd-min P ≤ sqp* **using** *assms* **by** *simp*
show *y + quote-gross P sqp ≤ quote-gross P* (*grd-max P*) **using** *assms*(*10*) **.**
qed *simp*+
have *base-net P′* (*quote-reach P* (*y + quote-gross P sqp*)) ≤
base-net P′ (*quote-reach P′* (*y + quote-gross P′ sqp*))
by (*rule clmm-base-net-antimono*[*THEN antimonoD*], (*simp add*: *assms leq′*)+)
hence *quote-swap P′ sqp y ≤ base-net P′ sqp* −
base-net P′ (*quote-reach P* (*y + quote-gross P sqp*))
unfolding *quote-swap-def* **by** *simp*
also have ... = *quote-swap P sqp y*
using *fee-diff-simil-base-net assms* **unfolding** *quote-swap-def*
by (*smt* (*verit*, *ccfv-SIG*) *clmm-quote-gross-pos quote-reach-gt*
quote-reach-leq-grd-max)
finally show *quote-swap P′ sqp y ≤ quote-swap P sqp y* **.**
qed

lemma *fee-ge-quote-swap-le*:
assumes *clmm-dsc P*
and *clmm-dsc P′*
and *nz-support* (*lq P*) ≠ {}
and *grd P = grd P′*
and *lq P = lq P′*
and *⋀i. fee P i ≤ fee P′ i*
and *0 ≤ y*
and *0 < sqp*
and *y + quote-gross P sqp ≤ quote-gross P* (*grd-max P*)
shows *quote-swap P′ sqp y ≤ quote-swap P sqp y*
proof (*cases y = 0*)
case *True*
then show *?thesis* **using** *quote-swap-zero′*
using *assms*(*1*−*3*,*5*,*8*) *quote-gross-grd-max-max* **by** *auto*

```
next
  case False
  show ?thesis
  proof (cases grd-min P ≤ sqp)
    case True
    show ?thesis
    proof (rule fee-diff-le-imp-quote-swap)
      show fee-diff-on P P' {idx-min (lq P')..idx-max (lq P') + 1}
        by (simp add: assms(5) assms(4) fee-diff-onI id-grid-onI)
      show nz-support (lq P) ≠ {} using assms by simp
      show grd-min P ≤ sqp using True .
      show 0 < y using assms ‹¬ y = 0› by simp
    qed (simp add: assms)+
  next
    case False
    hence sqp < grd-min P'
      using assms unfolding grd-min-def idx-min-img-def idx-min-def by simp
    have grd-min P = grd-min P'
      using assms unfolding grd-min-def idx-min-img-def idx-min-def by simp
    have quote-swap P' sqp y = quote-swap P' (grd-min P') y
      using ‹sqp < grd-min P'› assms(2,3,5,8) quote-swap-grd-min by auto
    also have ... ≤ quote-swap P (grd-min P') y
    proof (rule fee-diff-le-imp-quote-swap)
      show nz-support (lq P) ≠ {} using assms(3,5) by simp
      show fee-diff-on P P' {idx-min (lq P')..idx-max (lq P') + 1}
        by (simp add: assms(5) assms(4) fee-diff-onI id-grid-onI)
      show y + quote-gross P (grd-min P') ≤ quote-gross P (grd-max P)
        using False assms(1,5,4,9,8) clmm-quote-gross-grd-min-le grd-min-def
        by auto
      show grd-min P ≤ grd-min P' using ‹grd-min P = grd-min P'› by simp
      show 0 < y using assms ‹¬ y = 0› by simp
    qed (simp add: assms)+
    also have ... = quote-swap P sqp y
      using quote-swap-grd-min
      by (simp add: ‹grd-min P = grd-min P'› ‹sqp < grd-min P'› assms(1,3,8))
    finally show ?thesis .
  qed
qed

end

end
theory CLMM-Transformation imports CLMM-Description




begin
```

6 CLMM transformations

6.1 CLMM pool refinement

Given a pool P and a (square root) price π, the refinement operation consists in defining a new grid (if necessary) in such a way that π is one of the bounds on the grid.

definition *refine* **where**
refine P sqp = (*let i* = *lower-tick P sqp in*
(*if* (*grd P i* = *sqp*) *then P else*
(*wedge* (*grd P*) *i sqp*, *wedge* (*lq P*) *i* (*lq P i*), *wedge* (*fee P*) *i* (*fee P i*))))

lemma *refine-eq*:
assumes *i* = *lower-tick P sqp*
and *grd P i* = *sqp*
shows *refine P sqp* = *P* **using** *assms* **unfolding** *refine-def Let-def* **by** *simp*

lemma *refine-lq*:
assumes *i* = *lower-tick P sqp*
and *grd P i* ≠ *sqp*
and *P′* = *refine P sqp*
shows *lq P′* = *wedge* (*lq P*) *i* (*lq P i*)
using *assms* **unfolding** *Let-def refine-def lq-def* **by** *simp*

lemma *refine-fee*:
assumes *i* = *lower-tick P sqp*
and *grd P i* ≠ *sqp*
and *P′* = *refine P sqp*
shows *fee P′* = *wedge* (*fee P*) *i* (*fee P i*)
using *assms* **unfolding** *Let-def refine-def fee-def* **by** *simp*

lemma *refine-grd*:
assumes *i* = *lower-tick P sqp*
and *P′* = *refine P sqp*
and *grd P i* ≠ *sqp*
shows *grd P′* = *wedge* (*grd P*) *i sqp*
using *assms* **unfolding** *refine-def grd-def Let-def* **by** *simp*

lemma *refine-grd-cong*:
assumes *P1* = *refine P sqp*
and *P2* = *refine P′ sqp*
and *grd P* = *grd P′*
shows *grd P1* = *grd P2*
proof (*cases grd P* (*lower-tick P sqp*) = *sqp*)
case *True*
hence *grd P′* (*lower-tick P′ sqp*) = *sqp*
using *assms* **unfolding** *lower-tick-def rng-blw-def* **by** *simp*
then show *?thesis* **using** *assms True* **unfolding** *refine-def Let-def* **by** *simp*
next

case *False*
define i **where** $i = lower\text{-}tick\ P\ sqp$
hence $i = lower\text{-}tick\ P'\ sqp$
using *assms* **unfolding** *lower-tick-def rng-blw-def* **by** *simp*
have $grd\ P1 = wedge\ (grd\ P)\ i\ sqp$
using *False assms* **unfolding** *refine-def grd-def i-def Let-def* **by** *simp*
also have $... = wedge\ (grd\ P')\ i\ sqp$ **using** *assms* **by** *simp*
also have $... = grd\ P2$
using *False assms* ‹$i = lower\text{-}tick\ P'\ sqp$›
unfolding *refine-def grd-def i-def Let-def* **by** *simp*
finally show *?thesis* **.**
qed

lemma *refine-grd-arg-le*:
assumes $i = lower\text{-}tick\ P\ sqp$
and $P' = refine\ P\ sqp$
and $j \leq i$
shows $grd\ P'\ j = grd\ P\ j$
proof (*cases* $grd\ P\ (lower\text{-}tick\ P\ sqp) = sqp$)
case *True*
hence $P' = P$ **using** *assms refine-eq* **by** *simp*
then show *?thesis* **by** *simp*
next
case *False*
hence $grd\ P' = wedge\ (grd\ P)\ i\ sqp$ **using** *assms refine-grd* **by** *simp*
then show *?thesis* **using** *assms* **by** *simp*
qed

lemma *refine-grd-arg-gt*:
assumes $i = lower\text{-}tick\ P\ sqp$
and $P' = refine\ P\ sqp$
and $grd\ P\ (lower\text{-}tick\ P\ sqp) \neq sqp$
and $i < j$
shows $grd\ P'\ (j+1) = grd\ P\ j$
proof −
have $grd\ P' = wedge\ (grd\ P)\ i\ sqp$ **using** *assms refine-grd* **by** *simp*
then show *?thesis* **using** *assms* **by** *simp*
qed

lemma *refine-grd-arg-Suc*:
assumes $i = lower\text{-}tick\ P\ sqp$
and $P' = refine\ P\ sqp$
and $grd\ P\ (lower\text{-}tick\ P\ sqp) \neq sqp$
shows $grd\ P'\ (i+1) = sqp$
proof −
have $grd\ P' = wedge\ (grd\ P)\ i\ sqp$ **using** *assms refine-grd* **by** *simp*
then show *?thesis* **using** *assms* **by** *simp*
qed

lemma *refine-fee-arg-le*:
  assumes $i = lower\text{-}tick\ P\ sqp$
  and $P' = refine\ P\ sqp$
  and $j \leq i$
shows $fee\ P'\ j = fee\ P\ j$
proof $(cases\ grd\ P\ (lower\text{-}tick\ P\ sqp) = sqp)$
  case *True*
  hence $P' = P$ **using** *assms refine-eq* **by** *simp*
  then show *?thesis* **by** *simp*
next
  case *False*
  hence $fee\ P' = wedge\ (fee\ P)\ i\ (fee\ P\ i)$ **using** *assms refine-fee* **by** *simp*
  then show *?thesis* **using** *assms* **by** *simp*
qed

lemma *refine-fee-arg-gt*:
  assumes $i = lower\text{-}tick\ P\ sqp$
  and $P' = refine\ P\ sqp$
  and $grd\ P\ (lower\text{-}tick\ P\ sqp) \neq sqp$
  and $i < j$
shows $fee\ P'\ (j{+}1) = fee\ P\ j$
proof −
  have $fee\ P' = wedge\ (fee\ P)\ i\ (fee\ P\ i)$ **using** *assms refine-fee* **by** *simp*
  then show *?thesis* **using** *assms* **by** *simp*
qed

lemma *refine-fee-arg-Suc*:
  assumes $i = lower\text{-}tick\ P\ sqp$
  and $P' = refine\ P\ sqp$
  and $grd\ P\ (lower\text{-}tick\ P\ sqp) \neq sqp$
shows $fee\ P'\ (i{+}1) = fee\ P\ i$
proof −
  have $fee\ P' = wedge\ (fee\ P)\ i\ (fee\ P\ i)$ **using** *assms refine-fee* **by** *simp*
  then show *?thesis* **using** *assms* **by** *simp*
qed

lemma *refine-lq-arg-le*:
  assumes $i = lower\text{-}tick\ P\ sqp$
  and $P' = refine\ P\ sqp$
  and $grd\ P\ i \neq sqp$
  and $j \leq i$
shows $lq\ P'\ j = lq\ P\ j$
proof −
  have $lq\ P' = wedge\ (lq\ P)\ i\ (lq\ P\ i)$
    using *refine-lq assms* **by** *simp*
  thus *?thesis* **using** *assms* **by** *simp*
qed

lemma *refine-lq-arg-gt*:

assumes $i = lower\text{-}tick\ P\ sqp$
and $P' = refine\ P\ sqp$
and $grd\ P\ i \neq sqp$
and $i < j$
shows $lq\ P'\ (j{+}1) = lq\ P\ j$
proof −
have $lq\ P' = wedge\ (lq\ P)\ i\ (lq\ P\ i)$
using *refine-lq assms* **by** *simp*
thus *?thesis* **using** *assms* **by** *simp*
qed

lemma *refine-lq-arg-Suc*:
assumes $i = lower\text{-}tick\ P\ sqp$
and $P' = refine\ P\ sqp$
and $grd\ P\ i \neq sqp$
shows $lq\ P'\ (i{+}1) = lq\ P\ i$
proof −
have $lq\ P' = wedge\ (lq\ P)\ i\ (lq\ P\ i)$
using *refine-lq assms* **by** *simp*
thus *?thesis* **using** *assms* **by** *simp*
qed

lemma *refine-on-grd*:
assumes *clmm-dsc P*
and $grd\ P\ i = sqp$
shows $refine\ P\ sqp = P$
proof −
have $i = lower\text{-}tick\ P\ sqp$ **using** *assms lower-tick-eq* **by** *simp*
thus *?thesis* **using** *assms* **unfolding** *refine-def Let-def* **by** *simp*
qed

lemma *refine-encomp-at-grd*:
assumes *clmm-dsc P*
and $P' = refine\ P\ sqp$
and $grd\ P\ (lower\text{-}tick\ P\ sqp) = sqp$
shows $encomp\text{-}at\ (grd\ P')\ (grd\ P)\ j\ j$
proof −
have $P' = P$ **using** *refine-on-grd assms* **by** *simp*
have $encomp\text{-}at\ (grd\ P')\ (grd\ P)\ j\ j$
proof
show $grd\ P\ j \leq grd\ P'\ j$ **using** ‹$P' = P$› **by** *simp*
show $grd\ P'\ (j + 1) \leq grd\ P\ (j + 1)$ **using** ‹$P' = P$› **by** *simp*
qed
thus *?thesis* **by** *auto*
qed

lemma *refine-encomp-at-arg-le*:
assumes *clmm-dsc P*
and $P' = refine\ P\ sqp$

```
  and i = lower-tick P sqp
  and grd P i ≠ sqp
  and j ≤ i
shows encomp-at (grd P') (grd P) j j
proof −
  have grd: grd P' = wedge (grd P) i sqp
    using assms unfolding Let-def refine-def grd-def by simp
  hence grd P' j = grd P j using ‹j ≤ i› by (simp add: grd)
  moreover have grd P' (j+1) ≤ grd P (j+1)
  proof (cases j = i)
    case True
    hence grd P' (j+1) = sqp using grd by simp
    also have ... < grd P (j+1) using ‹j = i› assms
      by (meson lower-tick-ubound)
    finally show grd P' (j+1) ≤ grd P (j+1) by simp
  next
    case False
    hence grd P' (j+1) ≤ grd P (j+1) using ‹j ≤ i› grd by auto
    then show ?thesis by simp
  qed
  ultimately show ?thesis using encomp-atI[of grd P j grd P' j] by simp
qed

lemma refine-encomp-at-arg-ge-Suc:
  assumes clmm-dsc P
  and P' = refine P sqp
  and i = lower-tick P sqp
  and grd P i ≠ sqp
  and i+1 ≤ j
  and 0 < sqp
shows encomp-at (grd P') (grd P) j (j−1)
proof −
  have grd: grd P' = wedge (grd P) i sqp
    using assms unfolding Let-def refine-def grd-def by simp
  show ?thesis
  proof (cases i+1 = j)
    case True
    hence grd P i ≤ grd P' j
      using lower-tick-lbound assms grd
      by (metis wedge-arg-eq)
    moreover have grd P' (j+1) ≤ grd P (i+1)
      using True wedge-arg-gt[of i j+1 grd P sqp] grd
      by (simp add: add.commute)
    ultimately show ?thesis using encomp-atI True by auto
  next
    case False
    hence grd P (j − 1) ≤ grd P' j using assms grd by fastforce
    moreover have grd P' (j+1) ≤ grd P j using grd False assms by simp
    ultimately show ?thesis using encomp-atI[of grd P j−1] by simp
```

```
  qed
qed

lemma refine-finer-range:
  assumes clmm-dsc P
  and 0 < sqp
  and P' = refine P sqp
shows finer-range (grd P') (grd P)
proof −
  define i where i = lower-tick P sqp
  show ?thesis
  proof (cases grd P i = sqp)
    case True
    then show ?thesis
      using assms refine-encomp-at-grd i-def unfolding finer-range-def by metis
  next
    case False
    show ?thesis unfolding finer-range-def
    proof
      fix j
      show ∃ k. encomp-at (grd P') (grd P) j k
      proof (cases j ≤ i)
        case True
        then show ?thesis
          using refine-encomp-at-arg-le[of P P'] assms i-def False by auto
      next
        case False
        show ?thesis
        proof (cases j = i+1)
          case True
          then show ?thesis
            using refine-encomp-at-arg-ge-Suc assms i-def
            by (meson dual-order.refl refine-encomp-at-grd)
        next
          case False
          hence i+1 < j using ‹¬ j ≤ i› by simp
          then show ?thesis
            using refine-encomp-at-arg-ge-Suc False assms i-def
            by (meson refine-encomp-at-grd zle-add1-eq-le zless-add1-eq)
        qed
      qed
    qed
  qed
qed

lemma refine-finite-liq:
  assumes finite-liq P
  and P' = refine P sqp
shows finite-liq P'
```

```
proof (cases grd P (lower-tick P sqp) = sqp)
  case True
  then show ?thesis
    using assms clmm-dsc-liq unfolding refine-def Let-def by simp
next
  case False
  define j where j = lower-tick P sqp
  have grd: grd P' = wedge (grd P) j sqp using j-def assms False
    unfolding refine-def Let-def grd-def by simp
  have lq: lq P' = wedge (lq P) j (lq P j) using j-def assms False
    unfolding refine-def Let-def lq-def by simp
  show ?thesis
      using grd wedge-finite-nz-support assms clmm-dsc-liq(1)
      unfolding finite-liq-def by (metis lq)
qed

lemma refine-clmm:
  assumes clmm-dsc P
  and 0 < sqp
  and P' = refine P sqp
shows clmm-dsc P'
proof (cases grd P (lower-tick P sqp) = sqp)
  case True
  then show ?thesis using assms unfolding refine-def Let-def by simp
next
  case False
  define j where j = lower-tick P sqp
  hence grd P j ≤ sqp
    by (simp add: assms lower-tick-lbound)
  hence grd P j < sqp using False j-def by simp
  have sqp < grd P (j+1) using j-def assms lower-tick-ubound by simp
  have grd: grd P' = wedge (grd P) j sqp using j-def assms False
    unfolding refine-def Let-def grd-def by simp
  have lq: lq P' = wedge (lq P) j (lq P j) using j-def assms False
    unfolding refine-def Let-def lq-def by simp
  have fee: fee P' = wedge (fee P) j (fee P j) using j-def assms False
    unfolding refine-def Let-def fee-def by simp
  show ?thesis
  proof
    show span-grid P'
    proof
      show strict-mono (grd P')
      proof (rule wedge-strict-mono)
        show grd P' = wedge (grd P) j sqp using grd .
        show grd P j < sqp using ‹grd P j < sqp› .
        show sqp < grd P (j+1) using ‹sqp < grd P (j+1)› .
        show strict-mono (grd P) using assms by simp
      qed
      show ∀ i. 0 < grd P' i
```

```
      using grd assms wedge-gt by (metis clmm-dsc-grid(2))
    show ∀ r>0. ∃ i. grd P' i < r
      using grd wedge-as-lbound by (simp add: assms(1))
    show ∀ r. ∃ i. r < grd P' i
      using grd wedge-as-ubound by (simp add: assms(1))
  qed
  show ∀ i. 0 ≤ fee P' i using wedge-ge fee assms
    by (metis clmm-dsc-fees)
  show ∀ i. fee P' i < 1 using wedge-lt fee assms
    by (metis clmm-dsc-fees)
  show ∀ i. 0 ≤ lq P' i using wedge-ge lq assms
    by (metis clmm-dsc-liq(2))
  show finite-liq P'
    using refine-finite-liq assms clmm-dsc-liq by simp
 qed
qed

lemma refine-lower-tick-idx:
 assumes clmm-dsc P
 and 0 < sqp
 and i = lower-tick P sqp
 and P' = refine P sqp
 and grd P (lower-tick P sqp) ≠ sqp
shows lower-tick P' sqp = i+1
proof −
 have clmm-dsc P' using refine-clmm assms by simp
 moreover have grd P' (i+1) = sqp
  using refine-grd-arg-Suc assms by simp
 ultimately show ?thesis using ‹clmm-dsc P'› lower-tick-eq by simp
qed

lemma refine-ge-lower-tick-eq:
 assumes clmm-dsc P
 and 0 < sqp
 and i = lower-tick P sqp'
 and P' = refine P sqp
 and grd P (lower-tick P sqp) ≠ sqp
 and sqp ≤ sqp'
 and lower-tick P sqp = lower-tick P sqp'
shows lower-tick P' sqp' = i+1
proof (rule lower-tick-charact)
 show clmm-dsc P' using assms refine-clmm by simp
 show grd P' (i + 1) ≤ sqp' by (metis assms(3−7) refine-grd-arg-Suc)
 have sqp' < grd P (i+1)
  by (simp add: assms(1) assms(3) lower-tick-ubound)
 also have ... = grd P' (i + 1 + 1)
  by (metis assms(3−5,7)less-add-one refine-grd-arg-gt)
 finally show sqp' < grd P' (i + 1 + 1) .
qed
```

lemma *refine-ge-lower-tick-gt*:
assumes *clmm-dsc P*
and *0 < sqp*
and *sqp ≤ sqp′*
and *i = lower-tick P sqp′*
and *P′ = refine P sqp*
and *grd P (lower-tick P sqp) ≠ sqp*
and *lower-tick P sqp < lower-tick P sqp′*
shows *lower-tick P′ sqp′ = i+1*
proof (*rule lower-tick-charact*)
show *clmm-dsc P′* **using** *assms refine-clmm* **by** *simp*
define *j* **where** *j = lower-tick P sqp*
have *sqp < sqp′* **using** *assms lower-tick-lt* **by** *simp*
have *grd P′ = wedge (grd P) j sqp* **using** *assms j-def refine-grd* **by** *simp*
hence *grd P′ (i+1) = grd P i* **using** *assms(4,7) j-def* **by** *force*
also have *... ≤ sqp′* **using** *assms lower-tick-geq* **by** *simp*
finally show *grd P′ (i+1) ≤ sqp′* **.**
have *sqp′ < grd P (i+1)*
by (*simp add*: *assms(1) assms(4) lower-tick-ubound*)
also have *... = grd P′ (i+1 + 1)*
by (*metis assms(4−7) dual-order.strict-trans less-add-one refine-grd-arg-gt*)
finally show *sqp′ < grd P′ (i + 1 + 1)* **.**
qed

lemma *refine-ge-lower-tick*:
assumes *clmm-dsc P*
and *0 < sqp*
and *sqp ≤ sqp′*
and *i = lower-tick P sqp′*
and *P′ = refine P sqp*
and *grd P (lower-tick P sqp) ≠ sqp*
shows *lower-tick P′ sqp′ = i+1*
proof (*cases lower-tick P sqp = lower-tick P sqp′*)
case *True*
then show *?thesis* **using** *assms refine-ge-lower-tick-eq* **by** *simp*
next
case *False*
then show *?thesis* **using** *assms refine-ge-lower-tick-gt*
by (*smt* (*verit*) *lower-tick-mono*)
qed

lemma *refine-lower-tick*:
assumes *clmm-dsc P*
and *P′ = refine P sqp*
and *0 < sqp*
shows *grd P′ (lower-tick P′ sqp) = sqp*
proof (*cases grd P (lower-tick P sqp) = sqp*)
case *True*

then show *?thesis* **using** *refine-eq assms* **by** *simp*
next
case *False*
define i **where** $i = lower\text{-}tick\ P\ sqp$
have $grd\ P' = wedge\ (grd\ P)\ i\ sqp$
using *assms False refine-grd i-def* **by** *simp*
hence $grd\ P'\ (i+1) = sqp$ **using** *wedge-arg-eq assms* **by** *simp*
moreover have $clmm\text{-}dsc\ P'$ **using** *assms refine-clmm* **by** *simp*
ultimately show *?thesis* **by** (*simp add*: *lower-tick-eq*)
qed

lemma *refine-finer-ranges*:
assumes *clmm-dsc P*
and $0 < sqp$
and $P' = refine\ P\ sqp$
shows $finer\text{-}ranges\ (grd\ P')\ (grd\ P)$
proof (*rule finer-ranges.intro*)
show $strict\text{-}mono\ (grd\ P')$
using *refine-clmm clmm-dsc-grid assms* **by** *metis*
show $mono\ (grd\ P)$ **using** *assms clmm-dsc-grid*
by (*simp add*: *strict-mono-mono*)
show $finer\text{-}range\ (grd\ P')\ (grd\ P)$ **using** *assms refine-finer-range*
by (*metis*)
qed

lemma *refine-coarse-idx-grd*:
assumes *clmm-dsc P*
and $P' = refine\ P\ sqp$
and $grd\ P\ (lower\text{-}tick\ P\ sqp) = sqp$
shows $coarse\text{-}idx\ (grd\ P')\ (grd\ P)\ j = j$
proof −
interpret *finer-ranges grd P' grd P*
using *refine-finer-ranges*[*of P sqp P'*] *assms*
by (*metis clmm-dsc-grid*(*2*))
show *?thesis* **using** *coarse-idx-eq refine-encomp-at-grd assms*
by (*metis clmm-dsc-grd-Suc inf.cobounded1 inf.strict-order-iff*
refine-on-grd)
qed

lemma *refine-coarse-idx-arg-le*:
assumes *clmm-dsc P*
and $P' = refine\ P\ sqp$
and $i = lower\text{-}tick\ P\ sqp$
and $grd\ P\ i \neq sqp$
and $j \leq i$
and $0 < sqp$
shows $coarse\text{-}idx\ (grd\ P')\ (grd\ P)\ j = j$
proof −
interpret *finer-ranges grd P' grd P*

```
    using refine-finer-ranges[of P sqp P'] assms by metis
  show ?thesis using coarse-idx-eq refine-encomp-at-arg-le assms
    by (metis coarse-idx-bounds encomp-idx-unique)
qed

lemma refine-coarse-idx-arg-gt:
  assumes clmm-dsc P
  and 0 < sqp
  and P' = refine P sqp
  and i = lower-tick P sqp
  and grd P i ≠ sqp
  and i+1 ≤ j
shows coarse-idx (grd P') (grd P) j = j−1
proof −
  interpret finer-ranges grd P' grd P
    using refine-finer-ranges[of P sqp P'] assms by metis
  show ?thesis
    using coarse-idx-eq coarse-idx-bounds refine-encomp-at-arg-ge-Suc
    by (metis assms encomp-idx-unique)
qed

lemma refine-lq-idx-neq:
  assumes clmm-dsc P
  and 0 < sqp
  and P' = refine P sqp
  and grd P (lower-tick P sqp) ≠ sqp
  shows lq P' j = lq P (pool-coarse-idx P' P j)
proof −
  define i where i = lower-tick P sqp
  fix j
  {
    assume j ≤ i
    hence pool-coarse-idx P' P j = j
      using refine-coarse-idx-arg-le assms i-def
      unfolding pool-coarse-idx-def by simp
    moreover have lq P' j = lq P j
      using ‹j ≤ i› refine-lq assms i-def by simp
    ultimately have pool-coarse-idx P' P j = j lq P' j = lq P j
      by auto
  } note a = this
  {
    assume i+1 ≤ j
    hence pool-coarse-idx P' P j = j−1
      using refine-coarse-idx-arg-gt assms i-def
      unfolding pool-coarse-idx-def by simp
    moreover have lq P' j = lq P (j−1)
    proof (cases i+1 = j)
      case True
      then show ?thesis using refine-lq assms wedge-arg-eq unfolding i-def
```

```
      by auto
    next
      case False
      hence i+1 < j using ‹i+1 ≤ j› by simp
      then show ?thesis using refine-lq assms wedge-arg-gt unfolding i-def
        by simp
    qed
    ultimately have pool-coarse-idx P' P j = j−1 lq P' j = lq P (j−1)
      by auto
  } note b = this
  show lq P' j = lq P (pool-coarse-idx P' P j) using a b by fastforce
qed

lemma refine-fee-idx-neq:
  assumes clmm-dsc P
  and 0 < sqp
  and P' = refine P sqp
  and grd P (lower-tick P sqp) ≠ sqp
  shows fee P' j = fee P (pool-coarse-idx P' P j)
proof −
  define i where i = lower-tick P sqp
  fix j
  {
    assume j ≤ i
    hence pool-coarse-idx P' P j = j
      using refine-coarse-idx-arg-le assms i-def
      unfolding pool-coarse-idx-def by simp
    moreover have fee P' j = fee P j
      using ‹j ≤ i› refine-fee assms i-def by simp
    ultimately have pool-coarse-idx P' P j = j fee P' j = fee P j by auto
  } note a = this
  {
    assume i+1 ≤ j
    hence pool-coarse-idx P' P j = j−1
      using refine-coarse-idx-arg-gt assms i-def
      unfolding pool-coarse-idx-def by simp
    moreover have fee P' j = fee P (j−1)
    proof (cases i+1 = j)
      case True
      then show ?thesis using refine-fee assms wedge-arg-eq unfolding i-def
        by auto
    next
      case False
      hence i+1 < j using ‹i+1 ≤ j› by simp
      then show ?thesis using refine-fee assms wedge-arg-gt unfolding i-def
        by simp
    qed
    ultimately have pool-coarse-idx P' P j = j−1 fee P' j = fee P (j−1)
      by auto
```

} **note** $b = this$
show $fee\ P'\ j = fee\ P\ (pool\text{-}coarse\text{-}idx\ P'\ P\ j)$ **using** $a\ b$ **by** *fastforce*
qed

lemma *refine-cst-fees*:
assumes $\bigwedge i.\ fee\ P\ i = phi$
and $P' = refine\ P\ sqp$
shows $\bigwedge i.\ fee\ P'\ i = phi$
by (*smt* (*verit*, *ccfv-SIG*) *assms refine-eq refine-fee refine-fee-arg-Suc wedge-arg-gt wedge-arg-lt*)

lemma *refine-finer-neq*:
assumes *clmm-dsc P*
and $0 < sqp$
and $P' = refine\ P\ sqp$
and $grd\ P\ (lower\text{-}tick\ P\ sqp) \neq sqp$
shows $finer\text{-}pool\ P'\ P$
proof (*intro finer-poolI conjI allI*)
define i **where** $i = lower\text{-}tick\ P\ sqp$
show $finer\text{-}range\ (grd\ P')\ (grd\ P)$ **using** *refine-finer-range assms* **by** *simp*
show $\bigwedge j.\ lq\ P'\ j = lq\ P\ (pool\text{-}coarse\text{-}idx\ P'\ P\ j)$
using *refine-lq-idx-neq assms* **by** *simp*
show $\bigwedge j.\ fee\ P'\ j = fee\ P\ (pool\text{-}coarse\text{-}idx\ P'\ P\ j)$
using *refine-fee-idx-neq assms* **by** *simp*
qed

lemma *refine-finer*:
assumes *clmm-dsc P*
and $0 < sqp$
and $P' = refine\ P\ sqp$
shows $finer\text{-}pool\ P'\ P$
proof (*cases grd P* (*lower-tick P sqp*) = *sqp*)
case *True*
hence $P' = P$ **using** *assms refine-on-grd* **by** *simp*
then show *?thesis* **using** *finer-pool-refl assms* **by** *simp*
next
case *False*
then show *?thesis* **using** *assms refine-finer-neq* **by** *simp*
qed

lemma *refine-nz-lq-sub*:
assumes *clmm-dsc P*
and $0 < sqp$
and $P' = refine\ P\ sqp$
shows $(\lambda j.\ pool\text{-}coarse\text{-}idx\ P'\ P\ j)\ `\ nz\text{-}support\ (lq\ P') \subseteq$
$nz\text{-}support\ (lq\ P)$
proof (*cases grd P* (*lower-tick P sqp*) = *sqp*)
case *True*
then show *?thesis* **using** *refine-eq assms coarse-idx-refl*

```
    by (simp add: pool-coarse-idx-def)
next
  case False
  show ?thesis
  proof
    fix l
    assume l∈ (λj. pool-coarse-idx P' P j) ` nz-support (lq P')
    hence ∃k ∈ nz-support (lq P'). l = pool-coarse-idx P' P k by auto
    from this obtain k where k ∈ nz-support (lq P')
        and l = pool-coarse-idx P' P k by auto
    hence lq P' k≠ 0 unfolding nz-support-def by simp
    hence lq P l ≠ 0
      using assms ‹l = pool-coarse-idx P' P k› refine-lq-idx-neq False by simp
    thus l ∈ nz-support (lq P) unfolding nz-support-def by simp
  qed
qed

lemma refine-nz-lq-ne:
  assumes clmm-dsc P
  and P' = refine P sqp
  and nz-support (lq P) ≠ {}
shows nz-support (lq P') ≠ {}
proof (cases grd P (lower-tick P sqp) = sqp)
  case True
  then show ?thesis using refine-eq assms by simp
next
  case False
  have ∃j. j∈ (nz-support (lq P)) using assms by auto
  from this obtain j where j ∈ nz-support (lq P) by auto
  hence lq P j≠ 0 unfolding nz-support-def by simp
  hence j∈ nz-support (lq P') ∨ j+1 ∈ nz-support (lq P')
    unfolding nz-support-def
    by (smt (verit) False assms(2) mem-Collect-eq refine-lq refine-lq-arg-gt
        wedge-arg-lt)
  thus nz-support (lq P') ≠ {} by auto
qed

lemma refine-nz-lq-emp:
  assumes clmm-dsc P
  and P' = refine P sqp
  and nz-support (lq P) = {}
shows nz-support (lq P') = {}
proof (cases grd P (lower-tick P sqp) = sqp)
  case True
  then show ?thesis using refine-eq assms by simp
next
  case False
  {
    fix j
```

```
    have lq P j = 0 lq P (j−1) = 0
      using assms unfolding nz-support-def by simp+
    hence lq P′ j = 0 using assms refine-lq-arg-le refine-lq-arg-gt
      by (smt (verit, del-insts) False refine-lq wedge-arg-eq wedge-arg-gt)
  }
  thus ?thesis unfolding nz-support-def by simp
qed

lemma refine-idx-min-eq:
  assumes clmm-dsc P
  and P′ = refine P sqp
  and idx-min (lq P) ≤ lower-tick P sqp
shows idx-min (lq P′) = idx-min (lq P)
proof −
  interpret finite-liq-pool
    by (simp add: assms(1) clmm-dsc-liq(1) finite-liq-pool.intro)
  show ?thesis
  proof (cases nz-support (lq P) = {})
    case True
    hence nz-support (lq P′) = {} using assms refine-nz-lq-emp by simp
    then show ?thesis by (simp add: True idx-min-def)
  next
    case False
    show ?thesis
    proof (rule idx-min-finiteI[symmetric])
      define i where i = idx-min (lq P)
      hence lq P i ≠ 0 using False by (simp add: idx-min-mem nz-supportD)
      thus lq P′ i ≠ 0 using refine-lq-arg-le assms
        by (metis i-def refine-eq)
      show finite (nz-support (lq P′))
        using assms refine-finite-liq clmm-dsc-liq unfolding finite-liq-def
        by simp
      fix j
      assume j < i
      hence lq P j = 0 using i-def
        by (simp add: False fin-nz-sup idx-min-finite-lt)
      thus lq P′ j = 0 using refine-lq-arg-le assms
        by (metis ‹j < i› dual-order.strict-trans1 i-def leD nle-le refine-on-grd)
    qed
  qed
qed

lemma refine-idx-min-Suc-eq:
  assumes clmm-dsc P
  and P′ = refine P sqp
  and nz-support (lq P) ≠ {}
  and grd P (lower-tick P sqp) ≠ sqp
  and lower-tick P sqp < idx-min (lq P)
shows idx-min (lq P′) = idx-min (lq P) + 1
```

```
proof −
  interpret finite-liq-pool
    by (simp add: assms(1) clmm-dsc-liq(1) finite-liq-pool.intro)
  show ?thesis
  proof (rule idx-min-finiteI[symmetric])
    show finite (nz-support (lq P'))
      using assms refine-finite-liq clmm-dsc-liq unfolding finite-liq-def
      by simp
    define i where i = idx-min (lq P)
    hence lq P i ≠ 0 using assms by (simp add: idx-min-mem nz-supportD)
    thus lq P' (i+1) ≠ 0 using refine-lq-arg-gt assms i-def by simp
    fix j
    assume j < i + 1
    hence lq P (j−1) = 0 using i-def
      by (simp add: assms fin-nz-sup idx-min-finite-lt)
    show lq P' j = 0
    proof (cases j ≤ lower-tick P sqp)
      case True
      hence lq P j = 0 using assms
        by (simp add: fin-nz-sup idx-min-finite-lt)
      thus lq P' j = 0 using refine-lq-arg-le assms i-def True by simp
    next
      case False
      show ?thesis
      proof (cases j = lower-tick P sqp + 1)
        case True
        then show ?thesis
          using refine-lq-arg-Suc assms i-def ‹lq P (j − 1) = 0› by simp
      next
        case False
        hence lower-tick P sqp < j − 1 using ‹¬ j ≤ lower-tick P sqp› by simp
        thus ?thesis
          using ‹lq P (j−1) = 0› refine-lq-arg-gt assms i-def by fastforce
      qed
    qed
  qed
qed

lemma refine-grd-min:
  assumes clmm-dsc P
  and P' = refine P sqp
  and nz-support (lq P) ≠ {}
shows grd-min P = grd-min P'
proof (cases grd P (lower-tick P sqp) = sqp)
  case True
  hence P = P' using assms refine-eq by simp
  then show ?thesis by simp
next
  case False
```

```
  define i where i = idx-min (lq P)
  define k where k = lower-tick P sqp
  show ?thesis
  proof (cases i ≤ k)
    case True
    hence idx-min (lq P') = i
      using i-def refine-idx-min-eq k-def assms by simp
    moreover have grd P' i = grd P i
      using refine-grd-arg-le assms k-def True by simp
   ultimately show ?thesis unfolding grd-min-def idx-min-img-def i-def by simp
  next
    case False
    hence idx-min (lq P') = i+1
      using assms ‹grd P (lower-tick P sqp) ≠ sqp› refine-idx-min-Suc-eq
        k-def i-def
      by simp
    moreover have grd P' (i+1) = grd P i
      using refine-grd-arg-gt[of lower-tick P sqp P sqp P' i] ‹¬ i ≤ k›
        assms calculation i-def k-def refine-eq
      by fastforce
   ultimately show ?thesis unfolding grd-min-def idx-min-img-def i-def by simp
  qed
qed

lemma refine-idx-max-eq:
  assumes clmm-dsc P
  and P' = refine P sqp
  and idx-max (lq P) < lower-tick P sqp
shows idx-max (lq P') = idx-max (lq P)
proof −
  interpret finite-liq-pool
    by (simp add: assms(1) clmm-dsc-liq(1) finite-liq-pool.intro)
  show ?thesis
  proof (cases grd P (lower-tick P sqp) = sqp)
    case True
    hence P' = P using refine-eq assms by simp
    then show ?thesis by simp
  next
    case False
    show ?thesis
    proof (cases nz-support (lq P) = {})
      case True
      hence nz-support (lq P') = {} using assms refine-nz-lq-emp by simp
      then show ?thesis by (simp add: True idx-max-def)
    next
      case False
      show ?thesis
      proof (rule idx-max-finiteI[symmetric])
        define i where i = idx-max (lq P)
```

```
      hence lq P i ≠ 0 using False by (simp add: idx-max-mem nz-supportD)
      thus lq P' i ≠ 0 using refine-lq-arg-le assms
        by (metis i-def refine-on-grd zle-add1-eq-le zless-add1-eq)
      show finite (nz-support (lq P'))
        using assms refine-finite-liq clmm-dsc-liq unfolding finite-liq-def
        by simp
      fix j
      assume i < j
      hence lq P j = 0 using i-def False fin-nz-sup idx-max-finite-gt by auto
      show lq P' j = 0
      proof (cases j ≤ lower-tick P sqp)
        case True
        then show ?thesis
          by (metis ‹lq P j = 0› assms(2) refine-eq refine-lq-arg-le)
      next
        case False
        show ?thesis
        proof (cases j = lower-tick P sqp + 1)
          case True
          hence lq P' j = lq P (lower-tick P sqp)
            using assms refine-lq-arg-Suc
            by (metis ‹lq P j = 0›  fin-nz-sup idx-max-finite-gt refine-eq)
          also have ... = 0
            using assms idx-max-finite-gt by (metis fin-nz-sup)
          finally show ?thesis .
        next
          case False
          hence i < j−1 using i-def assms ‹¬ j ≤ lower-tick P sqp› by simp
          have lq P' j = lq P (j−1)
            using refine-lq-arg-gt[of - P sqp P' j−1] False
              ‹¬ j ≤ lower-tick P sqp› ‹grd P (lower-tick P sqp) ≠ sqp› assms
            by simp
          also have ... = 0
            using ‹i < j−1› i-def False fin-nz-sup idx-max-finite-gt by metis
          finally show ?thesis .
        qed
      qed
    qed
  qed
qed

lemma refine-idx-max-Suc-eq:
  assumes clmm-dsc P
  and P' = refine P sqp
  and nz-support (lq P) ≠ {}
  and grd P (lower-tick P sqp) ≠ sqp
  and lower-tick P sqp ≤ idx-max (lq P)
shows idx-max (lq P') = idx-max (lq P) + 1
```

```
proof −
  interpret finite-liq-pool
    by (simp add: assms(1) clmm-dsc-liq(1) finite-liq-pool.intro)
  show ?thesis
  proof (rule idx-max-finiteI[symmetric])
    show finite (nz-support (lq P'))
      using assms refine-finite-liq clmm-dsc-liq unfolding finite-liq-def
      by simp
    define i where i = idx-max (lq P)
    hence lq P i ≠ 0 using assms by (simp add: idx-max-mem nz-supportD)
    show lq P' (i+1) ≠ 0
    proof (cases i = lower-tick P sqp)
      case True
      then show ?thesis
        using refine-lq-arg-Suc assms ‹lq P i ≠ 0› unfolding i-def by simp
    next
      case False
      then show ?thesis using refine-lq-arg-gt assms ‹lq P i ≠ 0› i-def by simp
    qed
    fix j
    assume i + 1 < j
    hence lq P' j = lq P (j−1)
      using refine-lq-arg-gt[of - P - P' j−1] assms i-def ‹i + 1 < j›
        fin-nz-sup i-def
      by force
    also have ... = 0 using i-def ‹i + 1 < j›
      by (simp add: assms fin-nz-sup idx-max-finite-gt)
    finally show lq P' j = 0 .
  qed
qed

lemma refine-lower-tick-idx-max:
  assumes clmm-dsc P
  and 0 < sqp
  and P' = refine P sqp
  and nz-support (lq P) ≠ {}
  and lower-tick P sqp ≤ idx-max (lq P)
shows lower-tick P' sqp ≤ idx-max (lq P')
proof (cases grd P (lower-tick P sqp) = sqp)
  case True
  then show ?thesis using assms refine-eq by simp
next
  case False
  hence idx-max (lq P') = idx-max (lq P) + 1
    using assms refine-idx-max-Suc-eq by simp
  moreover have lower-tick P' sqp = lower-tick P sqp + 1
    using False refine-lower-tick-idx
    by (simp add: assms(1−3))
  ultimately show ?thesis using assms by simp
```

qed

lemma *refine-grd-max*:
assumes *clmm-dsc P*
and $P' = refine\ P\ sqp$
and $nz\text{-}support\ (lq\ P) \neq \{\}$
shows $grd\text{-}max\ P = grd\text{-}max\ P'$
proof (*cases grd P* (*lower-tick P sqp*) = *sqp*)
case *True*
hence $P = P'$ **using** *assms refine-eq* **by** *simp*
then show *?thesis* **by** *simp*
next
case *False*
define *i* **where** $i = idx\text{-}max\ (lq\ P)$
define *k* **where** $k = lower\text{-}tick\ P\ sqp$
show *?thesis*
proof (*cases i* < *k*)
case *True*
hence $idx\text{-}max\ (lq\ P') = i$
using *i-def refine-idx-max-eq k-def assms* **by** *simp*
moreover have $grd\ P'\ (i{+}1) = grd\ P\ (i{+}1)$
using *refine-grd-arg-le assms k-def True* **by** *simp*
ultimately show *?thesis*
unfolding *grd-max-def idx-max-img-def i-def* **by** *simp*
next
case *False*
hence $idx\text{-}max\ (lq\ P') = i{+}1$
using *assms* ‹*grd P* (*lower-tick P sqp*) ≠ *sqp*› *refine-idx-max-Suc-eq*
k-def i-def
by *simp*
moreover have $grd\ P'\ (i{+}2) = grd\ P\ (i{+}1)$
using *refine-grd-arg-gt*[*of lower-tick P sqp P sqp P' i*+*1*] ‹¬ *i* < *k*›
assms calculation i-def k-def refine-eq
by (*metis is-num-normalize*(*1*) *one-add-one verit-comp-simplify1*(*3*)
zle-add1-eq-le)
ultimately show *?thesis* **unfolding** *grd-max-def idx-max-img-def i-def*
by (*simp add*: *add.commute*)
qed
qed

lemma *refine-quote-gross*:
assumes *clmm-dsc P*
and $P' = refine\ P\ sqp$
and *0* < *sqp*
shows $quote\text{-}gross\ P' = quote\text{-}gross\ P$
proof (*rule finer-clmm.finer-quote-gross-eq*)
show *finer-clmm P' P*
proof
show *clmm-dsc P* **using** *assms* **by** *simp*

```
    show clmm-dsc P' using assms refine-clmm by simp
    show finer-pool P' P using assms refine-finer by simp
  qed
qed

lemma refine-nonzero-liq:
  assumes clmm-dsc P
  and lower-tick P sqp ≤ i
  and grd P (lower-tick P sqp) ≠ sqp
  and P' = refine P sqp
  and L = lq P
  and L' = lq P'
shows {l. L' l ≠ 0 ∧ i+1 < l} = (λi. i + 1) ` {k. L k ≠ 0 ∧ i < k}
proof
  show {l. L' l ≠ 0 ∧ i+1 < l} ⊆ (λi. i + 1) ` {k. L k ≠ 0 ∧ i < k}
  proof
    fix x
    assume x ∈ {l. L' l ≠ 0 ∧ i+1 < l}
    hence L' x ≠ 0 and i+1 < x by simp+
    hence L' x = L (x−1) using assms(2−6) refine-lq by auto
    moreover have i < x−1 using ‹i+1 < x› by simp
    ultimately have x−1 ∈ {k. L k ≠ 0 ∧ i < k} using ‹L' x ≠ 0› by auto
    thus x ∈ (λi. i + 1) ` {k. L k ≠ 0 ∧ i < k}
      by (simp add: rev-image-eqI)
  qed
next
  show (λi. i + 1) ` {k. L k ≠ 0 ∧ i < k} ⊆ {l. L' l ≠ 0 ∧ i + 1 < l}
  proof
    fix x
    assume x ∈ (λi. i + 1) ` {k. L k ≠ 0 ∧ i < k}
    hence ∃ y. y ∈ {k. L k ≠ 0 ∧ i < k} ∧ x = y+1 by auto
    from this obtain y where y ∈ {k. L k ≠ 0 ∧ i < k} and x = y+1 by auto
    hence L y ≠ 0 and i < y by simp+
    hence L' x ≠ 0 using ‹x = y + 1› assms(2−6) refine-lq-arg-gt by auto
    moreover have i+1 < x using ‹x = y+1› ‹i < y› by simp
    ultimately show x∈ {l. L' l ≠ 0 ∧ i + 1 < l} by auto
  qed
qed

lemma refine-pool-base-net-grd-eq:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and P' = refine P sqp
  and 0 < sqp
  and sqp < grd-max P
  and grd P (lower-tick P sqp) ≠ sqp
  and sqp ≤ sqp'
shows base-net P' sqp' = base-net P sqp'
proof −
```

have *clmm-dsc* P' **using** *assms refine-clmm* **by** *simp*
define L **where** $L = lq\ P$
define L' **where** $L' = lq\ P'$
define j **where** $j = lower\text{-}tick\ P'\ sqp'$
define i **where** $i = lower\text{-}tick\ P\ sqp'$
have *lower-tick* $P\ sqp \leq i$ **using** *assms*(1,4,7) *i-def lower-tick-mono* **by** *auto*
have $j = i + 1$ **using** *refine-ge-lower-tick assms j-def i-def* **by** *simp*
have *base-net* $P'\ sqp' = L'\ j * (inverse\ sqp' - inverse\ (grd\ P'\ (j + 1))) +$
$(\sum l \mid L'\ l \neq 0 \wedge j < l.$
$L'\ l * (inverse\ (grd\ P'\ l) - inverse\ (grd\ P'\ (l + 1))))$
using *base-net-sum j-def L'-def assms* ‹*clmm-dsc* P'› **by** *auto*
also have $... = L\ i * (inverse\ sqp' - inverse\ (grd\ P\ (i + 1))) +$
$(\sum l \mid L'\ l \neq 0 \wedge j < l.$
$L'\ l * (inverse\ (grd\ P'\ l) - inverse\ (grd\ P'\ (l + 1))))$
proof −
have $grd\ P'\ (j+1) = grd\ P\ (i+1)$
using *refine-grd-arg-gt* ‹$j = i + 1$› *assms*(1,3,4,6,7) *i-def lower-tick-mono zle-add1-eq-le*
by *presburger*
moreover have $L\ i = L'\ j$ **using** *refine-lq-arg-Suc* ‹$j = i + 1$›
by (*metis L'-def L-def assms*(1,3,4,6,7) *i-def lower-tick-mono refine-lq-arg-gt zle-add1-eq-le zless-add1-eq*)
ultimately show *?thesis* **by** *simp*
qed
also have $... = L\ i * (inverse\ sqp' - inverse\ (grd\ P\ (i + 1))) +$
$(\sum k \mid L\ k \neq 0 \wedge i < k.$
$L\ k * (inverse\ (grd\ P\ k) - inverse\ (grd\ P\ (k + 1))))$
proof −
have $(\sum l \mid L'\ l \neq 0 \wedge j < l.$
$L'\ l * (inverse\ (grd\ P'\ l) - inverse\ (grd\ P'\ (l + 1)))) =$
$(\sum k \mid L\ k \neq 0 \wedge i < k.$
$L\ k * (inverse\ (grd\ P\ k) - inverse\ (grd\ P\ (k + 1))))$
proof (*rule sum.reindex-cong*)
define *sc* **where** $sc = (\lambda(i::int).\ i + 1)$
show *inj-on sc* $\{k.\ L\ k \neq 0 \wedge i < k\}$ **using** *sc-def* **by** *simp*
have $\{l.\ L'\ l \neq 0 \wedge i+1 < l\} = (\lambda i.\ i + 1)\ `\ \{k.\ L\ k \neq 0 \wedge i < k\}$
proof (*rule refine-nonzero-liq*)
show $lower\text{-}tick\ P\ sqp \leq i$
using *i-def assms*(1,4,7) *lower-tick-mono* **by** *auto*
qed (*simp add*: *assms L-def L'-def*)+
thus $\{i.\ L'\ i \neq 0 \wedge j < i\} = (\lambda i.\ i + 1)\ `\ \{k.\ L\ k \neq 0 \wedge i < k\}$
using ‹$j = i+1$› **by** *simp*
fix x
assume *asx*: $x \in \{k.\ L\ k \neq 0 \wedge i < k\}$
hence $L'\ (x + 1) = L\ x$ **using** ‹*lower-tick* $P\ sqp \leq i$›
by (*simp add*: *L'-def L-def assms*(3) *assms*(6) *refine-lq-arg-gt*)
moreover have $grd\ P'\ (x + 1) = grd\ P\ x$
using ‹*lower-tick* $P\ sqp \leq i$› *asx assms*(3,6) *refine-grd-arg-gt* **by** *auto*
moreover have $grd\ P'\ (x + 1 + 1) = grd\ P\ (x + 1)$

```
    proof −
      have i < x + 1 using asx by simp
      thus ?thesis using ‹lower-tick P sqp ≤ i›
        by (metis assms(3) assms(6) order.strict-trans refine-grd-arg-gt
            zle-add1-eq-le zless-add1-eq)
    qed
    ultimately show L' (x + 1) *
       (inverse (grd P' (x + 1)) − inverse (grd P' (x + 1 + 1))) =
       L x * (inverse (grd P x) − inverse (grd P (x + 1)))
      by simp
  qed
  thus ?thesis by simp
 qed
 also have ... = base-net P sqp'
  using base-net-sum i-def L-def assms ‹clmm-dsc P'› by auto
 finally show ?thesis .
qed

lemma refine-base-net-eq:
 assumes clmm-dsc P
 and nz-support (lq P) ≠ {}
 and P' = refine P sqp
 and 0 < sqp
 and sqp < grd-max P
 and sqp ≤ sqp'
shows base-net P' sqp' = base-net P sqp'
proof (cases grd P (lower-tick P sqp) = sqp)
 case True
 then show ?thesis by (simp add: assms(3) refine-eq)
next
 case False
 then show ?thesis using assms refine-pool-base-net-grd-eq by simp
qed
```

6.2 CLMM pool restriction and slice

The restriction operation intuitively consists in deleting all the liquidity potentially available below the index provided as an argument.

```
definition restrict-pool where
restrict-pool i P =
 (grd P,
  (λj. if j < i then 0 else lq P j),
  (λj.  fee P j))

lemma restrict-pool-grd[simp]:
 shows grd (restrict-pool i P) = grd P
 unfolding restrict-pool-def grd-def by simp

lemma restrict-pool-lower-tick:
```

assumes $P' = \textit{restrict-pool}\ i\ P$
shows $\textit{lower-tick}\ P\ \textit{sqp} = \textit{lower-tick}\ P'\ \textit{sqp}$
using *assms* **unfolding** *lower-tick-def rng-blw-def* **by** *simp*

lemma *restrict-pool-lt*:
assumes $j < i$
shows $\textit{lq}\ (\textit{restrict-pool}\ i\ P)\ j = 0\ \textit{fee}\ (\textit{restrict-pool}\ i\ P)\ j = \textit{fee}\ P\ j$
using *assms* **unfolding** *restrict-pool-def lq-def fee-def* **by** *auto*

lemma *restrict-pool-ge*:
assumes $i \leq j$
shows $\textit{lq}\ (\textit{restrict-pool}\ i\ P)\ j = \textit{lq}\ P\ j$
$\textit{fee}\ (\textit{restrict-pool}\ i\ P)\ j = \textit{fee}\ P\ j$
using *assms* **unfolding** *restrict-pool-def lq-def fee-def* **by** *auto*

lemma *restrict-pool-lq-sub*:
assumes $P' = \textit{restrict-pool}\ i\ P$
shows $\textit{nz-support}\ (\textit{lq}\ P') \subseteq \textit{nz-support}\ (\textit{lq}\ P)$
proof
fix j
assume $j \in \textit{nz-support}\ (\textit{lq}\ P')$
hence $i \leq j$
using *restrict-pool-lt assms linorder-le-less-linear*
unfolding *nz-support-def* **by** *blast*
hence $\textit{lq}\ P\ j \neq 0$
by (*metis* ‹$j \in \textit{nz-support}\ (\textit{lq}\ P')$› *assms nz-supportD restrict-pool-ge*(*1*))
thus $j \in \textit{nz-support}\ (\textit{lq}\ P)$ **unfolding** *nz-support-def* **by** *auto*
qed

lemma *restrict-pool-finite-liq*:
assumes *finite-liq P*
and $P' = \textit{restrict-pool}\ i\ P$
shows *finite-liq P'* **using** *restrict-pool-lq-sub assms* **unfolding** *finite-liq-def*
by (*metis rev-finite-subset*)

lemma *restrict-pool-nz-liq*:
assumes *finite-liq P*
and $P' = \textit{restrict-pool}\ i\ P$
and $i \leq \textit{idx-max}\ (\textit{lq}\ P)$
and $\textit{nz-support}\ (\textit{lq}\ P) \neq \{\}$
shows $\textit{nz-support}\ (\textit{lq}\ P') \neq \{\}$
proof −
have $\textit{lq}\ P'\ (\textit{idx-max}\ (\textit{lq}\ P)) \neq 0$
by (*simp add*: *assms finite-liq-pool.idx-max-mem finite-liq-pool-def*
nz-supportD restrict-pool-ge(*1*))
thus *?thesis* **unfolding** *nz-support-def* **by** *auto*
qed

lemma *restrict-pool-idx-max*:

assumes *finite-liq P*
and $P' = $ *restrict-pool i P*
and $i \leq$ *idx-max* (*lq P*)
and *nz-support* (*lq P*) $\neq \{\}$
shows *idx-max* (*lq P*) = *idx-max* (*lq P'*)
proof (*rule idx-max-finiteI*)
show *finite* (*nz-support* (*lq P'*))
using *assms finite-liq-def restrict-pool-finite-liq* **by** *simp*
show *lq P'* (*idx-max* (*lq P*)) $\neq$ *0*
by (*simp add*: *assms finite-liq-pool.idx-max-mem finite-liq-pool-def*
nz-supportD restrict-pool-ge(*1*))
fix *j*
assume *idx-max* (*lq P*) < *j*
hence *lq P j* = *0*
using *assms*(*1*) *assms*(*4*) *finite-liq-def idx-max-finite-gt* **by** *blast*
thus *lq P' j* = *0*
using ‹*idx-max* (*lq P*) < *j*› *assms*(*2*) *assms*(*3*) *restrict-pool-ge*(*1*) **by** *auto*
qed

lemma *restrict-pool-clmm*:
assumes *clmm-dsc P*
and $P' = $ *restrict-pool i P*
shows *clmm-dsc P'*
proof
show *span-grid P'* **using** *assms restrict-pool-grd span-grid-eq* **by** *auto*
show *finite-liq P'*
using *assms restrict-pool-finite-liq clmm-dsc-liq* **by** *simp*
show $\forall$ *i*. *0* $\leq$ *lq P' i*
by (*metis assms clmm-dsc-liq*(*2*) *not-le-imp-less order-refl*
restrict-pool-ge(*1*) *restrict-pool-lt*(*1*))
show $\forall$ *i*. *0* $\leq$ *fee P' i*
by (*metis assms clmm-dsc-def leI restrict-pool-ge*(*2*)
restrict-pool-lt(*2*))
show $\forall$ *i*. *fee P' i* < *1*
by (*metis assms clmm-dsc-fees leI restrict-pool-ge*(*2*) *restrict-pool-lt*(*2*))
qed

lemma *restrict-pool-quote-gross-leq*:
assumes *mono* (*grd P*)
and *sqp* $\leq$ *grd P i*
and $P' = $ *restrict-pool i P*
shows *quote-gross P' sqp* = *0* **unfolding** *quote-gross-def*
proof (*rule gen-quote-zero*)
show *mono* (*grd P'*) **using** *assms restrict-pool-grd* **by** *simp*
fix *j*
assume *grd P' j* < *sqp*
hence *grd P j* < *sqp* **using** *restrict-pool-grd assms* **by** *simp*
hence *grd P j* < *grd P i* **using** *assms* **by** *simp*
hence *j* < *i* **using** *assms mono-strict-invE* **by** *auto*

hence *lq P′ j = 0* **by** (*simp add*: *assms restrict-pool-lt*)
thus *gross-fct* (*lq P′*) (*fee P′*) *j = 0* **unfolding** *gross-fct-def* **by** *simp*
qed

lemma *restrict-pool-quote-gross*:
assumes *clmm-dsc P*
and *P′ = restrict-pool j P*
and *0 < sqp*
and *j ≤ lower-tick P sqp*
shows *quote-gross P sqp − quote-gross P* (*grd P j*) *= quote-gross P′ sqp*
proof −
define *L* **where** *L = gross-fct* (*lq P*) (*fee P*)
define *L′* **where** *L′ = gross-fct* (*lq P′*) (*fee P′*)
define *k* **where** *k = lower-tick P sqp*
have *clmm-dsc P′* **using** *restrict-pool-clmm assms* **by** *simp*
have *eq*: ∀ *k ≥ j*. *L′ k = L k*
using *restrict-pool-ge gross-fct-cong L′-def L-def assms*(*2*) **by** *blast*
have *grd P k ≤ sqp* **using** *lower-tick-geq assms* **unfolding** *k-def* **by** *simp*
have *j = lower-tick P* (*grd P j*)
by (*simp add*: *assms*(*1*) *lower-tick-eq*)
hence *j = lower-tick P′* (*grd P j*)
using *restrict-pool-lower-tick*[*of P′*] *assms* **by** *simp*
have *k = lower-tick P′ sqp*
using *k-def assms restrict-pool-lower-tick* **by** *blast*
show *?thesis*
proof (*cases j = k*)
case *True*
have *quote-gross P sqp − quote-gross P* (*grd P j*) *= L j ∗* (*sqp − grd P j*)
using *clmm-quote-gross-diff-eq′*[*of P L j*]
by (*metis L-def True* ‹*grd P k ≤ sqp*› *assms*(*1*) *clmm-dsc-grid*(*2*) *k-def lower-tick-eq*)
also have *... = L′ j ∗* (*sqp − grd P j*) **using** *eq* **by** *simp*
also have *... = quote-gross P′ sqp − quote-gross P′* (*grd P j*)
proof (*rule clmm-quote-gross-diff-eq′*[*symmetric*])
show *clmm-dsc P′* **using** ‹*clmm-dsc P′*› **.**
show *L′ = gross-fct* (*lq P′*) (*fee P′*) **using** *L′-def* **by** *simp*
show *j = lower-tick P′ sqp*
using *True restrict-pool-lower-tick*[*of P′*] *assms k-def* **by** *simp*
show *j = lower-tick P′* (*grd P j*) **using** ‹*j = lower-tick P′* (*grd P j*)› **.**
show *0 < grd P j* **using** *assms* **by** *simp*
show *grd P j ≤ sqp* **using** *True* ‹*grd P k ≤ sqp*› *k-def* **by** *auto*
qed
also have *... = quote-gross P′ sqp*
using *assms restrict-pool-quote-gross-leq*
by (*simp add*: *strict-mono-mono*)
finally show *?thesis* **.**
next
case *False*
define *M* **where** *M* = {*l*. *L l ≠ 0 ∧ j <l ∧ l < k*}

```
  define M' where M' = {l. L' l ≠ 0 ∧ j <l ∧ l < k}
  have M = M' using eq unfolding M-def M'-def by auto
  have quote-gross P sqp − quote-gross P (grd P j) = L k ∗ (sqp − grd P k) +
      sum (λ l. L l ∗ (grd P (l+1) − grd P l)) M +
      L j ∗ (grd P (j+1) − grd P j) unfolding M-def
  proof (rule clmm-quote-gross-diff-eq)
    show j < k using assms k-def False by simp
    show L = gross-fct (lq P) (fee P) using L-def by simp
    show j = lower-tick P (grd P j) using assms lower-tick-eq by simp
    show clmm-dsc P using assms by simp
    show k = lower-tick P sqp using k-def by simp
    show 0 < grd P j using assms by simp
    show grd P j ≤ sqp
      using ‹grd P k ≤ sqp› ‹j < k› assms(1) clmm-dsc-grd-smono by fastforce
  qed
  also have ... = L' k ∗ (sqp − grd P' k) +
      (∑ l∈M'. L' l ∗ (grd P' (l + 1) − grd P' l)) +
      L' j ∗ (grd P' (j + 1) − grd P' j)
  proof −
    have L' k = L k using eq assms k-def by simp
    moreover have L j = L' j using eq by simp
    moreover have (∑ k∈M. L' k ∗ (grd P' (k + 1) − grd P' k)) =
        (∑ k∈M. L k ∗ (grd P (k + 1) − grd P k))
      using eq sum.cong M-def assms by simp
    ultimately show ?thesis using assms ‹M = M'› by simp
  qed
  also have ... = quote-gross P' sqp − quote-gross P' (grd P' j)
    unfolding M'-def
  proof (rule clmm-quote-gross-diff-eq[symmetric])
    show clmm-dsc P' using ‹clmm-dsc P'› .
    show L' = gross-fct (lq P') (fee P') using L'-def by simp
    show j = lower-tick P' (grd P' j)
      using ‹j = lower-tick P (grd P j)›
      by (simp add: ‹clmm-dsc P'› lower-tick-eq)
    show k = lower-tick P' sqp using ‹k = lower-tick P' sqp› .
    show 0 < grd P' j by (simp add: ‹clmm-dsc P'›)
    show grd P' j ≤ sqp
      using ‹j = lower-tick P (grd P j)› assms lower-tick-lt' by fastforce
    show j < k using assms False k-def by simp
  qed
  also have ... = quote-gross P' sqp
  proof −
    have quote-gross P' (grd P' j) = 0
      using assms restrict-pool-quote-gross-leq
      by (simp add: strict-mono-mono)
    thus ?thesis by simp
  qed
  finally show ?thesis .
qed
```

qed

lemma *restrict-pool-base-net-eq*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *P′* = *restrict-pool i P*
and *i* ≤ *idx-max* (*lq P*)
and *grd P i* ≤ *sqp′*
shows *base-net P′ sqp′* = *base-net P sqp′*
proof −
have *clmm-dsc P′* **using** *assms restrict-pool-clmm* **by** *simp*
have *0* < *grd P i* **using** *assms* **by** *simp*
hence *0* < *sqp′* **using** *assms* **by** *linarith*
define *L* **where** *L* = *lq P*
define *L′* **where** *L′* = *lq P′*
define *j* **where** *j* = *lower-tick P′ sqp′*
have *base-net P′ sqp′* = *L′ j* ∗ (*inverse sqp′* − *inverse* (*grd P′* (*j* + *1*))) +
(∑ *i* | *L′ i* ≠ *0* ∧ *j* < *i*.
L′ i ∗ (*inverse* (*grd P′ i*) − *inverse* (*grd P′* (*i* + *1*))))
using *base-net-sum j-def L′-def assms* ‹*clmm-dsc P′*› ‹*0* < *sqp′*› **by** *auto*
also have ... = *L j* ∗ (*inverse sqp′* − *inverse* (*grd P* (*j* + *1*))) +
(∑ *i* | *L i* ≠ *0* ∧ *j* < *i*.
L i ∗ (*inverse* (*grd P i*) − *inverse* (*grd P* (*i* + *1*))))
proof −
have *grd*: ⋀*k*. *i* < *k* ⟹ *grd P k* = *grd P′ k* **using** *assms* **by** *simp*
have *lq*: ⋀*k*. *j* ≤ *k* ⟹ *L k* = *L′ k*
by (*metis L′-def L-def* ‹*clmm-dsc P′*› *assms*(*3*) *assms*(*5*) *clmm-dsc-grid*(*2*)
j-def lower-tick-eq lower-tick-lt order.trans restrict-pool-ge(*1*)
restrict-pool-grd verit-comp-simplify1(*3*))
hence *L′ j* ∗ (*inverse sqp′* − *inverse* (*grd P′* (*j* + *1*))) =
L j ∗ (*inverse sqp′* − *inverse* (*grd P* (*j* + *1*)))
using *grd assms*(*3*) **by** *simp*
moreover have (∑ *i* | *L′ i* ≠ *0* ∧ *j* < *i*.
L′ i ∗ (*inverse* (*grd P′ i*) − *inverse* (*grd P′* (*i* + *1*)))) =
(∑ *i* | *L i* ≠ *0* ∧ *j* < *i*.
L i ∗ (*inverse* (*grd P i*) − *inverse* (*grd P* (*i* + *1*))))
proof (*rule sum.cong*)
show {*i*. *L′ i* ≠ *0* ∧ *j* < *i*} = {*i*. *L i* ≠ *0* ∧ *j* < *i*}
using *lq* **by** *auto*
fix *k*
assume *k* ∈ {*i*. *L i* ≠ *0* ∧ *j* < *i*}
thus *L′ k* ∗ (*inverse* (*grd P′ k*) − *inverse* (*grd P′* (*k* + *1*))) =
L k ∗ (*inverse* (*grd P k*) − *inverse* (*grd P* (*k* + *1*)))
using *lq grd assms*(*3*) **by** *simp*
qed
ultimately show *?thesis* **by** *simp*
qed
also have ... = *base-net P sqp′*
using *base-net-sum L-def assms j-def* ‹*0* < *sqp′*› *restrict-pool-lower-tick*

```
    by presburger
  finally show ?thesis .
qed

lemma restrict-pool-grd-min-le:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and P' = restrict-pool i P
  and i ≤ idx-max (lq P)
shows i ≤ idx-min (lq P')
  by (metis assms clmm-dsc-def finite-liq-def finite-subset idx-min-finite-in
      leI restrict-pool-lq-sub restrict-pool-lt(1) restrict-pool-nz-liq)

definition slice-pool where
slice-pool P sqp = (let P' = refine P sqp in restrict-pool (lower-tick P' sqp) P')

lemma slice-poolD:
  assumes P'' = refine P sqp
shows slice-pool P sqp = restrict-pool (lower-tick P'' sqp) P''
  using assms unfolding slice-pool-def Let-def by simp

lemma slice-pool-clmm-dsc:
  assumes clmm-dsc P
  and 0 < sqp
  and P' = slice-pool P sqp
shows clmm-dsc P'
proof −
  have clmm-dsc (restrict-pool (lower-tick (refine P sqp) sqp) (refine P sqp))
  proof (rule restrict-pool-clmm)
    show clmm-dsc (refine P sqp)
      by (rule refine-clmm, (auto simp add: assms)+)
  qed simp
  thus ?thesis using assms unfolding slice-pool-def Let-def by simp
qed

lemma slice-pool-nz-liq:
  assumes clmm-dsc P
  and 0 < sqp
  and P' = slice-pool P sqp
  and lower-tick P sqp ≤ idx-max (lq P)
  and nz-support (lq P) ≠ {}
shows nz-support (lq P') ≠ {}
proof (rule restrict-pool-nz-liq)
  define Pr where Pr = refine P sqp
  define i where i = lower-tick Pr sqp
  show P' = restrict-pool i Pr
    using slice-poolD assms Pr-def i-def by simp
  show finite-liq Pr using Pr-def
    by (meson assms(1) clmm-dsc-def refine-finite-liq)
```

```
  show nz-support (lq (refine P sqp)) ≠ {}
    using restrict-pool-nz-liq
    by (meson assms(1) assms(5) refine-nz-lq-ne)
  show i ≤ idx-max (lq Pr) using i-def Pr-def refine-lower-tick-idx-max
    by (simp add: assms(1,2,4,5))
qed

lemma slice-pool-tick-idx-max:
  assumes clmm-dsc P
  and 0 < sqp
  and P' = slice-pool P sqp
  and lower-tick P sqp ≤ idx-max (lq P)
  and nz-support (lq P) ≠ {}
shows lower-tick P' sqp ≤ idx-max (lq P')
proof −
  define Pr where Pr = refine P sqp
  define i where i = lower-tick Pr sqp
  have lower-tick P' sqp = i
    using assms restrict-pool-lower-tick Pr-def i-def
    unfolding slice-pool-def Let-def
    by presburger
  also have ... ≤ idx-max (lq Pr) using i-def Pr-def refine-lower-tick-idx-max
    by (simp add: assms(1,2,4,5))
  also have ... = idx-max (lq P')
   using Pr-def assms(1−5) clmm-dsc-liq(1) refine-clmm refine-lower-tick-idx-max
     refine-nz-lq-ne restrict-pool-idx-max slice-poolD
    by presburger
  finally show ?thesis .
qed

lemma slice-pool-nz-liq':
  assumes clmm-dsc P
  and P' = slice-pool P sqp
  and nz-support (lq P) ≠ {}
  and 0 < sqp
  and sqp < grd-max P
shows nz-support (lq P') ≠ {}
proof −
  have lower-tick P sqp < idx-max (lq P) + 1
  proof (rule lower-tick-lt')
    show clmm-dsc P using assms by simp
    show 0 < sqp using assms by simp
    show idx-max (lq P) + 1 = lower-tick P (grd-max P)
      by (simp add: assms(1) idx-max-img-def lower-tick-eq grd-max-def)
    show sqp < grd-max P using assms by simp
    show grd P (idx-max (lq P) + 1) = grd-max P
      unfolding grd-max-def idx-max-img-def by simp
  qed simp
  thus ?thesis using slice-pool-nz-liq by (simp add: assms)
```

qed

lemma *slice-pool-idx-min*:
assumes *clmm-dsc P*
and *0 < sqp*
and *nz-support (lq P) ≠ {}*
and *P′ = slice-pool P sqp*
and *i = lower-tick P sqp*
and *i ≤ idx-max (lq P)*
shows *i ≤ idx-min (lq P′)*
proof (*rule idx-min-finite-max*)
show *nz-support (lq P′) ≠ {}* **using** *assms slice-pool-nz-liq* **by** *simp*
show *finite (nz-support (lq P′))*
using *assms clmm-dsc-liq(1) finite-liq-def slice-pool-clmm-dsc* **by** *simp*
fix *j*
assume *j < i*
thus *lq P′ j = 0*
by (*metis assms(1,2,4,5) lower-tick-eq not-le-imp-less order.strict-trans2 order-less-imp-not-less refine-grd-arg-le refine-lower-tick restrict-pool-lt(1) slice-poolD*)
qed

lemma *slice-pool-grd-min*:
assumes *clmm-dsc P*
and *0 < sqp*
and *nz-support (lq P) ≠ {}*
and *P′ = slice-pool P sqp*
and *sqp < grd-max P*
shows *sqp ≤ grd-min P′*
proof −
define *Pr* **where** *Pr = refine P sqp*
define *i* **where** *i = lower-tick Pr sqp*
have *P′ = restrict-pool i Pr* **using** *i-def Pr-def*
by (*simp add*: *assms(4) slice-poolD*)
hence *i ≤ idx-min (lq P′)*
using *restrict-pool-grd-min-le Pr-def assms(1−3,5) i-def refine-clmm refine-lower-tick-idx-max refine-nz-lq-ne sqp-lt-grd-max-imp-idx*
by *presburger*
moreover have *grd P′ i = sqp*
using *Pr-def ‹P′ = restrict-pool i Pr› assms(1,2) i-def refine-lower-tick*
by *auto*
ultimately show *?thesis* **using** *grd-min-def idx-min-img-def*
by (*metis Pr-def ‹P′ = restrict-pool i Pr› assms(1,2) clmm-dsc-grd-mono refine-clmm restrict-pool-clmm*)
qed

lemma *slice-pool-grd-max*:
assumes *clmm-dsc P*
and *0 < sqp*

and *nz-support* (*lq P*) ≠ {}
and *P′* = *slice-pool P sqp*
and *lower-tick P sqp* ≤ *idx-max* (*lq P*)
shows *grd-max P* = *grd-max P′* **using** *assms slice-pool-tick-idx-max*
proof −
define *Pr* **where** *Pr* = *refine P sqp*
have *grd-max P* = *grd-max Pr* **using** *assms refine-grd-max Pr-def* **by** *simp*
also have ... = *grd-max P′*
using *restrict-pool-idx-max Pr-def assms*(1−5) *clmm-dsc-liq*(1)
refine-finite-liq refine-lower-tick-idx-max refine-nz-lq-ne
slice-poolD
unfolding *grd-max-def idx-max-img-def*
by *auto*
finally show *?thesis* **.**
qed

lemma *slice-pool-grd-max′*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *P′* = *slice-pool P sqp*
and *0* < *sqp*
and *sqp* < *grd-max P*
shows *grd-max P* = *grd-max P′*
proof −
define *Pr* **where** *Pr* = *refine P sqp*
have *grd-max P* = *grd-max Pr* **using** *assms refine-grd-max Pr-def* **by** *simp*
also have ... = *grd-max P′*
using *restrict-pool-idx-max Pr-def assms*(1−5)
clmm-dsc-liq(1) *refine-finite-liq refine-lower-tick-idx-max refine-nz-lq-ne*
slice-poolD restrict-pool-grd sqp-lt-grd-max-imp-idx
unfolding *grd-max-def idx-max-img-def* **by** *auto*
finally show *?thesis* **.**
qed

lemma *slice-pool-cst-fees*:
assumes *clmm-dsc P*
and *P′* = *slice-pool P sqp*
and ⋀*i*. *fee P i* = *phi*
shows ⋀*i*. *fee P′ i* = *phi*
by (*metis assms*(2,3) *refine-cst-fees restrict-pool-ge*(2) *restrict-pool-lt*(2)
slice-poolD verit-comp-simplify1(3))

lemma *slice-pool-quote-gross-leq*:
assumes *clmm-dsc P*
and *0* < *sqp*
and *sqp′* ≤ *sqp*
and *P′* = *slice-pool P sqp*
shows *quote-gross P′ sqp′* = *0*
proof (*rule restrict-pool-quote-gross-leq*)

define *Pr* **where** *Pr = refine P sqp*
define *i* **where** *i = lower-tick Pr sqp*
show *P' = restrict-pool i Pr* **using** *i-def Pr-def*
by (*simp add*: *assms*(*4*) *slice-poolD*)
show *mono* (*grd Pr*) **using** *Pr-def assms refine-clmm*
by (*simp add*: *clmm-dsc-grd-mono monoI*)
show *sqp' ≤ grd Pr i*
using *Pr-def assms*(*1−3*) *i-def refine-lower-tick* **by** *auto*
qed

lemma *slice-pool-quote-gross*:
assumes *clmm-dsc P*
and *0 < sqp*
and *sqp ≤ sqp'*
and *P' = slice-pool P sqp*
shows *quote-gross P' sqp' = quote-gross P sqp' − quote-gross P sqp*
proof −
define *Pr* **where** *Pr = refine P sqp*
define *i* **where** *i = lower-tick Pr sqp*
have *P' = restrict-pool i Pr* **using** *i-def Pr-def*
by (*simp add*: *assms*(*4*) *slice-poolD*)
have *quote-gross P' sqp' = quote-gross Pr sqp' − quote-gross Pr* (*grd Pr i*)
proof (*rule restrict-pool-quote-gross*[*symmetric*])
show *clmm-dsc Pr* **using** *Pr-def assms*(*1*,*2*) *refine-clmm* **by** *auto*
show *P' = restrict-pool i Pr* **using** ‹*P' = restrict-pool i Pr*› **.**
show *0 < sqp'* **using** *assms* **by** *simp*
show *i ≤ lower-tick Pr sqp'*
using *i-def* ‹*clmm-dsc Pr*› *assms*(*2*,*3*) *lower-tick-mono* **by** *auto*
qed
also have *... = quote-gross Pr sqp' − quote-gross Pr sqp*
proof −
have *quote-gross Pr* (*grd Pr i*) *= quote-gross Pr sqp*
using *Pr-def assms*(*1*,*2*) *i-def refine-lower-tick* **by** *auto*
thus *?thesis* **by** *simp*
qed
also have *... = quote-gross P sqp' − quote-gross P sqp*
using *Pr-def assms*(*1*,*2*) *refine-quote-gross* **by** *auto*
finally show *?thesis* **.**
qed

lemma *slice-pool-quote-gross-max-eq*:
assumes *clmm-dsc P*
and *P' = slice-pool P sqp*
and *nz-support* (*lq P*) ≠ {}
and *0 < sqp*
and *sqp < grd-max P*
and *i = lower-tick P sqp*
and *grd P i = sqp*
shows *quote-gross P'* (*grd-max P'*) *= quote-gross P* (*grd-max P*) *− quote-gross P*

sqp
proof −
have *grd-max P* = *grd-max P′*
by (*simp add*: *assms slice-pool-grd-max′*)
define *sqp′* **where** *sqp′* = *grd-max P*
have *quote-gross P′ sqp′* = *quote-gross P sqp′* − *quote-gross P sqp*
using *slice-pool-quote-gross assms sqp′-def* **by** *simp*
thus *?thesis* **using** ‹*grd-max P* = *grd-max P′*› *sqp′-def* **by** *simp*
qed

lemma *slice-pool-quote-gross-inv*:
assumes *clmm-dsc P*
and *0* < *sqp*
and *nz-support* (*lq P*) ≠ {}
and *sqp* < *grd-max P*
and *0* < *y*
and *P′* = *slice-pool P sqp*
shows *quote-gross P′* −‘{*y*} = *quote-gross P* −‘{*y* + *quote-gross P sqp*}
proof
have *clmm-dsc P′* **using** *assms slice-pool-clmm-dsc* **by** *simp*
have *nz-support* (*lq P′*) ≠ {} **using** *assms slice-pool-nz-liq′* **by** *simp*
show *quote-gross P′* −‘ {*y*} ⊆ *quote-gross P* −‘ {*y* + *quote-gross P sqp*}
proof
fix *sqp′*
assume *asm*: *sqp′* ∈ *quote-gross P′* −‘ {*y*}
hence *y* = *quote-gross P′ sqp′* **by** *simp*
also have ... = *quote-gross P sqp′* − *quote-gross P sqp*
by (*metis assms*(*1*,*2*,*5*,*6*) *calculation dual-order.irrefl nle-le*
slice-pool-quote-gross slice-pool-quote-gross-leq)
finally have *y* = *quote-gross P sqp′* − *quote-gross P sqp* **.**
hence *quote-gross P sqp′* = *y* + *quote-gross P sqp* **by** *simp*
thus *sqp′* ∈ *quote-gross P* −‘ {*y* + *quote-gross P sqp*} **by** *simp*
qed
show *quote-gross P* −‘ {*y* + *quote-gross P sqp*} ⊆ *quote-gross P′* −‘ {*y*}
proof
fix *sqp′*
assume *asm*: *sqp′* ∈ *quote-gross P* −‘ {*y* + *quote-gross P sqp*}
hence *eq*: *quote-gross P sqp′* = *y* + *quote-gross P sqp* **by** *simp*
hence *sqp* ≤ *sqp′*
by (*metis assms*(*1*) *assms*(*5*) *less-add-same-cancel2 order-less-imp-le*
quote-gross-imp-sqp-lt)
have *y* = *quote-gross P sqp′* − *quote-gross P sqp* **using** *eq assms* **by** *simp*
also have ... = *quote-gross P′ sqp′*
proof (*rule slice-pool-quote-gross*[*symmetric*, *of P*], *auto simp add*: *assms*)
show *sqp* ≤ *sqp′* **using** ‹*sqp* ≤ *sqp′*› **.**
qed
finally have *y* = *quote-gross P′ sqp′* **.**
thus *sqp′* ∈ *quote-gross P′* −‘ {*y*} **by** *simp*
qed

qed

lemma *slice-pool-quote-reach*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *0* < *sqp*
and *sqp* < *grd-max P*
and *0* < *y*
and *P′* = *slice-pool P sqp*
shows *quote-reach P′ y* = *quote-reach P* (*y* + *quote-gross P sqp*)
proof −
have *quote-reach P′ y* = *Inf* (*quote-gross P′* −‘ {*y*})
using *assms clmm-quote-gross-grd-min slice-pool-clmm-dsc slice-pool-nz-liq′*
unfolding *quote-reach-def* **by** *auto*
also have ... = *Inf* (*quote-gross P* −‘ {*y* + *quote-gross P sqp*})
using *assms slice-pool-quote-gross-inv* **by** *simp*
also have ... = *quote-reach P* (*y* + *quote-gross P sqp*)
using *assms* **unfolding** *quote-reach-def*
by (*metis add-pos-nonneg clmm-quote-gross-pos order-less-irrefl*)
finally show *?thesis* **.**
qed

lemma *slice-pool-base-net-eq*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *P′* = *slice-pool P sqp*
and *0* < *sqp*
and *sqp* < *grd-max P*
and *sqp* ≤ *sqp′*
shows *base-net P′ sqp′* = *base-net P sqp′*
proof −
define *Pr* **where** *Pr* = *refine P sqp*
define *i* **where** *i* = *lower-tick Pr sqp*
hence *i* ≤ *idx-max* (*lq Pr*)
using *Pr-def assms*(*1*,*2*,*4*,*5*) *refine-lower-tick-idx-max sqp-lt-grd-max-imp-idx*
by *presburger*
have *P′* = *restrict-pool i Pr* **using** *i-def Pr-def*
by (*simp add*: *assms*(*3*) *slice-poolD*)
hence *base-net P′ sqp′* = *base-net Pr sqp′*
using *restrict-pool-base-net-eq assms Pr-def* ‹*i* ≤ *idx-max* (*lq Pr*)›
by (*metis i-def refine-clmm refine-lower-tick refine-nz-lq-ne*)
also have ... = *base-net P sqp′* **using** *Pr-def assms refine-base-net-eq*
by *simp*
finally show *?thesis* **.**
qed

lemma *slice-pool-base-net-slice*:
assumes *clmm-dsc P*

and *nz-support* (*lq P*) ≠ {}
and *i* = *lower-tick P sqp*
and *P′* = *slice-pool P sqp*
and *sqp* < *grd-max P*
and *grd P i* = *sqp*
and *sqp′* ≤ *sqp*
and *0* < *sqp′*
shows *base-net P′ sqp′* = *base-net P′ sqp*
proof −
have *clmm-dsc P′* **using** *assms slice-pool-clmm-dsc* **by** *simp*
have *lower-tick P sqp* ≤ *idx-max* (*lq P*)
by (*metis assms*(*1*) *assms*(*2*) *assms*(*5*) *assms*(*6*) *clmm-dsc-grid*(*2*)
sqp-lt-grd-max-imp-idx)
hence *sqp* ≤ *grd-min P′* **using** *assms slice-pool-grd-min* **by** *simp*
hence *sqp′* ≤ *grd-min P′* **using** *assms* **by** *simp*
have *base-net P′ sqp′* = *base-net P′* (*grd-min P′*)
using *base-net-grd-min-le* ‹*sqp′* ≤ *grd-min P′*› *assms* ‹*clmm-dsc P′*›
by *blast*
also have ... = *base-net P′ sqp*
using *base-net-grd-min-le* ‹*sqp* ≤ *grd-min P′*› *assms* ‹*clmm-dsc P′*›
by (*metis clmm-dsc-grid*(*2*))
finally show *?thesis* **.**
qed

lemma *slice-pool-quote-swap-gt-zero*:
assumes *clmm-dsc P*
and *nz-support* (*lq P*) ≠ {}
and *grd P* (*lower-tick P sqp2*) = *sqp2*
and *P′* = *slice-pool P sqp2*
and *sqp1* ≤ *sqp2*
and *0* < *y*
and *0* <*sqp1*
and *y* + *quote-gross P sqp2* ≤ *quote-gross P* (*grd-max P*)
shows *quote-swap P′ sqp1 y* = *quote-swap P sqp2 y*
proof −
have *clmm-dsc P′* **using** *slice-pool-clmm-dsc assms* **by** *simp*
have *sqp2* < *grd-max P* **using** *assms quote-gross-imp-sqp-lt* **by** *simp*
hence *quote-gross P′ sqp1* = *0*
using *assms slice-pool-quote-gross-leq* **by** (*simp add*: *strict-mono-mono*)
hence *qeq*: *quote-reach P′* (*y* + *quote-gross P′ sqp1*) =
quote-reach P (*y* + *quote-gross P sqp2*)
using *assms* ‹*sqp2* < *grd-max P*› *slice-pool-quote-reach* **by** *simp*
have *sqp2* ≤ *quote-reach P* (*y* + *quote-gross P sqp2*)
using *quote-reach-gt*[*of P y sqp2*] *assms* **by** *simp*
hence *a*: *base-net P′* (*quote-reach P′* (*y* + *quote-gross P′ sqp1*)) =
base-net P (*quote-reach P* (*y* + *quote-gross P sqp2*))
using *qeq* ‹*sqp2* < *grd-max P*› *assms slice-pool-base-net-eq* **by** *auto*
have *base-net P′ sqp1* = *base-net P′ sqp2* **using** *slice-pool-base-net-slice*
by (*simp add*: ‹*sqp2* < *grd-max P*› *assms*)

```
  also have ... = base-net P sqp2
    using slice-pool-base-net-eq ‹sqp2 < grd-max P› assms
    by auto
  finally have base-net P' sqp1 = base-net P sqp2 .
  thus ?thesis using a unfolding quote-swap-def by simp
qed

lemma slice-pool-quote-swap:
  assumes clmm-dsc P
  and nz-support (lq P) ≠ {}
  and grd P (lower-tick P sqp2) = sqp2
  and P' = slice-pool P sqp2
  and sqp1 ≤ sqp2
  and sqp2 < grd-max P
  and 0 ≤ y
  and 0 < sqp1
  and y + quote-gross P sqp2 ≤ quote-gross P (grd-max P)
shows quote-swap P' sqp1 y = quote-swap P sqp2 y
proof (cases y = 0)
  case True
  have quote-swap P' sqp1 0 = 0
  proof (rule quote-swap-zero)
    show clmm-dsc P' using assms slice-pool-clmm-dsc by simp
    show nz-support (lq P') ≠ {}
      by (metis assms(1–4) assms(6) clmm-dsc-grid(2)
          slice-pool-nz-liq')
    show 0 < sqp1 using assms by simp
    show sqp1 ≤ grd-max P' using assms slice-pool-grd-max' by simp
  qed
  also have ... = quote-swap P sqp2 0
    using quote-swap-zero assms by simp
  finally show ?thesis using True by simp
next
  case False
  then show ?thesis
    using assms slice-pool-quote-swap-gt-zero by simp
qed
```

6.3 CLMM pool join

The join operation is meant to define a pool P on which swap operations can be viewed as a combination of swap operations on its two arguments. We use the convention that the pool fee is 0 on ranges where there is no liquidity.

```
definition pool-fee-join where
pool-fee-join P1 P2 i = fee-union (lq P1 i) (lq P2 i) (fee P1 i) (fee P2 i)

lemma pool-fee-join-com:
```

shows *pool-fee-join P1 P2 i = pool-fee-join P2 P1 i*
unfolding *pool-fee-join-def fee-union-def*
by (*simp add*: *add.commute*)

definition *joint-pools* **where**
joint-pools P P1 P2 $\longleftrightarrow$ (*grd P*) = (*grd P1*) $\wedge$ (*grd P*) = (*grd P2*) $\wedge$
($\forall$ *i*. *lq P i = lq P1 i + lq P2 i*) $\wedge$
($\forall$ *i*. *fee P i = pool-fee-join P1 P2 i*)

definition *pool-join* **where**
pool-join P1 P2 =
(*grd P1*, (λ*i*. *lq P1 i + lq P2 i*), (λ*i*. *pool-fee-join P1 P2 i*))

lemma *joint-poolsI*[*intro*]:
assumes *grd P = grd P1*
and *grd P = grd P2*
and $\bigwedge$*i*. *lq P i = lq P1 i + lq P2 i*
and $\bigwedge$*i*. *fee P i = pool-fee-join P1 P2 i*
shows *joint-pools P P1 P2* **using** *assms* **unfolding** *joint-pools-def* **by** *simp*

lemma *pool-join-joint*:
assumes *grd P1 = grd P2*
and *P = pool-join P1 P2*
shows *joint-pools P P1 P2* **using** *assms* **unfolding** *pool-join-def*
by (*simp add*: *fee-def grd-def joint-pools-def lq-def*)

lemma *joint-pools-grids*:
assumes *joint-pools P P1 P2*
shows (*grd P*) = (*grd P1*) (*grd P*) = (*grd P2*)
using *assms* **unfolding** *joint-pools-def* **by** *simp+*

lemma *joint-pools-lq*:
assumes *joint-pools P P1 P2*
shows *lq P i = lq P1 i + lq P2 i*
using *assms* **unfolding** *joint-pools-def* **by** *simp*

lemma *joint-pools-fee*:
assumes *joint-pools P P1 P2*
shows *fee P i = pool-fee-join P1 P2 i*
using *assms* **unfolding** *joint-pools-def* **by** *simp*

lemma *joint-pools-com*:
assumes *joint-pools P P1 P2*
shows *joint-pools P P2 P1*
proof
show *grd P = grd P2* **using** *assms joint-pools-grids* **by** *simp*
show *grd P = grd P1* **using** *assms joint-pools-grids* **by** *simp*
fix *i*
show *lq P i = lq P2 i + lq P1 i* **using** *assms joint-pools-lq* **by** *simp*

```
  show fee P i = pool-fee-join P2 P1 i
    using pool-fee-join-com joint-pools-fee assms by simp
qed

lemma joint-pools-nz-liq-sub:
  assumes joint-pools P P1 P2
  shows nz-support (lq P) ⊆ nz-support (lq P1) ∪ (nz-support (lq P2))
  unfolding nz-support-def
proof −
  define F1 where F1 = {i. lq P1 i ≠ 0}
  define F2 where F2 = {i. lq P2 i ≠ 0}
  define F where F = {i. lq P i ≠ 0}
  show F ⊆ F1 ∪ F2
  proof
    fix i
    assume i∈ F
    hence lq P1 i + lq P2 i ≠ 0 using F-def assms joint-pools-lq by auto
    hence lq P1 i ≠ 0 ∨ lq P2 i ≠ 0 by simp
    thus i ∈ F1 ∪ F2 using F1-def F2-def by auto
  qed
qed

lemma joint-pools-nz-liq-sup:
  assumes joint-pools P P1 P2
  and ⋀i. 0 ≤ lq P1 i
  and ⋀i. 0 ≤ lq P2 i
  shows nz-support (lq P1) ∪ (nz-support (lq P2)) ⊆ nz-support (lq P)
  unfolding nz-support-def
proof −
  define F1 where F1 = {i. lq P1 i ≠ 0}
  define F2 where F2 = {i. lq P2 i ≠ 0}
  define F where F = {i. lq P i ≠ 0}
  show F1 ∪ F2 ⊆ F
  proof
    fix j
    assume j∈ F1∪ F2
    hence lq P1 j ≠ 0 ∨ lq P2 j ≠ 0 unfolding F1-def F2-def by auto
    hence lq P1 j + lq P2 j ≠ 0 using joint-pools-lq
      by (simp add: add-nonneg-eq-0-iff assms)
    thus j ∈ F using F-def joint-pools-lq assms by auto
  qed
qed

lemma joint-pools-nz-liq:
  assumes joint-pools P P1 P2
  and ⋀i. 0 ≤ lq P1 i
  and ⋀i. 0 ≤ lq P2 i
shows nz-support (lq P1) ∪ (nz-support (lq P2)) = nz-support (lq P)
  using assms joint-pools-nz-liq-sup joint-pools-nz-liq-sub by blast
```

lemma *clmm-joint-pools-nz-liq*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *joint-pools P P1 P2*
shows *nz-support* (*lq P1*) ∪ (*nz-support* (*lq P2*)) = *nz-support* (*lq P*)
using *assms joint-pools-nz-liq* **by** (*simp add*: *clmm-dsc-liq*(*2*))

lemma *joint-pools-finite-liq*:
assumes *finite-liq P1*
and *finite-liq P2*
and *joint-pools P P1 P2*
shows *finite-liq P* **using** *assms joint-pools-nz-liq-sub*
by (*meson finite-UnI finite-liq-def rev-finite-subset*)

lemma *joint-pools-idx-min-min*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *joint-pools P P1 P2*
and *nz-support* (*lq P1*) ≠ {}
and *idx-min* (*lq P1*) ≤ *idx-min* (*lq P2*)
shows *idx-min* (*lq P*) = *idx-min* (*lq P1*)
proof (*rule idx-min-finiteI*[*symmetric*])
define *i* **where** *i* = *idx-min* (*lq P1*)
show *finite* (*nz-support* (*lq P*))
using *assms joint-pools-finite-liq* **by** (*meson clmm-dsc-def finite-liq-def*)
have *lq P1 i* ≠ *0* **using** *i-def idx-min-finite-in*
by (*metis* (*full-types*) ‹*finite* (*nz-support* (*lq P*))› *assms*(*1*−*4*)
clmm-joint-pools-nz-liq finite-Un)
thus *lq P i* ≠ *0*
by (*smt* (*verit*) *assms*(*1*−*3*) *clmm-dsc-liq*(*2*) *joint-pools-lq*)
fix *j*
assume *j* < *i*
hence *j* < *idx-min* (*lq P2*) **using** *assms i-def* **by** *simp*
have *lq P2 j* = *0*
using *assms idx-min-finite-lt*[*of lq P2 j*] *clmm-dsc-liq finite-liq-def*
by (*simp add*: ‹*j* < *idx-min* (*lq P2*)›)
moreover have *lq P1 j* = *0*
using ‹*j* < *i*› *i-def idx-max-finite-gt*[*of lq P2 j*]
by (*simp add*: *assms idx-min-lt-liq*)
ultimately show *lq P j* = *0*
using *assms*(*3*) *joint-pools-lq* **by** *auto*
qed

lemma *joint-pools-idx-min*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *joint-pools P P1 P2*
and *nz-support* (*lq P1*) ≠ {}

and *nz-support* (*lq P2*) ≠ {}
shows *idx-min* (*lq P*) = *min* (*idx-min* (*lq P1*)) (*idx-min* (*lq P2*))
using *joint-pools-idx-min-min*
by (*smt* (*z3*) *assms max-def nle-le joint-pools-com*)

lemma *joint-pools-idx-max-max*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *joint-pools P P1 P2*
and *nz-support* (*lq P2*) ≠ {}
and *idx-max* (*lq P1*) ≤ *idx-max* (*lq P2*)
shows *idx-max* (*lq P*) = *idx-max* (*lq P2*)
proof (*rule idx-max-finiteI*[*symmetric*])
define *i* **where** *i* = *idx-max* (*lq P2*)
show *finite* (*nz-support* (*lq P*))
using *assms joint-pools-finite-liq* **by** (*meson clmm-dsc-def finite-liq-def*)
have *lq P2 i* ≠ *0* **using** *i-def idx-max-finite-in*
by (*metis* (*full-types*) ‹*finite* (*nz-support* (*lq P*))› *assms*(*1*−*4*)
clmm-joint-pools-nz-liq finite-Un)
thus *lq P i* ≠ *0*
by (*smt* (*verit*) *assms*(*1*−*3*) *clmm-dsc-liq*(*2*) *joint-pools-lq*)
fix *j*
assume *i* < *j*
hence *idx-max* (*lq P1*) < *j* **using** *assms i-def* **by** *simp*
have *lq P1 j* = *0*
using *assms idx-max-finite-gt*[*of lq P1 j*] *clmm-dsc-liq finite-liq-def*
by (*simp add*: ‹*idx-max* (*lq P1*) < *j*›)
moreover have *lq P2 j* = *0*
using ‹*i* < *j*› *i-def idx-max-finite-gt*[*of lq P2 j*]
by (*simp add*: *assms*(*2*) *idx-max-gt-liq*)
ultimately show *lq P j* = *0*
using *assms*(*3*) *joint-pools-lq* **by** *auto*
qed

lemma *joint-pools-idx-max*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *joint-pools P P1 P2*
and *nz-support* (*lq P1*) ≠ {}
and *nz-support* (*lq P2*) ≠ {}
shows *idx-max* (*lq P*) = *max* (*idx-max* (*lq P1*)) (*idx-max* (*lq P2*))
using *joint-pools-idx-max-max*
by (*smt* (*z3*) *assms max-def nle-le joint-pools-com*)

lemma *joint-pools-clmm-dsc*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *joint-pools P P1 P2*
shows *clmm-dsc P*

proof
show *span-grid P* **using** *assms clmm-dsc-grid*[*of P1*] *joint-pools-grids*(*1*)
by (*simp add*: *span-grid-def*)
show *finite-liq P* **using** *assms joint-pools-finite-liq clmm-dsc-liq* **by** *meson*
show ∀ *i*. *0* ≤ *lq P i* **using** *assms joint-pools-lq clmm-dsc-liq*(*2*) **by** *simp*
show ∀ *i*. *0* ≤ *fee P i* **using** *assms joint-pools-fee fee-union-pos*
by (*simp add*: *clmm-dsc-def pool-fee-join-def*)
show ∀ *i*. *fee P i* < *1* **using** *assms joint-pools-fee fee-union-lt-1*
by (*simp add*: *clmm-dsc-def pool-fee-join-def*)
qed

lemma *join-gross-fct*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *joint-pools P P1 P2*
shows *gross-fct* (*lq P*) (*fee P*) *i* = *gross-fct* (*lq P1*) (*fee P1*) *i* +
gross-fct (*lq P2*) (*fee P2*) *i*
proof (*cases lq P i* = *0*)
case *True*
hence *lq P1 i* + *lq P2 i* = *0*
using *assms*(*3*) *joint-pools-lq* **by** *auto*
hence *lq P1 i* = *0 lq P2 i* = *0*
by (*simp add*: *clmm-dsc-liq*(*2*) *add-nonneg-eq-0-iff assms*(*1*) *assms*(*2*))+
then show *?thesis* **using** *True gross-fct-zero-if* **by** *auto*
next
case *False*
define *L* **where** *L* = *lq P*
define *F* **where** *F* = *fee P*
define *l1* **where** *l1* = *lq P1 i*
define *f1* **where** *f1* = *fee P1 i*
define *l2* **where** *l2* = *lq P2 i*
define *f2* **where** *f2* = *fee P2 i*
define *df* **where** *df* = *l1*∗(*1*−*f2*) + *l2*∗(*1*−*f1*)
have *0* ≤ *l1* **using** *assms l1-def clmm-dsc-liq* **by** *simp*
have *0* < *1* − *f2* **using** *assms f2-def clmm-dsc-fees* **by** *simp*
have *0* < *1* − *f1* **using** *assms f1-def clmm-dsc-fees* **by** *simp*
have *0* ≤ *l2* **using** *assms l2-def clmm-dsc-liq* **by** *simp*
have *0* < *lq P i*
using *False* ‹*0* ≤ *l1*› ‹*0* ≤ *l2*› *assms*(*3*) *l1-def l2-def joint-pools-lq* **by** *auto*
hence *0* < *l1* ∨ *0* < *l2* **using** *assms joint-pools-lq l1-def l2-def* **by** *auto*
hence *0* < *df* **using** *df-def l1-def f2-def l2-def f1-def*
by (*smt* (*verit*, *best*) ‹*0* < *1* − *f1*› ‹*0* < *1* − *f2*› ‹*0* ≤ *l1*› ‹*0* ≤ *l2*›
mult-nonneg-nonneg mult-pos-pos)
have *gross-fct* (*lq P*) (*fee P*) *i* =
(*l1* + *l2*)/(*one-cpl* (*pool-fee-join P1 P2*) *i*)
using *assms joint-pools-lq joint-pools-fee*
unfolding *gross-fct-def l1-def l2-def*
by (*simp add*: *one-cpl-def*)
also have ... = (*l1* + *l2*)/(*1* − ((*l1*∗*f1*∗(*1*−*f2*) + *l2*∗*f2*∗(*1*−*f1*))/

```
    df))
  using one-cpl-def pool-fee-join-def fee-union-def l1-def l2-def f1-def f2-def df-def
  by simp
also have ... = (l1 + l2)/((df − (l1*f1*(1−f2) + l2*f2*(1−f1))) / df)
proof −
  have 1 − ((l1*f1*(1−f2) + l2*f2*(1−f1)))/ df =
      df/df − ((l1*f1*(1−f2) + l2*f2*(1−f1)))/ df
    using ‹0 < df› by simp
  also have ... = (df − (l1*f1*(1−f2) + l2*f2*(1−f1))) / df
    by (rule diff-divide-distrib[symmetric])
  finally have 1 − ((l1*f1*(1−f2) + l2*f2*(1−f1)))/ df =
      (df − (l1*f1*(1−f2) + l2*f2*(1−f1))) / df .
  thus ?thesis by simp
qed
also have ... = ((l1 + l2) * df)/ (df − (l1*f1*(1−f2) + l2*f2*(1−f1)))
  by (rule divide-divide-eq-right)
also have ... = ((l1 + l2) * df)/ ((l1 + l2) * ((1 − f1) * (1 − f2)))
proof −
  have df − (l1*f1*(1−f2) + l2*f2*(1−f1)) =
      l1*(1−f2) + l2*(1−f1) − l1*f1*(1−f2) − l2*f2*(1−f1)
    unfolding df-def by simp
  also have ... = l1*(1−f2) − l1*f1*(1−f2) + (l2*(1−f1) − l2*f2*(1−f1))
    by simp
  also have ... = (l1 − l1 * f1) * (1 − f2) + (l2*(1−f1) − l2*f2*(1−f1))
    by (simp add: left-diff-distrib′)
  also have ... = (l1 − l1 * f1) * (1 − f2) + ((l2 − l2 * f2) * (1 − f1))
    by (simp add: left-diff-distrib′)
  also have ... = l1 * ((1 − f1) * (1 − f2)) + ((l2 − l2 * f2) * (1 − f1))
    by (simp add: vector-space-over-itself.scale-right-diff-distrib)
  also have ... = l1 * ((1 − f1) * (1 − f2)) + (l2 * ((1 − f2) * (1 − f1)))
    by (simp add: vector-space-over-itself.scale-right-diff-distrib)
  also have ... = l1 * ((1 − f1) * (1 − f2)) + (l2 * ((1 − f1) * (1 − f2)))
    by simp
  also have ... = (l1 + l2) * ((1 − f1) * (1 − f2))
    by (simp add: distrib-right)
  finally have df − (l1*f1*(1−f2) + l2*f2*(1−f1)) =
    (l1 + l2) * ((1 − f1) * (1 − f2)) .
  thus ?thesis by simp
qed
also have ... = df / ((1 − f1) * (1 − f2))
  using ‹0 < l1 ∨ 0 < l2› ‹0 ≤ l1› ‹0 ≤ l2› by fastforce
also have ... = l1*(1−f2)/((1 − f1) * (1 − f2)) +
    l2*(1−f1)/ ((1 − f1) * (1 − f2))
  using df-def by (simp add: add-divide-distrib)
also have ... = l1/(1−f1) + l2*(1−f1)/ ((1 − f1) * (1 − f2))
  using ‹0 < 1 − f2› by auto
also have ... = l1/(1−f1) + l2/(1−f2)
  using ‹0 < 1 − f1› by auto
also have ... = gross-fct (lq P1) (fee P1) i +
```

gross-fct (*lq P2*) (*fee P2*) *i*
by (*simp add*: *f1-def f2-def gross-fct-def l1-def l2-def one-cpl-def*)
finally show *?thesis* **.**
qed

lemma *quote-gross-join*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *joint-pools P P1 P2*
shows *quote-gross P x* = *quote-gross P1 x* + *quote-gross P2 x*
proof −
have *quote-gross P x* =
gen-quote (*grd P*) (*gross-fct* (*lq P1*) (*fee P1*)) *x* +
gen-quote (*grd P*) (*gross-fct* (*lq P2*) (*fee P2*)) *x*
unfolding *quote-gross-def*
proof (*rule finite-nz-support.gen-quote-plus*)
show *finite-nz-support* (*gross-fct* (*lq P*) (*fee P*))
using *finite-liq-pool.finite-liq-gross-fct joint-pools-finite-liq assms*
by (*metis clmm-dsc-liq*(*1*) *finite-liq-pool.intro finite-nz-support-def*
nz-support-def)
show ∀ *i*. *0* ≤ *gross-fct* (*lq P1*) (*fee P1*) *i*
using *clmm-dsc-fees clmm-dsc-liq*(*2*) *assms*(*1*) *gross-fct-sgn* **by** *blast*
show ∀ *i*. *0* ≤ *gross-fct* (*lq P2*) (*fee P2*) *i*
using *clmm-dsc-fees clmm-dsc-liq*(*2*) *assms*(*2*) *gross-fct-sgn* **by** *blast*
show ∀ *i*. *gross-fct* (*lq P*) (*fee P*) *i* = *gross-fct* (*lq P1*) (*fee P1*) *i* +
gross-fct (*lq P2*) (*fee P2*) *i*
using *join-gross-fct assms* **by** *auto*
qed
also have ... = *quote-gross P1 x* + *quote-gross P2 x*
using *assms joint-pools-grids* **unfolding** *quote-gross-def* **by** *simp*
finally show *?thesis* **.**
qed

lemma *quote-net-join*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *joint-pools P P1 P2*
shows *quote-net P x* = *quote-net P1 x* + *quote-net P2 x*
proof −
have *quote-net P x* = *gen-quote* (*grd P*) (*lq P1*) *x* +
gen-quote (*grd P*) (*lq P2*) *x*
unfolding *quote-net-def*
proof (*rule finite-nz-support.gen-quote-plus*)
show *finite-nz-support* (*lq P*)
by (*meson clmm-dsc-def assms finite-liq-def finite-nz-support-def*
joint-pools-finite-liq)
show ∀ *i*. *0* ≤ *lq P1 i* **using** *clmm-dsc-liq*(*2*) *assms*(*1*) **by** *auto*
show ∀ *i*. *0* ≤ *lq P2 i* **by** (*simp add*: *clmm-dsc-liq*(*2*) *assms*(*2*))
show ∀ *i*. *lq P i* = *lq P1 i* + *lq P2 i* **by** (*simp add*: *assms*(*3*) *joint-pools-lq*)

```
  qed
  also have ... = quote-net P1 x + quote-net P2 x
    using assms joint-pools-grids unfolding quote-net-def by simp
  finally show ?thesis .
qed

lemma base-gross-join:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and joint-pools P P1 P2
shows base-gross P x = base-gross P1 x + base-gross P2 x
proof −
  have base-gross P x =
      gen-base (grd P) (gross-fct (lq P1) (fee P1)) x +
      gen-base (grd P) (gross-fct (lq P2) (fee P2)) x
    unfolding base-gross-def
  proof (rule finite-nz-support.gen-base-gross)
    show finite-nz-support (gross-fct (lq P) (fee P))
      using finite-liq-pool.finite-liq-gross-fct joint-pools-finite-liq assms
      by (metis clmm-dsc-liq(1) finite-liq-pool.intro finite-nz-support-def
          nz-support-def)
    show ∀ i. 0 ≤ gross-fct (lq P1) (fee P1) i
      using clmm-dsc-fees clmm-dsc-liq(2) assms(1) gross-fct-sgn by blast
    show ∀ i. 0 ≤ gross-fct (lq P2) (fee P2) i
      using clmm-dsc-fees clmm-dsc-liq(2) assms(2) gross-fct-sgn by blast
    show ∀ i. gross-fct (lq P) (fee P) i = gross-fct (lq P1) (fee P1) i +
        gross-fct (lq P2) (fee P2) i
      using join-gross-fct assms by auto
  qed
  also have ... = base-gross P1 x + base-gross P2 x
    using assms joint-pools-grids unfolding base-gross-def by simp
  finally show ?thesis .
qed

lemma base-net-join:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and joint-pools P P1 P2
shows base-net P x = base-net P1 x + base-net P2 x
proof −
  have base-net P x = gen-base (grd P) (lq P1) x +
      gen-base (grd P) (lq P2) x
    unfolding base-net-def
  proof (rule finite-nz-support.gen-base-gross)
    show finite-nz-support (lq P)
      by (meson clmm-dsc-def assms finite-liq-def finite-nz-support-def
          joint-pools-finite-liq)
    show ∀ i. 0 ≤ lq P1 i using clmm-dsc-liq(2) assms(1) by auto
    show ∀ i. 0 ≤ lq P2 i by (simp add: clmm-dsc-liq(2) assms(2))
```

```
    show ∀ i. lq P i = lq P1 i + lq P2 i by (simp add: assms(3) joint-pools-lq)
  qed
  also have ... = base-net P1 x + base-net P2 x
    using assms joint-pools-grids unfolding base-net-def by simp
  finally show ?thesis .
qed

lemma mkt-depth-join:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and joint-pools P P1 P2
shows mkt-depth P x x' = mkt-depth P1 x x' + mkt-depth P2 x x'
  using assms unfolding mkt-depth-def
  by (simp add: quote-net-join base-net-join)

lemma joint-quote-gross-decomp:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and joint-pools P P1 P2
  and nz-support (lq P) ≠ {}
  and grd-min P ≤ x
  and 0 ≤ y
  and y + quote-gross P x ≤ quote-gross P (grd-max P)
  and x' = quote-reach P (y + quote-gross P x)
  and y1 = quote-gross P1 x' − quote-gross P1 x
  and y2 = quote-gross P2 x' − quote-gross P2 x
shows y = y1 + y2
proof −
  interpret finite-liq-pool P
    using assms joint-pools-finite-liq clmm-dsc-liq finite-liq-pool.intro
    by blast
  have clmm-dsc P using assms joint-pools-clmm-dsc[of P1] by simp
  have y1 + y2 = quote-gross P1 x' + quote-gross P2 x' −
      (quote-gross P1 x + quote-gross P2 x)
    using assms by simp
  also have ... = quote-gross P x' − quote-gross P x
    using quote-gross-join assms by auto
  also have ... = y
  proof −
    have quote-gross P (quote-reach P (y + quote-gross P x)) =
        y + quote-gross P x
    proof (rule quote-gross-reach-eq)
      show ∀ i. fee P i < 1 using ‹clmm-dsc P›
        by (simp add: clmm-dsc-fees)
      show mono (grd P)
        by (simp add: ‹clmm-dsc P› clmm-dsc-grd-mono monoI)
      show 0 ≤ y + quote-gross P x
        by (simp add: ‹clmm-dsc P› assms(6) clmm-quote-gross-pos)
      show y + quote-gross P x ≤ quote-gross P (grd-max P)
```

```
      using assms by simp
    show ∀ i. 0 ≤ lq P i
      by (simp add: ‹clmm-dsc P› clmm-dsc-liq(2))
  qed
  thus ?thesis using assms by simp
 qed
 finally show ?thesis by simp
qed

lemma joint-quote-gross-decomp':
 assumes clmm-dsc P1
 and clmm-dsc P2
 and joint-pools P P1 P2
 and nz-support (lq P) ≠ {}
 and 0 ≤ y
 and y ≤ quote-gross P (grd-max P)
 and x' = quote-reach P y
 and y1 = quote-gross P1 x'
 and y2 = quote-gross P2 x'
shows y = y1 + y2
proof −
 interpret finite-liq-pool P
  using assms joint-pools-finite-liq clmm-dsc-liq finite-liq-pool.intro
  by blast
 have clmm-dsc P using assms joint-pools-clmm-dsc[of P1] by simp
 have y1 + y2 = quote-gross P1 x' + quote-gross P2 x'
  using assms by simp
 also have ... = quote-gross P x'
  using quote-gross-join assms by auto
 also have ... = y
 proof −
  have quote-gross P (quote-reach P y) = y
  proof (rule quote-gross-reach-eq)
    show ∀ i. fee P i < 1 using ‹clmm-dsc P›
      by (simp add: clmm-dsc-fees)
    show mono (grd P)
      by (simp add: ‹clmm-dsc P› clmm-dsc-grd-mono monoI)
    show 0 ≤ y  using assms by simp
    show y ≤ quote-gross P (grd-max P) using assms by simp
    show ∀ i. 0 ≤ lq P i by (simp add: ‹clmm-dsc P› clmm-dsc-liq(2))
  qed
  thus ?thesis using assms by simp
 qed
 finally show ?thesis by simp
qed

lemma joint-base-net-decomp':
 assumes clmm-dsc P1
 and clmm-dsc P2
```

```
  and joint-pools P P1 P2
  and nz-support (lq P) ≠ {}
  and 0 ≤ y
  and y ≤ quote-gross P (grd-max P)
  and x' = quote-reach P y
  and y1 = base-net P1 x'
  and y2 = base-net P2 x'
shows base-net P x' = y1 + y2
proof −
  interpret finite-liq-pool P
    using assms joint-pools-finite-liq clmm-dsc-liq finite-liq-pool.intro
    by blast
  have clmm-dsc P using assms joint-pools-clmm-dsc[of P1] by simp
  have y1 + y2 = base-net P1 x' + base-net P2 x'
    using assms by simp
  also have ... = base-net P x'
    using base-net-join assms by auto
  finally show ?thesis by simp
qed

lemma joint-base-gross-decomp:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and joint-pools P P1 P2
  and nz-support (lq P) ≠ {}
  and x ≤ grd-max P
  and 0 ≤ y
  and y + base-gross P x ≤ base-gross P (grd-min P)
  and x' = base-reach P (y + base-gross P x)
  and y1 = base-gross P1 x' − base-gross P1 x
  and y2 = base-gross P2 x' − base-gross P2 x
shows y = y1 + y2
proof −
  interpret finite-liq-pool P
    using assms joint-pools-finite-liq clmm-dsc-liq finite-liq-pool.intro
    by blast
  have clmm-dsc P using assms joint-pools-clmm-dsc[of P1] by simp
  have y1 + y2 = base-gross P1 x' + base-gross P2 x' −
      (base-gross P1 x + base-gross P2 x)
    using assms by simp
  also have ... = base-gross P x' − base-gross P x
    using base-gross-join assms by auto
  also have ... = y
  proof −
    have base-gross P (base-reach P (y + base-gross P x)) =
        y + base-gross P x
    proof (rule base-gross-dwn)
      show ∀ i. fee P i < 1 using ‹clmm-dsc P› by (simp add: clmm-dsc-fees)
     show mono (grd P) by (simp add: ‹clmm-dsc P› clmm-dsc-grd-mono monoI)
```

```
    show grd-min P ≤ grd-max P
    proof (rule grd-min-max)
      show nz-support (lq P) ≠ {} using assms by simp
      show mono (grd P) using ‹clmm-dsc P› span-gridD clmm-dsc-grid
        by (simp add: strict-mono-on-imp-mono-on)
    qed
    have base-gross P (grd-max P) ≤ base-gross P x
      using assms clmm-base-gross-antimono ‹clmm-dsc P› antimonoD by blast
    show 0 ≤ y + base-gross P x
     using ‹base-gross P (grd-max P) ≤ base-gross P x› ‹mono (grd P)› assms(6)
        base-gross-grd-max fin-nz-sup
      by simp
    show y + base-gross P x ≤ base-gross P (grd-min P)
      using assms by simp
    show ∀ i. grd P i ≤ grd P (i + 1)
      using ‹clmm-dsc P› span-gridD clmm-dsc-grid
      by (simp add: strict-mono-leD)
    show ∀ i. 0 < grd P i
      using ‹clmm-dsc P› span-gridD clmm-dsc-grid by presburger
    show ∀ i. 0 ≤ lq P i
      by (simp add: ‹clmm-dsc P› clmm-dsc-liq(2))
   qed
   thus ?thesis using assms by simp
  qed
  finally show ?thesis by simp
qed

definition join-pools where
join-pools P1 P2 =
  (grd P1,
   (λi. lq P1 i + lq P2 i),
   (λi. pool-fee-join P1 P2 i))

lemma join-pools-grd[simp]:
  assumes P = join-pools P1 P2
  shows grd P = grd P1 using assms unfolding grd-def join-pools-def by simp

lemma join-pools-lq[simp]:
  assumes P = join-pools P1 P2
  shows lq P i =  lq P1 i + lq P2 i
  using assms unfolding lq-def join-pools-def by simp

lemma join-pools-fee[simp]:
  assumes P = join-pools P1 P2
  shows fee P i =  pool-fee-join P1 P2 i
  using assms unfolding fee-def join-pools-def by simp

lemma join-joint-pools:
  assumes grd P1 = grd P2
```

shows *joint-pools* (*join-pools P1 P2*) *P1 P2*
proof
show *grd* (*join-pools P1 P2*) = *grd P1* **by** *simp*
show *grd* (*join-pools P1 P2*) = *grd P2* **using** *assms* **by** *simp*
fix *i*
show *lq* (*join-pools P1 P2*) *i* = *lq P1 i* + *lq P2 i* **by** *simp*
show *fee* (*join-pools P1 P2*) *i* = *pool-fee-join P1 P2 i* **by** *simp*
qed

6.4 CLMM pool combination

definition *pool-comb* **where**
pool-comb P1 P2 sqp = (*let P′* = *refine P1 sqp in*
pool-join P′ (*slice-pool P2 sqp*))

lemma *pool-comb-joint*:
assumes *grd P1* = *grd P2*
shows *joint-pools* (*pool-comb P1 P2 sqp*) (*refine P1 sqp*)
(*slice-pool P2 sqp*) **unfolding** *pool-comb-def Let-def*
proof (*rule pool-join-joint*)
show *grd* (*refine P1 sqp*) = *grd* (*slice-pool P2 sqp*)
using *refine-grd-cong*[*of refine P1 sqp*] *assms*
by (*simp add*: *slice-poolD*)
qed *simp*+

lemma *pool-comb-refined-joint-nz-liq*:
assumes *grd P1* = *grd P2*
and *clmm-dsc P1*
and *clmm-dsc P2*
and *P* = *pool-comb P1 P2 sqp*
and *grd P1* (*lower-tick P1 sqp*) = *sqp*
shows *nz-support* (*lq P*) = *nz-support* (*lq P1*) ∪
(*nz-support* (*lq* (*slice-pool P2 sqp*)))
by (*metis assms*(1−5) *clmm-dsc-grid*(2) *clmm-joint-pools-nz-liq pool-comb-joint*
refine-eq slice-pool-clmm-dsc)

lemma *pool-comb-joint-refined*:
assumes *grd P1* = *grd P2*
and *grd P1* (*lower-tick P1 sqp*) = *sqp*
shows *joint-pools* (*pool-comb P1 P2 sqp*) *P1*
(*slice-pool P2 sqp*)
proof −
have *eq*: *grd P2* (*lower-tick P2 sqp*) = *sqp*
by (*metis assms*(1) *assms*(2) *grd-lower-tick-cong*)
have *refine P1 sqp* = *P1* **using** *assms refine-eq* **by** *simp*
moreover have *refine P2 sqp* = *P2* **using** *assms eq refine-eq* **by** *simp*
ultimately show *?thesis*
using *pool-join-joint assms* **unfolding** *pool-comb-def Let-def*
by (*metis pool-comb-def pool-comb-joint*)

qed

lemma *pool-comb-clmm-dsc*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *grd P1 = grd P2*
and *0 < sqp*
and *P3 = pool-comb P1 P2 sqp*
shows *clmm-dsc P3* **unfolding** *pool-comb-def Let-def*
proof (*rule joint-pools-clmm-dsc*)
define *P* **where** *P = refine P1 sqp*
define *P'* **where** *P' = slice-pool* (*refine P2 sqp*) *sqp*
show *clmm-dsc P* **using** *refine-clmm assms* **unfolding** *P-def* **by** *simp*
show *clmm-dsc P'*
proof (*rule slice-pool-clmm-dsc*)
show *clmm-dsc* (*refine P2 sqp*) **using** *refine-clmm assms* **by** *simp*
show *0 < sqp* **using** *assms* **by** *simp*
qed (*simp add*: *P'-def*)
show *joint-pools P3 P P'*
using *pool-join-joint assms P'-def P-def pool-comb-joint*
by (*metis refine-eq refine-lower-tick slice-poolD*)
qed

lemma *pool-comb-grd-min*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *grd P1 = grd P2*
and *nz-support* (*lq P1*) ≠ {}
and *nz-support* (*lq P2*) ≠ {}
and *0 < sqp*
and *sqp < grd-max P2*
and *P = pool-comb P1 P2 sqp*
shows *grd-min P = min* (*grd-min P1*) (*grd-min* (*slice-pool P2 sqp*))
proof −
define *i* **where** *i = idx-min* (*lq P*)
define *i1* **where** *i1 = idx-min* (*lq* (*refine P1 sqp*))
define *i2* **where** *i2 = idx-min* (*lq* (*slice-pool P2 sqp*))
have *clmm-dsc P* **using** *assms pool-comb-clmm-dsc*[*of P1*] **by** *simp*
have *i = min i1 i2* **unfolding** *i-def i1-def i2-def*
proof (*rule joint-pools-idx-min*)
show *clmm-dsc* (*refine P1 sqp*)
by (*meson assms*(*1*) *assms*(*6*) *refine-clmm*)
show *clmm-dsc* (*slice-pool P2 sqp*)
by (*meson assms*(*2*) *assms*(*6*) *refine-clmm slice-pool-clmm-dsc*)
show *nz-support* (*lq* (*refine P1 sqp*)) ≠ {}
using *assms*(*1*) *assms*(*4*) *refine-nz-lq-ne* **by** *auto*
show *nz-support* (*lq* (*slice-pool P2 sqp*)) ≠ {}
using *assms slice-pool-nz-liq' clmm-dsc-liq*(*1*) *finite-liq-pool.intro*
refine-grd-max refine-clmm refine-nz-lq-ne

```
    by presburger
  show joint-pools P (refine P1 sqp) (slice-pool P2 sqp)
    by (metis assms(3,8) pool-comb-joint)
 qed
 have grd-min P = grd P i
  using grd-min-def idx-min-img-def i-def by simp
 also have ... = min (grd P i1) (grd P i2)
  using ‹i = min i1 i2›
  by (metis ‹clmm-dsc P› clmm-dsc-grd-smono linorder-not-less min.absorb4
      min.order-iff min.strict-order-iff)
 also have ... = min (grd (refine P1 sqp) i1)
    (grd (slice-pool P2 sqp) i2)
 proof −
  have grd (refine P1 sqp) = grd P using assms
    by (metis pool-comb-joint joint-pools-grids(1))
  moreover have grd (slice-pool P2 sqp) = grd P
    by (metis assms(3) assms(8) pool-comb-joint joint-pools-def)
  ultimately show ?thesis by simp
 qed
 also have ... = min (grd-min (refine P1 sqp))
    (grd-min (slice-pool P2 sqp))
  using i1-def i2-def unfolding grd-min-def idx-min-img-def by simp
 also have ... = min (grd-min P1) (grd-min ( slice-pool P2 sqp))
  using refine-grd-min assms by simp
 finally show ?thesis .
qed

lemma pool-comb-le-grd-min:
 assumes clmm-dsc P1
 and clmm-dsc P2
 and grd P1 = grd P2
 and nz-support (lq P1) ≠ {}
 and nz-support (lq P2) ≠ {}
 and 0 < sqp
 and sqp < grd-max P2
 and grd-min P1 ≤ sqp
 and P = pool-comb P1 P2 sqp
shows grd-min P = grd-min P1
proof −
 have sqp ≤ grd-min (slice-pool P2 sqp)
  by (rule slice-pool-grd-min, auto simp add: assms)
 thus ?thesis using assms pool-comb-grd-min by simp
qed

lemma pool-comb-grd-max:
 assumes clmm-dsc P1
 and clmm-dsc P2
 and grd P1 = grd P2
 and nz-support (lq P1) ≠ {}
```

```
  and nz-support (lq P2) ≠ {}
  and 0 < sqp
  and sqp < grd-max P2
  and P = pool-comb P1 P2 sqp
shows grd-max P = max (grd-max P1) (grd-max P2)
proof −
  define i where i = idx-max (lq P)
  define i1 where i1 = idx-max (lq (refine P1 sqp))
  define i2 where i2 = idx-max (lq (slice-pool P2 sqp))
  have clmm-dsc P using assms pool-comb-clmm-dsc[of P1] by simp
  have i = max i1 i2 unfolding i-def i1-def i2-def
  proof (rule joint-pools-idx-max)
    show clmm-dsc (refine P1 sqp)
      by (meson assms(1) assms(6) refine-clmm)
    show clmm-dsc (slice-pool P2 sqp)
      by (meson assms(2) assms(6) refine-clmm slice-pool-clmm-dsc)
    show nz-support (lq (refine P1 sqp)) ≠ {}
      using assms(1) assms(4) refine-nz-lq-ne by auto
    show nz-support (lq (slice-pool P2 sqp)) ≠ {}
      using assms slice-pool-nz-liq' clmm-dsc-liq(1) finite-liq-pool.intro
        refine-grd-max refine-clmm refine-nz-lq-ne
      by presburger
    show joint-pools P (refine P1 sqp) (slice-pool P2 sqp)
      by (simp add: assms(3) assms(8) pool-comb-joint)
  qed
  hence i+1 = max (i1+1) (i2+1) by simp
  have grd-max P = grd P (i+1)
    using grd-max-def idx-max-img-def i-def by simp
  also have ... = max (grd P (i1+1)) (grd P (i2+1))
    using ‹i+1 = max (i1+1) (i2+1)›
    by (metis ‹clmm-dsc P› clmm-dsc-grd-mono max.orderE max-absorb2 nle-le)
  also have ... = max (grd (refine P1 sqp) (i1+1))
      (grd (slice-pool P2 sqp) (i2+1))
  proof −
    have grd (refine P1 sqp) = grd P using assms
      by (metis pool-comb-joint joint-pools-grids(1))
    moreover have grd (slice-pool P2 sqp) = grd P
      by (metis assms(3) assms(8) pool-comb-joint joint-pools-def)
    ultimately show ?thesis by simp
  qed
  also have ... = max (grd-max (refine P1 sqp))
      (grd-max ( slice-pool P2 sqp))
    using i1-def i2-def unfolding grd-max-def idx-max-img-def by simp
  also have ... = max (grd-max P1) (grd-max P2)
    using refine-grd-max slice-pool-grd-max' assms(1) assms(2) assms(4−7)
      refine-clmm refine-nz-lq-ne
    by presburger
  finally show ?thesis .
qed
```

lemma *pool-comb-grd-max-slice*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *grd P1 = grd P2*
and *nz-support (lq P1) ≠ {}*
and *nz-support (lq P2) ≠ {}*
and *0 < sqp*
and *sqp < grd-max P2*
and *P = pool-comb P1 P2 sqp*
shows *sqp < grd-max P*
proof (*cases grd-max P1 ≤ grd-max P2*)
case *True*
hence *grd-max P = grd-max P2* **using** *assms pool-comb-grd-max*
by (*metis max.absorb1 max.commute*)
then show *?thesis* **using** *assms* **by** *simp*
next
case *False*
hence *grd-max P = grd-max P1* **using** *assms pool-comb-grd-max*
by (*metis linorder-not-less max.absorb3*)
then show *?thesis* **using** *assms False* **by** *simp*
qed

lemma *pool-comb-quote-decomp*:
assumes *clmm-dsc P1*
and *clmm-dsc P2*
and *grd P1 = grd P2*
and *grd P1 (lower-tick P1 sqp) = sqp*
and *0 < sqp*
and *sqp′ ≤ grd-max P*
and *P = pool-comb P1 P2 sqp*
and *nz-support (lq P) ≠ {}*
shows *quote-gross P sqp′ = quote-gross P1 sqp′ + quote-gross (slice-pool P2 sqp) sqp′*
proof −
define *P″* **where** *P″ = slice-pool P2 sqp*
have *clmm-dsc P* **using** *pool-comb-clmm-dsc assms* **by** *simp*
interpret *finite-liq-pool*
by (*simp add*: ‹*clmm-dsc P*› *clmm-dsc-liq*(*1*) *finite-liq-pool.intro*)
define *sqp″* **where** *sqp″ = quote-reach P (quote-gross P sqp′)*
have *quote-gross P sqp′ = quote-gross P sqp″* **unfolding** *sqp″-def*
proof (*rule clmm-quote-gross-reach-eq*[*symmetric*])
show *clmm-dsc P* **using** ‹*clmm-dsc P*› **.**
show *nz-support (lq P) ≠ {}* **using** *assms* **by** *simp*
show *0 ≤ quote-gross P sqp′*
using *clmm-quote-gross-pos* ‹*clmm-dsc P*› **by** *simp*
show *quote-gross P sqp′ ≤ quote-gross P (grd-max P)*
using ‹*clmm-dsc P*› *clmm-quote-gross-mono assms* **by** (*metis monoD*)
qed

```
  also have ... = quote-gross P1 sqp'' + quote-gross P'' sqp''
  proof (rule joint-quote-gross-decomp')
    show joint: joint-pools P P1 P''
      using assms pool-comb-joint-refined unfolding P''-def by simp
    show clmm-dsc P1 using assms by simp
    show clmm-dsc P''
      using refine-clmm slice-pool-clmm-dsc assms
      unfolding P''-def by auto
    show nz-support (lq P) ≠ {} using assms by simp
    show 0 ≤ quote-gross P sqp''
      using clmm-quote-gross-pos ‹clmm-dsc P› by simp
    show sqp'' = quote-reach P (quote-gross P sqp'')
      using assms sqp''-def calculation by presburger
    show quote-gross P sqp'' ≤ quote-gross P (grd-max P)
      using assms clmm-quote-gross-mono ‹clmm-dsc P› monoD calculation
      by metis
  qed simp+
  also have ... = quote-gross P1 sqp' + quote-gross P'' sqp'
    by (metis P''-def assms(1−5,7) calculation pool-comb-joint-refined
        quote-gross-join slice-pool-clmm-dsc)
  finally show quote-gross P sqp' = quote-gross P1 sqp' + quote-gross P'' sqp' .
qed

lemma pool-comb-quote-le-slice:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and grd P1 = grd P2
  and grd P1 (lower-tick P1 sqp) = sqp
  and 0 < sqp
  and sqp' ≤ sqp
  and sqp ≤ grd-max P
  and P = pool-comb P1 P2 sqp
  and nz-support (lq P) ≠ {}
shows quote-gross P sqp' = quote-gross P1 sqp'
proof −
  have quote-gross P sqp' = quote-gross P1 sqp' +
      quote-gross (slice-pool P2 sqp) sqp'
    using assms pool-comb-quote-decomp by simp
  moreover have quote-gross (slice-pool P2 sqp) sqp' = 0
    using assms(2,5,6) slice-pool-quote-gross-leq by blast
  ultimately show ?thesis by simp
qed

lemma pool-comb-quote-diff-decomp:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and grd P1 = grd P2
  and grd P1 (lower-tick P1 sqp) = sqp
  and 0 < sqp
```

```
  and 0 < sqp'
  and 0 < sqp1
  and sqp' ≤ grd-max P
  and sqp1 ≤ grd-max P
  and P = pool-comb P1 P2 sqp
  and nz-support (lq P) ≠ {}
shows quote-gross P sqp' − quote-gross P sqp1 =
  quote-gross P1 sqp'− quote-gross P1 sqp1 +
  quote-gross (slice-pool P2 sqp) sqp' − quote-gross (slice-pool P2 sqp) sqp1
proof −
  define P'' where P'' = slice-pool P2 sqp
  have clmm-dsc P using pool-comb-clmm-dsc assms by simp
  interpret finite-liq-pool
    by (simp add: ‹clmm-dsc P› clmm-dsc-liq(1) finite-liq-pool.intro)
  have quote-gross P sqp' = quote-gross P1 sqp' + quote-gross P'' sqp'
    using assms P''-def pool-comb-quote-decomp by simp
  moreover have quote-gross P sqp1 = quote-gross P1 sqp1 + quote-gross P''
sqp1
    using assms P''-def pool-comb-quote-decomp by simp
  ultimately show ?thesis unfolding P''-def by linarith
qed

lemma pool-comb-base-net-plus:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and grd P1 = grd P2
  and grd P1 (lower-tick P1 sqp2) = sqp2
  and 0 < sqp2
  and 0 < y
  and y ≤ quote-gross P (grd-max P)
  and P = pool-comb P1 P2 sqp2
  and sqp' = quote-reach P y
  and sqp' ≤ sqp2
  and nz-support (lq P) ≠ {}
shows base-net P sqp' = base-net P1 sqp' + base-net (slice-pool P2 sqp2) sqp'
proof −
  define P'' where P'' = slice-pool P2 sqp2
  have clmm-dsc P using pool-comb-clmm-dsc assms by simp
  interpret finite-liq-pool
    by (simp add: ‹clmm-dsc P› clmm-dsc-liq(1) finite-liq-pool.intro)
  have y = quote-gross P sqp'
    using clmm-quote-gross-reach-eq assms ‹clmm-dsc P› by auto
  show base-net P sqp' = base-net P1 sqp' + base-net P'' sqp'
  proof (rule joint-base-net-decomp')
    show joint: joint-pools P P1 P''
      using assms pool-comb-joint-refined unfolding P''-def by simp
    show clmm-dsc P1 using assms by simp
    show clmm-dsc P''
      using refine-clmm slice-pool-clmm-dsc assms
```

```
    unfolding P''-def by auto
  show nz-support (lq P) ≠ {} using assms by simp
  show 0 ≤ quote-gross P sqp'
    using clmm-quote-gross-pos ‹clmm-dsc P› by simp
  show sqp' = quote-reach P (quote-gross P sqp')
    using assms ‹y = quote-gross P sqp'› by presburger
  show quote-gross P sqp' ≤ quote-gross P (grd-max P)
    using assms ‹y = quote-gross P sqp'› by linarith
 qed simp+
qed

lemma combo-quote-init1:
 assumes clmm-dsc P1
 and clmm-dsc P2
 and grd P1 = grd P2
 and grd P1 (lower-tick P1 sqp2) = sqp2
 and 0 < sqp2
 and P = pool-comb P1 P2 sqp2
 and 0 < y
 and nz-support (lq P1) ≠ {}
 and nz-support (lq P2) ≠ {}
 and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
 and sqp' = quote-reach P (y + quote-gross P sqp1)
 and sqp2 < grd-max P2
 and sqp1 ≤ sqp2
shows quote-gross P sqp1 = quote-gross P1 sqp1
proof (rule pool-comb-quote-le-slice)
 have clmm-dsc P using pool-comb-clmm-dsc assms by simp
 show nz-support (lq P) ≠ {} using pool-comb-refined-joint-nz-liq assms by simp
 hence qa: quote-gross P sqp' = y + quote-gross P sqp1
   using assms clmm-quote-gross-reach-eq ‹clmm-dsc P› clmm-quote-gross-pos
   by auto
 show clmm-dsc P1 using assms by simp
 show clmm-dsc P2 using assms by simp
 show grd P1 = grd P2 using assms by simp
 show grd P1 (lower-tick P1 sqp2) = sqp2 using assms by simp
 show P = pool-comb P1 P2 sqp2 using assms by simp
 show 0 < sqp2 using assms assms by simp
 show sqp1 ≤ sqp2 using assms by simp
 have sqp2 < grd-max P using pool-comb-grd-max-slice assms by simp
 thus sqp2 ≤ grd-max P by simp
qed

lemma combo-remain-quote-eq:
 assumes clmm-dsc P1
 and clmm-dsc P2
 and grd P1 = grd P2
 and grd P1 (lower-tick P1 sqp2) = sqp2
 and 0 < sqp2
```

```
  and P = pool-comb P1 P2 sqp2
  and nz-support (lq P) ≠ {}
  and nz-support (lq P2) ≠ {}
  and 0 < y
  and 0< sqp1
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
  and sqp' = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
  and sqp2 ≤ sqp'
  and sqp1 ≤ sqp2
  and rs1' = quote-reach P2 (y − y1 + quote-gross P2 sqp2)
shows quote-gross P2 sqp' = quote-gross P2 rs1'
proof −
  have clmm-dsc P using pool-comb-clmm-dsc assms by simp
  define P'' where P'' = slice-pool P2 sqp2
  define i where i = lower-tick P2 sqp2
  hence grd P2 i = sqp2
    using assms lower-tick-eq  by metis
  hence quote-gross P2 sqp' = quote-gross P'' sqp' + quote-gross P2 sqp2
    using slice-pool-quote-gross P''-def assms i-def
    by simp
  also have ... = quote-gross P sqp' − quote-gross P1 sqp' + quote-gross P2 sqp2
  proof −
    have quote-gross P sqp' = quote-gross P'' sqp'+ quote-gross P1 sqp'
      using pool-comb-quote-decomp P''-def assms pool-comb-joint-refined
        quote-gross-join slice-pool-clmm-dsc
      by (metis add.commute)
    thus ?thesis by simp
  qed
  also have ... = y − y1 + quote-gross P2 sqp2
  proof −
    have quote-gross P sqp' = y + quote-gross P sqp1
      using assms clmm-quote-gross-reach-eq ‹clmm-dsc P› clmm-quote-gross-pos
      by auto
    moreover have quote-gross P1 sqp' = y1 + quote-gross P1 sqp1
      using assms by simp
    moreover have quote-gross P sqp1 = quote-gross P1 sqp1
    proof (rule pool-comb-quote-le-slice)
      show clmm-dsc P1 using assms by simp
      show clmm-dsc P2 using assms by simp
      show grd P1 = grd P2 using assms by simp
      show grd P1 (lower-tick P1 sqp2) = sqp2 using assms by simp
      show nz-support (lq P) ≠ {} using assms by simp
      show P = pool-comb P1 P2 sqp2 using assms by simp
      show 0 < sqp2 using assms assms by simp
      show sqp1 ≤ sqp2 using assms by simp
      show sqp2 ≤ grd-max P
        by (metis ‹clmm-dsc P› assms(11) assms(12) assms(14) assms(7)
            calculation(1) quote-gross-imp-sqp-lt quote-gross-grd-max-ge
```

```
        grd-max-quote-reach linorder-not-less order-le-imp-less-or-eq)
    qed
    ultimately show ?thesis by simp
  qed
  also have ... = quote-gross P2 (quote-reach P2 (y − y1 + quote-gross P2 sqp2))
  proof (rule clmm-quote-gross-reach-eq[symmetric])
    show clmm-dsc P2 using assms by simp
    show nz-support (lq P2) ≠ {} using assms by simp
    show 0 ≤ y − y1 + quote-gross P2 sqp2
      by (metis assms(2) calculation clmm-quote-gross-pos)
    show y − y1 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)
      by (metis assms(2) assms(8) calculation quote-gross-grd-max-max)
  qed
  finally show ?thesis using assms by simp
qed

lemma comb-quote-gross-le:
  assumes clmm-dsc P1
  and clmm-dsc P2
  and grd P1 = grd P2
  and 0 < sqp
  and sqp < grd-max P
  and 0 < y
  and y ≤ quote-gross P sqp
  and y ≤ quote-gross P (grd-max P)
  and P = pool-comb P1 P2 sqp
  and sqp' = quote-reach P y
  and nz-support (lq P) ≠ {}
shows quote-gross P1 sqp' = y
proof −
  define P' where P' = refine P1 sqp
  define P'' where P'' = slice-pool P2 sqp
  hence quote-gross P'' sqp = 0 using slice-pool-quote-gross-leq
    by (metis assms(2) assms(4) dual-order.refl)
  have clmm-dsc P using pool-comb-clmm-dsc assms by simp
  interpret finite-liq-pool
    by (simp add: ‹clmm-dsc P› clmm-dsc-liq(1) finite-liq-pool.intro)
  have y = quote-gross P sqp'
    using clmm-quote-gross-reach-eq assms ‹clmm-dsc P› by auto
  hence sqp' ≤ sqp
    using ‹clmm-dsc P› quote-reach-le-gross assms order-less-imp-le
    by blast
  hence quote-gross P'' sqp' = 0 using slice-pool-quote-gross-leq
    by (metis P''-def assms(2,4))
  have quote-gross P sqp' = quote-gross P' sqp' + quote-gross P'' sqp
  proof (rule joint-quote-gross-decomp')
    show joint: joint-pools P P' P''
      using assms pool-comb-joint unfolding P'-def P''-def by simp
    show clmm-dsc P'
```

```
    using refine-clmm  assms unfolding P'-def by simp
  show clmm-dsc P''
    using refine-clmm slice-pool-clmm-dsc assms
    unfolding P''-def by auto
  show nz-support (lq P) ≠ {} using assms by simp
  show 0 ≤ quote-gross P sqp'
    using clmm-quote-gross-pos ‹clmm-dsc P› by simp
  show sqp' = quote-reach P (quote-gross P sqp')
    using assms ‹y = quote-gross P sqp'› by presburger
  show quote-gross P sqp' ≤ quote-gross P (grd-max P)
    using assms ‹y = quote-gross P sqp'› by linarith
  show quote-gross P'' sqp = quote-gross P'' sqp'
    by (simp add: ‹quote-gross P'' sqp = 0› ‹quote-gross P'' sqp' = 0›)
 qed simp
 also have ... = quote-gross P' sqp' using ‹quote-gross P'' sqp = 0› by simp
 also have ... = quote-gross P1 sqp'
   using refine-quote-gross assms P'-def by simp
 finally show ?thesis using ‹y = quote-gross P sqp'› by simp
qed

locale combined-pools =
 fixes P1 P2 P sqp2
 assumes cmb-P1: clmm-dsc P1
 and cmb-P2: clmm-dsc P2
 and cmb-grd-eq: grd P1 = grd P2
 and cmb-on-grid: grd P1 (lower-tick P1 sqp2) = sqp2
 and cmb-nz1: nz-support (lq P1) ≠ {}
 and cmb-nz2: nz-support (lq P2) ≠ {}
 and cmb-comb: P = pool-comb P1 P2 sqp2
 and cmb-pos: 0 < sqp2
 and cmb-max: sqp2 < grd-max P2

begin

lemma combined-P-prop:
 shows clmm-dsc P nz-support (lq P) ≠ {}
proof −
 show clmm-dsc P
   using cmb-P1 cmb-P2 cmb-comb pool-comb-clmm-dsc cmb-grd-eq cmb-pos by
blast
 show nz-support (lq P) ≠ {}
   using  pool-comb-refined-joint-nz-liq cmb-P1 cmb-P2 cmb-comb cmb-grd-eq
     cmb-nz1 cmb-on-grid
   by blast
qed

lemmas cmb-props = cmb-P1 cmb-P2 cmb-grd-eq cmb-on-grid cmb-nz1 cmb-nz2
 cmb-comb cmb-pos cmb-max combined-P-prop
```

lemma *combo-joint-quote-gross-decomp*:
assumes $0 < y$
and $0 < sqp1$
and $y + quote\text{-}gross\ P\ sqp1 \leq quote\text{-}gross\ P\ (grd\text{-}max\ P)$
and $sqp' = quote\text{-}reach\ P\ (y + quote\text{-}gross\ P\ sqp1)$
and $y1 = quote\text{-}gross\ P1\ sqp' - quote\text{-}gross\ P1\ sqp1$
and $P'' = slice\text{-}pool\ P2\ sqp2$
and $y2' = quote\text{-}gross\ P''\ sqp' - quote\text{-}gross\ P''\ sqp1$
shows $y = y1 + y2'\ y1 \leq y\ 0 \leq y1$
$y1 + quote\text{-}gross\ P1\ sqp1 \leq quote\text{-}gross\ P1\ (grd\text{-}max\ P1)$
$y2' \leq quote\text{-}gross\ P''\ (grd\text{-}max\ P2)$
proof −
have *clmm-dsc P* **using** *combined-P-prop* **by** *simp*
have $nz\text{-}support\ (lq\ P) \neq \{\}$ **using** *combined-P-prop* **by** *simp*
have $quote\text{-}gross\ P\ sqp1 < quote\text{-}gross\ P\ (grd\text{-}max\ P)$ **using** *assms* **by** *simp*
hence $sqp1 < grd\text{-}max\ P$
using ‹*clmm-dsc P*› *quote-gross-imp-sqp-lt* **by** *blast*
define $sqp1'$ **where** $sqp1' = quote\text{-}reach\ P1\ (quote\text{-}gross\ P1\ sqp')$
have $quote\text{-}gross\ P\ sqp1 < quote\text{-}gross\ P\ sqp'$
using *quote-reach-add-gt assms* ‹*clmm-dsc P*›
‹$nz\text{-}support\ (lq\ P) \neq \{\}$›
by *simp*
hence $sqp1 < sqp'$
using ‹*clmm-dsc P*› *quote-gross-imp-sqp-lt*[*of P*] **by** *simp*
hence $0 < sqp'$
using *assms liq-grd-min combined-pools-axioms*
unfolding *combined-pools-def* **by** *fastforce*
have $quote\text{-}gross\ P1\ sqp' \leq quote\text{-}gross\ P1\ (grd\text{-}max\ P1)$
by (*simp add*: *cmb-P1 cmb-nz1 quote-gross-grd-max-max*)
thus $y1 + quote\text{-}gross\ P1\ sqp1 \leq quote\text{-}gross\ P1\ (grd\text{-}max\ P1)$
by (*simp add*: *assms*(5))
have *clmm-dsc P''* **using** *assms slice-pool-clmm-dsc cmb-P2 cmb-pos* **by** *simp*
have $grd\text{-}max\ P2 = grd\text{-}max\ P''$ **using** *slice-pool-grd-max'*
by (*simp add*: *assms cmb-P2 cmb-max cmb-nz2 cmb-pos*)
have $nz\text{-}support\ (lq\ P'') \neq \{\}$ **using** *slice-pool-nz-liq'*
by (*simp add*: *assms cmb-P2 cmb-max cmb-nz2 cmb-pos*)
define $sqp2'$ **where** $sqp2' = quote\text{-}reach\ P''\ (quote\text{-}gross\ P''\ sqp')$
have $quote\text{-}gross\ P\ sqp1 < quote\text{-}gross\ P\ sqp'$
using *quote-reach-add-gt assms* ‹*clmm-dsc P*›
‹$nz\text{-}support\ (lq\ P) \neq \{\}$›
by *simp*
hence $sqp1 < sqp'$
using ‹*clmm-dsc P*› *quote-gross-imp-sqp-lt*[*of P*] **by** *simp*
have $0 \leq y2'$
by (*metis* ‹*clmm-dsc P''*› ‹$sqp1 < sqp'$› *quote-gross-imp-sqp-lt*
diff-ge-0-iff-ge eucl-less-le-not-le linorder-less-linear
verit-comp-simplify1(2) *assms*(7))
show $y = y1 + y2'$
proof −

```
    have quote-gross P sqp' = y + quote-gross P sqp1
      using assms clmm-quote-gross-reach-eq ‹clmm-dsc P› clmm-quote-gross-pos
        ‹nz-support (lq P) ≠ {}›
      by auto
    hence y = quote-gross P sqp' − quote-gross P sqp1 by simp
    also have ... = quote-gross P1 sqp' − quote-gross P1 sqp1 +
        quote-gross (slice-pool P2 sqp2) sqp' −
        quote-gross (slice-pool P2 sqp2) sqp1
   proof (rule pool-comb-quote-diff-decomp[OF cmb-P1 cmb-P2 cmb-grd-eq cmb-on-grid])
      show  nz-support (lq P) ≠ {} P = pool-comb P1 P2 sqp2
          sqp1 ≤ grd-max P
        using ‹nz-support (lq P) ≠ {}› ‹sqp1 < grd-max P› cmb-comb by auto
      show 0 < sqp1 using assms liq-grd-min cmb-P1 cmb-nz1 by fastforce
      have 0 < grd-min P
        using ‹nz-support (lq P) ≠ {}› ‹clmm-dsc P› liq-grd-min by simp
      thus 0 < sqp'
        using assms clmm-quote-reach-ge ‹nz-support (lq P) ≠ {}›
        by (metis ‹clmm-dsc P› ‹quote-gross P sqp' = y + quote-gross P sqp1›
            add-less-le-mono clmm-quote-gross-pos less-add-same-cancel1)
      show sqp' ≤ grd-max P
        using quote-reach-leq-grd-max assms ‹nz-support (lq P) ≠ {}›
        by (simp add: ‹clmm-dsc P› clmm-quote-gross-pos)
      show 0 < sqp2 using cmb-pos by simp
    qed
    also have ... = y1 + y2' using assms  by simp
    finally show ?thesis .
  qed
  show y1 ≤ y using ‹0 ≤ y2'› ‹y = y1 + y2'› by simp
  show y2' ≤ quote-gross P'' (grd-max P2)
    by (metis ‹clmm-dsc P''› ‹nz-support (lq P'') ≠ {}›
        ‹grd-max P2 = grd-max P''› add.commute add-diff-cancel assms(7)
        clmm-quote-gross-pos quote-gross-grd-max-max diff-add-cancel
        diff-ge-0-iff-ge dual-order.trans)
  show 0 ≤ y1
    by (smt (verit, del-insts) ‹sqp1 < sqp'› assms(5) combined-pools-axioms com-
bined-pools-def
        quote-gross-imp-sqp-lt)
qed

lemma combo-joint-quote-gross-leq-max:
  assumes 0 < y
  and 0 < sqp1
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
  and sqp' = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
shows y− y1 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)
proof −
  define P'' where P'' = slice-pool P2 sqp2
  define y2' where y2' = quote-gross P'' sqp' − quote-gross P'' sqp1
```

```
  have y − y1 = y2′ using y2′-def P′′-def assms combo-joint-quote-gross-decomp
    by (metis add-diff-cancel-left′)
  also have ... ≤ quote-gross P′′ (grd-max P2)
    using assms combo-joint-quote-gross-decomp
    unfolding y2′-def P′′-def
    by metis
  also have ... = quote-gross P2 (grd-max P2) − quote-gross P2 sqp2
    unfolding P′′-def
    using cmb-P2 cmb-grd-eq cmb-max cmb-on-grid cmb-pos lower-tick-eq
      slice-pool-quote-gross
    by auto
  finally have y − y1 ≤ quote-gross P2 (grd-max P2) − quote-gross P2 sqp2 .
  thus ?thesis by simp
qed

lemma combo-joint-quote-gross-price-le:
  assumes 0 < y
  and grd-min P1 ≤ sqp1
  and sqp1 ≤ sqp2
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
  and sqp′ = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp′ − quote-gross P1 sqp1
  and rs1 = quote-reach P1 (y1 + quote-gross P1 sqp1)
shows rs1 ≤ sqp′
proof (cases y1 + quote-gross P1 sqp1 = 0)
  case True
  hence rs1 = grd-min P1 using assms
    by (simp add: clmm-quote-reach-zero cmb-P1 cmb-nz1)
  also have ... = grd-min P using assms pool-comb-le-grd-min
   by (simp add: cmb-P1 cmb-P2 cmb-comb cmb-grd-eq cmb-max cmb-nz1 cmb-nz2
       cmb-pos)
  also have ... < sqp′
  proof −
    have 0 < y + quote-gross P sqp1
      by (simp add: add-pos-nonneg assms clmm-quote-gross-pos combined-P-prop)
    thus ?thesis using assms
      by (simp add: combined-P-prop quote-reach-gt-grd-min)
  qed
  finally show ?thesis by simp
next
  case False
  hence 0 < y1 + quote-gross P1 sqp1
    by (metis assms(6) clmm-quote-gross-pos cmb-P1 diff-add-cancel
       less-eq-real-def)
  show ?thesis
    by (smt (z3) ‹0 < y1 + quote-gross P1 sqp1› assms clmm-quote-gross-pos
       quote-reach-le-gross quote-gross-grd-max-max clmm-quote-reach-ge
       quote-reach-leq-grd-max liq-grd-min cmb-P1 cmb-nz1 combined-P-prop)
qed
```

lemma *combo-joint-quote-gross-decomp-leq*:
assumes $0 < y$
and $0 < sqp1$
and $y + quote\text{-}gross\ P\ sqp1 \leq quote\text{-}gross\ P\ (grd\text{-}max\ P)$
and $sqp' = quote\text{-}reach\ P\ (y + quote\text{-}gross\ P\ sqp1)$
and $y1 = quote\text{-}gross\ P1\ sqp' - quote\text{-}gross\ P1\ sqp1$
and $P'' = slice\text{-}pool\ P2\ sqp2$
and $sqp1 \leq sqp2$
and $y2' = quote\text{-}gross\ P''\ sqp'$
shows $y = y1 + y2'\ y1 \leq y\ 0 \leq y1$
$y1 + quote\text{-}gross\ P1\ sqp1 \leq quote\text{-}gross\ P1\ (grd\text{-}max\ P1)$
$y2' \leq quote\text{-}gross\ P''\ (grd\text{-}max\ P2)$
proof −
have $quote\text{-}gross\ P''\ sqp1 = 0$
by (*smt* (*verit*) *assms*(2) *assms*(6) *assms*(7) *sqp-lt-grd-max-imp-idx*
clmm-quote-gross-grd-min-le cmb-P2 cmb-grd-eq cmb-max cmb-nz2
cmb-on-grid grd-lower-tick-cong slice-pool-clmm-dsc slice-pool-grd-min)
hence *eq*: $y2' = quote\text{-}gross\ P''\ sqp' - quote\text{-}gross\ P''\ sqp1$ **using** *assms* **by** *simp*
thus $y = y1 + y2'$ **using** *assms combo-joint-quote-gross-decomp* **by** *blast*
show $y1 \leq y$ **using** *assms combo-joint-quote-gross-decomp*(2) **by** *blast*
show $y1 + quote\text{-}gross\ P1\ sqp1 \leq quote\text{-}gross\ P1\ (grd\text{-}max\ P1)$
using *assms combo-joint-quote-gross-decomp*(4) **by** *blast*
show $y2' \leq quote\text{-}gross\ P''\ (grd\text{-}max\ P2)$
using *eq assms combo-joint-quote-gross-decomp*(5) **by** *blast*
show $0 \leq y1$
using *eq assms combo-joint-quote-gross-decomp*(3) **by** *blast*
qed

lemma *combo-quote-swap-slice-eq*:
assumes $0 < sqp1$
and $0 < y$
and $y + quote\text{-}gross\ P\ sqp1 \leq quote\text{-}gross\ P\ (grd\text{-}max\ P)$
and $sqp' = quote\text{-}reach\ P\ (y + quote\text{-}gross\ P\ sqp1)$
and $y1 = quote\text{-}gross\ P1\ sqp' - quote\text{-}gross\ P1\ sqp1$
shows $quote\text{-}swap\ P\ sqp1\ y = quote\text{-}swap\ P1\ sqp1\ y1 +$
$quote\text{-}swap\ (slice\text{-}pool\ P2\ sqp2)\ sqp1\ (y - y1)$
proof −
have *clmm-dsc P* **using** *combined-P-prop* **by** *simp*
have $nz\text{-}support\ (lq\ P) \neq \{\}$ **using** *combined-P-prop* **by** *simp*
have $quote\text{-}gross\ P\ sqp1 < quote\text{-}gross\ P\ (grd\text{-}max\ P)$ **using** *assms* **by** *simp*
hence $sqp1 < grd\text{-}max\ P$
using ‹*clmm-dsc P*› *quote-gross-imp-sqp-lt* **by** *blast*
define $sqp1'$ **where** $sqp1' = quote\text{-}reach\ P1\ (quote\text{-}gross\ P1\ sqp')$
have $quote\text{-}gross\ P\ sqp1 < quote\text{-}gross\ P\ sqp'$
using *quote-reach-add-gt assms* ‹*clmm-dsc P*›
‹$nz\text{-}support\ (lq\ P) \neq \{\}$›
by *simp*

hence *sqp1 < sqp'*
using ‹*clmm-dsc P*› *quote-gross-imp-sqp-lt*[*of P*] **by** *simp*
hence *0 < sqp'*
using *assms liq-grd-min combined-pools-axioms*
unfolding *combined-pools-def* **by** *fastforce*
define *P''* **where** *P'' = slice-pool P2 sqp2*
have *clmm-dsc P''*
using *P''-def assms slice-pool-clmm-dsc cmb-pos* **by** (*simp add*: *cmb-P2*)
have *grd-max P2 = grd-max P''* **using** *slice-pool-grd-max'*
by (*simp add*: *P''-def cmb-P2 cmb-max cmb-nz2 cmb-pos*)
have *nz-support* (*lq P''*) ≠ {} **using** *slice-pool-nz-liq'*
by (*simp add*: *P''-def cmb-P2 cmb-max cmb-nz2 cmb-pos*)
define *y2'* **where** *y2' = quote-gross P'' sqp' − quote-gross P'' sqp1*
define *sqp2'* **where** *sqp2' = quote-reach P''* (*quote-gross P'' sqp'*)
have *quote-gross P sqp1 < quote-gross P sqp'*
using *quote-reach-add-gt assms* ‹*clmm-dsc P*›
‹*nz-support* (*lq P*) ≠ {}›
by *simp*
hence *sqp1 < sqp'*
using ‹*clmm-dsc P*› *quote-gross-imp-sqp-lt*[*of P*] **by** *simp*
have *0 ≤ y2'*
by (*metis* ‹*clmm-dsc P''*› ‹*sqp1 < sqp'*› *quote-gross-imp-sqp-lt*
diff-ge-0-iff-ge eucl-less-le-not-le linorder-less-linear
verit-comp-simplify1(*2*) *y2'-def*)
have *y = y1 + y2'* **using** *assms combo-joint-quote-gross-decomp y2'-def P''-def*
by *blast*
have *quote-swap P sqp1 y = base-net P sqp1 − base-net P sqp'*
using *assms* **unfolding** *quote-swap-def* **by** *simp*
also have *... = base-net P1 sqp1 + base-net P'' sqp1 −*
(*base-net P1 sqp' + base-net P'' sqp'*)
using *assms pool-comb-base-net-plus combined-pools-axioms*
unfolding *combined-pools-def*
by (*metis P''-def* ‹*clmm-dsc P''*› *base-net-join pool-comb-joint-refined*)
also have *... = base-net P1 sqp1 − base-net P1 sqp' +*
base-net P'' sqp1 − base-net P'' sqp'
by *linarith*
also have *... = quote-swap P1 sqp1 y1 + quote-swap P'' sqp1 y2'*
proof −
have *base-net P1 sqp1' = base-net P1 sqp'*
using *quote-reach-base-net*
by (*simp add*: ‹*0 < sqp'*› *cmb-P1 cmb-nz1 sqp1'-def*)
moreover have *base-net P'' sqp2' = base-net P'' sqp'*
using *quote-reach-base-net*
by (*simp add*: ‹*0 < sqp'*› ‹*clmm-dsc P''*› ‹*nz-support* (*lq P''*) ≠ {}› *sqp2'-def*)
ultimately show *?thesis*
unfolding *quote-swap-def*
by (*simp add*: *assms sqp1'-def sqp2'-def y2'-def*)
qed
finally show *?thesis*

```
    using ‹y = y1 + y2'› P''-def by simp
qed

lemma combo-quote-swap-eq:
  assumes 0 < sqp1
  and 0 < y
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
  and sqp' = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
  and sqp1 ≤ sqp2
shows quote-swap P sqp1 y = quote-swap P1 sqp1 y1 +
      quote-swap P2 sqp2 (y − y1)
proof −
  define P'' where P'' = slice-pool P2 sqp2
  define y2' where y2' = quote-gross P'' sqp' − quote-gross P'' sqp1
  have y = y1 + y2' using assms combo-joint-quote-gross-decomp y2'-def P''-def
    by blast
  hence y2' = y − y1 by simp
  have y1 ≤ y using assms combo-joint-quote-gross-decomp(2) by simp
  hence 0 ≤ y2' using ‹y2' = y − y1›  y2'-def by simp
  have quote-swap P sqp1 y = quote-swap P1 sqp1 y1 +
      quote-swap P'' sqp1 y2'
    using assms combo-quote-swap-slice-eq P''-def ‹y2' = y − y1›
    by blast
  also have ... = quote-swap P1 sqp1 y1 +
      quote-swap P2 sqp2 y2'
  proof −
    have quote-swap P'' sqp1 y2' = quote-swap P2 sqp2 y2'
    proof (rule slice-pool-quote-swap)
      show clmm-dsc P2 using cmb-P2 by simp
      show nz-support (lq P2) ≠ {} using cmb-nz2 by simp
      show grd P2 (lower-tick P2 sqp2) = sqp2
        using cmb-P2 cmb-grd-eq cmb-on-grid lower-tick-eq by auto
      show P'' = slice-pool P2 sqp2 using P''-def by simp
      show sqp1 ≤ sqp2 using assms by simp
      show sqp2 < grd-max P2 using cmb-max by simp
      show 0 < sqp1 using assms liq-grd-min cmb-P1 cmb-nz1 by fastforce
      show 0 ≤ y2' using ‹0 ≤ y2'› .
      have y2' ≤ quote-gross P'' (grd-max P2)
        using combo-joint-quote-gross-decomp(5)
        by (simp add: P''-def assms(1−4) y2'-def)
      also have ... = quote-gross P2 (grd-max P2) − quote-gross P2 sqp2
        using P''-def ‹grd P2 (lower-tick P2 sqp2) = sqp2› cmb-P2 assms(1−2)
          slice-pool-quote-gross cmb-max cmb-pos
        by auto
      finally show y2' + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)
        by simp
    qed
    thus ?thesis by simp
```

```
  qed
  finally show ?thesis using ‹y = y1 + y2'› P''-def by simp
qed

lemma comb-add-above-gt:
  assumes 0 < y
  and 0< sqp1
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
  and sqp' = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
  and y1 < y
  and sqp1 ≤ sqp2
shows sqp2 < sqp'
proof −
  define P'' where P'' = slice-pool P2 sqp2
  have y + quote-gross P1 sqp1 = y + quote-gross P sqp1
  proof −
    have quote-gross P sqp1 = quote-gross P1 sqp1
    proof (rule pool-comb-quote-le-slice)
      show grd P1 (lower-tick P1 sqp2) = sqp2
        using cmb-grd-eq cmb-on-grid by auto
      show sqp2 ≤ grd-max P
        using cmb-max pool-comb-grd-max
          by (simp add: cmb-P1 cmb-P2 cmb-comb cmb-grd-eq cmb-nz1 cmb-nz2
cmb-pos)
      show nz-support (lq P) ≠ {}
        using cmb-comb combined-P-prop(2) by auto
    qed (auto simp add: cmb-props assms)
    thus ?thesis by simp
  qed
  also have ... = quote-gross P sqp'
    using clmm-quote-gross-reach-eq assms clmm-quote-gross-pos
      combined-P-prop
    by auto
  also have ... = quote-gross P1 sqp' + quote-gross P'' sqp'
     using pool-comb-quote-decomp P''-def cmb-P1 cmb-P2 cmb-comb cmb-grd-eq
cmb-pos
      cmb-on-grid pool-comb-joint-refined quote-gross-join slice-pool-clmm-dsc
    by simp
  also have ... = y1 +  quote-gross P1 sqp1 + quote-gross P'' sqp'
  proof −
    have quote-gross P1 sqp' = y1 +  quote-gross P1 sqp1
      using clmm-quote-gross-reach-eq assms by simp
    thus ?thesis by simp
  qed
  finally have y + quote-gross P1 sqp1 = y1 +  quote-gross P1 sqp1 +
     quote-gross P'' sqp' .
  hence quote-gross P'' sqp' = y − y1 by simp
  hence 0 < quote-gross P'' sqp' using assms by simp
```

hence *grd-min P'' < sqp'*
by (*metis P''-def cmb-P2 cmb-max cmb-nz2 cmb-pos quote-gross-pos-gt-grd-min slice-pool-clmm-dsc slice-pool-nz-liq'*)
moreover have *sqp2 ≤ grd-min P''*
unfolding *P''-def* **using** *slice-pool-grd-min*
by (*metis cmb-P2 cmb-max cmb-nz2 cmb-pos*)
ultimately show *?thesis* **by** *simp*
qed

lemma *comb-add-above-add-eq*:
assumes *y1 = quote-gross P1 sqp' − quote-gross P1 sqp1*
and *rs1 = quote-reach P1 (y1 + quote-gross P1 sqp1)*
shows *quote-gross P1 sqp' = quote-gross P1 rs1*
by (*simp add*: *assms clmm-quote-gross-pos quote-gross-grd-max-max clmm-quote-gross-reach-eq cmb-P1 cmb-nz1*)

lemma *comb-add-above-add-eq2*:
assumes *0 < y*
and *grd-min P1 ≤ sqp1*
and *y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)*
and *sqp' = quote-reach P (y + quote-gross P sqp1)*
and *y1 = quote-gross P1 sqp' − quote-gross P1 sqp1*
and *sqp1 ≤ sqp2*
and *y1 < y*
and *rs1' = quote-reach P2 (y − y1 + quote-gross P2 sqp2)*
shows *quote-gross P2 sqp' = quote-gross P2 rs1'*
using *combo-remain-quote-eq comb-add-above-gt liq-grd-min combined-P-prop liq-grd-min cmb-props combined-P-prop*
by (*smt* (*verit*) *assms*(*3−8*) *clmm-quote-gross-pos quote-gross-grd-max-max clmm-quote-gross-reach-eq joint-quote-gross-decomp' lower-tick-eq pool-comb-grd-max-slice pool-comb-joint-refined pool-comb-quote-le-slice slice-pool-clmm-dsc slice-pool-quote-gross*)

lemma *combo-joint-rest-qty-slice*:
assumes *0 < y*
and *0 < sqp1*
and *sqp1 ≤ sqp2*
and *y + quote-gross P sqp1 < quote-gross P (grd-max P)*
and *sqp' = quote-reach P (y + quote-gross P sqp1)*
and *y1 = quote-gross P1 sqp' − quote-gross P1 sqp1*
and *P'' = slice-pool P2 sqp2*
shows *y − y1 = quote-gross P'' sqp'*
by (*smt* (*verit*, *ccfv-SIG*) *assms combo-joint-quote-gross-decomp-leq*(*1*) *combined-pools-axioms*)

lemma *combo-joint-rest-qty*:
assumes *0 < y*
and *0 < sqp1*

and *sqp1* ≤ *sqp2*
and *y* + *quote-gross P sqp1* < *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*)
and *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1*
and *sqp2* ≤ *sqp′*
shows *y* − *y1* = *quote-gross P2 sqp′* − *quote-gross P2 sqp2*
proof −
define *P″* **where** *P″* = *slice-pool P2 sqp2*
have *y* − *y1* = *quote-gross P″ sqp′*
using *assms P″-def combo-joint-rest-qty-slice* **by** *simp*
also have ... = *quote-gross P2 sqp′* − *quote-gross P2 sqp2*
using *P″-def slice-pool-quote-gross assms*(7) *cmb-P2 cmb-grd-eq cmb-on-grid cmb-pos lower-tick-eq*
by *auto*
finally show *?thesis* **.**
qed

lemma *combo-joint-rest-qty-le*:
assumes *0* < *y*
and *0* < *sqp1*
and *sqp1* ≤ *sqp2*
and *y* + *quote-gross P sqp1* < *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*)
and *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1*
shows *y* − *y1* + *quote-gross P2 sqp2* ≤ *quote-gross P2* (*grd-max P2*)
proof −
define *P″* **where** *P″* = *slice-pool P2 sqp2*
have *y* − *y1* ≤ *quote-gross P″* (*grd-max P2*)
proof (*rule combo-joint-quote-gross-decomp-leq*(5))
show *0* < *y* **using** *assms* **by** *simp*
show *0* < *sqp1* **using** *assms grd-min-pos cmb-P1 cmb-nz1* **by** *fastforce*
show *y* − *y1* = *quote-gross P″ sqp′*
using *combo-joint-rest-qty-slice assms P″-def* **by** *simp*
show *y* + *quote-gross P sqp1* ≤ *quote-gross P* (*grd-max P*) **using** *assms* **by** *simp*
show *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*) **using** *assms* **by** *simp*
show *sqp1* ≤ *sqp2* **using** *assms* **by** *simp*
show *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1* **using** *assms* **by** *simp*
show *P″* = *slice-pool P2 sqp2* **using** *P″-def* **by** *simp*
qed
also have ... = *quote-gross P2* (*grd-max P2*) − *quote-gross P2 sqp2*
using *P″-def cmb-P2 cmb-grd-eq cmb-max cmb-on-grid cmb-pos lower-tick-eq slice-pool-quote-gross* **by** *simp*
finally have *y* − *y1* ≤ *quote-gross P2* (*grd-max P2*) − *quote-gross P2 sqp2* **.**
thus *?thesis* **by** *simp*
qed

lemma *combo-joint-rest-price-pos*:
assumes *0* < *y*

and *grd-min P1* $\leq$ *sqp1*
and *y* + *quote-gross P sqp1* $\leq$ *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*)
and *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1*
and *rs1′* = *quote-reach P2* (*y* − *y1* + *quote-gross P2 sqp2*)
and *y1* < *y*
shows *0* < *rs1′*
using *clmm-quote-reach-pos*
by (*metis* (*no-types*, *opaque-lifting*) *add-strict-increasing assms clmm-quote-gross-pos liq-grd-min cmb-P1 cmb-P2 cmb-nz1 cmb-nz2 combo-joint-quote-gross-leq-max diff-gt-0-iff-gt less-add-same-cancel1 less-eq-real-def*)

lemma *combo-joint-quote-gross-price-le′*:
assumes *0* < *y*
and *grd-min P1* $\leq$ *sqp1*
and *sqp1* $\leq$ *sqp2*
and *y* + *quote-gross P sqp1* $\leq$ *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*)
and *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1*
and *y1* < *y*
and *rs1′* = *quote-reach P2* (*y* − *y1* + *quote-gross P2 sqp2*)
shows *rs1′* $\leq$ *sqp′*
proof (*rule clmm-quote-reach-le*)
show *clmm-dsc P2* **using** *cmb-P2* **.**
show *nz-support* (*lq P2*) $\neq$ {} **using** *cmb-nz2* **.**
show *0* < *y* − *y1* + *quote-gross P2 sqp2*
by (*simp add*: *add-pos-nonneg assms*(*7*) *clmm-quote-gross-pos cmb-P2*)
show *rs1′* = *quote-reach P2* (*y* − *y1* + *quote-gross P2 sqp2*)
using *assms* **by** *simp*
have *primeq*: *quote-gross P2 sqp′* = *quote-gross P2 rs1′*
using *assms comb-add-above-add-eq2* **by** *simp*
have *q1′*: *quote-gross P2 rs1′* = *y* − *y1* + *quote-gross P2 sqp2*
using *clmm-quote-gross-reach-eq assms*
clmm-quote-gross-pos cmb-P2 cmb-nz2
by (*smt* (*verit*, *best*) *liq-grd-min cmb-P1 cmb-nz1*
combo-joint-quote-gross-leq-max combined-pools-axioms)
show *sqp′* $\in$ *quote-gross P2* −‘ {*y* − *y1* + *quote-gross P2 sqp2*}
using *primeq q1′* **by** *simp*
qed

lemma *comb-add-above-price1-leq*:
assumes *0* < *y*
and *grd-min P1* $\leq$ *sqp1*
and *y* + *quote-gross P sqp1* $\leq$ *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*)
and *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1*
and *sqp1* $\leq$ *sqp2*
and *0* $\leq$ *y2*

```
  and y − y2 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)
  and y2 < y1
  and y1 < y
  and rs1 = quote-reach P1 (y1 + quote-gross P1 sqp1)
  and rs2 = quote-reach P1 (y2 + quote-gross P1 sqp1)
shows rs2 ≤ rs1
proof (rule quote-reach-mono)
  have q1: quote-gross P1 rs1 = y1 + quote-gross P1 sqp1
    using clmm-quote-gross-reach-eq
    by (simp add: assms clmm-quote-gross-pos quote-gross-grd-max-max
        cmb-P1 cmb-nz1)
  show clmm-dsc P1 using cmb-P1 by simp
  show nz-support (lq P1) ≠ {} using cmb-nz1 by simp
  show y2 + quote-gross P1 sqp1 ≤ y1 + quote-gross P1 sqp1 using assms by
simp
  show y1 + quote-gross P1 sqp1 ≤ quote-gross P1 (grd-max P1)
    by (metis quote-gross-grd-max-max cmb-P1 cmb-nz1 q1)
  show 0 ≤ y2 + quote-gross P1 sqp1 using assms
    by (simp add: clmm-quote-gross-pos cmb-P1)
qed (simp add: assms)+

lemma comb-add-above-price2-geq:
  assumes 0 < y
  and grd-min P1 ≤ sqp1
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
  and sqp' = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
  and sqp1 ≤ sqp2
  and 0 ≤ y2
  and y − y2 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)
  and y2 < y1
  and y1 < y
  and rs1' = quote-reach P2 (y − y1 + quote-gross P2 sqp2)
  and rs2' = quote-reach P2 (y − y2 + quote-gross P2 sqp2)
shows rs1' ≤ rs2'
proof (rule quote-reach-mono)
  show clmm-dsc P2 using cmb-P2 by simp
  show nz-support (lq P2) ≠ {} using cmb-nz2 by simp
  have 0 ≤ y − y1 using assms by simp
  thus 0 ≤ y − y1 + quote-gross P2 sqp2
    by (simp add: clmm-quote-gross-pos cmb-P2)
  show y − y1 + quote-gross P2 sqp2 ≤ y − y2 + quote-gross P2 sqp2
    using assms by simp
  show y − y2 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)
    using assms by simp
qed (simp add: assms)+

lemma comb-add-above-price2-geq':
  assumes 0 < y
```

and *grd-min P1* ≤ *sqp1*
and *y* + *quote-gross P sqp1* ≤ *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*)
and *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1*
and *sqp1* ≤ *sqp2*
and *0* ≤ *y2*
and *y* − *y1* + *quote-gross P2 sqp2* ≤ *quote-gross P2* (*grd-max P2*)
and *y1* < *y2*
and *y2* ≤ *y*
and *rs1′* = *quote-reach P2* (*y* − *y1* + *quote-gross P2 sqp2*)
and *rs2′* = *quote-reach P2* (*y* − *y2* + *quote-gross P2 sqp2*)
shows *rs2′* ≤ *rs1′*
proof (*rule quote-reach-mono*)
show *clmm-dsc P2* **using** *cmb-P2* **by** *simp*
show *nz-support* (*lq P2*) ≠ {} **using** *cmb-nz2* **by** *simp*
have *0* ≤ *y* − *y2* **using** *assms* **by** *simp*
thus *0* ≤ *y* − *y2* + *quote-gross P2 sqp2*
by (*simp add*: *clmm-quote-gross-pos cmb-P2*)
show *y* − *y2* + *quote-gross P2 sqp2* ≤ *y* − *y1* + *quote-gross P2 sqp2*
using *assms* **by** *simp*
show *y* − *y1* + *quote-gross P2 sqp2* ≤ *quote-gross P2* (*grd-max P2*)
using *assms* **by** *simp*
qed (*simp add*: *assms*)+

lemma *comb-add-above-price2-lt*:
assumes *0* < *y*
and *grd-min P1* ≤ *sqp1*
and *y* + *quote-gross P sqp1* ≤ *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*)
and *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1*
and *sqp1* ≤ *sqp2*
and *0* ≤ *y2*
and *y* − *y2* + *quote-gross P2 sqp2* ≤ *quote-gross P2* (*grd-max P2*)
and *y2* < *y1*
and *y1* < *y*
and *rs2′* = *quote-reach P2* (*y* − *y2* + *quote-gross P2 sqp2*)
shows *sqp′* < *rs2′*
proof (*rule lt-quote-gross-imp-up-price*)
define *rs1′* **where** *rs1′* = *quote-reach P2* (*y* − *y1* + *quote-gross P2 sqp2*)
have *primeq*: *quote-gross P2 sqp′* = *quote-gross P2 rs1′*
using *assms comb-add-above-add-eq2 rs1′-def* **by** *simp*
have *q1′*: *quote-gross P2 rs1′* = *y* − *y1* + *quote-gross P2 sqp2*
unfolding *rs1′-def*
using *clmm-quote-gross-reach-eq assms*(*8*−*10*)
clmm-quote-gross-pos cmb-P2 cmb-nz2 **by** *auto*
show *clmm-dsc P2* **using** *cmb-P2* **.**
show *nz-support* (*lq P2*) ≠ {} **using** *cmb-nz2* **.**
show *0* < *y* − *y2* + *quote-gross P2 sqp2*
by (*metis add.commute add-strict-increasing2 assms*(*10*) *assms*(*9*)

clmm-quote-gross-pos cmb-P2 diff-add-cancel less-add-same-cancel1 pos-add-strict)
show *y − y2 + quote-gross P2 sqp2 ≤ quote-gross P2* (*grd-max P2*)
using *assms* **by** *simp*
show *quote-gross P2 sqp′ < y − y2 + quote-gross P2 sqp2*
by (*simp add*: *assms*(*9*) *primeq q1′*)
show *rs2′ = quote-reach P2* (*y − y2 + quote-gross P2 sqp2*)
using *assms* **by** *simp*
qed

lemma *combo-joint-reached-price-pos*:
assumes *0 < y*
and *y + quote-gross P sqp1 ≤ quote-gross P* (*grd-max P*)
and *sqp′ = quote-reach P* (*y + quote-gross P sqp1*)
shows *0 < sqp′* **using** *clmm-quote-reach-pos*
using *assms clmm-quote-gross-pos combined-P-prop* **by** *auto*

lemma *combo-joint-base-reached-eq*:
assumes *0 < y*
and *grd-min P1 ≤ sqp1*
and *y + quote-gross P sqp1 ≤ quote-gross P* (*grd-max P*)
and *sqp′ = quote-reach P* (*y + quote-gross P sqp1*)
and *y1 = quote-gross P1 sqp′ − quote-gross P1 sqp1*
and *sqp1 ≤ sqp2*
and *rs1 = quote-reach P1* (*y1 + quote-gross P1 sqp1*)
shows *base-net P1 sqp′ = base-net P1 rs1*
proof (*rule quote-gross-equiv-base-net*[*symmetric*])
show *quote-gross P1 rs1 = quote-gross P1 sqp′*
using *assms comb-add-above-add-eq* **by** *metis*
show *clmm-dsc P1* **using** *cmb-P1* **.**
show *0 < rs1* **using** *clmm-quote-reach-pos*
by (*simp add*: *assms*(*5*) *assms*(*7*) *clmm-quote-gross-pos quote-gross-grd-max-max cmb-P1 cmb-nz1*)
show *rs1 ≤ sqp′*
using *assms combo-joint-quote-gross-price-le* **by** *blast*
qed

lemma *combo-joint-base-reached-eq2*:
assumes *0 < y*
and *grd-min P1 ≤ sqp1*
and *y + quote-gross P sqp1 ≤ quote-gross P* (*grd-max P*)
and *sqp′ = quote-reach P* (*y + quote-gross P sqp1*)
and *y1 = quote-gross P1 sqp′ − quote-gross P1 sqp1*
and *sqp1 ≤ sqp2*
and *y1 < y*
and *rs1′ = quote-reach P2* (*y − y1 + quote-gross P2 sqp2*)
shows *base-net P2 sqp′ = base-net P2 rs1′*
proof −
have *quoteq*: *quote-gross P2 sqp′ = quote-gross P2 rs1′*

```
    using assms comb-add-above-add-eq2 by simp
  show ?thesis
  proof (cases rs1' ≤ sqp')
    case True
    show ?thesis
    proof (rule quote-gross-equiv-base-net[symmetric])
      show clmm-dsc P2 using cmb-P2 .
      show rs1' ≤ sqp' using True .
      show quote-gross P2 rs1' = quote-gross P2 sqp' using quoteq by simp
      show 0 < rs1' using combo-joint-rest-price-pos assms by simp
    qed
  next
    case False
    show ?thesis
    proof (rule quote-gross-equiv-base-net)
      show clmm-dsc P2 using cmb-P2 .
      show sqp' ≤ rs1' using False by simp
      show quote-gross P2 sqp' = quote-gross P2 rs1' using quoteq by simp
      show 0 < sqp' using combo-joint-reached-price-pos assms by simp
    qed
  qed
qed

lemma quote-gross-price-eq1:
  assumes y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
  and rs1 = quote-reach P1 (y1 + quote-gross P1 sqp1)
shows quote-gross P1 rs1 = y1 + quote-gross P1 sqp1
    using clmm-quote-gross-reach-eq
    by (simp add: assms clmm-quote-gross-pos quote-gross-grd-max-max
        cmb-P1 cmb-nz1)

lemma quote-gross-price-eq2:
  assumes 0 ≤ y2
  and y2 + quote-gross P1 sqp1 ≤ quote-gross P1 (grd-max P1)
  and rs2 = quote-reach P1 (y2 + quote-gross P1 sqp1)
shows quote-gross P1 rs2 = y2 + quote-gross P1 sqp1
  using clmm-quote-gross-reach-eq
  by (simp add: assms clmm-quote-gross-pos cmb-P1 cmb-nz1)

end
```

6.5 Optimality result on quote tokens

When the fees in two pools are constant and equal, swapping a given amount of quote tokens in their combination permits to determine the optimal quantities of quote tokens to swap in each individual pool.

```
locale combined-pools-cst-fee = combined-pools +
  fixes phi
  assumes fee1: ∀ i. fee P1 i = phi
```

and *fee2*: $\forall\, i.$ *fee P2 i* = *phi*

begin

lemma *fee-props*:
shows *0* ≤ *phi phi* < *1* **using** *cmb-P1 clmm-dsc-fees*[*of P1*] *fee1* **by** *auto*

lemma *quote-swap-opt-above*:
assumes *0* < *y*
and *grd-min P1* ≤ *sqp1*
and *y* + *quote-gross P sqp1* ≤ *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*)
and *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1*
and *sqp1* ≤ *sqp2*
and *0* ≤ *y2*
and *y* − *y2* + *quote-gross P2 sqp2* ≤ *quote-gross P2* (*grd-max P2*)
and *y2* < *y1*
and *y1* < *y*
shows *quote-swap P1 sqp1 y2* + *quote-swap P2 sqp2* (*y* − *y2*) ≤ *quote-swap P sqp1 y*
proof −
define *rs1* **where** *rs1* = *quote-reach P1* (*y1* + *quote-gross P1 sqp1*)
define *rs2* **where** *rs2* = *quote-reach P1* (*y2* + *quote-gross P1 sqp1*)
define *rs1′* **where** *rs1′* = *quote-reach P2* (*y* − *y1* + *quote-gross P2 sqp2*)
define *rs2′* **where** *rs2′* = *quote-reach P2* (*y* − *y2* + *quote-gross P2 sqp2*)
have *q1*: *quote-gross P1 rs1* = *y1* + *quote-gross P1 sqp1*
unfolding *rs1-def* **using** *quote-gross-price-eq1 assms* **by** *simp*
have *q2*: *quote-gross P1 rs2* = *y2* + *quote-gross P1 sqp1*
unfolding *rs2-def* **using** *quote-gross-price-eq2*
by (*smt* (*verit*) *assms*(*7*) *assms*(*9*) *clmm-quote-gross-pos quote-gross-grd-max-max cmb-P1 cmb-nz1 q1*)
have *q2′*: *quote-gross P2 rs2′* = *y* − *y2* + *quote-gross P2 sqp2*
unfolding *rs2′-def*
using *clmm-quote-gross-reach-eq assms*(*8*−*10*)
clmm-quote-gross-pos cmb-P2 cmb-nz2 **by** *auto*
have *q1′*: *quote-gross P2 rs1′* = *y* − *y1* + *quote-gross P2 sqp2*
unfolding *rs1′-def*
using *clmm-quote-gross-reach-eq assms*(*8*−*10*)
clmm-quote-gross-pos cmb-P2 cmb-nz2 **by** *auto*
have *primeq*: *quote-gross P2 sqp′* = *quote-gross P2 rs1′*
using *assms comb-add-above-add-eq2 rs1′-def* **by** *simp*
have *rs1* ≤ *sqp′*
using *assms rs1-def combo-joint-quote-gross-price-le* **by** *simp*
have *0* < *sqp′* **using** *combo-joint-reached-price-pos assms* **by** *simp*
have *0* < *grd-min P1*
using *assms grd-min-pos liq-grd-min cmb-P1 cmb-nz1* **by** *blast*
have *sqp1* ≤ *rs1* **using** *quote-reach-strict-mono*
by (*metis assms*(*7*) *assms*(*9*) *quote-gross-imp-sqp-lt cmb-P1 less-add-same-cancel2 linorder-not-less nle-le order.strict-trans q1*)

hence *0 < rs1* **using** ‹*0 < grd-min P1*› *assms* **by** *simp*
hence *1/sqp′ ≤ 1/rs1* **using** ‹*rs1 ≤ sqp′*› ‹*0 < sqp′*› **by** (*simp add*: *frac-le*)
have *rs2 ≤ rs1* **using** *assms rs2-def rs1-def comb-add-above-price1-leq* **by** *simp*
have *rs1′ ≤ rs2′*
using *assms rs1′-def rs2′-def comb-add-above-price2-geq* **by** *simp*
have *sqp′ < rs2′* **using** *assms rs2′-def comb-add-above-price2-lt* **by** *simp*
have *(1 −phi) ∗ (quote-gross P1 rs1 − quote-gross P1 rs2)/(rs1 ∗ rs1) −*
(1 − phi) ∗ (quote-gross P2 rs2′ − quote-gross P2 sqp′)/(sqp′ ∗ sqp′) =
(1 − phi) ∗ (y1 − y2)/(rs1 ∗ rs1) −
(1 − phi) ∗ (quote-gross P2 rs2′ − quote-gross P2 sqp′)/(sqp′ ∗ sqp′)
using *q1 q2* **by** *simp*
also have *... =*
(1 − phi) ∗ (y1 + quote-gross P1 sqp1 − (y2 + quote-gross P1 sqp1))/(rs1 ∗ rs1) −
(1 − phi) ∗ (quote-gross P2 rs2′ − quote-gross P2 sqp′)/(sqp′ ∗ sqp′)
by *simp*
also have *... =*
(1 − phi) ∗ (y1 − y2)/(rs1 ∗ rs1) −
(1 − phi) ∗ (quote-gross P2 rs2′ − quote-gross P2 rs1′)/(sqp′ ∗ sqp′)
using *primeq* **by** *simp*
also have *... = (1 − phi) ∗ (y1 − y2)/(rs1 ∗ rs1) −*
(1 − phi) ∗ (y1 − y2)/(sqp′ ∗ sqp′)
using *q1′ q2′* **by** *simp*
also have *... = (1 − phi) ∗ (y1 − y2) ∗ (1/(rs1 ∗ rs1) − 1/(sqp′ ∗ sqp′))*
by (*simp add*: *vector-space-over-itself.scale-right-diff-distrib*)
finally have *(1 −phi) ∗ (quote-gross P1 rs1 − quote-gross P1 rs2)/(rs1 ∗ rs1) −*
(1 − phi) ∗ (quote-gross P2 rs2′ − quote-gross P2 sqp′)/(sqp′ ∗ sqp′) =
(1 − phi) ∗ (y1 − y2) ∗ (1/(rs1 ∗ rs1) − 1/(sqp′ ∗ sqp′)) **.**
moreover have *0 ≤ (1 − phi) ∗ (y1 − y2) ∗ (1/(rs1 ∗ rs1) − 1/(sqp′ ∗ sqp′))*
proof −
have *rs1 ∗ rs1 ≤ sqp′ ∗ sqp′*
using ‹*0 < rs1*› ‹*rs1 ≤ sqp′*› **by** (*simp add*: *mult-mono′*)
hence *0 ≤ 1/(rs1 ∗ rs1) − 1/(sqp′ ∗ sqp′)*
by (*simp add*: ‹*0 < rs1*› *frac-le*)
thus *?thesis*
using *assms*(*9*) *fee-props*(*2*) **by** *fastforce*
qed
ultimately have *0 ≤ (1 −phi) ∗*
(quote-gross P1 rs1 − quote-gross P1 rs2)/(rs1 ∗ rs1) −
(1 − phi) ∗ (quote-gross P2 rs2′ − quote-gross P2 sqp′)/(sqp′ ∗ sqp′)
by *simp*
also have *... ≤ base-net P1 rs2 − base-net P1 rs1 −*
(1 − phi) ∗ (quote-gross P2 rs2′ − quote-gross P2 sqp′)/(sqp′ ∗ sqp′)
proof −
have *(1 −phi) ∗ (quote-gross P1 rs1 − quote-gross P1 rs2)/(rs1 ∗ rs1) ≤*
base-net P1 rs2 − base-net P1 rs1
proof (*rule base-net-quote-lbound*)
show *clmm-dsc P1* **using** *cmb-P1* **.**

```
      show ⋀i. fee P1 i = phi by (simp add: fee1)
      show 0 < rs2 using clmm-quote-reach-pos
        by (metis clmm-quote-gross-pos quote-gross-grd-max-max cmb-P1
            cmb-nz1 q2 rs2-def)
      show rs2 ≤ rs1 using ‹rs2 ≤ rs1› .
    qed
    thus ?thesis by simp
  qed
  also have ... ≤ base-net P1 rs2 − base-net P1 rs1 −
      (base-net P2 sqp' − base-net P2 rs2')
  proof −
    have base-net P2 sqp' − base-net P2 rs2' ≤
        (1 − phi) ∗ (quote-gross P2 rs2' − quote-gross P2 sqp')/(sqp' ∗ sqp')
    proof (rule base-net-quote-ubound)
      show clmm-dsc P2 using cmb-P2 .
      show phi < 1 using fee-props(2) .
      show sqp' ≤ rs2' using ‹sqp' < rs2'› by simp
      show ⋀i. fee P2 i = phi by (simp add: fee2)
      show 0 < sqp' using ‹0 < sqp'› .
    qed
    thus ?thesis by (simp add: diff-left-mono)
  qed
  also have ... = base-net P1 rs2 − base-net P1 rs1 −
      (base-net P2 rs1' − base-net P2 rs2')
  proof −
    have base-net P2 sqp' = base-net P2 rs1'
      using assms combo-joint-base-reached-eq2 order-less-imp-le rs1'-def by blast
    thus ?thesis by simp
  qed
  also have ... = quote-swap P1 sqp1 y1 − quote-swap P1 sqp1 y2−
      (quote-swap P2 sqp2 (y − y2) − quote-swap P2 sqp2 (y − y1))
    unfolding quote-swap-def rs1-def rs2-def rs1'-def rs2'-def by simp
  also have ... = quote-swap P sqp1 y −
      (quote-swap P1 sqp1 y2 + quote-swap P2 sqp2 (y − y2))
    using assms combo-quote-swap-eq ‹0 < grd-min P1› by simp
  finally have 0 ≤ quote-swap P sqp1 y −
      (quote-swap P1 sqp1 y2 + quote-swap P2 sqp2 (y − y2)) .
  thus ?thesis by simp
qed

lemma quote-swap-opt-above':
  assumes 0 < y
  and grd-min P1 ≤ sqp1
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
  and sqp' = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
  and sqp1 ≤ sqp2
  and y2 + quote-gross P1 sqp1 ≤ quote-gross P1 (grd-max P1)
  and 0 ≤ y − y2
```

and *y1* < *y2*
and *y2* ≤ *y*
shows *quote-swap P1 sqp1 y2 + quote-swap P2 sqp2 (y − y2)* ≤ *quote-swap P sqp1 y*
proof −
define *lp1* **where** *lp1 = quote-reach P1 (quote-gross P1 sqp1)*
have *qlp*: *quote-gross P1 lp1 = quote-gross P1 sqp1*
by (*simp add*: *clmm-quote-gross-pos quote-gross-grd-max-max clmm-quote-gross-reach-eq cmb-P1 cmb-nz1 lp1-def*)
have *lpgeq*: *grd-min P1* ≤ *lp1*
by (*simp add*: *clmm-quote-gross-pos quote-gross-grd-max-max clmm-quote-reach-ge cmb-P1 cmb-nz1 lp1-def*)
define *rs1* **where** *rs1 = quote-reach P1 (y1 + quote-gross P1 lp1)*
define *rs2* **where** *rs2 = quote-reach P1 (y2 + quote-gross P1 lp1)*
define *rs1′* **where** *rs1′ = quote-reach P2 (y − y1 + quote-gross P2 sqp2)*
define *rs2′* **where** *rs2′ = quote-reach P2 (y − y2 + quote-gross P2 sqp2)*
have *0* < *grd-min P1*
using *assms grd-min-pos liq-grd-min cmb-P1 cmb-nz1* **by** *blast*
have *q′*: *quote-gross P1 sqp′ = y1 + quote-gross P1 sqp1* **using** *assms* **by** *simp*
have *q1*: *quote-gross P1 rs1 = y1 + quote-gross P1 sqp1*
unfolding *rs1-def* **using** *quote-gross-price-eq1 assms qlp* **by** *simp*
have *q2*: *quote-gross P1 rs2 = y2 + quote-gross P1 sqp1*
unfolding *rs2-def* **using** *quote-gross-price-eq2 qlp*
by (*metis* ‹*0* < *grd-min P1*› *assms*(*1*−*5*) *assms*(*7*) *assms*(*9*) *combo-joint-quote-gross-decomp*(*3*) *leD nless-le order.trans*)
have *quote-gross P1 sqp′* < *quote-gross P1 rs2* **using** *q′ q2 assms* **by** *simp*
have *y − y1 + quote-gross P2 sqp2* ≤ *quote-gross P2 (grd-max P2)*
using ‹*0* < *grd-min P1*› *assms*(*1*−*5*) *combined-pools.combo-joint-quote-gross-leq-max combined-pools-axioms*
by *auto*
hence *q2′*: *quote-gross P2 rs2′ = y − y2 + quote-gross P2 sqp2*
unfolding *rs2′-def*
using *clmm-quote-gross-reach-eq assms*(*8*−*10*) *clmm-quote-gross-pos cmb-P2 cmb-nz2* **by** *auto*
have *q1′*: *quote-gross P2 rs1′ = y − y1 + quote-gross P2 sqp2*
unfolding *rs1′-def*
using *clmm-quote-gross-reach-eq assms*(*8*−*10*) *clmm-quote-gross-pos cmb-P2 cmb-nz2*
by (*simp add*: ‹*y − y1 + quote-gross P2 sqp2* ≤ *quote-gross P2 (grd-max P2)*›)
have *quoteq*: *quote-gross P2 sqp′ = quote-gross P2 rs1′*
using *assms comb-add-above-add-eq2 rs1′-def* **by** *simp*
have *rs1* ≤ *sqp′*
using *assms rs1-def combo-joint-quote-gross-price-le qlp* **by** *force*
have *rs1′* ≤ *sqp′* **using** *combo-joint-quote-gross-price-le′ assms rs1′-def* **by** *simp*
have *0* < *sqp′* **using** *combo-joint-reached-price-pos assms* **by** *simp*
have *0* < *rs1′* **using** *assms rs1′-def*
by (*metis quote-gross-imp-sqp-lt cmb-P2 cmb-pos diff-gt-0-iff-gt dual-order.strict-trans less-add-same-cancel2 order-less-le-trans q1′*)
have *baseq2*: *base-net P2 sqp′ = base-net P2 rs1′*

using *assms combo-joint-base-reached-eq2 rs1'-def* **by** *simp*
have *baseq*: *base-net P1 sqp' = base-net P1 rs1*
using *assms combo-joint-base-reached-eq rs1-def qlp* **by** *simp*
have *0 < sqp'* **using** *clmm-quote-reach-pos*
by (*metis assms(1) assms(3) assms(4) clmm-quote-gross-pos combined-P-prop dual-order.trans le-add-same-cancel1 less-eq-real-def*)
have *lp1 ≤ rs1*
proof (*rule quote-reach-mono*)
show *lp1 = quote-reach P1 (quote-gross P1 sqp1)* **using** *lp1-def* **by** *simp*
show *clmm-dsc P1* **using** *cmb-P1* **.**
show *nz-support (lq P1) ≠ {}* **using** *cmb-nz1* **.**
show *0 ≤ quote-gross P1 sqp1* **using** *clmm-quote-gross-pos cmb-P1* **by** *simp*
show *rs1 = quote-reach P1 (y1 + quote-gross P1 lp1)* **using** *rs1-def* **by** *simp*
show *quote-gross P1 sqp1 ≤ y1 + quote-gross P1 lp1*
using *qlp ‹0 < grd-min P1› assms combo-joint-quote-gross-decomp(3)*
by *auto*
show *y1 + quote-gross P1 lp1 ≤ quote-gross P1 (grd-max P1)*
using *qlp assms* **by** *simp*
qed
hence *0 < rs1* **using** *‹0 < grd-min P1› assms lpgeq* **by** *simp*
hence *1/sqp' ≤ 1/rs1* **using** *‹rs1 ≤ sqp'› ‹0 < sqp'›* **by** (*simp add*: *frac-le*)
have *rs1 ≤ rs2* **using** *assms rs2-def rs1-def*
by (*metis add-le-cancel-right quote-gross-imp-sqp-lt cmb-P1 linorder-not-less order.asym q1 q2*)
have *rs2' ≤ rs1'*
proof (*rule comb-add-above-price2-geq'*[*of y sqp1 sqp' y1 y2*])
have *0 ≤ y1*
using *combo-joint-quote-gross-decomp-leq(3) assms*
by (*meson ‹0 < grd-min P1› order-less-imp-le order-less-le-trans*)
thus *0 ≤ y2* **using** *assms* **by** *simp*
show *y − y1 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)*
using *‹y − y1 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)›* **.**
show *y1 < y2* **using** *assms* **by** *simp*
show *y2 ≤ y* **using** *assms* **by** *simp*
show *rs1' = quote-reach P2 (y − y1 + quote-gross P2 sqp2)*
using *rs1'-def* **by** *simp*
show *rs2' = quote-reach P2 (y − y2 + quote-gross P2 sqp2)*
using *rs2'-def* **by** *simp*
qed (*simp add*: *assms*)+
have *0 = (1 − phi) * (y2−y1)/(sqp' * sqp') − (1 − phi) *
(quote-gross P1 rs2 − quote-gross P1 sqp') / (sqp' * sqp')*
using *assms q2* **by** *simp*
also have *... = (1 − phi) * (quote-gross P2 sqp' − quote-gross P2 rs2') /
(sqp' * sqp') − (1 − phi) * (quote-gross P1 rs2 − quote-gross P1 sqp') /
(sqp' * sqp')*
by (*simp add*: *quoteq q1' q2'*)
also have *... ≤ (base-net P2 rs2' − base-net P2 sqp') − (1 − phi) *
(quote-gross P1 rs2 − quote-gross P1 sqp') / (sqp' * sqp')*
proof −

```
    have (1 − phi) ∗ (quote-gross P2 sqp′ − quote-gross P2 rs2′) /
       (sqp′ ∗ sqp′) ≤ base-net P2 rs2′ − base-net P2 sqp′
    proof (rule base-net-quote-lbound[of P2 phi rs2′ sqp′])
      show clmm-dsc P2 using cmb-P2 .
      show ⋀i. fee P2 i = phi using fee2 by simp
      show rs2′ ≤ sqp′ using ‹rs1′ ≤ sqp′› ‹rs2′ ≤ rs1′› by simp
      show 0 < rs2′ using clmm-quote-reach-pos
        by (metis clmm-quote-gross-pos quote-gross-grd-max-max cmb-P2
           cmb-nz2 q2′ rs2′-def)
    qed
    thus ?thesis by simp
  qed
  also have ... ≤ (base-net P2 rs2′ − base-net P2 sqp′) −
     (base-net P1 sqp′ − base-net P1 rs2)
  proof −
    have base-net P1 sqp′ − base-net P1 rs2 ≤ (1 − phi) ∗
       (quote-gross P1 rs2 − quote-gross P1 sqp′) / (sqp′ ∗ sqp′)
    proof (rule base-net-quote-ubound)
      show clmm-dsc P1 using cmb-P1 .
      show ⋀i. fee P1 i = phi using fee1 by simp
      show phi < 1 using fee-props by simp
      show 0 < sqp′ using ‹0 < sqp′› .
      show sqp′ ≤ rs2
        using ‹quote-gross P1 sqp′ < quote-gross P1 rs2›
          quote-gross-imp-sqp-lt cmb-P1
        by fastforce
    qed
    thus ?thesis by simp
  qed
  also have ... = (base-net P2 rs2′ − base-net P2 rs1′) −
     (base-net P1 rs1 − base-net P1 rs2)
    using baseq2 baseq by simp
  also have ... = base-net P1 rs2 − base-net P1 rs1 −
     (base-net P2 rs1′ − base-net P2 rs2′)
    by simp
  also have ... = quote-swap P1 sqp1 y1 − quote-swap P1 sqp1 y2−
     (quote-swap P2 sqp2 (y − y2) − quote-swap P2 sqp2 (y − y1))
    unfolding quote-swap-def rs1-def rs2-def rs1′-def rs2′-def using qlp by simp
  also have ... = quote-swap P sqp1 y −
     (quote-swap P1 sqp1 y2 + quote-swap P2 sqp2 (y − y2))
    using assms combo-quote-swap-eq ‹0 < grd-min P1› by simp
  finally have 0 ≤ quote-swap P sqp1 y −
     (quote-swap P1 sqp1 y2 + quote-swap P2 sqp2 (y − y2)) .
  thus ?thesis by simp
qed

lemma combo-slice-no-addition2:
  assumes 0 < y
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
```

```
  and sqp' = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
  and sqp1 ≤ sqp2
  and y1 = y
  and 0 ≤ y2
  and y2 ≤ y
  and y − y2 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)
  and y1 ≠ y2
  and P'' = slice-pool P2 sqp2
shows quote-gross P'' sqp' = 0
proof −
  have y1 + quote-gross P1 sqp1 = y + quote-gross P sqp1
  proof −
    have quote-gross P sqp1 = quote-gross P1 sqp1
    proof (rule combo-quote-init1)
      show clmm-dsc P1 using cmb-P1 .
      show clmm-dsc P2 using cmb-P2 .
      show grd P1 = grd P2 by (simp add: cmb-grd-eq)
      show 0 < sqp2 by (simp add: cmb-pos)
      show grd P1 (lower-tick P1 sqp2) = sqp2 by (simp add: cmb-on-grid)
      show P = pool-comb P1 P2 sqp2 by (simp add: cmb-comb)
      show 0 < y using assms by simp
      show y + quote-gross P sqp1 ≤ quote-gross P (grd-max P) using assms by
simp
      show nz-support (lq P1) ≠ {} using cmb-nz1 .
      show nz-support (lq P2) ≠ {} using cmb-nz2 .
      show sqp' = quote-reach P (y + quote-gross P sqp1) using assms by simp
      show sqp1 ≤ sqp2 using assms by simp
      show sqp2 < grd-max P2 by (simp add: cmb-max)
    qed
    thus ?thesis using assms by simp
  qed
  also have ... = quote-gross P sqp'
    using clmm-quote-gross-reach-eq assms clmm-quote-gross-pos combined-P-prop
    by auto
  also have ... = quote-gross P1 sqp' + quote-gross P'' sqp'
     using pool-comb-quote-decomp cmb-P1 cmb-P2 cmb-comb cmb-grd-eq assms
cmb-pos
      cmb-on-grid pool-comb-joint-refined quote-gross-join slice-pool-clmm-dsc
    by presburger
  also have ... = y1 + quote-gross P1 sqp1 + quote-gross P'' sqp'
    using assms by simp
  finally have y1 + quote-gross P1 sqp1 =
      y1 + quote-gross P1 sqp1 + quote-gross P'' sqp' .
  thus quote-gross P'' sqp' = 0 by simp
qed

lemma quote-swap-opt-below:
  assumes 0 < y
```

and *grd-min P1* ≤ *sqp1*
and *y* + *quote-gross P sqp1* ≤ *quote-gross P* (*grd-max P*)
and *sqp′* = *quote-reach P* (*y* + *quote-gross P sqp1*)
and *y1* = *quote-gross P1 sqp′* − *quote-gross P1 sqp1*
and *sqp1* ≤ *sqp2*
and *y1* = *y*
and *0* ≤ *y2*
and *y2* ≤ *y*
and *y* − *y2* + *quote-gross P2 sqp2* ≤ *quote-gross P2* (*grd-max P2*)
and *y1* ≠ *y2*
shows *quote-swap P1 sqp1 y2* + *quote-swap P2 sqp2* (*y* − *y2*) ≤ *quote-swap P sqp1 y*
proof −
define *P″* **where** *P″* = *slice-pool P2 sqp2*
have *y2* < *y1* **using** *assms* **by** *simp*
define *lp1* **where** *lp1* = *quote-reach P1* (*quote-gross P1 sqp1*)
have *qlp*: *quote-gross P1 lp1* = *quote-gross P1 sqp1*
by (*simp add*: *clmm-quote-gross-pos quote-gross-grd-max-max clmm-quote-gross-reach-eq cmb-P1 cmb-nz1 lp1-def*)
have *lpgeq*: *grd-min P1* ≤ *lp1*
by (*simp add*: *clmm-quote-gross-pos quote-gross-grd-max-max clmm-quote-reach-ge cmb-P1 cmb-nz1 lp1-def*)
define *rs1* **where** *rs1* = *quote-reach P1* (*y1* + *quote-gross P1 lp1*)
define *rs2* **where** *rs2* = *quote-reach P1* (*y2* + *quote-gross P1 lp1*)
define *rs1′* **where** *rs1′* = *quote-reach P2* (*y* − *y1* + *quote-gross P2 sqp2*)
define *rs2′* **where** *rs2′* = *quote-reach P2* (*y* − *y2* + *quote-gross P2 sqp2*)
define *rs3′* **where** *rs3′* = *quote-reach P″* (*y* − *y2*)
have *0* < *grd-min P1*
using *assms grd-min-pos liq-grd-min cmb-P1 cmb-nz1* **by** *blast*
have *q′*: *quote-gross P1 sqp′* = *y1* + *quote-gross P1 sqp1* **using** *assms* **by** *simp*
have *q1*: *quote-gross P1 rs1* = *y1* + *quote-gross P1 sqp1*
unfolding *rs1-def* **using** *quote-gross-price-eq1 assms qlp* **by** *metis*
have *q2*: *quote-gross P1 rs2* = *y2* + *quote-gross P1 lp1*
unfolding *rs2-def* **using** *clmm-quote-gross-reach-eq*
by (*smt* (*z3*) *assms*(*7*−*9*) *clmm-quote-gross-pos quote-gross-grd-max-max cmb-P1 cmb-nz1 q1 qlp*)
have *q2′*: *quote-gross P2 rs2′* = *y* − *y2* + *quote-gross P2 sqp2*
unfolding *rs2′-def*
using *clmm-quote-gross-reach-eq assms*(*8*−*10*)
clmm-quote-gross-pos cmb-P2 cmb-nz2 **by** *auto*
have *q1′*: *quote-gross P2 rs1′* = *y* − *y1* + *quote-gross P2 sqp2*
unfolding *rs1′-def*
using *clmm-quote-gross-reach-eq assms*(*7*−*10*)
clmm-quote-gross-pos cmb-P2 cmb-nz2 **by** *auto*
have *rs1* ≤ *sqp′*
by (*smt* (*verit*, *ccfv-SIG*) ‹*y2* < *y1*› *clmm-quote-gross-pos clmm-quote-reach-pos cmb-P1 cmb-nz1 q′*
q2 qlp quote-gross-grd-max-max quote-gross-imp-sqp-lt quote-reach-le-gross rs1-def rs2-def)

have *0 < sqp′* **using** *combo-joint-reached-price-pos assms* **by** *simp*
have *0 < grd-min P1*
using *assms grd-min-pos liq-grd-min cmb-P1 cmb-nz1* **by** *blast*
have *sqp1 < rs1* **using** *quote-reach-strict-mono*
by (*metis assms*(*1*) *assms*(*5*) *assms*(*7*) *quote-gross-imp-sqp-lt cmb-P1*
diff-gt-0-iff-gt q′ q1)
hence *0 < rs1* **using** ‹*0 < grd-min P1*› *assms* **by** *simp*
hence *1*/*sqp′ ≤ 1*/*rs1* **using** ‹*rs1 ≤ sqp′*› ‹*0 < sqp′*› **by** (*simp add*: *frac-le*)
have *rs2 ≤ rs1* **using** *assms rs2-def rs1-def* ‹*y2 < y1*›
by (*smt* (*verit*) *quote-gross-imp-sqp-lt cmb-P1 q1 q2 qlp*)
have *quote-gross P2 sqp2 < quote-gross P2 rs2′*
using *q2′* ‹*y2 < y1*› *assms*(*7*) **by** *simp*
hence *sqp2 < rs2′* **using** *quote-gross-imp-sqp-lt cmb-P2* **by** *blast*
have *quote-gross P′′ sqp′ = 0*
using *assms P′′-def combined-pools-cst-fee.combo-slice-no-addition2*
combined-pools-cst-fee-axioms
by *simp*
have *quote-gross P′′ sqp2 = 0* **using** *P′′-def slice-pool-quote-gross*
by (*simp add*: *cmb-P2 cmb-pos*)
have *q3′*: *quote-gross P′′ rs3′ = y − y2* **unfolding** *rs3′-def*
proof (*rule clmm-quote-gross-reach-eq*)
show *clmm-dsc P′′* **using** *P′′-def cmb-P2 slice-pool-clmm-dsc cmb-pos* **by** *simp*
show *nz-support* (*lq P′′*) *≠* {}
using *P′′-def cmb-P2 cmb-max cmb-nz2 cmb-pos slice-pool-nz-liq′* **by** *auto*
have *quote-gross P′′* (*grd-max P′′*) =
quote-gross P2 (*grd-max P2*) *− quote-gross P2 sqp2*
using *slice-pool-quote-gross-max-eq*
by (*metis P′′-def cmb-P2 cmb-grd-eq cmb-max cmb-nz2 cmb-on-grid cmb-pos*
lower-tick-eq)
thus *y − y2 ≤ quote-gross P′′* (*grd-max P′′*) **using** *assms* **by** *simp*
show *0 ≤ y − y2* **using** *assms* **by** *simp*
qed
hence *quote-gross P′′ sqp′ < quote-gross P′′ rs3′*
using ‹*quote-gross P′′ sqp′ = 0*› *assms* **by** *simp*
hence *sqp′ < rs3′*
using *P′′-def quote-gross-imp-sqp-lt cmb-P2 slice-pool-clmm-dsc cmb-pos*
by *blast*
have *rs1 ≤ sqp′* **using** ‹*rs1 ≤ sqp′*› **by** *simp*
hence *0 ≤* (*1 − phi*) *∗* (*y1−y2*) *∗* (*1*/(*rs1 ∗ rs1*) *− 1*/(*sqp′ ∗ sqp′*))
proof *−*
have *f1*: *0 ≤ 1 − phi*
using *fee-props*(*2*) **by** *force*
have *f2*: *0 ≤ rs1*
using ‹*0 < rs1*› **by** *linarith*
have *0 ≤ sqp′*
using ‹*0 < sqp′*› **by** *linarith*
then have *0 ≤ 1 / (rs1 ∗ rs1) − 1 / (sqp′ ∗ sqp′)*
using *f2* **by** (*simp add*: ‹*0 < rs1*› ‹*rs1 ≤ sqp′*› *frac-le mult-mono*)
then show *?thesis*

```
    using f1 ‹y2 < y1› by force
  qed
  also have ... =  (1 − phi) ∗ (y1−y2)/(rs1 ∗ rs1) − (1 − phi) ∗
    (y1−y2) / (sqp′ ∗ sqp′)
    by (simp add: right-diff-distrib)
  also have ... = (1 − phi) ∗ (quote-gross P1 rs1 − quote-gross P1 rs2) /
    (rs1 ∗ rs1) − (1 − phi) ∗ (quote-gross P′′ rs3′ − quote-gross P′′ sqp′) /
    (sqp′ ∗ sqp′)
    using q1 q2 qlp q3′ assms ‹quote-gross P′′ sqp′ = 0› by simp
  also have ... ≤ (base-net P1 rs2 − base-net P1 rs1) − (1 − phi) ∗
    (quote-gross P′′ rs3′ − quote-gross P′′ sqp′) / (sqp′ ∗ sqp′)
  proof −
    have (1 − phi) ∗ (quote-gross P1 rs1 − quote-gross P1 rs2) /
      (rs1 ∗ rs1) ≤ base-net P1 rs2 − base-net P1 rs1
    proof (rule base-net-quote-lbound)
      show clmm-dsc P1 using cmb-P1 .
      show ⋀i. fee P1 i = phi using fee1 by simp
      show rs2 ≤ rs1 using ‹rs2 ≤ rs1› .
      show 0 < rs2 using clmm-quote-reach-pos
        by (metis clmm-quote-gross-pos quote-gross-grd-max-max
          cmb-P1 cmb-nz1 q2 rs2-def)
    qed
    thus ?thesis by simp
  qed
  also have ... ≤ (base-net P1 rs2 − base-net P1 rs1) −
    (base-net P′′ sqp′ − base-net P′′ rs3′)
  proof −
    have base-net P′′ sqp′ − base-net P′′ rs3′ ≤ (1 − phi) ∗
      (quote-gross P′′ rs3′ − quote-gross P′′ sqp′) / (sqp′ ∗ sqp′)
    proof (rule base-net-quote-ubound)
      show clmm-dsc P′′ using cmb-P2 P′′-def slice-pool-clmm-dsc cmb-pos by
simp
      show ⋀i. fee P′′ i = phi
        using fee2 slice-pool-cst-fees P′′-def cmb-P2 by simp
      show phi < 1 using fee-props by simp
      show 0 < sqp′ using ‹0 < sqp′› .
      show sqp′ ≤ rs3′ using ‹sqp′ < rs3′› by simp
    qed
    thus ?thesis by simp
  qed
  also have ... = (base-net P1 rs2 − base-net P1 rs1) −
    (base-net P′′ sqp2 − base-net P′′ rs3′)
  proof −
    have base-net P′′ sqp′ = base-net P′′ sqp2
    proof (rule quote-gross-equiv-base-net′)
      show clmm-dsc P′′
        using P′′-def cmb-P2 slice-pool-clmm-dsc cmb-pos by simp
      show quote-gross P′′ sqp′ = quote-gross P′′ sqp2
        by (simp add: ‹quote-gross P′′ sqp′ = 0› ‹quote-gross P′′ sqp2 = 0›)
```

```
    show 0 < sqp2 by (simp add: cmb-pos)
    show 0 < sqp' using ‹0 < sqp'› .
  qed
  thus ?thesis by simp
qed
also have ... = quote-swap P1 sqp1 y1 − quote-swap P1 sqp1 y2−
    (base-net P'' sqp2 − base-net P'' rs3')
  unfolding quote-swap-def rs1-def rs2-def using qlp by simp
also have ... = quote-swap P1 sqp1 y1 − quote-swap P1 sqp1 y2−
    (quote-swap P'' sqp2 (y − y2) − quote-swap P'' sqp2 (y − y1))
proof −
  have base-net P'' sqp2 = base-net P''
      (quote-reach P'' (y − y1 + quote-gross P'' sqp2))
  proof (rule quote-gross-equiv-base-net')
    show clmm-dsc P''
      using P''-def cmb-P2 slice-pool-clmm-dsc cmb-pos by simp
    show 0 < sqp2 by (simp add: cmb-pos)
    have y − y1 + quote-gross P'' sqp2 = 0
      by (simp add: ‹quote-gross P'' sqp2 = 0› assms(7))
    hence quote-reach P'' (y − y1 + quote-gross P'' sqp2) = grd-min P''
      using clmm-quote-reach-zero
      by (metis P''-def cmb-P2 cmb-max cmb-nz2 cmb-pos slice-pool-clmm-dsc
          slice-pool-nz-liq')
    moreover have 0 < grd-min P'' using grd-min-pos
      by (metis P''-def ‹clmm-dsc P''› liq-grd-min cmb-P2 cmb-max cmb-nz2
          cmb-pos slice-pool-nz-liq')
    ultimately show 0 < quote-reach P'' (y − y1 + quote-gross P'' sqp2)
      by simp
    have quote-gross P'' (quote-reach P'' (y − y1 + quote-gross P'' sqp2)) = 0
      using ‹quote-reach P'' (y − y1 + quote-gross P'' sqp2) = grd-min P''›
        ‹0 < quote-reach P'' (y − y1 + quote-gross P'' sqp2)› ‹clmm-dsc P''›
        clmm-quote-gross-grd-min-le
      by auto
    thus quote-gross P'' sqp2 =
        quote-gross P'' (quote-reach P'' (y − y1 + quote-gross P'' sqp2))
      using ‹quote-gross P'' sqp2 = 0› by simp
  qed
  hence base-net P'' sqp2 − base-net P'' rs3' =
      quote-swap P'' sqp2 (y − y2) − quote-swap P'' sqp2 (y − y1)
   unfolding quote-swap-def rs3'-def using ‹quote-gross P'' sqp2 = 0› by simp
  thus ?thesis by simp
qed
also have ... = quote-swap P1 sqp1 y1 − quote-swap P1 sqp1 y2−
    (quote-swap P'' sqp2 (y − y2))
proof −
  have quote-swap P'' sqp2 (y − y1) = 0
    using quote-swap-zero assms P''-def cmb-P2 cmb-max cmb-nz2 cmb-pos
      slice-pool-clmm-dsc slice-pool-nz-liq' slice-pool-grd-max'
    by auto
```

```
    thus ?thesis by simp
  qed
  also have ... = quote-swap P sqp1 y −
      (quote-swap P1 sqp1 y2 + quote-swap P'' sqp2 (y − y2))
  proof −
    have quote-swap P sqp1 y =  quote-swap P1 sqp1 y1 +
        quote-swap P'' sqp1 (y − y1)
      using assms combo-quote-swap-slice-eq ‹0 < grd-min P1› P''-def by simp
    moreover have quote-swap P'' sqp1 (y − y1) = 0
      using quote-swap-zero assms P''-def cmb-P2 cmb-max cmb-nz2 cmb-pos
        slice-pool-clmm-dsc slice-pool-nz-liq' slice-pool-grd-max' ‹0 < grd-min P1›
      by auto
    ultimately show ?thesis by simp
  qed
  finally have 0 ≤ quote-swap P sqp1 y −
      (quote-swap P1 sqp1 y2 + quote-swap P'' sqp2 (y − y2)) .
  moreover have quote-swap P'' sqp2 (y − y2) = quote-swap P2 sqp2 (y − y2)
    using assms P''-def slice-pool-quote-swap-gt-zero
   by (smt (z3) cmb-P2 cmb-grd-eq cmb-nz2 cmb-on-grid cmb-pos grd-lower-tick-cong)
  ultimately show ?thesis by simp
qed

lemma quote-swap-optimum':
  assumes 0 < y
  and grd-min P1 ≤ sqp1
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
  and sqp' = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
  and sqp1 ≤ sqp2
  and 0 ≤ y2
  and y2 ≤ y
  and y2 + quote-gross P1 sqp1 ≤ quote-gross P1 (grd-max P1)
  and y − y2 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)
  and y1 ≠ y2
shows quote-swap P1 sqp1 y2 + quote-swap P2 sqp2 (y − y2) ≤ quote-swap P
sqp1 y
proof (cases y = y1)
  case True
  then show ?thesis using quote-swap-opt-below assms by simp
next
  case False
  have y1 ≤ y
  proof (rule combo-joint-quote-gross-decomp-leq(2))
    show 0 < sqp1 using assms grd-min-pos cmb-P1 cmb-nz1 by fastforce
  qed (simp add: assms)+
  hence y1 < y using False by simp
  show ?thesis
  proof (cases y2 < y1)
    case True
```

```
    show ?thesis
    proof (rule quote-swap-opt-above)
      show y2 < y1 using True .
      show y1 < y using ‹y1 < y› .
    qed (auto simp add: assms)
  next
    case False
    hence y1 < y2 using assms by simp
    show ?thesis
    proof (rule quote-swap-opt-above')
      show y1 < y2 using ‹y1 < y2› .
    qed (auto simp add: assms)
  qed
qed

lemma quote-swap-optimum:
  assumes 0 < y
  and 0 < sqp1
  and y + quote-gross P sqp1 ≤ quote-gross P (grd-max P)
  and sqp' = quote-reach P (y + quote-gross P sqp1)
  and y1 = quote-gross P1 sqp' − quote-gross P1 sqp1
  and sqp1 ≤ sqp2
  and grd-min P1 ≤ sqp2
  and 0 ≤ y2
  and y2 ≤ y
  and y2 + quote-gross P1 sqp1 ≤ quote-gross P1 (grd-max P1)
  and y − y2 + quote-gross P2 sqp2 ≤ quote-gross P2 (grd-max P2)
  and y1 ≠ y2
shows quote-swap P1 sqp1 y2 + quote-swap P2 sqp2 (y − y2) ≤ quote-swap P
sqp1 y
proof (cases grd-min P1 ≤ sqp1)
  case True
  then show ?thesis using quote-swap-optimum' assms by simp
next
  case False
  hence q1: quote-gross P1 sqp1 = quote-gross P1 (grd-min P1)
    using assms(2) clmm-quote-gross-grd-min-le cmb-P1 by auto
  have grd-min P = grd-min P1
    using pool-comb-le-grd-min by (simp add: assms(7) cmb-props)
  hence q: quote-gross P sqp1 = quote-gross P (grd-min P1)
   using False assms(2) clmm-quote-gross-grd-min-le combined-P-prop(1) by auto
  have quote-swap P1 sqp1 y2 + quote-swap P2 sqp2 (y − y2) =
      quote-swap P1 (grd-min P1) y2 + quote-swap P2 sqp2 (y − y2)
    using quote-swap-grd-min False assms(2) cmb-P1 cmb-nz1 by simp
  also have ... ≤ quote-swap P (grd-min P1) y using quote-swap-optimum'
    by (metis q1 q assms(1) assms(7−12) assms(3−5) order-refl)
  also have ... = quote-swap P sqp1 y
    using quote-swap-grd-min ‹grd-min P = grd-min P1› False assms(2)
      combined-P-prop
```

```
    by simp
  finally show ?thesis .
qed

end

end
```

References

[1] M. Echenim, E. Gobet, and A.-C. Maurice. Uniswap v3: impermanent loss modeling and swap fees asymptotic analysis, Sept. 2025. Available at https://hal.science/hal-04214315v3/file/Article_IL_Uniswapv3_revision_HAL.pdf.