Tensor Products in Hilbert Spaces

Dominique Unruh

December 4, 2024

Abstract

We formalize the tensor product of Hilbert spaces, and related material. Specifically, we define the product of vectors in Hilbert spaces, of operators on Hilbert spaces, and of subspaces of Hilbert spaces, and of von Neumann algebras, and study their properties.

The theory is based on the AFP entry `Complex_Bounded_Operators` that introduces Hilbert spaces and operators and related concepts, but in addition to their work, we defined and study a number of additional concepts needed for the tensor product.

Specifically: Hilbert-Schmidt and trace-class operators; compact operators; positive operators; the weak operator, strong operator, and weak* topology; the spectral theorem for compact operators; and the double commutant theorem.

Contents

1 *Misc-Tensor-Product* – Miscelleanous results missing from other theories

theory *Misc-Tensor-Product*
imports *HOL−Analysis.Elementary-Topology HOL−Analysis.Abstract-Topology*
HOL−Analysis.Abstract-Limits HOL−Analysis.Function-Topology HOL−Cardinals.Cardinals
HOL−Analysis.Infinite-Sum HOL−Analysis.Harmonic-Numbers Containers.Containers-Auxiliary
Complex-Bounded-Operators.Extra-General
Complex-Bounded-Operators.Extra-Vector-Spaces
Complex-Bounded-Operators.Extra-Ordered-Fields
begin

unbundle *lattice-syntax*

lemma *local-defE*: $(\bigwedge x.\ x{=}y \Longrightarrow P) \Longrightarrow P$ **by** *metis*
— A helper lemma to introduce a local "definition" in the current goal when backwards reasoning. *apply* (*rule local-defE*[*where x=*‹*stuff*›]) will insert $x = stuff$ as a premise. This can be useful before using *apply transfer* because it will introduce some additional knowledge about the properties of x into the transferred goal.

lemma *inv-prod-swap*[*simp*]: ‹*inv prod.swap = prod.swap*›
by (*simp add*: *inv-unique-comp*)

lemma *filterlim-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique S*›
shows ‹((*R* ===> *S*) ===> *rel-filter S* ===> *rel-filter R* ===> (=)) *filterlim filterlim*›
using *filtermap-parametric*[*transfer-rule*] *le-filter-parametric*[*transfer-rule*] **apply** *fail?*
unfolding *filterlim-def*
by *transfer-prover*

definition *rel-topology* :: ‹('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a topology* ⇒ '*b topology* ⇒ *bool*)› **where**
‹*rel-topology R S T* ⟷ (*rel-fun* (*rel-set R*) (=)) (*openin S*) (*openin T*)
∧ (∀ *U*. *openin S U* ⟶ *Domainp* (*rel-set R*) *U*) ∧ (∀ *U*. *openin T U* ⟶ *Rangep* (*rel-set R*) *U*)›

lemma *rel-topology-eq*[*relator-eq*]: ‹*rel-topology* (=) = (=)›
unfolding *rel-topology-def* **using** *openin-inject*
by (*auto simp*: *rel-fun-eq rel-set-eq fun-eq-iff*)

lemma *Rangep-conversep*[*simp*]: ‹*Rangep* (R^{-1-1}) = *Domainp R*›
by *blast*

lemma *Domainp-conversep*[*simp*]: ‹*Domainp* (R^{-1-1}) = *Rangep R*›
by *blast*

lemma *conversep-rel-fun*:
includes *lifting-syntax*
shows ‹(*T* ===> *U*)$^{-1-1}$ = (*T*$^{-1-1}$) ===> (*U*$^{-1-1}$)›
by (*auto simp*: *rel-fun-def*)

lemma *rel-topology-conversep*[*simp*]: ‹*rel-topology* (R^{-1-1}) = ((*rel-topology R*)$^{-1-1}$)›
by (*auto simp add*: *rel-topology-def*[*abs-def*] *simp flip*: *conversep-rel-fun*[**where** *U*=‹(=)›, *simplified*])

lemma *openin-parametric*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-topology R* ===> *rel-set R* ===> (=)) *openin openin*›
by (*auto simp add*: *rel-fun-def rel-topology-def*)

lemma *topspace-parametric* [*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-topology R* ===> *rel-set R*) *topspace topspace*›
proof −
have ∗: ‹∃ *y*∈*topspace T′*. *R x y*› **if** ‹*rel-topology R T T′*› ‹*x* ∈ *topspace T*› **for** *x T T′* **and** *R* :: ‹′*q* ⇒ ′*r* ⇒ *bool*›
proof −
from *that* **obtain** *U* **where** ‹*openin T U*› **and** ‹*x* ∈ *U*›
unfolding *topspace-def*
by *auto*
from ‹*openin T U*›
have ‹*Domainp* (*rel-set R*) *U*›
using ‹*rel-topology R T T′*› *rel-topology-def* **by** *blast*
then obtain *V* **where** [*transfer-rule*]: ‹*rel-set R U V*›
by *blast*
with ‹*x* ∈ *U*› **obtain** *y* **where** ‹*R x y*› **and** ‹*y* ∈ *V*›
by (*meson rel-set-def*)
from ‹*rel-set R U V*› ‹*rel-topology R T T′*› ‹*openin T U*›
have ‹*openin T′ V*›
by (*simp add*: *rel-topology-def rel-fun-def*)
with ‹*y* ∈ *V*› **have** ‹*y* ∈ *topspace T′*›
using *openin-subset* **by** *auto*
with ‹*R x y*› **show** ‹∃ *y*∈*topspace T′*. *R x y*›
by *auto*
qed

show *?thesis*
using ∗[**where** *?R.1*=*R*]
using ∗[**where** *?R.1*=‹R^{-1-1}›]
by (*auto intro*!: *rel-setI*)
qed

lemma [*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-total S*›
assumes [*transfer-rule*]: ‹*bi-unique S*›
assumes [*transfer-rule*]: ‹*bi-total R*›
assumes [*transfer-rule*]: ‹*bi-unique R*›
shows ‹(*rel-topology R* ===> *rel-topology S* ===> (*R* ===> *S*) ===> (=)) *continuous-map continuous-map*›
unfolding *continuous-map-def Pi-def*
by *transfer-prover*

lemma *limitin-closedin*:
assumes ‹*limitin T f c F*›
assumes ‹*range f* ⊆ *S*›
assumes ‹*closedin T S*›
assumes ‹¬ *trivial-limit F*›

shows ‹$c \in S$›
proof −
from *assms* **have** ‹T *closure-of* $S = S$›
by (*simp add*: *closure-of-eq*)
moreover have ‹$c \in T$ *closure-of* S›
using *assms*(*1*) - *assms*(*4*) **apply** (*rule limitin-closure-of*)
using *range-subsetD*[*OF assms*(*2*)] **by** *auto*
ultimately show *?thesis*
by *simp*
qed

lemma *closure-nhds-principal*: ‹$a \in$ *closure* $A \longleftrightarrow$ *inf* (*nhds* a) (*principal* A) $\neq$ *bot*›
proof (*rule iffI*)
show ‹*inf* (*nhds* a) (*principal* A) $\neq$ *bot*› **if** ‹$a \in$ *closure* A›
proof (*cases* ‹$a \in A$›)
case *True*
thus *?thesis*
unfolding *filter-eq-iff eventually-inf eventually-principal eventually-nhds* **by** *force*
next
case *False*
have *at a within* $A \neq$ *bot*
using *False that* **by** (*subst at-within-eq-bot-iff*) *auto*
also have *at a within* $A =$ *inf* (*nhds* a) (*principal* A)
using *False* **by** (*simp add*: *at-within-def*)
finally show *?thesis* **.**
qed
show ‹$a \in$ *closure* A› **if** ‹*inf* (*nhds* a) (*principal* A) $\neq$ *bot*›
by (*metis Diff-empty Diff-insert0 at-within-def closure-subset not-in-closure-trivial-limitI subsetD that*)
qed

lemma *limit-in-closure*:
assumes *lim*: ‹($f \longrightarrow x$) F›
assumes *nt*: ‹$F \neq$ *bot*›
assumes *inA*: ‹$\forall_F\ x$ *in* F. $f\ x \in A$›
shows ‹$x \in$ *closure* A›
proof (*rule Lim-in-closed-set*[*of* - - F])
show $\forall_F\ x$ *in* F. $f\ x \in$ *closure* A
using *inA* **by** *eventually-elim* (*use closure-subset* **in** *blast*)
qed (*use assms* **in** *auto*)

lemma *filterlim-nhdsin-iff-limitin*:
‹$l \in$ *topspace* $T \wedge$ *filterlim* f (*nhdsin* $T\ l$) $F \longleftrightarrow$ *limitin* $T\ f\ l\ F$›
unfolding *limitin-def*
proof *safe*
fix U **assume** $*$: $l \in$ *topspace* T *filterlim* f (*nhdsin* $T\ l$) F *openin* $T\ U\ l \in U$

```
  hence eventually (λy. y ∈ U) (nhdsin T l)
    unfolding eventually-nhdsin by blast
  thus eventually (λx. f x ∈ U) F
    using ∗(2) eventually-compose-filterlim by blast
next
  assume ∗: l ∈ topspace T ∀ U. openin T U ∧ l ∈ U ⟶ (∀F x in F. f x ∈ U)
  show filterlim f (nhdsin T l) F
    unfolding filterlim-def le-filter-def eventually-filtermap
  proof safe
    fix P :: 'a ⇒ bool
    assume eventually P (nhdsin T l)
    then obtain U where U: openin T U l ∈ U ∀x∈U. P x
      using ∗(1) unfolding eventually-nhdsin by blast
    with ∗ have eventually (λx. f x ∈ U) F
      by blast
    thus eventually (λx. P (f x)) F
      by eventually-elim (use U in blast)
  qed
qed

lemma pullback-topology-bi-cont:
  fixes g :: ‹'a ⇒ ('b ⇒ 'c::topological-space)›
    and f :: ‹'a ⇒ 'a ⇒ 'a› and f' :: ‹'c ⇒ 'c ⇒ 'c›
  assumes gf-f'g: ‹⋀a b i. g (f a b) i = f' (g a i) (g b i)›
  assumes f'-cont: ‹⋀a' b'. (case-prod f' ⟶ f' a' b') (nhds a' ×F nhds b')›
  defines ‹T ≡ pullback-topology UNIV g euclidean›
  shows ‹LIM (x,y) nhdsin T a ×F nhdsin T b. f x y :> nhdsin T (f a b)›
proof −
  have topspace[simp]: ‹topspace T = UNIV›
    unfolding T-def topspace-pullback-topology by simp
  have openin: ‹openin T U ⟷ (∃ V. open V ∧ U = g −‘ V)› for U
    by (simp add: assms openin-pullback-topology)

  have 1: ‹nhdsin T a = filtercomap g (nhds (g a))›
    for a :: 'a
      by (auto simp add: filter-eq-iff eventually-filtercomap eventually-nhds eventually-nhdsin
openin)

  have ‹((g ∘ case-prod f) ⟶ g (f a b)) (nhdsin T a ×F nhdsin T b)›
  proof (unfold tendsto-def, intro allI impI)
    fix S assume ‹open S› and gfS: ‹g (f a b) ∈ S›
    obtain U where gf-PiE: ‹g (f a b) ∈ Pi_E UNIV U› and openU: ‹∀ i. openin euclidean (U
i)›
      and finiteD: ‹finite {i. U i ≠ topspace euclidean}› and US: ‹Pi_E UNIV U ⊆ S›
      using product-topology-open-contains-basis[OF ‹open S›[unfolded open-fun-def] gfS]
      by auto

    define D where ‹D = {i. U i ≠ UNIV}›
    with finiteD have ‹finite D›
```

by *auto*

from *openU* **have** *openU*: ‹*open* (*U i*)› **for** *i*
by *auto*

have ∗: ‹*f′* (*g a i*) (*g b i*) ∈ *U i*› **for** *i*
by (*metis PiE-mem gf-PiE iso-tuple-UNIV-I gf-f′g*)

have ‹∀$_F$ *x in nhds* (*g a i*) ×$_F$ *nhds* (*g b i*). *case-prod f′ x* ∈ *U i*› **for** *i*
using *tendsto-def*[*THEN iffD1*, *rule-format*, *OF f′-cont openU* ∗, *of i*] **by** −

then obtain *Pa Pb* **where** ‹*eventually* (*Pa i*) (*nhds* (*g a i*))› **and** ‹*eventually* (*Pb i*) (*nhds* (*g b i*))›
and *PaPb-plus*: ‹(∀ *x y*. *Pa i x* ⟶ *Pb i y* ⟶ *f′ x y* ∈ *U i*)› **for** *i*
unfolding *eventually-prod-filter* **by** (*metis prod.simps*(*2*))

from ‹⋀*i*. *eventually* (*Pa i*) (*nhds* (*g a i*))›
obtain *Ua* **where** ‹*open* (*Ua i*)› **and** *a-Ua*: ‹*g a i* ∈ *Ua i*› **and** *Ua-Pa*: ‹*Ua i* ⊆ *Collect* (*Pa i*)› **for** *i*
unfolding *eventually-nhds*
apply *atomize-elim*
by (*metis mem-Collect-eq subsetI*)
from ‹⋀*i*. *eventually* (*Pb i*) (*nhds* (*g b i*))›
obtain *Ub* **where** ‹*open* (*Ub i*)› **and** *b-Ub*: ‹*g b i* ∈ *Ub i*› **and** *Ub-Pb*: ‹*Ub i* ⊆ *Collect* (*Pb i*)› **for** *i*
unfolding *eventually-nhds*
apply *atomize-elim*
by (*metis mem-Collect-eq subsetI*)
have *UaUb-plus*: ‹*x* ∈ *Ua i* ⟹ *y* ∈ *Ub i* ⟹ *f′ x y* ∈ *U i*› **for** *i x y*
by (*metis PaPb-plus Ua-Pa Ub-Pb mem-Collect-eq subsetD*)

define *Ua′* **where** ‹*Ua′ i* = (*if i*∈*D then Ua i else UNIV*)› **for** *i*
define *Ub′* **where** ‹*Ub′ i* = (*if i*∈*D then Ub i else UNIV*)› **for** *i*

have *Ua′-UNIV*: ‹*U i* = *UNIV* ⟹ *Ua′ i* = *UNIV*› **for** *i*
by (*simp add*: *D-def Ua′-def*)
have *Ub′-UNIV*: ‹*U i* = *UNIV* ⟹ *Ub′ i* = *UNIV*› **for** *i*
by (*simp add*: *D-def Ub′-def*)
have [*simp*]: ‹*open* (*Ua′ i*)› **for** *i*
by (*simp add*: *Ua′-def* ‹*open* (*Ua* -)›)
have [*simp*]: ‹*open* (*Ub′ i*)› **for** *i*
by (*simp add*: *Ub′-def* ‹*open* (*Ub* -)›)
have *a-Ua′*: ‹*g a i* ∈ *Ua′ i*› **for** *i*
by (*simp add*: *Ua′-def a-Ua*)
have *b-Ub′*: ‹*g b i* ∈ *Ub′ i*› **for** *i*
by (*simp add*: *Ub′-def b-Ub*)
have *UaUb′-plus*: ‹*a′* ∈ *Ua′ i* ⟹ *b′* ∈ *Ub′ i* ⟹ *f′ a′ b′* ∈ *U i*› **for** *i a′ b′*
apply (*cases* ‹*i* ∈ *D*›)
using *UaUb-plus* **by** (*auto simp add*: *Ua′-def Ub′-def D-def*)

define Ua'' **where** ‹$Ua'' = Pi\ UNIV\ Ua'$›
define Ub'' **where** ‹$Ub'' = Pi\ UNIV\ Ub'$›

have ‹*open* Ua''›
  using *finiteD* Ua'-*UNIV*
  by (*auto simp add*: *open-fun-def* Ua''-*def PiE-UNIV-domain*
      openin-product-topology-alt D-def intro!: *exI*[**where** x=‹Ua'›] *intro*: *rev-finite-subset*)
have ‹*open* Ub''›
  using *finiteD* Ub'-*UNIV*
  by (*auto simp add*: *open-fun-def* Ub''-*def PiE-UNIV-domain*
      openin-product-topology-alt D-def intro!: *exI*[**where** x=‹Ub'›] *intro*: *rev-finite-subset*)
have a-Ua'': ‹$g\ a \in Ua''$›
  by (*simp add*: Ua''-*def a-Ua'*)
have b-Ub'': ‹$g\ b \in Ub''$›
  by (*simp add*: Ub''-*def b-Ub'*)
have $UaUb''$-*plus*: ‹$a' \in Ua'' \Longrightarrow b' \in Ub'' \Longrightarrow f'\ (a'\ i)\ (b'\ i) \in U\ i$› **for** $i\ a'\ b'$
  using $UaUb'$-*plus* **by** (*force simp add*: Ua''-*def* Ub''-*def*)

define Ua''' **where** ‹$Ua''' = g -` Ua''$›
define Ub''' **where** ‹$Ub''' = g -` Ub''$›
have ‹*openin* $T\ Ua'''$›
  using ‹*open* Ua''› **by** (*auto simp*: *openin* Ua'''-*def*)
have ‹*openin* $T\ Ub'''$›
  using ‹*open* Ub''› **by** (*auto simp*: *openin* Ub'''-*def*)
have a-Ua'': ‹$a \in Ua'''$›
  by (*simp add*: Ua'''-*def a-Ua''*)
have b-Ub'': ‹$b \in Ub'''$›
  by (*simp add*: Ub'''-*def b-Ub''*)
have $UaUb'''$-*plus*: ‹$a \in Ua''' \Longrightarrow b \in Ub''' \Longrightarrow f'\ (g\ a\ i)\ (g\ b\ i) \in U\ i$› **for** $i\ a\ b$
  by (*simp add*: Ua'''-*def UaUb''-plus* Ub'''-*def*)

define Pa' **where** ‹$Pa'\ a \longleftrightarrow a \in Ua'''$› **for** a
define Pb' **where** ‹$Pb'\ b \longleftrightarrow b \in Ub'''$› **for** b

have Pa'-*nhd*: ‹*eventually* Pa' (*nhdsin* $T\ a$)›
  using ‹*openin* $T\ Ua'''$›
  by (*auto simp add*: Pa'-*def eventually-nhdsin intro*!: *exI*[*of* - ‹Ua'''›] a-Ua'')
have Pb'-*nhd*: ‹*eventually* Pb' (*nhdsin* $T\ b$)›
  using ‹*openin* $T\ Ub'''$›
  by (*auto simp add*: Pb'-*def eventually-nhdsin intro*!: *exI*[*of* - ‹Ub'''›] b-Ub'')
have $Pa'Pb'$-*plus*: ‹$(g \circ case\text{-}prod\ f)\ (a,\ b) \in S$› **if** ‹$Pa'\ a$› ‹$Pb'\ b$› **for** $a\ b$
  using *that* $UaUb'''$-*plus US*
  by (*auto simp add*: Pa'-*def* Pb'-*def PiE-UNIV-domain Pi-iff gf-f'g*)

show ‹$\forall_F\ x$ *in nhdsin* $T\ a \times_F$ *nhdsin* $T\ b.\ (g \circ case\text{-}prod\ f)\ x \in S$›
  using Pa'-*nhd* Pb'-*nhd* $Pa'Pb'$-*plus*
  unfolding *eventually-prod-filter*
  apply −

apply (*rule exI*[*of* - *Pa'*])
apply (*rule exI*[*of* - *Pb'*])
by *simp*
qed
then show *?thesis*
unfolding *1 filterlim-filtercomap-iff* **by** −
qed

definition ‹*has-sum-in T f A x* ⟷ *limitin T* (*sum f*) *x* (*finite-subsets-at-top A*)›

lemma *has-sum-in-finite*:
assumes *finite F*
assumes ‹*sum f F* ∈ *topspace T*›
shows *has-sum-in T f F* (*sum f F*)
using *assms*
by (*simp add*: *finite-subsets-at-top-finite has-sum-in-def limitin-def eventually-principal*)

definition ‹*summable-on-in T f A* ⟷ (∃ *x*. *has-sum-in T f A x*)›

definition ‹*infsum-in T f A* = (*let L* = *Collect* (*has-sum-in T f A*) *in if card L* = *1 then the-elem L else 0*)›

lemma *hausdorff-OFCLASS-t2-space*: ‹*OFCLASS*('*a*::*topological-space*, *t2-space-class*)› **if** ‹*Hausdorff-space* (*euclidean* :: '*a topology*)›
proof *intro-classes*
fix *a b* :: '*a*
assume ‹*a* ≠ *b*›
from *that*
show ‹∃ *U V*. *open U* ∧ *open V* ∧ *a* ∈ *U* ∧ *b* ∈ *V* ∧ *U* ∩ *V* = {}›
unfolding *Hausdorff-space-def disjnt-def*
using ‹*a* ≠ *b*› **by** *auto*
qed

lemma *hausdorffI*:
assumes ‹⋀*x y*. *x* ∈ *topspace T* ⟹ *y* ∈ *topspace T* ⟹ *x* ≠ *y* ⟹ ∃ *U V*. *openin T U* ∧ *openin T V* ∧ *x* ∈ *U* ∧ *y* ∈ *V* ∧ *U* ∩ *V* = {}›
shows ‹*Hausdorff-space T*›
using *assms* **by** (*auto simp*: *Hausdorff-space-def disjnt-def*)

lemma *hausdorff-euclidean*[*simp*]: ‹*Hausdorff-space* (*euclidean* :: -::*t2-space topology*)›
apply (*rule hausdorffI*)
by (*metis* (*mono-tags*, *lifting*) *hausdorff open-openin*)

lemma *has-sum-in-unique*:
assumes ‹*Hausdorff-space T*›
assumes ‹*has-sum-in T f A l*›

assumes ‹*has-sum-in T f A l'*›
shows ‹*l = l'*›
using *assms(2,3)*[*unfolded has-sum-in-def*] - *assms(1)*
apply (*rule limitin-Hausdorff-unique*)
by *simp*

lemma *infsum-in-def'*:
assumes ‹*Hausdorff-space T*›
shows ‹*infsum-in T f A = (if summable-on-in T f A then (THE s. has-sum-in T f A s) else 0)*›
proof (*cases* ‹*Collect* (*has-sum-in T f A*) = {}›)
case *True*
then show *?thesis* **using** *True*
by (*auto simp*: *infsum-in-def summable-on-in-def Let-def card-1-singleton-iff*)
next
case *False*
then have ‹*summable-on-in T f A*›
by (*metis* (*no-types*, *lifting*) *empty-Collect-eq summable-on-in-def*)
from *False* ‹*Hausdorff-space T*›
have ‹*card* (*Collect* (*has-sum-in T f A*)) = *1*›
by (*metis* (*mono-tags*, *opaque-lifting*) *has-sum-in-unique is-singletonI' is-singleton-altdef mem-Collect-eq*)
then show *?thesis*
using ‹*summable-on-in T f A*›
by (*smt* (*verit*, *best*) *assms card-1-singletonE has-sum-in-unique infsum-in-def mem-Collect-eq singletonI the-elem-eq the-equality*)
qed

lemma *has-sum-in-infsum-in*:
assumes ‹*Hausdorff-space T*› **and** *summable*: ‹*summable-on-in T f A*›
shows ‹*has-sum-in T f A* (*infsum-in T f A*)›
apply (*simp add*: *infsum-in-def'*[*OF* ‹*Hausdorff-space T*›] *summable*)
apply (*rule theI'*[*of* ‹*has-sum-in T f A*›])
using *has-sum-in-unique*[*OF* ‹*Hausdorff-space T*›, *of f A*] *summable*
by (*meson summable-on-in-def*)

lemma *nhdsin-mono*:
assumes [*simp*]: ‹⋀*x*. *openin T' x* ⟹ *openin T x*›
assumes [*simp*]: ‹*topspace T = topspace T'*›
shows ‹*nhdsin T a* ≤ *nhdsin T' a*›
unfolding *nhdsin-def*
by (*auto intro*!: *INF-superset-mono*)

lemma *has-sum-in-cong*:
assumes ⋀*x*. *x*∈*A* ⟹ *f x = g x*
shows *has-sum-in T f A x* ⟷ *has-sum-in T g A x*
proof −

have ‹$(\forall_F\ x$ *in finite-subsets-at-top* $A.$ *sum* $f\ x \in U) \longleftrightarrow (\forall_F\ x$ *in finite-subsets-at-top* $A.$ *sum* $g\ x \in U)$› **for** U
apply (*rule eventually-subst*)
apply (*subst eventually-finite-subsets-at-top*)
by (*metis* (*mono-tags*, *lifting*) *assms empty-subsetI finite.emptyI subset-eq sum.cong*)
then show *?thesis*
by (*simp add*: *has-sum-in-def limitin-def*)
qed

lemma *infsum-in-eqI'*:
fixes $f\ g :: \langle 'a \Rightarrow 'b{::}comm\text{-}monoid\text{-}add\rangle$
assumes ‹$\bigwedge x.$ *has-sum-in* $T\ f\ A\ x \longleftrightarrow$ *has-sum-in* $T\ g\ B\ x$›
shows ‹*infsum-in* $T\ f\ A =$ *infsum-in* $T\ g\ B$›
by (*simp add*: *infsum-in-def assms*[*abs-def*] *summable-on-in-def*)

lemma *infsum-in-cong*:
assumes $\bigwedge x.\ x{\in}A \Longrightarrow f\ x = g\ x$
shows *infsum-in* $T\ f\ A =$ *infsum-in* $T\ g\ A$
using *assms infsum-in-eqI' has-sum-in-cong* **by** *blast*

lemma *limitin-cong*: *limitin* $T\ f\ c\ F \longleftrightarrow$ *limitin* $T\ g\ c\ F$ **if** *eventually* $(\lambda x.\ f\ x = g\ x)\ F$
by (*smt* (*verit*, *best*) *eventually-elim2 limitin-transform-eventually that*)

lemma *has-sum-in-reindex*:
assumes ‹*inj-on* $h\ A$›
shows ‹*has-sum-in* $T\ g\ (h\ `\ A)\ x \longleftrightarrow$ *has-sum-in* $T\ (g \circ h)\ A\ x$›
proof −
have ‹*has-sum-in* $T\ g\ (h\ `\ A)\ x \longleftrightarrow$ *limitin* $T$ (*sum* $g$) $x$ (*finite-subsets-at-top* $(h\ `\ A)$)›
by (*simp add*: *has-sum-in-def*)
also have ‹$\ldots \longleftrightarrow$ *limitin* $T\ (\lambda F.$ *sum* $g\ (h\ `\ F))\ x$ (*finite-subsets-at-top* A)›
apply (*subst filtermap-image-finite-subsets-at-top*[*symmetric*])
by (*simp-all add*: *assms eventually-filtermap limitin-def*)
also have ‹$\ldots \longleftrightarrow$ *limitin* T (*sum* $(g \circ h))\ x$ (*finite-subsets-at-top* A)›
apply (*rule limitin-cong*)
apply (*rule eventually-finite-subsets-at-top-weakI*)
apply (*rule sum.reindex*)
using *assms subset-inj-on* **by** *blast*
also have ‹$\ldots \longleftrightarrow$ *has-sum-in* $T\ (g \circ h)\ A\ x$›
by (*simp add*: *has-sum-in-def*)
finally show *?thesis* **.**
qed

lemma *summable-on-in-reindex*:
assumes ‹*inj-on* $h\ A$›
shows ‹*summable-on-in* $T\ g\ (h\ `\ A) \longleftrightarrow$ *summable-on-in* $T\ (g \circ h)\ A$›
by (*simp add*: *assms summable-on-in-def has-sum-in-reindex*)

lemma *infsum-in-reindex*:
assumes ‹*inj-on* $h\ A$›

shows ‹*infsum-in T g* (*h* ‘ *A*) = *infsum-in T* (*g* ∘ *h*) *A*›
by (*metis Collect-cong assms has-sum-in-reindex infsum-in-def*)

lemma *has-sum-in-reindex-bij-betw*:
assumes *bij-betw g A B*
shows *has-sum-in T* (λ*x*. *f* (*g x*)) *A s* ⟷ *has-sum-in T f B s*
proof −
have ‹*has-sum-in T* (λ*x*. *f* (*g x*)) *A s* ⟷ *has-sum-in T f* (*g* ‘ *A*) *s*›
by (*metis* (*mono-tags*, *lifting*) *assms bij-betw-imp-inj-on has-sum-in-cong has-sum-in-reindex o-def*)
also have ‹. . . = *has-sum-in T f B s*›
using *assms bij-betw-imp-surj-on* **by** *blast*
finally show *?thesis* **.**
qed

lemma *has-sum-euclidean-iff*: ‹*has-sum-in euclidean f A s* ⟷ (*f has-sum s*) *A*›
by (*simp add*: *has-sum-def has-sum-in-def*)

lemma *summable-on-euclidean-eq*: ‹*summable-on-in euclidean f A* ⟷ *f summable-on A*›
by (*auto simp add*: *infsum-def infsum-in-def has-sum-euclidean-iff*[*abs-def*] *has-sum-def t2-space-class.Lim-def summable-on-def summable-on-in-def*)

lemma *infsum-euclidean-eq*: ‹*infsum-in euclidean f A* = *infsum f A*›
by (*auto simp add*: *infsum-def infsum-in-def′ summable-on-euclidean-eq has-sum-euclidean-iff*[*abs-def*] *has-sum-def t2-space-class.Lim-def*)

lemma *infsum-in-reindex-bij-betw*:
assumes *bij-betw g A B*
shows *infsum-in T* (λ*x*. *f* (*g x*)) *A* = *infsum-in T f B*
proof −
have ‹*infsum-in T* (λ*x*. *f* (*g x*)) *A* = *infsum-in T f* (*g* ‘ *A*)›
by (*metis* (*mono-tags*, *lifting*) *assms bij-betw-imp-inj-on infsum-in-cong infsum-in-reindex o-def*)
also have ‹. . . = *infsum-in T f B*›
using *assms bij-betw-imp-surj-on* **by** *blast*
finally show *?thesis* **.**
qed

lemma *limitin-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique S*›
shows ‹(*rel-topology S* ===> (*R* ===> *S*) ===> *S* ===> *rel-filter R* ===> (⟷)) *limitin limitin*›
proof (*intro rel-funI*, *rename-tac T T′ f f′ l l′ F F′*)
fix *T T′ f f′ l l′ F F′*
assume [*transfer-rule*]: ‹*rel-topology S T T′*›
assume [*transfer-rule*]: ‹(*R* ===> *S*) *f f′*›
assume [*transfer-rule*]: ‹*S l l′*›

assume [*transfer-rule*]: ‹*rel-filter R F F'*›

have *topspace*: ‹*l* ∈ *topspace T* ⟷ *l'* ∈ *topspace T'*›
by *transfer-prover*

have *open1*: ‹∀ *F x in F*. *f x* ∈ *U*›
if ‹*openin T U*› **and** ‹*l* ∈ *U*› **and** *lhs*: ‹(∀ *V*. *openin T' V* ∧ *l'* ∈ *V* ⟶ (∀ *F x in F'*. *f' x* ∈ *V*))›
for *U*
proof −
from ‹*rel-topology S T T'*› ‹*openin T U*›
obtain *V* **where** ‹*openin T' V*› **and** [*transfer-rule*]: ‹*rel-set S U V*›
by (*smt* (*verit*, *best*) *Domainp.cases rel-fun-def rel-topology-def*)
with ‹*S l l'*› **have** ‹*l'* ∈ *V*›
by (*metis* (*no-types*, *lifting*) *assms bi-uniqueDr rel-setD1 that*(*2*))
with *lhs* ‹*openin T' V*›
have ‹∀ *F x in F'*. *f' x* ∈ *V*›
by *auto*
then show ‹∀ *F x in F*. *f x* ∈ *U*›
by *transfer simp*
qed

have *open2*: ‹∀ *F x in F'*. *f' x* ∈ *V*›
if ‹*openin T' V*› **and** ‹*l'* ∈ *V*› **and** *lhs*: ‹(∀ *U*. *openin T U* ∧ *l* ∈ *U* ⟶ (∀ *F x in F*. *f x* ∈ *U*))›
for *V*
proof −
from ‹*rel-topology S T T'*› ‹*openin T' V*›
obtain *U* **where** ‹*openin T U*› **and** [*transfer-rule*]: ‹*rel-set S U V*›
by (*auto simp*: *rel-topology-def rel-fun-def*)
with ‹*S l l'*› **have** ‹*l* ∈ *U*›
by (*metis* (*full-types*) *assms bi-unique-def rel-setD2 that*(*2*))
with *lhs* ‹*openin T U*›
have ‹∀ *F x in F*. *f x* ∈ *U*›
by *auto*
then show ‹∀ *F x in F'*. *f' x* ∈ *V*›
by *transfer simp*
qed

from *topspace open1 open2*
show ‹*limitin T f l F* = *limitin T' f' l' F'*›
unfolding *limitin-def* **by** *auto*
qed

lemma *finite-subsets-at-top-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique R*›
shows ‹(*rel-set R* ===> *rel-filter* (*rel-set R*)) *finite-subsets-at-top finite-subsets-at-top*›
proof (*intro rel-funI*)

fix A B **assume** ‹*rel-set R A B*›
from ‹*bi-unique R*› **obtain** f **where** *Rf*: ‹$R\ x\ (f\ x)$› **if** ‹$x \in A$› **for** x
by (*metis* (*no-types*, *opaque-lifting*) ‹*rel-set R A B*› *rel-setD1*)
have ‹*inj-on f A*›
by (*metis* (*no-types*, *lifting*) *Rf assms bi-unique-def inj-onI*)
have ‹$B = f\ `\ A$›
by (*metis* (*mono-tags*, *lifting*) *Rf* ‹*rel-set R A B*› *assms bi-uniqueDr bi-unique-rel-set-lemma image-cong*)

have *RfX*: ‹*rel-set R X* ($f\ `\ X$)› **if** ‹$X \subseteq A$› **for** X
apply (*rule rel-setI*)
subgoal
by (*metis* (*no-types*, *lifting*) *Rf* ‹*inj-on f A*› *in-mono inj-on-image-mem-iff that*)
subgoal
by (*metis* (*no-types*, *lifting*) *Rf imageE subsetD that*)
done

have *Piff*: ‹$(\exists X.\ \mathit{finite}\ X \wedge X \subseteq A \wedge (\forall Y.\ \mathit{finite}\ Y \wedge X \subseteq Y \wedge Y \subseteq A \longrightarrow P\ (f\ `\ Y))) \longleftrightarrow$
$(\exists X.\ \mathit{finite}\ X \wedge X \subseteq B \wedge (\forall Y.\ \mathit{finite}\ Y \wedge X \subseteq Y \wedge Y \subseteq B \longrightarrow P\ Y))$› **for** P
proof (*rule iffI*)
assume ‹$\exists X.\ \mathit{finite}\ X \wedge X \subseteq A \wedge (\forall Y.\ \mathit{finite}\ Y \wedge X \subseteq Y \wedge Y \subseteq A \longrightarrow P\ (f\ `\ Y))$›
then obtain X **where** ‹*finite X*› **and** ‹$X \subseteq A$› **and** *XP*: ‹*finite* $Y \Longrightarrow X \subseteq Y \Longrightarrow Y \subseteq A \Longrightarrow P\ (f\ `\ Y)$› **for** Y
by *auto*
define X' **where** ‹$X' = f\ `\ X$›
have ‹*finite X'*›
by (*metis X'-def* ‹*finite X*› *finite-imageI*)
have ‹$X' \subseteq B$›
by (*smt* (*verit*, *best*) *Rf X'-def* ‹$X \subseteq A$› ‹*rel-set R A B*› *assms bi-uniqueDr image-subset-iff rel-setD1 subsetD*)
have ‹$P\ Y'$› **if** ‹*finite* Y'› **and** ‹$X' \subseteq Y'$› **and** ‹$Y' \subseteq B$› **for** Y'
proof −
define Y **where** ‹$Y = (f\ -`\ Y') \cap A$›
have ‹*finite Y*›
by (*metis Y-def* ‹*inj-on f A*› *finite-vimage-IntI that*(*1*))
moreover have ‹$X \subseteq Y$›
by (*metis* (*no-types*, *lifting*) *X'-def Y-def* ‹$X \subseteq A$› *image-subset-iff-subset-vimage le-inf-iff that*(*2*))
moreover have ‹$Y \subseteq A$›
by (*metis* (*no-types*, *lifting*) *Y-def inf-le2*)
ultimately have ‹$P\ (f\ `\ Y)$›
by (*rule XP*)
then show ‹$P\ Y'$›
by (*metis* (*no-types*, *lifting*) *Int-greatest Y-def* ‹$B = f\ `\ A$› *dual-order.refl image-subset-iff-subset-vimage inf-le1 subset-antisym subset-image-iff that*(*3*))
qed
then show ‹$\exists X.\ \mathit{finite}\ X \wedge X \subseteq B \wedge (\forall Y.\ \mathit{finite}\ Y \wedge X \subseteq Y \wedge Y \subseteq B \longrightarrow P\ Y)$›
by (*metis* (*no-types*, *opaque-lifting*) ‹$X' \subseteq B$› ‹*finite X'*›)
next

assume ‹$\exists X.$ *finite* $X \wedge X \subseteq B \wedge (\forall Y.$ *finite* $Y \wedge X \subseteq Y \wedge Y \subseteq B \longrightarrow P\ Y)$›
then obtain X **where** ‹*finite* X› **and** ‹$X \subseteq B$› **and** XP: ‹*finite* $Y \Longrightarrow X \subseteq Y \Longrightarrow Y \subseteq B \Longrightarrow P\ Y$› **for** Y
by *auto*
define X' **where** ‹$X' = (f -` X) \cap A$›
have ‹*finite* X'›
by (*simp add*: X'*-def* ‹*finite* X› ‹*inj-on* $f\ A$› *finite-vimage-IntI*)
have ‹$X' \subseteq A$›
by (*simp add*: X'*-def*)
have ‹$P\ (f\ `\ Y')$› **if** ‹*finite* Y'› **and** ‹$X' \subseteq Y'$› **and** ‹$Y' \subseteq A$› **for** Y'
proof −
define Y **where** ‹$Y = f\ `\ Y'$›
have ‹*finite* Y›
by (*metis Y-def finite-imageI that*(*1*))
moreover have ‹$X \subseteq Y$›
using X'*-def Y-def* ‹$B = f\ `\ A$› ‹$X \subseteq B$› *that*(*2*) **by** *blast*
moreover have ‹$Y \subseteq B$›
by (*metis Y-def* ‹$B = f\ `\ A$› *image-mono that*(*3*))
ultimately have ‹$P\ Y$›
by (*rule XP*)
then show ‹$P\ (f\ `\ Y')$›
by (*smt* (*z3*) *Y-def* ‹$B = f\ `\ A$› *imageE imageI subset-antisym subset-iff that*(*3*) *vimage-eq*)
qed
then show ‹$\exists X.$ *finite* $X \wedge X \subseteq A \wedge (\forall Y.$ *finite* $Y \wedge X \subseteq Y \wedge Y \subseteq A \longrightarrow P\ (f\ `\ Y))$›
by (*metis* ‹$X' \subseteq A$› ‹*finite* X'›)
qed

define Z **where** ‹$Z =$ *filtermap* $(\lambda M.\ (M, f`M))$ (*finite-subsets-at-top* A)›
have ‹$\forall_F\ (x, y)$ *in* $Z.$ *rel-set* $R\ x\ y$›
by (*auto intro*!: *eventually-finite-subsets-at-top-weakI simp add*: *Z-def eventually-filtermap RfX*)
moreover have ‹*map-filter-on* $\{(x, y).$ *rel-set* $R\ x\ y\}$ *fst* $Z =$ *finite-subsets-at-top* A›
apply (*rule filter-eq-iff*[*THEN iffD2*])
apply (*subst eventually-map-filter-on*)
subgoal
by (*auto intro*!: *eventually-finite-subsets-at-top-weakI simp add*: *Z-def eventually-filtermap RfX*)[*1*]
subgoal
by (*auto simp add*: *Z-def eventually-filtermap eventually-finite-subsets-at-top RfX*)
done
moreover have ‹*map-filter-on* $\{(x, y).$ *rel-set* $R\ x\ y\}$ *snd* $Z =$ *finite-subsets-at-top* B›
apply (*rule filter-eq-iff*[*THEN iffD2*])
apply (*subst eventually-map-filter-on*)
subgoal
by (*auto intro*!: *eventually-finite-subsets-at-top-weakI simp add*: *Z-def eventually-filtermap RfX*)[*1*]
subgoal
by (*simp add*: *Z-def eventually-filtermap eventually-finite-subsets-at-top RfX Piff*)
done

ultimately show ‹*rel-filter* (*rel-set R*) (*finite-subsets-at-top A*) (*finite-subsets-at-top B*)›
by (*rule rel-filter.intros*[**where** *Z*=*Z*])
qed

lemma *sum-parametric'*[*transfer-rule*]:
includes *lifting-syntax*
fixes *R* :: ‹'*a* ⇒ '*b* ⇒ *bool*› **and** *S* :: ‹'*c*::*comm-monoid-add* ⇒ '*d*::*comm-monoid-add* ⇒ *bool*›
assumes [*transfer-rule*]: ‹*bi-unique R*›
assumes [*transfer-rule*]: ‹(*S* ===> *S* ===> *S*) (+) (+)›
assumes [*transfer-rule*]: ‹*S* 0 0›
shows ‹((*R* ===> *S*) ===> *rel-set R* ===> *S*) *sum sum*›
proof (*intro rel-funI*)
fix *A B f g* **assume** ‹*rel-set R A B*› **and** ‹(*R* ===> *S*) *f g*›
from ‹*bi-unique R*› **obtain** *p* **where** *Rf*: ‹*R x* (*p x*)› **if** ‹*x* ∈ *A*› **for** *x*
by (*metis* (*no-types*, *opaque-lifting*) ‹*rel-set R A B*› *rel-setD1*)
have ‹*inj-on p A*›
by (*metis* (*no-types*, *lifting*) *Rf* ‹*bi-unique R*› *bi-unique-def inj-onI*)
have ‹*B* = *p* ‘ *A*›
by (*metis* (*mono-tags*, *lifting*) *Rf* ‹*rel-set R A B*› ‹*bi-unique R*› *bi-uniqueDr bi-unique-rel-set-lemma image-cong*)

define *A-copy* **where** ‹*A-copy* = *A*›

have ∗: ‹*S* (*f x* + *sum f F*) (*g* (*p x*) + *sum g* (*p* ‘ *F*))›
if [*transfer-rule*]: ‹*S* (*sum f F*) (*sum g* (*p* ‘ *F*))› **and** [*simp*]: ‹*x* ∈ *A*› **for** *x F*
by (*metis* (*no-types*, *opaque-lifting*) *Rf* ‹(*R* ===> *S*) *f g*› *assms*(*2*) *rel-fun-def that*(*1*) *that*(*2*))
have *ind-step*: ‹*S* (*sum f* (*insert x F*)) (*sum g* (*p* ‘ *insert x F*))›
if ‹*S* (*sum f F*) (*sum g* (*p* ‘ *F*))› ‹*x* ∈ *A*› ‹*x* ∉ *F*› ‹*finite F*› ‹*F* ⊆ *A*› **for** *x F*
proof −
have *sum g* (*p* ‘ *insert x F*) = *g* (*p x*) + *sum g* (*p* ‘ *F*)
unfolding *image-insert* **using** *that*
by (*subst sum.insert*) (*use inj-onD*[*OF* ‹*inj-on p A*›, *of x*] **in** ‹*auto*›)
thus *?thesis*
using *that* ∗ **by** *simp*
qed

have ‹*S* (∑ *x*∈*A*. *f x*) (∑ *x*∈*p* ‘ *A*. *g x*)› **if** ‹*A* ⊆ *A-copy*›
using *that*
apply (*induction A rule*:*infinite-finite-induct*)
unfolding *A-copy-def*
subgoal
by (*metis* (*no-types*, *lifting*) ‹*inj-on p A*› *assms*(*3*) *finite-image-iff subset-inj-on sum.infinite*)
using ‹*S* 0 0› *ind-step* **by** *auto*
hence ‹*S* (∑ *x*∈*A*. *f x*) (∑ *x*∈*p* ‘ *A*. *g x*)›
by (*simp add*: *A-copy-def*)
also have ‹... = (∑ *x*∈*B*. *g x*)›
by (*metis* (*full-types*) ‹*B* = *p* ‘ *A*›)
finally show ‹*S* (∑ *x*∈*A*. *f x*) (∑ *x*∈*B*. *g x*)›

by −
qed

lemma *has-sum-in-parametric*[*transfer-rule*]:
includes *lifting-syntax*
fixes $R :: \langle 'a \Rightarrow 'b \Rightarrow bool \rangle$ **and** $S :: \langle 'c{::}comm\text{-}monoid\text{-}add \Rightarrow 'd{::}comm\text{-}monoid\text{-}add \Rightarrow bool \rangle$
assumes [*transfer-rule*]: ‹*bi-unique R*›
assumes [*transfer-rule*]: ‹*bi-unique S*›
assumes [*transfer-rule*]: ‹(*S* ===> *S* ===> *S*) (+) (+)›
assumes [*transfer-rule*]: ‹*S 0 0*›
shows ‹(*rel-topology S* ===> (*R* ===> *S*) ===> (*rel-set R*) ===> *S* ===> (=)) *has-sum-in has-sum-in*›
proof −
note *sum-parametric'*[*transfer-rule*]
show *?thesis*
unfolding *has-sum-in-def*
by *transfer-prover*
qed

lemma *has-sum-in-topspace*: ‹*has-sum-in T f A s* ⟹ *s* ∈ *topspace T*›
by (*metis has-sum-in-def limitin-def*)

lemma *summable-on-in-parametric*[*transfer-rule*]:
includes *lifting-syntax*
fixes $R :: \langle 'a \Rightarrow 'b \Rightarrow bool \rangle$
assumes [*transfer-rule*]: ‹*bi-unique R*›
assumes [*transfer-rule*]: ‹*bi-unique S*›
assumes [*transfer-rule*]: ‹(*S* ===> *S* ===> *S*) (+) (+)›
assumes [*transfer-rule*]: ‹*S 0 0*›
shows ‹(*rel-topology S* ===> (*R* ===> *S*) ===> (*rel-set R*) ===> (=)) *summable-on-in summable-on-in*›
proof (*intro rel-funI*)
fix *T T'* **assume** [*transfer-rule*]: ‹*rel-topology S T T'*›
fix *f f'* **assume** [*transfer-rule*]: ‹(*R* ===> *S*) *f f'*›
fix *A A'* **assume** [*transfer-rule*]: ‹*rel-set R A A'*›

define *ExT ExT'* **where** ‹*ExT P* ⟷ (∃ *x*∈*Collect* (*Domainp S*). *P x*)› **and** ‹*ExT' P'* ⟷ (∃ *x*∈*Collect* (*Rangep S*). *P' x*)› **for** *P P'*
have [*transfer-rule*]: ‹((*S* ===> (⟷)) ===> (⟷)) *ExT ExT'*›
by (*smt* (*z3*) *Domainp-iff ExT'-def ExT-def RangePI Rangep.cases mem-Collect-eq rel-fun-def*)
from ‹*rel-topology S T T'*› **have** *top1*: ‹*topspace T* ⊆ *Collect* (*Domainp S*)›
unfolding *rel-topology-def*
by (*metis* (*no-types*, *lifting*) *Domainp-set mem-Collect-eq openin-topspace subsetI*)
from ‹*rel-topology S T T'*› **have** *top2*: ‹*topspace T'* ⊆ *Collect* (*Rangep S*)›
unfolding *rel-topology-def*
by (*metis* (*no-types*, *lifting*) *RangePI Rangep.cases mem-Collect-eq openin-topspace rel-setD2 subsetI*)

have ‹*ExT* (*has-sum-in T f A*) = *ExT'* (*has-sum-in T' f' A'*)›
by *transfer-prover*
with *top1 top2* **show** ‹*summable-on-in T f A* ⟷ *summable-on-in T' f' A'*›
by (*metis ExT'-def ExT-def has-sum-in-topspace in-mono summable-on-in-def*)
qed

lemma *not-summable-infsum-in-0*: ‹¬ *summable-on-in T f A* ⟹ *infsum-in T f A* = *0*›
by (*smt* (*verit*, *del-insts*) *Collect-empty-eq card-eq-0-iff infsum-in-def summable-on-in-def zero-neq-one*)

lemma *infsum-in-parametric*[*transfer-rule*]:
includes *lifting-syntax*
fixes *R* :: ‹'*a* ⇒ '*b* ⇒ *bool*›
assumes [*transfer-rule*]: ‹*bi-unique R*›
assumes [*transfer-rule*]: ‹*bi-unique S*›
assumes [*transfer-rule*]: ‹(*S* ===> *S* ===> *S*) (+) (+)›
assumes [*transfer-rule*]: ‹*S 0 0*›
shows ‹(*rel-topology S* ===> (*R* ===> *S*) ===> (*rel-set R*) ===> *S*) *infsum-in infsum-in*›
proof (*intro rel-funI*)
fix *T T'* **assume** [*transfer-rule*]: ‹*rel-topology S T T'*›
fix *f f'* **assume** [*transfer-rule*]: ‹(*R* ===> *S*) *f f'*›
fix *A A'* **assume** [*transfer-rule*]: ‹*rel-set R A A'*›
have *S-has-sum*: ‹(*S* ===> (=)) (*has-sum-in T f A*) (*has-sum-in T' f' A'*)›
by *transfer-prover*

have *sum-iff*: ‹*summable-on-in T f A* ⟷ *summable-on-in T' f' A'*›
by *transfer-prover*

define *L L'* **where** ‹*L* = *Collect* (*has-sum-in T f A*)› **and** ‹*L'* = *Collect* (*has-sum-in T' f' A'*)›

have *LT*: ‹*L* ⊆ *topspace T*›
by (*metis* (*mono-tags*, *lifting*) *Ball-Collect L-def has-sum-in-topspace subset-iff*)
have *TS*: ‹*topspace T* ⊆ *Collect* (*Domainp S*)›
by (*metis* (*no-types*, *lifting*) *Ball-Collect Domainp-set* ‹*rel-topology S T T'*› *openin-topspace rel-topology-def*)
have *LT'*: ‹*L'* ⊆ *topspace T'*›
by (*metis Ball-Collect L'-def has-sum-in-topspace subset-eq*)
have *T'S*: ‹*topspace T'* ⊆ *Collect* (*Rangep S*)›
by (*metis* (*mono-tags*, *opaque-lifting*) *Ball-Collect RangePI* ‹*rel-topology S T T'*› *rel-fun-def rel-setD2 topspace-parametric*)
have *Sss'*: ‹*S s s'* ⟹ *has-sum-in T f A s* ⟷ *has-sum-in T' f' A' s'*› **for** *s s'*
using *S-has-sum*
by (*metis* (*mono-tags*, *lifting*) *rel-funE*)

have ‹*rel-set S L L'*›
by (*smt* (*verit*) *Domainp.cases L'-def L-def Rangep.cases* ‹*L* ⊆ *topspace T*› ‹*L'* ⊆ *topspace T'*› ‹⋀*s' s*. *S s s'* ⟹ *has-sum-in T f A s* = *has-sum-in T' f' A' s'*› ‹*topspace T* ⊆ *Collect* (*Domainp S*)› ‹*topspace T'* ⊆ *Collect* (*Rangep S*)› *in-mono mem-Collect-eq rel-setI*)

```
  have cardLL': ‹card L = 1 ⟷ card L' = 1›
    by (metis (no-types, lifting) ‹rel-set S L L'› assms(2) bi-unique-rel-set-lemma card-image)

  have ‹S (infsum-in T f A) (infsum-in T' f' A')› if ‹card L ≠ 1›
   using that cardLL' by (simp add: infsum-in-def L'-def L-def Let-def that ‹S 0 0› flip: sum-iff)

  moreover have ‹S (infsum-in T f A) (infsum-in T' f' A')› if [simp]: ‹card L = 1›
  proof −
    have [simp]: ‹card L' = 1›
      using that cardLL' by simp
    have ‹S (the-elem L) (the-elem L')›
      using ‹rel-set S L L'›
      by (metis (no-types, opaque-lifting) ‹card L' = 1› is-singleton-altdef is-singleton-the-elem
rel-setD1 singleton-iff that)
    then show ?thesis
      by (simp add: infsum-in-def flip: L'-def L-def)
  qed

  ultimately show ‹S (infsum-in T f A) (infsum-in T' f' A')›
    by auto
qed

lemma infsum-parametric[transfer-rule]:
  includes lifting-syntax
  assumes [transfer-rule]: ‹bi-unique R›
  shows ‹((R ===> (=)) ===> (rel-set R) ===> (=)) infsum infsum›
  unfolding infsum-euclidean-eq[symmetric]
  by transfer-prover

lemma summable-on-transfer[transfer-rule]:
  includes lifting-syntax
  assumes [transfer-rule]: ‹bi-unique R›
   shows ‹((R ===> (=)) ===> (rel-set R) ===> (=)) Infinite-Sum.summable-on Infi-
nite-Sum.summable-on›
  unfolding summable-on-euclidean-eq[symmetric]
  by transfer-prover

lemma abs-gbinomial: ‹abs (a gchoose n) = (−1)^(n − nat (ceiling a)) ∗ (a gchoose n)›
proof −
  have ‹(∏i=0..<n. abs (a − of-nat i)) = (− 1) ^ (n − nat (ceiling a)) ∗ (∏i=0..<n. a −
of-nat i)›
  proof (induction n)
    case 0
    then show ?case
      by simp
  next
    case (Suc n)
    consider (geq) ‹of-int n ≥ a› | (lt) ‹of-int n < a›
      by fastforce
```

```
    then show ?case
    proof cases
      case geq
      from geq have ‹abs (a − of-int n) = − (a − of-int n)›
        by simp
      moreover from geq have ‹(Suc n − nat (ceiling a)) = (n − nat (ceiling a)) + 1›
        by (metis Suc-diff-le Suc-eq-plus1 ceiling-le nat-le-iff)
      ultimately show ?thesis
        apply (simp add: Suc)
         by (metis (no-types, lifting) ‹|a − of-int (int n)| = − (a − of-int (int n))› mult.assoc
mult-minus-right of-int-of-nat-eq)
    next
      case lt
      from lt have ‹abs (a − of-int n) = (a − of-int n)›
        by simp
      moreover from lt have ‹(Suc n − nat (ceiling a)) = (n − nat (ceiling a))›
      by (smt (verit, ccfv-threshold) Suc-leI cancel-comm-monoid-add-class.diff-cancel diff-commute
diff-diff-cancel diff-le-self less-ceiling-iff linorder-not-le order-less-le zless-nat-eq-int-zless)
      ultimately show ?thesis
        by (simp add: Suc)
    qed
  qed
  then show ?thesis
    by (simp add: gbinomial-prod-rev abs-prod)
qed

lemma gbinomial-sum-lower-abs:
  fixes a :: ‹'a::{floor-ceiling}›
  defines ‹a' ≡ nat (ceiling a)›
  assumes ‹of-nat m ≥ a−1›
  shows (∑k≤m. abs (a gchoose k)) =
                (−1)^a' ∗ ((−1) ^ m ∗ (a − 1 gchoose m))
              − (−1)^a' ∗ of-bool (a'>0) ∗ ((−1) ^ (a'−1) ∗ (a−1 gchoose (a'−1)))
              + (∑k<a'. abs (a gchoose k))
proof −
  from assms
  have ‹a' ≤ Suc m›
    using ceiling-mono by force

  have ‹(∑k≤m. abs (a gchoose k)) = (∑k=a'..m. abs (a gchoose k)) + (∑k<a'. abs (a
gchoose k))›
    apply (subst asm-rl[of ‹{..m} = {a'..m} ∪ {..<a'}›])
    using ‹a' ≤ Suc m› apply auto[1]
    apply (subst sum.union-disjoint)
    by auto
  also have ‹... = (∑k=a'..m. (−1)^(k−a') ∗ (a gchoose k)) + (∑k<a'. abs (a gchoose k))›
    apply (rule arg-cong[where f=‹λx. x + -›])
    apply (rule sum.cong[OF refl])
    apply (subst abs-gbinomial)
```

```
    using a'-def by blast
  also have ‹... = (∑k=a'..m. (-1)^k * (-1)^a' * (a gchoose k)) + (∑k<a'. abs (a gchoose
k))›
    apply (rule arg-cong[where f=‹λx. x + -›])
    apply (rule sum.cong[OF refl])
    by (simp add: power-diff-conv-inverse)
  also have ‹... = (-1)^a' * (∑k=a'..m. (a gchoose k) * (-1)^k) + (∑k<a'. abs (a gchoose
k))›
    by (auto intro: sum.cong simp: sum-distrib-left)
  also have ‹... = (-1)^a' * (∑k≤m. (a gchoose k) * (-1)^k) - (-1)^a' * (∑k<a'. (a
gchoose k) * (-1)^k) + (∑k<a'. abs (a gchoose k))›
    apply (subst asm-rl[of ‹{..m} = {..<a'} ∪ {a'..m}›])
    using ‹a' ≤ Suc m› apply auto[1]
    apply (subst sum.union-disjoint)
    by (auto simp: distrib-left)
  also have ‹... = (-1)^a' * ((- 1) ^ m * (a - 1 gchoose m)) - (-1)^a' * (∑k<a'. (a
gchoose k) * (-1)^k) + (∑k<a'. abs (a gchoose k))›
    apply (subst gbinomial-sum-lower-neg)
    by simp
  also have ‹... = (-1)^a' * ((-1) ^ m * (a - 1 gchoose m)) - (-1)^a'
            * of-bool (a'>0) * ((-1) ^ (a'-1) * (a-1 gchoose (a'-1)))
            + (∑k<a'. abs (a gchoose k))›
    apply (cases ‹a' = 0›)
    subgoal
      by simp
    subgoal
      by (subst asm-rl[of ‹{..<a'} = {..a'-1}›]) (auto simp: gbinomial-sum-lower-neg)
    done
  finally show ?thesis
    by -
qed

lemma abs-gbinomial-leq1:
  fixes a :: ‹'a :: {linordered-field}›
  assumes ‹abs a ≤ 1›
  shows ‹abs (a gchoose b) ≤ 1›
proof -
  have *: ‹-1 ≤ a› ‹a ≤ 1›
    using abs-le-D2 assms minus-le-iff abs-le-iff assms by auto
  have ‹abs (a gchoose b) = abs ((∏ i = 0..<b. a - of-nat i) / fact b)›
    by (simp add: gbinomial-prod-rev)
  also have ‹... = abs ((∏ i=0..<b. a - of-nat i)) / fact b›
    apply (subst abs-divide)
    by simp
  also have ‹... = (∏ i=0..<b. abs (a - of-nat i)) / fact b›
    apply (subst abs-prod) by simp
  also have ‹... ≤ (∏ i=0..<b. of-nat (Suc i)) / fact b›
  proof (intro divide-right-mono prod-mono conjI)
    fix i assume i ∈ {0..<b}
```

```
    have |a − of-nat i| ≤ |a| + |of-nat i|
      by linarith
    also have |a| ≤ 1
      by fact
    finally show |a − of-nat i| ≤ of-nat (Suc i)
      by simp
  qed auto
  also have ‹... = fact b / fact b›
    by (subst (2) fact-prod-Suc) auto
  also have ‹... = 1›
    by simp
  finally show ?thesis
    by −
qed

lemma gbinomial-summable-abs:
  fixes a :: real
  assumes ‹a ≥ 0› and ‹a ≤ 1›
  shows ‹summable (λn. abs (a gchoose n))›
proof −
  define a' where ‹a' = nat (ceiling a)›
  have a': ‹a' = 0 ∨ a' = 1›
    by (metis One-nat-def a'-def assms(2) ceiling-le-one less-one nat-1 nat-mono order-le-less)
  have aux1: ‹abs x ≤ x' ⟹ abs y ≤ y' ⟹ abs z ≤ z' ⟹ x − y + z ≤ x' + y' + z'› for x
y z x' y' z' :: real
    by auto
  have ‹(∑ i≤n. |a gchoose i|) = (− 1) ^ a' * ((− 1) ^ n * (a − 1 gchoose n)) −
    (− 1) ^ a' * of-bool (0 < a') * ((− 1) ^ (a' − 1) * (a − 1 gchoose (a' − 1))) +
    (∑ k<a'. |a gchoose k|)› for n
    unfolding a'-def by (rule gbinomial-sum-lower-abs) (use assms in auto)
  also have ‹... n ≤ 1 + 1 + 1› for n
    by (rule aux1) (use a' in ‹auto simp add: abs-mult abs-gbinomial-leq1 assms›)
  also have ‹... = 3›
    by simp
  finally show ?thesis
    by (meson abs-ge-zero bounded-imp-summable)
qed

lemma summable-tendsto-times-n:
  fixes f :: ‹nat ⇒ real›
  assumes pos: ‹⋀n. f n ≥ 0›
  assumes dec: ‹decseq (λn. (n+M) * f (n + M))›
  assumes sum: ‹summable f›
  shows ‹(λn. n * f n) ⟶ 0›
proof (rule ccontr)
  assume lim-not-0: ‹¬ (λn. n * f n) ⟶ 0›
  obtain B where ‹(λn. (n+M) * f (n+M)) ⟶ B› and nfB': ‹(n+M) * f (n+M) ≥ B›
for n
    apply (rule decseq-convergent[where B=0, OF dec])
```

```
    using pos that by auto
  then have lim-B: ‹(λn. n * f n) ⟶ B›
    by − (rule LIMSEQ-offset)
  have ‹B ≥ 0›
    apply (subgoal-tac ‹⋀n. n * f n ≥ 0›)
    using Lim-bounded2 lim-B apply blast
    by (simp add: pos)
  moreover have ‹B ≠ 0›
    using lim-B lim-not-0 by blast
  ultimately have ‹B > 0›
    by linarith

  have ge: ‹f n ≥ B / n› if ‹n ≥ M› for n
    using nfB'[of ‹n−M›] that ‹B > 0›  by (auto simp: divide-simps mult-ac)

  have ‹summable (λn. B / n)›
    by (rule summable-comparison-test'[where N=M]) (use sum ‹B > 0› ge in auto)

  moreover have ‹¬ summable (λn. B / n)›
  proof (rule ccontr)
    define C where ‹C = (∑ n. 1 / real n)›
    assume ‹¬ ¬ summable (λn. B / real n)›
    then have ‹summable (λn. inverse B * (B / real n))›
      using summable-mult by blast
    then have ‹summable (λn. 1 / real n)›
      using ‹B ≠ 0› by auto
    then have ‹(∑ n=1..m. 1 / real n) ≤ C› for m
      unfolding C-def by (rule sum-le-suminf) auto
    then have ‹harm m ≤ C› for m
      by (simp add: harm-def inverse-eq-divide)
    then have ‹harm (nat (ceiling (exp C))) ≤ C›
      by −
    then have ‹ln (real (nat (ceiling (exp C))) + 1) ≤ C›
      by (smt (verit, best) ln-le-harm)
    then show False
      by (smt (z3) exp-ln ln-ge-iff of-nat-0-le-iff real-nat-ceiling-ge)
  qed

  ultimately show False
    by simp
qed



lemma gbinomial-tendsto-0:
  fixes a :: real
  assumes ‹a > −1›
  shows ‹(λn. (a gchoose n)) ⟶ 0›
proof −
```

```
have thesis1: ‹(λn. (a gchoose n)) ⟶ 0› if ‹a ≥ 0› for a :: real
proof −
  define m where ‹m = nat (floor a)›
  have m: ‹a ≥ real m› ‹a ≤ real m + 1›
    by (simp-all add: m-def that)
  show ?thesis
  proof (insert m, induction m arbitrary: a)
    case 0
    then have ∗: ‹a ≥ 0› ‹a ≤ 1›
      using assms by auto
    show ?case
      using gbinomial-summable-abs[OF ∗]
      using summable-LIMSEQ-zero tendsto-rabs-zero-iff by blast
  next
    case (Suc m)
    have 1: ‹(λn. (a−1 gchoose n)) ⟶ 0›
      by (rule Suc.IH) (use Suc.prems in auto)
    then have ‹(λn. (a−1 gchoose Suc n)) ⟶ 0›
      using filterlim-sequentially-Suc by blast
    with 1 have ‹(λn. (a−1 gchoose n) + (a−1 gchoose Suc n)) ⟶ 0›
      by (simp add: tendsto-add-zero)
    then have ‹(λn. (a gchoose Suc n)) ⟶ 0›
      using gbinomial-Suc-Suc[of ‹a−1›] by simp
    then show ?case
      using filterlim-sequentially-Suc by blast
  qed
qed

have thesis2: ‹(λn. (a gchoose n)) ⟶ 0› if ‹a > −1› ‹a ≤ 0›
proof −
  have decseq: ‹decseq (λn. abs (a gchoose n))›
  proof (rule decseq-SucI)
    fix n
    have ‹|a gchoose Suc n| = |a gchoose n| ∗ (|a − real n| / (1 + n))›
      unfolding gbinomial-prod-rev by (simp add: abs-mult)
    also have ‹... ≤ |a gchoose n|›
      apply (rule mult-left-le)
      using assms that(2) by auto
    finally show ‹|a gchoose Suc n| ≤ |a gchoose n|›
      by −
  qed
  have abs-a1: ‹abs (a+1) = a+1›
    using assms by auto

  have ‹0 ≤ |a + 1 gchoose n|› for n
    by simp
  moreover have ‹decseq (λn. (n+1) ∗ abs (a+1 gchoose (n+1)))›
    using decseq apply (simp add: gbinomial-rec abs-mult)
    by (smt (verit, best) decseq-def mult.commute mult-left-mono)
```

```
    moreover have ‹summable (λn. abs (a+1 gchoose n))›
      apply (rule gbinomial-summable-abs)
      using that by auto
    ultimately have ‹(λn. n * abs (a+1 gchoose n)) ⟶ 0›
      by (rule summable-tendsto-times-n)
    then have ‹(λn. Suc n * abs (a+1 gchoose Suc n)) ⟶ 0›
      by (rule-tac LIMSEQ-ignore-initial-segment[where k=1 and a=0, simplified])
    then have ‹(λn. abs (Suc n * (a+1 gchoose Suc n))) ⟶ 0›
      by (simp add: abs-mult)
    then have ‹(λn. (a+1) * abs (a gchoose n)) ⟶ 0›
      apply (subst (asm) gbinomial-absorption)
      by (simp add: abs-mult abs-a1)
    then have ‹(λn. abs (a gchoose n)) ⟶ 0›
      using that(1) by force
    then show ‹(λn. (a gchoose n)) ⟶ 0›
      by (rule tendsto-rabs-zero-cancel)
  qed

  from thesis1 thesis2 assms show ?thesis
    using linorder-linear by blast
qed




lemma gbinomial-abs-sum:
  fixes a :: real
  assumes ‹a > 0› and ‹a ≤ 1›
  shows ‹(λn. abs (a gchoose n)) sums 2›
proof −
  define a' where ‹a' = nat (ceiling a)›
  have ‹a' = 1›
    using a'-def assms(1) assms(2) by linarith
  have lim: ‹(λn. (a − 1 gchoose n)) ⟶ 0›
    by (simp add: assms(1) gbinomial-tendsto-0)
  have ‹(∑k≤n. abs (a gchoose k)) = (− 1) ^ a' * ((− 1) ^ n * (a − 1 gchoose n)) −
    (− 1) ^ a' * of-bool (0 < a') * ((− 1) ^ (a'−1) * (a − 1 gchoose (a' − 1))) +
    (∑k<a'. |a gchoose k|)› for n
    unfolding a'-def
    apply (rule gbinomial-sum-lower-abs)
    using assms(2) by linarith
  also have ‹... n = 2 − (− 1) ^ n * (a − 1 gchoose n)› for n
    using assms
    by (auto simp add: ‹a' = 1›)
  finally have ‹(∑k≤n. abs (a gchoose k)) = 2 − (− 1) ^ n * (a − 1 gchoose n)› for n
    by −
  moreover have ‹(λn. 2 − (− 1) ^ n * (a − 1 gchoose n)) ⟶ 2›
  proof −
    have ‹(λn. ((−1) ^ n * (a−1 gchoose n))) ⟶ 0›
```

```
    by (rule tendsto-rabs-zero-cancel) (use lim in ‹simp add: abs-mult tendsto-rabs-zero-iff›)
  then have ‹(λn. 2 − (− 1) ^ n ∗ (a − 1 gchoose n)) ⟶ 2 − 0›
    by (rule tendsto-diff[rotated]) simp
  then show ?thesis
    by simp
 qed
 ultimately have ‹(λn. ∑k≤n. abs (a gchoose k)) ⟶ 2›
  by auto
 then show ?thesis
  using sums-def-le by blast
qed

lemma sums-has-sum:
 fixes s :: ‹'a :: banach›
 assumes sums: ‹f sums s›
 assumes abs-sum: ‹summable (λn. norm (f n))›
 shows ‹(f has-sum s) UNIV›
proof (rule has-sumI-metric)
 fix e :: real assume ‹0 < e›
 define e' where ‹e' = e/2›
 then have ‹e' > 0›
  using ‹0 < e› half-gt-zero by blast
 from suminf-exist-split[where r=e', OF ‹0<e'› abs-sum]
 obtain N where ‹norm (∑i. norm (f (i + N))) < e'›
  by auto
 then have N: ‹(∑i. norm (f (i + N))) < e'›
  by auto
 then have N': ‹norm (∑i. f (i + N)) < e'›
  apply (rule dual-order.strict-trans2)
  by (auto intro!: summable-norm summable-iff-shift[THEN iffD2] abs-sum)

 define X where ‹X = {..<N}›
 then have ‹finite X›
  by auto
 moreover have ‹dist (sum f Y) s < e› if ‹finite Y› and ‹X ⊆ Y› for Y
 proof −
  have ‹dist (sum f Y) s = norm (s − sum f {..<N} − sum f (Y−{..<N}))›
      by (metis X-def diff-diff-eq2 dist-norm norm-minus-commute sum.subset-diff that(1)
that(2))
  also have ‹... ≤ norm (s − sum f {..<N}) + norm (sum f (Y−{..<N}))›
    using norm-triangle-ineq4 by blast
  also have ‹... = norm (∑i. f (i + N)) + norm (sum f (Y−{..<N}))›
    apply (subst suminf-minus-initial-segment)
    using sums sums-summable apply blast
    using sums sums-unique by blast
  also have ‹... < e' + norm (sum f (Y−{..<N}))›
    using N' by simp
  also have ‹... ≤ e' + norm (∑i∈Y−{..<N}. norm (f i))›
    apply (rule add-left-mono)
```

```
    by (smt (verit, best) real-norm-def sum-norm-le)
  also have ‹... ≤ e' + (∑ i∈Y−{..<N}. norm (f i))›
    apply (rule add-left-mono)
    by (simp add: sum-nonneg)
  also have (∑ i∈Y−{..<N}. norm (f i)) = (∑ i|i+N∈Y. norm (f (i + N)))
    by (rule sum.reindex-bij-witness[of - λi. i + N λi. i − N]) auto
  also have ‹e' + ... ≤ e' + (∑ i. norm (f (i + N)))›
     by (auto intro!: add-left-mono sum-le-suminf summable-iff-shift[THEN iffD2] abs-sum
finite-inverse-image ‹finite Y›)
  also have ‹... ≤ e' + e'›
    using N by simp
  also have ‹... = e›
    by (simp add: e'-def)
  finally show ?thesis
    by −
 qed
 ultimately show ‹∃ X. finite X ∧ X ⊆ UNIV ∧ (∀ Y. finite Y ∧ X ⊆ Y ∧ Y ⊆ UNIV ⟶
dist (sum f Y) s < e)›
  by auto
qed

lemma sums-has-sum-pos:
 fixes s :: real
 assumes ‹f sums s›
 assumes ‹⋀n. f n ≥ 0›
 shows ‹(f has-sum s) UNIV›
 apply (rule sums-has-sum)
 apply (simp add: assms(1))
 using assms(1) assms(2) summable-def by auto

lemma gbinomial-abs-has-sum:
 fixes a :: real
 assumes ‹a > 0› and ‹a ≤ 1›
 shows ‹((λn. abs (a gchoose n)) has-sum 2) UNIV›
 apply (rule sums-has-sum-pos)
  apply (rule gbinomial-abs-sum)
 using assms by auto

lemma gbinomial-abs-has-sum-1:
 fixes a :: real
 assumes ‹a > 0› and ‹a ≤ 1›
 shows ‹((λn. abs (a gchoose n)) has-sum 1) (UNIV−{0})›
proof −
 have ‹((λn. abs (a gchoose n)) has-sum 2−(∑ n∈{0}. abs (a gchoose n))) (UNIV−{0})›
  apply (rule has-sum-Diff)
   apply (rule gbinomial-abs-has-sum)
  using assms apply auto[2]
  apply (rule has-sum-finite)
  by auto
```

then show *?thesis*
by *simp*
qed

lemma *gbinomial-abs-summable*:
fixes a :: *real*
assumes ‹$a > 0$› **and** ‹$a \leq 1$›
shows ‹$(\lambda n.\ (a\ gchoose\ n))$ *abs-summable-on UNIV*›
using *assms* **by** (*auto intro*!: *has-sum-imp-summable gbinomial-abs-has-sum*)

lemma *gbinomial-abs-summable-1*:
fixes a :: *real*
assumes ‹$a > 0$› **and** ‹$a \leq 1$›
shows ‹$(\lambda n.\ (a\ gchoose\ n))$ *abs-summable-on UNIV*$-\{0\}$›
using *assms* **by** (*auto intro*!: *has-sum-imp-summable gbinomial-abs-has-sum-1*)

lemma *has-sum-singleton*[*simp*]: ‹$(f$ *has-sum* $y)\ \{x\} \longleftrightarrow f\ x = y$› **for** y :: ‹$'a$ :: {*comm-monoid-add*, *t2-space*}›
using *has-sum-finite*[*of* ‹$\{x\}$›] *infsumI*[*of* f $\{x\}$ y] **by** *auto*

lemma *has-sum-sums*: ‹f *sums* s› **if** ‹$(f$ *has-sum* $s)$ *UNIV*›
proof −
have ‹$(\lambda n.$ *sum* f $\{..<n\}) \longrightarrow s$›
proof (*unfold tendsto-def*, *intro allI impI*)
fix S **assume** ‹*open* S› **and** ‹$s \in S$›
with ‹$(f$ *has-sum* $s)$ *UNIV*›
have ‹$\forall_F$ F *in finite-subsets-at-top UNIV*. *sum* $f\ F \in S$›
using *has-sum-def tendsto-def* **by** *blast*
then
show ‹$\forall_F$ x *in sequentially*. *sum* f $\{..<x\} \in S$›
using *eventually-compose-filterlim filterlim-lessThan-at-top* **by** *blast*
qed
then show *?thesis*
by (*simp add*: *sums-def*)
qed

lemma *The-eqI1*:
assumes ‹$\bigwedge x\ y.\ F\ x \Longrightarrow F\ y \Longrightarrow x = y$›
assumes ‹$\exists\, z.\ F\ z$›
assumes ‹$\bigwedge x.\ F\ x \Longrightarrow P\ x = Q\ x$›
shows ‹$P\ (The\ F) = Q\ (The\ F)$›
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *theI*)

lemma *summable-on-uminus*[*intro*!]:
fixes f :: ‹$'a \Rightarrow 'b$ :: *real-normed-vector*›
assumes ‹f *summable-on* A›
shows ‹$(\lambda i.\ -\ f\ i)$ *summable-on* A›
apply (*subst asm-rl*[*of* ‹$(\lambda i.\ -\ f\ i) = (\lambda i.\ (-1) *_R f\ i)$›])

```
  apply simp
  using assms by (rule summable-on-scaleR-right)

lemma summable-on-diff:
  fixes f g :: 'a ⇒ 'b::real-normed-vector
  assumes ‹f summable-on A›
  assumes ‹g summable-on A›
  shows ‹(λx. f x − g x) summable-on A›
  using summable-on-add[where f=f and g=‹λx. − g x›] summable-on-uminus[where f=g]
  using assms by auto


lemma gbinomial-1: ‹(1 gchoose n) = of-bool (n≤1)›
proof −
  consider (0) ‹n=0› | (1) ‹n=1› | (bigger) m where ‹n=Suc (Suc m)›
    by (metis One-nat-def not0-implies-Suc)
  then show ?thesis
  proof cases
    case 0
    then show ?thesis
      by simp
  next
    case 1
    then show ?thesis
      by simp
  next
    case bigger
    then show ?thesis
      using gbinomial-rec[where a=0 and k=‹Suc m›]
      by simp
  qed
qed



lemma gbinomial-a-Suc-n:
  ‹(a gchoose Suc n) = (a gchoose n) * (a−n) / Suc n›
  by (simp add: gbinomial-prod-rev)

lemma has-sum-in-0[simp]:
  assumes ‹0 ∈ topspace T›
  assumes ‹⋀x. x∈A ⟹ f x = 0›
  shows ‹has-sum-in T f A 0›
proof −
  have ‹has-sum-in T (λ-. 0) A 0›
    using assms
    by (simp add: has-sum-in-def sum.neutral-const[abs-def])
  then show ?thesis
    apply (rule has-sum-in-cong[THEN iffD2, rotated])
```

```
    using assms by simp
qed

lemma has-sum-diff:
  fixes f g :: 'a ⇒ 'b::{topological-ab-group-add}
  assumes ‹(f has-sum a) A›
  assumes ‹(g has-sum b) A›
  shows ‹((λx. f x − g x) has-sum (a − b)) A›
 by (auto intro!: has-sum-add has-sum-uminus[THEN iffD2] assms simp add: simp flip: add-uminus-conv-diff)

lemma has-sum-of-real:
  fixes f :: 'a ⇒ real
  assumes ‹(f has-sum a) A›
  shows ‹((λx. of-real (f x)) has-sum (of-real a :: 'b::{real-algebra-1,real-normed-vector})) A›
  apply (rule has-sum-comm-additive[unfolded o-def, where f=of-real])
  by (auto intro!: additive.intro assms tendsto-of-real)

lemma summable-on-cdivide:
  fixes f :: 'a ⇒ 'b :: {t2-space, topological-semigroup-mult, division-ring}
  assumes ‹f summable-on A›
  shows (λx. f x / c) summable-on A
  apply (subst division-ring-class.divide-inverse)
  using assms summable-on-cmult-left by blast

lemma norm-abs[simp]: ‹norm (abs x) = norm x› for x :: ‹'a :: {idom-abs-sgn, real-normed-div-algebra}›
proof −
  have ‹norm x = norm (sgn x * abs x)›
    by (simp add: sgn-mult-abs)
  also have ‹... = norm |x|›
    by (simp add: norm-mult norm-sgn)
  finally show ?thesis
    by simp
qed

 thm abs-summable-product
lemma abs-summable-product:
  fixes x :: 'a ⇒ 'b::real-normed-div-algebra
  assumes x2-sum: (λi. (x i)^2) abs-summable-on A
    and y2-sum: (λi. (y i)^2) abs-summable-on A
  shows (λi. x i * y i) abs-summable-on A
proof (rule nonneg-bdd-above-summable-on)
  show ‹0 ≤ norm (x i * y i)› for i
    by simp
  show ‹bdd-above (sum (λi. norm (x i * y i)) ` {F. F ⊆ A ∧ finite F})›
  proof (rule bdd-aboveI2, rename-tac F)
    fix F assume ‹F ∈ {F. F ⊆ A ∧ finite F}›
    then have finite F and F ⊆ A
      by auto
```

have *norm* $(x\ i * y\ i) \leq$ *norm* $(x\ i * x\ i)$ + *norm* $(y\ i * y\ i)$ **for** i
unfolding *norm-mult*
by (*smt mult-left-mono mult-nonneg-nonneg mult-right-mono norm-ge-zero*)
hence $(\sum i{\in}F.$ *norm* $(x\ i * y\ i)) \leq (\sum i{\in}F.$ *norm* $((x\ i)^2)$ + *norm* $((y\ i)^2))$
using [[*simp-trace*]]
by (*simp add*: *power2-eq-square sum-mono*)
also have $\ldots = (\sum i{\in}F.$ *norm* $((x\ i)^2)) + (\sum i{\in}F.$ *norm* $((y\ i)^2))$
by (*simp add*: *sum.distrib*)
also have $\ldots \leq (\sum_\infty i{\in}A.$ *norm* $((x\ i)^2)) + (\sum_\infty i{\in}A.$ *norm* $((y\ i)^2))$
using *x2-sum y2-sum* ‹*finite F*› ‹$F \subseteq A$› **by** (*auto intro*!: *finite-sum-le-infsum add-mono*)
finally show ‹$(\sum xa{\in}F.$ *norm* $(x\ xa * y\ xa)) \leq (\sum_\infty i{\in}A.$ *norm* $((x\ i)^2)) + (\sum_\infty i{\in}A.$ *norm* $((y\ i)^2))$›
by *simp*
qed
qed

lemma *Cauchy-Schwarz-ineq-infsum*:
fixes $x :: 'a \Rightarrow 'b{::}\{$*real-normed-div-algebra*$\}$
assumes *x2-sum*: $(\lambda i.\ (x\ i)^2)$ *abs-summable-on* A
and *y2-sum*: $(\lambda i.\ (y\ i)^2)$ *abs-summable-on* A
shows ‹$(\sum_\infty i{\in}A.$ *norm* $(x\ i * y\ i)) \leq$ *sqrt* $(\sum_\infty i{\in}A.$ $(norm\ (x\ i))^2)$ * *sqrt* $(\sum_\infty i{\in}A.$ $(norm\ (y\ i))^2)$›
proof −
have ‹$(\sum_\infty i{\in}A.$ *norm* $(x\ i * y\ i)) \leq$ *sqrt* $(\sum_\infty i{\in}A.$ $(norm\ (x\ i))^2)$ * *sqrt* $(\sum_\infty i{\in}A.$ $(norm\ (y\ i))^2)$›
proof (*rule infsum-le-finite-sums*)
show ‹$(\lambda i.\ x\ i * y\ i)$ *abs-summable-on* A›
using *Misc-Tensor-Product.abs-summable-product x2-sum y2-sum* **by** *blast*
fix F **assume** ‹*finite F*› **and** ‹$F \subseteq A$›

have *sum1*: ‹$(\lambda i.\ (norm\ (x\ i))^2)$ *summable-on* A›
by (*metis* (*mono-tags*, *lifting*) *norm-power summable-on-cong x2-sum*)
have *sum2*: ‹$(\lambda i.\ (norm\ (y\ i))^2)$ *summable-on* A›
by (*metis* (*no-types*, *lifting*) *norm-power summable-on-cong y2-sum*)

have ‹$(\sum i{\in}F.$ *norm* $(x\ i * y\ i))^2 = (\sum i{\in}F.$ *norm* $(x\ i)$ * *norm* $(y\ i))^2$›
by (*simp add*: *norm-mult*)
also have ‹$\ldots \leq (\sum i{\in}F.$ $(norm\ (x\ i))^2)$ * $(\sum i{\in}F.$ $(norm\ (y\ i))^2)$›
using *Cauchy-Schwarz-ineq-sum* **by** *fastforce*
also have ‹$\ldots \leq (\sum_\infty i{\in}A.$ $(norm\ (x\ i))^2)$ * $(\sum i{\in}F.$ $(norm\ (y\ i))^2)$›
using *sum1* ‹*finite F*› ‹$F \subseteq A$›
by (*auto intro*!: *mult-right-mono finite-sum-le-infsum sum-nonneg*)
also have ‹$\ldots \leq (\sum_\infty i{\in}A.$ $(norm\ (x\ i))^2)$ * $(\sum_\infty i{\in}A.$ $(norm\ (y\ i))^2)$›
using *sum2* ‹*finite F*› ‹$F \subseteq A$›
by (*auto intro*!: *mult-left-mono finite-sum-le-infsum infsum-nonneg*)
also have ‹$\ldots =$ *sqrt* $(\sum_\infty i{\in}A.$ $(norm\ (x\ i))^2)$ * *sqrt* $(\sum_\infty i{\in}A.$ $(norm\ (y\ i))^2))^2$›
by (*smt* (*verit*, *best*) *calculation real-sqrt-mult real-sqrt-pow2 zero-le-power2*)
finally show ‹$(\sum i{\in}F.$ *norm* $(x\ i * y\ i)) \leq$ *sqrt* $(\sum_\infty i{\in}A.$ $(norm\ (x\ i))^2)$ * *sqrt* $(\sum_\infty i{\in}A.$

(*norm* (*y i*))^2)›
apply (*rule power2-le-imp-le*)
by (*auto intro*!: *mult-nonneg-nonneg infsum-nonneg*)
qed
then show *?thesis*
by −
qed

lemma *continuous-map-pullback-both*:
assumes *cont*: ‹*continuous-map T1 T2 g'*›
assumes *g'g*: ‹⋀*x*. *f1 x* ∈ *topspace T1* ⟹ *x* ∈ *A1* ⟹ *g'* (*f1 x*) = *f2* (*g x*)›
assumes *top1*: ‹*f1* −‘ *topspace T1* ∩ *A1* ⊆ *g* −‘ *A2*›
shows ‹*continuous-map* (*pullback-topology A1 f1 T1*) (*pullback-topology A2 f2 T2*) *g*›
proof −
from *cont*
have ‹*continuous-map* (*pullback-topology A1 f1 T1*) *T2* (*g'* ∘ *f1*)›
by (*rule continuous-map-pullback*)
then have ‹*continuous-map* (*pullback-topology A1 f1 T1*) *T2* (*f2* ∘ *g*)›
apply (*rule continuous-map-eq*)
by (*simp add*: *g'g topspace-pullback-topology*)
then show *?thesis*
apply (*rule continuous-map-pullback'*)
by (*simp add*: *top1 topspace-pullback-topology*)
qed

lemma *onorm-case-prod-plus-leq*: ‹*onorm* (*case-prod plus* :: - ⇒ *'a*::*real-normed-vector*) ≤ *sqrt 2*›
apply (*rule onorm-bound*)
using *norm-plus-leq-norm-prod* **by** *auto*

lemma *bounded-linear-case-prod-plus*[*simp*]: ‹*bounded-linear* (*case-prod plus*)›
apply (*rule bounded-linear-intro*[**where** *K*=‹*sqrt 2*›])
by (*auto simp add*: *scaleR-right-distrib norm-plus-leq-norm-prod mult.commute*)

lemma *pullback-topology-twice*:
assumes ‹(*f* −‘ *B*) ∩ *A* = *C*›
shows ‹*pullback-topology A f* (*pullback-topology B g T*) = *pullback-topology C* (*g o f*) *T*›
proof −
have *aux*: ‹*S* = *A* ⟷ *S* = *B*› **if** ‹*A* = *B*› **for** *A B S* :: *'z*
using *that* **by** *simp*
have ∗: ‹(∃ *V*. (*openin T U* ∧ *V* = *g* −‘ *U* ∩ *B*) ∧ *S* = *f* −‘ *V* ∩ *A*) = (*openin T U* ∧ *S* = (*g* ∘ *f*) −‘ *U* ∩ *C*)› **for** *S U*
apply (*cases* ‹*openin T U*›)
using *assms*
by (*auto intro*!: *aux simp*: *vimage-comp*)
then have ∗: ‹(∃ *V*. (∃ *U*. *openin T U* ∧ *V* = *g* −‘ *U* ∩ *B*) ∧ *S* = *f* −‘ *V* ∩ *A*) = (∃ *U*. *openin T U* ∧ *S* = (*g* ∘ *f*) −‘ *U* ∩ *C*)› **for** *S*
by *metis*
show *?thesis*

by (*auto intro*!: ∗ *simp*: *topology-eq openin-pullback-topology*)
qed

lemma *pullback-topology-homeo-cong*:
assumes ‹*homeomorphic-map T S g*›
assumes ‹*range f* ⊆ *topspace T*›
shows ‹*pullback-topology A f T* = *pullback-topology A* (*g o f*) *S*›
proof −
have ‹∃ *Us*. *openin S Us* ∧ *f* −‘ *Ut* ∩ *A* = (*g* ∘ *f*) −‘ *Us* ∩ *A*› **if** ‹*openin T Ut*› **for** *Ut*
apply (*rule exI*[*of* - ‹*g* ‘ *Ut*›])
using *assms that* **apply** *auto*
using *homeomorphic-map-openness-eq* **apply** *blast*
by (*smt* (*verit*, *best*) *homeomorphic-map-maps homeomorphic-maps-map openin-subset rangeI subsetD*)
moreover have ‹∃ *Ut*. *openin T Ut* ∧ (*g* ∘ *f*) −‘ *Us* ∩ *A* = *f* −‘ *Ut* ∩ *A*› **if** ‹*openin S Us*› **for** *Us*
apply (*rule exI*[*of* - ‹(*g* −‘ *Us*) ∩ *topspace T*›])
using *assms that* **apply** *auto*
by (*meson continuous-map-open homeomorphic-imp-continuous-map*)
ultimately show *?thesis*
by (*auto simp*: *topology-eq openin-pullback-topology*)
qed

definition ‹*opensets-in T* = *Collect* (*openin T*)›
— This behaves more nicely with the *transfer*-method (and friends) than *openin*. So when rewriting a subgoal, using, e.g., ∃ *U*∈*opensets T*. *xxx* instead of ∃ *U*. *openin T U* ⟶ *xxx* can make *transfer* work better.

lemma *opensets-in-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes ‹*bi-unique R*›
shows ‹(*rel-topology R* ===> *rel-set* (*rel-set R*)) *opensets-in opensets-in*›
proof (*intro rel-funI rel-setI*)
fix *S T*
assume *rel-topo*: ‹*rel-topology R S T*›
fix *U*
assume ‹*U* ∈ *opensets-in S*›
then show ‹∃ *V* ∈ *opensets-in T*. *rel-set R U V*›
by (*smt* (*verit*, *del-insts*) *Domainp.cases mem-Collect-eq opensets-in-def rel-fun-def rel-topo rel-topology-def*)
next
fix *S T* **assume** *rel-topo*: ‹*rel-topology R S T*›
fix *U* **assume** ‹*U* ∈ *opensets-in T*›
then show ‹∃ *V* ∈ *opensets-in S*. *rel-set R V U*›
by (*smt* (*verit*) *RangepE mem-Collect-eq opensets-in-def rel-fun-def rel-topo rel-topology-def*)
qed

lemma *hausdorff-parametric*[*transfer-rule*]:
includes *lifting-syntax*

assumes [*transfer-rule*]: ‹*bi-unique R*›
shows ‹(*rel-topology R* ===> (⟷)) *Hausdorff-space Hausdorff-space*›
proof −
have *Hausdorff-space-def'*: ‹*Hausdorff-space T* ⟷ (∀ *x*∈*topspace T*. ∀ *y*∈*topspace T*. *x* ≠ *y* ⟶ (∃ *U* ∈ *opensets-in T*. ∃ *V* ∈ *opensets-in T*. *x* ∈ *U* ∧ *y* ∈ *V* ∧ *U* ∩ *V* = {}))›
for *T* :: ‹'*z topology*›
unfolding *opensets-in-def Hausdorff-space-def disjnt-def Bex-def* **by** *auto*
show *?thesis*
unfolding *Hausdorff-space-def'*
by *transfer-prover*
qed

lemma *sum-cmod-pos*:
assumes ‹⋀*x*. *x*∈*A* ⟹ *f x* ≥ *0*›
shows ‹(∑ *x*∈*A*. *cmod* (*f x*)) = *cmod* (∑ *x*∈*A*. *f x*)›
by (*metis* (*mono-tags*, *lifting*) *Re-sum assms cmod-Re sum.cong sum-nonneg*)

lemma *min-power-distrib-left*: ‹(*min x y*) ^ *n* = *min* (*x* ^ *n*) (*y* ^ *n*)› **if** ‹*x* ≥ *0*› **and** ‹*y* ≥ *0*›
for *x y* :: ‹- :: *linordered-semidom*›
by (*metis linorder-le-cases min.absorb-iff2 min.order-iff power-mono that*(*1*) *that*(*2*))

lemma *abs-summable-times*:
fixes *f* :: ‹'*a* ⇒ '*c*::{*real-normed-algebra*}› **and** *g* :: ‹'*b* ⇒ '*c*›
assumes *sum-f*: ‹*f abs-summable-on A*›
assumes *sum-g*: ‹*g abs-summable-on B*›
shows ‹(λ(*i*,*j*). *f i* ∗ *g j*) *abs-summable-on A* × *B*›
proof −
have *a1*: ‹(λ*j*. *norm* (*f i*) ∗ *norm* (*g j*)) *abs-summable-on B*› **if** ‹*i* ∈ *A*› **for** *i*
using *sum-g* **by** (*simp add*: *summable-on-cmult-right*)
then have *a2*: ‹(λ*j*. *f i* ∗ *g j*) *abs-summable-on B*› **if** ‹*i* ∈ *A*› **for** *i*
apply (*rule abs-summable-on-comparison-test*)
apply (*fact that*)
by (*simp add*: *norm-mult-ineq*)
from *sum-f*
have ‹(λ*x*. ∑ ∞ *y*∈*B*. *norm* (*f x*) ∗ *norm* (*g y*)) *abs-summable-on A*›
by (*auto simp add*: *infsum-cmult-right' infsum-nonneg intro*!: *summable-on-cmult-left*)
then have *b1*: ‹(λ*x*. ∑ ∞ *y*∈*B*. *norm* (*f x* ∗ *g y*)) *abs-summable-on A*›
apply (*rule abs-summable-on-comparison-test*)
using *a1 a2* **by** (*simp-all add*: *norm-mult-ineq infsum-mono infsum-nonneg*)
from *a2 b1* **show** *?thesis*
by (*intro abs-summable-on-Sigma-iff*[*THEN iffD2*]) *auto*
qed

definition ‹*the-default def S* = (*if card S* = *1 then* (*THE x*. *x* ∈ *S*) *else def*)›

lemma *card1I*:
assumes *a* ∈ *A*
assumes ⋀*x*. *x* ∈ *A* ⟹ *x* = *a*

shows ‹*card A = 1*›
by (*metis One-nat-def assms*(*1*) *assms*(*2*) *card-eq-Suc-0-ex1*)

lemma *the-default-CollectI*:
assumes *P a*
and ⋀*x*. *P x* ⟹ *x = a*
shows *P* (*the-default d* (*Collect P*))
proof −
have *card*: ‹*card* (*Collect P*) *= 1*›
apply (*rule card1I*)
using *assms* **by** *auto*
from *assms* **have** ‹*P* (*THE x*. *P x*)›
by (*rule theI*)
then show *?thesis*
by (*simp add*: *the-default-def card*)
qed

lemma *the-default-singleton*[*simp*]: ‹*the-default def* {*x*} *= x*›
unfolding *the-default-def* **by** *auto*

lemma *the-default-empty*[*simp*]: ‹*the-default def* {} *= def*›
unfolding *the-default-def* **by** *auto*

lemma *the-default-The*: ‹*the-default z S* = (*THE x*. *x* ∈ *S*)› **if** ‹*card S = 1*›
by (*simp add*: *that the-default-def*)

lemma *the-default-parametricity*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*›
shows ‹(*T* ===> *rel-set T* ===> *T*) *the-default the-default*›
proof (*intro rel-funI*, *rename-tac def def' S S'*)
fix *def def'* **assume** [*transfer-rule*]: ‹*T def def'*›
fix *S S'* **assume** [*transfer-rule*]: ‹*rel-set T S S'*›
have *card-eq*: ‹*card S = card S'*›
by *transfer-prover*
show ‹*T* (*the-default def S*) (*the-default def' S'*)›
proof (*cases* ‹*card S = 1*›)
case *True*
define *theS theS'* **where** [*no-atp*]: ‹*theS* = (*THE x*. *x* ∈ *S*)› **and** [*no-atp*]: ‹*theS'* = (*THE x*. *x* ∈ *S'*)›
from *True* **have** *cardS'*: ‹*card S' = 1*›
by (*simp add*: *card-eq*)
have ‹*theS* ∈ *S*›
unfolding *theS-def*
by (*rule theI'*) (*use True* **in** ‹*simp add*: *card-eq-Suc-0-ex1*›)
moreover have ‹*theS'* ∈ *S'*›
unfolding *theS'-def*
by (*rule theI'*) (*use cardS'* **in** ‹*simp add*: *card-eq-Suc-0-ex1*›)

ultimately have ‹*T theS theS'*›
using ‹*rel-set T S S'*› *True cardS'*
by (*auto simp*: *rel-set-def card-1-singleton-iff*)
then show *?thesis*
by (*simp add*: *True cardS' the-default-def theS-def theS'-def*)
next
case *False*
then have *cardS'*: ‹*card S' ≠ 1*›
by (*simp add*: *card-eq*)
show *?thesis*
using *False cardS'* ‹*T def def'*›
by (*auto simp add*: *the-default-def*)
qed
qed

definition ‹*rel-pred T P Q = rel-set T* (*Collect P*) (*Collect Q*)›

lemma *Collect-parametric*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-pred T* ===> *rel-set T*) *Collect Collect*›
by (*auto simp*: *rel-pred-def*)

lemma *fold-graph-finite*:
— Exists as *comp-fun-commute-on.fold-graph-finite*, but the *comp-fun-commute-on*-assumption is not needed.
assumes *fold-graph f z A y*
shows *finite A*
using *assms* **by** *induct simp-all*

lemma *fold-graph-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*, *simp*]: ‹*bi-unique T*›
shows ‹((*T* ===> *U* ===> *U*) ===> *U* ===> *rel-set T* ===> *rel-pred U*)
fold-graph fold-graph›
proof (*intro rel-funI*, *rename-tac f f' z z' A A'*)
fix *f f'* **assume** [*transfer-rule*, *simp*]: ‹(*T* ===> *U* ===> *U*) *f f'*›
fix *z z'* **assume** [*transfer-rule*, *simp*]: ‹*U z z'*›
fix *A A'* **assume** [*transfer-rule*, *simp*]: ‹*rel-set T A A'*›
have *one-direction*: ‹∃ *y'*. *fold-graph f' z' A' y'* ∧ *U y y'*› **if** ‹*fold-graph f z A y*›
and [*transfer-rule*]: ‹*U z z'*› ‹(*T* ===> *U* ===> *U*) *f f'*› ‹*rel-set T A A'*› ‹*bi-unique T*›
for *f f' z z' A A' y* **and** *U* :: ‹*'c1* ⇒ *'d1* ⇒ *bool*› **and** *T* :: ‹*'a1* ⇒ *'b1* ⇒ *bool*›
using ‹*fold-graph f z A y*› ‹*rel-set T A A'*›
proof (*induction arbitrary*: *A'*)
case *emptyI*
then show *?case*
by (*metis* ‹*U z z'*› *empty-iff equals0I fold-graph.intros*(*1*) *rel-setD2*)
next
case (*insertI x A y*)
from *insertI* **have** *foldA*: ‹*fold-graph f z A y*› **and** *T-xA*[*transfer-rule*]: ‹*rel-set T* (*insert x*

```
A) A'› and xA: ‹x ∉ A›
    by simp-all
  define DT RT where ‹DT = Collect (Domainp T)› and ‹RT = Collect (Rangep T)›
  from T-xA have ‹x ∈ DT›
    by (metis DT-def DomainPI insertCI mem-Collect-eq rel-set-def)
  then obtain x' where [transfer-rule]: ‹T x x'›
    unfolding DT-def by blast
  have ‹x' ∈ A'›
    apply transfer by simp
  define A'' where ‹A'' = A' - {x'}›
  then have A'-def: ‹A' = insert x' A''›
    using ‹x' ∈ A'› by fastforce
  have ‹x' ∉ A''›
    unfolding A''-def by simp
  have [transfer-rule]: ‹rel-set T A A''›
    apply (subst asm-rl[of ‹A = (insert x A) - {x}›])
    using insertI.hyps apply blast
    unfolding A''-def
    by transfer-prover
  from insertI.IH[OF this]
  obtain y'' where foldA'': ‹fold-graph f' z' A'' y''› and [transfer-rule]: ‹U y y''›
    by auto
  define y' where ‹y' = f' x' y''›
  have ‹fold-graph f' z' A' y'›
    unfolding A'-def y'-def
    using ‹x' ∉ A''› foldA''
    by (rule fold-graph.intros)
  moreover have ‹U (f x y) y'›
    unfolding y'-def by transfer-prover
  ultimately show ?case
    by auto
 qed

 show ‹rel-pred U (fold-graph f z A) (fold-graph f' z' A')›
  unfolding rel-pred-def rel-set-def bex-simps
  apply safe
  subgoal
    by (rule one-direction[of f z A - U z' T f']) auto
  subgoal
    by (rule one-direction[of f' z' A' - ‹U^-1-1› z ‹T^-1-1› f, simplified])
       (auto simp flip: conversep-rel-fun)
  done
qed

lemma Domainp-rel-filter:
 assumes ‹Domainp r = S›
 shows ‹Domainp (rel-filter r) F ⟷ (F ≤ principal (Collect S))›
proof (intro iffI, elim Domainp.cases, hypsubst)
 fix G
```

assume ‹*rel-filter r F G*›
then obtain *Z* **where** *rZ*: ‹$\forall_F$ (*x*, *y*) *in Z*. *r x y*›
and *ZF*: *map-filter-on* {(*x*, *y*). *r x y*} *fst Z* = *F*
and *map-filter-on* {(*x*, *y*). *r x y*} *snd Z* = *G*
using *rel-filter.simps* **by** *blast*
show ‹*F* ≤ *principal* (*Collect S*)›
using *rZ*
by (*auto simp flip*: *ZF assms intro*!: *filter-leI elim*!: *eventually-mono*
simp: *eventually-principal eventually-map-filter-on case-prod-unfold DomainPI*)
next
assume *asm*: ‹*F* ≤ *principal* (*Collect S*)›
define *Z* **where** ‹*Z* = *inf* (*filtercomap fst F*) (*principal* {(*x*, *y*). *r x y*})›
have *rZ*: ‹$\forall_F$ (*x*, *y*) *in Z*. *r x y*›
by (*simp add*: *Z-def eventually-inf-principal*)
moreover
have ‹($\forall_F$ *x in Z*. *P* (*fst x*) ∧ (*case x of* (*x*, *xa*) ⇒ *r x xa*)) = *eventually P F*› **for** *P*
using *asm* **apply** (*auto simp add*: *le-principal Z-def eventually-inf-principal eventually-filtercomap*)
by (*smt* (*verit*, *del-insts*) *DomainpE assms eventually-elim2*)
then have ‹*map-filter-on* {(*x*, *y*). *r x y*} *fst Z* = *F*›
by (*simp add*: *filter-eq-iff eventually-map-filter-on rZ*)
ultimately show ‹*Domainp* (*rel-filter r*) *F*›
by (*auto simp*: *Domainp-iff intro*!: *exI rel-filter.intros*)
qed

lemma *map-filter-on-cong*:
assumes [*simp*]: ‹$\forall_F$ *x in F*. *x* ∈ *D*›
assumes ‹⋀*x*. *x* ∈ *D* ⟹ *f x* = *g x*›
shows ‹*map-filter-on D f F* = *map-filter-on D g F*›
apply (*rule filter-eq-iff*[*THEN iffD2*, *rule-format*])
apply (*simp add*: *eventually-map-filter-on*)
apply (*rule eventually-subst*)
apply (*rule always-eventually*)
using *assms*(*2*) **by** *auto*

lemma *filtermap-cong*:
assumes ‹$\forall_F$ *x in F*. *f x* = *g x*›
shows ‹*filtermap f F* = *filtermap g F*›
apply (*rule filter-eq-iff*[*THEN iffD2*, *rule-format*])
apply (*simp add*: *eventually-filtermap*)
by (*smt* (*verit*, *del-insts*) *assms eventually-elim2*)

lemma *filtermap-INF-eq*:
assumes *inj-f*: ‹*inj-on f X*›
assumes *B-nonempty*: ‹*B* ≠ {}›
assumes *F-bounded*: ‹⋀*b*. *b*∈*B* ⟹ *F b* ≤ *principal X*›
shows ‹*filtermap f* (⨅ (*F* ‘ *B*)) = (⨅ *b*∈*B*. *filtermap f* (*F b*))›
proof (*rule antisym*)

```
  show ‹filtermap f (⨅ (F ‘ B)) ≤ (⨅ b∈B. filtermap f (F b))›
    by (rule filtermap-INF)
  define f1 where ‹f1 = inv-into X f›
  have f1f: ‹x ∈ X ⟹ f1 (f x) = x› for x
    by (simp add: inj-f f1-def)
  have ff1: ‹x ∈ f ‘ X ⟹ x = f (f1 x)› for x
    by (simp add: f1-def f-inv-into-f)

  have ‹filtermap f (F b) ≤ principal (f ‘ X)› if ‹b ∈ B› for b
    by (metis F-bounded filtermap-mono filtermap-principal that)
  then have ‹(⨅ b∈B. filtermap f (F b)) ≤ (⨅ b∈B. principal (f ‘ X))›
    by (simp add: INF-greatest INF-lower2)
  also have ‹... = principal (f ‘ X)›
    by (simp add: B-nonempty)
  finally have ‹∀F x in ⨅ b∈B. filtermap f (F b). x ∈ f ‘ X›
    using B-nonempty le-principal by auto
  then have ∗: ‹∀F x in ⨅ b∈B. filtermap f (F b). x = f (f1 x)›
    apply (rule eventually-mono)
    by (simp add: ff1)

  have ‹∀F x in F b. x ∈ X› if ‹b ∈ B› for b
    using F-bounded le-principal that by blast
  then have ∗∗: ‹∀F x in F b. f1 (f x) = x› if ‹b ∈ B› for b
    apply (rule eventually-mono)
    using that by (simp-all add: f1f)

  have ‹(⨅ b∈B. filtermap f (F b)) = filtermap f (filtermap f1 (⨅ b∈B. filtermap f (F b)))›
    apply (simp add: filtermap-filtermap)
    using ∗ by (rule filtermap-cong[where f=id, simplified])
  also have ‹... ≤ filtermap f (⨅ b∈B. filtermap f1 (filtermap f (F b)))›
    apply (rule filtermap-mono)
    by (rule filtermap-INF)
  also have ‹... = filtermap f (⨅ b∈B. F b)›
    apply (rule arg-cong[where f=‹filtermap -›])
    apply (rule INF-cong, rule refl)
    unfolding filtermap-filtermap
    using ∗∗ by (rule filtermap-cong[where g=id, simplified])
  finally show ‹(⨅ b∈B. filtermap f (F b)) ≤ filtermap f (⨅ (F ‘ B))›
    by −
qed

lemma filtermap-inf-eq:
  assumes ‹inj-on f X›
  assumes ‹F1 ≤ principal X›
  assumes ‹F2 ≤ principal X›
  shows ‹filtermap f (F1 ⊓ F2) = filtermap f F1 ⊓ filtermap f F2›
proof −
  have ‹filtermap f (F1 ⊓ F2) = filtermap f (INF F∈{F1,F2}. F)›
    by simp
```

also have ‹... = (*INF* F∈{*F1*,*F2*}. *filtermap f F*)›
apply (*rule filtermap-INF-eq*[**where** X=X])
using *assms* **by** *auto*
also have ‹... = *filtermap f F1* ⊓ *filtermap f F2*›
by *simp*
finally show *?thesis*
by −
qed

definition ‹*transfer-bounded-filter-Inf B M* = *Inf M* ⊓ *principal B*›

lemma *Inf-transfer-bounded-filter-Inf*: ‹*Inf M* = *transfer-bounded-filter-Inf UNIV M*›
by (*metis inf-top.right-neutral top-eq-principal-UNIV transfer-bounded-filter-Inf-def*)

lemma *Inf-bounded-transfer-bounded-filter-Inf*:
assumes ‹⋀F. F ∈ M ⟹ F ≤ *principal B*›
assumes ‹M ≠ {}›
shows ‹*Inf M* = *transfer-bounded-filter-Inf B M*›
by (*simp add*: *Inf-less-eq assms*(*1*) *assms*(*2*) *inf-absorb1 transfer-bounded-filter-Inf-def*)

lemma *transfer-bounded-filter-Inf-parametric*[*transfer-rule*]:
includes *lifting-syntax*
fixes r :: ‹*'rep* ⇒ *'abs* ⇒ *bool*›
assumes [*transfer-rule*]: ‹*bi-unique r*›
shows ‹(*rel-set r* ===> *rel-set* (*rel-filter r*) ===> *rel-filter r*)
transfer-bounded-filter-Inf transfer-bounded-filter-Inf›
proof (*intro rel-funI*, *unfold transfer-bounded-filter-Inf-def*)
fix *BF BG* **assume** *BFBG*[*transfer-rule*]: ‹*rel-set r BF BG*›
fix *Fs Gs* **assume** *FsGs*[*transfer-rule*]: ‹*rel-set* (*rel-filter r*) *Fs Gs*›
define *D R* **where** ‹*D* = *Collect* (*Domainp r*)› **and** ‹*R* = *Collect* (*Rangep r*)›

have ‹*rel-set r D R*›
by (*smt* (*verit*) *D-def Domainp-iff R-def RangePI Rangep.cases mem-Collect-eq rel-setI*)
with ‹*bi-unique r*›
obtain *f* **where** ‹*R* = *f* ‘ *D*› **and** [*simp*]: ‹*inj-on f D*› **and** *rf0*: ‹x∈*D* ⟹ *r x* (*f x*)› **for** x
using *bi-unique-rel-set-lemma*
by *metis*
have *rf*: ‹*r x y* ⟷ x ∈ *D* ∧ *f x* = *y*› **for** *x y*
apply (*auto simp*: *rf0*)
using *D-def* **apply** *auto*[*1*]
using *D-def assms bi-uniqueDr rf0* **by** *fastforce*

from *BFBG*
have ‹*BF* ⊆ *D*›
by (*metis rel-setD1 rf subsetI*)

have *G*: ‹*G* = *filtermap f F*› **if** ‹*rel-filter r F G*› **for** *F G*

```
  using that proof cases
  case (1 Z)
  then have Z[simp]: ‹∀_F (x, y) in Z. r x y›
    by −
  then have ‹filtermap f F = filtermap f (map-filter-on {(x, y). r x y} fst Z)›
    using 1 by simp
  also have ‹... = map-filter-on {(x, y). r x y} (f ∘ fst) Z›
    unfolding map-filter-on-UNIV[symmetric]
    apply (subst map-filter-on-comp)
    using Z by simp-all
  also have ‹... = G›
    apply (simp add: o-def rf)
    apply (subst map-filter-on-cong[where g=snd])
    using Z apply (rule eventually-mono)
    using 1 by (auto simp: rf)
  finally show ?thesis
    by simp
qed

have rf-filter: ‹rel-filter r F G ⟷ F ≤ principal D ∧ filtermap f F = G› for F G
  apply (intro iffI conjI)
    apply (metis D-def DomainPI Domainp-rel-filter)
  using G apply simp
  by (metis D-def Domainp-iff Domainp-rel-filter G)

have FD: ‹F ≤ principal D› if ‹F ∈ Fs› for F
  by (meson FsGs rel-setD1 rf-filter that)

from BFBG
have [simp]: ‹BG = f ‘ BF›
  by (auto simp: rel-set-def rf)

from FsGs
have [simp]: ‹Gs = filtermap f ‘ Fs›
  using G apply (auto simp: rel-set-def rf)
  by fastforce

show ‹rel-filter r (⨅ Fs ⊓ principal BF) (⨅ Gs ⊓ principal BG)›
proof (cases ‹Fs = {}›)
  case True
  then have ‹Gs = {}›
    by transfer
  have ‹rel-filter r (principal BF) (principal BG)›
    by transfer-prover
  with True ‹Gs = {}› show ?thesis
    by simp
next
  case False
  note False[simp]
```

then have [*simp*]: ‹$Gs \neq \{\}$›
by *transfer*
have ‹*rel-filter r* (⨅ *Fs* ⊓ *principal BF*) (*filtermap f* (⨅ *Fs* ⊓ *principal BF*))›
apply (*rule rf-filter*[*THEN iffD2*])
by (*simp add*: ‹$BF \subseteq D$› *le-infI2*)
then show *?thesis*
using *FD* ‹$BF \subseteq D$›
by (*simp add*: *Inf-less-eq*
flip: *filtermap-inf-eq*[**where** $X=D$] *filtermap-INF-eq*[**where** $X=D$] *flip*: *filtermap-principal*)
qed
qed

definition ‹*transfer-inf-principal F M* = *F* ⊓ *principal M*›

lemma *transfer-inf-principal-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*›
shows ‹(*rel-filter T* ===> *rel-set T* ===> *rel-filter T*) *transfer-inf-principal transfer-inf-principal*›
proof −
have ∗: ‹*transfer-inf-principal F M* = *transfer-bounded-filter-Inf M* {*F*}› **for** *F* :: ‹'*z filter*›
and *M*
by (*simp add*: *transfer-inf-principal-def*[*abs-def*] *transfer-bounded-filter-Inf-def*)
show *?thesis*
unfolding ∗
apply *transfer-prover-start*
apply *transfer-step*+
by *transfer-prover*
qed

lemma *continuous-map-is-continuous-at-point*:
assumes ‹*continuous-map T U f*›
shows ‹*filterlim f* (*nhdsin U* (*f l*)) (*atin T l*)›
by (*metis assms atin-degenerate bot.extremum continuous-map-atin filterlim-iff-le-filtercomap filterlim-nhdsin-iff-limitin*)

lemma *set-compr-2-image-collect*: ‹{*f x y* |*x y*. *P x y*} = *case-prod f* ‘ *Collect* (*case-prod P*)›
by *fast*

lemma *closure-image-closure*: ‹*continuous-on* (*closure S*) *f* ⟹ *closure* (*f* ‘ *closure S*) = *closure* (*f* ‘ *S*)›
by (*smt* (*verit*) *closed-closure closure-closure closure-mono closure-subset image-closure-subset image-mono set-eq-subset*)

lemma *has-sum-reindex-bij-betw*:
assumes *bij-betw g A B*
shows ((λ*x*. *f* (*g x*)) *has-sum l*) *A* ⟷ (*f has-sum l*) *B*

proof −
have ‹((λx. f (g x)) has-sum l) A ⟷ (f has-sum l) (g ‘ A)›
apply (*rule has-sum-reindex*[*symmetric*, *unfolded o-def*])
using *assms bij-betw-imp-inj-on* **by** *blast*
also have ‹... ⟷ (f has-sum l) B›
using *assms bij-betw-imp-surj-on* **by** *blast*
finally show *?thesis* **.**
qed

lemma *enum-inj*:
assumes $i < CARD('a)$ **and** $j < CARD('a)$
shows (*Enum.enum* ! *i* :: 'a::*enum*) = *Enum.enum* ! *j* ⟷ *i* = *j*
using *inj-on-nth*[*OF enum-distinct*, **where** *I*=‹{..<*CARD*('a)}›]
using *assms* **by** (*auto dest*: *inj-onD simp flip*: *card-UNIV-length-enum*)

lemma *closedin-vimage*:
assumes ‹*closedin U S*›
assumes ‹*continuous-map T U f*›
shows ‹*closedin T* (*topspace T* ∩ (*f* −‘ *S*))›
by (*meson assms*(*1*) *assms*(*2*) *continuous-map-closedin-preimage-eq*)

lemma *join-forall*: ‹(∀x. P x) ∧ (∀x. Q x) ⟷ (∀x. P x ∧ Q x)›
by *auto*

lemma *closedin-if-converge-inside*:
fixes *A* :: ‹'*a set*›
assumes *AT*: ‹*A* ⊆ *topspace T*›
assumes *xA*: ‹⋀(*F*::'*a filter*) *f x*. *F* ≠ ⊥ ⟹ *limitin T f x F* ⟹ *range f* ⊆ *A* ⟹ *x* ∈ *A*›
shows ‹*closedin T A*›
proof (*cases* ‹*A* = {}›)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then obtain *a* **where** ‹*a* ∈ *A*›
by *auto*
define *Ac* **where** ‹*Ac* = *topspace T* − *A*›
have ‹∃*U*. *openin T U* ∧ *x* ∈ *U* ∧ *U* ⊆ *Ac*› **if** ‹*x* ∈ *Ac*› **for** *x*
proof (*rule ccontr*)
assume ‹∄*U*. *openin T U* ∧ *x* ∈ *U* ∧ *U* ⊆ *Ac*›
then have *UA*: ‹*U* ∩ *A* ≠ {}› **if** ‹*openin T U*›**and** ‹*x* ∈ *U*› **for** *U*
by (*metis Ac-def Diff-mono Diff-triv openin-subset subset-refl that*)
have [*simp*]: ‹*x* ∈ *topspace T*›
using *that* **by** (*simp add*: *Ac-def*)

define *F* **where** ‹*F* = *nhdsin T x* ⊓ *principal A*›
have ‹*F* ≠ ⊥›
apply (*subst filter-eq-iff*)
apply (*auto intro*!: *exI*[*of* - ‹λ-. *False*›] *simp*: *F-def eventually-inf eventually-principal*

```
      eventually-nhdsin)
    by (meson UA disjoint-iff)

  define f where ‹f y = (if y∈A then y else a)› for y
  with ‹a ∈ A› have ‹range f ⊆ A›
    by force

  have ‹∀F y in F. f y ∈ U› if ‹openin T U› and ‹x ∈ U› for U
  proof −
    have ‹eventually (λx. x ∈ U) (nhdsin T x)›
      using eventually-nhdsin that by fastforce
    moreover have ‹∃R. (∀x∈A. R x) ∧ (∀x. x ∈ U ⟶ R x ⟶ f x ∈ U)›
      apply (rule exI[of - ‹λx. x ∈ A›])
      by (simp add: f-def)
    ultimately show ?thesis
      by (auto simp add: F-def eventually-inf eventually-principal)
  qed
  then have ‹limitin T f x F›
    unfolding limitin-def by simp
  with ‹F ≠ ⊥› ‹range f ⊆ A› xA
  have ‹x ∈ A›
    by simp
  with that show False
    by (simp add: Ac-def)
 qed
 then have ‹openin T Ac›
  apply (rule-tac openin-subopen[THEN iffD2])
  by simp
 then show ?thesis
  by (simp add: Ac-def AT closedin-def)
qed

lemma cmod-mono: ‹0 ≤ a ⟹ a ≤ b ⟹ cmod a ≤ cmod b›
 by (simp add: cmod-Re less-eq-complex-def)

lemma has-sum-mono-neutral-complex:
 fixes f :: 'a ⇒ complex
 assumes ‹(f has-sum a) A› and (g has-sum b) B
 assumes ‹⋀x. x ∈ A∩B ⟹ f x ≤ g x›
 assumes ‹⋀x. x ∈ A−B ⟹ f x ≤ 0›
 assumes ‹⋀x. x ∈ B−A ⟹ g x ≥ 0›
 shows a ≤ b
proof −
 have ‹((λx. Re (f x)) has-sum Re a) A›
  using assms(1) has-sum-Re has-sum-cong by blast
 moreover have ‹((λx. Re (g x)) has-sum Re b) B›
  using assms(2) has-sum-Re has-sum-cong by blast
 ultimately have Re: ‹Re a ≤ Re b›
  apply (rule has-sum-mono-neutral)
```

using *assms*(*3*−*5*) **by** (*simp-all add*: *less-eq-complex-def*)
have ‹((λ*x*. *Im* (*f x*)) *has-sum Im a*) *A*›
using *assms*(*1*) *has-sum-Im has-sum-cong* **by** *blast*
then have ‹((λ*x*. *Im* (*f x*)) *has-sum Im a*) (*A* ∩ *B*)›
apply (*rule has-sum-cong-neutral*[*THEN iffD1*, *rotated* −*1*])
using *assms*(*3*−*5*) **by** (*auto simp add*: *less-eq-complex-def*)
moreover have ‹((λ*x*. *Im* (*g x*)) *has-sum Im b*) *B*›
using *assms*(*2*) *has-sum-Im has-sum-cong* **by** *blast*
then have ‹((λ*x*. *Im* (*f x*)) *has-sum Im b*) (*A* ∩ *B*)›
apply (*rule has-sum-cong-neutral*[*THEN iffD1*, *rotated* −*1*])
using *assms*(*3*−*5*) **by** (*auto simp add*: *less-eq-complex-def*)
ultimately have *Im*: ‹*Im a* = *Im b*›
by (*rule has-sum-unique*)
from *Re Im* **show** *?thesis*
using *less-eq-complexI* **by** *blast*
qed

lemma *choice2*: ‹∃*f*. (∀*x*. *Q1 x* (*f x*)) ∧ (∀*x*. *Q2 x* (*f x*))›
if ‹∀*x*. ∃*y*. *Q1 x y* ∧ *Q2 x y*›
by (*meson that*)

lemma *choice3*: ‹∃*f*. (∀*x*. *Q1 x* (*f x*)) ∧ (∀*x*. *Q2 x* (*f x*)) ∧ (∀*x*. *Q3 x* (*f x*))›
if ‹∀*x*. ∃*y*. *Q1 x y* ∧ *Q2 x y* ∧ *Q3 x y*›
by (*meson that*)

lemma *choice4*: ‹∃*f*. (∀*x*. *Q1 x* (*f x*)) ∧ (∀*x*. *Q2 x* (*f x*)) ∧ (∀*x*. *Q3 x* (*f x*)) ∧ (∀*x*. *Q4 x* (*f x*))›
if ‹∀*x*. ∃*y*. *Q1 x y* ∧ *Q2 x y* ∧ *Q3 x y* ∧ *Q4 x y*›
by (*meson that*)

lemma *choice5*: ‹∃*f*. (∀*x*. *Q1 x* (*f x*)) ∧ (∀*x*. *Q2 x* (*f x*)) ∧ (∀*x*. *Q3 x* (*f x*)) ∧ (∀*x*. *Q4 x* (*f x*)) ∧ (∀*x*. *Q5 x* (*f x*))›
if ‹∀*x*. ∃*y*. *Q1 x y* ∧ *Q2 x y* ∧ *Q3 x y* ∧ *Q4 x y* ∧ *Q5 x y*›
apply (*simp only*: *flip*: *all-conj-distrib*)
using *that* **by** (*rule choice*)

definition (**in** *order*) ‹*is-Sup X s* ⟷ (∀*x*∈*X*. *x* ≤ *s*) ∧ (∀*y*. (∀*x*∈*X*. *x* ≤ *y*) ⟶ *s* ≤ *y*)›
definition (**in** *order*) ‹*has-Sup X* ⟷ (∃*s*. *is-Sup X s*)›

lemma (**in** *order*) *is-SupI*:
assumes ‹⋀*x*. *x* ∈ *X* ⟹ *x* ≤ *s*›
assumes ‹⋀*y*. (⋀*x*. *x* ∈ *X* ⟹ *x* ≤ *y*) ⟹ *s* ≤ *y*›
shows ‹*is-Sup X s*›
using *assms* **by** (*auto simp add*: *is-Sup-def*)

lemma *is-Sup-unique*: ‹*is-Sup X a* ⟹ *is-Sup X b* ⟹ *a*=*b*›
by (*simp add*: *Orderings.order-eq-iff is-Sup-def*)

lemma *has-Sup-bdd-above*: ‹*has-Sup X* ⟹ *bdd-above X*›

by (*metis bdd-above.unfold has-Sup-def is-Sup-def*)

lemma *is-Sup-has-Sup*: ‹*is-Sup X s* ⟹ *has-Sup X*›
using *has-Sup-def* **by** *blast*

class *Sup-order* = *order* + *Sup* + *sup* +
assumes *is-Sup-Sup*: ‹*has-Sup X* ⟹ *is-Sup X* (*Sup X*)›
assumes *is-Sup-sup*: ‹*has-Sup* {*x*,*y*} ⟹ *is-Sup* {*x*,*y*} (*sup x y*)›

lemma (**in** *Sup-order*) *is-Sup-eq-Sup*:
assumes ‹*is-Sup X s*›
shows ‹*s* = *Sup X*›
by (*meson assms local.dual-order.antisym local.has-Sup-def local.is-Sup-Sup local.is-Sup-def*)

lemma *is-Sup-cSup*:
fixes *X* :: ‹'*a*::*conditionally-complete-lattice set*›
assumes ‹*bdd-above X*› **and** ‹*X* ≠ {}›
shows ‹*is-Sup X* (*Sup X*)›
using *assms* **by** (*auto intro*!: *cSup-upper cSup-least simp*: *is-Sup-def*)

lemma *continuous-map-iff-preserves-convergence*:
assumes ‹⋀*F a*. *a* ∈ *topspace T* ⟹ *limitin T id a F* ⟹ *limitin U f* (*f a*) *F*›
shows ‹*continuous-map T U f*›
apply (*rule continuous-map-atin*[*THEN iffD2*], *intro ballI*)
using *assms*
by (*simp add*: *limitin-continuous-map*)

lemma *SMT-choices*:
— Was included as SMT.choices in Isabelle and disappeared
⋀*Q*. ∀ *x*. ∃ *y ya*. *Q x y ya* ⟹ ∃ *f fa*. ∀ *x*. *Q x* (*f x*) (*fa x*)
⋀*Q*. ∀ *x*. ∃ *y ya yb*. *Q x y ya yb* ⟹ ∃ *f fa fb*. ∀ *x*. *Q x* (*f x*) (*fa x*) (*fb x*)
⋀*Q*. ∀ *x*. ∃ *y ya yb yc*. *Q x y ya yb yc* ⟹ ∃ *f fa fb fc*. ∀ *x*. *Q x* (*f x*) (*fa x*) (*fb x*) (*fc x*)
⋀*Q*. ∀ *x*. ∃ *y ya yb yc yd*. *Q x y ya yb yc yd* ⟹
∃ *f fa fb fc fd*. ∀ *x*. *Q x* (*f x*) (*fa x*) (*fb x*) (*fc x*) (*fd x*)
⋀*Q*. ∀ *x*. ∃ *y ya yb yc yd ye*. *Q x y ya yb yc yd ye* ⟹
∃ *f fa fb fc fd fe*. ∀ *x*. *Q x* (*f x*) (*fa x*) (*fb x*) (*fc x*) (*fd x*) (*fe x*)
⋀*Q*. ∀ *x*. ∃ *y ya yb yc yd ye yf*. *Q x y ya yb yc yd ye yf* ⟹
∃ *f fa fb fc fd fe ff*. ∀ *x*. *Q x* (*f x*) (*fa x*) (*fb x*) (*fc x*) (*fd x*) (*fe x*) (*ff x*)
⋀*Q*. ∀ *x*. ∃ *y ya yb yc yd ye yf yg*. *Q x y ya yb yc yd ye yf yg* ⟹
∃ *f fa fb fc fd fe ff fg*. ∀ *x*. *Q x* (*f x*) (*fa x*) (*fb x*) (*fc x*) (*fd x*) (*fe x*) (*ff x*) (*fg x*)
by *metis*+

lemma *closedin-pullback-topology*:
closedin (*pullback-topology A f T*) *S* ⟷ (∃ *C*. *closedin T C* ∧ *S* = *f*−‘*C* ∩ *A*)
proof (*rule iffI*)
define *TT PT* **where** ‹*TT* = *topspace T*› **and** ‹*PT* = *topspace* (*pullback-topology A f T*)›
assume *closed*: ‹*closedin* (*pullback-topology A f T*) *S*›

then have ‹$S \subseteq PT$›
using *PT-def closedin-subset* **by** *blast*
from *closed* **have** ‹*openin* (*pullback-topology A f T*) (*PT* − *S*)›
by (*auto intro*!: *simp*: *closedin-def PT-def*)
then obtain *U* **where** ‹*openin T U*› **and** *S-fUA*: ‹$PT - S = f -` U \cap A$›
by (*auto simp*: *openin-pullback-topology*)
define *C* **where** ‹$C = TT - U$›
have ‹*closedin T C*›
using *C-def TT-def* ‹*openin T U*› **by** *blast*
moreover have ‹$S = f -` C \cap A$›
using *S-fUA* ‹$S \subseteq PT$›
apply (*simp only*: *C-def PT-def TT-def*)
by (*metis Diff-Diff-Int Diff-Int-distrib2 inf.absorb-iff2 topspace-pullback-topology vimage-Diff*)
ultimately show ‹$\exists C.$ *closedin T C* $\wedge$ $S = f -` C \cap A$›
by *auto*
next
assume ‹$\exists U.$ *closedin T U* $\wedge$ $S = f -` U \cap A$›
then show ‹*closedin* (*pullback-topology A f T*) *S*›
apply (*simp add*: *openin-pullback-topology closedin-def topspace-pullback-topology*)
by *blast*
qed

lemma *regular-space-pullback*[*intro*]:
assumes ‹*regular-space T*›
shows ‹*regular-space* (*pullback-topology A f T*)›
proof (*unfold regular-space-def*, *intro allI impI*)
define *TT PT* **where** ‹$TT = topspace\ T$› **and** ‹$PT = topspace$ (*pullback-topology A f T*)›
fix *S y*
assume *asm*: ‹*closedin* (*pullback-topology A f T*) *S* $\wedge$ $y \in PT - S$›
from *asm* **obtain** *C* **where** ‹*closedin T C*› **and** *S-fCA*: ‹$S = f -` C \cap A$›
by (*auto simp*: *closedin-pullback-topology*)
from *asm S-fCA*
have ‹$f\ y \in TT - C$›
by (*auto simp*: *PT-def TT-def topspace-pullback-topology*)
then obtain *U′ V′* **where** ‹*openin T U′*› **and** ‹*openin T V′*› **and** ‹$f\ y \in U'$› **and** ‹$C \subseteq V'$›
and ‹$U' \cap V' = \{\}$›
by (*metis TT-def* ‹*closedin T C*› *assms regular-space-def disjnt-def*)
define *U V* **where** ‹$U = f -` U' \cap A$› **and** ‹$V = f -` V' \cap A$›
have ‹*openin* (*pullback-topology A f T*) *U*›
using *U-def* ‹*openin T U′*› *openin-pullback-topology* **by** *blast*
moreover have ‹*openin* (*pullback-topology A f T*) *V*›
using *V-def* ‹*openin T V′*› *openin-pullback-topology* **by** *blast*
moreover have ‹$y \in U$›
by (*metis DiffD1 Int-iff PT-def U-def* ‹$f\ y \in U'$› *asm topspace-pullback-topology vimageI*)
moreover have ‹$S \subseteq V$›
using *S-fCA V-def* ‹$C \subseteq V'$› **by** *blast*
moreover have ‹*disjnt U V*›
using *U-def V-def* ‹$U' \cap V' = \{\}$› *disjnt-def* **by** *blast*

ultimately show ‹∃ *U V*. *openin* (*pullback-topology A f T*) *U* ∧ *openin* (*pullback-topology A f T*) *V* ∧ *y* ∈ *U* ∧ *S* ⊆ *V* ∧ *disjnt U V*›
apply (*rule-tac exI*[*of - U*], *rule-tac exI*[*of - V*])
by *auto*
qed

lemma *t3-space-euclidean-regular*[*iff*]: ‹*regular-space* (*euclidean* :: *'a*::*t3-space topology*)›
using *t3-space*
apply (*simp add*: *regular-space-def disjnt-def*)
by *fast*

definition *increasing-filter* :: ‹*'a*::*order filter* ⇒ *bool*› **where**
— Definition suggested by [5]
‹*increasing-filter F* ⟷ (∀ *F x in F*. ∀ *F y in F*. *y* ≥ *x*)›

lemma *increasing-filtermap*:
fixes *F* :: ‹*'a*::*order filter*› **and** *f* :: ‹*'a* ⇒ *'b*::*order*› **and** *X* :: ‹*'a set*›
assumes *increasing*: ‹*increasing-filter F*›
assumes *mono*: ‹*mono-on X f*›
assumes *ev-X*: ‹*eventually* (λ*x*. *x* ∈ *X*) *F*›
shows ‹*increasing-filter* (*filtermap f F*)›
proof −
from *increasing*
have *incr*: ‹∀ *F x in F*. ∀ *F y in F*. *x* ≤ *y*›
apply (*simp add*: *increasing-filter-def*)
by −
have ‹∀ *F x in F*. ∀ *F y in F*. *f x* ≤ *f y*›
proof (*rule eventually-elim2*[*OF ev-X incr*])
fix *x*
assume ‹*x* ∈ *X*›
assume ‹∀ *F y in F*. *x* ≤ *y*›
then show ‹∀ *F y in F*. *f x* ≤ *f y*›
proof (*rule eventually-elim2*[*OF ev-X*])
fix *y* **assume** ‹*y* ∈ *X*› **and** ‹*x* ≤ *y*›
with ‹*x* ∈ *X*› **show** ‹*f x* ≤ *f y*›
using *mono* **by** (*simp add*: *mono-on-def*)
qed
qed
then show ‹*increasing-filter* (*filtermap f F*)›
by (*simp add*: *increasing-filter-def eventually-filtermap*)
qed

lemma *increasing-finite-subsets-at-top*[*simp*]: ‹*increasing-filter* (*finite-subsets-at-top X*)›
apply (*simp add*: *increasing-filter-def eventually-finite-subsets-at-top*)
by *force*

lemma *monotone-convergence*:
— Following [5]

```
  fixes f :: ‹'b ⇒ 'a::{order-topology, conditionally-complete-linorder}›
  assumes bounded: ‹∀⇩F x in F. f x ≤ B›
  assumes increasing: ‹increasing-filter (filtermap f F)›
  shows ‹∃ l. (f ⟶⇩F l) F›
proof (cases ‹F ≠ ⊥›)
  case True
  note [simp] = True
  define S l where ‹S x ⟷ (∀⇩F y in F. f y ≥ x) ∧ x ≤ B›
    and ‹l = Sup (Collect S)› for x
  from bounded increasing
  have ev-S: ‹eventually S (filtermap f F)›
   by (auto intro!: eventually-conj simp: S-def[abs-def] increasing-filter-def eventually-filtermap)
  have bdd-S: ‹bdd-above (Collect S)›
    by (auto simp: S-def)
  have S-nonempty: ‹Collect S ≠ {}›
    using ev-S
    by (metis Collect-empty-eq-bot Set.empty-def True eventually-False filtermap-bot-iff)
  have ‹(f ⟶⇩F l) F›
  proof (rule order-tendstoI; rename-tac x)
    fix x
    assume ‹x < l›
    then obtain s where ‹S s› and ‹x < s›
      using less-cSupD[OF S-nonempty] l-def
      by blast
    then
    show ‹∀⇩F y in F. x < f y›
      using S-def basic-trans-rules(22) eventually-mono by force
  next
    fix x
    assume asm: ‹l < x›
    from ev-S
    show ‹∀⇩F y in F. f y < x›
      unfolding eventually-filtermap
      apply (rule eventually-mono)
      using asm
      by (metis bdd-S cSup-upper dual-order.strict-trans2 l-def mem-Collect-eq)
  qed
  then show ‹∃ l. (f ⟶⇩F l) F›
    by (auto intro!: exI[of - l] simp: filterlim-def)
next
  case False
  then show ‹∃ l. (f ⟶⇩F l) F›
    by (auto intro!: exI)
qed


lemma monotone-convergence-complex:
  fixes f :: ‹'b ⇒ complex›
  assumes bounded: ‹∀⇩F x in F. f x ≤ B›
```

```
  assumes increasing: ‹increasing-filter (filtermap f F)›
  shows ‹∃ l. (f ⟶ l) F›
proof −
  have inc-re: ‹increasing-filter (filtermap (λx. Re (f x)) F)›
    using increasing-filtermap[OF increasing, where f=Re and X=UNIV]
    by (simp add: less-eq-complex-def[abs-def] mono-def monotone-def filtermap-filtermap)
  from bounded have ‹∀⇩F x in F. Re (f x) ≤ Re B›
    using eventually-mono less-eq-complex-def by fastforce
  from monotone-convergence[OF this inc-re]
  obtain re where lim-re: ‹((λx. Re (f x)) ⟶ re) F›
    by auto
  from bounded have ‹∀⇩F x in F. Im (f x) = Im B›
    by (simp add: less-eq-complex-def[abs-def] eventually-mono)
  then have lim-im: ‹((λx. Im (f x)) ⟶ Im B) F›
    by (simp add: tendsto-eventually)
  from lim-re lim-im have ‹(f ⟶ Complex re (Im B)) F›
    by (simp add: tendsto-complex-iff)
  then show ?thesis
    by auto
qed

lemma compact-closed-subset:
  assumes ‹compact s›
  assumes ‹closed t›
  assumes ‹t ⊆ s›
  shows ‹compact t›
  by (metis assms(1) assms(2) assms(3) compact-Int-closed inf.absorb-iff2)

definition separable where ‹separable S ⟷ (∃ B. countable B ∧ S ⊆ closure B)›

lemma compact-imp-separable: ‹separable S› if ‹compact S› for S :: ‹'a::metric-space set›
proof −
  from that
  obtain K where ‹finite (K n)› and K-cover-S: ‹S ⊆ (⋃k∈K n. ball k (1 / of-nat (n+1)))›
for n :: nat
  proof (atomize-elim, intro choice2 allI)
    fix n
    have ‹S ⊆ (⋃k∈UNIV. ball k (1 / of-nat (n+1)))›
      apply (auto intro!: simp: )
      by (smt (verit, del-insts) dist-eq-0-iff linordered-field-class.divide-pos-pos of-nat-less-0-iff)
    then show ‹∃ K. finite K ∧ S ⊆ (⋃k∈K. ball k (1 / real (n + 1)))›
      apply (simp add: compact-eq-Heine-Borel)
      by (meson Elementary-Metric-Spaces.open-ball compactE-image ‹compact S›)
  qed
  define B where ‹B = (⋃ n. K n)›
  have ‹countable B›
    using B-def ‹finite (K -)› uncountable-infinite by blast
  have ‹S ⊆ closure B›
  proof (intro subsetI closure-approachable[THEN iffD2, rule-format])
```

```
    fix x assume ‹x ∈ S›
    fix e :: real assume ‹e > 0›
    define n :: nat where ‹n = nat (ceiling (1/e))›
    with ‹e > 0› have ne: ‹1 / real (n+1) ≤ e›
    proof −
      have ‹1 / real (n+1) ≤ 1 / ceiling (1/e)›
        by (simp add: ‹0 < e› linordered-field-class.frac-le n-def)
      also have ‹... ≤ 1 / (1/e)›
          by (smt (verit, del-insts) ‹0 < e› le-of-int-ceiling linordered-field-class.divide-pos-pos
linordered-field-class.frac-le)
      also have ‹... = e›
        by simp
      finally show ?thesis
        by −
    qed
    have ‹S ⊆ (⋃k∈K n. ball k (1 / of-nat (n+1)))›
      using K-cover-S by presburger
    then obtain k where ‹k ∈ K n› and x-ball: ‹x ∈ ball k (1 / of-nat (n+1))›
      using ‹x ∈ S› by auto
    from ‹k ∈ K n› have ‹k ∈ B›
      using B-def by blast
    moreover from x-ball have ‹dist k x < e›
      by (smt (verit) ne mem-ball)
    ultimately show ‹∃k∈B. dist k x < e›
      by fast
  qed
  show ‹separable S›
    using ‹S ⊆ closure B› ‹countable B› separable-def by blast
qed

lemma ex-norm1-not-singleton:
  shows ‹∃x::'a::{real-normed-vector, not-singleton}. norm x = 1›
  apply (rule ex-norm1)
  by simp

lemma is-Sup-approx-below:
  fixes b :: ‹'a::linordered-ab-group-add›
  assumes ‹is-Sup A b›
  assumes ‹ε > 0›
  shows ‹∃x∈A. b − ε ≤ x›
proof (rule ccontr)
  assume ‹¬ (∃x∈A. b − ε ≤ x)›
  then have ‹b − ε ≥ x› if ‹x ∈ A› for x
    using that by auto
  with assms
  have ‹b ≤ b − ε›
    by (simp add: is-Sup-def)
  with ‹ε > 0›
  show False
```

by *simp*
qed

lemma *sums-le-complex*:
fixes *f g* :: *nat* ⇒ *complex*
assumes ‹⋀*n*. *f n* ≤ *g n*›
assumes ‹*f sums s*›
assumes ‹*g sums t*›
shows ‹*s* ≤ *t*›
proof −
have ‹*Re* (*f n*) ≤ *Re* (*g n*)› **for** *n*
by (*simp add*: *Re-mono assms*(*1*))
moreover have ‹(λ*n*. *Re* (*f n*)) *sums Re s*›
using *assms*(*2*) *sums-Re* **by** *auto*
moreover have ‹(λ*n*. *Re* (*g n*)) *sums Re t*›
using *assms*(*3*) *sums-Re* **by** *auto*
ultimately have *re*: ‹*Re s* ≤ *Re t*›
by (*rule sums-le*)
have ‹*Im* (*f n*) = *Im* (*g n*)› **for** *n*
by (*simp add*: *assms*(*1*) *comp-Im-same*)
moreover have ‹(λ*n*. *Im* (*f n*)) *sums Im s*›
using *assms*(*2*) *sums-Im* **by** *auto*
moreover have ‹(λ*n*. *Im* (*g n*)) *sums Im t*›
using *assms*(*3*) *sums-Im* **by** *auto*
ultimately have *im*: ‹*Im s* = *Im t*›
using *sums-unique2* **by** *auto*
from *re im* **show** ‹*s* ≤ *t*›
using *less-eq-complexI* **by** *auto*
qed

lemma *infsum-single*:
assumes ⋀*j*. *j* ≠ *i* ⟹ *j*∈*A* ⟹ *f j* = *0*
shows *infsum f A* = (*if i*∈*A then f i else 0*)
apply (*subst infsum-cong-neutral*[**where** *T*=‹*A* ∩ {*i*}› **and** *g*=*f*])
using *assms* **by** *auto*

lemma *suminf-eqI*:
fixes *x* :: ‹-::{*comm-monoid-add*,*t2-space*}›
assumes ‹*f sums x*›
shows ‹*suminf f* = *x*›
using *assms*
by (*auto intro*!: *simp*: *sums-iff*)

lemma *suminf-If-finite-set*:
fixes *f* :: ‹- ⇒ -::{*comm-monoid-add*,*t2-space*}›
assumes ‹*finite F*›
shows ‹(∑ *x*∈*F*. *f x*) = (∑ *x*. *if x*∈*F then f x else 0*)›
by (*auto intro*!: *suminf-eqI*[*symmetric*] *sums-If-finite-set simp*: *assms*)

definition *separating-set* :: ‹(('a ⇒ 'b) ⇒ bool) ⇒ 'a set ⇒ bool› **where**
‹*separating-set P S* ⟷ (∀ *f g*. *P f* ⟶ *P g* ⟶ (∀ *x*∈*S*. *f x* = *g x*) ⟶ *f* = *g*)›

lemma *separating-set-mono*: ‹*S* ⊆ *T* ⟹ *separating-set P S* ⟹ *separating-set P T*›
unfolding *separating-set-def* **by** *fast*

lemma *separating-set-UNIV*[*simp*]: ‹*separating-set P UNIV*›
by (*auto intro*!: *ext simp*: *separating-set-def*)

lemma *eq-from-separatingI*:
assumes ‹*separating-set P S*›
assumes ‹*P f*› **and** ‹*P g*›
assumes ‹⋀*x*. *x* ∈ *S* ⟹ *f x* = *g x*›
shows ‹*f* = *g*›
using *assms* **by** (*simp add*: *separating-set-def*)

lemma *eq-from-separatingI2*:
assumes ‹*separating-set P* ((λ(*x*,*y*). *h x y*) ‘ (*S*×*T*))›
assumes ‹*P f*› **and** ‹*P g*›
assumes ‹⋀*x y*. *x* ∈ *S* ⟹ *y* ∈ *T* ⟹ *f* (*h x y*) = *g* (*h x y*)›
shows ‹*f* = *g*›
apply (*rule eq-from-separatingI*[*OF assms*(*1*)])
using *assms*(*2*−*4*) **by** *auto*

lemma *separating-setI*:
assumes ‹⋀*f g*. *P f* ⟹ *P g* ⟹ (⋀*x*. *x*∈*S* ⟹ *f x* = *g x*) ⟹ *f* = *g*›
shows ‹*separating-set P S*›
by (*simp add*: *assms separating-set-def*)

lemma *tendsto-le-complex*:
fixes *x y* :: *complex*
assumes *F*: ¬ *trivial-limit F*
and *x*: (*f* ⟶⟶ *x*) *F*
and *y*: (*g* ⟶⟶ *y*) *F*
and *ev*: *eventually* (λ*x*. *g x* ≤ *f x*) *F*
shows *y* ≤ *x*
proof (*rule less-eq-complexI*)
note *F*
moreover have ‹((λ*x*. *Re* (*f x*)) ⟶⟶ *Re x*) *F*›
by (*simp add*: *assms tendsto-Re*)
moreover have ‹((λ*x*. *Re* (*g x*)) ⟶⟶ *Re y*) *F*›
by (*simp add*: *assms tendsto-Re*)
moreover from *ev* **have** *eventually* (λ*x*. *Re* (*g x*) ≤ *Re* (*f x*)) *F*
apply (*rule eventually-mono*)
by (*simp add*: *less-eq-complex-def*)
ultimately show ‹*Re y* ≤ *Re x*›
by (*rule tendsto-le*)
next

note *F*
moreover have ‹((λ*x*. *Im* (*g x*)) ⟶ *Im y*) *F*›
by (*simp add*: *assms tendsto-Im*)
moreover
have ‹((λ*x*. *Im* (*g x*)) ⟶ *Im x*) *F*›
proof −
have ‹((λ*x*. *Im* (*f x*)) ⟶ *Im x*) *F*›
by (*simp add*: *assms tendsto-Im*)
moreover from *ev* **have** ‹∀ *F x in F*. *Im* (*f x*) = *Im* (*g x*)›
apply (*rule eventually-mono*)
by (*simp add*: *less-eq-complex-def*)
ultimately show *?thesis*
by (*rule Lim-transform-eventually*)
qed
ultimately show ‹*Im y* = *Im x*›
by (*rule tendsto-unique*)
qed

lemma *bdd-above-mono2*:
assumes ‹*bdd-above* (*g* ‘ *B*)›
assumes ‹*A* ⊆ *B*›
assumes ‹⋀*x*. *x* ∈ *A* ⟹ *f x* ≤ *g x*›
shows ‹*bdd-above* (*f* ‘ *A*)›
by (*smt* (*verit*, *del-insts*) *Set.basic-monos*(*7*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *basic-trans-rules*(*23*) *bdd-above.I2 bdd-above.unfold imageI*)

end

2 *Misc-Tensor-Product-BO* − Miscelleanous results missing from `Complex_Bounded_Operators`

theory *Misc-Tensor-Product-BO*
imports
Complex-Bounded-Operators.Complex-L2
Misc-Tensor-Product
HOL−Library.Function-Algebras
begin

no-notation *Set-Algebras.elt-set-eq* (**infix** =*o* *50*)

unbundle *cblinfun-notation*

instance *cblinfun* :: (*chilbert-space*,*chilbert-space*) *ordered-comm-monoid-add*
by *intro-classes*

lemma *rank1-scaleR*[*simp*]: ‹*rank1* (*c* ∗*R a*)› **if** ‹*rank1 a*› **and** ‹*c* ≠ *0*›

by (*simp add*: *rank1-scaleC scaleR-scaleC that*(*1*) *that*(*2*))

lemma *rank1-butterfly*[*simp*]: ‹*rank1* (*butterfly x y*)›
apply (*cases* ‹*y* = *0*›)
by (*auto intro*: *exI*[*of* - *0*] *simp*: *rank1-def butterfly-is-rank1*)

definition ‹*cfinite-dim S* ⟷ (∃ *B*. *finite B* ∧ *S* ⊆ *cspan B*)›

lemma *cfinite-dim-subspace-has-basis*:
assumes ‹*cfinite-dim S*› **and** ‹*csubspace S*›
shows ‹∃ *B*. *finite B* ∧ *cindependent B* ∧ *cspan B* = *S*›
proof −
obtain *B* **where** ‹*cindependent B*› **and** ‹*cspan B* = *S*›
by (*rule complex-vector.maximal-independent-subset*[**where** *V*=*S*])
(*use* ‹*csubspace S*› *complex-vector.span-subspace* **in** *blast*)
from ‹*cfinite-dim S*›
obtain *C* **where** ‹*finite C*› **and** ‹*S* ⊆ *cspan C*›
using *cfinite-dim-def* **by** *auto*
from ‹*cspan B* = *S*› **and** ‹*S* ⊆ *cspan C*›
have ‹*B* ⊆ *cspan C*›
using *complex-vector.span-superset* **by** *force*
from ‹*finite C*› ‹*cindependent B*› *this*
have ‹*finite B*›
by (*rule complex-vector.independent-span-bound*[*THEN conjunct1*])
from *this* **and** ‹*cindependent B*› **and** ‹*cspan B* = *S*›
show *?thesis*
by *auto*
qed

lemma *cfinite-dim-subspace-has-onb*:
assumes ‹*cfinite-dim S*› **and** ‹*csubspace S*›
shows ‹∃ *B*. *finite B* ∧ *is-ortho-set B* ∧ *cspan B* = *S* ∧ (∀ *x*∈*B*. *norm x* = *1*)›
proof −
from *assms*
obtain *C* **where** ‹*finite C*› **and** ‹*cindependent C*› **and** ‹*cspan C* = *S*›
using *cfinite-dim-subspace-has-basis* **by** *blast*
obtain *B* **where** ‹*finite B*› **and** ‹*is-ortho-set B*› **and** ‹*cspan B* = *cspan C*›
and *norm*: ‹*x* ∈ *B* ⟹ *norm x* = *1*› **for** *x*
using *orthonormal-basis-of-cspan*[*OF* ‹*finite C*›]
by *blast*
with ‹*cspan C* = *S*› **have** ‹*cspan B* = *S*›
by *simp*
with ‹*finite B*› **and** ‹*is-ortho-set B*› **and** *norm*
show *?thesis*
by *blast*
qed

lemma *cspan-finite-dim*[*intro*]: ‹*cfinite-dim* (*cspan B*)› **if** ‹*finite B*›
using *cfinite-dim-def that* **by** *auto*

lift-definition *finite-dim-ccsubspace* :: ‹'*a*::*complex-normed-vector ccsubspace* ⇒ *bool*› **is** *cfinite-dim*.

lemma *ccspan-finite-dim*[*intro*]: ‹*finite-dim-ccsubspace* (*ccspan B*)› **if** ‹*finite B*›
using *ccspan-finite finite-dim-ccsubspace.rep-eq that* **by** *fastforce*

lemma *finite-dim-ccsubspace-zero*[*iff*]: ‹*finite-dim-ccsubspace 0*›
proof −
have ∗: ‹*cfinite-dim* (*cspan* {*0*})›
by *blast*
show *?thesis*
apply *transfer*
using ∗ **by** *simp*
qed

lemma *finite-dim-ccsubspace-bot*[*iff*]: ‹*finite-dim-ccsubspace* ⊥›
using *finite-dim-ccsubspace-zero* **by** *auto*

lemma *compact-scaleC*:
fixes *s* :: '*a*::*complex-normed-vector set*
assumes *compact s*
shows *compact* (*scaleC c* ‘ *s*)
by (*auto intro*!: *compact-continuous-image assms continuous-at-imp-continuous-on*)

lemma *Proj-nearest*:
assumes ‹*x* ∈ *space-as-set S*›
shows ‹*dist* (*Proj S m*) *m* ≤ *dist x m*›
proof −
have ‹*is-projection-on* (*Proj S*) (*space-as-set S*)›
by (*simp add*: *Proj.rep-eq*)
then have ‹*is-arg-min* (λ*x*. *dist x m*) (λ*x*. *x* ∈ *space-as-set S*) (*Proj S m*)›
by (*simp add*: *is-projection-on-def*)
with *assms* **show** *?thesis*
by (*auto simp*: *is-arg-min-def*)
qed

lemma *norm-cblinfun-bound-unit*:
assumes ‹*b* ≥ *0*›
assumes ‹⋀ψ. *norm* ψ = *1* ⟹ *norm* (*a* ∗*V* ψ) ≤ *b*›
shows ‹*norm a* ≤ *b*›
proof (*rule norm-cblinfun-bound*)
from *assms* **show** ‹*b* ≥ *0*› **by** *simp*
fix *x*

```
  show ‹norm (a *V x) ≤ b * norm x›
  proof (cases ‹x = 0›)
    case True
    then show ?thesis by simp
  next
    case False
    have ‹norm (a *V x) = norm (a *V (norm x *C sgn x))›
      by simp
    also have ‹... = norm (a *V sgn x) * norm x›
      by (simp add: cblinfun.scaleC-right del: norm-scaleC-sgn)
    also have ‹... ≤ (b * norm (sgn x)) * norm x›
      by (simp add: assms(2) norm-sgn)
    also have ‹... = b * norm x›
      by (simp add: norm-sgn)
    finally show ?thesis
      by −
  qed
qed



lemma cblinfun-norm-is-Sup-cinner:
  — [2], Proposition II.2.13
fixes A :: ‹'a::{not-singleton,chilbert-space} ⇒CL 'a›
  assumes Aselfadj: ‹selfadjoint A›
  shows ‹is-Sup ((λψ. cmod (ψ •C (A *V ψ))) ‘ {ψ. norm ψ = 1}) (norm A)›
proof (rule is-SupI)
  fix b assume ‹b ∈ (λψ. cmod (ψ •C (A *V ψ))) ‘ {ψ. norm ψ = 1}›
  then obtain ψ where ‹norm ψ = 1› and b-ψ: ‹b = cmod (ψ •C (A *V ψ))›
    by blast
  have ‹b ≤ norm (A ψ)›
    using b-ψ ‹norm ψ = 1›
    by (metis complex-inner-class.Cauchy-Schwarz-ineq2 mult-cancel-right2)
  also have ‹... ≤ norm A›
    using ‹norm ψ = 1›
    by (metis mult-cancel-left2 norm-cblinfun)
  finally show ‹b ≤ norm A›
    by −
next
  fix c assume asm: ‹(⋀b. b ∈ (λψ. cmod (ψ •C A ψ)) ‘ {ψ. norm ψ = 1} ⟹ b ≤ c)›
  have c-upper: ‹cmod (ψ •C (A *V ψ)) ≤ c› if ‹norm ψ = 1› for ψ
    using that using asm[of ‹cmod (ψ •C (A *V ψ))›] by auto
  have ‹c ≥ 0›
    by (smt (z3) ex-norm1-not-singleton c-upper norm-ge-zero)
  have *: ‹Re (g •C A h) ≤ c› if ‹norm g = 1› and ‹norm h = 1› for g h
  proof −
    have c-upper': ‹cmod (ψ •C (A *V ψ)) ≤ c * (norm ψ)^2› for ψ
      apply (cases ‹ψ = 0›, simp)
      apply (subst (2) norm-scaleC-sgn[symmetric, of ψ])
```

apply (*subst norm-scaleC-sgn*[*symmetric*])
apply (*simp only*: *cinner-scaleC-left cinner-scaleC-right cblinfun.scaleC-right*)
using *c-upper*[*of* ‹*sgn ψ*›]
by (*simp add*: *norm-mult norm-sgn power2-eq-square*)
from *Aselfadj* **have** *Aselfadj'*: $x \cdot_C (A *_V y) = (A *_V x) \cdot_C y$ **for** *x y*
using *cinner-adj-right*[*of x A y*] **by** (*auto simp*: *selfadjoint-def*)
from *Aselfadj* **have** *Aselfadj''*: $(A *_V x) \cdot_C y = cnj ((A *_V y) \cdot_C x)$ **for** *x y*
by (*subst cinner-commute*, *subst Aselfadj'*) *auto*

have *1*: ‹$(h + g) \cdot_C A (h + g) = h \cdot_C A h + 2 * Re (g \cdot_C A h) + g \cdot_C A g$›
by (*simp add*: *cblinfun.cbilinear-simps algebra-simps*
Aselfadj' Aselfadj''[*of h g*] *complex-add-cnj del*: *cinner-commute'*)
from *Aselfadj* **have** *2*: ‹$(h - g) \cdot_C A (h - g) = h \cdot_C A h - 2 * Re (g \cdot_C A h) + g \cdot_C A g$›
by (*simp add*: *cblinfun.cbilinear-simps algebra-simps Aselfadj'*
Aselfadj''[*of h g*] *complex-add-cnj del*: *cinner-commute'*)
have ‹$4 * Re (g \cdot_C A h) = Re ((h + g) \cdot_C A (h + g)) - Re ((h - g) \cdot_C A (h - g))$›
by (*smt* (*verit*, *ccfv-SIG*) *1 2 Re-complex-of-real minus-complex.simps*(*1*) *plus-complex.sel*(*1*))
also have ‹$\dots \leq c * (norm (h + g))^2 - Re ((h - g) \cdot_C A (h - g))$›
using *c-upper'*[*of* ‹$h + g$›]
by (*smt* (*verit*, *best*) *complex-Re-le-cmod*)
also have ‹$\dots \leq c * (norm (h + g))^2 + c * (norm (h - g))^2$›
unfolding *diff-conv-add-uminus*
by (*rule add-left-mono*)
(*use c-upper'*[*of* ‹$h - g$›] **in** ‹*smt* (*verit*) *abs-Re-le-cmod add-uminus-conv-diff*›)
also have ‹$\dots = 2 * c * ((norm h)^2 + (norm g)^2)$›
by (*auto intro*!: *simp*: *polar-identity polar-identity-minus ring-distribs*)
also have ‹$\dots \leq 4 * c$›
by (*simp add*: ‹*norm h = 1*› ‹*norm g = 1*›)
finally show ‹$Re (g \cdot_C (A *_V h)) \leq c$›
by *simp*
qed
have $*$: ‹$cmod (g \cdot_C A h) \leq c$› **if** ‹*norm g = 1*› **and** ‹*norm h = 1*› **for** *g h*
proof −
define γ **where** ‹$\gamma = (if\ g \cdot_C A h = 0\ then\ 1\ else\ sgn (g \cdot_C A h))$›
have γ: ‹$\gamma * cmod (g \cdot_C A h) = g \cdot_C A h$›
by (*simp add*: γ*-def sgn-eq*)
have ‹$norm \gamma = 1$›
by (*simp add*: γ*-def norm-sgn*)
have ‹$cmod (g \cdot_C A h) = Re (complex\text{-}of\text{-}real (norm (g \cdot_C A h)))$›
by *simp*
also have ‹$\dots = Re (g \cdot_C (A (h /_C \gamma)))$›
using γ ‹*cmod* γ *= 1*›
by (*smt* (*verit*) *Groups.mult-ac*(*2*) *Groups.mult-ac*(*3*) *cblinfun.scaleC-right cinner-scaleC-right left-inverse more-arith-simps*(*6*) *norm-eq-zero*)
also have ‹$\dots \leq c$›
using ‹*norm* γ *= 1*›
by (*auto intro*!: $*$ *simp*: *that norm-inverse*)
finally show ‹$cmod (g \cdot_C (A *_V h)) \leq c$›
by −

```
  qed
  have ‹norm (A h) ≤ c› if ‹norm h = 1› for h
    by (cases ‹A h = 0›)
       (use *[OF - that, of ‹sgn (A h)›] in ‹simp-all add: norm-sgn ‹0 ≤ c››)
  then show ‹norm A ≤ c›
    using ‹c ≥ 0› by (auto intro!: norm-cblinfun-bound-unit)
qed

lemma cblinfun-norm-approx-witness-cinner:
  fixes A :: ‹'a::{not-singleton,chilbert-space} ⇒CL 'a›
  assumes ‹selfadjoint A› and ‹ε > 0›
  shows ‹∃ψ. cmod (ψ •C (A *V ψ)) ≥ norm A − ε ∧ norm ψ = 1›
  using is-Sup-approx-below[OF cblinfun-norm-is-Sup-cinner[OF assms(1)] assms(2)]
  by blast

lemma cblinfun-norm-approx-witness-cinner':
  fixes A :: ‹'a::chilbert-space ⇒CL 'a›
  assumes ‹selfadjoint A› and ‹ε > 0›
  shows ‹∃ψ. cmod (ψ •C A ψ) / (norm ψ)^2 ≥ norm A − ε›
proof (cases ‹class.not-singleton TYPE('a)›)
  case True
  obtain ψ where ‹cmod (ψ •C A ψ) ≥ norm A − ε› and ‹norm ψ = 1›
    apply atomize-elim
    using chilbert-space-axioms True assms
    by (rule cblinfun-norm-approx-witness-cinner[internalize-sort' 'a])
  then have ‹cmod (ψ •C A ψ) / (norm ψ)^2 ≥ norm A − ε›
    by simp
  then show ?thesis
    by auto
next
  case False
  show ?thesis
    apply (subst not-not-singleton-cblinfun-zero[OF False])
     apply simp
    apply (subst not-not-singleton-cblinfun-zero[OF False])
    using ‹ε > 0› by simp
qed

lemma has-sum-mono-neutral-cblinfun:
  fixes f :: 'a ⇒ ('b::chilbert-space ⇒CL 'b)
  assumes ‹(f has-sum a) A› and (g has-sum b) B
  assumes ‹⋀x. x ∈ A∩B ⟹ f x ≤ g x›
  assumes ‹⋀x. x ∈ A−B ⟹ f x ≤ 0›
  assumes ‹⋀x. x ∈ B−A ⟹ g x ≥ 0›
  shows a ≤ b
proof −
  from assms
  have sum-hfh: ‹((λx. h •C f x h) has-sum h •C a h) A› for h
    by (intro has-sum-cinner-left has-sum-cblinfun-apply-left)
```

from *assms*
have *sum-hgh*: ‹((λx. h •$_C$ g x h) has-sum h •$_C$ b h) B› **for** h
by (*intro has-sum-cinner-left has-sum-cblinfun-apply-left*)
from *sum-hfh sum-hgh*
have ‹h •$_C$ a h ≤ h •$_C$ b h› **for** h
apply (*rule has-sum-mono-neutral-complex*)
using *assms*
by (*auto intro*!: *simp*: *less-eq-cblinfun-def*)
then show ‹a ≤ b›
by (*simp add*: *less-eq-cblinfun-def*)
qed

lemma *sums-mono-cblinfun*:
fixes f :: *nat* ⇒ ('b::*chilbert-space* ⇒$_{CL}$ 'b)
assumes ‹f *sums* a› **and** g *sums* b
assumes ‹⋀n. f n ≤ g n›
shows a ≤ b
proof (*rule cblinfun-leI*)
fix h
from ‹f *sums* a›
have *sum1*: ‹(λn. h •$_C$ (f n *$_V$ h)) *sums* (h •$_C$ (a *$_V$ h))›
apply (*rule bounded-linear.sums*[*rotated*])
using *bounded-clinear.bounded-linear bounded-clinear-cinner-right bounded-linear-compose cblinfun.real.bounded-linear-left* **by** *blast*
from ‹g *sums* b›
have *sum2*: ‹(λn. h •$_C$ (g n *$_V$ h)) *sums* (h •$_C$ (b *$_V$ h))›
apply (*rule bounded-linear.sums*[*rotated*])
by (*metis bounded-linear-compose cblinfun.real.bounded-linear-left cblinfun.real.bounded-linear-right cblinfun-cinner-right.rep-eq*)
have ‹h •$_C$ (f n *$_V$ h) ≤ h •$_C$ (g n *$_V$ h)› **for** n
using *assms*(*3*) *less-eq-cblinfun-def* **by** *auto*
with *sum1 sum2*
show ‹h •$_C$ (a *$_V$ h) ≤ h •$_C$ (b *$_V$ h)›
by (*rule sums-le-complex*[*rotated*])
qed

lemma *scaleC-scaleR-commute*: ‹a *$_C$ b *$_R$ x = b *$_R$ a *$_C$ x› **for** x :: ‹-::*complex-normed-vector*›
by (*simp add*: *scaleR-scaleC scaleC-left-commute*)

lemma *sandwich-scaleC-left*: ‹*sandwich* (c *$_C$ e) = (*cmod* c)^2 *$_C$ *sandwich* e›
by (*auto intro*!: *cblinfun-eqI simp*: *sandwich-apply cnj-x-x abs-complex-def*)

lemma *sandwich-scaleR-left*: ‹*sandwich* (r *$_R$ e) = r^2 *$_R$ *sandwich* e›
by (*simp add*: *scaleR-scaleC sandwich-scaleC-left flip*: *of-real-power*)

lemma *infsum-product*:
fixes f :: ‹'a ⇒ 'c :: {*topological-semigroup-mult*,*division-ring*,*banach*}›
assumes ‹(λ(x, y). f x * g y) *summable-on* X × Y›

shows ‹$(\sum_\infty x{\in}X.\ f\ x) * (\sum_\infty y{\in}Y.\ g\ y) = (\sum_\infty (x,y){\in}X\times Y.\ f\ x * g\ y)$›
using *assms*
by (*simp add*: *infsum-cmult-right' infsum-cmult-left' flip*: *infsum-Sigma'-banach*)

lemma *infsum-product'*:
fixes $f :: \langle 'a \Rightarrow 'c :: \{banach,times,real\text{-}normed\text{-}algebra\}\rangle$ **and** $g :: \langle 'b \Rightarrow 'c\rangle$
assumes ‹*f abs-summable-on X*›
assumes ‹*g abs-summable-on Y*›
shows ‹$(\sum_\infty x{\in}X.\ f\ x) * (\sum_\infty y{\in}Y.\ g\ y) = (\sum_\infty (x,y){\in}X\times Y.\ f\ x * g\ y)$›
using *assms*
by (*simp add*: *abs-summable-times infsum-cmult-right infsum-cmult-left abs-summable-summable flip*: *infsum-Sigma'-banach*)

lemma *Proj-o-Proj-subspace-right*:
assumes ‹$A \geq B$›
shows ‹*Proj A* o_{CL} *Proj B* = *Proj B*›
by (*simp add*: *Proj-compose-cancelI assms*)

lemma *Proj-o-Proj-subspace-left*:
assumes ‹$A \leq B$›
shows ‹*Proj A* o_{CL} *Proj B* = *Proj A*›
by (*metis Proj-o-Proj-subspace-right adj-Proj adj-cblinfun-compose assms*)

lemma *orthogonal-spaces-SUP-left*:
assumes ‹$\bigwedge x.\ x \in X \Longrightarrow$ *orthogonal-spaces* $(A\ x)\ B$›
shows ‹*orthogonal-spaces* $(\bigsqcup x{\in}X.\ A\ x)\ B$›
by (*meson SUP-least assms orthogonal-spaces-leq-compl*)

lemma *orthogonal-spaces-SUP-right*:
assumes ‹$\bigwedge x.\ x \in X \Longrightarrow$ *orthogonal-spaces* $A\ (B\ x)$›
shows ‹*orthogonal-spaces* A $(\bigsqcup x{\in}X.\ B\ x)$›
by (*meson assms orthogonal-spaces-SUP-left orthogonal-spaces-sym*)

lemma *orthogonal-bot-left*[*simp*]: ‹*orthogonal-spaces bot S*›
by (*simp add*: *orthogonal-spaces-def*)

lemma *infsum-bounded-linear-invertible*:
assumes ‹*bounded-linear h*›
assumes ‹*bounded-linear h'*›
assumes ‹$h'\ o\ h = id$›
shows ‹*infsum* $(\lambda x.\ h\ (f\ x))\ A = h\ (infsum\ f\ A)$›
proof (*cases* ‹*f summable-on A*›)
case *True*
then show *?thesis*
using *assms(1) infsum-bounded-linear* **by** *blast*
next
case *False*
have ‹$\neg$ $(\lambda x.\ h\ (f\ x))$ *summable-on A*›

proof (*rule ccontr*)
assume ‹¬ ¬ (λ*x*. *h* (*f x*)) *summable-on A*›
with ‹*bounded-linear h'*› **have** ‹*h'* *o* *h* *o* *f summable-on A*›
by (*auto intro*: *summable-on-bounded-linear simp*: *o-def*)
then have ‹*f summable-on A*›
by (*simp add*: *assms*(*3*))
with *False* **show** *False*
by *blast*
qed
then show *?thesis*
by (*simp add*: *False assms*(*1*) *infsum-not-exists linear-simps*(*3*))
qed

lemma *cblinfun-eq-from-separatingI*:
fixes *a b* :: ‹'*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector*›
assumes ‹*separating-set* (*bounded-clinear* :: ('*a* ⇒ '*b*) ⇒ *bool*) *S*›
assumes ‹⋀*x*. *x* ∈ *S* ⟹ *a x* = *b x*›
shows ‹*a* = *b*›
apply (*rule cblinfun-eqI*, *rule fun-cong*[**where** *f*=‹*cblinfun-apply* -›])
using *assms*(*1*) **apply** (*rule eq-from-separatingI*)
using *assms*(*2*) **by** (*auto intro*!: *bounded-cbilinear-apply-bounded-clinear cblinfun.bounded-cbilinear-axioms simp*:)

lemma *cblinfun-eq-from-separatingI2*:
fixes *a b* :: ‹'*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector*›
assumes ‹*separating-set* (*bounded-clinear* :: ('*a* ⇒ '*b*) ⇒ *bool*) ((λ(*x*,*y*). *h x y*) ‘ (*S*×*T*))›
assumes ‹⋀*x y*. *x* ∈ *S* ⟹ *y* ∈ *T* ⟹ *a* (*h x y*) = *b* (*h x y*)›
shows ‹*a* = *b*›
apply (*rule cblinfun-eqI*, *rule fun-cong*[**where** *f*=‹*cblinfun-apply* -›])
using *assms*(*1*) **apply** (*rule eq-from-separatingI2*)
using *assms*(*2*) **by** (*auto intro*!: *bounded-cbilinear-apply-bounded-clinear cblinfun.bounded-cbilinear-axioms simp*:)

lemma *separating-set-bounded-clinear-dense*:
assumes ‹*ccspan S* = ⊤›
shows ‹*separating-set bounded-clinear S*›
unfolding *separating-set-def*
by (*intro allI impI ext*, *rule bounded-clinear-eq-on-closure*[**where** *G*=*S*])
(*use assms ccspan.rep-eq* **in** *force*)+

lemma *separating-set-ket*: ‹*separating-set bounded-clinear* (*range ket*)›
by (*simp add*: *bounded-clinear-equal-ket separating-setI*)

lemma *separating-set-bounded-cbilinear-nested*:
assumes ‹*separating-set* (*bounded-clinear* :: (- => '*e*::*complex-normed-vector*) ⇒ -) ((λ(*x*, *y*). *h x y*) ‘ (*UNIV* × *UNIV*))›
assumes ‹*bounded-cbilinear h*›
assumes ‹*separating-set* (*bounded-clinear* :: (- => '*e*) ⇒ -) *A*›
assumes ‹*separating-set* (*bounded-clinear* :: (- => '*e*) ⇒ -) *B*›

shows ‹*separating-set* (*bounded-clinear* :: (- => 'e) ⇒ -) ((λ(x,y). *h x y*) ‘ (*A* × *B*))›
proof (*rule separating-setI*)
fix *f g* :: ‹'a ⇒ 'e›
assume [*simp*]: ‹*bounded-clinear f*› ‹*bounded-clinear g*›
have [*simp*]: ‹*bounded-clinear* (λ*x*. *f* (*h x y*))› **for** *y*
apply (*rule bounded-clinear-compose*[*OF* ‹*bounded-clinear f*›])
using *assms*(*2*) **by** (*rule bounded-cbilinear.bounded-clinear-left*)
have [*simp*]: ‹*bounded-clinear* (λ*x*. *g* (*h x y*))› **for** *y*
apply (*rule bounded-clinear-compose*[*OF* ‹*bounded-clinear g*›])
using *assms*(*2*) **by** (*rule bounded-cbilinear.bounded-clinear-left*)
have [*simp*]: ‹*bounded-clinear* (λ*y*. *f* (*h x y*))› **for** *x*
apply (*rule bounded-clinear-compose*[*OF* ‹*bounded-clinear f*›])
using *assms*(*2*) **by** (*rule bounded-cbilinear.bounded-clinear-right*)
have [*simp*]: ‹*bounded-clinear* (λ*y*. *g* (*h x y*))› **for** *x*
apply (*rule bounded-clinear-compose*[*OF* ‹*bounded-clinear g*›])
using *assms*(*2*) **by** (*rule bounded-cbilinear.bounded-clinear-right*)

assume ‹*z* ∈ (λ(*x*, *y*). *h x y*) ‘ (*A* × *B*) ⟹ *f z* = *g z*› **for** *z*
then have ‹*f* (*h x y*) = *g* (*h x y*)› **if** ‹*x* ∈ *A*› **and** ‹*y* ∈ *B*› **for** *x y*
using *that* **by** *auto*
then have ‹(λ*x*. *f* (*h x y*)) = (λ*x*. *g* (*h x y*))› **if** ‹*y* ∈ *B*› **for** *y*
by (*intro eq-from-separatingI*[*OF assms*(*3*)]) (*use that* **in** *auto*)
then have ‹(λ*y*. *f* (*h x y*)) = (λ*y*. *g* (*h x y*))› **for** *x*
apply (*intro eq-from-separatingI*[*OF assms*(*4*)])
subgoal by *simp*
subgoal by *simp*
subgoal by *meson*
done
then have ‹*f* (*h x y*) = *g* (*h x y*)› **for** *x y*
by *meson*
with ‹*bounded-clinear f*› ‹*bounded-clinear g*›
show ‹*f* = *g*›
by (*rule eq-from-separatingI2*[**where** *f*=*f* **and** *g*=*g* **and** *P*=*bounded-clinear* **and** *S*=*UNIV* **and** *T*=*UNIV*, *rotated 1*])
(*fact assms*(*1*))
qed

lemma *separating-set-bounded-clinear-antilinear*:
assumes ‹*separating-set* (*bounded-clinear* :: (- => 'e::*complex-normed-vector conjugate-space*) ⇒ -) *A*›
shows ‹*separating-set* (*bounded-antilinear* :: (- => 'e) ⇒ -) *A*›
proof (*rule separating-setI*)
fix *f g* :: ‹'a ⇒ 'e›
assume ‹*bounded-antilinear f*›
then have *lin-f*: ‹*bounded-clinear* (*to-conjugate-space o f*)›
by (*simp add*: *bounded-antilinear-o-bounded-antilinear'*)
assume ‹*bounded-antilinear g*›
then have *lin-g*: ‹*bounded-clinear* (*to-conjugate-space o g*)›

```
    by (simp add: bounded-antilinear-o-bounded-antilinear')
  assume ‹f x = g x› if ‹x ∈ A› for x
  then have ‹(to-conjugate-space o f) x = (to-conjugate-space o g) x› if ‹x ∈ A› for x
    by (simp add: that)
  with lin-f lin-g
  have ‹to-conjugate-space o f = to-conjugate-space o g›
    by (rule eq-from-separatingI[OF assms])
  then show ‹f = g›
    by (metis UNIV-I fun.inj-map-strong to-conjugate-space-inverse)
qed

lemma separating-set-bounded-sesquilinear-nested:
  assumes ‹separating-set (bounded-clinear :: (- => 'e::complex-normed-vector) ⇒ -) ((λ(x, y).
h x y) ` (UNIV × UNIV))›
  assumes ‹bounded-sesquilinear h›
  assumes sep-A: ‹separating-set (bounded-clinear :: (- => 'e conjugate-space) ⇒ -) A›
  assumes sep-B: ‹separating-set (bounded-clinear :: (- => 'e) ⇒ -) B›
  shows ‹separating-set (bounded-clinear :: (- => 'e) ⇒ -) ((λ(x,y). h x y) ` (A × B))›
proof (rule separating-setI)
  fix f g :: ‹'a ⇒ 'e›
  assume [simp]: ‹bounded-clinear f› ‹bounded-clinear g›
  have [simp]: ‹bounded-antilinear (λx. f (h x y))› for y
    apply (rule bounded-clinear-o-bounded-antilinear[OF ‹bounded-clinear f›])
    using assms(2) by (rule bounded-sesquilinear.bounded-antilinear-left)
  have [simp]: ‹bounded-antilinear (λx. g (h x y))› for y
    apply (rule bounded-clinear-o-bounded-antilinear[OF ‹bounded-clinear g›])
    using assms(2) by (rule bounded-sesquilinear.bounded-antilinear-left)
  have [simp]: ‹bounded-clinear (λy. f (h x y))› for x
    apply (rule bounded-clinear-compose[OF ‹bounded-clinear f›])
    using assms(2) by (rule bounded-sesquilinear.bounded-clinear-right)
  have [simp]: ‹bounded-clinear (λy. g (h x y))› for x
    apply (rule bounded-clinear-compose[OF ‹bounded-clinear g›])
    using assms(2) by (rule bounded-sesquilinear.bounded-clinear-right)

  from sep-A have sep-A': ‹separating-set (bounded-antilinear :: (- => 'e) ⇒ -) A›
    by (rule separating-set-bounded-clinear-antilinear)
  assume ‹z ∈ (λ(x, y). h x y) ` (A × B) ⟹ f z = g z› for z
  then have ‹f (h x y) = g (h x y)› if ‹x ∈ A› and ‹y ∈ B› for x y
    using that by auto
  then have ‹(λx. f (h x y)) = (λx. g (h x y))› if ‹y ∈ B› for y
    by (intro eq-from-separatingI[OF sep-A']) (use that in auto)
  then have ‹(λy. f (h x y)) = (λy. g (h x y))› for x
    apply (intro eq-from-separatingI[OF sep-B])
    subgoal by simp
    subgoal by simp
    subgoal by meson
    done
  then have ‹f (h x y) = g (h x y)› for x y
    by meson
```

with ‹*bounded-clinear f*› ‹*bounded-clinear g*›
show ‹*f* = *g*›
by (*rule eq-from-separatingI2*[**where** *f*=*f* **and** *g*=*g* **and** *P*=*bounded-clinear* **and** *S*=*UNIV* **and** *T*=*UNIV*, *rotated 1*])
(*fact assms*(*1*))
qed

lemma *eq-on-ccsubspaces-Sup*:
fixes *a b* :: ‹'*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector*›
assumes ‹⋀*i h*. *i* ∈ *I* ⟹ *h* ∈ *space-as-set* (*X i*) ⟹ *a h* = *b h*›
shows ‹⋀*h*. *h* ∈ *space-as-set* (⨆ *i*∈*I*. *X i*) ⟹ *a h* = *b h*›
proof −
from *assms*
have ‹*X i* ≤ *kernel* (*a* − *b*)› **if** ‹*i* ∈ *I*› **for** *i*
using *that* **by** (*auto intro*!: *ccsubspace-leI simp*: *kernel.rep-eq minus-cblinfun.rep-eq*)
then have ‹(⨆ *i*∈*I*. *X i*) ≤ *kernel* (*a* − *b*)›
by (*simp add*: *SUP-least*)
then show ‹*h* ∈ *space-as-set* (⨆ *i*∈*I*. *X i*) ⟹ *a h* = *b h*› **for** *h*
using *kernel-memberD less-eq-ccsubspace.rep-eq*
by (*metis* (*no-types*, *opaque-lifting*) *cblinfun.diff-left cblinfun.real.diff-right cblinfun.real.zero-left diff-eq-diff-eq double-diff mem-simps*(*6*) *subset-refl*)
qed

lemma *eq-on-ccsubspaces-sup*:
fixes *a b* :: ‹'*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector*›
assumes ‹⋀*h i*. *h* ∈ *space-as-set S* ⟹ *a h* = *b h*›
assumes ‹⋀*h i*. *h* ∈ *space-as-set T* ⟹ *a h* = *b h*›
shows ‹⋀*h*. *h* ∈ *space-as-set* (*S* ⊔ *T*) ⟹ *a h* = *b h*›
apply (*rule eq-on-ccsubspaces-Sup*[**where** *I*=‹{*True*,*False*}› **and** *X*=‹λ*i*. *if i then T else S*›])
using *assms*
apply *presburger*
by *fastforce*

lemma *ccsubspace-contains-unit*:
assumes ‹*E* ≠ ⊥›
shows ‹∃ *h*∈*space-as-set E*. *norm h* = *1*›
proof −
from *assms* **have** ‹*space-as-set E* ≠ {*0*}›
by (*metis bot-ccsubspace.rep-eq space-as-set-inject*)
then obtain h_0 **where** ‹h_0 ∈ *space-as-set E*› **and** ‹h_0 ≠ *0*›
by *auto*
then have ‹*sgn* h_0 ∈ *space-as-set E*›
using *csubspace-space-as-set*
by (*auto intro*!: *complex-vector.subspace-scale*
simp add: *sgn-div-norm scaleR-scaleC*)
moreover from ‹h_0 ≠ *0*› **have** ‹*norm* (*sgn* h_0) = *1*›
by (*simp add*: *norm-sgn*)
ultimately show *?thesis*

by *auto*
qed

lemma *Proj-0-compl*: ‹*Proj S x = 0*› **if** ‹*x ∈ space-as-set (−S)*›
by (*simp add*: *kernel-memberD that*)

lemma *csubspace-has-basis*:
assumes ‹*csubspace S*›
shows ‹∃ *B*. *cindependent B ∧ cspan B = S*›
proof −
obtain *B* **where** ‹*cindependent B*› **and** ‹*cspan B = S*›
by (*rule complex-vector.maximal-independent-subset*[**where** *V=S*])
(*use assms complex-vector.span-subspace* **in** *blast*)
then show *?thesis*
by *auto*
qed

lemma *inj-scaleC*:
fixes *A* :: ‹'*a*::*complex-vector set*›
assumes ‹*c ≠ 0*›
shows ‹*inj-on* (*scaleC c*) *A*›
by (*meson assms inj-onI scaleC-left-imp-eq*)

definition *diagonal-operator* **where** ‹*diagonal-operator f =*
(*if bdd-above* (*range* (*λx*. *cmod* (*f x*))) *then explicit-cblinfun* (*λx y*. *of-bool* (*x=y*) ∗ *f x*) *else 0*)›

lemma *diagonal-operator-exists*:
assumes ‹*bdd-above* (*range* (*λx*. *cmod* (*f x*)))›
shows ‹*explicit-cblinfun-exists* (*λx y*. *of-bool* (*x = y*) ∗ *f x*)›
proof −
from *assms* **obtain** *B* **where** *B*: ‹*cmod* (*f x*) ≤ *B*› **for** *x*
by (*auto simp*: *bdd-above-def*)
show *?thesis*
proof (*rule explicit-cblinfun-exists-bounded*)
fix *S T* :: ‹'*a set*› **and** *ψ* :: ‹'*a* ⇒ *complex*›
assume [*simp*]: ‹*finite S*› ‹*finite T*›
assume ‹*ψ a = 0*› **if** ‹*a ∉ T*› **for** *a*
have ‹(∑ *b*∈*S*. (*cmod* (∑ *a*∈*T*. *ψ a* ∗$_C$ (*of-bool* (*b = a*) ∗ *f b*)))^2)
= (∑ *b*∈*S*. (*cmod* (*of-bool* (*b ∈ T*) ∗ *ψ b* ∗ *f b*))^2)›
apply (*rule sum.cong*[*OF refl*])
subgoal for *b*
apply (*subst sum-single*[**where** *i=b*])
by *auto*
by −
also have ‹*...* = (∑ *b*∈*S*∩*T*. (*cmod* (*ψ b* ∗ *f b*))^2)›

```
    apply (rule sum.mono-neutral-cong-right)
    by auto
  also have ‹... ≤ (∑ b∈T. (cmod (ψ b * f b))^2)›
    by (simp add: sum-mono2)
  also have ‹... ≤ (∑ b∈T. B^2 * (cmod (ψ b))^2)›
    by (rule sum-mono)
      (auto intro!: mult-left-mono  power-mono B
            simp: norm-mult power-mult-distrib mult.commute[of B ^ 2])
  also have ‹... = B^2 * (∑ b∈T. (cmod (ψ b))^2)›
    by (simp add: vector-space-over-itself.scale-sum-right)
  finally
  show ‹(∑ b∈S. (cmod (∑ a∈T. ψ a *_C (of-bool (b = a) * f b)))^2)
    ≤ B^2 * (∑ a∈T. (cmod (ψ a))^2)› .
 qed
qed


lemma diagonal-operator-ket:
 assumes ‹bdd-above (range (λx. cmod (f x)))›
 shows ‹diagonal-operator f (ket x) = f x *_C ket x›
proof −
 have [simp]: ‹has-ell2-norm (λb. of-bool (b = x) * f b)›
   by (auto intro!: finite-nonzero-values-imp-summable-on simp: has-ell2-norm-def)
 have ‹Abs-ell2 (λb. of-bool (b = x) * f b) = f x *_C ket x›
   by (rule Rep-ell2-inject[THEN iffD1])
     (auto simp: Abs-ell2-inverse scaleC-ell2.rep-eq ket.rep-eq)
 then show ?thesis
  by (auto intro!: simp: diagonal-operator-def assms explicit-cblinfun-ket diagonal-operator-exists)
qed

lemma diagonal-operator-invalid:
 assumes ‹¬ bdd-above (range (λx. cmod (f x)))›
 shows ‹diagonal-operator f = 0›
 by (simp add: assms diagonal-operator-def)


lemma diagonal-operator-adj: ‹diagonal-operator f* = diagonal-operator (λx. cnj (f x))›
 by (cases ‹bdd-above (range (λx. cmod (f x)))›)
   (auto intro!: equal-ket cinner-ket-eqI
        simp: diagonal-operator-ket cinner-adj-right diagonal-operator-invalid)

lemma diagonal-operator-comp:
 assumes ‹bdd-above (range (λx. cmod (f x)))›
 assumes ‹bdd-above (range (λx. cmod (g x)))›
 shows ‹diagonal-operator f o_CL diagonal-operator g = diagonal-operator (λx. (f x * g x))›
proof −
 have ‹bdd-above (range (λx. cmod (f x * g x)))›
 proof −
   from assms(1) obtain F where ‹cmod (f x) ≤ F› for x
```

by (*auto simp*: *bdd-above-def*)
moreover from *assms*(*2*) **obtain** *G* **where** ‹*cmod* (*g x*) ≤ *G*› **for** *x*
by (*auto simp*: *bdd-above-def*)
ultimately have ‹*cmod* (*f x* ∗ *g x*) ≤ *F* ∗ *G*› **for** *x*
by (*smt* (*verit*, *del-insts*) *mult-right-mono norm-ge-zero norm-mult ordered-comm-semiring-class.comm-mult-left-m*
then show *?thesis*
by *fast*
qed
then show *?thesis*
by (*auto intro*!: *equal-ket simp*: *diagonal-operator-ket assms cblinfun.scaleC-right*)
qed

lemma *summable-on-bdd-above-real*: ‹*bdd-above* (*f* ‘ *M*)› **if** ‹*f summable-on M*› **for** *f* :: ‹*'a* ⇒ *real*›
proof −
from *that* **have** ‹*f abs-summable-on M*›
unfolding *summable-on-iff-abs-summable-on-real*[*symmetric*]
by −
then have ‹*bdd-above* (*sum* (λ*x*. *norm* (*f x*)) ‘ {*F*. *F* ⊆ *M* ∧ *finite F*})›
unfolding *abs-summable-iff-bdd-above* **by** *simp*
then have ‹*bdd-above* (*sum* (λ*x*. *norm* (*f x*)) ‘ (λ*x*. {*x*}) ‘ *M*)›
by (*rule bdd-above-mono*) *auto*
then have ‹*bdd-above* ((λ*x*. *norm* (*f x*)) ‘ *M*)›
by (*simp add*: *image-image*)
then show *?thesis*
by (*simp add*: *bdd-above-mono2*)
qed

lemma *separating-set-clinear-cspan*:
assumes ‹*cspan S* = *UNIV*›
shows ‹*separating-set clinear S*›
using *assms*
by (*auto intro*: *complex-vector.linear-eq-on simp*: *separating-set-def*)

lemma *less-eq-cblinfunI*:
fixes *a b* :: ‹*'a* ⇒$_{CL}$ *'a*::*chilbert-space*›
assumes ‹⋀*h*. *h* •$_C$ *a h* ≤ *h* •$_C$ *b h*›
shows ‹*a* ≤ *b*›
using *assms*
by (*simp add*: *less-eq-cblinfun-def*)

end

3 *Strong-Operator-Topology* − Strong operator topology on complex bounded operators

theory *Strong-Operator-Topology*
imports *Misc-Tensor-Product-BO*

begin

unbundle *cblinfun-notation*

typedef (**overloaded**) ('*a*,'*b*) *cblinfun-sot* = ‹*UNIV* :: ('*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector*) *set*› **..**
setup-lifting *type-definition-cblinfun-sot*

instantiation *cblinfun-sot* :: (*complex-normed-vector*, *complex-normed-vector*) *complex-vector*
begin
lift-definition *scaleC-cblinfun-sot* :: ‹*complex* ⇒ ('*a*, '*b*) *cblinfun-sot* ⇒ ('*a*, '*b*) *cblinfun-sot*›
is ‹*scaleC*› **.**
lift-definition *uminus-cblinfun-sot* :: ‹('*a*, '*b*) *cblinfun-sot* ⇒ ('*a*, '*b*) *cblinfun-sot*› **is** *uminus* **.**
lift-definition *zero-cblinfun-sot* :: ‹('*a*, '*b*) *cblinfun-sot*› **is** *0* **.**
lift-definition *minus-cblinfun-sot* :: ‹('*a*, '*b*) *cblinfun-sot* ⇒ ('*a*, '*b*) *cblinfun-sot* ⇒ ('*a*, '*b*) *cblinfun-sot*› **is** *minus* **.**
lift-definition *plus-cblinfun-sot* :: ‹('*a*, '*b*) *cblinfun-sot* ⇒ ('*a*, '*b*) *cblinfun-sot* ⇒ ('*a*, '*b*) *cblinfun-sot*› **is** *plus* **.**
lift-definition *scaleR-cblinfun-sot* :: ‹*real* ⇒ ('*a*, '*b*) *cblinfun-sot* ⇒ ('*a*, '*b*) *cblinfun-sot*› **is** *scaleR* **.**
instance
apply (*intro-classes*; *transfer*)
by (*auto simp add*: *scaleR-scaleC scaleC-add-right scaleC-add-left*)
end

instantiation *cblinfun-sot* :: (*complex-normed-vector*, *complex-normed-vector*) *topological-space*
begin
lift-definition *open-cblinfun-sot* :: ‹('*a*, '*b*) *cblinfun-sot set* ⇒ *bool*› **is** ‹*openin cstrong-operator-topology*›
.
instance
proof *intro-classes*
show ‹*open* (*UNIV* :: ('*a*,'*b*) *cblinfun-sot set*)›
apply *transfer*
by (*metis cstrong-operator-topology-topspace openin-topspace*)
show ‹*open S* ⟹ *open T* ⟹ *open* (*S* ∩ *T*)› **for** *S T* :: ‹('*a*,'*b*) *cblinfun-sot set*›
apply *transfer* **by** *auto*
show ‹∀ *S*∈*K*. *open S* ⟹ *open* (⋃ *K*)› **for** *K* :: ‹('*a*,'*b*) *cblinfun-sot set set*›
apply *transfer* **by** *auto*
qed
end

lemma *transfer-nhds-cstrong-operator-topology*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*cr-cblinfun-sot* ===> *rel-filter cr-cblinfun-sot*) (*nhdsin cstrong-operator-topology*) *nhds*›
unfolding *nhds-def nhdsin-def*

apply (*simp add*: *cstrong-operator-topology-topspace*)
by *transfer-prover*

lemma *filterlim-cstrong-operator-topology*: ‹*filterlim f* (*nhdsin cstrong-operator-topology l*) = *limitin cstrong-operator-topology f l*›
by (*auto simp*: *cstrong-operator-topology-topspace simp flip*: *filterlim-nhdsin-iff-limitin*)

lemma *hausdorff-sot*[*simp*]: ‹*Hausdorff-space cstrong-operator-topology*›
proof (*rule hausdorffI*)
fix *a b* :: ‹*'a* ⇒$_{CL}$ *'b*›
assume ‹*a* ≠ *b*›
then obtain *ψ* **where** ‹*a* *$_V$ *ψ* ≠ *b* *$_V$ *ψ*›
by (*meson cblinfun-eqI*)
then obtain *U' V'* **where** ‹*open U'*› ‹*open V'*› ‹*a* *$_V$ *ψ* ∈ *U'*› ‹*b* *$_V$ *ψ* ∈ *V'*› ‹*U'* ∩ *V'* = {}›
by (*meson hausdorff*)
define *U V* **where** ‹*U* = {*f*. ∀*i*∈{()}. *f* *$_V$ *ψ* ∈ *U'*}› **and** ‹*V* = {*f*. ∀*i*∈{()}. *f* *$_V$ *ψ* ∈ *V'*}›
have *1*: ‹*openin cstrong-operator-topology U*›
unfolding *U-def* **apply** (*rule cstrong-operator-topology-basis*)
using ‹*open U'*› **by** *auto*
have *2*: ‹*openin cstrong-operator-topology V*›
unfolding *V-def* **apply** (*rule cstrong-operator-topology-basis*)
using ‹*open V'*› **by** *auto*
show ‹∃ *U V*. *openin cstrong-operator-topology U* ∧ *openin cstrong-operator-topology V* ∧ *a* ∈ *U* ∧ *b* ∈ *V* ∧ *U* ∩ *V* = {}›
by (*rule exI*[*of* - *U*], *rule exI*[*of* - *V*])
(*use 1 2* ‹*a* *$_V$ *ψ* ∈ *U'*› ‹*b* *$_V$ *ψ* ∈ *V'*› ‹*U'* ∩ *V'* = {}› **in** ‹*auto simp*: *U-def V-def*›)
qed

instance *cblinfun-sot* :: (*complex-normed-vector*, *complex-normed-vector*) *t2-space*
proof *intro-classes*
fix *a b* :: ‹('*a*,'*b*) *cblinfun-sot*›
show ‹*a* ≠ *b* ⟹ ∃ *U V*. *open U* ∧ *open V* ∧ *a* ∈ *U* ∧ *b* ∈ *V* ∧ *U* ∩ *V* = {}›
apply *transfer* **using** *hausdorff-sot*
by (*metis UNIV-I cstrong-operator-topology-topspace Hausdorff-space-def disjnt-def*)
qed

lemma *Domainp-cr-cblinfun-sot*[*simp*]: ‹*Domainp cr-cblinfun-sot* = (λ-. *True*)›
by (*metis* (*no-types*, *opaque-lifting*) *DomainPI cblinfun-sot.left-total left-totalE*)

lemma *Rangep-cr-cblinfun-sot*[*simp*]: ‹*Rangep cr-cblinfun-sot* = (λ-. *True*)›
by (*meson RangePI cr-cblinfun-sot-def*)

lemma *Rangep-set*[*relator-domain*]: *Rangep* (*rel-set T*) = (λ*A*. *Ball A* (*Rangep T*))
by (*metis* (*no-types*, *opaque-lifting*) *Domainp-conversep Domainp-set rel-set-conversep*)

lemma *transfer-euclidean-cstrong-operator-topology*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-topology cr-cblinfun-sot*) *cstrong-operator-topology euclidean*›
proof (*unfold rel-topology-def*, *intro conjI allI impI*)
show ‹(*rel-set cr-cblinfun-sot* ===> (=)) (*openin cstrong-operator-topology*) (*openin euclidean*)›
unfolding *rel-fun-def rel-set-def open-openin* [*symmetric*] *cr-cblinfun-sot-def*
by (*transfer*, *intro allI impI arg-cong*[*of - - openin x* **for** *x*]) *blast*
next
fix *U* :: ‹('*a* ⇒$_{CL}$ '*b*) *set*›
assume ‹*openin cstrong-operator-topology U*›
show ‹*Domainp* (*rel-set cr-cblinfun-sot*) *U*›
by (*simp add*: *Domainp-set*)
next
fix *U* :: ‹('*a*, '*b*) *cblinfun-sot set*›
assume ‹*openin euclidean U*›
show ‹*Rangep* (*rel-set cr-cblinfun-sot*) *U*›
by (*simp add*: *Rangep-set*)
qed

lemma *openin-cstrong-operator-topology*: ‹*openin cstrong-operator-topology U* ⟷ (∃ *V*. *open V* ∧ *U* = (*∗$_V$*) −‘ *V*)›
by (*simp add*: *cstrong-operator-topology-def openin-pullback-topology*)

lemma *cstrong-operator-topology-plus-cont*: ‹*LIM* (*x*,*y*) *nhdsin cstrong-operator-topology a* ×$_F$ *nhdsin cstrong-operator-topology b*.
x + *y* :> *nhdsin cstrong-operator-topology* (*a* + *b*)›
unfolding *cstrong-operator-topology-def*
by (*rule pullback-topology-bi-cont*[**where** *f'*=*plus*])
(*auto simp*: *case-prod-unfold tendsto-add-Pair cblinfun.add-left*)

instance *cblinfun-sot* :: (*complex-normed-vector*, *complex-normed-vector*) *topological-group-add*
proof *intro-classes*
show ‹((λ*x*. *fst x* + *snd x*) ⟶ *a* + *b*) (*nhds a* ×$_F$ *nhds b*)› **for** *a b* :: ‹('*a*,'*b*) *cblinfun-sot*›
apply *transfer*
using *cstrong-operator-topology-plus-cont*
by (*auto simp*: *case-prod-unfold*)

have ∗: ‹*continuous-map cstrong-operator-topology cstrong-operator-topology uminus*›
apply (*subst continuous-on-cstrong-operator-topo-iff-coordinatewise*)
apply (*rewrite at* ‹(λ*y*. − *y* ∗$_V$ *x*)› **in** ‹∀ *x*. ¤› *to* ‹(λ*y*. *y* ∗$_V$ − *x*)› *DEADID.rel-mono-strong*)
by (*auto simp*: *cstrong-operator-topology-continuous-evaluation cblinfun.minus-left cblinfun.minus-right*)
show ‹(*uminus* ⟶ − *a*) (*nhds a*)› **for** *a* :: ‹('*a*,'*b*) *cblinfun-sot*›
apply (*subst tendsto-at-iff-tendsto-nhds*[*symmetric*])
apply (*subst isCont-def*[*symmetric*])
apply (*rule continuous-on-interior*[**where** *S*=*UNIV*])
apply (*subst continuous-map-iff-continuous2*[*symmetric*])

```
    apply transfer
   using ∗ by auto
qed

lemma continuous-map-left-comp-sot[continuous-intros]:
  fixes b :: ‹'b::complex-normed-vector ⇒CL 'c::complex-normed-vector›
    and f :: ‹'a ⇒ 'd::complex-normed-vector ⇒CL 'b›
  assumes ‹continuous-map T cstrong-operator-topology f›
  shows ‹continuous-map T cstrong-operator-topology (λx. b oCL f x)›
proof −
  have ∗: ‹open B ⟹ open ((∗V) b −‘ B)› for B
    by (simp add: continuous-open-vimage)
  have ∗∗: ‹((λa. b ∗V a ψ) −‘ B ∩ UNIV) = (PiE UNIV (λi. if i=ψ then (λa. b ∗V a) −‘ B
else UNIV))›
    for ψ :: 'd and B
    by (auto simp: PiE-def Pi-def)
  have ∗: ‹continuous-on UNIV (λ(a::'d ⇒ 'b). b ∗V (a ψ))› for ψ
    unfolding continuous-on-open-vimage[OF open-UNIV]
    apply (intro allI impI)
    apply (subst ∗∗)
    apply (rule open-PiE)
    using ∗ by auto
  have ∗: ‹continuous-on UNIV (λ(a::'d ⇒ 'b) ψ. b ∗V a ψ)›
    apply (rule continuous-on-coordinatewise-then-product)
    by (rule ∗)
  have ‹continuous-map cstrong-operator-topology cstrong-operator-topology
            (λx :: 'd ⇒CL 'b. b oCL x)›
    unfolding cstrong-operator-topology-def
    apply (rule continuous-map-pullback')
    subgoal
      apply (subst asm-rl[of ‹(∗V) ∘ (oCL) b = (λa x. b ∗V (a x)) ∘ (∗V)›])
      subgoal by force
      subgoal by (rule continuous-map-pullback) (use ∗ in auto)
      done
    subgoal using ∗ by auto
    done
  from continuous-map-compose[OF assms this, unfolded o-def]
  show ?thesis
    by −
qed

lemma continuous-cstrong-operator-topology-plus[continuous-intros]:
  assumes ‹continuous-map T cstrong-operator-topology f›
  assumes ‹continuous-map T cstrong-operator-topology g›
  shows ‹continuous-map T cstrong-operator-topology (λx. f x + g x)›
  using assms
  by (auto intro!: continuous-map-add
      simp: continuous-on-cstrong-operator-topo-iff-coordinatewise cblinfun.add-left)
```

lemma *continuous-cstrong-operator-topology-uminus*[*continuous-intros*]:
assumes ‹*continuous-map T cstrong-operator-topology f*›
shows ‹*continuous-map T cstrong-operator-topology* (λ*x*. − *f x*)›
using *assms*
by (*auto simp add*: *continuous-on-cstrong-operator-topo-iff-coordinatewise cblinfun.minus-left*)

lemma *continuous-cstrong-operator-topology-minus*[*continuous-intros*]:
assumes ‹*continuous-map T cstrong-operator-topology f*›
assumes ‹*continuous-map T cstrong-operator-topology g*›
shows ‹*continuous-map T cstrong-operator-topology* (λ*x*. *f x* − *g x*)›
apply (*subst diff-conv-add-uminus*)
by (*intro continuous-intros assms*)

lemma *continuous-map-right-comp-sot*[*continuous-intros*]:
assumes ‹*continuous-map T cstrong-operator-topology f*›
shows ‹*continuous-map T cstrong-operator-topology* (λ*x*. *f x* o_{CL} *a*)›
apply (*rule continuous-map-compose*[*OF assms*, *unfolded o-def*])
by (*simp add*: *continuous-on-cstrong-operator-topo-iff-coordinatewise cstrong-operator-topology-continuous-evaluatio*

lemma *continuous-map-scaleC-sot*[*continuous-intros*]:
assumes ‹*continuous-map T cstrong-operator-topology f*›
shows ‹*continuous-map T cstrong-operator-topology* (λ*x*. *c* $*_C$ *f x*)›
apply (*subst asm-rl*[*of* ‹*scaleC c* = (o_{CL}) (*c* $*_C$ *id-cblinfun*)›])
apply *auto*[*1*]
using *assms* **by** (*rule continuous-map-left-comp-sot*)

lemma *continuous-scaleC-sot*[*continuous-intros*]:
fixes *f* :: ‹'*a*::*topological-space* ⇒ (-,-) *cblinfun-sot*›
assumes ‹*continuous-on X f*›
shows ‹*continuous-on X* (λ*x*. *c* $*_C$ *f x*)›
apply (*rule continuous-on-compose*[*OF assms*, *unfolded o-def*])
apply (*rule continuous-on-subset*[*rotated*, **where** *s*=*UNIV*], *simp*)
apply (*subst continuous-map-iff-continuous2*[*symmetric*])
apply *transfer*
apply (*rule continuous-map-scaleC-sot*)
by *simp*

lemma *sot-closure-is-csubspace*[*simp*]:
fixes *A*::('*a*::*complex-normed-vector*, '*b*::*complex-normed-vector*) *cblinfun-sot set*
assumes ‹*csubspace A*›
shows ‹*csubspace* (*closure A*)›
proof (*rule complex-vector.subspaceI*)
include *lattice-syntax*
show *0*: ‹*0* ∈ *closure A*›
by (*simp add*: *assms closure-def complex-vector.subspace-0*)
show ‹*x* + *y* ∈ *closure A*› **if** ‹*x* ∈ *closure A*› ‹*y* ∈ *closure A*› **for** *x y*
proof −

```
    define FF where ‹FF = ((nhds x ⊓ principal A) ×_F (nhds y ⊓ principal A))›
    have nt: ‹FF ≠ bot›
      by (simp add: prod-filter-eq-bot that(1) that(2) FF-def flip: closure-nhds-principal)
    have ‹∀_F x in FF. fst x ∈ A›
      unfolding FF-def
      by (smt (verit, ccfv-SIG) eventually-prod-filter fst-conv inf-sup-ord(2) le-principal)
    moreover have ‹∀_F x in FF. snd x ∈ A›
      unfolding FF-def
      by (smt (verit, ccfv-SIG) eventually-prod-filter snd-conv inf-sup-ord(2) le-principal)
    ultimately have FF-plus: ‹∀_F x in FF. fst x + snd x ∈ A›
      by (smt (verit, best) assms complex-vector.subspace-add eventually-elim2)

    have ‹(fst ⟶⟶ x) ((nhds x ⊓ principal A) ×_F (nhds y ⊓ principal A))›
      apply (simp add: filterlim-def)
      using filtermap-fst-prod-filter
      using le-inf-iff by blast
    moreover have ‹(snd ⟶⟶ y) ((nhds x ⊓ principal A) ×_F (nhds y ⊓ principal A))›
      apply (simp add: filterlim-def)
      using filtermap-snd-prod-filter
      using le-inf-iff by blast
    ultimately have ‹(id ⟶⟶ (x,y)) FF›
      by (simp add: filterlim-def nhds-prod prod-filter-mono FF-def)

    moreover note tendsto-add-Pair[of x y]
    ultimately have ‹(((λx. fst x + snd x) o id) ⟶⟶ (λx. fst x + snd x) (x,y)) FF›
      unfolding filterlim-def nhds-prod
        by (smt (verit, best) filterlim-compose filterlim-def filterlim-filtermap fst-conv snd-conv
tendsto-compose-filtermap)

    then have ‹((λx. fst x + snd x) ⟶⟶ (x+y)) FF›
      by simp
    then show ‹x + y ∈ closure A›
      using nt FF-plus by (rule limit-in-closure)
  qed
  show ‹c *_C x ∈ closure A› if ‹x ∈ closure A› for x c
  proof (cases c = 0)
    case False
    have (*_C) c ‘ A ⊆ closure A
      using csubspace-scaleC-invariant[of c A] assms False closure-subset[of A] by auto
    hence (*_C) c ‘ closure A ⊆ closure A
      by (intro image-closure-subset) (auto intro!: continuous-intros)
    thus ?thesis
      using that by blast
  qed (use 0 in auto)
qed

lemma limitin-cstrong-operator-topology:
  ‹limitin cstrong-operator-topology f l F ⟷ (∀ i. ((λj. f j *_V i) ⟶⟶ l *_V i) F)›
  by (simp add: cstrong-operator-topology-def limitin-pullback-topology
```

tendsto-coordinatewise)

lemma *cstrong-operator-topology-in-closureI*:
assumes ‹$\bigwedge M\ \varepsilon.\ \varepsilon > 0 \Longrightarrow$ *finite* $M \Longrightarrow \exists\, a{\in}A.\ \forall\, v{\in}M.$ *norm* $((b-a) *_V v) \leq \varepsilon$›
shows ‹$b \in$ *cstrong-operator-topology closure-of* A›
proof −
define $F ::$ ‹('*a set* × *real*) *filter*› **where** ‹$F =$ *finite-subsets-at-top UNIV* $\times_F$ *at-right* 0›
obtain f **where** *fA*: ‹$f\ M\ \varepsilon \in A$› **and** f: ‹$v \in M \Longrightarrow$ *norm* $((f\ M\ \varepsilon - b) *_V v) \leq \varepsilon$› **if** ‹*finite* M› **and** ‹$\varepsilon > 0$› **for** $M\ \varepsilon\ v$
apply *atomize-elim*
apply (*intro allI choice2*)
using *assms*
by (*metis cblinfun.diff-left norm-minus-commute*)
have *F-props*: ‹$\forall_F\ (M,\varepsilon)$ *in* F. *finite* $M \wedge \varepsilon > 0$›
by (*auto intro*!: *eventually-prodI simp*: *F-def case-prod-unfold eventually-at-right-less*)
then have *inA*: ‹$\forall_F\ (M,\varepsilon)$ *in* F. $f\ M\ \varepsilon \in A$›
apply (*rule eventually-rev-mp*)
using *fA* **by** (*auto intro*!: *always-eventually*)
have ‹*limitin cstrong-operator-topology* (*case-prod* f) b F›
proof −
have ‹$\forall_F\ (M,\varepsilon)$ *in* F. *norm* $(f\ M\ \varepsilon *_V v - b *_V v) < e$› **if** ‹$e > 0$› **for** $e\ v$
proof −
have *1*: ‹$\forall_F\ (M,\varepsilon)$ *in* F. (*finite* $M \wedge v \in M$) $\wedge$ ($\varepsilon > 0 \wedge \varepsilon < e$)›
apply (*unfold F-def case-prod-unfold*, *rule eventually-prodI*)
using *eventually-at-right that*
by (*auto simp add*: *eventually-finite-subsets-at-top*)
have *2*: ‹*norm* $(f\ M\ \varepsilon *_V v - b *_V v) < e$› **if** ‹(*finite* $M \wedge v \in M$) $\wedge$ ($\varepsilon > 0 \wedge \varepsilon < e$)› **for** $M\ \varepsilon$
by (*smt* (*verit*) *cblinfun.diff-left f that*)
show *?thesis*
using *1* **apply** (*rule eventually-mono*)
using *2* **by** *auto*
qed
then have ‹$((\lambda(M,\varepsilon).\ f\ M\ \varepsilon *_V v) \longrightarrow b *_V v)\ F$› **for** v
by (*simp add*: *tendsto-iff dist-norm case-prod-unfold*)
then show *?thesis*
by (*simp add*: *case-prod-unfold limitin-cstrong-operator-topology*)
qed
then show *?thesis*
apply (*rule limitin-closure-of*)
using *inA* **by** (*auto simp*: *F-def case-prod-unfold prod-filter-eq-bot*)
qed

lemma *sot-weaker-than-norm-limitin*: ‹*limitin cstrong-operator-topology a A F*› **if** ‹($a \longrightarrow A$) F›
proof −

from *that* **have** ‹((λx. *a x* ∗$_V$ ψ) ⟶ *A* ψ) *F*› **for** ψ
by (*auto intro*!: *cblinfun.tendsto*)
then show *?thesis*
by (*simp add*: *limitin-cstrong-operator-topology*)
qed

lemma [*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-cblinfun-sot* ===> (=)) *csubspace csubspace*›
unfolding *complex-vector.subspace-def*
by *transfer-prover*

lemma [*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-cblinfun-sot* ===> (=)) (*closedin cstrong-operator-topology*) *closed*›
apply (*simp add*: *closed-def*[*abs-def*] *closedin-def*[*abs-def*] *cstrong-operator-topology-topspace Compl-eq-Diff-UNIV*)
by *transfer-prover*

lemma [*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-cblinfun-sot* ===> *rel-set cr-cblinfun-sot*) (*Abstract-Topology.closure-of cstrong-operator-topology*) *closure*›
apply (*subst closure-of-hull*[**where** *X*=*cstrong-operator-topology*, *unfolded cstrong-operator-topology-topspace*, *simplified*, *abs-def*])
apply (*subst closure-hull*[*abs-def*])
unfolding *hull-def*
by *transfer-prover*

lemma *sot-closure-is-csubspace'*[*simp*]:
fixes *A*::('*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector*) *set*
assumes ‹*csubspace A*›
shows ‹*csubspace* (*cstrong-operator-topology closure-of A*)›
using *sot-closure-is-csubspace*[*of* ‹*Abs-cblinfun-sot* ‘ *A*›] *assms*
apply (*transfer fixing*: *A*)
by *simp*

lemma *has-sum-closed-cstrong-operator-topology*:
assumes *aA*: ‹⋀*i*. *a i* ∈ *A*›
assumes *closed*: ‹*closedin cstrong-operator-topology A*›
assumes *subspace*: ‹*csubspace A*›
assumes *has-sum*: ‹⋀ψ. ((λ*i*. *a i* ∗$_V$ ψ) *has-sum* (*b* ∗$_V$ ψ)) *I*›
shows ‹*b* ∈ *A*›
proof −
have *1*: ‹*range* (*sum a*) ⊆ *A*›
proof −
have ‹*sum a X* ∈ *A*› **for** *X*
apply (*induction X rule*:*infinite-finite-induct*)
by (*auto simp add*: *subspace complex-vector.subspace-0 aA complex-vector.subspace-add*)

```
    then show ?thesis
      by auto
  qed

  from has-sum
  have ‹((λF. ∑ i∈F. a i *V ψ) ⟶ b *V ψ) (finite-subsets-at-top I)› for ψ
    using has-sum-def by blast
  then have ‹limitin cstrong-operator-topology (λF. ∑ i∈F. a i) b (finite-subsets-at-top I)›
    by (auto simp add: limitin-cstrong-operator-topology cblinfun.sum-left)
  then show ‹b ∈ A›
    using 1 closed apply (rule limitin-closedin)
    by simp
qed


lemma has-sum-in-cstrong-operator-topology:
  ‹has-sum-in cstrong-operator-topology f A l ⟷ (∀ψ. ((λi. f i *V ψ) has-sum (l *V ψ)) A)›
  by (simp add: cblinfun.sum-left has-sum-in-def limitin-cstrong-operator-topology has-sum-def)

lemma summable-sot-absI:
  fixes b :: ‹'a ⇒ 'b::complex-normed-vector ⇒CL 'c::chilbert-space›
  assumes ‹⋀F f. finite F ⟹ (∑ n∈F. norm (b n *V f)) ≤ K * norm f›
  shows ‹summable-on-in cstrong-operator-topology b UNIV›
proof −
  obtain B' where B': ‹((λn. b n *V f) has-sum (B' f)) UNIV› for f
  proof (atomize-elim, intro choice allI)
    fix f
    have ‹(λn. b n *V f) abs-summable-on UNIV›
      apply (rule nonneg-bdd-above-summable-on)
      using assms by (auto intro!: bdd-aboveI[where M=‹K * norm f›])
    then show ‹∃l. ((λn. b n *V f) has-sum l) UNIV›
      by (metis abs-summable-summable summable-on-def)
  qed
  have ‹bounded-clinear B'›
  proof (intro bounded-clinearI allI)
    fix x y :: 'b and c :: complex
    from B'[of x] B'[of y]
    have ‹((λn. b n *V x + b n *V y) has-sum B' x + B' y) UNIV›
      by (simp add: has-sum-add)
    with B'[of ‹x + y›]
    show ‹B' (x + y) = B' x + B' y›
      by (metis (no-types, lifting) cblinfun.add-right has-sum-cong infsumI)
    from B'[of x]
    have ‹((λn. c *C (b n *V x)) has-sum c *C B' x) UNIV›
      by (metis cblinfun-scaleC-right.rep-eq has-sum-cblinfun-apply)
    with B'[of ‹c *C x›]
    show ‹B' (c *C x) = c *C B' x›
      by (metis (no-types, lifting) cblinfun.scaleC-right has-sum-cong infsumI)
    show ‹norm (B' x) ≤ norm x * K›
```

```
    proof −
      have ∗: ‹(λn. b n ∗_V x) abs-summable-on UNIV›
        apply (rule nonneg-bdd-above-summable-on)
        using assms by (auto intro!: bdd-aboveI[where M=‹K ∗ norm x›])
      have ‹norm (B' x) ≤ (∑∞n. norm (b n ∗_V x))›
        using - B'[of x] apply (rule norm-has-sum-bound)
        using ∗ summable-iff-has-sum-infsum by blast
      also have ‹(∑∞n. norm (b n ∗_V x)) ≤ K ∗ norm x›
        using ∗ apply (rule infsum-le-finite-sums)
        using assms by simp
      finally show ?thesis
        by (simp add: mult.commute)
    qed
  qed
  define B where ‹B = CBlinfun B'›
  with ‹bounded-clinear B'› have BB': ‹B ∗_V f = B' f› for f
    by (simp add: bounded-clinear-CBlinfun-apply)
  have ‹has-sum-in cstrong-operator-topology b UNIV B›
    using B' by (simp add: has-sum-in-cstrong-operator-topology BB')
  then show ?thesis
    using summable-on-in-def by blast
qed

declare cstrong-operator-topology-topspace[simp]

lift-definition cblinfun-compose-sot :: ‹('a::complex-normed-vector,'b::complex-normed-vector)
cblinfun-sot ⇒ ('c::complex-normed-vector,'a) cblinfun-sot ⇒ ('c,'b) cblinfun-sot›
  is cblinfun-compose .

lemma isCont-cblinfun-compose-sot-right[simp]: ‹isCont (λF. cblinfun-compose-sot F G) x›
  apply (rule continuous-on-interior[where S=UNIV, rotated], simp)
  apply (rule continuous-map-iff-continuous2[THEN iffD1])
  apply transfer
  by (simp add: continuous-map-right-comp-sot)

lemma isCont-cblinfun-compose-sot-left[simp]: ‹isCont (λF. cblinfun-compose-sot G F) x›
  apply (rule continuous-on-interior[where S=UNIV, rotated], simp)
  apply (rule continuous-map-iff-continuous2[THEN iffD1])
  apply transfer
  by (simp add: continuous-map-left-comp-sot)

lemma additive-cblinfun-compose-sot-right[simp]: ‹additive (λF. cblinfun-compose-sot F G)›
  unfolding additive-def
  apply transfer
  by (simp add: cblinfun-compose-add-left)

lemma additive-cblinfun-compose-sot-left[simp]: ‹additive (λF. cblinfun-compose-sot G F)›
  unfolding additive-def
  apply transfer
```

by (*simp add*: *cblinfun-compose-add-right*)

lemma *transfer-infsum-sot*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique R*›
shows ‹((*R* ===> *cr-cblinfun-sot*) ===> *rel-set R* ===> *cr-cblinfun-sot*) (*infsum-in cstrong-operator-topology*) *infsum*›
apply (*simp add*: *infsum-euclidean-eq*[*abs-def*, *symmetric*])
by *transfer-prover*

lemma *transfer-summable-on-sot*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique R*›
shows ‹((*R* ===> *cr-cblinfun-sot*) ===> *rel-set R* ===> (⟷)) (*summable-on-in cstrong-operator-topology*) (*summable-on*)›
apply (*simp add*: *summable-on-euclidean-eq*[*abs-def*, *symmetric*])
by *transfer-prover*

lemma *sandwich-sot-cont*[*continuous-intros*]:
assumes ‹*continuous-map T cstrong-operator-topology f*›
shows ‹*continuous-map T cstrong-operator-topology* (λ*x*. *sandwich A* (*f x*))›
apply (*simp add*: *sandwich-apply*)
by (*intro continuous-intros assms*)

lemma *closed-map-sot-unitary-sandwich*:
fixes *U* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
assumes ‹*unitary U*›
shows ‹*closed-map cstrong-operator-topology cstrong-operator-topology* (λ*x*. *sandwich U x*)›
apply (*rule closed-eq-continuous-inverse-map*[**where** *g*=‹*sandwich* (*U*∗)›, *THEN iffD2*])
using *assms*
by (*auto intro*!: *continuous-intros*
simp flip: *sandwich-compose cblinfun-apply-cblinfun-compose*)

unbundle *no-cblinfun-notation*

end

4 *Positive-Operators* – Positive bounded operators

theory *Positive-Operators*
imports
Ordinary-Differential-Equations.Cones
Misc-Tensor-Product-BO
Strong-Operator-Topology
begin

no-notation *Infinite-Set-Sum.abs-summable-on* (**infix** *abs'-summable'-on 50*)
hide-const (**open**) *Infinite-Set-Sum.abs-summable-on*
hide-fact (**open**) *Infinite-Set-Sum.abs-summable-on-Sigma-iff*

unbundle *cblinfun-notation*

lemma *cinner-pos-if-pos*: ‹f $\cdot_C$ (A $*_V$ f) $\geq$ 0› **if** ‹$A \geq 0$›
using *less-eq-cblinfun-def that* **by** *force*

definition *sqrt-op* :: ‹('*a*::*chilbert-space* $\Rightarrow_{CL}$ '*a*) $\Rightarrow$ ('*a* $\Rightarrow_{CL}$ '*a*)› **where**
‹*sqrt-op a* = (*if* ($\exists$ *b* :: '*a* $\Rightarrow_{CL}$ '*a*. *b* $\geq$ *0* $\wedge$ *b** o_{CL} *b* = *a*) *then* (*SOME b*. *b* $\geq$ *0* $\wedge$ *b** o_{CL} *b* = *a*) *else 0*)›

lemma *sqrt-op-nonpos*: ‹*sqrt-op a* = *0*› **if** ‹$\neg$ *a* $\geq$ *0*›
proof −
have ‹$\neg$ ($\exists$ *b*. *b* $\geq$ *0* $\wedge$ *b** o_{CL} *b* = *a*)›
using *positive-cblinfun-squareI that* **by** *blast*
then show *?thesis*
by (*auto simp add*: *sqrt-op-def*)
qed

lemma *generalized-Cauchy-Schwarz*:
fixes *inner A*
assumes *Apos*: ‹$A \geq 0$›
defines *inner x y* $\equiv$ x $\cdot_C$ (A $*_V$ y)
shows ‹*complex-of-real* ((*norm* (*inner x y*))^2) $\leq$ *inner x x* $*$ *inner y y*›
proof (*cases* ‹*inner y y* = *0*›)
case *True*
have [*simp*]: ‹*inner* (s $*_C$ x) y = *cnj s* $*$ *inner x y*› **for** s x y
by (*simp add*: *assms*(*2*))
have [*simp*]: ‹*inner x* (s $*_C$ y) = s $*$ *inner x y*› **for** s x y
by (*simp add*: *assms*(*2*) *cblinfun.scaleC-right*)
have [*simp*]: ‹*inner* (x − x') y = *inner x y* − *inner x'* *y*› **for** x x' y
by (*simp add*: *cinner-diff-left inner-def*)
have [*simp*]: ‹*inner x* (y − y') = *inner x y* − *inner x y'*› **for** x y y'
by (*simp add*: *cblinfun.diff-right cinner-diff-right inner-def*)
have *Re0*: ‹*Re* (*inner x y*) = *0*› **for** x
proof −
have $*$: ‹*Re* (*inner x y*) = (*inner x y* + *inner y x*) / *2*›
by (*smt* (*verit*, *del-insts*) *assms*(*1*) *assms*(*2*) *cinner-adj-left cinner-commute complex-Re-numeral complex-add-cnj field-sum-of-halves numeral-One numeral-plus-numeral of-real-divide of-real-numeral one-complex.simps*(*1*) *positive-hermitianI semiring-norm*(*2*))
have ‹*0* $\leq$ *Re* (*inner* (x − s $*_C$ y) (x − s $*_C$ y))› **for** s
by (*metis Re-mono assms*(*1*) *assms*(*2*) *cinner-pos-if-pos zero-complex.simps*(*1*))
also have ‹... s = *Re* (*inner x x*) − s $*$ *2* $*$ *Re* (*inner x y*)› **for** s
apply (*auto simp*: *True*)
by (*smt* (*verit*, *ccfv-threshold*) *Re-complex-of-real assms*(*1*) *assms*(*2*) *cinner-adj-right cinner-commute complex-add-cnj diff-minus-eq-add minus-complex.simps*(*1*) *positive-hermitianI uminus-complex.sel*(*1*))

finally show ‹*Re* (*inner x y*) = *0*›
by (*metis add-le-same-cancel1 ge-iff-diff-ge-0 nonzero-eq-divide-eq not-numeral-le-zero zero-neq-numeral*)
qed
have ‹*Im* (*inner x y*) = *Re* (*inner* (*imaginary-unit* $*_C$ *x*) *y*)›
by *simp*
also have ‹... = *0*›
by (*rule Re0*)
finally have ‹*inner x y* = *0*›
using *Re0*[*of x*]
using *complex-eq-iff zero-complex.simps*(*1*) *zero-complex.simps*(*2*) **by** *presburger*
then show *?thesis*
by (*auto simp*: *True*)
next
case *False*
have *inner-commute*: ‹*inner x y* = *cnj* (*inner y x*)›
by (*metis Apos cinner-adj-left cinner-commute′ inner-def positive-hermitianI*)
have [*simp*]: *cnj* (*inner y y*) = *inner y y* **for** *y*
by (*metis assms*(*1*) *cinner-adj-right cinner-commute′ inner-def positive-hermitianI*)
define *r* **where** *r* = *cnj* (*inner x y*) / *inner y y*
have *0* ≤ *inner* (*x* − *scaleC r y*) (*x* − *scaleC r y*)
by (*simp add*: *Apos inner-def cinner-pos-if-pos*)
also have ... = *inner x x* − *r* ∗ *inner x y* − *cnj r* ∗ *inner y x* + *r* ∗ *cnj r* ∗ *inner y y*
unfolding *cinner-diff-left cinner-diff-right cinner-scaleC-left cinner-scaleC-right inner-def*
by (*smt* (*verit*, *ccfv-threshold*) *cblinfun.diff-right cblinfun.scaleC-right cblinfun-cinner-right.rep-eq cinner-scaleC-left cinner-scaleC-right diff-add-eq diff-diff-eq2 mult.assoc*)
also have ... = *inner x x* − *inner y x* ∗ *cnj r*
unfolding *r-def* **by** *auto*
also have ... = *inner x x* − *inner x y* ∗ *cnj* (*inner x y*) / *inner y y*
unfolding *r-def*
by (*metis assms*(*1*) *assms*(*2*) *cinner-adj-right cinner-commute complex-cnj-divide mult.commute positive-hermitianI times-divide-eq-left*)
finally have *0* ≤ *inner x x* − *inner x y* ∗ *cnj* (*inner x y*) / *inner y y* **.**
hence *inner x y* ∗ *cnj* (*inner x y*) / *inner y y* ≤ *inner x x*
by (*simp add*: *le-diff-eq*)
hence ‹(*norm* (*inner x y*)) ^ *2* / *inner y y* ≤ *inner x x*›
using *complex-norm-square* **by** *presburger*
then show *?thesis*
by (*metis False assms*(*1*) *assms*(*2*) *cinner-pos-if-pos mult-right-mono nonzero-eq-divide-eq*)
qed

lemma *sandwich-pos*[*intro*]: ‹*sandwich b a* ≥ *0*› **if** ‹*a* ≥ *0*›
by (*metis* (*no-types*, *opaque-lifting*) *positive-cblinfunI cblinfun-apply-cblinfun-compose cinner-adj-left cinner-pos-if-pos sandwich-apply that*)

lemma *cblinfun-power-pos*: ‹*cblinfun-power a n* ≥ *0*› **if** ‹*a* ≥ *0*›
proof (*cases* ‹*even n*›)
case *True*
have ‹*0* ≤ (*cblinfun-power a* (*n div 2*))∗ o_{CL} (*cblinfun-power a* (*n div 2*))›

using *positive-cblinfun-squareI* **by** *blast*
also have ‹... = *cblinfun-power a* (*n div 2* + *n div 2*)›
by (*metis cblinfun-power-adj cblinfun-power-compose positive-hermitianI that*)
also from *True* **have** ‹... = *cblinfun-power a n*›
by (*metis add-self-div-2 div-plus-div-distrib-dvd-right*)
finally show *?thesis*
by −
next
case *False*
have ‹*0* ≤ *sandwich* (*cblinfun-power a* (*n div 2*)) *a*›
using ‹*a* ≥ *0*› **by** (*rule sandwich-pos*)
also have ‹... = *cblinfun-power a* (*n div 2* + *1* + *n div 2*)›
unfolding *sandwich-apply*
by (*metis* (*no-types*, *lifting*) *One-nat-def cblinfun-compose-id-right cblinfun-power-0 cblinfun-power-Suc′ cblinfun-power-adj cblinfun-power-compose positive-hermitianI that*)
also from *False* **have** ‹... = *cblinfun-power a n*›
by (*smt* (*verit*, *del-insts*) *Suc-1 add.commute add.left-commute add-mult-distrib2 add-self-div-2 nat.simps*(*3*) *nonzero-mult-div-cancel-left odd-two-times-div-two-succ*)
finally show *?thesis*
by −
qed

lemma *sqrt-op-existence*:
fixes *A* :: ‹*'a*::*chilbert-space* ⇒$_{CL}$ *'a*::*chilbert-space*›
assumes *Apos*: ‹*A* ≥ *0*›
shows ‹∃ *B*. *B* ≥ *0* ∧ *B* *o*$_{CL}$ *B* = *A* ∧ (∀ *F*. *A* *o*$_{CL}$ *F* = *F* *o*$_{CL}$ *A* ⟶ *B* *o*$_{CL}$ *F* = *F* *o*$_{CL}$ *B*)
∧ *B* ∈ *closure* (*cspan* (*range* (*cblinfun-power A*)))›
proof −
define *k S* **where** ‹*k* = *norm A*› **and** ‹*S* = *A* /$_R$ *k* − *id-cblinfun*›
have ‹*S* ≤ *0*›
proof (*rule cblinfun-leI*)
fix *x* :: *'a* **assume** [*simp*]: ‹*norm x* = *1*›
with *assms* **have** *aux1*: ‹*complex-of-real* (*inverse* (*norm A*)) ∗ (*x* ∙$_C$ (*A* ∗$_V$ *x*)) ≤ *1*›
by (*smt* (*verit*, *del-insts*) *Reals-cnj-iff cinner-adj-left cinner-commute cinner-scaleR-left cinner-scaleR-right cmod-Re complex-inner-class.Cauchy-Schwarz-ineq2 left-inverse less-eq-complex-def linordered-field-class.inverse-nonnegative-iff-nonnegative mult-cancel-left2 mult-left-mono norm-cblinfun norm-ge-zero norm-mult norm-of-real norm-one positive-hermitianI reals-zero-comparable zero-less-one-class.zero-le-o*
show ‹*x* ∙$_C$ (*S* ∗$_V$ *x*) ≤ *x* ∙$_C$ (*0* ∗$_V$ *x*)›
by (*auto simp*: *S-def cinner-diff-right cblinfun.diff-left scaleR-scaleC cdot-square-norm k-def complex-of-real-mono-iff*[**where** *y*=*1*, *simplified*]
simp flip: *assms of-real-inverse of-real-power of-real-mult power-mult-distrib power-inverse intro*!: *power-le-one aux1*)
qed
have [*simp*]: ‹*S*∗ = *S*›
using ‹*S* ≤ *0*› *adj-0 comparable-hermitean′ selfadjoint-def* **by** *blast*
have ‹− *id-cblinfun* ≤ *S*›

by (*simp add*: *S-def assms k-def scaleR-nonneg-nonneg*)
then have ‹*norm* (*S* $*_V$ *f*) ≤ *norm f*› **for** *f*
proof −
have *1*: ‹− *S* ≥ *0*›
by (*simp add*: ‹*S* ≤ *0*›)
have *2*: ‹*f* $\cdot_C$ (− *S* $*_V$ *f*) ≤ *f* $\cdot_C$ *f*›
by (*metis* ‹− *id-cblinfun* ≤ *S*› *id-cblinfun-apply less-eq-cblinfun-def minus-le-iff*)

have ‹(*norm* (*S* $*_V$ *f*))^4 = *complex-of-real* ((*cmod* ((− *S* $*_V$ *f*) $\cdot_C$ (− *S* $*_V$ *f*)))^2)›
apply (*auto simp*: *power4-eq-xxxx cblinfun.minus-left complex-of-real-cmod power2-eq-square simp flip*: *power2-norm-eq-cinner*)
by (*smt* (*verit*, *ccfv-SIG*) *complex-of-real-cmod mult.assoc norm-ge-zero norm-mult norm-of-real of-real-mult*)
also have ‹... ≤ (− *S* $*_V$ *f*) $\cdot_C$ (− *S* $*_V$ − *S* $*_V$ *f*) * (*f* $\cdot_C$ (− *S* $*_V$ *f*))›
apply (*rule generalized-Cauchy-Schwarz*[**where** *A*=‹−*S*› **and** *x* = ‹−*S* $*_V$ *f*› **and** *y* = *f*])
by (*fact 1*)
also have ‹... ≤ (− *S* $*_V$ *f*) $\cdot_C$ (− *S* $*_V$ − *S* $*_V$ *f*) * (*f* $\cdot_C$ *f*)›
using *2* **apply** (*rule mult-left-mono*)
using *1 cinner-pos-if-pos* **by** *blast*
also have ‹... ≤ (− *S* $*_V$ *f*) $\cdot_C$ (− *S* $*_V$ *f*) * (*f* $\cdot_C$ *f*)›
apply (*rule mult-right-mono*)
apply (*metis* ‹− *id-cblinfun* ≤ *S*› *id-cblinfun-apply less-eq-cblinfun-def neg-le-iff-le verit-minus-simplify*(*4*))
by *simp*
also have ‹... = (*norm* (−*S* $*_V$ *f*))^2 * (*norm f*)^2›
by (*simp add*: *cdot-square-norm*)
also have ‹... = (*norm* (*S* $*_V$ *f*))^2 * (*norm f*)^2›
by (*simp add*: *cblinfun.minus-left*)
finally have ‹*norm* (*S* $*_V$ *f*) ^ 4 ≤ (*norm* (*S* $*_V$ *f*))^2 * (*norm f*)^2›
using *complex-of-real-mono-iff* **by** *blast*
then have ‹(*norm* (*S* $*_V$ *f*))^2 ≤ (*norm f*)^2›
by (*smt* (*verit*, *best*) ‹*complex-of-real* (*norm* (*S* $*_V$ *f*) ^ 4) = *complex-of-real* ((*cmod* ((− *S* $*_V$ *f*) $\cdot_C$ (− *S* $*_V$ *f*)))^2)› *cblinfun.real.minus-left cinner-ge-zero cmod-Re mult-cancel-left mult-left-mono norm-minus-cancel of-real-eq-iff power2-eq-square power2-norm-eq-cinner' zero-less-norm-iff*)
then show ‹*norm* (*S* $*_V$ *f*) ≤ *norm f*›
by *auto*
qed
then have *norm-Snf*: ‹*norm* (*cblinfun-power S n* $*_V$ *f*) ≤ *norm f*› **for** *f n*
by (*induction n*, *auto simp*: *cblinfun-power-Suc' intro*: *order.trans*)
have *fSnf*: ‹*cmod* (*f* $\cdot_C$ (*cblinfun-power S n* $*_V$ *f*)) ≤ *cmod* (*f* $\cdot_C$ *f*)› **for** *f n*
by (*smt* (*z3*) *One-nat-def Re-complex-of-real Suc-1 cdot-square-norm cinner-ge-zero cmod-Re complex-inner-class.Cauchy-Schwarz-ineq2 mult.commute mult-cancel-right1 mult-left-mono norm-Snf norm-ge-zero power-0 power-Suc*)
from *norm-Snf* **have** *norm-Sn*: ‹*norm* (*cblinfun-power S n*) ≤ *1*› **for** *n*
apply (*rule-tac norm-cblinfun-bound*)
by *auto*
define *b* **where** ‹*b* = (λ*n*. (*1* / *2 gchoose n*) $*_R$ *cblinfun-power S n*)›
define *B0 B* **where** ‹*B0* = *infsum b UNIV*› **and** ‹*B* = *sqrt k* $*_R$ *B0*›

have *sum-norm-b*: ‹(∑ *n*∈*F*. *norm* (*b n*)) ≤ *3*› (**is** ‹*?lhs* ≤ *?rhs*›) **if** ‹*finite F*› **for** *F*

```
  proof −
    have [simp]: ‹⌈1 / 2 :: real⌉ = 1›
      by (simp add: ceiling-eq-iff)
    from ‹finite F› obtain d where ‹F ⊆ {..d}› and [simp]: ‹d > 0›
      by (metis Icc-subset-Iic-iff atLeast0AtMost bot-nat-0.extremum bot-nat-0.not-eq-extremum
dual-order.trans finite-nat-iff-bounded-le less-one)

    have ‹?lhs = (∑ n∈F. norm ((1 / 2 gchoose n) *R (cblinfun-power S n)))›
      by (simp add: b-def scaleR-cblinfun.rep-eq)
    also have ‹... ≤ (∑ n∈F. abs ((1 / 2 gchoose n)))›
      apply (auto intro!: sum-mono)
      using norm-Sn
      by (metis norm-cmul-rule-thm norm-scaleR verit-prod-simplify(2))
    also have ‹... ≤ (∑ n≤d. abs (1/2 gchoose n))›
      using ‹F ⊆ {..d}› by (auto intro!: mult-right-mono sum-mono2)
    also have ‹... = (2 − (− 1) ^ d ∗ (− (1 / 2) gchoose d))›
      apply (subst gbinomial-sum-lower-abs)
      by auto
    also have ‹... ≤ (2 + norm (− (1/2) gchoose d :: real))›
      apply (auto intro!: mult-right-mono)
     by (smt (verit) left-minus-one-mult-self mult.assoc mult-minus-left power2-eq-iff power2-eq-square)
    also have ‹... ≤ 3›
      apply (subgoal-tac ‹abs (− (1/2) gchoose d :: real) ≤ 1›)
     apply (metis add-le-cancel-left is-num-normalize(1) mult.commute mult-left-mono norm-ge-zero
numeral-Bit0 numeral-Bit1 one-add-one real-norm-def)
      apply (rule abs-gbinomial-leq1)
      by auto
    finally show ?thesis
      by −
  qed

  have has-sum-b: ‹(b has-sum B0) UNIV›
    apply (auto intro!: has-sum-infsum abs-summable-summable[where f=b] bdd-aboveI[where
M=3] simp: B0-def abs-summable-iff-bdd-above)
    using sum-norm-b
    by simp

  have ‹B0 ≥ 0›
  proof (rule positive-cblinfunI)
    fix f :: 'a assume [simp]: ‹norm f = 1›
    from has-sum-b
    have sum1: ‹(λn. f •C (b n *V f)) summable-on UNIV›
      apply (intro summable-on-cinner-left summable-on-cblinfun-apply-left)
      by (simp add: has-sum-imp-summable)
    have sum2: ‹(λx. − (complex-of-real |1 / 2 gchoose x| ∗ (f •C f))) summable-on UNIV −
{0}›
      apply (rule abs-summable-summable)
      using gbinomial-abs-summable-1[of ‹1/2›]
      by (auto simp add: cnorm-eq-1[THEN iffD1])
```

from *sum1* **have** *sum3*: ‹(λn. *complex-of-real* (*1 / 2 gchoose n*) ∗ (*f* ∙$_C$ (*cblinfun-power S n* ∗$_V$ *f*))) *summable-on UNIV* − {*0*}›
unfolding *b-def*
by (*metis* (*no-types*, *lifting*) *cinner-scaleR-right finite.emptyI finite-insert scaleR-cblinfun.rep-eq summable-on-cofin-subset summable-on-cong*)

have *aux*: ‹*a* ≥ − *b*› **if** ‹*norm a* ≤ *norm b*› **and** ‹*a* ∈ ℝ› **and** ‹*b* ≥ *0*› **for** *a b* :: *complex*
using *cmod-eq-Re complex-is-Real-iff less-eq-complex-def that*(*1*) *that*(*2*) *that*(*3*) **by** *force*

from *has-sum-b*
have ‹*f* ∙$_C$ (*B0* ∗$_V$ *f*) = (∑$_\infty$*n*. *f* ∙$_C$ (*b n* ∗$_V$ *f*))›
by (*metis B0-def infsum-cblinfun-apply-left infsum-cinner-left summable-on-cblinfun-apply-left summable-on-def*)
moreover have ‹... = (∑$_\infty$*n*∈*UNIV*−{*0*}. *f* ∙$_C$ (*b n* ∗$_V$ *f*)) + *f* ∙$_C$ (*b 0* ∗$_V$ *f*)›
apply (*subst infsum-Diff*)
using *sum1* **by** *auto*
moreover have ‹... = *f* ∙$_C$ (*b 0* ∗$_V$ *f*) + (∑$_\infty$*n*∈*UNIV*−{*0*}. *f* ∙$_C$ ((*1/2 gchoose n*) ∗$_R$ *cblinfun-power S n* ∗$_V$ *f*))›
unfolding *b-def* **by** *simp*
moreover have ‹... = *f* ∙$_C$ (*b 0* ∗$_V$ *f*) + (∑$_\infty$*n*∈*UNIV*−{*0*}. *of-real* (*1/2 gchoose n*) ∗ (*f* ∙$_C$ (*cblinfun-power S n* ∗$_V$ *f*)))›
by (*simp add*: *scaleR-cblinfun.rep-eq*)
moreover have ‹... ≥ *f* ∙$_C$ (*b 0* ∗$_V$ *f*) − (∑$_\infty$*n*∈*UNIV*−{*0*}. *of-real* (*abs* (*1/2 gchoose n*)) ∗ (*f* ∙$_C$ *f*))› (**is** ‹- ≥ ...›)
proof −
have ∗: ‹− (*complex-of-real* (*abs* (*1 / 2 gchoose x*)) ∗ (*f* ∙$_C$ *f*))
≤ *complex-of-real* (*1 / 2 gchoose x*) ∗ (*f* ∙$_C$ (*cblinfun-power S x* ∗$_V$ *f*))› **for** *x*
apply (*rule aux*)
by (*auto simp*: *cblinfun-power-adj norm-mult fSnf selfadjoint-def*
intro!: *cinner-real cinner-hermitian-real mult-left-mono Reals-mult mult-nonneg-nonneg*)
show *?thesis*
apply (*subst diff-conv-add-uminus*) **apply** (*rule add-left-mono*)
apply (*subst infsum-uminus*[*symmetric*]) **apply** (*rule infsum-mono-complex*)
apply (*rule sum2*)
apply (*rule sum3*)
by (*rule* ∗)
qed
moreover have ‹... = *f* ∙$_C$ (*b 0* ∗$_V$ *f*) − (∑$_\infty$*n*∈*UNIV*−{*0*}. *of-real* (*abs* (*1/2 gchoose n*))) ∗ (*f* ∙$_C$ *f*)›
by (*simp add*: *infsum-cmult-left′*)
moreover have ‹... = *of-real* (*1* − (∑$_\infty$*n*∈*UNIV*−{*0*}. (*abs* (*1/2 gchoose n*)))) ∗ (*f* ∙$_C$ *f*)›
by (*simp add*: *b-def left-diff-distrib infsum-of-real*)
moreover have ‹... ≥ *0* ∗ (*f* ∙$_C$ *f*)› (**is** ‹- ≥ ...›)
apply (*auto intro*!: *mult-nonneg-nonneg*)
using *gbinomial-abs-has-sum-1*[**where** *a*=‹*1/2*›]
by (*auto simp add*: *infsumI*)
moreover have ‹... = *0*›
by *simp*

```
    ultimately show ‹f ·C (B0 *V f) ≥ 0›
      by force
  qed
  then have ‹B ≥ 0›
    by (simp add: B-def k-def scaleR-nonneg-nonneg)
  then have ‹B = B*›
    by (simp add: positive-hermitianI)
  have ‹B0 oCL B0 = id-cblinfun + S›
  proof (rule cblinfun-cinner-eqI)
    fix ψ
    define s bb where ‹s = ψ ·C ((B0 oCL B0) *V ψ)› and ‹bb k = (∑ n≤k. (b n *V ψ) ·C (b
(k − n) *V ψ))› for k

     have ‹bb k = (∑ n≤k. of-real ((1 / 2 gchoose (k − n)) * (1 / 2 gchoose n)) * (ψ ·C
(cblinfun-power S k *V ψ)))› for k
      by (simp add: bb-def[abs-def] b-def cblinfun.scaleR-left cblinfun-power-adj mult.assoc
          flip: cinner-adj-right cblinfun-apply-cblinfun-compose)
    also have ‹... k = of-real (∑ n≤k. ((1 / 2 gchoose n) * (1 / 2 gchoose (k − n)))) * (ψ ·C
(cblinfun-power S k *V ψ))› for k
      apply (subst mult.commute) by (simp add: sum-distrib-right)
    also have ‹... k = of-real (1 gchoose k) * (ψ ·C (cblinfun-power S k *V ψ))› for k
      apply (simp only: atMost-atLeast0 gbinomial-Vandermonde)
      by simp
    also have ‹... k = of-bool (k ≤ 1) * (ψ ·C (cblinfun-power S k *V ψ))› for k
      by (simp add: gbinomial-1)
    finally have bb-simp: ‹bb k = of-bool (k ≤ 1) * (ψ ·C (cblinfun-power S k *V ψ))› for k
      by −

    have bb-sum: ‹bb summable-on UNIV›
      apply (rule summable-on-cong-neutral[where T=‹{..1}› and g=bb, THEN iffD2])
      by (auto simp: bb-simp)

    from has-sum-b have bψ-sum: ‹(λn. b n *V ψ) summable-on UNIV›
      by (simp add: has-sum-imp-summable summable-on-cblinfun-apply-left)

    have b2-pos: ‹(b i *V ψ) ·C (b j *V ψ) ≥ 0› if ‹i≠0› ‹j≠0› for i j
    proof −
      have gchoose-sign: ‹(−1) ^(i+1) * ((1/2 :: real) gchoose i) ≥ 0› if ‹i≠0› for i
      proof −
        obtain j where j: ‹Suc j = i›
          using ‹i ≠ 0› not0-implies-Suc by blast
        show ?thesis
        proof (unfold j[symmetric], induction j)
          case 0
          then show ?case
            by simp
        next
          case (Suc j)
          have ‹(− 1) ^(Suc (Suc j) + 1) * (1 / 2 gchoose Suc (Suc j))
```

```
            = ((− 1) ^ (Suc j + 1) * (1 / 2 gchoose Suc j)) * ((−1) * (1/2−Suc j) / (Suc
(Suc j)))›
          apply (simp add: gbinomial-a-Suc-n)
        by (smt (verit, ccfv-threshold) divide-divide-eq-left' divide-divide-eq-right minus-divide-right)
       also have ‹... ≥ 0›
         apply (rule mult-nonneg-nonneg)
          apply (rule Suc.IH)
         apply (rule divide-nonneg-pos)
          apply (rule mult-nonpos-nonpos)
         by auto
       finally show ?case
         by −
     qed
   qed
   from ‹S ≤ 0›
   have Sn-sign: ‹ψ ·C (cblinfun-power (− S) (i + j) *V ψ) ≥ 0›
     by (auto intro!: cinner-pos-if-pos cblinfun-power-pos)
   have *: ‹(− 1) ^ (i + (j + (i + j))) = (1::complex)›
     by (metis Parity.ring-1-class.power-minus-even even-add power-one)

   have ‹(b i *V ψ) ·C (b j *V ψ)
      = complex-of-real (1 / 2 gchoose i) * complex-of-real (1 / 2 gchoose j)
        * (ψ ·C (cblinfun-power S (i + j) *V ψ))›
     by (simp add: b-def cblinfun.scaleR-right cblinfun.scaleR-left cblinfun-power-adj
        flip: cinner-adj-right cblinfun-apply-cblinfun-compose)
     also have ‹... = complex-of-real ((−1)^(i+1) * (1 / 2 gchoose i)) * complex-of-real
((−1)^(j+1) * (1 / 2 gchoose j))
        * (ψ ·C (cblinfun-power (−S) (i + j) *V ψ))›
     by (simp add: cblinfun.scaleR-left cblinfun-power-uminus * flip: power-add)
   also have ‹... ≥ 0›
     apply (rule mult-nonneg-nonneg)
     apply (rule mult-nonneg-nonneg)
     using complex-of-real-nn-iff gchoose-sign that(1) apply blast
     using complex-of-real-nn-iff gchoose-sign that(2) apply blast
     by (fact Sn-sign)
   finally show ?thesis
     by −
 qed

 have ‹s = (B0 *V ψ) ·C (B0 *V ψ)›
   by (metis ‹0 ≤ B0› cblinfun-apply-cblinfun-compose cinner-adj-left positive-hermitianI
s-def)
 also have ‹... = (∑∞n. b n *V ψ) ·C (∑∞n. b n *V ψ)›
   by (metis B0-def has-sum-b infsum-cblinfun-apply-left has-sum-imp-summable)
 also have ‹... = (∑∞n. bb n)›
   using bψ-sum bψ-sum unfolding bb-def
   apply (rule Cauchy-cinner-product-infsum[symmetric])
   using bψ-sum bψ-sum
   apply (rule Cauchy-cinner-product-summable[where X=‹{0}› and Y=‹{0}›])
```

using *b2-pos* **by** *auto*
also have ‹... = *bb 0* + *bb 1*›
apply (*subst infsum-cong-neutral*[**where** *T*=‹{..*1*}› **and** *g*=*bb*])
by (*auto simp*: *bb-simp*)
also have ‹... = ψ •$_C$ ((*id-cblinfun* + *S*) *$_V$ ψ)›
by (*simp add*: *cblinfun-power-Suc cblinfun.add-left cinner-add-right bb-simp*)
finally show ‹*s* = ψ •$_C$ ((*id-cblinfun* + *S*) *$_V$ ψ)›
by −
qed
then have ‹*B* o_{CL} *B* = *norm A* *$_C$ (*id-cblinfun* + *S*)›
apply (*simp add*: *k-def B-def power2-eq-square scaleR-scaleC*)
by (*metis norm-imp-pos-and-ge of-real-power power2-eq-square real-sqrt-pow2*)
also have ‹... = *A*›
by (*metis* (*no-types*, *lifting*) *k-def S-def add.commute cancel-comm-monoid-add-class.diff-cancel diff-add-cancel norm-eq-zero of-real-1 of-real-mult right-inverse scaleC-diff-right scaleC-one scaleC-scaleC scaleR-scaleC*)
finally have *B2A*: ‹*B* o_{CL} *B* = *A*›
by −
have *BF-comm*: ‹*B* o_{CL} *F* = *F* o_{CL} *B*› **if** ‹*A* o_{CL} *F* = *F* o_{CL} *A*› **for** *F*
proof −
have ‹*S* o_{CL} *F* = *F* o_{CL} *S*›
by (*simp add*: *S-def that*[*symmetric*] *cblinfun-compose-minus-right cblinfun-compose-minus-left flip*: *cblinfun-compose-assoc*)
then have ‹*cblinfun-power S n* o_{CL} *F* = *F* o_{CL} *cblinfun-power S n*› **for** *n*
apply (*induction n*)
apply (*simp-all add*: *cblinfun-power-Suc′ cblinfun-compose-assoc*)
by (*simp flip*: *cblinfun-compose-assoc*)
then have ∗: ‹*b n* o_{CL} *F* = *F* o_{CL} *b n*› **for** *n*
by (*simp add*: *b-def*)
have ‹($\sum_\infty$*n*. *b n*) o_{CL} *F* = *F* o_{CL} ($\sum_\infty$*n*. *b n*)›
proof −
have [*simp*]: ‹*b summable-on UNIV*›
using *has-sum-b* **by** (*auto simp add*: *summable-on-def*)
have ‹($\sum_\infty$*n*. *b n*) o_{CL} *F* = ($\sum_\infty$*n*. (*b n*) o_{CL} *F*)›
apply (*subst infsum-comm-additive*[**where** *f*=‹λ*x*. *x* o_{CL} *F*›, *symmetric*])
by (*auto simp*: *o-def isCont-cblinfun-compose-left*)
also have ‹... = ($\sum_\infty$*n*. *F* o_{CL} (*b n*))›
by (*simp add*: ∗)
also have ‹... = *F* o_{CL} ($\sum_\infty$*n*. *b n*)›
apply (*subst infsum-comm-additive*[**where** *f*=‹λ*x*. *F* o_{CL} *x*›, *symmetric*])
by (*auto simp*: *o-def isCont-cblinfun-compose-right*)
finally show *?thesis*
by −
qed
then have ‹*B0* o_{CL} *F* = *F* o_{CL} *B0*›
unfolding *B0-def*
unfolding *infsum-euclidean-eq*[*abs-def*, *symmetric*]
apply (*transfer fixing*: *b F*)

```
    by simp
  then show ?thesis
    by (auto simp: B-def)
qed
have B-closure: ‹B ∈ closure (cspan (range (cblinfun-power A)))›
proof (cases ‹k = 0›)
  case True
  then show ?thesis
    unfolding B-def using closure-subset complex-vector.span-zero by auto
next
  case False
  then have ‹k ≠ 0›
    by −
  from has-sum-b
  have limit: ‹(sum b ⟶ B0) (finite-subsets-at-top UNIV)›
    by (simp add: has-sum-def)
  have ‹cblinfun-power (A /ℝ k − id-cblinfun) n ∈ cspan (range (cblinfun-power A))› for n
  proof (induction n)
    case 0
    then show ?case
      by (auto intro!: complex-vector.span-base range-eqI[where x=0])
  next
    case (Suc n)
    define pow-n where ‹pow-n = cblinfun-power (A /ℝ k − id-cblinfun) n›
    have pow-n-span: ‹pow-n ∈ cspan (range (cblinfun-power A))›
      using Suc by (simp add: pow-n-def)
    have A-pow-n-span: ‹A o_CL pow-n ∈ cspan (range (cblinfun-power A))›
    proof −
      from pow-n-span
      obtain F r where ‹finite F› and F-A: ‹F ⊆ range (cblinfun-power A)›
        and pow-n-sum: ‹pow-n = (∑ a∈F. r a *_C a)›
        by (auto simp add: complex-vector.span-explicit)
      have ‹A o_CL a ∈ range (cblinfun-power A)› if ‹a ∈ F› for a
      proof −
        from that obtain m where ‹a = cblinfun-power A m›
          using F-A by auto
        then have ‹A o_CL a = cblinfun-power A (Suc m)›
          by (simp add: cblinfun-power-Suc')
        then show ?thesis
          by auto
      qed
      then have ‹(∑ a∈F. r a *_C (A o_CL a)) ∈ cspan (range (cblinfun-power A))›
          by (meson basic-trans-rules(31) complex-vector.span-scale complex-vector.span-sum
complex-vector.span-superset)
      moreover have ‹A o_CL pow-n = (∑ a∈F. r a *_C (A o_CL a))›
        by (simp add: pow-n-sum cblinfun-compose-sum-right flip: cblinfun.scaleC-left)
      ultimately show ?thesis
        by simp
    qed
```

```
    have ‹cblinfun-power (A /R k − id-cblinfun) (Suc n) = (A oCL pow-n) /R k − pow-n›
      by (simp add: cblinfun-power-Suc′ cblinfun-compose-minus-left flip: pow-n-def)
    also from pow-n-span A-pow-n-span
    have ‹... ∈ cspan (range (cblinfun-power A))›
      by (auto intro!: complex-vector.span-diff complex-vector.span-scale
          simp: scaleR-scaleC)
    finally show ?case
      by −
  qed
  then have b-range: ‹b n ∈ cspan (range (cblinfun-power A))› for n
    by (simp add: b-def S-def scaleR-scaleC complex-vector.span-scale)
  have sum-bF: ‹sum b F ∈ cspan (range (cblinfun-power A))› if ‹finite F› for F
    using that apply induction
    using b-range complex-vector.span-add complex-vector.span-zero by auto
  have ‹B0 ∈ closure (cspan (range (cblinfun-power A)))›
    using limit apply (rule limit-in-closure)
    using sum-bF by (simp-all add: eventually-finite-subsets-at-top-weakI)
  also have ‹... = closure ((λx. inverse (sqrt k) *R x) ‘ cspan (range (cblinfun-power A)))›
    using ‹k ≠ 0› by (simp add: scaleR-scaleC csubspace-scaleC-invariant)
  also have ‹... = (λx. inverse (sqrt k) *R x) ‘ closure (cspan (range (cblinfun-power A)))›
    by (simp add: closure-scaleR)
  finally show ?thesis
    apply (simp add: B-def image-def)
    using ‹k ≠ 0› by force
qed
from ‹B ≥ 0› B2A BF-comm B-closure
show ?thesis
  by metis
qed


lemma wecken35hilfssatz:
  — Auxiliary lemma from [9]
  ‹∃P. is-Proj P ∧ (∀F. F oCL (W − T) = (W − T) oCL F ⟶ F oCL P = P oCL F)
    ∧ (∀f. W f = 0 ⟶ P f = f)
    ∧ (W = (2 *C P − id-cblinfun) oCL T)›
  if WT-comm: ‹W oCL T = T oCL W› and ‹W = W*› and ‹T = T*›
    and WW-TT: ‹W oCL W = T oCL T›
  for W T :: ‹'a::chilbert-space ⇒CL 'a›
proof (rule exI, intro conjI allI impI)
  define P where ‹P = Proj (kernel (W−T))›
  show ‹is-Proj P›
    by (simp add: P-def)
  show thesis1: ‹F oCL P = P oCL F› if ‹F oCL (W − T) = (W − T) oCL F› for F
  proof −
    have 1: ‹F oCL P = P oCL F oCL P› if ‹F oCL (W − T) = (W − T) oCL F› for F
    proof (rule cblinfun-eqI)
      fix ψ
      have ‹P *V ψ ∈ space-as-set (kernel (W − T))›
```

```
    by (metis P-def Proj-range cblinfun-apply-in-image)
  then have ‹(W − T) *V P *V ψ = 0›
    using kernel-memberD by blast
  then have ‹(W − T) *V F *V P *V ψ = 0›
    by (metis cblinfun.zero-right cblinfun-apply-cblinfun-compose that)
  then have ‹F *V P *V ψ ∈ space-as-set (kernel (W − T))›
    using kernel-memberI by blast
  then have ‹P *V (F *V P *V ψ) = F *V P *V ψ›
    using P-def Proj-fixes-image by blast
  then show ‹(F oCL P) *V ψ = (P oCL F oCL P) *V ψ›
    by simp
qed
have 2: ‹F* oCL (W − T) = (W − T) oCL F*›
  by (metis ‹T = T*› ‹W = W*› adj-cblinfun-compose adj-minus that)
have ‹F oCL P = P oCL F oCL P› and ‹F* oCL P = P oCL F* oCL P›
  using 1[OF that] 1[OF 2] by auto
then show ‹F oCL P = P oCL F›
  by (metis P-def adj-Proj adj-cblinfun-compose cblinfun-assoc-left(1) double-adj)
qed
show thesis2: ‹P *V f = f› if ‹W *V f = 0› for f
proof −
  from that
  have ‹0 = (W *V f) •C (W *V f)›
    by simp
  also from ‹W = W*› have ‹... = f •C ((W oCL W) *V f)›
    by (simp add: that)
  also from WW-TT have ‹... = f •C ((T oCL T) *V f)›
    by simp
  also from ‹T = T*› have ‹... = (T *V f) •C (T *V f)›
    by (metis cblinfun-apply-cblinfun-compose cinner-adj-left)
  finally have ‹T *V f = 0›
    by simp
  then have ‹(W − T) *V f = 0›
    by (simp add: cblinfun.diff-left that)
  then show ‹P *V f = f›
    using P-def Proj-fixes-image kernel-memberI by blast
qed
show thesis3: ‹W = (2 *C P − id-cblinfun) oCL T›
proof −
  from WW-TT WT-comm have WT-binomial: ‹(W − T) oCL (W + T) = 0›
    by (simp add: cblinfun-compose-add-right cblinfun-compose-minus-left)
  have PWT: ‹P oCL (W + T) = W + T›
  proof (rule cblinfun-eqI)
    fix ψ
    from WT-binomial have ‹(W + T) *V ψ ∈ space-as-set (kernel (W−T))›
      by (metis cblinfun-apply-cblinfun-compose kernel-memberI zero-cblinfun.rep-eq)
    then show ‹(P oCL (W + T)) *V ψ = (W + T) *V ψ›
    by (metis P-def Proj-idempotent Proj-range cblinfun-apply-cblinfun-compose cblinfun-fixes-range)
  qed
```

from *P-def* **have** ‹$(W - T)\ o_{CL}\ P = 0$›
by (*metis Proj-range thesis1 cblinfun-apply-cblinfun-compose cblinfun-apply-in-image cblinfun-eqI kernel-memberD zero-cblinfun.rep-eq*)
with *PWT WT-comm thesis1* **have** ‹$2 *_C\ T\ o_{CL}\ P = W + T$›
by (*metis* (*no-types*, *lifting*) *bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose cblinfun-compose-add-right cblinfun-compose-minus-left cblinfun-compose-minus-right eq-iff-diff-eq-0 scaleC-2*)
with *that*(*2*) *that*(*3*) **show** *?thesis*
by (*smt* (*verit*, *ccfv-threshold*) *P-def add-diff-cancel adj-Proj adj-cblinfun-compose adj-plus cblinfun-compose-id-right cblinfun-compose-minus-left cblinfun-compose-scaleC-left id-cblinfun-adjoint scaleC-2*)
qed
qed

lemma *sqrt-op-pos*[*simp*]: ‹*sqrt-op a* $\geq$ *0*›
proof (*cases* ‹$a \geq 0$›)
case *True*
from *sqrt-op-existence*[*OF True*]
have $*$: ‹$\exists\, b{::}'a \Rightarrow_{CL} {}'a.\ b \geq 0 \wedge b* \ o_{CL}\ b = a$›
by (*metis positive-hermitianI*)
then show *?thesis*
using $*$ **by** (*smt* (*verit*, *ccfv-threshold*) *someI-ex sqrt-op-def*)
next
case *False*
then show *?thesis*
by (*simp add*: *sqrt-op-nonpos*)
qed

lemma *sqrt-op-square*[*simp*]:
assumes ‹$a \geq 0$›
shows ‹*sqrt-op a* o_{CL} *sqrt-op a* = *a*›
proof −
from *sqrt-op-existence*[*OF assms*]
have $*$: ‹$\exists\, b{::}'a \Rightarrow_{CL} {}'a.\ b \geq 0 \wedge b* \ o_{CL}\ b = a$›
by (*metis positive-hermitianI*)
have ‹*sqrt-op a* o_{CL} *sqrt-op a* = (*sqrt-op a*)$*$ o_{CL} *sqrt-op a*›
by (*metis positive-hermitianI sqrt-op-pos*)
also have ‹(*sqrt-op a*)$*$ o_{CL} *sqrt-op a* = *a*›
using $*$ **by** (*metis* (*mono-tags*, *lifting*) *someI-ex sqrt-op-def*)
finally show *?thesis*
by −
qed

lemma *sqrt-op-unique*:
— Proof follows [9]
assumes ‹$b \geq 0$› **and** ‹$b* \ o_{CL}\ b = a$›
shows ‹*b* = *sqrt-op a*›
proof −
have ‹$a \geq 0$›

using *assms(2) positive-cblinfun-squareI* **by** *blast*
from *sqrt-op-existence*[*OF* ‹$a \geq 0$›]
obtain *sq* **where** ‹$sq \geq 0$› **and** ‹$sq\ o_{CL}\ sq = a$› **and** *a-comm*: ‹$a\ o_{CL}\ F = F\ o_{CL}\ a \Longrightarrow sq\ o_{CL}\ F = F\ o_{CL}\ sq$› **for** *F*
by *metis*
have *eq-sq*: ‹$b = sq$› **if** ‹$b \geq 0$› **and** ‹$b* \ o_{CL}\ b = a$› **for** *b*
proof −
have ‹$b\ o_{CL}\ a = a\ o_{CL}\ b$›
by (*metis cblinfun-assoc-left(1) positive-hermitianI that(1) that(2)*)
then have *b-sqrt-comm*: ‹$b\ o_{CL}\ sq = sq\ o_{CL}\ b$›
using *a-comm* **by** *force*
from ‹$b \geq 0$› **have** ‹$b = b*$›
by (*simp add*: *assms(1) positive-hermitianI*)
have *sqrt-adj*: ‹$sq = sq*$›
by (*simp add*: ‹$0 \leq sq$› *positive-hermitianI*)
have *bb-sqrt*: ‹$b\ o_{CL}\ b = sq\ o_{CL}\ sq$›
using ‹$b = b*$› ‹$sq\ o_{CL}\ sq = a$› *that(2)* **by** *fastforce*

from *wecken35hilfssatz*[*OF b-sqrt-comm* ‹$b = b*$› *sqrt-adj bb-sqrt*]
obtain *P* **where** ‹*is-Proj P*› **and** *b-P-sq*: ‹$b = (2 *_C P - id\text{-}cblinfun)\ o_{CL}\ sq$›
and *Pcomm*: ‹$F\ o_{CL}\ (b - sq) = (b - sq)\ o_{CL}\ F \Longrightarrow F\ o_{CL}\ P = P\ o_{CL}\ F$› **for** *F*
by *metis*

have *1*: ‹*sandwich* (*id-cblinfun* − *P*) *b* = (*id-cblinfun* − *P*) o_{CL} *b*›
by (*smt* (*verit*, *del-insts*) *Pcomm* ‹*is-Proj P*› *b-sqrt-comm cblinfun-assoc-left(1) cblinfun-compose-id-left cblinfun-compose-id-right cblinfun-compose-minus-left cblinfun-compose-minus-right cblinfun-compose-zero-left diff-0-right is-Proj-algebraic is-Proj-complement is-Proj-idempotent sandwich-apply*)
also have *2*: ‹$\ldots = -\ (id\text{-}cblinfun - P)\ o_{CL}\ sq$›
apply (*simp add*: *b-P-sq*)
by (*smt* (*verit*, *del-insts*) ‹$0 \leq sq$› ‹*is-Proj P*› *add-diff-cancel-left' cancel-comm-monoid-add-class.diff-cancel cblinfun-compose-assoc cblinfun-compose-id-right cblinfun-compose-minus-right diff-diff-eq2 is-Proj-algebraic is-Proj-complement minus-diff-eq scaleC-2*)
also have ‹$\ldots = -$ *sandwich* (*id-cblinfun* − *P*) *sq*›
by (*metis* ‹(*id-cblinfun* − *P*) o_{CL} *b* = − (*id-cblinfun* − *P*) o_{CL} *sq*› *calculation cblinfun-compose-uminus-left sandwich-apply*)
also have ‹$\ldots \leq 0$›
by (*simp add*: ‹$0 \leq sq$› *sandwich-pos*)
finally have ‹*sandwich* (*id-cblinfun* − *P*) *b* ≤ 0›
by −
moreover from ‹$b \geq 0$› **have** ‹*sandwich* (*id-cblinfun* − *P*) *b* ≥ 0›
by (*simp add*: *sandwich-pos*)
ultimately have ‹*sandwich* (*id-cblinfun* − *P*) *b* = 0›
by *auto*
with *1 2* **have** ‹(*id-cblinfun* − *P*) o_{CL} *sq* = 0›
by (*metis add.inverse-neutral cblinfun-compose-uminus-left minus-diff-eq*)
with *b-P-sq* **show** ‹*b* = *sq*›
by (*metis* (*no-types*, *lifting*) *add.inverse-neutral add-diff-cancel-right' adj-cblinfun-compose cblinfun-compose-id-right cblinfun-compose-minus-left diff-0 diff-eq-diff-eq id-cblinfun-adjoint scaleC-2*

sqrt-adj)
qed

from *eq-sq* **have** ‹*sqrt-op a = sq*›
by (*simp add*: ‹*0 ≤ a*› *comparable-hermitean*[*unfolded selfadjoint-def*])
moreover from *eq-sq* **have** ‹*b = sq*›
by (*simp add*: *assms*(*1*) *assms*(*2*))
ultimately show ‹*b = sqrt-op a*›
by *simp*
qed

lemma *sqrt-op-in-closure*: ‹*sqrt-op a ∈ closure* (*cspan* (*range* (*cblinfun-power a*)))›
proof (*cases* ‹*a ≥ 0*›)
case *True*
from *sqrt-op-existence*[*OF True*]
obtain *B* :: ‹*'a ⇒*$_{CL}$ *'a*› **where** ‹*B ≥ 0*› **and** ‹*B* o_{CL} *B = a*›
and *B-closure*: ‹*B ∈ closure* (*cspan* (*range* (*cblinfun-power a*)))›
by *metis*
then have ‹*sqrt-op a = B*›
by (*metis positive-hermitianI sqrt-op-unique*)
with *B-closure* **show** *?thesis*
by *simp*
next
case *False*
then have ‹*sqrt-op a = 0*›
by (*simp add*: *sqrt-op-nonpos*)
also have ‹*0 ∈ closure* (*cspan* (*range* (*cblinfun-power a*)))›
using *closure-subset complex-vector.span-zero* **by** *blast*
finally show *?thesis*
by −
qed

lemma *sqrt-op-commute*:
assumes ‹*A ≥ 0*›
assumes ‹*A* o_{CL} *F = F* o_{CL} *A*›
shows ‹*sqrt-op A* o_{CL} *F = F* o_{CL} *sqrt-op A*›
by (*metis assms*(*1*) *assms*(*2*) *positive-hermitianI sqrt-op-existence sqrt-op-unique*)

lemma *sqrt-op-0*[*simp*]: ‹*sqrt-op 0 = 0*›
apply (*rule sqrt-op-unique*[*symmetric*])
by *auto*

lemma *sqrt-op-scaleC*:
assumes ‹*c ≥ 0*› **and** ‹*a ≥ 0*›
shows ‹*sqrt-op* (*c* $*_C$ *a*) = *sqrt c* $*_C$ *sqrt-op a*›
apply (*rule sqrt-op-unique*[*symmetric*])
using *assms* **apply** (*auto simp*: *split-scaleC-pos-le positive-hermitianI*)
by (*metis of-real-power power2-eq-square real-sqrt-pow2*)

definition *abs-op* :: ‹'a::*chilbert-space* ⇒$_{CL}$ 'b::*complex-inner* ⇒ 'a ⇒$_{CL}$ 'a› **where** ‹*abs-op a* = *sqrt-op* (*a*∗ *o*$_{CL}$ *a*)›

lemma *abs-op-pos*[*simp*]: ‹*abs-op a* ≥ *0*›
by (*simp add*: *abs-op-def positive-cblinfun-squareI sqrt-op-pos*)

lemma *abs-op-0*[*simp*]: ‹*abs-op 0* = *0*›
unfolding *abs-op-def* **by** *auto*

lemma *abs-op-idem*[*simp*]: ‹*abs-op* (*abs-op a*) = *abs-op a*›
by (*metis abs-op-def abs-op-pos sqrt-op-unique*)

lemma *abs-op-uminus*[*simp*]: ‹*abs-op* (− *a*) = *abs-op a*›
by (*simp add*: *abs-op-def adj-uminus bounded-cbilinear.minus-left*
bounded-cbilinear.minus-right bounded-cbilinear-cblinfun-compose)

lemma *selfbutter-pos*[*simp*]: ‹*selfbutter x* ≥ *0*›
by (*metis butterfly-def double-adj positive-cblinfun-squareI*)

lemma *abs-op-butterfly*[*simp*]: ‹*abs-op* (*butterfly x y*) = (*norm x* / *norm y*) ∗$_R$ *selfbutter y*› **for** *x* :: ‹'a::*chilbert-space*› **and** *y* :: ‹'b::*chilbert-space*›
proof (*cases* ‹*y*=*0*›)
case *False*
have ‹*abs-op* (*butterfly x y*) = *sqrt-op* (*cinner x x* ∗$_C$ *selfbutter y*)›
unfolding *abs-op-def* **by** *simp*
also have ‹... = (*norm x* / *norm y*) ∗$_R$ *selfbutter y*›
apply (*rule sqrt-op-unique*[*symmetric*])
using *False* **by** (*auto intro*!: *scaleC-nonneg-nonneg simp*: *scaleR-scaleC power2-eq-square simp flip*: *power2-norm-eq-cinner*)
finally show *?thesis*
by −
next
case *True*
then show *?thesis*
by *simp*
qed

lemma *abs-op-nondegenerate*: ‹*a* = *0*› **if** ‹*abs-op a* = *0*›
proof −
from *that*
have ‹*sqrt-op* (*a*∗ *o*$_{CL}$ *a*) = *0*›
by (*simp add*: *abs-op-def*)
then have ‹*0*∗ *o*$_{CL}$ *0* = (*a*∗ *o*$_{CL}$ *a*)›
by (*metis cblinfun-compose-zero-right positive-cblinfun-squareI sqrt-op-square*)
then show ‹*a* = *0*›
apply (*rule-tac op-square-nondegenerate*)
by *simp*

qed

lemma *abs-op-scaleC*: ‹*abs-op* (*c* $*_C$ *a*) = |*c*| $*_C$ *abs-op a*›
proof −
define *aa* **where** ‹*aa* = *a** o_{CL} *a*›
have ‹*abs-op* (*c* $*_C$ *a*) = *sqrt-op* (|*c*|^2 $*_C$ *aa*)›
by (*simp add*: *abs-op-def x-cnj-x aa-def*)
also have ‹... = |*c*| $*_C$ *sqrt-op aa*›
by (*smt* (*verit*, *best*) *aa-def abs-complex-def abs-nn cblinfun-compose-scaleC-left cblinfun-compose-scaleC-right complex-cnj-complex-of-real o-apply positive-cblinfun-squareI power2-eq-square scaleC-adj scaleC-nonneg-nonneg scaleC-scaleC sqrt-op-pos sqrt-op-square sqrt-op-unique*)
also have ‹... = |*c*| $*_C$ *abs-op a*›
by (*simp add*: *aa-def abs-op-def*)
finally show *?thesis*
by −
qed

lemma *kernel-abs-op*[*simp*]: ‹*kernel* (*abs-op a*) = *kernel a*›
proof (*rule ccsubspace-eqI*)
fix *x*
have ‹*x* ∈ *space-as-set* (*kernel* (*abs-op a*)) ⟷ *abs-op a x* = *0*›
using *kernel-memberD kernel-memberI* **by** *blast*
also have ‹... ⟷ *abs-op a x* $\cdot_C$ *abs-op a x* = *0*›
by *simp*
also have ‹... ⟷ *x* $\cdot_C$ ((*abs-op a*)* o_{CL} *abs-op a*) *x* = *0*›
by (*simp add*: *cinner-adj-right*)
also have ‹... ⟷ *x* $\cdot_C$ (*a** o_{CL} *a*) *x* = *0*›
by (*simp add*: *abs-op-def positive-cblinfun-squareI positive-hermitianI*)
also have ‹... ⟷ *a x* $\cdot_C$ *a x* = *0*›
by (*simp add*: *cinner-adj-right*)
also have ‹... ⟷ *a x* = *0*›
by *simp*
also have ‹... ⟷ *x* ∈ *space-as-set* (*kernel a*)›
using *kernel-memberD kernel-memberI* **by** *auto*
finally show ‹*x* ∈ *space-as-set* (*kernel* (*abs-op a*)) ⟷ *x* ∈ *space-as-set* (*kernel a*)›
by −
qed

definition *polar-decomposition* **where**
— [1], 3.9 Polar Decomposition
‹*polar-decomposition A* = *cblinfun-extension* (*range* (*abs-op A*)) (λψ. *A* $*_V$ *inv* (*abs-op A*) ψ) o_{CL} *Proj* (*abs-op A* $*_S$ *top*)›
for *A* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*complex-inner*›

lemma
fixes *A* :: ‹'*a* :: *chilbert-space* ⇒$_{CL}$ '*b* :: *chilbert-space*›
— [1], 3.9 Polar Decomposition
shows *polar-decomposition-correct*: ‹*polar-decomposition A* o_{CL} *abs-op A* = *A*›

and *polar-decomposition-final-space*: ‹*polar-decomposition A* $*_S$ *top* = *A* $*_S$ *top*›
and *polar-decomposition-initial-space*[*simp*]: ‹*kernel* (*polar-decomposition A*) = *kernel A*›
and *polar-decomposition-partial-isometry*[*simp*]: ‹*partial-isometry* (*polar-decomposition A*)›
proof −
have *abs-A-norm*: ‹*norm* (*abs-op A h*) = *norm* (*A h*)› **for** *h*
proof −
have ‹*complex-of-real* ((*norm* (*A h*))^2) = *A h* $\cdot_C$ *A h*›
by (*simp add*: *cdot-square-norm*)
also have ‹... = (*A*$*$ o_{CL} *A*) *h* $\cdot_C$ *h*›
by (*simp add*: *cinner-adj-left*)
also have ‹... = ((*abs-op A*)$*$ o_{CL} *abs-op A*) *h* $\cdot_C$ *h*›
by (*simp add*: *abs-op-def positive-cblinfun-squareI positive-hermitianI*)
also have ‹... = *abs-op A h* $\cdot_C$ *abs-op A h*›
by (*simp add*: *cinner-adj-left*)
also have ‹... = *complex-of-real* ((*norm* (*abs-op A h*))^2)›
using *cnorm-eq-square* **by** *blast*
finally show *?thesis*
by (*simp add*: *cdot-square-norm cnorm-eq*)
qed

define *W W' P*
where ‹*W* = (*λψ*. *A* $*_V$ *inv* (*abs-op A*) *ψ*)›
and ‹*W'* = *cblinfun-extension* (*range* (*abs-op A*)) *W*›
and ‹*P* = *Proj* (*abs-op A* $*_S$ *top*)›

have *pdA*: ‹*polar-decomposition A* = *W'* o_{CL} *P*›
by (*auto simp*: *polar-decomposition-def W'-def W-def P-def*)

have *AA-norm*: ‹*norm* (*W ψ*) = *norm ψ*› **if** ‹*ψ* ∈ *range* (*abs-op A*)› **for** *ψ*
proof −
define *h* **where** ‹*h* = *inv* (*abs-op A*) *ψ*›
from *that* **have** *absA-h*: ‹*abs-op A h* = *ψ*›
by (*simp add*: *f-inv-into-f h-def*)
have ‹*complex-of-real* ((*norm* (*W ψ*))^2) = *complex-of-real* ((*norm* (*A h*))^2)›
using *W-def h-def* **by** *blast*
also have ‹... = *A h* $\cdot_C$ *A h*›
by (*simp add*: *cdot-square-norm*)
also have ‹... = (*A*$*$ o_{CL} *A*) *h* $\cdot_C$ *h*›
by (*simp add*: *cinner-adj-left*)
also have ‹... = ((*abs-op A*)$*$ o_{CL} *abs-op A*) *h* $\cdot_C$ *h*›
by (*simp add*: *abs-op-def positive-cblinfun-squareI positive-hermitianI*)
also have ‹... = *abs-op A h* $\cdot_C$ *abs-op A h*›
by (*simp add*: *cinner-adj-left*)
also have ‹... = *complex-of-real* ((*norm* (*abs-op A h*))^2)›
using *cnorm-eq-square* **by** *blast*
also have ‹... = *complex-of-real* ((*norm ψ*)^2)›
using *absA-h* **by** *fastforce*
finally show ‹*norm* (*W ψ*) = *norm ψ*›
by (*simp add*: *cdot-square-norm cnorm-eq*)

qed
then have *AA-norm'*: ‹*norm* (*W* ψ) $\leq$ *1* $*$ *norm* ψ› **if** ‹ψ $\in$ *range* (*abs-op A*)› **for** ψ
using *that* **by** *simp*

have *W-absA*: ‹*W* (*abs-op A h*) = *A h*› **for** *h*
proof −
have ‹*A h* = *A h'*› **if** ‹*abs-op A h* = *abs-op A h'*› **for** *h h'*
proof −
from *that* **have** ‹*norm* (*abs-op A* (*h* − *h'*)) = *0*›
by (*simp add*: *cblinfun.diff-right*)
with *AA-norm* **have** ‹*norm* (*A* (*h* − *h'*)) = *0*›
by (*simp add*: *abs-A-norm*)
then show ‹*A h* = *A h'*›
by (*simp add*: *cblinfun.diff-right*)
qed
then show *?thesis*
by (*metis W-def f-inv-into-f rangeI*)
qed

have *range-subspace*: ‹*csubspace* (*range* (*abs-op A*))›
by (*auto intro*!: *range-is-csubspace*)

have *exP*: ‹∃ *P*. *is-Proj P* ∧ *range* ((*$*$_V*) *P*) = *closure* (*range* ((*$*$_V*) (*abs-op A*)))›
apply (*rule exI*[*of* - ‹*Proj* (*abs-op A* *$*$_S* ⊤)›])
by (*metis* (*no-types*, *opaque-lifting*) *Proj-is-Proj Proj-range Proj-range-closed cblinfun-image.rep-eq closure-closed space-as-set-top*)

have *add*: ‹*W* (*x* + *y*) = *W x* + *W y*› **if** *x-in*: ‹*x* $\in$ *range* (*abs-op A*)› **and** *y-in*: ‹*y* $\in$ *range* (*abs-op A*)› **for** *x y*
proof −
obtain *x' y'* **where** ‹*x* = *abs-op A x'*› **and** ‹*y* = *abs-op A y'*›
using *x-in y-in* **by** *blast*
then show *?thesis*
by (*simp flip*: *cblinfun.add-right add*: *W-absA*)
qed

have *scale*: ‹*W* (*c* *$*$_C* *x*) = *c* *$*$_C* *W x*› **if** *x-in*: ‹*x* $\in$ *range* (*abs-op A*)› **for** *c x*
proof −
obtain *x'* **where** ‹*x* = *abs-op A x'*›
using *x-in* **by** *blast*
then show *?thesis*
by (*simp flip*: *cblinfun.scaleC-right add*: *W-absA*)
qed

have ‹*cblinfun-extension-exists* (*range* (*abs-op A*)) *W*›
using *range-subspace exP add scale AA-norm'*
by (*rule cblinfun-extension-exists-proj*)

then have *W'-apply*: ‹*W'* *$*$_V* ψ = *W* ψ› **if** ‹ψ $\in$ *range* (*abs-op A*)› **for** ψ

by (*simp add*: *W'-def cblinfun-extension-apply that*)

have ‹*norm* (*W'* *ψ*) − *norm* *ψ* = *0*› **if** ‹*ψ* ∈ *range* (*abs-op A*)› **for** *ψ*
by (*simp add*: *W'-apply AA-norm that*)
then have ‹*norm* (*W'* *ψ*) − *norm* *ψ* = *0*› **if** ‹*ψ* ∈ *closure* (*range* (*abs-op A*))› **for** *ψ*
apply (*rule-tac continuous-constant-on-closure*[**where** *S*=‹*range* (*abs-op A*)›])
using *that* **by** (*auto intro*!: *continuous-at-imp-continuous-on*)
then have *norm-W'*: ‹*norm* (*W'* *ψ*) = *norm* *ψ*› **if** ‹*ψ* ∈ *space-as-set* (*abs-op A* ∗*S* *top*)› **for** *ψ*
using *cblinfun-image.rep-eq that* **by** *force*

show *correct*: ‹*polar-decomposition A* *oCL* *abs-op A* = *A*›
proof (*rule cblinfun-eqI*)
fix *ψ* :: *'a*
have ‹(*polar-decomposition A* *oCL* *abs-op A*) ∗*V* *ψ* = *W* (*P* (*abs-op A* *ψ*))›
by (*simp add*: *W'-apply P-def pdA Proj-fixes-image*)
also have ‹... = *W* (*abs-op A* *ψ*)›
by (*auto simp*: *P-def Proj-fixes-image*)
also have ‹... = *A* *ψ*›
by (*simp add*: *W-absA*)

finally show ‹(*polar-decomposition A* *oCL* *abs-op A*) ∗*V* *ψ* = *A* ∗*V* *ψ*›
by −
qed

show ‹*polar-decomposition A* ∗*S* *top* = *A* ∗*S* *top*›
proof (*rule antisym*)
have ∗: ‹*A* ∗*S* *top* = *polar-decomposition A* ∗*S* *abs-op A* ∗*S* *top*›
by (*simp add*: *cblinfun-assoc-left*(*2*) *correct*)
also have ‹... ≤ *polar-decomposition A* ∗*S* *top*›
by (*simp add*: *cblinfun-image-mono*)
finally show ‹*A* ∗*S* *top* ≤ *polar-decomposition A* ∗*S* *top*›
by −

have ‹*W'* *ψ* ∈ *range A*› **if** ‹*ψ* ∈ *range* (*abs-op A*)› **for** *ψ*
using *W'-apply W-def that* **by** *blast*
then have ‹*W'* *ψ* ∈ *closure* (*range A*)› **if** ‹*ψ* ∈ *closure* (*range* (*abs-op A*))› **for** *ψ*
using ∗
by (*metis* (*mono-tags*, *lifting*) *P-def Proj-range Proj-fixes-image cblinfun-apply-cblinfun-compose cblinfun-apply-in-image cblinfun-compose-image cblinfun-image.rep-eq pdA that top-ccsubspace.rep-eq*)
then have ‹*W'* *ψ* ∈ *space-as-set* (*A* ∗*S* *top*)› **if** ‹*ψ* ∈ *space-as-set* (*abs-op A* ∗*S* *top*)› **for** *ψ*
by (*metis cblinfun-image.rep-eq that top-ccsubspace.rep-eq*)
then have ‹*polar-decomposition A* *ψ* ∈ *space-as-set* (*A* ∗*S* *top*)› **for** *ψ*
by (*metis P-def Proj-range cblinfun-apply-cblinfun-compose cblinfun-apply-in-image pdA*)
then show ‹*polar-decomposition A* ∗*S* *top* ≤ *A* ∗*S* *top*›
using ∗
by (*metis* (*no-types*, *lifting*) *Proj-idempotent Proj-range cblinfun-compose-image dual-order.eq-iff polar-decomposition-def*)
qed

show ‹*partial-isometry* (*polar-decomposition A*)›
apply (*rule partial-isometryI'*[**where** *V*=‹*abs-op A* $*_S$ *top*›])
by (*auto simp add*: *P-def Proj-fixes-image norm-W' pdA kernel-memberD*)

have ‹*kernel* (*polar-decomposition A*) = − (*abs-op A* $*_S$ *top*)›
apply (*rule partial-isometry-initial*[**where** *V*=‹*abs-op A* $*_S$ *top*›])
by (*auto simp add*: *P-def Proj-fixes-image norm-W' pdA kernel-memberD*)
also have ‹... = *kernel* (*abs-op A*)›
by (*metis abs-op-pos kernel-compl-adj-range positive-hermitianI*)
also have ‹... = *kernel A*›
by (*simp add*: *kernel-abs-op*)
finally show ‹*kernel* (*polar-decomposition A*) = *kernel A*›
by −
qed

lemma *polar-decomposition-correct'*: ‹(*polar-decomposition A*)* o_{CL} *A* = *abs-op A*›
for *A* :: ‹'*a* :: *chilbert-space* ⇒$_{CL}$ '*b* :: *chilbert-space*›
proof −
have ‹*polar-decomposition A** o_{CL} *A* = (*polar-decomposition A** o_{CL} *polar-decomposition A*) o_{CL} *abs-op A*›
by (*simp add*: *cblinfun-compose-assoc polar-decomposition-correct*)
also have ‹... = *Proj* (− *kernel* (*polar-decomposition A*)) o_{CL} *abs-op A*›
by (*simp add*: *partial-isometry-adj-a-o-a polar-decomposition-partial-isometry*)
also have ‹... = *Proj* (− *kernel A*) o_{CL} *abs-op A*›
by (*simp add*: *polar-decomposition-initial-space*)
also have ‹... = *Proj* (− *kernel* (*abs-op A*)) o_{CL} *abs-op A*›
by *simp*
also have ‹... = *Proj* (*abs-op A* $*_S$ *top*) o_{CL} *abs-op A*›
by (*metis abs-op-pos kernel-compl-adj-range ortho-involution positive-hermitianI*)
also have ‹... = *abs-op A*›
by (*simp add*: *Proj-fixes-image cblinfun-eqI*)
finally show *?thesis*
by −
qed

lemma *abs-op-adj*: ‹*abs-op* (*a**) = *sandwich* (*polar-decomposition a*) (*abs-op a*)›
proof −
have *pos*: ‹*sandwich* (*polar-decomposition a*) (*abs-op a*) ≥ *0*›
by (*simp add*: *sandwich-pos*)
have ‹(*sandwich* (*polar-decomposition a*) (*abs-op a*))* o_{CL} (*sandwich* (*polar-decomposition a*) (*abs-op a*))
= *polar-decomposition a* o_{CL} (*abs-op a*)* o_{CL} *abs-op a* o_{CL} (*polar-decomposition a*)*›
apply (*simp add*: *sandwich-apply*)
by (*metis* (*no-types*, *lifting*) *cblinfun-assoc-left*(*1*) *polar-decomposition-correct polar-decomposition-correct'*)
also have ‹... = *a* o_{CL} *a**›
by (*metis abs-op-pos adj-cblinfun-compose cblinfun-assoc-left*(*1*) *polar-decomposition-correct positive-hermitianI*)
finally have ‹*sandwich* (*polar-decomposition a*) (*abs-op a*) = *sqrt-op* (*a* o_{CL} *a**)›

```
    using pos by (simp add: sqrt-op-unique)
  also have ‹... = abs-op (a*)›
    by (simp add: abs-op-def)
  finally show ?thesis
    by simp
qed

lemma abs-opI:
  assumes ‹a* o_CL a = b* o_CL b›
  assumes ‹a ≥ 0›
  shows ‹a = abs-op b›
  by (simp add: abs-op-def assms(1) assms(2) sqrt-op-unique)

lemma abs-op-id-on-pos: ‹a ≥ 0 ⟹ abs-op a = a›
  using abs-opI by force

lemma norm-abs-op[simp]: ‹norm (abs-op a) = norm a›
  for a :: ‹'a::chilbert-space ⇒_CL 'b::chilbert-space›
proof −
  have ‹(norm (abs-op a))² = norm (abs-op a* o_CL abs-op a)›
    by simp
  also have ‹... = norm (a* o_CL a)›
    by (simp add: abs-op-def positive-cblinfun-squareI positive-hermitianI)
  also have ‹... = (norm a)²›
    by simp
  finally show ?thesis
    by simp
qed



lemma partial-isometry-iff-square-proj:
  — [2], Exercise VIII.3.15
  fixes A :: ‹'a :: chilbert-space ⇒_CL 'b :: chilbert-space›
  shows ‹partial-isometry A ⟷ is-Proj (A* o_CL A)›
proof (rule iffI)
  show ‹is-Proj (A* o_CL A)› if ‹partial-isometry A›
    by (simp add: partial-isometry-square-proj that)
next
  show ‹partial-isometry A› if ‹is-Proj (A* o_CL A)›
  proof (rule partial-isometryI)
    fix h
    from that have ‹norm (A* o_CL A) ≤ 1›
      using norm-is-Proj by blast
    then have normA: ‹norm A ≤ 1› and normAadj: ‹norm (A*) ≤ 1›
      by (simp-all add: norm-AadjA abs-square-le-1)
    assume ‹h ∈ space-as-set (− kernel A)›
    also have ‹... = space-as-set (− kernel (A* o_CL A))›
     by (metis (no-types, lifting) abs-opI is-Proj-algebraic kernel-abs-op positive-cblinfun-squareI
```

that)
also have ‹... = *space-as-set* ((*A** o_{CL} *A*) $*_S$ ⊤)›
by (*simp add*: *kernel-compl-adj-range*)
finally have ‹*A** $*_V$ *A* $*_V$ *h* = *h*›
by (*metis Proj-fixes-image Proj-on-own-range that cblinfun-apply-cblinfun-compose*)
then have ‹*norm h* = *norm* (*A** $*_V$ *A* $*_V$ *h*)›
by *simp*
also have ‹... ≤ *norm* (*A* $*_V$ *h*)›
by (*smt* (*verit*) *normAadj mult-left-le-one-le norm-cblinfun norm-ge-zero*)
also have ‹... ≤ *norm h*›
by (*smt* (*verit*) *normA mult-left-le-one-le norm-cblinfun norm-ge-zero*)
ultimately show ‹*norm* (*A* $*_V$ *h*) = *norm h*›
by *simp*
qed
qed

lemma *abs-op-square*: ‹(*abs-op A*)* o_{CL} *abs-op A* = *A** o_{CL} *A*›
by (*simp add*: *abs-op-def positive-cblinfun-squareI positive-hermitianI*)

lemma *polar-decomposition-0*[*simp*]: ‹*polar-decomposition 0* = (*0* :: *'a*::*chilbert-space* ⇒$_{CL}$ *'b*::*chilbert-space*)›
proof −
have ‹*polar-decomposition* (*0* :: *'a*::*chilbert-space* ⇒$_{CL}$ *'b*::*chilbert-space*) $*_S$ ⊤ = *0* $*_S$ ⊤›
by (*simp add*: *polar-decomposition-final-space*)
then show *?thesis*
by *simp*
qed

lemma *polar-decomposition-unique*:
fixes *A* :: ‹*'a*::*chilbert-space* ⇒$_{CL}$ *'b*::*chilbert-space*›
assumes *ker*: ‹*kernel X* = *kernel A*›
assumes *comp*: ‹*X* o_{CL} *abs-op A* = *A*›
shows ‹*X* = *polar-decomposition A*›
proof −
have ‹*X* ψ = *polar-decomposition A* ψ› **if** ‹ψ ∈ *space-as-set* (*kernel A*)› **for** ψ
proof −
have ‹ψ ∈ *space-as-set* (*kernel X*)›
by (*simp add*: *ker that*)
then have ‹*X* ψ = *0*›
by (*simp add*: *kernel.rep-eq*)
moreover
have ‹ψ ∈ *space-as-set* (*kernel* (*polar-decomposition A*))›
by (*simp add*: *polar-decomposition-initial-space that*)
then have ‹*polar-decomposition A* ψ = *0*›
by (*simp add*: *kernel.rep-eq del*: *polar-decomposition-initial-space*)
ultimately show *?thesis*
by *simp*
qed
then have *1*: ‹*X* o_{CL} *Proj* (*kernel A*) = *polar-decomposition A* o_{CL} *Proj* (*kernel A*)›
by (*metis assms*(*1*) *cblinfun-compose-Proj-kernel polar-decomposition-initial-space*)

have $*$: ‹*abs-op A* $*_S$ $\top$ = $-$ *kernel A*›
by (*metis* (*mono-tags*, *opaque-lifting*) *abs-op-pos kernel-abs-op kernel-compl-adj-range orthomodular-lattice.ortho-involution positive-hermitianI*)

have ‹*X* o_{CL} *abs-op A* = *polar-decomposition A* o_{CL} *abs-op A*›
by (*simp add*: *comp polar-decomposition-correct*)
then have ‹*X* ψ = *polar-decomposition A* ψ› **if** ‹ψ $\in$ *space-as-set* (*abs-op A* $*_S$ $\top$)› **for** ψ
by (*simp add*: *cblinfun-same-on-image that*)
then have *2*: ‹*X* o_{CL} *Proj* ($-$ *kernel A*) = *polar-decomposition A* o_{CL} *Proj* ($-$ *kernel A*)›
using $*$
by (*metis* (*no-types*, *opaque-lifting*) *Proj-idempotent cblinfun-eqI lift-cblinfun-comp*(*4*) *norm-Proj-apply*)
from *1 2* **have** ‹*X* o_{CL} *Proj* ($-$ *kernel A*) + *X* o_{CL} *Proj* (*kernel A*)
= *polar-decomposition A* o_{CL} *Proj* ($-$ *kernel A*) + *polar-decomposition A* o_{CL} *Proj* (*kernel A*)›
by *simp*
then show *?thesis*
by (*simp add*: *Proj-ortho-compl flip*: *cblinfun-compose-add-right*)
qed

lemma *norm-cblinfun-mono*:
— Would logically belong in *Complex-Bounded-Operators.Complex-Bounded-Linear-Function* but uses *sqrt-op* from this theory in the proof.
fixes *A B* :: ‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*a*›
assumes ‹*A* $\geq$ *0*›
assumes ‹*A* $\leq$ *B*›
shows ‹*norm A* $\leq$ *norm B*›
proof $-$
have ‹*B* $\geq$ *0*›
using *assms* **by** *force*
have *sqrtA*: ‹(*sqrt-op A*)$*$ o_{CL} *sqrt-op A* = *A*›
by (*simp add*: ‹*A* $\geq$ *0*› *positive-hermitianI*)
have *sqrtB*: ‹(*sqrt-op B*)$*$ o_{CL} *sqrt-op B* = *B*›
by (*simp add*: ‹*B* $\geq$ *0*› *positive-hermitianI*)
have ‹*norm* (*sqrt-op A* ψ) $\leq$ *norm* (*sqrt-op B* ψ)› **for** ψ
apply (*auto intro*!: *cnorm-le*[*THEN iffD2*]
simp: *sqrtA sqrtB*
simp flip: *cinner-adj-right cblinfun-apply-cblinfun-compose*)
using *assms less-eq-cblinfun-def* **by** *auto*
then have ‹*norm* (*sqrt-op A*) $\leq$ *norm* (*sqrt-op B*)›
by (*meson dual-order.trans norm-cblinfun norm-cblinfun-bound norm-ge-zero*)
moreover have ‹*norm A* = (*norm* (*sqrt-op A*))^2›
by (*metis norm-AadjA sqrtA*)
moreover have ‹*norm B* = (*norm* (*sqrt-op B*))^2›
by (*metis norm-AadjA sqrtB*)
ultimately show ‹*norm A* $\leq$ *norm B*›
by *force*
qed

lemma *sandwich-mono*: ‹*sandwich A B* $\leq$ *sandwich A C*› **if** ‹*B* $\leq$ *C*›

by (*metis cblinfun.real.diff-right diff-ge-0-iff-ge sandwich-pos that*)

lemma *sums-pos-cblinfun*:
fixes $f :: nat \Rightarrow ('b::chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes ‹*f sums a*›
assumes ‹$\bigwedge n.\ f\ n \geq 0$›
shows $a \geq 0$
apply (*rule sums-mono-cblinfun*[**where** *f*=‹λ-. *0*› **and** *g*=*f*])
using *assms* **by** *auto*

lemma *has-sum-mono-cblinfun*:
fixes $f :: 'a \Rightarrow ('b::chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes $(f\ has\text{-}sum\ x)\ A$ **and** $(g\ has\text{-}sum\ y)\ A$
assumes ‹$\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x$›
shows $x \leq y$
using *assms has-sum-mono-neutral-cblinfun* **by** *force*

lemma *infsum-mono-cblinfun*:
fixes $f :: 'a \Rightarrow ('b::chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes *f summable-on A* **and** *g summable-on A*
assumes ‹$\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x$›
shows *infsum f A* $\leq$ *infsum g A*
by (*meson assms has-sum-infsum has-sum-mono-cblinfun*)

lemma *suminf-mono-cblinfun*:
fixes $f :: nat \Rightarrow ('b::chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes *summable f* **and** *summable g*
assumes ‹$\bigwedge x.\ f\ x \leq g\ x$›
shows *suminf f* $\leq$ *suminf g*
using *assms sums-mono-cblinfun* **by** *blast*

lemma *suminf-pos-cblinfun*:
fixes $f :: nat \Rightarrow ('b::chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes ‹*summable f*›
assumes ‹$\bigwedge x.\ f\ x \geq 0$›
shows *suminf f* $\geq$ *0*
using *assms sums-mono-cblinfun* **by** *blast*

lemma *infsum-mono-neutral-cblinfun*:
fixes $f :: 'a \Rightarrow ('b::chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes *f summable-on A* **and** *g summable-on B*
assumes ‹$\bigwedge x.\ x \in A \cap B \Longrightarrow f\ x \leq g\ x$›
assumes ‹$\bigwedge x.\ x \in A - B \Longrightarrow f\ x \leq 0$›
assumes ‹$\bigwedge x.\ x \in B - A \Longrightarrow g\ x \geq 0$›
shows *infsum f A* $\leq$ *infsum g B*
by (*smt* (*verit*, *del-insts*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *has-sum-infsum has-sum-mono-neutral-cblinfun*)

lemma *abs-op-geq*: ‹*abs-op a* ≥ *a*› **if** ‹*selfadjoint a*›
proof −
define *A P* **where** ‹*A* = *abs-op a*› **and** ‹*P* = *Proj* (*kernel* (*A* + *a*))›
from *that* **have** [*simp*]: ‹*a*∗ = *a*›
by (*simp add*: *selfadjoint-def*)
have [*simp*]: ‹*A* ≥ *0*›
by (*simp add*: *A-def*)
then have [*simp*]: ‹*A*∗ = *A*›
using *positive-hermitianI* **by** *fastforce*
have *aa-AA*: ‹*a* o_{CL} *a* = *A* o_{CL} *A*›
by (*metis A-def* ‹*A*∗ = *A*› *abs-op-square that selfadjoint-def*)
have [*simp*]: ‹*P*∗ = *P*›
by (*simp add*: *P-def adj-Proj*)
have *Aa-aA*: ‹*A* o_{CL} *a* = *a* o_{CL} *A*›
by (*metis* (*full-types*) *A-def lift-cblinfun-comp*(*2*) *abs-op-def positive-cblinfun-squareI sqrt-op-commute that selfadjoint-def*)

have ‹(*A*−*a*) *ψ* ·$_C$ (*A*+*a*) *φ* = *0*› **for** *φ ψ*
by (*simp add*: *adj-minus that* ‹*A*∗ = *A*› *aa-AA Aa-aA cblinfun-compose-add-right cblinfun-compose-minus-left*
flip: *cinner-adj-right cblinfun-apply-cblinfun-compose*)
then have ‹(*A*−*a*) *ψ* ∈ *space-as-set* (*kernel* (*A*+*a*))› **for** *ψ*
by (*metis* ‹*A*∗ = *A*› *adj-plus call-zero-iff cinner-adj-left kernel-memberI that selfadjoint-def*)
then have *P-fix*: ‹*P* o_{CL} (*A*−*a*) = (*A*−*a*)›
by (*simp add*: *P-def Proj-fixes-image cblinfun-eqI*)
then have ‹*P* o_{CL} (*A*−*a*) o_{CL} *P* = (*A*−*a*) o_{CL} *P*›
by *simp*
also have ‹... = (*P* o_{CL} (*A*−*a*))∗›
by (*simp add*: *adj-minus* ‹*A*∗ = *A*› *that* ‹*P*∗ = *P*›)
also have ‹... = (*A*−*a*)∗›
by (*simp add*: *P-fix*)
also have ‹... = *A*−*a*›
by (*simp add*: ‹*A*∗ = *A*› *that adj-minus*)
finally have *1*: ‹*P* o_{CL} (*A* − *a*) o_{CL} *P* = *A* − *a*›
by −
have *2*: ‹*P* o_{CL} (*A* + *a*) o_{CL} *P* = *0*›
by (*simp add*: *P-def cblinfun-compose-assoc*)
have ‹*A* − *a* = *P* o_{CL} (*A* − *a*) o_{CL} *P* + *P* o_{CL} (*A* + *a*) o_{CL} *P*›
by (*simp add*: *1 2*)
also have ‹... = *sandwich P* (*2* ∗$_C$ *A*)›
by (*simp add*: *sandwich-apply cblinfun-compose-minus-left cblinfun-compose-minus-right cblinfun-compose-add-left cblinfun-compose-add-right scaleC-2* ‹*P*∗ = *P*›)
also have ‹... ≥ *0*›
by (*auto intro*!: *sandwich-pos scaleC-nonneg-nonneg simp*: *less-eq-complex-def*)
finally show ‹*A* ≥ *a*›
by *auto*
qed

lemma *abs-op-geq-neq*: ‹*abs-op a* ≥ − *a*› **if** ‹*selfadjoint a*›

by (*metis abs-op-geq abs-op-uminus adj-uminus that selfadjoint-def*)

lemma *infsum-nonneg-cblinfun*:
fixes $f :: 'a \Rightarrow 'b{::}chilbert\text{-}space \Rightarrow_{CL} 'b$
assumes $\bigwedge x.\ x \in M \Longrightarrow 0 \leq f\ x$
shows $infsum\ f\ M \geq 0$
apply (*cases* ‹*f summable-on M*›)
apply (*subst infsum-0-simp*[*symmetric*])
apply (*rule infsum-mono-cblinfun*)
using *assms* **by** (*auto simp*: *infsum-not-exists*)

lemma *adj-abs-op*[*simp*]: ‹(*abs-op a*)* = *abs-op a*›
by (*simp add*: *positive-hermitianI*)

lemma *cblinfun-image-less-eqI*:
fixes $A :: \langle 'a{::}complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'b{::}complex\text{-}normed\text{-}vector\rangle$
assumes ‹$\bigwedge h.\ h \in space\text{-}as\text{-}set\ S \Longrightarrow A\ h \in space\text{-}as\text{-}set\ T$›
shows ‹$A *_S S \leq T$›
proof −
from *assms* **have** ‹A ‘ *space-as-set S* ⊆ *space-as-set T*›
by *blast*
then have ‹*closure* (*A* ‘ *space-as-set S*) ⊆ *closure* (*space-as-set T*)›
by (*rule closure-mono*)
also have ‹... = *space-as-set T*›
by *force*
finally show *?thesis*
apply (*transfer fixing*: *A*)
by (*simp add*: *cblinfun-image.rep-eq ccsubspace-leI*)
qed

lemma *abs-op-plus-orthogonal*:
assumes ‹$a* \circ_{CL} b = 0$› **and** ‹$a \circ_{CL} b* = 0$›
shows ‹*abs-op* (*a* + *b*) = *abs-op a* + *abs-op b*›
proof (*rule abs-opI*[*symmetric*])
have *ba*: ‹$b* \circ_{CL} a = 0$›
apply (*rule cblinfun-eqI*, *rule cinner-extensionality*)
apply (*simp add*: *cinner-adj-right flip*: *cinner-adj-left*)
by (*simp add*: *assms simp-a-oCL-b'*)
have *abs-ab*: ‹*abs-op a* $\circ_{CL}$ *abs-op b* = 0›
proof −
have ‹*abs-op b* $*_S$ ⊤ = − *kernel* (*abs-op b*)›
by (*simp add*: *kernel-compl-adj-range positive-hermitianI*)
also have ‹... = − *kernel b*›
by *simp*
also have ‹... = (*b**) $*_S$ ⊤›
by (*simp add*: *kernel-compl-adj-range*)
also have ‹... ≤ *kernel a*›

apply (*auto intro*!: *cblinfun-image-less-eqI kernel-memberI simp*:)
by (*simp add*: *assms flip*: *cblinfun-apply-cblinfun-compose*)
also have ‹... = *kernel* (*abs-op a*)›
by *simp*
finally show ‹*abs-op a* o_{CL} *abs-op b* = *0*›
by (*metis Proj-compose-cancelI cblinfun-compose-Proj-kernel cblinfun-compose-assoc cblinfun-compose-zero-left*)
qed
then have *abs-ba*: ‹*abs-op b* o_{CL} *abs-op a* = *0*›
by (*metis abs-op-pos adj-0 adj-cblinfun-compose positive-hermitianI*)
have ‹(*a* + *b*)* o_{CL} (*a* + *b*) = (*a**) o_{CL} *a* + (*b**) o_{CL} *b*›
by (*simp add*: *cblinfun-compose-add-left cblinfun-compose-add-right adj-plus assms ba*)
also have ‹... = (*abs-op a* + *abs-op b*)* o_{CL} (*abs-op a* + *abs-op b*)›
by (*simp add*: *cblinfun-compose-add-left cblinfun-compose-add-right adj-plus abs-ab abs-ba flip*: *abs-op-square*)
finally show ‹(*abs-op a* + *abs-op b*)* o_{CL} (*abs-op a* + *abs-op b*) = (*a* + *b*)* o_{CL} (*a* + *b*)›
by *simp*
show ‹*0* ≤ *abs-op a* + *abs-op b*›
by *simp*
qed

definition *pos-op* :: ‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*a* ⇒ '*a* $\Rightarrow_{CL}$ '*a*› **where**
‹*pos-op a* = (*abs-op a* + *a*) /$_R$ *2*›

definition *neg-op* :: ‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*a* ⇒ '*a* $\Rightarrow_{CL}$ '*a*› **where**
‹*neg-op a* = (*abs-op a* − *a*) /$_R$ *2*›

lemma *pos-op-pos*:
assumes ‹*selfadjoint a*›
shows ‹*pos-op a* ≥ *0*›
using *abs-op-geq-neq*[*OF assms*]
apply (*simp add*: *pos-op-def*)
by (*smt* (*verit*, *best*) *add-le-cancel-right more-arith-simps*(*3*) *scaleR-nonneg-nonneg zero-le-divide-iff*)

lemma *neg-op-pos*:
assumes ‹*selfadjoint a*›
shows ‹*neg-op a* ≥ *0*›
using *abs-op-geq*[*OF assms*]
by (*simp add*: *neg-op-def scaleR-nonneg-nonneg*)

lemma *pos-op-neg-op-ortho*:
assumes ‹*selfadjoint a*›
shows ‹*pos-op a* o_{CL} *neg-op a* = *0*›
apply (*auto intro*!: *simp*: *pos-op-def neg-op-def cblinfun-compose-add-left cblinfun-compose-minus-right*)
by (*metis* (*no-types*, *opaque-lifting*) *Groups.add-ac*(*2*) *abs-op-def abs-op-pos abs-op-square assms cblinfun-assoc-left*(*1*) *positive-cblinfun-squareI positive-hermitianI selfadjoint-def sqrt-op-commute*)

lemma *pos-op-plus-neg-op*: ‹*pos-op a + neg-op a = abs-op a*›
by (*simp add*: *pos-op-def neg-op-def scaleR-diff-right scaleR-add-right pth-8*)

lemma *pos-op-minus-neg-op*: ‹*pos-op a − neg-op a = a*›
by (*simp add*: *pos-op-def neg-op-def scaleR-diff-right scaleR-add-right pth-8*)

lemma *pos-op-neg-op-unique*:
assumes *bca*: ‹*b − c = a*›
assumes ‹*b ≥ 0*› **and** ‹*c ≥ 0*›
assumes *bc*: ‹*b* o_{CL} *c = 0*›
shows ‹*b = pos-op a*› **and** ‹*c = neg-op a*›
proof −
from *bc* **have** *cb*: ‹*c* o_{CL} *b = 0*›
by (*metis adj-0 adj-cblinfun-compose assms*(*2*) *assms*(*3*) *positive-hermitianI*)
from ‹*b ≥ 0*› **have** [*simp*]: ‹*b∗ = b*›
by (*simp add*: *positive-hermitianI*)
from ‹*c ≥ 0*› **have** [*simp*]: ‹*c∗ = c*›
by (*simp add*: *positive-hermitianI*)
have *bc-abs*: ‹*b + c = abs-op a*›
proof −
have ‹(*b + c*)∗ o_{CL} (*b + c*) = *b* o_{CL} *b + c* o_{CL} *c*›
by (*simp add*: *cblinfun-compose-add-left cblinfun-compose-add-right bc cb adj-plus*)
also have ‹. . . = (*b − c*)∗ o_{CL} (*b − c*)›
by (*simp add*: *cblinfun-compose-minus-left cblinfun-compose-minus-right bc cb adj-minus*)
also from *bca* **have** ‹. . . = *a∗* o_{CL} *a*›
by *blast*
finally show *?thesis*
apply (*rule abs-opI*)
by (*simp add*: ‹*b ≥ 0*› ‹*c ≥ 0*›)
qed
from *arg-cong2*[*OF bca bc-abs*, *of plus*]
arg-cong2[*OF pos-op-minus-neg-op*[*of a*] *pos-op-plus-neg-op*[*of a*], *of plus*, *symmetric*]
show ‹*b = pos-op a*›
by (*simp flip*: *scaleR-2*)
from *arg-cong2*[*OF bc-abs bca*, *of minus*]
arg-cong2[*OF pos-op-plus-neg-op*[*of a*] *pos-op-minus-neg-op*[*of a*], *of minus*, *symmetric*]
show ‹*c = neg-op a*›
by (*simp flip*: *scaleR-2*)
qed

lemma *pos-imp-selfadjoint*: ‹*a ≥ 0 ⟹ selfadjoint a*›
using *positive-hermitianI selfadjoint-def* **by** *blast*

lemma *abs-op-one-dim*: ‹*abs-op x = one-dim-iso* (*abs* (*one-dim-iso x* :: *complex*))›
by (*metis* (*mono-tags*, *lifting*) *abs-opI abs-op-scaleC of-complex-def one-cblinfun-adj one-comp-one-cblinfun one-dim-iso-is-of-complex one-dim-iso-of-one one-dim-iso-of-zero one-dim-loewner-order one-dim-scaleC-1*

zero-less-one-class.zero-le-one)

lemma *pos-selfadjoint*: ‹*selfadjoint a*› **if** ‹$a \geq 0$›
using *adj-0 comparable-hermitean selfadjoint-def that* **by** *blast*

lemma *one-dim-loewner-order-strict*: ‹$A > B \longleftrightarrow$ *one-dim-iso A* > (*one-dim-iso B* :: *complex*)›
for *A B* :: ‹'*a* ⇒$_{CL}$ '*a*::{*chilbert-space*, *one-dim*}›
by (*auto simp*: *less-cblinfun-def one-dim-loewner-order*)

lemma *one-dim-cblinfun-zero-le-one*: ‹$0 < (1 ::$ '*a*::*one-dim* ⇒$_{CL}$ '*a*)›
by (*simp add*: *one-dim-loewner-order-strict*)
lemma *one-dim-cblinfun-one-pos*: ‹$0 \leq (1 ::$ '*a*::*one-dim* ⇒$_{CL}$ '*a*)›
by (*simp add*: *one-dim-loewner-order*)

lemma *Proj-pos*[*iff*]: ‹*Proj S* ≥ 0›
apply (*rule positive-cblinfun-squareI*[**where** *B*=‹*Proj S*›])
by (*simp add*: *adj-Proj*)

lemma *abs-op-Proj*[*simp*]: ‹*abs-op* (*Proj S*) = *Proj S*›
by (*simp add*: *abs-op-id-on-pos*)

lemma *diagonal-operator-pos*:
assumes ‹$\bigwedge x.\ f\ x \geq 0$›
shows ‹*diagonal-operator f* ≥ 0›
proof (*cases* ‹*bdd-above* (*range* (λ*x*. *cmod* (*f x*)))›)
case *True*
have [*simp*]: ‹*csqrt* (*f x*) = *sqrt* (*cmod* (*f x*))› **for** *x*
by (*simp add*: *Extra-Ordered-Fields.complex-of-real-cmod assms abs-pos of-real-sqrt*)
have *bdd*: ‹*bdd-above* (*range* (λ*x*. *sqrt* (*cmod* (*f x*))))›
proof −
from *True* **obtain** *B* **where** ‹*cmod* (*f x*) ≤ *B*› **for** *x*
by (*auto simp*: *bdd-above-def*)
then show *?thesis*
by (*auto intro*!: *bdd-aboveI*[**where** *M*=‹*sqrt B*›] *simp*:)
qed
show *?thesis*
apply (*rule positive-cblinfun-squareI*[**where** *B*=‹*diagonal-operator* (λ*x*. *csqrt* (*f x*))›])
by (*simp add*: *assms diagonal-operator-adj diagonal-operator-comp bdd complex-of-real-cmod abs-pos*
flip: *of-real-mult*)
next
case *False*
then show *?thesis*
by (*simp add*: *diagonal-operator-invalid*)
qed

lemma *abs-op-diagonal-operator*:
 ‹*abs-op* (*diagonal-operator f*) = *diagonal-operator* (λ*x*. *abs* (*f x*))›
proof (*cases* ‹*bdd-above* (*range* (λ*x*. *cmod* (*f x*)))›)
 case *True*
 show *?thesis*
  apply (*rule abs-opI*[*symmetric*])
  by (*auto intro*!: *diagonal-operator-pos abs-nn simp*: *True diagonal-operator-adj diagonal-operator-comp cnj-x-x*)
next
 case *False*
 then show *?thesis*
  by (*simp add*: *diagonal-operator-invalid*)
qed

end

5 *HS2Ell2* – Representing any Hilbert space as $\ell_2(X)$

theory *HS2Ell2*
 imports *Complex-Bounded-Operators.Complex-L2 Misc-Tensor-Product-BO*
begin

unbundle *cblinfun-notation*

typedef (**overloaded**) *'a*::‹{*chilbert-space*, *not-singleton*}› *chilbert2ell2* = ‹*some-chilbert-basis* :: *'a set*›
 using *some-chilbert-basis-nonempty* **by** *auto*

definition *ell2-to-hilbert* **where** ‹*ell2-to-hilbert* = *cblinfun-extension* (*range ket*) (*Rep-chilbert2ell2* ∘ *inv ket*)›

lemma *ell2-to-hilbert-ket*: ‹*ell2-to-hilbert* ∗$_V$ *ket x* = *Rep-chilbert2ell2 x*›
proof −
 have ‹*cblinfun-extension-exists* (*range ket*) (*Rep-chilbert2ell2* ∘ *inv ket*)›
 proof (*rule cblinfun-extension-exists-ortho*[**where** *B=1*])
  fix *x y* :: *'b chilbert2ell2 ell2*
  assume *x* ∈ *range ket y* ∈ *range ket x* ≠ *y*
  then obtain *x' y'* **where** *x'-y'*: *x* = *ket x' y* = *ket y' x'* ≠ *y'*
   by *auto*
  have *is-orthogonal* (*Rep-chilbert2ell2 x'*) (*Rep-chilbert2ell2 y'*)
  by (*meson Rep-chilbert2ell2 Rep-chilbert2ell2-inject* ‹*x'* ≠ *y'*› *is-ortho-set-def is-ortho-set-some-chilbert-basis*)
  thus *is-orthogonal* ((*Rep-chilbert2ell2* ∘ *inv ket*) *x*) ((*Rep-chilbert2ell2* ∘ *inv ket*) *y*)
   using *x'-y'* **by** *auto*
 qed (*auto simp*: *Rep-chilbert2ell2 is-normal-some-chilbert-basis*)
 from *cblinfun-extension-apply*[*OF this*]
 have *cblinfun-extension* (*range ket*) (*Rep-chilbert2ell2* ∘ *inv ket*) ∗$_V$ (*ket x*) =

```
      (Rep-chilbert2ell2 ∘ inv ket) (ket x)
    by blast
  thus ?thesis
    by (simp add: ell2-to-hilbert-def)
qed

lemma norm-ell2-to-hilbert: ‹norm ell2-to-hilbert = 1›
proof (rule order.antisym)
  show ‹norm ell2-to-hilbert ≤ 1›
    unfolding ell2-to-hilbert-def
  proof (rule cblinfun-extension-exists-ortho-norm[where B=1])
    fix x y :: 'b chilbert2ell2 ell2
    assume x ∈ range ket y ∈ range ket x ≠ y
    then obtain x' y' where x'-y': x = ket x' y = ket y' x' ≠ y'
      by auto
    have is-orthogonal (Rep-chilbert2ell2 x') (Rep-chilbert2ell2 y')
    by (meson Rep-chilbert2ell2 Rep-chilbert2ell2-inject ‹x' ≠ y'› is-ortho-set-def is-ortho-set-some-chilbert-basis)
    thus is-orthogonal ((Rep-chilbert2ell2 ∘ inv ket) x) ((Rep-chilbert2ell2 ∘ inv ket) y)
      using x'-y' by auto
  qed (auto simp: Rep-chilbert2ell2 is-normal-some-chilbert-basis)
  show ‹norm ell2-to-hilbert ≥ 1›
    by (rule cblinfun-norm-geqI[where x=‹ket undefined›])
       (auto simp: ell2-to-hilbert-ket Rep-chilbert2ell2 is-normal-some-chilbert-basis)
qed

lemma unitary-ell2-to-hilbert[simp]: ‹unitary ell2-to-hilbert›
proof (rule surj-isometry-is-unitary)
  show ‹isometry (ell2-to-hilbert :: 'a chilbert2ell2 ell2 ⇒CL -)›
  proof (rule orthogonal-on-basis-is-isometry)
    show ‹ccspan (range ket) = top›
      by auto
    fix x y :: ‹'a chilbert2ell2 ell2›
    assume ‹x ∈ range ket› ‹y ∈ range ket›
    then obtain x' y' where [simp]: ‹x = ket x'› ‹y = ket y'›
      by auto
    show ‹(ell2-to-hilbert *V x) •C (ell2-to-hilbert *V y) = x •C y›
    proof (cases ‹x'=y'›)
      case True
      hence Rep-chilbert2ell2 y' •C Rep-chilbert2ell2 y' = 1
        using Rep-chilbert2ell2 cnorm-eq-1 is-normal-some-chilbert-basis by blast
      then show ?thesis using True
        by (auto simp: ell2-to-hilbert-ket)
    next
      case False
      hence is-orthogonal (Rep-chilbert2ell2 x') (Rep-chilbert2ell2 y')
      by (metis Rep-chilbert2ell2 Rep-chilbert2ell2-inject is-ortho-set-def is-ortho-set-some-chilbert-basis)
      then show ?thesis
        using False by (auto simp: ell2-to-hilbert-ket cinner-ket)
    qed
```

qed
have ‹*cblinfun-apply ell2-to-hilbert ‘ range ket* ⊇ *some-chilbert-basis*›
by (*metis Rep-chilbert2ell2-cases UNIV-I ell2-to-hilbert-ket image-eqI subsetI*)
then have ‹*ell2-to-hilbert* $*_S$ *top* ≥ *ccspan some-chilbert-basis*› (**is** ‹- ≥ ...›)
by (*smt* (*verit*, *del-insts*) *cblinfun-image-ccspan ccspan-mono ccspan-range-ket*)
also have ‹... = *top*›
by *simp*
finally show ‹*ell2-to-hilbert* $*_S$ *top* = *top*›
by (*simp add*: *top.extremum-unique*)
qed

lemma *ell2-to-hilbert-adj-ket*: ‹*ell2-to-hilbert*∗ $*_V$ ψ = *ket* (*Abs-chilbert2ell2* ψ)› **if** ‹ψ ∈ *some-chilbert-basis*›
using *ell2-to-hilbert-ket unitary-ell2-to-hilbert*
by (*metis* (*no-types*, *lifting*) *cblinfun-apply-cblinfun-compose cblinfun-id-cblinfun-apply that type-definition.Abs-inverse type-definition-chilbert2ell2 unitaryD1*)

definition ‹*cr-chilbert2ell2-ell2 x y* ⟷ *ell2-to-hilbert* $*_V$ *x* = *y*›

lemma *bi-unique-cr-chilbert2ell2-ell2*[*transfer-rule*]: ‹*bi-unique cr-chilbert2ell2-ell2*›
by (*metis* (*no-types*, *opaque-lifting*) *bi-unique-def cblinfun-apply-cblinfun-compose cr-chilbert2ell2-ell2-def id-cblinfun-apply unitaryD1 unitary-ell2-to-hilbert*)
lemma *bi-total-cr-chilbert2ell2-ell2*[*transfer-rule*]: ‹*bi-total cr-chilbert2ell2-ell2*›
by (*metis* (*no-types*, *opaque-lifting*) *bi-total-def cblinfun-apply-cblinfun-compose cr-chilbert2ell2-ell2-def id-cblinfun-apply unitaryD2 unitary-ell2-to-hilbert*)

named-theorems *c2l2l2*

lemma *c2l2l2-cinner*[*c2l2l2*]:
includes *lifting-syntax*
shows ‹(*cr-chilbert2ell2-ell2* ===> *cr-chilbert2ell2-ell2* ===> (=)) *cinner cinner*›
proof −
have ∗: ‹*ket x* $•_C$ *ket y* = (*ell2-to-hilbert* $*_V$ *ket x*) $•_C$ (*ell2-to-hilbert* $*_V$ *ket y*)› **for** *x y* :: ‹*'a chilbert2ell2*›
by (*metis Rep-chilbert2ell2 Rep-chilbert2ell2-inverse cinner-adj-right ell2-to-hilbert-adj-ket ell2-to-hilbert-ket*)
have ‹*x* $•_C$ *y* = (*ell2-to-hilbert* $*_V$ *x*) $•_C$ (*ell2-to-hilbert* $*_V$ *y*) › **for** *x y* :: ‹*'a chilbert2ell2 ell2*›
apply (*rule fun-cong*[**where** *x*=*x*])
apply (*rule bounded-antilinear-equal-ket*)
apply (*intro bounded-linear-intros*)
apply (*intro bounded-linear-intros*)
apply (*rule fun-cong*[**where** *x*=*y*])
apply (*rule bounded-clinear-equal-ket*)
apply (*intro bounded-linear-intros*)
apply (*intro bounded-linear-intros*)
by (*simp add*: ∗)
then show *?thesis*
by (*auto intro*!: *rel-funI simp*: *cr-chilbert2ell2-ell2-def*)

qed

lemma *c2l2l2-norm*[*c2l2l2*]:
includes *lifting-syntax*
shows ‹(*cr-chilbert2ell2-ell2* ===> (=)) *norm norm*›
apply (*subst norm-eq-sqrt-cinner*[*abs-def*])
apply (*subst* (2) *norm-eq-sqrt-cinner*[*abs-def*])
using *c2l2l2-cinner*[*transfer-rule*] **apply** *fail?*
by *transfer-prover*

lemma *c2l2l2-scaleC*[*c2l2l2*]:
includes *lifting-syntax*
shows ‹((=) ===> *cr-chilbert2ell2-ell2* ===> *cr-chilbert2ell2-ell2*) *scaleC scaleC*›
proof −
have ‹*ell2-to-hilbert* $*_V$ *c* $*_C$ *x* = *c* $*_C$ (*ell2-to-hilbert* $*_V$ *x*)› **for** *c* **and** *x* :: ‹'*a chilbert2ell2 ell2*›
by (*simp add*: *cblinfun.scaleC-right*)
then show *?thesis*
by (*auto intro*!: *rel-funI simp*: *cr-chilbert2ell2-ell2-def*)
qed

lemma *c2l2l2-zero*[*c2l2l2*]:
includes *lifting-syntax*
shows ‹*cr-chilbert2ell2-ell2 0 0*›
unfolding *cr-chilbert2ell2-ell2-def* **by** *simp*

lemma *c2l2l2-is-ortho-set*[*c2l2l2*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-chilbert2ell2-ell2* ===> (=)) *is-ortho-set* (*is-ortho-set* :: '*a*::{*chilbert-space*,*not-singleton*} *set* ⇒ *bool*)›
unfolding *is-ortho-set-def*
using *c2l2l2*[**where** *'a*='*a*, *transfer-rule*] **apply** *fail?*
by *transfer-prover*

lemma *c2l2l2-ccspan*[*c2l2l2*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-chilbert2ell2-ell2* ===> *rel-ccsubspace cr-chilbert2ell2-ell2*) *ccspan ccspan*›
proof (*rule rel-funI*, *rename-tac A B*)
fix *A* **and** *B* :: ‹'*a set*›
assume ‹*rel-set cr-chilbert2ell2-ell2 A B*›
then have ‹*B* = *ell2-to-hilbert* ‘ *A*›
by (*metis* (*no-types*, *lifting*) *bi-unique-cr-chilbert2ell2-ell2 bi-unique-rel-set-lemma cr-chilbert2ell2-ell2-def image-cong*)
then have ‹*space-as-set* (*ccspan B*) = *ell2-to-hilbert* ‘ *space-as-set* (*ccspan A*)›
by (*subst space-as-set-image-commute*[**where** *V*=‹*ell2-to-hilbert*∗›])
(*auto intro*: *unitaryD2 simp*: *cblinfun-image-ccspan*)

then have ‹*rel-set cr-chilbert2ell2-ell2* (*space-as-set* (*ccspan A*)) (*space-as-set* (*ccspan B*))›
by (*smt* (*verit*, *best*) *cr-chilbert2ell2-ell2-def image-iff rel-setI*)
then show ‹*rel-ccsubspace cr-chilbert2ell2-ell2* (*ccspan A*) (*ccspan B*)›
by (*simp add*: *rel-ccsubspace-def*)
qed

lemma *ell2-to-hilbert-adj-ell2-to-hilbert* [*simp*]: *ell2-to-hilbert* $*_V$ ell2-to-hilbert $*_V$ x = x*
using *unitary-ell2-to-hilbert* **unfolding** *unitary-def*
by (*metis cblinfun-apply-cblinfun-compose cblinfun-id-cblinfun-apply*)

lemma *ell2-to-hilbert-ell2-to-hilbert-adj* [*simp*]: *ell2-to-hilbert $*_V$ ell2-to-hilbert* $*_V$ x = x*
using *unitary-ell2-to-hilbert* **unfolding** *unitary-def*
by (*metis cblinfun-apply-cblinfun-compose cblinfun-id-cblinfun-apply*)

lemma *bi-total-rel-ccsubspace-cr-chilbert2ell2-ell2*[*transfer-rule*]:
‹*bi-total* (*rel-ccsubspace cr-chilbert2ell2-ell2*)›
apply (*rule bi-totalI*)
subgoal
by (*rule left-total-rel-ccsubspace*[**where** *U*=*ell2-to-hilbert* **and** *V*=‹*ell2-to-hilbert**›])
(*auto simp*: *cr-chilbert2ell2-ell2-def*)[*3*]
subgoal
by (*rule right-total-rel-ccsubspace*[**where** *U*=‹*ell2-to-hilbert**› **and** *V*=‹*ell2-to-hilbert*›])
(*auto simp*: *cr-chilbert2ell2-ell2-def*)
done

lemma *c2l2l2-top*[*c2l2l2*]:
includes *lifting-syntax*
shows ‹(*rel-ccsubspace cr-chilbert2ell2-ell2*) *top top*›
unfolding *rel-ccsubspace-def*
by (*simp add*: *UNIV-transfer bi-total-cr-chilbert2ell2-ell2*)

lemma *c2l2l2-is-onb*[*c2l2l2*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-chilbert2ell2-ell2* ===> (=)) *is-onb is-onb*›
unfolding *is-onb-def*
using *c2l2l2*[**where** *'a='a*, *transfer-rule*] **apply** *fail?*
by *transfer-prover*

unbundle *no-cblinfun-notation*

end

6 *Weak-Operator-Topology* – Weak operator topology on complex bounded operators

theory *Weak-Operator-Topology*
imports *Misc-Tensor-Product Strong-Operator-Topology Positive-Operators Wlog.Wlog*

begin

unbundle *cblinfun-notation*

definition *cweak-operator-topology*::('a::*complex-normed-vector* ⇒$_{CL}$ 'b::*complex-inner*) *topology*
  where *cweak-operator-topology* = *pullback-topology UNIV* (λa (x,y). *cinner x* (a ∗$_V$ y)) *euclidean*

lemma *cweak-operator-topology-topspace*[*simp*]:
  topspace cweak-operator-topology = *UNIV*
  unfolding *cweak-operator-topology-def topspace-pullback-topology topspace-euclidean* **by** *auto*

lemma *cweak-operator-topology-basis*:
  fixes *f*::('a::*complex-normed-vector* ⇒$_{CL}$ 'b::*complex-inner*) **and** *U*::'*i* ⇒ *complex set* **and** *x*::'*i* ⇒ 'b **and** *y*::‹'*i* ⇒ 'a›
  assumes *finite I* ⋀*i*. *i* ∈ *I* ⟹ *open* (*U i*)
  shows *openin cweak-operator-topology* {*f*. ∀ *i*∈*I*. *cinner* (*x i*) (*f* ∗$_V$ *y i*) ∈ *U i*}
proof −
  have *open* {*g*::(('b×'a)⇒*complex*). ∀ *i*∈*I*. *g* (*x i*, *y i*) ∈ *U i*}
    by (*rule product-topology-basis'*[*OF assms*])
  moreover have {*f*. ∀ *i*∈*I*. *cinner* (*x i*) (*f* ∗$_V$ *y i*) ∈ *U i*}
            = (λ*f* (*x*,*y*). *cinner x* (*f* ∗$_V$ *y*))−‘... ∩ *UNIV*
    by *auto*
  ultimately show *?thesis*
    unfolding *cweak-operator-topology-def* **by** (*subst openin-pullback-topology*) *auto*
qed

lemma *wot-weaker-than-sot*:
  continuous-map cstrong-operator-topology cweak-operator-topology (λ*f*. *f*)
proof −
  have ∗: ‹*continuous-on UNIV* ((λ*z*. *cinner x z*) ∘ (λ*f*. *f y*))› **for** *x* :: 'b **and** *y* :: 'a
    apply (*rule continuous-on-compose*)
    by (*auto intro*: *continuous-on-compose continuous-at-imp-continuous-on*)
  have ∗: ‹*continuous-map euclidean euclidean* (λ*f* (*x*::'b, *y*::'a). *x* ∙$_C$ *f y*)›
    apply *simp*
    apply (*rule continuous-on-coordinatewise-then-product*)
    using ∗ **by** *auto*
  have ∗: ‹*continuous-map* (*pullback-topology UNIV* (∗$_V$) *euclidean*) *euclidean* ((λ*f* (*x*::'b, *a*::'a). *x* ∙$_C$ *f a*) ∘ (∗$_V$))›
    apply (*rule continuous-map-pullback*)
    using ∗ **by** *simp*
  have ∗: ‹*continuous-map* (*pullback-topology UNIV* (∗$_V$) *euclidean*) *euclidean* ((λ*a* (*x*::'b, *y*::'a). *x* ∙$_C$ (*a* ∗$_V$ *y*)) ∘ (λ*f*. *f*))›
    apply (*subst asm-rl*[*of* ‹((λ*a* (*x*, *y*). *x* ∙$_C$ (*a* ∗$_V$ *y*)) ∘ (λ*f*. *f*)) = (λ*f* (*a*,*b*). *cinner a* (*f b*)) ∘ (∗$_V$)›])
    using ∗ **by** *auto*
  show *?thesis*
    unfolding *cstrong-operator-topology-def cweak-operator-topology-def*
    apply (*rule continuous-map-pullback'*)

using $*$ **by** *auto*
qed

lemma *cweak-operator-topology-weaker-than-euclidean*:
continuous-map euclidean cweak-operator-topology ($\lambda f.\ f$)
by (*metis* (*mono-tags*, *lifting*) *continuous-map-compose continuous-map-eq cstrong-operator-topology-weaker-than-eu*
wot-weaker-than-sot o-def)

lemma *cweak-operator-topology-cinner-continuous*:
continuous-map cweak-operator-topology euclidean ($\lambda f.$ *cinner x* (f $*_V$ y))
proof −
have *continuous-map cweak-operator-topology euclidean* (($\lambda f.$ f (x,y)) *o* (λa (x,y). *cinner x* (*a*
$*_V$ y)))
unfolding *cweak-operator-topology-def* **apply** (*rule continuous-map-pullback*)
using *continuous-on-product-coordinates* **by** *fastforce*
then show *?thesis* **unfolding** *comp-def* **by** *simp*
qed

lemma *continuous-on-cweak-operator-topo-iff-coordinatewise*:
continuous-map T cweak-operator-topology f
$\longleftrightarrow$ ($\forall$ *x y. continuous-map T euclidean* ($\lambda z.$ *cinner x* (f z $*_V$ y)))
proof (*intro iffI allI*)
fix x::$'c$ **and** y::$'b$
assume *continuous-map T cweak-operator-topology f*
with *continuous-map-compose*[*OF this cweak-operator-topology-cinner-continuous*]
have *continuous-map T euclidean* (($\lambda f.$ *cinner x* (f $*_V$ y)) *o* f)
by *simp*
then show *continuous-map T euclidean* ($\lambda z.$ *cinner x* (f z $*_V$ y))
unfolding *comp-def* **by** *auto*
next
assume $*$: ‹$\forall$ *x y. continuous-map T euclidean* ($\lambda z.$ x $\cdot_C$ (f z $*_V$ y))›
then have $*$: *continuous-map T euclidean* ((λa (x,y). *cinner x* (*a* $*_V$ y)) *o* f)
by (*auto simp flip*: *euclidean-product-topology*)
show *continuous-map T cweak-operator-topology f*
unfolding *cweak-operator-topology-def*
apply (*rule continuous-map-pullback'*)
by (*auto simp add*: $*$)
qed

typedef (**overloaded**) ($'a$,$'b$) *cblinfun-wot* = ‹*UNIV* :: ($'a$::*complex-normed-vector* $\Rightarrow_{CL}$ $'b$::*complex-inner*)
set› **..**
setup-lifting *type-definition-cblinfun-wot*

instantiation *cblinfun-wot* :: (*complex-normed-vector*, *complex-inner*) *complex-vector* **begin**
lift-definition *scaleC-cblinfun-wot* :: ‹*complex* $\Rightarrow$ ($'a$, $'b$) *cblinfun-wot* $\Rightarrow$ ($'a$, $'b$) *cblinfun-wot*›

is ‹*scaleC*› **.**

lift-definition *uminus-cblinfun-wot* :: ‹('a, 'b) *cblinfun-wot* ⇒ ('a, 'b) *cblinfun-wot*› **is** *uminus*
.
lift-definition *zero-cblinfun-wot* :: ‹('a, 'b) *cblinfun-wot*› **is** *0* .
lift-definition *minus-cblinfun-wot* :: ‹('a, 'b) *cblinfun-wot* ⇒ ('a, 'b) *cblinfun-wot* ⇒ ('a, 'b) *cblinfun-wot*› **is** *minus* .
lift-definition *plus-cblinfun-wot* :: ‹('a, 'b) *cblinfun-wot* ⇒ ('a, 'b) *cblinfun-wot* ⇒ ('a, 'b) *cblinfun-wot*› **is** *plus* .
lift-definition *scaleR-cblinfun-wot* :: ‹*real* ⇒ ('a, 'b) *cblinfun-wot* ⇒ ('a, 'b) *cblinfun-wot*› **is** *scaleR* .
instance
apply (*intro-classes*; *transfer*)
by (*auto simp add*: *scaleR-scaleC scaleC-add-right scaleC-add-left*)
end

instantiation *cblinfun-wot* :: (*complex-normed-vector*, *complex-inner*) *topological-space* **begin**
lift-definition *open-cblinfun-wot* :: ‹('a, 'b) *cblinfun-wot set* ⇒ *bool*› **is** ‹*openin cweak-operator-topology*›
.
instance
proof *intro-classes*
show ‹*open* (*UNIV* :: ('a,'b) *cblinfun-wot set*)›
apply *transfer*
by (*metis cweak-operator-topology-topspace openin-topspace*)
show ‹*open S* ⟹ *open T* ⟹ *open* (*S* ∩ *T*)› **for** *S T* :: ‹('a,'b) *cblinfun-wot set*›
apply *transfer* **by** *auto*
show ‹∀*S*∈*K*. *open S* ⟹ *open* (⋃ *K*)› **for** *K* :: ‹('a,'b) *cblinfun-wot set set*›
apply *transfer* **by** *auto*
qed
end

lemma *transfer-nhds-cweak-operator-topology*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*cr-cblinfun-wot* ===> *rel-filter cr-cblinfun-wot*) (*nhdsin cweak-operator-topology*) *nhds*›
unfolding *nhds-def nhdsin-def*
apply (*simp add*: *cweak-operator-topology-topspace*)
by *transfer-prover*

lemma *limitin-cweak-operator-topology*:
‹*limitin cweak-operator-topology f l F* ⟷ (∀ *a b*. ((λ*i*. *a* ∙*C* (*f i* ∗*V* *b*)) ⟶ *a* ∙*C* (*l* ∗*V* *b*)) *F*)›
by (*simp add*: *cweak-operator-topology-def limitin-pullback-topology tendsto-coordinatewise*)

lemma *filterlim-cweak-operator-topology*: ‹*filterlim f* (*nhdsin cweak-operator-topology l*) = *limitin cweak-operator-topology f l*›
by (*auto simp*: *cweak-operator-topology-topspace simp flip*: *filterlim-nhdsin-iff-limitin*)

instance *cblinfun-wot* :: (*complex-normed-vector*, *complex-inner*) *t2-space*
proof *intro-classes*
fix *a b* :: ‹('a,'b) *cblinfun-wot*›

show ‹$a \neq b \Longrightarrow \exists\, U\ V.\ open\ U \wedge open\ V \wedge a \in U \wedge b \in V \wedge U \cap V = \{\}$›
proof *transfer*
fix $a\ b :: {}'a \Rightarrow_{CL} {}'b$›
assume ‹$a \neq b$›
then obtain $\varphi\ \psi$ **where** ‹$\varphi \cdot_C (a *_V \psi) \neq \varphi \cdot_C (b *_V \psi)$›
by (*meson cblinfun-eqI cinner-extensionality*)
then obtain $U'\ V'$ **where** ‹*open* U'› ‹*open* V'› **and** *inU'*: ‹$\varphi \cdot_C (a *_V \psi) \in U'$› **and** *inV'*: ‹$\varphi \cdot_C (b *_V \psi) \in V'$› **and** ‹$U' \cap V' = \{\}$›
by (*meson hausdorff*)
define $U\ V :: {}'a \Rightarrow_{CL} {}'b)\ set$› **where** ‹$U = \{f.\ \forall i{\in}\{()\}.\ \varphi \cdot_C (f *_V \psi) \in U'\}$› **and** ‹$V = \{f.\ \forall i{\in}\{()\}.\ \varphi \cdot_C (f *_V \psi) \in V'\}$›
have ‹*openin cweak-operator-topology U*›
unfolding *U-def* **apply** (*rule cweak-operator-topology-basis*)
using ‹*open* U'› **by** *auto*
moreover have ‹*openin cweak-operator-topology V*›
unfolding *V-def* **apply** (*rule cweak-operator-topology-basis*)
using ‹*open* V'› **by** *auto*
ultimately show ‹$\exists\, U\ V.$ *openin cweak-operator-topology U* $\wedge$ *openin cweak-operator-topology V* $\wedge\ a \in U \wedge b \in V \wedge U \cap V = \{\}$›
apply (*rule-tac exI*[*of* - *U*])
apply (*rule-tac exI*[*of* - *V*])
using *inU' inV'* ‹$U' \cap V' = \{\}$› **by** (*auto simp*: *U-def V-def*)
qed
qed

lemma *Domainp-cr-cblinfun-wot*[*simp*]: ‹*Domainp cr-cblinfun-wot* = (λ-. *True*)›
by (*metis* (*no-types*, *opaque-lifting*) *DomainPI cblinfun-wot.left-total left-totalE*)

lemma *Rangep-cr-cblinfun-wot*[*simp*]: ‹*Rangep cr-cblinfun-wot* = (λ-. *True*)›
by (*meson RangePI cr-cblinfun-wot-def*)

lemma *transfer-euclidean-cweak-operator-topology*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-topology cr-cblinfun-wot*) *cweak-operator-topology euclidean*›
proof (*unfold rel-topology-def*, *intro conjI allI impI*)
show ‹(*rel-set cr-cblinfun-wot* ===> (=)) (*openin cweak-operator-topology*) (*openin euclidean*)›
apply (*auto simp*: *rel-topology-def cr-cblinfun-wot-def rel-set-def intro*!: *rel-funI*)
apply *transfer*
apply *auto*
apply (*meson openin-subopen subsetI*)
apply *transfer*
apply *auto*
by (*meson openin-subopen subsetI*)
next
fix $U :: {}'a \Rightarrow_{CL} {}'b)\ set$›
assume ‹*openin cweak-operator-topology U*›
show ‹*Domainp* (*rel-set cr-cblinfun-wot*) *U*›
by (*simp add*: *Domainp-set*)
next

fix U :: ‹('a, 'b) *cblinfun-wot set*›
assume ‹*openin euclidean U*›
show ‹*Rangep* (*rel-set cr-cblinfun-wot*) *U*›
by (*simp add*: *Rangep-set*)
qed

lemma *openin-cweak-operator-topology*: ‹*openin cweak-operator-topology U* ⟷ (∃ *V*. *open V* ∧ *U* = (λ*a* (*x*,*y*). *cinner x* (*a* *$_V$ *y*)) −‘ *V*)›
by (*simp add*: *cweak-operator-topology-def openin-pullback-topology*)

lemma *cweak-operator-topology-plus-cont*: ‹*LIM* (*x*,*y*) *nhdsin cweak-operator-topology a* ×$_F$ *nhdsin cweak-operator-topology b*.
x + *y* :> *nhdsin cweak-operator-topology* (*a* + *b*)›
proof −
show *?thesis*
unfolding *cweak-operator-topology-def*
apply (*rule-tac pullback-topology-bi-cont*[**where** *f'*=*plus*])
by (*auto simp*: *case-prod-unfold tendsto-add-Pair cinner-add-right cblinfun.add-left*)
qed

instance *cblinfun-wot* :: (*complex-normed-vector*, *complex-inner*) *topological-group-add*
proof *intro-classes*
show ‹((λ*x*. *fst x* + *snd x*) ⟶ *a* + *b*) (*nhds a* ×$_F$ *nhds b*)› **for** *a b* :: ‹('a,'b) *cblinfun-wot*›
apply *transfer*
using *cweak-operator-topology-plus-cont*
by (*auto simp*: *case-prod-unfold*)

have ∗: ‹*continuous-map cweak-operator-topology cweak-operator-topology uminus*›
apply (*subst continuous-on-cweak-operator-topo-iff-coordinatewise*)
apply (*rewrite at* ‹(λ*z*. *x* ·$_C$ (− *z* *$_V$ *y*))› **in** ‹∀*x y*. ⌑› *to* ‹(λ*z*. − *x* ·$_C$ (*z* *$_V$ *y*))› *DEAD.rel-mono-strong*)
by (*auto simp*: *cweak-operator-topology-cinner-continuous cblinfun.minus-left cblinfun.minus-right*)
show ‹(*uminus* ⟶ − *a*) (*nhds a*)› **for** *a* :: ‹('a,'b) *cblinfun-wot*›
apply (*subst tendsto-at-iff-tendsto-nhds*[*symmetric*])
apply (*subst isCont-def*[*symmetric*])
apply (*rule continuous-on-interior*[**where** *S*=*UNIV*])
apply (*subst continuous-map-iff-continuous2*[*symmetric*])
apply *transfer*
using ∗ **by** *auto*
qed

lemma *continuous-map-left-comp-wot*:
‹*continuous-map cweak-operator-topology cweak-operator-topology* (λ*a*::'*a*::*complex-normed-vector* ⇒$_{CL}$ -. *b* *$_{CL}$ *a*)›
for *b* :: ‹'*b*::*chilbert-space* ⇒$_{CL}$ '*c*::*complex-inner*›
proof −
have ∗∗: ‹((λ*f*::'*b* × '*a* ⇒ *complex*. *f* (*b*∗ *$_V$ *x*, *y*)) −‘ *B* ∩ *UNIV*)
= (*Pi$_E$ UNIV* (λ*z*. *if z* = (*b*∗ *$_V$ *x*, *y*) *then B else UNIV*))›
for *x* :: '*c* **and** *y* :: '*a* **and** *B* :: ‹*complex set*›

```
    by (auto simp: PiE-def Pi-def)
  have *: ‹continuous-on UNIV (λf::'b × 'a ⇒ complex. f (b* *V x, y))› for x y
    unfolding continuous-on-open-vimage[OF open-UNIV]
    apply (intro allI impI)
    apply (subst **)
    apply (rule open-PiE)
    by auto
  have *: ‹continuous-on UNIV (λ(f::'b × 'a ⇒ complex) (x, y). f (b* *V x, y))›
    apply (rule continuous-on-coordinatewise-then-product)
    using * by auto
  show ?thesis
    unfolding cweak-operator-topology-def
    apply (rule continuous-map-pullback')
     apply (subst asm-rl[of ‹((λ(a::'a⇒CL'c) (x, y). x •C (a *V y)) ∘ (oCL) b) = (λf (x,y). f
(b* *V x,y)) ∘ (λa (x, y). x •C (a *V y))›])
      apply (auto intro!: ext simp: cinner-adj-left)[1]
     apply (rule continuous-map-pullback)
    using * by auto
qed

lemma continuous-map-scaleC-wot: ‹continuous-map cweak-operator-topology cweak-operator-topology

                    (scaleC c :: ('a::complex-normed-vector ⇒CL 'b::chilbert-space) ⇒ -)›
  apply (subst asm-rl[of ‹scaleC c = (oCL) (c *C id-cblinfun)›])
   apply auto[1]
  by (rule continuous-map-left-comp-wot)

lemma continuous-scaleC-wot: ‹continuous-on X (scaleC c :: (-::complex-normed-vector,-::chilbert-space)
cblinfun-wot ⇒ -)›
  apply (rule continuous-on-subset[rotated, where s=UNIV], simp)
  apply (subst continuous-map-iff-continuous2[symmetric])
  apply transfer
  by (rule continuous-map-scaleC-wot)

lemma wot-closure-is-csubspace[simp]:
  fixes A::('a::complex-normed-vector, 'b::chilbert-space) cblinfun-wot set
  assumes ‹csubspace A›
  shows ‹csubspace (closure A)›
proof (rule complex-vector.subspaceI)
  include lattice-syntax
  show 0: ‹0 ∈ closure A›
    by (simp add: assms closure-def complex-vector.subspace-0)
  show ‹x + y ∈ closure A› if ‹x ∈ closure A› ‹y ∈ closure A› for x y
  proof −
    define FF where ‹FF = ((nhds x ⊓ principal A) ×F (nhds y ⊓ principal A))›
    have nt: ‹FF ≠ bot›
      by (simp add: prod-filter-eq-bot that(1) that(2) FF-def flip: closure-nhds-principal)
    have ‹∀F x in FF. fst x ∈ A›
      unfolding FF-def
```

by (*smt* (*verit*, *ccfv-SIG*) *eventually-prod-filter fst-conv inf-sup-ord*(*2*) *le-principal*)
moreover have ‹$\forall_F$ *x in FF*. *snd x* ∈ *A*›
unfolding *FF-def*
by (*smt* (*verit*, *ccfv-SIG*) *eventually-prod-filter snd-conv inf-sup-ord*(*2*) *le-principal*)
ultimately have *FF-plus*: ‹$\forall_F$ *x in FF*. *fst x* + *snd x* ∈ *A*›
by (*smt* (*verit*, *best*) *assms complex-vector.subspace-add eventually-elim2*)

have ‹(*fst* ⟶ *x*) ((*nhds x* ⊓ *principal A*) ×$_F$ (*nhds y* ⊓ *principal A*))›
apply (*simp add*: *filterlim-def*)
using *filtermap-fst-prod-filter*
using *le-inf-iff* **by** *blast*
moreover have ‹(*snd* ⟶ *y*) ((*nhds x* ⊓ *principal A*) ×$_F$ (*nhds y* ⊓ *principal A*))›
apply (*simp add*: *filterlim-def*)
using *filtermap-snd-prod-filter*
using *le-inf-iff* **by** *blast*
ultimately have ‹(*id* ⟶ (*x*,*y*)) *FF*›
by (*simp add*: *filterlim-def nhds-prod prod-filter-mono FF-def*)

moreover note *tendsto-add-Pair*[*of x y*]
ultimately have ‹(((λ*x*. *fst x* + *snd x*) *o id*) ⟶ (λ*x*. *fst x* + *snd x*) (*x*,*y*)) *FF*›
unfolding *filterlim-def nhds-prod*
by (*smt* (*verit*, *best*) *filterlim-compose filterlim-def filterlim-filtermap fst-conv snd-conv tendsto-compose-filtermap*)

then have ‹((λ*x*. *fst x* + *snd x*) ⟶ (*x*+*y*)) *FF*›
by *simp*
then show ‹*x* + *y* ∈ *closure A*›
using *nt FF-plus* **by** (*rule limit-in-closure*)
qed
show ‹*c* *$_C$ *x* ∈ *closure A*› **if** ‹*x* ∈ *closure A*› **for** *x c*
using *that*
using *image-closure-subset*[**where** *S*=*A* **and** *T*=‹*closure A*› **and** *f*=‹*scaleC c*›, *OF continuous-scaleC-wot*]
apply *auto*
by (*metis 0 assms closure-subset csubspace-scaleC-invariant imageI in-mono scaleC-eq-0-iff*)
qed

lemma [*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-cblinfun-wot* ===> (=)) *csubspace csubspace*›
unfolding *complex-vector.subspace-def*
by *transfer-prover*

lemma [*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-cblinfun-wot* ===> (=)) (*closedin cweak-operator-topology*) *closed*›
apply (*simp add*: *closed-def*[*abs-def*] *closedin-def*[*abs-def*] *cweak-operator-topology-topspace Compl-eq-Diff-UNIV*)

by *transfer-prover*

lemma [*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-cblinfun-wot* ===> *rel-set cr-cblinfun-wot*) (*Abstract-Topology.closure-of cweak-operator-topology*) *closure*›
apply (*subst closure-of-hull*[**where** *X*=*cweak-operator-topology*, *unfolded cweak-operator-topology-topspace*, *simplified*, *abs-def*])
apply (*subst closure-hull*[*abs-def*])
unfolding *hull-def*
by *transfer-prover*

lemma *wot-closure-is-csubspace'*[*simp*]:
fixes *A*::('*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*chilbert-space*) *set*
assumes ‹*csubspace A*›
shows ‹*csubspace* (*cweak-operator-topology closure-of A*)›
using *wot-closure-is-csubspace*[*of* ‹*Abs-cblinfun-wot* ' *A*›] *assms*
apply (*transfer fixing*: *A*)
by *simp*

lemma *has-sum-closed-cweak-operator-topology*:
fixes *A* :: ‹('*b*::*complex-normed-vector* ⇒$_{CL}$ '*c*::*complex-inner*) *set*›
assumes *aA*: ‹⋀*i*. *a i* ∈ *A*›
assumes *closed*: ‹*closedin cweak-operator-topology A*›
assumes *subspace*: ‹*csubspace A*›
assumes *has-sum*: ‹⋀φ ψ. ((λ*i*. φ ∙$_C$ (*a i* *$_V$ ψ)) *has-sum* φ ∙$_C$ (*b* *$_V$ ψ)) *I*›
shows ‹*b* ∈ *A*›
proof −
have *1*: ‹*range* (*sum a*) ⊆ *A*›
proof −
have ‹*sum a X* ∈ *A*› **for** *X*
apply (*induction X rule*:*infinite-finite-induct*)
by (*auto simp add*: *subspace complex-vector.subspace-0 aA complex-vector.subspace-add*)
then show *?thesis*
by *auto*
qed

from *has-sum*
have ‹((λ*F*. ∑*i*∈*F*. φ ∙$_C$ (*a i* *$_V$ ψ)) ⟶ φ ∙$_C$ (*b* *$_V$ ψ)) (*finite-subsets-at-top I*)› **for** ψ φ
using *has-sum-def* **by** *blast*
then have ‹*limitin cweak-operator-topology* (λ*F*. ∑*i*∈*F*. *a i*) *b* (*finite-subsets-at-top I*)›
by (*auto simp add*: *limitin-cweak-operator-topology cblinfun.sum-left cinner-sum-right*)
then show ‹*b* ∈ *A*›
using *1 closed* **apply** (*rule limitin-closedin*)
by *simp*
qed

lemma *limitin-adj-wot*:
assumes ‹*limitin cweak-operator-topology f l F*›

shows ‹*limitin cweak-operator-topology* (λ*i*. (*f i*)*) (*l**) *F*›
proof −
from *assms*
have ‹((λ*i*. *a* ·$_C$ (*f i* *$_V$ *b*)) ⟶ *a* ·$_C$ (*l* *$_V$ *b*)) *F*› **for** *a b*
by (*simp add*: *limitin-cweak-operator-topology*)
then have ‹((λ*i*. *cnj* (*a* ·$_C$ (*f i* *$_V$ *b*))) ⟶ *cnj* (*a* ·$_C$ (*l* *$_V$ *b*))) *F*› **for** *a b*
using *tendsto-cnj* **by** *blast*
then have ‹((λ*i*. *cnj* (((*f i*)* *$_V$ *a*) ·$_C$ *b*)) ⟶ *cnj* ((*l** *$_V$ *a*) ·$_C$ *b*)) *F*› **for** *a b*
by (*simp add*: *cinner-adj-left*)
then have ‹((λ*i*. *b* ·$_C$ ((*f i*)* *$_V$ *a*)) ⟶ *b* ·$_C$ (*l** *$_V$ *a*)) *F*› **for** *a b*
by *simp*
then show *?thesis*
by (*simp add*: *limitin-cweak-operator-topology*)
qed

lemma *hausdorff-cweak-operator-topology*[*simp*]: ‹*Hausdorff-space cweak-operator-topology*›
proof (*rule hausdorffI*)
fix *A B* :: ‹'*a* ⇒$_{CL}$ '*b*› **assume** ‹*A* ≠ *B*›
then obtain *y* **where** ‹*A* *$_V$ *y* ≠ *B* *$_V$ *y*›
by (*meson cblinfun-eqI*)
then obtain *x* **where** ‹*x* ·$_C$ (*A* *$_V$ *y*) ≠ *x* ·$_C$ (*B* *$_V$ *y*)›
using *cinner-extensionality* **by** *blast*
then obtain *U' V'* **where** ‹*open U'*› ‹*open V'*› **and** *inside*: ‹*x* ·$_C$ (*A* *$_V$ *y*) ∈ *U'*› ‹*x* ·$_C$ (*B* *$_V$ *y*) ∈ *V'*› **and** *disj*: ‹*U'* ∩ *V'* = {}›
by (*meson separation-t2*)
define *U'' V''* **where** ‹*U''* = *Pi$_E$ UNIV* (λ*i*. *if i*=(*x*,*y*) *then U' else UNIV*)› **and** ‹*V''* = *Pi$_E$ UNIV* (λ*i*. *if i*=(*x*,*y*) *then V' else UNIV*)›
from ‹*open U'*› ‹*open V'*›
have ‹*open U''*› **and** ‹*open V''*›
by (*auto simp*: *U''-def V''-def open-fun-def intro*!: *product-topology-basis*)
define *U V* :: ‹('*a* ⇒$_{CL}$ '*b*) *set*› **where** ‹*U* = (λ*A* (*x*, *y*). *x* ·$_C$ (*A* *$_V$ *y*)) −‘ *U''*› **and** ‹*V* = (λ*a* (*x*, *y*). *x* ·$_C$ (*a* *$_V$ *y*)) −‘ *V''*›
have *openin*: ‹*openin cweak-operator-topology U*› ‹*openin cweak-operator-topology V*›
using *U-def V-def* ‹*open U''*› ‹*open V''*› *openin-cweak-operator-topology* **by** *blast*+
have ‹*A* ∈ *U*› ‹*B* ∈ *V*›
using *inside* **by** (*auto simp*: *U-def V-def U''-def V''-def*)
have ‹*U* ∩ *V* = {}›
using *disj* **apply** (*auto simp*: *U-def V-def U''-def V''-def PiE-def Pi-iff*)
by (*metis disjoint-iff*)
show ‹∃ *U V*. *openin cweak-operator-topology U* ∧ *openin cweak-operator-topology V* ∧ *A* ∈ *U* ∧ *B* ∈ *V* ∧ *U* ∩ *V* = {}›
using ‹*A* ∈ *U*› ‹*B* ∈ *V*› ‹*U* ∩ *V* = {}› *openin* **by** *blast*
qed

lemma *hermitian-limit-hermitian-wot*:
assumes ‹*F* ≠ *bot*›
assumes *herm*: ‹⋀*i*. (*a i*)* = *a i*›
assumes *lim*: ‹*limitin cweak-operator-topology a A F*›
shows ‹*A** = *A*›

using - - ‹$F \neq bot$› *hausdorff-cweak-operator-topology*
apply (*rule limitin-Hausdorff-unique*)
using *lim* **apply** (*rule limitin-adj-wot*)
unfolding *herm* **by** (*fact lim*)

lemma *wot-weaker-than-sot-openin*:
‹*openin cweak-operator-topology x* $\Longrightarrow$ *openin cstrong-operator-topology x*›
using *wot-weaker-than-sot* **unfolding** *continuous-map-def* **by** *auto*

lemma *wot-weaker-than-sot-limitin*: ‹*limitin cweak-operator-topology a A F*› **if** ‹*limitin cstrong-operator-topology a A F*›
using *that* **unfolding** *filterlim-cweak-operator-topology*[*symmetric*] *filterlim-cstrong-operator-topology*[*symmetric*]
apply (*rule filterlim-mono*)
apply (*rule nhdsin-mono*)
by (*auto simp*: *wot-weaker-than-sot-openin*)

lemma *hermitian-limit-hermitian-sot*:
assumes ‹$F \neq bot$›
assumes ‹$\bigwedge i.\ (a\ i)* = a\ i$›
assumes ‹*limitin cstrong-operator-topology a A F*›
shows ‹$A* = A$›
using *assms*(*1*,*2*) **apply** (*rule hermitian-limit-hermitian-wot*[**where** *a*=*a* **and** *F*=*F*])
using *assms*(*3*) **by** (*rule wot-weaker-than-sot-limitin*)

lemma *hermitian-sum-hermitian-sot*:
assumes *herm*: ‹$\bigwedge i.\ (a\ i)* = a\ i$›
assumes *sum*: ‹*has-sum-in cstrong-operator-topology a X A*›
shows ‹$A* = A$›
proof −
from *herm* **have** *herm-sum*: ‹$(sum\ a\ F)* = sum\ a\ F$› **for** *F*
by (*simp add*: *sum-adj*)
show *?thesis*
using - *herm-sum sum*[*unfolded has-sum-in-def*]
apply (*rule hermitian-limit-hermitian-sot*)
by *simp*
qed

lemma *wot-is-norm-topology-findim*[*simp*]:
‹(*cweak-operator-topology* :: ('*a*::{*cfinite-dim*,*chilbert-space*} $\Rightarrow_{CL}$ '*b*::{*cfinite-dim*,*chilbert-space*}) *topology*) = *euclidean*›
proof −
have ‹*continuous-map euclidean cweak-operator-topology id*›
by (*simp add*: *id-def cweak-operator-topology-weaker-than-euclidean*)
moreover have ‹*continuous-map cweak-operator-topology euclidean* (*id* :: '*a* $\Rightarrow_{CL}$ '*b* $\Rightarrow$ -)›
proof (*rule continuous-map-iff-preserves-convergence*)
fix *l* **and** *F* :: ‹('*a* $\Rightarrow_{CL}$ '*b*) *filter*›
assume *lim-wot*: ‹*limitin cweak-operator-topology id l F*›
obtain *A* :: ‹'*a set*› **where** ‹*is-onb A*›

using *is-onb-some-chilbert-basis* **by** *blast*
then have *idA*: ‹*id-cblinfun* = $(\sum x{\in}A.$ *selfbutter* $x)$›
using *butterflies-sum-id-finite* **by** *blast*
obtain B :: ‹'*b set*› **where** ‹*is-onb B*›
using *is-onb-some-chilbert-basis* **by** *blast*
then have *idB*: ‹*id-cblinfun* = $(\sum x{\in}B.$ *selfbutter* $x)$›
using *butterflies-sum-id-finite* **by** *blast*
from *lim-wot* **have** ‹$((\lambda x.\ b \cdot_C (x *_V a)) \longrightarrow b \cdot_C (l *_V a))$ *F*› **for** *a b*
by (*simp add*: *limitin-cweak-operator-topology*)
then have ‹$((\lambda x.\ (b \cdot_C (x *_V a)) *_C$ *butterfly b a*$) \longrightarrow (b \cdot_C (l *_V a)) *_C$ *butterfly b a*$)$ *F*› **for** *a b*
by (*simp add*: *tendsto-scaleC*)
then have ‹$((\lambda x.$ *selfbutter b* o_{CL} *x* o_{CL} *selfbutter a*$) \longrightarrow ($*selfbutter b* o_{CL} *l* o_{CL} *selfbutter a*$))$ *F*› **for** *a b*
by (*simp add*: *cblinfun-comp-butterfly*)
then have ‹$((\lambda x.\ \sum b{\in}B.$ *selfbutter b* o_{CL} *x* o_{CL} *selfbutter a*$) \longrightarrow (\sum b{\in}B.$ *selfbutter b* o_{CL} *l* o_{CL} *selfbutter a*$))$ *F*› **for** *a*
by (*rule tendsto-sum*)
then have ‹$((\lambda x.$ *x* o_{CL} *selfbutter a*$) \longrightarrow ($*l* o_{CL} *selfbutter a*$))$ *F*› **for** *a*
by (*simp add*: *flip*: *cblinfun-compose-sum-left idB*)
then have ‹$((\lambda x.\ \sum a{\in}A.$ *x* o_{CL} *selfbutter a*$) \longrightarrow (\sum a{\in}A.$ *l* o_{CL} *selfbutter a*$))$ *F*›
by (*rule tendsto-sum*)
then have ‹(*id* $\longrightarrow$ *l*) *F*›
by (*simp add*: *flip*: *cblinfun-compose-sum-right idA id-def*)
then show ‹*limitin euclidean id* (*id l*) *F*›
by *simp*
qed
ultimately show *?thesis*
by (*auto simp*: *topology-finer-continuous-id*[*symmetric*] *simp flip*: *openin-inject*)
qed

lemma *sot-is-norm-topology-fin-dim*[*simp*]:

‹(*cstrong-operator-topology* :: ('*a*::{*cfinite-dim*,*chilbert-space*} $\Rightarrow_{CL}$ '*b*::{*cfinite-dim*,*chilbert-space*}) *topology*) = *euclidean*›
proof −
have *1*: ‹*continuous-map euclidean cstrong-operator-topology* (*id* :: '*a*$\Rightarrow_{CL}$'*b* ⇒ -)›
by (*simp add*: *id-def cstrong-operator-topology-weaker-than-euclidean*)
have ‹*continuous-map cstrong-operator-topology cweak-operator-topology* (*id* :: '*a*$\Rightarrow_{CL}$'*b* ⇒ -)›
by (*metis eq-id-iff wot-weaker-than-sot*)
then have *2*: ‹*continuous-map cstrong-operator-topology euclidean* (*id* :: '*a*$\Rightarrow_{CL}$'*b* ⇒ -)›
by (*simp only*: *wot-is-norm-topology-findim*)
from *1 2*
show *?thesis*
by (*auto simp*: *topology-finer-continuous-id*[*symmetric*] *simp flip*: *openin-inject*)
qed

lemma *regular-space-wot*: ‹*regular-space cweak-operator-topology*›

proof −
have ‹*regular-space* (*product-topology* (λ*i*::′*b*×′*a*. *euclidean* :: *complex topology*) *UNIV*)›
by (*simp add*: *regular-space-product-topology*)
then have ‹*regular-space* (*euclidean* :: (′*b*×′*a* ⇒ *complex*) *topology*)›
using *euclidean-product-topology* **by** *metis*
then show *?thesis*
unfolding *cweak-operator-topology-def*
by (*rule-tac regular-space-pullback*)
qed

instance *cblinfun-wot* :: (*complex-normed-vector*, *complex-inner*) *t3-space*
apply *intro-classes*
apply *transfer*
using *regular-space-wot*
by (*auto simp add*: *regular-space-def disjnt-def*)

instantiation *cblinfun-wot* :: (*chilbert-space*, *chilbert-space*) *order* **begin**
lift-definition *less-eq-cblinfun-wot* :: ‹(′*a*, ′*b*) *cblinfun-wot* ⇒ (′*a*, ′*b*) *cblinfun-wot* ⇒ *bool*› **is** *less-eq*.
lift-definition *less-cblinfun-wot* :: ‹(′*a*, ′*b*) *cblinfun-wot* ⇒ (′*a*, ′*b*) *cblinfun-wot* ⇒ *bool*› **is** *less*.
instance
apply (*intro-classes*; *transfer′*)
by *auto*
end

instance *cblinfun-wot* :: (*chilbert-space*,*chilbert-space*) *ordered-comm-monoid-add*
proof
fix *a b c* :: ‹(′*a*,′*b*) *cblinfun-wot*›
assume ‹*a* ≤ *b*›
then show ‹*c* + *a* ≤ *c* + *b*›
apply *transfer′*
by *simp*
qed

lemma *limitin-wot-add*:
assumes ‹*limitin cweak-operator-topology f a F*›
assumes ‹*limitin cweak-operator-topology g b F*›
shows ‹*limitin cweak-operator-topology* (λ*x*. *f x* + *g x*) (*a* + *b*) *F*›
proof −
have ‹*LIM x F*. (*f x*, *g x*) :> *nhdsin cweak-operator-topology a* ×$_F$ *nhdsin cweak-operator-topology b*›
apply (*rule filterlim-Pair*)
using *assms* **by** (*simp-all add*: *filterlim-cweak-operator-topology*)
then have ‹*LIM x F*. *case* (*f x*, *g x*) *of* (*x*, *y*) ⇒ *x* + *y* :> *nhdsin cweak-operator-topology* (*a* + *b*)›
apply (*rule filterlim-compose*[*rotated*])

```
    by (rule cweak-operator-topology-plus-cont)
  then show ?thesis
    by (simp add: filterlim-cweak-operator-topology)
qed

lemma monotone-convergence-wot:
  — [1], Proposition 43.1 (i), (ii), but translated to filters.
  fixes f :: ‹'b ⇒ ('a ⇒_CL 'a::chilbert-space)›
  assumes bounded: ‹∀_F x in F. f x ≤ B›
  assumes increasing: ‹increasing-filter (filtermap f F)›
  shows ‹∃ L. limitin cweak-operator-topology f L F›
proof −
  wlog nontrivial: ‹F ≠ ⊥›
    using negation by (auto intro!: limitin-trivial)
  wlog pos: ‹∀_F x in F. f x ≥ 0› generalizing f B keeping bounded increasing nontrivial
  proof −
    from increasing
    have ‹∀_F y in F. ∃ L. limitin cweak-operator-topology f L F›
      unfolding increasing-filter-def eventually-filtermap
    proof (rule eventually-mono)
      fix x0
      assume f-lower: ‹∀_F x in F. f x0 ≤ f x›
      define g where ‹g x = f x − f x0› for x
      from bounded
      have bounded-g: ‹∀_F x in F. g x ≤ B − f x0›
        apply (rule eventually-mono)
        by (simp add: g-def)
      from f-lower
      have pos-g: ‹∀_F x in F. g x ≥ 0›
        apply (rule eventually-mono)
        by (simp add: g-def)
      from increasing
      have increasing-g: ‹increasing-filter (filtermap g F)›
        unfolding increasing-filter-def eventually-filtermap
        apply (rule eventually-mono)
        apply (erule eventually-mono)
        by (simp add: g-def[abs-def])
      obtain L where ‹limitin cweak-operator-topology g L F›
        apply atomize-elim
        using pos-g bounded-g increasing-g nontrivial by (rule hypothesis)
      then have ‹limitin cweak-operator-topology (λx. g x + f x0) (L + f x0) F›
        apply (rule limitin-wot-add)
        by simp
      then have ‹limitin cweak-operator-topology f (L + f x0) F›
        by (auto intro!: simp: g-def[abs-def])
      then show ‹∃ L. limitin cweak-operator-topology f L F›
        by auto
    qed
    then show ?thesis
```

```
    by (simp add: nontrivial eventually-const)
  qed

  define surround where ‹surround ψ a = ψ ·C (a *V ψ)› for ψ :: 'a and a
  have mono-surround: ‹mono (surround ψ)› for ψ
    by (auto intro!: monoI simp: surround-def less-eq-cblinfun-def)
  obtain l' where  tendsto-l': ‹((λx. surround ψ (f x)) ⟶ l' ψ) F› for ψ
  proof (atomize-elim, intro choice allI)
    fix ψ :: 'a
    from bounded
    have surround-bound: ‹∀F x in F. surround ψ (f x) ≤ surround ψ B›
      unfolding surround-def
      apply (rule eventually-mono)
      by (simp add: less-eq-cblinfun-def)
    moreover have ‹increasing-filter (filtermap (λx. surround ψ (f x)) F)›
      using increasing-filtermap[OF increasing mono-surround]
      by (simp add: filtermap-filtermap)
    ultimately obtain l' where ‹((λx. surround ψ (f x)) ⟶ l') F›
      apply atomize-elim
      by (auto intro!: monotone-convergence-complex increasing mono-surround
          simp: eventually-filtermap)
    then show ‹∃l'. ((λx. surround ψ (f x)) ⟶ l') F›
      by auto
  qed
  define l where ‹l φ ψ = (l' (φ+ψ) - l' (φ-ψ) - i * l' (φ + i *C ψ) + i * l' (φ - i *C ψ))
/ 4› for φ ψ :: 'a
  have polar: ‹φ ·C a ψ = (surround (φ+ψ) a - surround (φ-ψ) a - i * surround (φ + i *C
ψ) a + i * surround (φ - i *C ψ) a) / 4› for a :: ‹'a ⇒CL 'a› and φ ψ
    by (simp add: surround-def cblinfun.add-right cinner-add cblinfun.diff-right
        cinner-diff cblinfun.scaleC-right ring-distribs)
  have tendsto-l: ‹((λx. φ ·C f x ψ) ⟶ l φ ψ) F› for φ ψ
    by (auto intro!: tendsto-divide tendsto-add tendsto-diff tendsto-l' simp: l-def polar)
  have l-bound: ‹norm (l φ ψ) ≤ norm B * norm φ * norm ψ› for φ ψ
  proof -
    from bounded pos
    have ‹∀F x in F. norm (φ ·C f x ψ) ≤ norm B * norm φ * norm ψ› for φ ψ
    proof (rule eventually-elim2)
      fix x
      assume ‹f x ≤ B› and ‹0 ≤ f x›
      have ‹cmod (φ ·C (f x *V ψ)) ≤ norm φ * norm (f x *V ψ)›
        using complex-inner-class.Cauchy-Schwarz-ineq2 by blast
      also have ‹... ≤ norm φ * (norm (f x) * norm ψ)›
        by (simp add: mult-left-mono norm-cblinfun)
      also from ‹f x ≤ B› ‹0 ≤ f x›
      have ‹... ≤ norm φ * (norm B * norm ψ)›
        by (auto intro!: mult-left-mono mult-right-mono norm-cblinfun-mono simp: )
      also have ‹... = norm B * norm φ * norm ψ›
        by simp
      finally show ‹norm (φ ·C f x ψ) ≤ norm B * norm φ * norm ψ›
```

```
      by −
    qed
    moreover from tendsto-l
    have ‹((λx. norm (φ ·C f x ψ)) ⟶ norm (l φ ψ)) F› for φ ψ
      using tendsto-norm by blast
    ultimately show ?thesis
      using nontrivial tendsto-upperbound by blast
  qed
  have ‹bounded-sesquilinear l›
  proof (rule bounded-sesquilinear.intro)
    fix φ φ' ψ ψ' and r :: complex
    from tendsto-l have ‹((λx. φ ·C f x ψ + φ ·C f x ψ') ⟶ l φ ψ + l φ ψ') F›
      by (simp add: tendsto-add)
    moreover from tendsto-l have ‹((λx. φ ·C f x ψ + φ ·C f x ψ') ⟶ l φ (ψ + ψ')) F›
      by (simp flip: cinner-add-right cblinfun.add-right)
    ultimately show ‹l φ (ψ + ψ') = l φ ψ + l φ ψ'›
      by (rule tendsto-unique[OF nontrivial, rotated])
    from tendsto-l have ‹((λx. φ ·C f x ψ + φ' ·C f x ψ) ⟶ l φ ψ + l φ' ψ) F›
      by (simp add: tendsto-add)
    moreover from tendsto-l have ‹((λx. φ ·C f x ψ + φ' ·C f x ψ) ⟶ l (φ + φ') ψ) F›
      by (simp flip: cinner-add-left cblinfun.add-left)
    ultimately show ‹l (φ + φ') ψ = l φ ψ + l φ' ψ›
      by (rule tendsto-unique[OF nontrivial, rotated])
    from tendsto-l have ‹((λx. r *C (φ ·C f x ψ)) ⟶ r *C l φ ψ) F›
      using tendsto-scaleC by blast
    moreover from tendsto-l have ‹((λx. r *C (φ ·C f x ψ)) ⟶ l φ (r *C ψ)) F›
      by (simp flip: cinner-scaleC-right cblinfun.scaleC-right)
    ultimately show ‹l φ (r *C ψ) = r *C l φ ψ›
      by (rule tendsto-unique[OF nontrivial, rotated])
    from tendsto-l have ‹((λx. cnj r *C (φ ·C f x ψ)) ⟶ cnj r *C l φ ψ) F›
      using tendsto-scaleC by blast
    moreover from tendsto-l have ‹((λx. cnj r *C (φ ·C f x ψ)) ⟶ l (r *C φ) ψ) F›
      by (simp flip: cinner-scaleC-left cblinfun.scaleC-left)
    ultimately show ‹l (r *C φ) ψ = cnj r *C l φ ψ›
      by (rule tendsto-unique[OF nontrivial, rotated])
    show ‹∃K. ∀a b. cmod (l a b) ≤ norm a * norm b * K›
      using l-bound by (auto intro!: exI[of - ‹norm B›] simp: mult-ac)
  qed
  define L where ‹L = (the-riesz-rep-sesqui l)*›
  then have ‹φ ·C L ψ = l φ ψ› for φ ψ
    by (auto intro!: ‹bounded-sesquilinear l› the-riesz-rep-sesqui-apply simp: cinner-adj-right)
  with tendsto-l have ‹((λx. φ ·C f x ψ) ⟶ φ ·C L ψ) F› for φ ψ
    by simp
  then have ‹limitin cweak-operator-topology f L F›
    by (simp add: limitin-cweak-operator-topology)
  then show ?thesis
    by auto
qed
```

lemma *summable-wot-boundedI*:
fixes $f :: \langle 'b \Rightarrow ('a \Rightarrow_{CL} 'a{::}chilbert\text{-}space)\rangle$
assumes *bounded*: $\langle \bigwedge F.\ finite\ F \Longrightarrow F \subseteq X \Longrightarrow sum\ f\ F \leq B\rangle$
assumes *pos*: $\langle \bigwedge x.\ x \in X \Longrightarrow f\ x \geq 0\rangle$
shows $\langle summable\text{-}on\text{-}in\ cweak\text{-}operator\text{-}topology\ f\ X\rangle$
proof −
from *pos* **have** *incr*: $\langle increasing\text{-}filter\ (filtermap\ (sum\ f)\ (finite\text{-}subsets\text{-}at\text{-}top\ X))\rangle$
by (*auto intro*!: *increasing-filtermap*[**where** $X{=}\langle\{F.\ finite\ F \wedge F \subseteq X\}\rangle$] *mono-onI sum-mono2*)
show *?thesis*
apply (*simp add*: *summable-on-in-def has-sum-in-def*)
by (*safe intro*!: *bounded incr monotone-convergence-wot*[**where** $B{=}B$] *eventually-finite-subsets-at-top-weakI*)
qed

lemma *summable-wot-boundedI'*:
fixes $f :: \langle 'b \Rightarrow ('a{::}chilbert\text{-}space,\ 'a)\ cblinfun\text{-}wot\rangle$
assumes *bounded*: $\langle \bigwedge F.\ finite\ F \Longrightarrow F \subseteq X \Longrightarrow sum\ f\ F \leq B\rangle$
assumes *pos*: $\langle \bigwedge x.\ x \in X \Longrightarrow f\ x \geq 0\rangle$
shows $\langle f\ summable\text{-}on\ X\rangle$
apply (*subst summable-on-euclidean-eq*[*symmetric*])
using *assms*
apply (*transfer' fixing*: X)
apply (*rule summable-wot-boundedI*)
by *auto*

lemma *has-sum-mono-neutral-wot*:
fixes $f :: 'a \Rightarrow ('b{::}chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes $\langle has\text{-}sum\text{-}in\ cweak\text{-}operator\text{-}topology\ f\ A\ a\rangle$ **and** *has-sum-in cweak-operator-topology* $g\ B\ b$
assumes $\langle \bigwedge x.\ x \in A \cap B \Longrightarrow f\ x \leq g\ x\rangle$
assumes $\langle \bigwedge x.\ x \in A - B \Longrightarrow f\ x \leq 0\rangle$
assumes $\langle \bigwedge x.\ x \in B - A \Longrightarrow g\ x \geq 0\rangle$
shows $a \leq b$
proof −
have ψ-*eq*: $\langle \psi \cdot_C a\ \psi \leq \psi \cdot_C b\ \psi\rangle$ **for** ψ
proof −
from *assms*(*1*)
have *sumA*: $\langle((\lambda x.\ \psi \cdot_C f\ x\ \psi)\ has\text{-}sum\ \psi \cdot_C a\ \psi)\ A\rangle$
by (*simp add*: *has-sum-in-def has-sum-def limitin-cweak-operator-topology cblinfun.sum-left cinner-sum-right*)
from *assms*(*2*)
have *sumB*: $\langle((\lambda x.\ \psi \cdot_C g\ x\ \psi)\ has\text{-}sum\ \psi \cdot_C b\ \psi)\ B\rangle$
by (*simp add*: *has-sum-in-def has-sum-def limitin-cweak-operator-topology cblinfun.sum-left cinner-sum-right*)
from *sumA sumB*
show *?thesis*
apply (*rule has-sum-mono-neutral-complex*)
using *assms*(*3*−*5*)

by (*auto simp*: *less-eq-cblinfun-def*)
qed
then show ‹$a \leq b$›
by (*simp add*: *less-eq-cblinfun-def*)
qed

lemma *has-sum-mono-wot*:
fixes $f :: 'a \Rightarrow ('b::chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes *has-sum-in cweak-operator-topology f A x* **and** *has-sum-in cweak-operator-topology g A y*
assumes ‹$\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x$›
shows $x \leq y$
using *assms has-sum-mono-neutral-wot* **by** *force*

lemma *infsum-mono-neutral-wot*:
fixes $f :: 'a \Rightarrow ('b::chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes *summable-on-in cweak-operator-topology f A* **and** *summable-on-in cweak-operator-topology g B*
assumes ‹$\bigwedge x.\ x \in A \cap B \Longrightarrow f\ x \leq g\ x$›
assumes ‹$\bigwedge x.\ x \in A - B \Longrightarrow f\ x \leq 0$›
assumes ‹$\bigwedge x.\ x \in B - A \Longrightarrow g\ x \geq 0$›
shows *infsum-in cweak-operator-topology f A* $\leq$ *infsum-in cweak-operator-topology g B*
using *assms*
by (*metis* (*mono-tags*, *lifting*) *has-sum-in-infsum-in has-sum-mono-neutral-wot hausdorff-cweak-operator-topology*)

lemma *has-sum-on-wot-transfer*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(((=) ===> *cr-cblinfun-wot*) ===> (=) ===> *cr-cblinfun-wot* ===> ($\longleftrightarrow$)) (*has-sum-in cweak-operator-topology*) *HAS-SUM*›
unfolding *has-sum-euclidean-iff*[*abs-def*, *symmetric*] *has-sum-in-def*[*abs-def*]
by *transfer-prover*

lemma *summable-on-wot-transfer*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(((=) ===> *cr-cblinfun-wot*) ===> (=) ===> ($\longleftrightarrow$)) (*summable-on-in cweak-operator-topology*) (*summable-on*)›
apply (*auto intro*!: *simp*: *summable-on-def*[*abs-def*] *summable-on-in-def*[*abs-def*])
by *transfer-prover*

lemma *Abs-cblinfun-wot-transfer*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹((=) ===> *cr-cblinfun-wot*) *id Abs-cblinfun-wot*›
by (*auto intro*!: *rel-funI simp*: *cr-cblinfun-wot-def Abs-cblinfun-wot-inverse*)

lemma *infsum-mono-neutral-wot'*:

fixes $f :: 'a \Rightarrow ('b::chilbert\text{-}space, 'b)$ *cblinfun-wot*
assumes f *summable-on* A **and** g *summable-on* B
assumes $\langle\bigwedge x.\ x \in A \cap B \Longrightarrow f\ x \leq g\ x\rangle$
assumes $\langle\bigwedge x.\ x \in A - B \Longrightarrow f\ x \leq 0\rangle$
assumes $\langle\bigwedge x.\ x \in B - A \Longrightarrow g\ x \geq 0\rangle$
shows *infsum* $f\ A \leq$ *infsum* $g\ B$
unfolding *infsum-euclidean-eq*[*symmetric*]
using *assms*
apply (*transfer' fixing*: $A\ B$)
apply (*rule infsum-mono-neutral-wot*)
by *auto*

lemma *infsum-nonneg-wot'*:
fixes $f :: 'a \Rightarrow ('c::chilbert\text{-}space,'c)$ *cblinfun-wot*
assumes $\bigwedge x.\ x \in M \Longrightarrow 0 \leq f\ x$
shows *infsum* $f\ M \geq 0$
proof (*cases* ‹f *summable-on* M›)
case *True*
show *?thesis*
apply (*subst infsum-0*[*symmetric*, *OF refl*])
apply (*rule infsum-mono-neutral-wot'*[**where** $A{=}M$ **and** $B{=}M$])
using *assms True* **by** *auto*
next
case *False*
then have ‹*infsum* $f\ M = 0$›
using *infsum-not-exists* **by** *blast*
then show *?thesis*
by *simp*
qed

lemma *summable-on-Sigma-wotI*:
fixes $f ::$ ‹$'a \times 'b \Rightarrow ('c::chilbert\text{-}space,'c)$ *cblinfun-wot*›
assumes ‹$\bigwedge x\ y.\ x \in A \Longrightarrow y \in B\ x \Longrightarrow f\ (x,y) \geq 0$›
assumes *summableA*: ‹$(\lambda x.\ \sum_\infty y{\in}B\ x.\ f\ (x,y))$ *summable-on* A›
assumes *summableB*: ‹$\bigwedge x.\ x{\in}A \Longrightarrow (\lambda y.\ f\ (x,\ y))$ *summable-on* $(B\ x)$›
shows ‹f *summable-on Sigma* $A\ B$›
proof (*rule summable-wot-boundedI'*)
show ‹$f\ x \geq 0$› **if** ‹$x \in$ *Sigma* $A\ B$› **for** x
using *assms that* **by** *blast*
show ‹*sum* $f\ F \leq (\sum_\infty x{\in}A.\ \sum_\infty y{\in}B\ x.\ f\ (x,y))$› **if** ‹*finite* F› **and** ‹$F \subseteq$ *Sigma* $A\ B$› **for** F
proof −
define *FA* **where** ‹$FA =$ *fst* ‘ F›
define *FB* **where** ‹$FB\ x = \{y.\ (x,y){\in}F\}$› **for** x
have *F-FAB*: ‹$F =$ *Sigma FA FB*›
by (*auto simp*: *FA-def FB-def image-iff Bex-def*)
have [*simp*]: ‹*finite FA*› ‹*finite* $(FB\ x)$› **for** x
using ‹*finite* F› **by** (*auto intro*!: *finite-inverse-image injI simp*: *FA-def FB-def*)
have *FA-A*: ‹$FA \subseteq A$›

```
    using FA-def that(2) by auto
  have FB-B: ‹FB x ⊆ B x› if ‹x ∈ A› for x
    using FB-def ‹F ⊆ Sigma A B› by auto
  have ‹sum f F = (∑ x∈FA. ∑ y∈FB x. f (x,y))›
    apply (subst sum.Sigma)
    by (auto simp: F-FAB)
  also have ‹... = (∑ x∈FA. ∑∞ y∈FB x. f (x,y))›
    by fastforce
  also have ‹... ≤ (∑ x∈FA. ∑∞ y∈B x. f (x,y))›
    apply (rule sum-mono)
    apply (rule infsum-mono-neutral-wot')
    using FA-A FB-B assms by auto
  also have ‹... = (∑∞ x∈FA. ∑∞ y∈B x. f (x,y))›
    by fastforce
  also have ‹... ≤ (∑∞ x∈A. ∑∞ y∈B x. f (x,y))›
    apply (rule infsum-mono-neutral-wot')
    using FA-A assms by (auto intro!: infsum-nonneg-wot')
  finally show ‹sum f F ≤ (∑∞ x∈A. ∑∞ y∈B x. f (x,y))›
    by −
 qed
qed

lift-definition compose-wot :: ‹('b::complex-inner,'c::complex-inner) cblinfun-wot ⇒ ('a::complex-normed-vector,'b)
cblinfun-wot ⇒ ('a,'c) cblinfun-wot› is
  cblinfun-compose.

lift-definition adj-wot :: ‹('a::chilbert-space, 'b::complex-inner) cblinfun-wot ⇒ ('b, 'a) cblin-
fun-wot› is adj.

lemma infsum-wot-is-Sup:
  fixes f :: ‹'b ⇒ ('a ⇒CL 'a::chilbert-space)›
  assumes summable: ‹summable-on-in cweak-operator-topology f X›
    — See also summable-wot-boundedI for proving this.
  assumes pos: ‹⋀x. x ∈ X ⟹ f x ≥ 0›
  defines ‹S ≡ infsum-in cweak-operator-topology f X›
  shows ‹is-Sup ((λF. ∑ x∈F. f x) ` {F. finite F ∧ F ⊆ X}) S›
proof (rule is-SupI)
  have has-sum: ‹has-sum-in cweak-operator-topology f X S›
    unfolding S-def
    apply (rule has-sum-in-infsum-in)
    using assms by auto
  show ‹s ≤ S› if ‹s ∈ ((λF. ∑ x∈F. f x) ` {F. finite F ∧ F ⊆ X})› for s
  proof −
    from that obtain F where [simp]: ‹finite F› and ‹F ⊆ X› and s-def: ‹s = (∑ x∈F. f x)›
      by auto
    show ?thesis
    proof (rule has-sum-mono-neutral-wot)
      show ‹has-sum-in cweak-operator-topology f F s›
        by (auto intro!: has-sum-in-finite simp: s-def)
```

show ‹*has-sum-in cweak-operator-topology f X S*›
by (*fact has-sum*)
show ‹$f\ x \leq f\ x$› **for** x
by *simp*
show ‹$f\ x \leq 0$› **if** ‹$x \in F - X$› **for** x
using ‹$F \subseteq X$› *that* **by** *auto*
show ‹$f\ x \geq 0$› **if** ‹$x \in X - F$› **for** x
using *that pos* **by** *auto*
qed
qed
show ‹$S \leq y$›
if *y-bound*: ‹$\bigwedge x.\ x \in ((\lambda F.\ \sum x{\in}F.\ f\ x)\ `\ \{F.\ \mathit{finite}\ F \wedge F \subseteq X\}) \Longrightarrow x \leq y$› **for** y
proof (*rule cblinfun-leI*, *rename-tac* ψ)
fix $\psi :: 'a$
define g **where** ‹$g\ x = \psi\ \cdot_C$ *Rep-cblinfun-wot* $x\ \psi$› **for** x
from *has-sum* **have** *lim*: ‹$((\lambda i.\ \psi \cdot_C ((\sum x{\in}i.\ f\ x) *_V \psi)) \longrightarrow \psi \cdot_C (S *_V \psi))$ (*finite-subsets-at-top X*)›
by (*simp add*: *has-sum-in-def limitin-cweak-operator-topology*)
have *bound*: ‹$\psi \cdot_C (\sum x{\in}F.\ f\ x)\ \psi \leq \psi \cdot_C y\ \psi$› **if** ‹*finite F*› ‹$F \subseteq X$› **for** F
using *y-bound less-eq-cblinfun-def that*(*1*) *that*(*2*) **by** *fastforce*
show ‹$\psi \cdot_C (S *_V \psi) \leq \psi \cdot_C y\ \psi$›
using *finite-subsets-at-top-neq-bot tendsto-const lim* **apply** (*rule tendsto-le-complex*)
using *bound* **by** (*auto intro*!: *eventually-finite-subsets-at-top-weakI*)
qed
qed

lemma *has-sum-in-cweak-operator-topology-pointwise*:
‹*has-sum-in cweak-operator-topology f X s* $\longleftrightarrow$ $(\forall \psi\ \varphi.\ ((\lambda x.\ \psi \cdot_C f\ x\ \varphi)\ \mathit{has\text{-}sum}\ \psi \cdot_C s\ \varphi)\ X)$›
by (*simp add*: *has-sum-in-def has-sum-def limitin-cweak-operator-topology cblinfun.sum-left cinner-sum-right*)

lemma *summable-wot-bdd-above*:
fixes f :: ‹$'b \Rightarrow ('a \Rightarrow_{CL} 'a{::}\mathit{chilbert\text{-}space})$›
assumes *summable*: ‹*summable-on-in cweak-operator-topology f X*›
— See also *summable-wot-boundedI* for proving this.
assumes *pos*: ‹$\bigwedge x.\ x \in X \Longrightarrow f\ x \geq 0$›
shows ‹*bdd-above* (*sum f* ‘ $\{F.\ \mathit{finite}\ F \wedge F \subseteq X\}$)›
using *infsum-wot-is-Sup*[*OF assms*]
by (*auto intro*!: *simp*: *is-Sup-def bdd-above-def*)

lemma *summable-on-in-cweak-operator-topology-pointwise*:
assumes ‹*summable-on-in cweak-operator-topology f X*›
shows ‹$(\lambda x.\ a \cdot_C f\ x\ b)$ *summable-on* X›
using *assms*
by (*auto simp*: *summable-on-in-def summable-on-def has-sum-in-cweak-operator-topology-pointwise*)

lemma *infsum-in-cweak-operator-topology-pointwise*:
assumes ‹*summable-on-in cweak-operator-topology f X*›

shows ‹a $\cdot_C$ (*infsum-in cweak-operator-topology* f X) b = ($\sum_\infty x{\in}X.$ a $\cdot_C$ f x b)›
by (*metis* (*mono-tags*, *lifting*) *assms has-sum-in-cweak-operator-topology-pointwise has-sum-in-infsum-in hausdorff-cweak-operator-topology infsumI*)

instance *cblinfun-wot* :: (*complex-normed-vector*, *complex-inner*) *topological-ab-group-add*
by *intro-classes*

lemma *has-sum-in-wot-compose-left*:
fixes f :: ‹'c ⇒ 'a::*complex-normed-vector* ⇒$_{CL}$ 'b::*chilbert-space*›
assumes ‹*has-sum-in cweak-operator-topology* f X s›
shows ‹*has-sum-in cweak-operator-topology* ($\lambda x.$ a o_{CL} f x) X (a o_{CL} s)›
proof (*rule has-sum-in-cweak-operator-topology-pointwise*[*THEN iffD2*], *intro allI*, *rename-tac* g h)
fix g h
from *assms* **have** ‹(($\lambda x.$ ($a*$) g $\cdot_C$ f x h) *has-sum* ($a*$) g $\cdot_C$ s h) X›
by (*metis has-sum-in-cweak-operator-topology-pointwise*)
then show ‹(($\lambda x.$ g $\cdot_C$ (a o_{CL} f x) h) *has-sum* g $\cdot_C$ (a o_{CL} s) h) X›
by (*metis* (*no-types*, *lifting*) *cblinfun-apply-cblinfun-compose cinner-adj-left has-sum-cong*)
qed

lemma *has-sum-in-wot-compose-right*:
fixes f :: ‹'c ⇒ 'a::*complex-normed-vector* ⇒$_{CL}$ 'b::*complex-inner*›
assumes ‹*has-sum-in cweak-operator-topology* f X s›
shows ‹*has-sum-in cweak-operator-topology* ($\lambda x.$ f x o_{CL} a) X (s o_{CL} a)›
proof (*rule has-sum-in-cweak-operator-topology-pointwise*[*THEN iffD2*], *intro allI*, *rename-tac* g h)
fix g h
from *assms* **have** ‹(($\lambda x.$ g $\cdot_C$ f x (a h)) *has-sum* g $\cdot_C$ s (a h)) X›
by (*metis has-sum-in-cweak-operator-topology-pointwise*)
then show ‹(($\lambda x.$ g $\cdot_C$ (f x o_{CL} a) h) *has-sum* g $\cdot_C$ (s o_{CL} a) h) X›
by *simp*
qed

lemma *summable-on-in-wot-compose-left*:
fixes f :: ‹'c ⇒ 'a::*complex-normed-vector* ⇒$_{CL}$ 'b::*chilbert-space*›
assumes ‹*summable-on-in cweak-operator-topology* f X›
shows ‹*summable-on-in cweak-operator-topology* ($\lambda x.$ a o_{CL} f x) X›
using *has-sum-in-wot-compose-left assms*
by (*fastforce simp*: *summable-on-in-def*)

lemma *summable-on-in-wot-compose-right*:
assumes ‹*summable-on-in cweak-operator-topology* f X›
shows ‹*summable-on-in cweak-operator-topology* ($\lambda x.$ f x o_{CL} a) X›
using *has-sum-in-wot-compose-right assms*
by (*fastforce simp*: *summable-on-in-def*)

lemma *infsum-in-wot-compose-left*:
fixes $f :: \langle 'c \Rightarrow 'a{::}complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'b{::}chilbert\text{-}space\rangle$
assumes ‹*summable-on-in cweak-operator-topology f X*›
shows ‹*infsum-in cweak-operator-topology* ($\lambda x.\ a\ o_{CL}\ f\ x$) *X* = *a* o_{CL} (*infsum-in cweak-operator-topology f X*)›
by (*metis* (*mono-tags*, *lifting*) *assms has-sum-in-infsum-in has-sum-in-unique hausdorff-cweak-operator-topology has-sum-in-wot-compose-left summable-on-in-wot-compose-left*)

lemma *infsum-in-wot-compose-right*:
fixes $f :: \langle 'c \Rightarrow 'a{::}complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'b{::}complex\text{-}inner\rangle$
assumes ‹*summable-on-in cweak-operator-topology f X*›
shows ‹*infsum-in cweak-operator-topology* ($\lambda x.\ f\ x\ o_{CL}\ a$) *X* = (*infsum-in cweak-operator-topology f X*) o_{CL} *a*›
by (*metis* (*mono-tags*, *lifting*) *assms has-sum-in-infsum-in has-sum-in-unique hausdorff-cweak-operator-topology has-sum-in-wot-compose-right summable-on-in-wot-compose-right*)

lemma *infsum-wot-boundedI*:
fixes $f :: \langle 'b \Rightarrow ('a \Rightarrow_{CL} 'a{::}chilbert\text{-}space)\rangle$
assumes *bounded*: ‹$\bigwedge F.$ *finite F* $\Longrightarrow$ *F* $\subseteq$ *X* $\Longrightarrow$ *sum f F* $\leq$ *B*›
assumes *pos*: ‹$\bigwedge x.$ *x* $\in$ *X* $\Longrightarrow$ *f x* $\geq$ *0*›
shows ‹*infsum-in cweak-operator-topology f X* $\leq$ *B*›
proof (*rule less-eq-cblinfunI*)
fix *h*
have *summ*: ‹*summable-on-in cweak-operator-topology f X*›
using *assms* **by** (*rule summable-wot-boundedI*)
then have ‹*h* $\cdot_C$ (*infsum-in cweak-operator-topology f X* $*_V$ *h*) = ($\sum_\infty x{\in}X.$ *h* $\cdot_C$ (*f x* $*_V$ *h*))›
by (*rule infsum-in-cweak-operator-topology-pointwise*)
also have ‹... $\leq$ *h* $\cdot_C$ *B h*›
proof (*rule less-eq-complexI*)
from *summ* **have** *summ'*: ‹($\lambda x.$ *h* $\cdot_C$ (*f x* $*_V$ *h*)) *summable-on X*›
by (*auto intro*!: *summable-on-in-cweak-operator-topology-pointwise*)
have $*$: ‹($\sum x{\in}F.$ *h* $\cdot_C$ (*f x* $*_V$ *h*)) $\leq$ *h* $\cdot_C$ *B h*› **if** ‹*finite F*› **and** ‹*F* $\subseteq$ *X*› **for** *F*
using *that bounded*
by (*simp add*: *less-eq-cblinfun-def flip*: *cinner-sum-right cblinfun.sum-left*)
show ‹*Im* ($\sum_\infty x{\in}X.$ *h* $\cdot_C$ (*f x* $*_V$ *h*)) = *Im* (*h* $\cdot_C$ (*B* $*_V$ *h*))›
proof −
from $*$[*of* ‹{}›] **have** ‹*h* $\cdot_C$ *B h* $\geq$ *0*›
by *simp*
then have ‹*Im* (*h* $\cdot_C$ *B h*) = *0*›
using *comp-Im-same zero-complex.sel*(*2*) **by** *presburger*
moreover then have ‹*Im* (*h* $\cdot_C$ (*f x* $*_V$ *h*)) = *0*› **if** ‹*x* $\in$ *X*› **for** *x*
using $*$[*of* ‹{*x*}›] *that*
by (*simp add*: *less-eq-complex-def*)
ultimately show ‹*Im* ($\sum_\infty x{\in}X.$ *h* $\cdot_C$ (*f x* $*_V$ *h*)) = *Im* (*h* $\cdot_C$ (*B* $*_V$ *h*))›
by (*auto intro*!: *infsum-0 simp*: *summ' simp flip*: *infsum-Im*)
qed
show ‹*Re* ($\sum_\infty x{\in}X.$ *h* $\cdot_C$ (*f x* $*_V$ *h*)) $\leq$ *Re* (*h* $\cdot_C$ (*B* $*_V$ *h*))›
apply (*auto intro*!: *summable-on-Re infsum-le-finite-sums simp*: *summ' simp flip*: *infsum-Re*)

using *summ'*
by (*metis* ∗ *Re-sum less-eq-complex-def*)
qed
finally show ‹*h* ⋅$_C$ (*infsum-in cweak-operator-topology f X* ∗$_V$ *h*) ≤ *h* ⋅$_C$ (*B* ∗$_V$ *h*)›
by −
qed

end

7 *Misc-Tensor-Product-TTS* − Miscelleanous results missing from `Complex_Bounded_Operators`

Here specifically results obtained from lifting existing results using the types to sets mechanism ([6]).

theory *Misc-Tensor-Product-TTS*
imports
Complex-Bounded-Operators.Complex-Bounded-Linear-Function
Misc-Tensor-Product
Misc-Tensor-Product-BO
With-Type.With-Type
begin

unbundle *lattice-syntax*
unbundle *cblinfun-notation*

7.1 Retrieving axioms

attribute-setup *axiom* = ‹*Scan.lift Parse.name-position* >> (*fn name-pos* => *Thm.rule-attribute*
[]
(*fn context* => *fn* - =>
let val thy = *Context.theory-of context*
val (*full-name*, -) = *Name-Space.check context* (*Theory.axiom-table thy*) *name-pos*
in Thm.axiom thy full-name end))›
‹*Retrieve an axiom by name. E.g., write* @{*thm* [*source*] [[*axiom HOL.refl*]]}.›

7.2 Auxiliary lemmas

named-theorems *unoverload-def*

locale *local-typedef* = **fixes** *S* ::*'b set* **and** *s*::*'s itself*
assumes *Ex-type-definition-S*: ∃(*Rep*::*'s* ⇒ *'b*) (*Abs*::*'b* ⇒ *'s*). *type-definition Rep Abs S*
begin
definition *Rep* = *fst* (*SOME* (*Rep*::*'s* ⇒ *'b*, *Abs*). *type-definition Rep Abs S*)
definition *Abs* = *snd* (*SOME* (*Rep*::*'s* ⇒ *'b*, *Abs*). *type-definition Rep Abs S*)
lemma *type-definition-S*: *type-definition Rep Abs S*

unfolding *Abs-def Rep-def split-beta′*
by (*rule someI-ex*) (*use Ex-type-definition-S* **in** *auto*)
lemma *rep-in-S*[*simp*]: *Rep x* ∈ *S*
and *rep-inverse*[*simp*]: *Abs* (*Rep x*) = *x*
and *Abs-inverse*[*simp*]: *y* ∈ *S* ⟹ *Rep* (*Abs y*) = *y*
using *type-definition-S*
unfolding *type-definition-def* **by** *auto*
definition *cr-S* **where** *cr-S* ≡ λ*s b*. *s* = *Rep b*
lemma *Domainp-cr-S*[*transfer-domain-rule*]: *Domainp cr-S* = (λ*x*. *x* ∈ *S*)
by (*metis Abs-inverse Domainp.simps cr-S-def rep-in-S*)
lemma *right-total-cr-S*[*transfer-rule*]: *right-total cr-S*
by (*rule typedef-right-total*[*OF type-definition-S cr-S-def*])
lemma *bi-unique-cr-S*[*transfer-rule*]: *bi-unique cr-S*
by (*rule typedef-bi-unique*[*OF type-definition-S cr-S-def*])
lemma *left-unique-cr-S*[*transfer-rule*]: *left-unique cr-S*
by (*rule typedef-left-unique*[*OF type-definition-S cr-S-def*])
lemma *right-unique-cr-S*[*transfer-rule*]: *right-unique cr-S*
by (*rule typedef-right-unique*[*OF type-definition-S cr-S-def*])
lemma *cr-S-Rep*[*intro*, *simp*]: *cr-S* (*Rep a*) *a* **by** (*simp add*: *cr-S-def*)
lemma *cr-S-Abs*[*intro*, *simp*]: *a*∈*S* ⟹ *cr-S a* (*Abs a*) **by** (*simp add*: *cr-S-def*)
lemma *UNIV-transfer*[*transfer-rule*]: ‹*rel-set cr-S S UNIV*›
using *Domainp-cr-S right-total-cr-S right-total-UNIV-transfer* **by** *fastforce*
end

lemma *complete-space-as-set*[*simp*]: ‹*complete* (*space-as-set V*)› **for** *V* :: ‹-::*cbanach ccsubspace*›
by (*simp add*: *complete-eq-closed*)

definition ‹*transfer-ball-range A P* ⟷ (∀*f*. *range f* ⊆ *A* ⟶ *P f*)›

lemma *transfer-ball-range-parametric′*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*, *simp*]: ‹*right-unique T*› ‹*bi-total T*› ‹*bi-unique U*›
shows ‹(*rel-set U* ===> ((*T* ===> *U*) ===> (⟶)) ===> (⟶)) *transfer-ball-range transfer-ball-range*›
proof (*intro rel-funI impI*, *rename-tac A B P Q*)
fix *A B P Q*
assume [*transfer-rule*]: ‹*rel-set U A B*›
assume *TUPQ*[*transfer-rule*]: ‹((*T* ===> *U*) ===> (⟶)) *P Q*›
assume ‹*transfer-ball-range A P*›
then have *Pf*: ‹*P f*› **if** ‹*range f* ⊆ *A*› **for** *f*
unfolding *transfer-ball-range-def* **using** *that* **by** *auto*
have ‹*Q g*› **if** ‹*range g* ⊆ *B*› **for** *g*
proof −
from *that* ‹*rel-set U A B*›
have ‹*Rangep* (*T* ===> *U*) *g*›
apply (*auto simp add*: *conversep-rel-fun Domainp-pred-fun-eq simp flip*: *Domainp-conversep*)
apply (*simp add*: *Domainp-conversep*)
by (*metis Rangep.simps range-subsetD rel-setD2*)
then obtain *f* **where** *TUfg*[*transfer-rule*]: ‹(*T* ===> *U*) *f g*›

by *blast*
from *that* **have** ‹*range f* ⊆ *A*›
by *transfer*
then have ‹*P f*›
by (*simp add*: *Pf*)
then show ‹*Q g*›
by (*metis TUfg TUPQ rel-funD*)
qed
then show ‹*transfer-ball-range B Q*›
by (*simp add*: *transfer-ball-range-def*)
qed

lemma *transfer-ball-range-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*, *simp*]: ‹*bi-unique T*› ‹*bi-total T*› ‹*bi-unique U*›
shows ‹(*rel-set U* ===> ((*T* ===> *U*) ===> (⟷)) ===> (⟷)) *transfer-ball-range transfer-ball-range*›
proof −
have ‹(*rel-set U* ===> ((*T* ===> *U*) ===> (⟶)) ===> (⟶)) *transfer-ball-range transfer-ball-range*›
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *bi-unique-alt-def transfer-ball-range-parametric′* **by** *blast*
then have *1*: ‹(*rel-set U* ===> ((*T* ===> *U*) ===> (⟷)) ===> (⟶)) *transfer-ball-range transfer-ball-range*›
apply (*rule rev-mp*)
apply (*intro rel-fun-mono′*)
by *auto*

have ‹(*rel-set* (U^{-1-1}) ===> ((T^{-1-1} ===> U^{-1-1}) ===> (⟶)) ===> (⟶)) *transfer-ball-range transfer-ball-range*›
apply (*rule transfer-ball-range-parametric′*)
using *assms*(*1*) *bi-unique-alt-def bi-unique-conversep* **apply** *blast*
by *auto*
then have ‹(*rel-set U* ===> ((*T* ===> *U*) ===> $(⟶)^{-1-1}$) ===> $(⟶)^{-1-1}$) *transfer-ball-range transfer-ball-range*›
apply (*rule-tac conversepD*[**where** *r*=‹(*rel-set U* ===> ((*T* ===> *U*) ===> $(⟶)^{-1-1}$) ===> $(⟶)^{-1-1}$)›])
by (*simp add*: *conversep-rel-fun del*: *conversep-iff*)
then have *2*: ‹(*rel-set U* ===> ((*T* ===> *U*) ===> (⟷)) ===> $(⟶)^{-1-1}$) *transfer-ball-range transfer-ball-range*›
apply (*rule rev-mp*)
apply (*intro rel-fun-mono′*)
by (*auto simp*: *rev-implies-def*)

from *1 2* **show** *?thesis*
apply (*auto intro*!: *rel-funI simp*: *conversep-iff*[*abs-def*])
apply (*smt* (*z3*) *rel-funE*)
by (*smt* (*verit*) *rel-funE rev-implies-def*)
qed

definition ‹*transfer-Times A B* = *A* × *B*›

lemma *transfer-Times-parametricity*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-set T* ===> *rel-set U* ===> *rel-set* (*rel-prod T U*)) *transfer-Times transfer-Times*›
by (*auto intro*!: *rel-funI simp add*: *transfer-Times-def rel-set-def*)

lemma *csubspace-nonempty*: ‹*csubspace X* ⟹ *X* ≠ {}›
using *complex-vector.subspace-0* **by** *auto*

definition ‹*transfer-vimage-into f U s* = (*f* −‘ *U*) ∩ *s*›

lemma *transfer-vimage-into-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*› ‹*bi-unique B*›
shows ‹((*A* ===> *B*) ===> *rel-set B* ===> *rel-set A* ===> *rel-set A*) *transfer-vimage-into transfer-vimage-into*›
unfolding *transfer-vimage-into-def*
apply (*auto intro*!: *rel-funI simp*: *rel-set-def*)
by (*metis Int-iff apply-rsp' assms bi-unique-def vimage-eq*)+

lemma *make-parametricity-proof-friendly*:
shows ‹(∀ *x*. *P* ⟶ *Q x*) ⟷ (*P* ⟶ (∀ *x*. *Q x*))›
and ‹(∀ *x*. *x* ∈ *S* ⟶ *Q x*) ⟷ (∀ *x*∈*S*. *Q x*)›
and ‹(∀ *x*⊆*S*. *R x*) ⟷ (∀ *x*∈*Pow S*. *R x*)›
and ‹{*x*∈*S*. *Q x*} = *Set.filter Q S*›
and ‹{*x*. *x* ⊆ *S* ∧ *R x*} = *Set.filter R* (*Pow S*)›
and ‹⋀*P*. (∀ *f*. *range f* ⊆ *A* ⟶ *P f*) = *transfer-ball-range A P*›
and ‹⋀*A B*. *A* × *B* = *transfer-Times A B*›
and ‹⋀*B P*. (∃ *A*⊆*B*. *P A*) ⟷ (∃ *A*∈*Pow B*. *P A*)›
and ‹⋀*f U s*. (*f* −‘ *U*) ∩ *s* = *transfer-vimage-into f U s*›
and ‹⋀*M B*. ⨅ *M* ⊓ *principal B* = *transfer-bounded-filter-Inf B M*›
and ‹⋀*F M*. *F* ⊓ *principal M* = *transfer-inf-principal F M*›
by (*auto simp*: *transfer-ball-range-def transfer-Times-def transfer-vimage-into-def transfer-bounded-filter-Inf-def transfer-inf-principal-def*)

7.3 *plus*

locale *plus-ow* =
fixes *U plus*
assumes ‹∀ *x*∈*U*. ∀ *y*∈*U*. *plus x y* ∈ *U*›
lemma *plus-ow-parametricity*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> *A*) ===> (=))

plus-ow plus-ow›
unfolding *plus-ow-def*
by *transfer-prover*

7.3.1 *minus*

locale *minus-ow* = **fixes** *U minus* **assumes** ‹∀ *x*∈*U*. ∀ *y*∈*U*. *minus x y* ∈ *U*›

lemma *minus-ow-parametricity*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> *A*) ===> (=))
minus-ow minus-ow›
unfolding *minus-ow-def*
by *transfer-prover*

7.3.2 *uminus*

locale *uminus-ow* = **fixes** *U uminus* **assumes** ‹∀ *x*∈*U*. *uminus x* ∈ *U*›

lemma *uminus-ow-parametricity*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A*) ===> (=))
uminus-ow uminus-ow›
unfolding *uminus-ow-def*
by *transfer-prover*

7.4 *semigroup*

locale *semigroup-ow* = *plus-ow U plus* **for** *U plus* +
assumes ‹∀ *x*∈*U*. ∀ *y*∈*U*. ∀ *z*∈*U*. *plus x* (*plus y z*) = *plus* (*plus x y*) *z*›

lemma *semigroup-ow-parametricity*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> *A*) ===> (=))
semigroup-ow semigroup-ow›
unfolding *semigroup-ow-def semigroup-ow-axioms-def*
by *transfer-prover*

lemma *semigroup-ow-typeclass*[*simp*, *iff*]: ‹*semigroup-ow V* (+)›
if ‹⋀*x y*. *x*∈*V* ⟹ *y*∈*V* ⟹ *x* + *y* ∈ *V*› **for** *V* :: ‹'*a* :: *semigroup-add set*›
by (*auto intro*!: *plus-ow.intro semigroup-ow.intro semigroup-ow-axioms.intro simp*: *Groups.add-ac that*)

lemma *class-semigroup-add-ud*[*unoverload-def*]: ‹*class.semigroup-add* = *semigroup-ow UNIV*›
by (*auto intro*!: *ext plus-ow.intro simp*: *class.semigroup-add-def semigroup-ow-def semigroup-ow-axioms-def*)

7.5 *abel-semigroup*

locale *abel-semigroup-ow = semigroup-ow U plus* **for** *U plus +*
assumes ‹$\forall x{\in}U.\ \forall y{\in}U.\ plus\ x\ y = plus\ y\ x$›

lemma *abel-semigroup-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> *A*) ===> (=))
abel-semigroup-ow abel-semigroup-ow›
unfolding *abel-semigroup-ow-def abel-semigroup-ow-axioms-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *abel-semigroup-ow-typeclass*[*simp*, *iff*]: ‹*abel-semigroup-ow V* (+)›
if ‹$\bigwedge x\ y.\ x{\in}V \Longrightarrow y{\in}V \Longrightarrow x + y \in V$› **for** *V* :: ‹*'a* :: *ab-semigroup-add set*›
by (*auto simp*: *abel-semigroup-ow-def abel-semigroup-ow-axioms-def Groups.add-ac that*)

lemma *class-ab-semigroup-add-ud*[*unoverload-def*]: ‹*class.ab-semigroup-add = abel-semigroup-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.ab-semigroup-add-def abel-semigroup-ow-def class-semigroup-add-ud abel-semigroup-ow-axioms-def class.ab-semigroup-add-axioms-def*)

7.6 *comm-monoid*

locale *comm-monoid-ow = abel-semigroup-ow U plus* **for** *U plus +*
fixes *zero*
assumes ‹*zero* ∈ *U*›
assumes ‹$\forall x{\in}U.\ plus\ x\ zero = x$›

lemma *comm-monoid-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> *A*) ===> *A* ===> (=))
comm-monoid-ow comm-monoid-ow›
unfolding *comm-monoid-ow-def comm-monoid-ow-axioms-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *comm-monoid-ow-typeclass*[*simp*, *iff*]: ‹*comm-monoid-ow V* (+) *0*›
if ‹*0* ∈ *V*› **and** ‹$\bigwedge x\ y.\ x{\in}V \Longrightarrow y{\in}V \Longrightarrow x + y \in V$› **for** *V* :: ‹*'a* :: *comm-monoid-add set*›
by (*auto simp*: *comm-monoid-ow-def comm-monoid-ow-axioms-def that*)

lemma *class-comm-monoid-add-ud*[*unoverload-def*]: ‹*class.comm-monoid-add = comm-monoid-ow UNIV*›
apply (*auto intro*!: *ext simp*: *class.comm-monoid-add-def comm-monoid-ow-def class-ab-semigroup-add-ud class.comm-monoid-add-axioms-def comm-monoid-ow-axioms-def*)
by (*simp-all add*: *abel-semigroup-ow-def abel-semigroup-ow-axioms-def*)

7.7 *topological-space*

locale *topological-space-ow =*

fixes *U open*
assumes ‹*open U*›
assumes ‹$\forall S{\subseteq}U.\ \forall T{\subseteq}U.$ *open S* $\longrightarrow$ *open T* $\longrightarrow$ *open* $(S \cap T)$›
assumes $\forall K{\subseteq}Pow\ U.\ (\forall S{\in}K.$ *open S*$) \longrightarrow$ *open* $(\bigcup K)$

lemma *topological-space-ow-parametricity*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*rel-set A* ===> (=)) ===> (=))
topological-space-ow topological-space-ow›
unfolding *topological-space-ow-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *class-topological-space-ud*[*unoverload-def*]: ‹*class.topological-space* = *topological-space-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.topological-space-def topological-space-ow-def*)

lemma *topological-space-ow-from-topology*[*simp*]: ‹*topological-space-ow* (*topspace T*) (*openin T*)›
by (*auto intro*!: *topological-space-ow.intro*)

7.8 *sum*

definition ‹*sum-ow z plus f S* =
(*if finite S then the-default z* (*Collect* (*fold-graph* (*plus o f*) *z S*)) *else z*)›
for *U z plus S*

lemma *sum-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*› ‹*bi-unique U*›
shows ‹(*T* ===> (*V* ===> *T* ===> *T*) ===> (*U* ===> *V*) ===> *rel-set U* ===> *T*)
sum-ow sum-ow›
unfolding *sum-ow-def*
by *transfer-prover*

lemma (**in** *comm-monoid-set*) *comp-fun-commute-onI*: ‹*Finite-Set.comp-fun-commute-on UNIV* ((∗) ∘ *g*)›
apply (*rule Finite-Set.comp-fun-commute-on.intro*)
by (*simp add*: *o-def left-commute*)

lemma (**in** *comm-monoid-set*) *F-via-the-default*: ‹*F g A* = *the-default def* (*Collect* (*fold-graph* ((∗) ∘ *g*) **1** *A*))›
if ‹*finite A*›
proof −
have ‹*y* = *x*› **if** ‹*fold-graph* ((∗) ∘ *g*) **1** *A x*› **and** ‹*fold-graph* ((∗) ∘ *g*) **1** *A y*› **for** *x y*
using *that* **apply** (*rule Finite-Set.comp-fun-commute-on.fold-graph-determ*[*rotated 2*, **where** *S*=*UNIV*])
by (*simp-all add*: *comp-fun-commute-onI*)
then have ‹*Ex1* (*fold-graph* ((∗) ∘ *g*) **1** *A*)›

by (*meson finite-imp-fold-graph that*)
then have ‹*card* (*Collect* (*fold-graph* ((∗) ∘ *g*) **1** *A*)) = *1*›
using *card-eq-Suc-0-ex1* **by** *fastforce*
then show *?thesis*
using *that* **by** (*auto simp add*: *the-default-The eq-fold Finite-Set.fold-def*)
qed

lemma *sum-ud*[*unoverload-def*]: ‹*sum* = *sum-ow 0 plus*›
apply (*auto intro*!: *ext simp*: *sum-def sum-ow-def comm-monoid-set.F-via-the-default*)
apply (*subst comm-monoid-set.F-via-the-default*)
apply (*auto simp add*: *sum.comm-monoid-set-axioms*)
by (*metis comm-monoid-add-class.sum-def sum.infinite*)

7.9 *t2-space*

locale *t2-space-ow* = *topological-space-ow* +
assumes ‹∀ *x*∈*U*. ∀ *y*∈*U*. *x* ≠ *y* ⟶ (∃ *S*⊆*U*. ∃ *T*⊆*U*. *open S* ∧ *open T* ∧ *x* ∈ *S* ∧ *y* ∈ *T* ∧ *S* ∩ *T* = {})›

lemma *t2-space-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*rel-set A* ===> (=)) ===> (=))
t2-space-ow t2-space-ow›
unfolding *t2-space-ow-def t2-space-ow-axioms-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *class-t2-space-ud*[*unoverload-def*]: ‹*class.t2-space* = *t2-space-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.t2-space-def class.t2-space-axioms-def t2-space-ow-def*
t2-space-ow-axioms-def class-topological-space-ud)

lemma *t2-space-ow-from-topology*[*simp*, *iff*]: ‹*t2-space-ow* (*topspace T*) (*openin T*)› **if** ‹*Hausdorff-space T*›
using *that*
apply (*auto intro*!: *t2-space-ow.intro simp*: *t2-space-ow-axioms-def Hausdorff-space-def disjnt-def*)
by (*metis openin-subset*)

7.9.1 *continuous-on*

definition *continuous-on-ow* **where** ‹*continuous-on-ow A B opnA opnB s f*
⟷ (∀ *U*⊆*B*. *opnB U* ⟶ (∃ *V*⊆*A*. *opnA V* ∧ (*V* ∩ *s*) = (*f* −‘ *U*) ∩ *s*))›
for *f* :: ‹*'a* ⇒ *'b*›

lemma *continuous-on-ud*[*unoverload-def*]: ‹*continuous-on s f* ⟷ *continuous-on-ow UNIV UNIV open open s f*›
for *f* :: ‹*'a*::*topological-space* ⇒ *'b*::*topological-space*›
unfolding *continuous-on-ow-def continuous-on-open-invariant* **by** *auto*

lemma *continuous-on-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*› ‹*bi-unique B*›
shows ‹(*rel-set A* ===> *rel-set B* ===> (*rel-set A* ===> (⟷)) ===> (*rel-set B* ===> (⟷)) ===> *rel-set A* ===> (*A* ===> *B*) ===> (⟷)) *continuous-on-ow continuous-on-ow*›
unfolding *continuous-on-ow-def make-parametricity-proof-friendly*
by *transfer-prover*

7.10 *scaleR*

locale *scaleR-ow* =
fixes *U* **and** *scaleR* :: ‹*real* ⇒ *'a* ⇒ *'a*›
assumes *scaleR-closed*: ‹∀ *a* ∈ *U*. *scaleR r a* ∈ *U*›

lemma *scaleR-ow-typeclass*[*simp*]: ‹*scaleR-ow UNIV scaleR*› **for** *scaleR*
by (*simp add*: *scaleR-ow-def*)

lemma *scaleR-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> ((=) ===> *A* ===> *A*) ===> (=)) *scaleR-ow scaleR-ow*›
unfolding *scaleR-ow-def make-parametricity-proof-friendly*
by *transfer-prover*

7.11 *scaleC*

locale *scaleC-ow* = *scaleR-ow* +
fixes *scaleC*
assumes *scaleC-closed*: ‹∀ *a*∈*U*. *scaleC c a* ∈ *U*›
assumes ‹∀ *a*∈*U*. *scaleR r a* = *scaleC* (*complex-of-real r*) *a*›

lemma *scaleC-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> ((=) ===> *A* ===> *A*) ===> ((=) ===> *A* ===> *A*) ===> (=)) *scaleC-ow scaleC-ow*›
unfolding *scaleC-ow-def scaleC-ow-axioms-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *class-scaleC-ud*[*unoverload-def*]: ‹*class.scaleC* = *scaleC-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.scaleC-def scaleC-ow-def scaleR-ow-def scaleC-ow-axioms-def*)

7.12 *ab-group-add*

locale *ab-group-add-ow* = *comm-monoid-ow U plus zero* + *minus-ow U minus* + *uminus-ow U uminus*
for *U plus zero minus uminus* +

assumes ‹∀ *a*∈*U*. *uminus a* ∈ *U*›
assumes ∀ *a*∈*U*. *plus* (*uminus a*) *a* = *zero*
assumes ∀ *a*∈*U*. ∀ *b*∈*U*. *minus a b* = *plus a* (*uminus b*)

lemma *ab-group-add-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> *A*) ===> *A* ===> (*A* ===> *A* ===> *A*) ===> (*A* ===> *A*) ===> (=))
ab-group-add-ow ab-group-add-ow›
unfolding *ab-group-add-ow-def ab-group-add-ow-axioms-def*
apply *transfer-prover-start*
apply *transfer-step+*
by *transfer-prover*

lemma *ab-group-add-ow-typeclass*[*simp*]:
‹*ab-group-add-ow V* (+) *0* (−) *uminus*›
if ‹*0* ∈ *V*› ‹∀ *x*∈*V*. −*x* ∈ *V*› ‹∀ *x*∈*V*. ∀ *y*∈*V*. *x* + *y* ∈ *V*›
for *V* :: ‹- :: *ab-group-add set*›
using *that*
apply (*auto intro*!: *ab-group-add-ow.intro ab-group-add-ow-axioms.intro comm-monoid-ow-typeclass minus-ow.intro uminus-ow.intro*)
by *force*

lemma *class-ab-group-add-ud*[*unoverload-def*]: ‹*class.ab-group-add* = *ab-group-add-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.ab-group-add-def ab-group-add-ow-def class-comm-monoid-add-ud minus-ow-def uminus-ow-def ab-group-add-ow-axioms-def class.ab-group-add-axioms-def*)

7.13 *vector-space*

locale *vector-space-ow* = *ab-group-add-ow U plus zero minus uminus*
for *U plus zero minus uminus* +
fixes *scale* :: *'f*::*field* ⇒ *'a* ⇒ *'a*
assumes
‹∀ *x*∈*U*. *scale a x* ∈ *U*›
∀ *x*∈*U*. ∀ *y*∈*U*. *scale a* (*plus x y*) = *plus* (*scale a x*) (*scale a y*)
∀ *x*∈*U*. *scale* (*a* + *b*) *x* = *plus* (*scale a x*) (*scale b x*)
∀ *x*∈*U*. *scale a* (*scale b x*) = *scale* (*a* ∗ *b*) *x*
∀ *x*∈*U*. *scale 1 x* = *x*

lemma *vector-space-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> *A*) ===> *A* ===> (*A* ===> *A* ===> *A*) ===> (*A* ===> *A*) ===> ((=) ===> *A* ===> *A*) ===> (=))
vector-space-ow vector-space-ow›
unfolding *vector-space-ow-def vector-space-ow-axioms-def*
apply *transfer-prover-start*
apply *transfer-step+*

by *simp*

7.14 *complex-vector*

locale *complex-vector-ow = vector-space-ow U plus zero minus uminus scaleC + scaleC-ow U scaleR scaleC*
for *U scaleR scaleC plus zero minus uminus*

lemma *complex-vector-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> ((=) ===> *A* ===> *A*) ===> ((=) ===> *A* ===> *A*) ===> (*A* ===> *A* ===> *A*) ===>
A ===> (*A* ===> *A* ===> *A*) ===> (*A* ===> *A*) ===> (=))
complex-vector-ow complex-vector-ow›
unfolding *complex-vector-ow-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *class-complex-vector-ud*[*unoverload-def*]: ‹*class.complex-vector = complex-vector-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.complex-vector-def vector-space-ow-def vector-space-ow-axioms-def class.complex-vector-axioms-def class.scaleC-def complex-vector-ow-def class-scaleC-ud class-ab-group-add-ud*)

lemma *vector-space-ow-typeclass*[*simp*]:
‹*vector-space-ow V* (+) *0* (−) *uminus* (∗$_C$)›
if [*simp*]: ‹*csubspace V*›
for *V* :: ‹-::*complex-vector set*›
by (*auto intro*!: *vector-space-ow.intro ab-group-add-ow-typeclass scaleC-left.add vector-space-ow-axioms.intro complex-vector.subspace-neg scaleC-add-right complex-vector.subspace-add complex-vector.subspace-scale complex-vector.subspace-0*)

lemma *complex-vector-ow-typeclass*[*simp*]:
‹*complex-vector-ow V* (∗$_R$) (∗$_C$) (+) *0* (−) *uminus*› **if** [*simp*]: ‹*csubspace V*›
by (*auto intro*!: *scaleC-ow-def simp add*: *complex-vector-ow-def scaleC-ow-def scaleC-ow-axioms-def scaleR-ow-def scaleR-scaleC complex-vector.subspace-scale*)

7.15 *open-uniformity*

locale *open-uniformity-ow = open open + uniformity uniformity*
for *A open uniformity +*
assumes *open-uniformity*:
$\bigwedge U.\ U \subseteq A \Longrightarrow$ *open U* $\longleftrightarrow$ ($\forall x{\in}U.$ *eventually* ($\lambda(x', y).\ x' = x \longrightarrow y \in U$) *uniformity*)

lemma *open-uniformity-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*rel-set A* ===> (=)) ===> *rel-filter* (*rel-prod A A*) ===> (=))
open-uniformity-ow open-uniformity-ow›
unfolding *open-uniformity-ow-def make-parametricity-proof-friendly*

by *transfer-prover*

lemma *class-open-uniformity-ud*[*unoverload-def*]: ‹*class.open-uniformity* = *open-uniformity-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.open-uniformity-def open-uniformity-ow-def*)

lemma *open-uniformity-on-typeclass*[*simp*]:
fixes *V* :: ‹-::*open-uniformity set*›
assumes ‹*closed V*›
shows ‹*open-uniformity-ow V* (*openin* (*top-of-set V*)) (*uniformity-on V*)›
proof (*rule open-uniformity-ow.intro*, *intro allI impI iffI ballI*)
fix *U* **assume** ‹*U* ⊆ *V*›
assume ‹*openin* (*top-of-set V*) *U*›
then obtain *T* **where** ‹*U* = *T* ∩ *V*› **and** ‹*open T*›
by (*metis Int-ac*(*3*) *openin-open*)
with *open-uniformity*
have ∗: ‹∀ *F* (*x*′, *y*) *in uniformity*. *x*′ = *x* ⟶ *y* ∈ *T*› **if** ‹*x* ∈ *T*› **for** *x*
using *that* **by** *blast*
have ‹∀ *F* (*x*′, *y*) *in uniformity-on V*. *x*′ = *x* ⟶ *y* ∈ *U*› **if** ‹*x* ∈ *U*› **for** *x*
apply (*rule eventually-inf-principal*[*THEN iffD2*])
using ∗[*of x*] **apply** (*rule eventually-rev-mp*)
using ‹*U* = *T* ∩ *V*› *that* **by** (*auto intro*!: *always-eventually*)
then show ‹∀ *F* (*x*′, *y*) *in uniformity-on V*. *x*′ = *x* ⟶ *y* ∈ *U*› **if** ‹*x* ∈ *U*› **for** *x*
using *that* **by** *blast*
next
fix *U* **assume** ‹*U* ⊆ *V*›
assume *asm*: ‹∀ *x*∈*U*. ∀ *F* (*x*′, *y*) *in uniformity-on V*. *x*′ = *x* ⟶ *y* ∈ *U*›
from *asm*[*rule-format*]
have ‹∀ *F* (*x*′, *y*) *in uniformity*. *x*′ ∈ *V* ∧ *y* ∈ *V* ∧ *x*′ = *x* ⟶ *y* ∈ *U* ∪ − *V*› **if** ‹*x* ∈ *U*› **for** *x*
unfolding *eventually-inf-principal*
apply (*rule eventually-rev-mp*)
using *that* **by** (*auto intro*!: *always-eventually*)
then have *xU*: ‹∀ *F* (*x*′, *y*) *in uniformity*. *x*′ = *x* ⟶ *y* ∈ *U* ∪ − *V*› **if** ‹*x* ∈ *U*› **for** *x*
apply (*rule eventually-rev-mp*)
using *that* ‹*U* ⊆ *V*› **by** (*auto intro*!: *always-eventually*)
have ‹*open* (− *V*)›
using *assms* **by** *auto*
with *open-uniformity*
have ‹∀ *F* (*x*′, *y*) *in uniformity*. *x*′ = *x* ⟶ *y* ∈ − *V*› **if** ‹*x* ∈ − *V*› **for** *x*
using *that* **by** *blast*
then have *xV*: ‹∀ *F* (*x*′, *y*) *in uniformity*. *x*′ = *x* ⟶ *y* ∈ *U* ∪ − *V*› **if** ‹*x* ∈ − *V*› **for** *x*
apply (*rule eventually-rev-mp*)
apply (*rule that*)
apply (*rule always-eventually*)
by *auto*
have ‹∀ *F* (*x*′, *y*) *in uniformity*. *x*′ = *x* ⟶ *y* ∈ *U* ∪ − *V*› **if** ‹*x* ∈ *U* ∪ − *V*› **for** *x*
using *xV*[*of x*] *xU*[*of x*] *that*
by *auto*

then have ‹*open* (*U* ∪ − *V*)›
using *open-uniformity* **by** *blast*
then show ‹*openin* (*top-of-set V*) *U*›
using ‹*U* ⊆ *V*›
by (*auto intro*!: *exI simp*: *openin-open*)
qed

7.16 *uniformity-dist*

locale *uniformity-dist-ow* = *dist dist* + *uniformity uniformity* **for** *U dist uniformity* +
assumes *uniformity-dist*: *uniformity* = (⨅ *e*∈{*0*<..}. *principal* {(*x*, *y*)∈*U*×*U*. *dist x y* < *e*})

lemma *class-uniformity-dist-ud*[*unoverload-def*]: ‹*class.uniformity-dist* = *uniformity-dist-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.uniformity-dist-def uniformity-dist-ow-def*)

lemma *uniformity-dist-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> (=)) ===> *rel-filter* (*rel-prod A A*) ===> (=))
uniformity-dist-ow uniformity-dist-ow›
proof −
have ∗: *uniformity-dist-ow U dist uniformity* ⟷
uniformity = *transfer-bounded-filter-Inf* (*transfer-Times U U*)
((λ*e*. *principal* (*Set.filter* (λ(*x*,*y*). *dist x y* < *e*) (*transfer-Times U U*))) ‘ {*0*<..})
for *U dist uniformity*
unfolding *uniformity-dist-ow-def make-parametricity-proof-friendly case-prod-unfold prod.collapse*
apply (*subst Inf-bounded-transfer-bounded-filter-Inf*[**where** *B*=‹*U*×*U*›])
by (*auto simp*: *transfer-Times-def*)
show *?thesis*
unfolding ∗[*abs-def*]
by *transfer-prover*
qed

lemma *uniformity-dist-on-typeclass*[*simp*]: ‹*uniformity-dist-ow V dist* (*uniformity-on V*)› **for** *V* :: ‹-::*uniformity-dist set*›
apply (*auto simp add*: *uniformity-dist-ow-def uniformity-dist simp flip*: *INF-inf-const2*)
apply (*subst asm-rl*[*of* ‹⋀*x*. *Restr* {(*xa*, *y*). *dist xa y* < *x*} *V* = {(*xa*, *y*). *xa* ∈ *V* ∧ *y* ∈ *V* ∧ *dist xa y* < *x*}›, *rule-format*])
by *auto*

7.17 *sgn*

locale *sgn-ow* =
fixes *U* **and** *sgn* :: ‹*'a* ⇒ *'a*›
assumes *sgn-closed*: ‹∀ *a*∈*U*. *sgn a* ∈ *U*›

lemma *sgn-ow-parametric*[*transfer-rule*]:

includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A*) ===> (=))
sgn-ow sgn-ow›
unfolding *sgn-ow-def*
by *transfer-prover*

7.18 *sgn-div-norm*

locale *sgn-div-norm-ow* = *scaleR-ow U scaleR* + *norm norm* + *sgn-ow U sgn* **for** *U sgn norm scaleR* +
assumes ∀ *x*∈*U*. *sgn x* = *scaleR* (*inverse* (*norm x*)) *x*

lemma *class-sgn-div-norm-ud*[*unoverload-def*]: ‹*class.sgn-div-norm* = *sgn-div-norm-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.sgn-div-norm-def sgn-div-norm-ow-def sgn-div-norm-ow-axioms-def unoverload-def sgn-ow-def*)

lemma *sgn-div-norm-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A*) ===> (*A* ===> (=)) ===> ((=) ===> *A* ===> *A*) ===> (=))
sgn-div-norm-ow sgn-div-norm-ow›
unfolding *sgn-div-norm-ow-def sgn-div-norm-ow-axioms-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *sgn-div-norm-on-typeclass*[*simp*]:
fixes *V* :: ‹-::*sgn-div-norm set*›
assumes ‹⋀*v r*. *v*∈*V* ⟹ *scaleR r v* ∈ *V*›
shows ‹*sgn-div-norm-ow V sgn norm* (∗*R*)›
using *assms*
by (*auto simp add*: *sgn-ow-def sgn-div-norm-ow-axioms-def scaleR-ow-def sgn-div-norm-ow-def sgn-div-norm*)

7.19 *dist-norm*

locale *dist-norm-ow* = *dist dist* + *norm norm* + *minus-ow U minus* **for** *U minus dist norm* +
assumes *dist-norm*: ∀ *x*∈*U*. ∀ *y*∈*U*. *dist x y* = *norm* (*minus x y*)

lemma *dist-norm-ud*[*unoverload-def*]: ‹*class.dist-norm* = *dist-norm-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.dist-norm-def dist-norm-ow-def dist-norm-ow-axioms-def minus-ow-def unoverload-def*)

lemma *dist-norm-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> *A*) ===> (*A* ===> *A* ===> (=)) ===> (*A* ===> (=)) ===> (=))
dist-norm-ow dist-norm-ow›

unfolding *dist-norm-ow-def dist-norm-ow-axioms-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *dist-norm-ow-typeclass*[*simp*]:
fixes A :: ‹-::*dist-norm set*›
assumes ‹$\bigwedge a\ b.\ [\![\ a \in A;\ b \in A\ ]\!] \Longrightarrow a - b \in A$›
shows ‹*dist-norm-ow A* (−) *dist norm*›
by (*auto simp add*: *assms dist-norm-ow-def minus-ow-def dist-norm-ow-axioms-def dist-norm*)

7.20 *complex-inner*

locale *complex-inner-ow* = *complex-vector-ow U scaleR scaleC plus zero minus uminus*
+ *dist-norm-ow U minus dist norm* + *sgn-div-norm-ow U sgn norm scaleR*
+ *uniformity-dist-ow U dist uniformity*
+ *open-uniformity-ow U open uniformity*
for *U scaleR scaleC plus zero minus uminus dist norm sgn uniformity open* +
fixes *cinner* :: $'a \Rightarrow 'a \Rightarrow$ *complex*
assumes $\forall x{\in}U.\ \forall y{\in}U.$ *cinner x y* = *cnj* (*cinner y x*)
and $\forall x{\in}U.\ \forall y{\in}U.\ \forall z{\in}U.$ *cinner* (*plus x y*) *z* = *cinner x z* + *cinner y z*
and $\forall x{\in}U.\ \forall y{\in}U.$ *cinner* (*scaleC r x*) *y* = *cnj r* ∗ *cinner x y*
and $\forall x{\in}U.$ *0* ≤ *cinner x x*
and $\forall x{\in}U.$ *cinner x x* = *0* ⟷ *x* = *zero*
and $\forall x{\in}U.$ *norm x* = *sqrt* (*cmod* (*cinner x x*))

lemma *class-complex-inner-ud*[*unoverload-def*]: ‹*class.complex-inner* = *complex-inner-ow UNIV*›
apply (*intro ext*)
by (*simp add*: *class.complex-inner-def class.complex-inner-axioms-def complex-inner-ow-def complex-inner-ow-axioms-def unoverload-def*)

lemma *complex-inner-ow-parametricity*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*›
shows ‹(*rel-set T* ===> ((=) ===> *T* ===> *T*) ===> ((=) ===> *T* ===> *T*) ===> (*T* ===> *T* ===> *T*) ===> *T*
===> (*T* ===> *T* ===> *T*) ===> (*T* ===> *T*) ===> (*T* ===> *T* ===> (=)) ===> (*T* ===> (=))
===> (*T* ===> *T*) ===> *rel-filter* (*rel-prod T T*) ===> (*rel-set T* ===> (=))
===> (*T* ===> *T* ===> (=)) ===> (=)) *complex-inner-ow complex-inner-ow*›
unfolding *complex-inner-ow-def complex-inner-ow-axioms-def*
by *transfer-prover*

lemma *complex-inner-ow-typeclass*[*simp*]:
fixes V :: ‹-::*complex-inner set*›
assumes [*simp*]: ‹*closed V*› ‹*csubspace V*›
shows ‹*complex-inner-ow V* ($*_R$) ($*_C$) (+) *0* (−) *uminus dist norm sgn* (*uniformity-on V*) (*openin* (*top-of-set V*)) ($\cdot_C$)›
apply (*auto intro*!: *complex-vector-ow-typeclass dist-norm-ow-typeclass sgn-div-norm-on-typeclass*
simp: *complex-inner-ow-def cinner-simps complex-vector.subspace-diff complex-inner-ow-axioms-def*
scaleR-scaleC complex-vector.subspace-scale

simp flip: *norm-eq-sqrt-cinner*)
by −

7.21 *is-ortho-set*

definition *is-ortho-set-ow* **where** ‹*is-ortho-set-ow zero cinner S* ⟷
(($\forall x{\in}S.\ \forall y{\in}S.\ x \neq y \longrightarrow$ *cinner x y* = *0*) ∧ *zero* ∉ *S*)›
for *zero cinner*

lemma *is-ortho-set-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*A* ===> (*A* ===> *A* ===> (=)) ===> *rel-set A* ===> (=))
is-ortho-set-ow is-ortho-set-ow›
unfolding *is-ortho-set-ow-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *is-ortho-set-ud*[*unoverload-def*]: ‹*is-ortho-set* = *is-ortho-set-ow 0 cinner*›
by (*auto simp*: *is-ortho-set-ow-def is-ortho-set-def*)

7.22 *metric-space*

locale *metric-space-ow* = *uniformity-dist-ow U dist uniformity* + *open-uniformity-ow U open uniformity*
for *U dist uniformity open* +
assumes $\forall x \in U.\ \forall y \in U.$ *dist x y* = *0* ⟷ *x* = *y*
and $\forall x{\in}U.\ \forall y{\in}U.\ \forall z{\in}U.$ *dist x y* ≤ *dist x z* + *dist y z*

lemma *metric-space-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*A* ===> *A* ===> (=)) ===> *rel-filter* (*rel-prod A A*) ===>
(*rel-set A* ===> (=)) ===> (=))
metric-space-ow metric-space-ow›
unfolding *metric-space-ow-def metric-space-ow-axioms-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *class-metric-space-ud*[*unoverload-def*]: ‹*class.metric-space* = *metric-space-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.metric-space-def class.metric-space-axioms-def metric-space-ow-def metric-space-ow-axioms-def unoverload-def*)

lemma *metric-space-ow-typeclass*[*simp*]:
fixes *V* :: ‹-::*metric-space set*›
assumes ‹*closed V*›
shows ‹*metric-space-ow V dist* (*uniformity-on V*) (*openin* (*top-of-set V*))›
by (*auto simp*: *assms metric-space-ow-def metric-space-ow-axioms-def class.metric-space-axioms-def dist-triangle2*)

7.23 *nhds*

definition *nhds-ow* **where** ‹*nhds-ow U open a* = (*INF S*∈{*S*. *S* ⊆ *U* ∧ *open S* ∧ *a* ∈ *S*}. *principal S*) ⊓ *principal U*›
for *U open*

lemma *nhds-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> (*rel-set A* ===> (=)) ===> *A* ===> *rel-filter A*)
nhds-ow nhds-ow›
unfolding *nhds-ow-def*[*folded transfer-bounded-filter-Inf-def*] *make-parametricity-proof-friendly*
by *transfer-prover*

lemma *topological-space-nhds-ud*[*unoverload-def*]: ‹*topological-space.nhds* = *nhds-ow UNIV*›
by (*auto intro*!: *ext simp add*: *nhds-ow-def* [[*axiom topological-space.nhds-def-raw*]])

lemma *nhds-ud*[*unoverload-def*]: ‹*nhds* = *nhds-ow UNIV open*›
by (*auto intro*!: *ext simp add*: *nhds-ow-def nhds-def*)

lemma *nhds-ow-topology*[*simp*]: ‹*nhds-ow* (*topspace T*) (*openin T*) *x* = *nhdsin T x*› **if** ‹*x* ∈ *topspace T*›
using *that* **apply** (*auto intro*!: *ext simp add*: *nhds-ow-def nhdsin-def*[*abs-def*])
apply (*subst INF-inf-const2*[*symmetric*])
using *openin-subset* **by** (*auto intro*!: *INF-cong*)

7.24 *at-within*

definition ‹*at-within-ow U open a s* = *nhds-ow U open a* ⊓ *principal* (*s* − {*a*})›
for *U open a s*

lemma *at-within-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*›
shows ‹((*rel-set T*) ===> (*rel-set T* ===> (=)) ===> *T* ===> *rel-set T* ===> *rel-filter T*)
at-within-ow at-within-ow›
unfolding *at-within-ow-def make-parametricity-proof-friendly transfer-inf-principal-def*[*symmetric*]
by *transfer-prover*

lemma *at-within-ud*[*unoverload-def*]: ‹*at-within* = *at-within-ow UNIV open*›
by (*auto intro*!: *ext simp*: *at-within-def at-within-ow-def unoverload-def*)

lemma *at-within-ow-topology*:
‹*at-within-ow* (*topspace T*) (*openin T*) *a S* = *nhdsin T a* ⊓ *principal* (*S* − {*a*})›
if ‹*a* ∈ *topspace T*›
using *that* **unfolding** *at-within-ow-def* **by** (*simp add*: *nhds-ow-topology*)

7.25 (*has-sum*)

definition ‹*has-sum-ow U plus zero open f A x* =
filterlim (*sum-ow zero plus f*) (*nhds-ow U* (λ*S*. *open S*) *x*)
(*finite-subsets-at-top A*)›
for *U plus zero open f A x*

lemma *has-sum-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*› ‹*bi-unique U*›
shows ‹(*rel-set T* ===> (*V* ===> *T* ===> *T*) ===> *T* ===> (*rel-set T* ===> (=))
===> (*U* ===> *V*) ===> *rel-set U* ===> *T* ===> (=))
has-sum-ow has-sum-ow›
unfolding *has-sum-ow-def*
by *transfer-prover*

lemma *has-sum-ud*[*unoverload-def*]: ‹*HAS-SUM* = *has-sum-ow UNIV plus* (*0*::'*a*::{*comm-monoid-add*,*topological-spa*
open›
by (*auto intro*!: *ext simp*: *has-sum-def has-sum-ow-def unoverload-def*)

lemma *has-sum-ow-topology*:
assumes ‹*l* ∈ *topspace T*›
assumes ‹*0* ∈ *topspace T*›
assumes ‹⋀*x y*. *x* ∈ *topspace T* ⟹ *y* ∈ *topspace T* ⟹ *x* + *y* ∈ *topspace T*›
shows ‹*has-sum-ow* (*topspace T*) (+) *0* (*openin T*) *f S l* ⟷ *has-sum-in T f S l*›
using *assms* **apply** (*simp add*: *has-sum-ow-def has-sum-in-def nhds-ow-topology sum-ud*[*symmetric*])
by (*metis filterlim-nhdsin-iff-limitin*)

7.26 *filterlim*

7.27 *convergent*

definition *convergent-ow* **where**
‹*convergent-ow U open X* ⟷ (∃ *L*∈*U*. *filterlim X* (*nhds-ow U open L*) *sequentially*)›
for *U open*

lemma *convergent-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*›
shows ‹(*rel-set T* ===> (*rel-set T* ===> (=)) ===> ((=) ===> *T*) ===> (⟷))
convergent-ow convergent-ow›
unfolding *convergent-ow-def*
by *transfer-prover*

lemma *convergent-ud*[*unoverload-def*]: ‹*convergent* = *convergent-ow UNIV open*›
by (*auto simp*: *convergent-ow-def*[*abs-def*] *convergent-def*[*abs-def*] *unoverload-def*)

lemma *topological-space-convergent-ud*[*unoverload-def*]: ‹*topological-space.convergent* = *convergent-ow UNIV*›
by (*auto intro*!: *ext simp*: [[*axiom topological-space.convergent-def-raw*]]

convergent-ow-def unoverload-def)

lemma *convergent-ow-topology*[*simp*]:
‹*convergent-ow* (*topspace T*) (*openin T*) *f* ⟷ (∃ *l*. *limitin T f l sequentially*)›
by (*auto simp*: *convergent-ow-def simp flip*: *filterlim-nhdsin-iff-limitin*)

lemma *convergent-ow-typeclass*[*simp*]:
‹*convergent-ow V* (*openin* (*top-of-set V*)) *f* ⟷ (∃ *l*. *limitin* (*top-of-set V*) *f l sequentially*)›
by (*simp flip*: *convergent-ow-topology*)

7.28 *uniform-space.cauchy-filter*

lemma *cauchy-filter-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique T*
shows (*rel-filter* (*rel-prod T T*) ===> *rel-filter T* ===> (=))
uniform-space.cauchy-filter
uniform-space.cauchy-filter
unfolding [[*axiom uniform-space.cauchy-filter-def-raw*]]
by *transfer-prover*

7.29 *uniform-space.Cauchy*

lemma *uniform-space-Cauchy-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique T*
shows (*rel-filter* (*rel-prod T T*) ===> ((=) ===> *T*) ===> (=))
uniform-space.Cauchy
uniform-space.Cauchy
unfolding [[*axiom uniform-space.Cauchy-uniform-raw*]]
using *filtermap-parametric*[*transfer-rule*] **apply** *fail?*
by *transfer-prover*

7.30 *complete-space*

locale *complete-space-ow* = *metric-space-ow U dist uniformity open*
for *U dist uniformity open* +
assumes ‹*range X* ⊆ *U* ⟶ *uniform-space.Cauchy uniformity X* ⟶ *convergent-ow U open X*›

lemma *class-complete-space-ud*[*unoverload-def*]: ‹*class.complete-space* = *complete-space-ow UNIV*›
by (*auto intro*!: *ext simp*: *class.complete-space-def class.complete-space-axioms-def complete-space-ow-def complete-space-ow-axioms-def unoverload-def*)

lemma *complete-space-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique T*
shows (*rel-set T* ===> (*T* ===> *T* ===> (=)) ===> *rel-filter* (*rel-prod T T*) ===> (*rel-set T* ===> (=)) ===> (=))

complete-space-ow complete-space-ow
unfolding *complete-space-ow-def complete-space-ow-axioms-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *complete-space-ow-typeclass*[*simp*]:
fixes *V* :: ‹-::*uniform-space set*›
assumes ‹*complete V*›
shows ‹*complete-space-ow V dist* (*uniformity-on V*) (*openin* (*top-of-set V*))›
proof (*rule complete-space-ow.intro*)
show ‹*metric-space-ow V dist* (*uniformity-on V*) (*openin* (*top-of-set V*))›
apply (*rule metric-space-ow-typeclass*)
by (*simp add*: *assms complete-imp-closed*)
have ‹∃ *l*. *limitin* (*top-of-set V*) *X l sequentially*›
if *XV*: ‹⋀*n*. *X n* ∈ *V*› **and** *cauchy*: ‹*uniform-space.Cauchy* (*uniformity-on V*) *X*› **for** *X*
proof −
from *cauchy*
have ‹*uniform-space.cauchy-filter* (*uniformity-on V*) (*filtermap X sequentially*)›
by (*simp add*: [[*axiom uniform-space.Cauchy-uniform-raw*]])
then have ‹*cauchy-filter* (*filtermap X sequentially*)›
by (*auto simp*: *cauchy-filter-def* [[*axiom uniform-space.cauchy-filter-def-raw*]])
then have ‹*Cauchy X*›
by (*simp add*: *Cauchy-uniform*)
with ‹*complete V*› *XV* **obtain** *l* **where** *l*: ‹*X* ⟶ *l*› ‹*l* ∈ *V*›
apply *atomize-elim*
by (*meson completeE*)
with *XV l* **show** *?thesis*
by (*auto intro*!: *exI*[*of* - *l*] *simp*: *convergent-def limitin-subtopology*)
qed
then show ‹*complete-space-ow-axioms V* (*uniformity-on V*) (*openin* (*top-of-set V*))›
apply (*auto simp*: *complete-space-ow-axioms-def complete-imp-closed assms*)
by *blast*
qed

7.31 *chilbert-space*

locale *chilbert-space-ow* = *complex-inner-ow* + *complete-space-ow*

lemma *chilbert-space-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›
shows ‹(*rel-set A* ===> ((=) ===> *A* ===> *A*) ===> ((=) ===> *A* ===> *A*) ===> (*A* ===> *A* ===> *A*) ===>
A ===> (*A* ===> *A* ===> *A*) ===> (*A* ===> *A*) ===> (*A* ===> *A* ===> (=)) ===> (*A* ===> (=)) ===>
(*A* ===> *A*) ===> *rel-filter* (*rel-prod A A*) ===> (*rel-set A* ===> (=)) ===> (*A* ===> *A* ===> (=)) ===> (=))
chilbert-space-ow chilbert-space-ow›
unfolding *chilbert-space-ow-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *chilbert-space-on-typeclass*[*simp*]:
fixes V :: ‹-::*complex-inner set*›
assumes ‹*complete V*› ‹*csubspace V*›
shows ‹*chilbert-space-ow V* $(*_R)$ $(*_C)$ $(+)$ *0* $(-)$ *uminus dist norm sgn*
(uniformity-on V) (openin (top-of-set V)) $(\cdot_C)$›
by (*auto intro*!: *chilbert-space-ow.intro complex-inner-ow-typeclass*
simp: *assms complete-imp-closed*)

lemma *class-chilbert-space-ud*[*unoverload-def*]:
‹*class.chilbert-space = chilbert-space-ow UNIV*›
by (*auto intro*!: *ext simp add*: *class.chilbert-space-def chilbert-space-ow-def unoverload-def*)

7.32 (*hull*)

definition ‹*hull-ow A S s* = $((\lambda x.\ S\ x \wedge x \subseteq A)$ *hull s*$) \cap A$›

lemma *hull-ow-nondegenerate*: ‹*hull-ow A S s* = $((\lambda x.\ S\ x \wedge x \subseteq A)$ *hull s*)› **if** ‹$x \subseteq A$› **and** ‹$s \subseteq x$› **and** ‹*S x*›
proof −
have ‹$((\lambda x.\ S\ x \wedge x \subseteq A)$ *hull s*$) \subseteq x$›
apply (*rule hull-minimal*)
using *that* **by** *auto*
also note ‹$x \subseteq A$›
finally show *?thesis*
unfolding *hull-ow-def* **by** *auto*
qed

definition ‹*transfer-bounded-Inf B M* = *Inf M* $\sqcap$ *B*›

lemma *transfer-bounded-Inf-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes ‹*bi-unique T*›
shows ‹(*rel-set T* ===> *rel-set* (*rel-set T*) ===> *rel-set T*) *transfer-bounded-Inf transfer-bounded-Inf*›
apply (*auto intro*!: *rel-funI simp*: *transfer-bounded-Inf-def rel-set-def Bex-def*)
apply (*metis* (*full-types*) *assms bi-uniqueDr*)
by (*metis* (*full-types*) *assms bi-uniqueDl*)

lemma *hull-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique T*
shows (*rel-set T* ===> (*rel-set T* ===> (=)) ===> *rel-set T* ===> *rel-set T*)
hull-ow hull-ow
proof −
have $*$: ‹*hull-ow A S s* = *transfer-bounded-Inf A* (*Set.filter* $(\lambda x.\ S\ x \wedge s \subseteq x)$ (*Pow A*))› **for** *A S s*
by (*auto simp add*: *hull-ow-def hull-def transfer-bounded-Inf-def*)
show *?thesis*

unfolding $*$
by *transfer-prover*
qed

lemma *hull-ow-ud*[*unoverload-def*]: ‹(*hull*) = *hull-ow UNIV*›
unfolding *hull-def hull-ow-def* **by** *auto*

7.33 *csubspace*

definition
‹*subspace-ow plus zero scale S* = (*zero* ∈ *S* ∧ (∀ *x*∈*S*. ∀ *y*∈*S*. *plus x y* ∈ *S*) ∧ (∀ *c*. ∀ *x*∈*S*. *scale c x* ∈ *S*))›
for *plus zero scale S*

lemma *subspace-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*›
shows ‹((*T* ===> *T* ===> *T*) ===> *T* ===> ((=) ===> *T* ===> *T*) ===> *rel-set T* ===> (=))
subspace-ow subspace-ow›
unfolding *subspace-ow-def*
by *transfer-prover*

lemma *module-subspace-ud*[*unoverload-def*]: ‹*module.subspace* = *subspace-ow plus 0*›
by (*auto intro*!: *ext simp*: [[*axiom module.subspace-def-raw*]] *subspace-ow-def*)

lemma *csubspace-ud*[*unoverload-def*]: ‹*csubspace* = *subspace-ow* (+) *0* ($*_C$)›
by (*simp add*: *csubspace-raw-def module-subspace-ud*)

7.34 *cspan*

definition
‹*span-ow U plus zero scale b* = *hull-ow U* (*subspace-ow plus zero scale*) *b*›
for *U plus zero scale b*

lemma *span-ow-on-typeclass*:
assumes ‹*csubspace U*›
assumes ‹*B* ⊆ *U*›
shows ‹*span-ow U plus 0 scaleC B* = *cspan B*›
proof −
have ‹*span-ow U plus 0 scaleC B* = (λ*x*. *csubspace x* ∧ *x* ⊆ *U*) *hull B*›
using *assms*
by (*auto simp add*: *span-ow-def hull-ow-nondegenerate*[**where** *x*=*U*] *csubspace-raw-def simp flip*: *csubspace-ud*)
also have ‹(λ*x*. *csubspace x* ∧ *x* ⊆ *U*) *hull B* = *cspan B*›
apply (*rule hull-unique*)
using *assms*(*2*) *complex-vector.span-superset* **apply** *force*
by (*simp-all add*: *assms complex-vector.span-minimal*)
finally show *?thesis*

by −
qed

lemma (**in** *Modules.module*) *span-ud*[*unoverload-def*]: ‹*span* = *span-ow UNIV plus 0 scale*›
by (*auto intro*!: *ext simp*: *span-def span-ow-def*
module-subspace-ud hull-ow-ud)

lemmas *cspan-ud*[*unoverload-def*] = *complex-vector.span-ud*

lemma *span-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*›
shows ‹(*rel-set T* ===> (*T* ===> *T* ===> *T*) ===> *T* ===> ((=) ===> *T* ===> *T*) ===> *rel-set T* ===> *rel-set T*)
span-ow span-ow›
unfolding *span-ow-def*
by *transfer-prover*

7.34.1 (*islimpt*)

definition ‹*islimpt-ow U open x S* ⟷ (∀ *T*⊆*U*. *x*∈*T* ⟶ *open T* ⟶ (∃ *y*∈*S*. *y*∈*T* ∧ *y*≠*x*))›
for *open*

lemma *islimpt-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*›
shows ‹(*rel-set T* ===> (*rel-set T* ===> (=)) ===> *T* ===> *rel-set T* ===> (⟷)) *islimpt-ow islimpt-ow*›
unfolding *islimpt-ow-def make-parametricity-proof-friendly*
by *transfer-prover*

definition ‹*islimptin T x S* ⟷ *x* ∈ *topspace T* ∧ (∀ *V*. *x* ∈ *V* ⟶ *openin T V* ⟶ (∃ *y*∈*S*. *y* ∈ *V* ∧ *y* ≠ *x*))›

lemma *islimpt-ow-from-topology*: ‹*islimpt-ow* (*topspace T*) (*openin T*) *x S* ⟷ *islimptin T x S* ∨ *x* ∉ *topspace T*›
apply (*cases* ‹*x* ∈ *topspace T*›)
apply (*simp-all add*: *islimpt-ow-def islimptin-def Pow-def*)
by *blast*+

7.34.2 *closure*

definition ‹*closure-ow U open S* = *S* ∪ {*x*∈*U*. *islimpt-ow U open x S*}› **for** *open*

lemma *closure-ow-with-typeclass*[*simp*]:
‹*closure-ow X* (*openin* (*top-of-set X*)) *S* = (*X* ∩ *closure* (*X* ∩ *S*)) ∪ *S*›
proof −
have ‹*closure-ow X* (*openin* (*top-of-set X*)) *S* = (*top-of-set X*) *closure-of S* ∪ *S*›
apply (*simp add*: *closure-ow-def islimpt-ow-def closure-of-def*)

apply *safe*
apply (*meson PowI openin-imp-subset*)
by *auto*
also have ‹... = (*X* ∩ *closure* (*X* ∩ *S*)) ∪ *S*›
by (*simp add*: *closure-of-subtopology*)
finally show *?thesis*
by −
qed

lemma *closure-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique T*›
shows ‹(*rel-set T* ===> (*rel-set T* ===> (=)) ===> *rel-set T* ===> *rel-set T*) *closure-ow closure-ow*›
unfolding *closure-ow-def make-parametricity-proof-friendly*
by *transfer-prover*

lemma *closure-ow-from-topology*: ‹*closure-ow* (*topspace T*) (*openin T*) *S* = *T closure-of S*› **if** ‹*S* ⊆ *topspace T*›
using *that* **apply** (*auto simp*: *closure-ow-def islimpt-ow-from-topology in-closure-of*)
apply (*meson in-closure-of islimptin-def*)
by (*metis islimptin-def*)

lemma *closure-ud*[*unoverload-def*]: ‹*closure* = *closure-ow UNIV open*›
unfolding *closure-def closure-ow-def islimpt-def islimpt-ow-def* **by** *auto*

7.35 *continuous*

lemma *continuous-on-ow-from-topology*: ‹*continuous-on-ow* (*topspace T*) (*topspace U*) (*openin T*) (*openin U*) (*topspace T*) *f* ⟷ *continuous-map T U f*›
if ‹*f* ‘ *topspace T* ⊆ *topspace U*›
apply (*simp add*: *continuous-on-ow-def continuous-map-def*)
apply *safe*
apply (*meson image-subset-iff that*)
apply (*smt* (*verit*) *Collect-mono-iff Int-def inf-absorb1 mem-Collect-eq openin-subopen openin-subset vimage-eq*)
by *blast*

7.36 *is-onb*

definition
‹*is-onb-ow U scaleC plus zero norm open cinner E* ⟷ *is-ortho-set-ow zero cinner E* ∧ (∀ *b*∈*E*. *norm b* = *1*) ∧
closure-ow U open (*span-ow U plus zero scaleC E*) = *U*›
for *U scaleC plus zero norm open cinner*

lemma *is-onb-ow-parametric*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: ‹*bi-unique A*›

shows ‹(*rel-set A* ===>
((=) ===> *A* ===> *A*) ===>
(*A* ===> *A* ===> *A*) ===>
A ===>
(*A* ===> (=)) ===> (*rel-set A* ===> (=)) ===> (*A* ===> *A* ===> (=)) ===>
rel-set A ===> (=))
is-onb-ow is-onb-ow›
unfolding *is-onb-ow-def*
by *transfer-prover*

lemma *is-onb-ud*[*unoverload-def*]:
‹*is-onb* = *is-onb-ow UNIV scaleC plus 0 norm open cinner*›
unfolding *is-onb-def is-onb-ow-def*
apply (*subst asm-rl*[*of* ‹⋀*E*. *ccspan E* = ⊤ ⟷ *closure* (*cspan E*) = *UNIV*›, *rule-format*])
apply (*transfer*, *rule*)
unfolding *unoverload-def*
apply *transfer* **by** *auto*

7.37 Transferring theorems

lemma *closure-of-eqI*:
fixes *f g* :: ‹'*a* ⇒ '*b*› **and** *T* :: ‹'*a topology*› **and** *U* :: ‹'*b topology*›
assumes *hausdorff*: ‹*Hausdorff-space U*›
assumes *f-eq-g*: ‹⋀*x*. *x* ∈ *S* ⟹ *f x* = *g x*›
assumes *x*: ‹*x* ∈ *T closure-of S*›
assumes *f*: ‹*continuous-map T U f*› **and** *g*: ‹*continuous-map T U g*›
shows ‹*f x* = *g x*›
proof −
have ‹*topspace T* ≠ {}›
by (*metis assms*(*3*) *equals0D in-closure-of*)
have ‹*topspace U* ≠ {}›
using ‹*topspace T* ≠ {}› *assms*(*5*) *continuous-map-image-subset-topspace* **by** *blast*

{
assume ∃(*Rep* :: '*t* ⇒ '*a*) *Abs*. *type-definition Rep Abs* (*topspace T*)
then interpret *T*: *local-typedef* ‹*topspace T*› ‹*TYPE*('*t*)›
by *unfold-locales*
assume ∃(*Rep* :: '*u* ⇒ '*b*) *Abs*. *type-definition Rep Abs* (*topspace U*)
then interpret *U*: *local-typedef* ‹*topspace U*› ‹*TYPE*('*u*)›
by *unfold-locales*

note *on-closure-eqI*
note *this*[*unfolded unoverload-def*]
note *this*[*unoverload-type* '*b*, *unoverload-type* '*a*]
note *this*[*unfolded unoverload-def*]
note *this*[**where** '*a*='*t* **and** '*b*='*u*]
note *this*[*untransferred*]
note *this*[**where** *f*=*f* **and** *g*=*g* **and** *S*=‹*S* ∩ *topspace T*› **and** *x*=*x* **and** *?open*=*openin T*
and *opena*=‹*openin U*›]

note *this*[*simplified*]
}
note $* = $ *this*[*cancel-type-definition*, *OF* ‹*topspace T* ≠ {}›, *cancel-type-definition*, *OF* ‹*topspace U* ≠ {}›]

have *2*: ‹*f* ‘ *topspace T* ⊆ *topspace U*›
by (*meson assms*(*4*) *continuous-map-image-subset-topspace*)
have *3*: ‹*g* ‘ *topspace T* ⊆ *topspace U*›
by (*simp add*: *continuous-map-image-subset-topspace g*)
have *4*: ‹*x* ∈ *topspace T*›
by (*meson assms*(*3*) *in-closure-of*)
have *5*: ‹*topological-space-ow* (*topspace T*) (*openin T*)›
by *simp*
have *6*: ‹*t2-space-ow* (*topspace U*) (*openin U*)›
by (*simp add*: *hausdorff*)
from *x* **have** ‹*x* ∈ *T closure-of* (*S* ∩ *topspace T*)›
by (*metis closure-of-restrict inf-commute*)
then have *7*: ‹*x* ∈ *closure-ow* (*topspace T*) (*openin T*) (*S* ∩ *topspace T*)›
by (*simp add*: *closure-ow-from-topology*)
have *8*: ‹*continuous-on-ow* (*topspace T*) (*topspace U*) (*openin T*) (*openin U*) (*topspace T*) *f*›
by (*meson 2 continuous-on-ow-from-topology f*)
have *9*: ‹*continuous-on-ow* (*topspace T*) (*topspace U*) (*openin T*) (*openin U*) (*topspace T*) *g*›
by (*simp add*: *3 continuous-on-ow-from-topology g*)

show *?thesis*
apply (*rule* $*$)
using *2 3 4 5 6 f-eq-g 7 8 9* **by** *auto*
qed

lemma *orthonormal-subspace-basis-exists*:
fixes *S* :: ‹'*a*::*chilbert-space set*›
assumes ‹*is-ortho-set S*› **and** *norm*: ‹⋀*x*. *x*∈*S* ⟹ *norm x* = *1*› **and** ‹*S* ⊆ *space-as-set V*›
shows ‹∃ *B*. *B* ⊇ *S* ∧ *is-ortho-set B* ∧ (∀ *x*∈*B*. *norm x* = *1*) ∧ *ccspan B* = *V*›
proof −
{
assume ∃ (*Rep* :: '*t* ⇒ '*a*) *Abs*. *type-definition Rep Abs* (*space-as-set V*)
then interpret *T*: *local-typedef* ‹*space-as-set V*› ‹*TYPE*('*t*)›
by *unfold-locales*

note *orthonormal-basis-exists*
note *this*[*unfolded unoverload-def*]
note *this*[*unoverload-type* '*a*]
note *this*[*unfolded unoverload-def*]
note *this*[**where** '*a*='*t*]
note *this*[*untransferred*]
note *this*[**where** *plus*=*plus* **and** *scaleC*=*scaleC* **and** *scaleR*=*scaleR* **and** *zero*=*0* **and** *minus*=*minus*
and *uminus*=*uminus* **and** *sgn*=*sgn* **and** *S*=*S* **and** *norm*=*norm* **and** *cinner*=*cinner* **and**

dist=dist
and *?open=‹openin (top-of-set (space-as-set V))›*
and *uniformity=‹uniformity-on (space-as-set V)›*]
note *this*[*simplified Domainp-rel-filter prod.Domainp-rel T.Domainp-cr-S*]
}
note $* = $ *this*[*cancel-type-definition*]
have *1*: *‹uniformity-on (space-as-set V)*
$\leq$ *principal (Collect (pred-prod (λx. x ∈ space-as-set V) (λx. x ∈ space-as-set V)))›*
by (*auto simp*: *uniformity-dist intro*!: *le-infI2*)
have *‹∃B∈{A. ∀x∈A. x ∈ space-as-set V}.*
S ⊆ B ∧ is-onb-ow (space-as-set V) (∗C) (+) 0 norm (openin (top-of-set (space-as-set V))) (·C) B›
apply (*rule* $*$)
using *‹S ⊆ space-as-set V› ‹is-ortho-set S›*
by (*auto simp flip*: *unoverload-def*
intro!: *complex-vector.subspace-scale real-vector.subspace-scale csubspace-is-subspace csubspace-nonempty complex-vector.subspace-add complex-vector.subspace-diff complex-vector.subspace-neg sgn-in-spaceI 1 norm*)

then obtain *B* **where** *‹B ⊆ space-as-set V›* **and** *‹S ⊆ B›*
and *is-onb*: *‹is-onb-ow (space-as-set V) (∗C) (+) 0 norm (openin (top-of-set (space-as-set V))) (·C) B›*
by *auto*

from *‹B ⊆ space-as-set V›*
have [*simp*]: *‹cspan B ∩ space-as-set V = cspan B›*
by (*smt* (*verit*) *basic-trans-rules*(*8*) *ccspan.rep-eq ccspan-leqI ccspan-superset complex-vector.span-span inf-absorb1 less-eq-ccsubspace.rep-eq*)
then have [*simp*]: *‹space-as-set V ∩ cspan B = cspan B›*
by *blast*
from *‹B ⊆ space-as-set V›*
have [*simp*]: *‹space-as-set V ∩ closure (cspan B) = closure (cspan B)›*
by (*metis Int-absorb1 ccspan.rep-eq ccspan-leqI less-eq-ccsubspace.rep-eq*)
have [*simp*]: *‹closure X ∪ X = closure X›* **for** *X* :: *‹'z::topological-space set›*
using *closure-subset* **by** *blast*

from *is-onb* **have** *‹is-ortho-set B›*
by (*auto simp*: *is-onb-ow-def unoverload-def*)

moreover from *is-onb* **have** *‹norm x = 1›* **if** *‹x ∈ B›* **for** *x*
by (*auto simp*: *is-onb-ow-def that*)

moreover from *is-onb* **have** *‹closure (cspan B) = space-as-set V›*
by (*simp add*: *is-onb-ow-def ‹B ⊆ space-as-set V›*
closure-ow-with-typeclass span-ow-on-typeclass flip: *unoverload-def*)
then have *‹ccspan B = V›*
by (*simp add*: *ccspan.abs-eq space-as-set-inverse*)

ultimately show *?thesis*

using ‹$S \subseteq B$› **by** *auto*
qed

lemma *has-sum-in-comm-additive-general*:
fixes $f :: \langle 'a \Rightarrow 'b :: comm\text{-}monoid\text{-}add\rangle$
and $g :: \langle 'b \Rightarrow 'c :: comm\text{-}monoid\text{-}add\rangle$
assumes *T0*[*simp*]: ‹$0 \in topspace\ T$› **and** *Tplus*[*simp*]: ‹$\bigwedge x\ y.\ x \in topspace\ T \Longrightarrow y \in topspace\ T \Longrightarrow x+y \in topspace\ T$›
assumes *Uplus*[*simp*]: ‹$\bigwedge x\ y.\ x \in topspace\ U \Longrightarrow y \in topspace\ U \Longrightarrow x+y \in topspace\ U$›
assumes *grange*: ‹$g\ `\ topspace\ T \subseteq topspace\ U$›
assumes *g0*: ‹$g\ 0 = 0$›
assumes *frange*: ‹$f\ `\ S \subseteq topspace\ T$›
assumes *gcont*: ‹$filterlim\ g\ (nhdsin\ U\ (g\ l))\ (atin\ T\ l)$›
assumes *gadd*: ‹$\bigwedge x\ y.\ x \in topspace\ T \Longrightarrow y \in topspace\ T \Longrightarrow g\ (x+y) = g\ x + g\ y$›
assumes *sumf*: ‹*has-sum-in T f S l*›
shows ‹*has-sum-in U* $(g\ o\ f)$ *S* $(g\ l)$›
proof −
define f' **where** ‹$f'\ x = (if\ x \in S\ then\ f\ x\ else\ 0)$› **for** x
have ‹$topspace\ T \neq \{\}$›
using *T0* **by** *blast*
then have ‹$topspace\ U \neq \{\}$›
using *grange* **by** *blast*
{
assume $\exists (Rep :: 't \Rightarrow 'b)\ Abs.\ type\text{-}definition\ Rep\ Abs\ (topspace\ T)$
then interpret *T*: *local-typedef* ‹*topspace T*› ‹$TYPE('t)$›
by *unfold-locales*
assume $\exists (Rep :: 'u \Rightarrow 'c)\ Abs.\ type\text{-}definition\ Rep\ Abs\ (topspace\ U)$
then interpret *U*: *local-typedef* ‹*topspace U*› ‹$TYPE('u)$›
by *unfold-locales*

note [[*show-types*]]
note *has-sum-comm-additive-general*
note *this*[*unfolded unoverload-def*]
note *this*[*unoverload-type 'b*, *unoverload-type 'c*]
note *this*[**where** *'b='t* **and** *'c='u* **and** *'a='a*]
note *this*[*unfolded unoverload-def*]
thm *this*[*no-vars*]
note *this*[*untransferred*]
note *this*[**where** *f=g* **and** *g=f'* **and** *zero=0* **and** *zeroa=0* **and** *plus=plus* **and** *plusa=plus* **and** *?open=*‹*openin U*› **and** *opena=*‹*openin T*› **and** *x=l* **and** *S=S* **and** *T=*‹*topspace T*›]
note *this*[*simplified*]
}
note $* = this$[*cancel-type-definition*, *OF* ‹*topspace T* $\neq \{\}$›, *cancel-type-definition*, *OF* ‹*topspace U* $\neq \{\}$›]

have $f'T$[*simp*]: ‹$f'\ x \in topspace\ T$› **for** x
using *frange f'-def* **by** *force*
have [*simp*]: ‹$l \in topspace\ T$›

```
    using sumf has-sum-in-topspace by blast
  have [simp]: ‹x ∈ topspace T ⟹ g x ∈ topspace U› for x
    using grange by auto
  have sumf'T: ‹(∑ x∈F. f' x) ∈ topspace T› if ‹finite F› for F
    using that apply induction
    by auto
  have [simp]: ‹(∑ x∈F. f x) ∈ topspace T› if ‹F ⊆ S› for F
    using that apply (induction F rule:infinite-finite-induct)
      apply auto
    by (metis Tplus f'T f'-def)
  have sum-gf: ‹(∑ x∈F. g (f' x)) = g (∑ x∈F. f' x)›
    if ‹finite F› and ‹F ⊆ S› for F
  proof −
    have ‹(∑ x∈F. g (f' x)) = (∑ x∈F. g (f x))›
      apply (rule sum.cong)
      using frange that by (auto simp: f'-def)
    also have ‹... = g (∑ x∈F. f x)›
      using ‹finite F› ‹F ⊆ S› apply induction
      using g0 frange apply auto
      apply (subst gadd)
      by (auto simp: f'-def)
    also have ‹... = g (∑ x∈F. f' x)›
      apply (rule arg-cong[where f=g])
      apply (rule sum.cong)
      using that by (auto simp: f'-def)
    finally show ?thesis
      by −
  qed
  from sumf have sumf': ‹has-sum-in T f' S l›
    apply (rule has-sum-in-cong[THEN iffD2, rotated])
    unfolding f'-def by auto
  have [simp]: ‹g l ∈ topspace U›
    using grange by auto
  from gcont have contg': ‹filterlim g (nhdsin U (g l)) (nhdsin T l ⊓ principal (topspace T −
{l}))›
    apply (rule filterlim-cong[THEN iffD1, rotated −1])
      apply (rule refl)
     apply (simp add: atin-def)
    by (auto intro!: exI simp add: eventually-atin)
  from T0 grange g0 have [simp]: ‹0 ∈ topspace U›
    by auto

  have [simp]:
    ‹comm-monoid-ow (topspace T) (+) 0›
    ‹comm-monoid-ow (topspace U) (+) 0›
    by (simp-all add: comm-monoid-ow-def abel-semigroup-ow-def
        semigroup-ow-def plus-ow-def semigroup-ow-axioms-def
        comm-monoid-ow-axioms-def Groups.add-ac abel-semigroup-ow-axioms-def)
```

have ‹*has-sum-ow* (*topspace U*) (+) *0* (*openin U*) (*g* ∘ *f'*) *S* (*g l*)›
apply (*rule* ∗)
by (*auto simp*: *topological-space-ow-from-topology sum-gf sumf' sum-ud*[*symmetric*] *at-within-ow-topology has-sum-ow-topology contg' sumf'T*)

then have ‹*has-sum-in U* (*g* ∘ *f'*) *S* (*g l*)›
apply (*rule has-sum-ow-topology*[*THEN iffD1*, *rotated* −*1*])
by *simp-all*
then have ‹*has-sum-in U* (*g* ∘ *f'*) *S* (*g l*)›
by *simp*
then show *?thesis*
apply (*rule has-sum-in-cong*[*THEN iffD1*, *rotated*])
unfolding *f'-def* **using** *frange grange* **by** *auto*
qed

lemma *has-sum-in-comm-additive*:
fixes *f* :: ‹'*a* ⇒ '*b* :: *ab-group-add*›
and *g* :: ‹'*b* ⇒ '*c* :: *ab-group-add*›
assumes ‹*topspace T* = *UNIV*› **and** ‹*topspace U* = *UNIV*›
assumes ‹*Modules.additive g*›
assumes *gcont*: ‹*continuous-map T U g*›
assumes *sumf*: ‹*has-sum-in T f S l*›
shows ‹*has-sum-in U* (*g o f*) *S* (*g l*)›
apply (*rule has-sum-in-comm-additive-general*[**where** *T*=*T* **and** *U*=*U*])
using *assms*
by (*auto simp*: *additive.zero Modules.additive-def intro*!: *continuous-map-is-continuous-at-point*)

8 Stuff relying on the above lifting

definition ‹*some-onb-of X* = (*SOME B. is-ortho-set B* ∧ (∀ *b*∈*B. norm b* = *1*) ∧ *ccspan B* = *X*)›

lemma
fixes *X* :: ‹'*a*::*chilbert-space ccsubspace*›
shows *some-onb-of-is-ortho-set*[*iff*]: ‹*is-ortho-set* (*some-onb-of X*)›
and *some-onb-of-norm1*: ‹*b* ∈ *some-onb-of X* ⟹ *norm b* = *1*›
and *some-onb-of-ccspan*[*simp*]: ‹*ccspan* (*some-onb-of X*) = *X*›
proof −
let *?P* = ‹λ*B. is-ortho-set B* ∧ (∀ *b*∈*B. norm b* = *1*) ∧ *ccspan B* = *X*›
have ‹*Ex ?P*›
using *orthonormal-subspace-basis-exists*[**where** *S*=‹{}› **and** *V*=*X*]
by *auto*
then have ‹*?P* (*some-onb-of X*)›
by (*simp add*: *some-onb-of-def verit-sko-ex*)
then show *is-ortho-set-some-onb-of*: ‹*is-ortho-set* (*some-onb-of X*)›
and ‹*b* ∈ *some-onb-of X* ⟹ *norm b* = *1*›
and ‹*ccspan* (*some-onb-of X*) = *X*›

```
    by auto
qed

lemma ccsubspace-as-whole-type:
  fixes X :: ‹'a::chilbert-space ccsubspace›
  assumes ‹X ≠ 0›
  shows ‹let 'b::type = some-onb-of X in
        ∃ U::'b ell2 ⇒CL 'a. isometry U ∧ U *S ⊤ = X›
proof with-type-intro
  show ‹some-onb-of X ≠ {}›
    using some-onb-of-ccspan[of X] assms
    by (auto simp del: some-onb-of-ccspan)
  fix Rep :: ‹'b ⇒ 'a› and Abs
  assume ‹bij-betw Rep UNIV (some-onb-of X)›
  then interpret type-definition Rep ‹inv Rep› ‹some-onb-of X›
    by (simp add: type-definition-bij-betw-iff)
  define U where ‹U = cblinfun-extension (range ket) (Rep o inv ket)›
  have [simp]: ‹Rep i •C Rep j = 0› if ‹i ≠ j› for i j
    using Rep some-onb-of-is-ortho-set[unfolded is-ortho-set-def] that
    by (smt (verit) Rep-inverse)
  moreover have [simp]: ‹norm (Rep i) = 1› for i
    using Rep[of i] some-onb-of-norm1
    by auto
  ultimately have ‹cblinfun-extension-exists (range ket) (Rep o inv ket)›
    apply (rule-tac cblinfun-extension-exists-ortho)
    by auto
  then have U-ket[simp]: ‹U (ket i) = Rep i› for i
    by (auto simp: cblinfun-extension-apply U-def)
  have ‹isometry U›
    apply (rule orthogonal-on-basis-is-isometry[where B=‹range ket›])
    by (auto simp: cinner-ket simp flip: cnorm-eq-1)
  moreover have ‹U *S ccspan (range ket) = X›
    apply (subst cblinfun-image-ccspan)
    by (simp add: Rep-range image-image)
  ultimately show ‹∃ U :: 'b ell2 ⇒CL 'a. isometry U ∧ U *S ⊤ = X›
    by auto
qed

lemma some-onb-of-0[simp]: ‹some-onb-of (0 :: 'a::chilbert-space ccsubspace) = {}›
proof −
  have no0: ‹0 ∉ some-onb-of (0 :: 'a ccsubspace)›
    using some-onb-of-norm1
    by fastforce
  have ‹ccspan (some-onb-of 0) = (0 :: 'a ccsubspace)›
    by simp
  then have ‹some-onb-of 0 ⊆ space-as-set (0 :: 'a ccsubspace)›
    by (metis ccspan-superset)
  also have ‹... = {0}›
    by simp
```

finally show *?thesis*
using *no0*
by *blast*
qed

lemma *some-onb-of-finite-dim*:
fixes S :: ‹'*a*::*chilbert-space ccsubspace*›
assumes ‹*finite-dim-ccsubspace S*›
shows ‹*finite* (*some-onb-of S*)›
proof −
from *assms* **obtain** C **where** *CS*: ‹*cspan C* = *space-as-set S*› **and** ‹*finite C*›
by (*meson cfinite-dim-subspace-has-basis csubspace-space-as-set finite-dim-ccsubspace.rep-eq*)
then show ‹*finite* (*some-onb-of S*)›
using *ccspan-superset complex-vector.independent-span-bound is-ortho-set-cindependent* **by** *fastforce*
qed

lemma *some-onb-of-in-space*[*iff*]:
fixes S :: ‹'*a*::*chilbert-space ccsubspace*›
shows ‹*some-onb-of S* ⊆ *space-as-set S*›
using *ccspan-superset* **by** *fastforce*

lemma *sum-some-onb-of-butterfly*:
fixes S :: ‹'*a*::*chilbert-space ccsubspace*›
assumes ‹*finite-dim-ccsubspace S*›
shows ‹(∑ *x*∈*some-onb-of S*. *butterfly x x*) = *Proj S*›
proof −
obtain B **where** *onb-S-in-B*: ‹*some-onb-of S* ⊆ *B*› **and** ‹*is-onb B*›
apply *atomize-elim*
apply (*rule orthonormal-basis-exists*)
by (*simp-all add*: *some-onb-of-norm1*)
have *S-ccspan*: ‹*S* = *ccspan* (*some-onb-of S*)›
by *simp*

show *?thesis*
proof (*rule cblinfun-eq-gen-eqI*[**where** *G*=*B*])
show ‹*ccspan B* = ⊤›
using ‹*is-onb B*› *is-onb-def* **by** *blast*
fix b **assume** ‹*b* ∈ *B*›
show ‹(∑ *x*∈*some-onb-of S*. *selfbutter x*) *∗*$_V$ *b* = *Proj S ∗*$_V$ *b*›
proof (*cases* ‹*b* ∈ *some-onb-of S*›)
case *True*
have ‹(∑ *x*∈*some-onb-of S*. *selfbutter x*) *∗*$_V$ *b* = (∑ *x*∈*some-onb-of S*. *selfbutter x ∗*$_V$ *b*)›
using *cblinfun.sum-left* **by** *blast*
also have ‹. . . = *b*›
apply (*subst sum-single*[**where** *i*=*b*])
using *True* **apply** (*auto intro*!: *simp add*: *assms some-onb-of-finite-dim*)

```
      using is-ortho-set-def apply fastforce
      using cnorm-eq-1 some-onb-of-norm1 by force
    also have ‹... = Proj S *_V b›
      apply (rule Proj-fixes-image[symmetric])
      using True some-onb-of-in-space by blast
    finally show ?thesis
      by −
  next
    case False
    have *: ‹is-orthogonal x b› if ‹x ∈ some-onb-of S› and ‹x ≠ 0› for x
    proof −
      have ‹x ∈ B›
        using onb-S-in-B that(1) by fastforce
      moreover note ‹b ∈ B›
      moreover have ‹x ≠ b›
        using False that(1) by blast
      moreover note ‹is-onb B›
      ultimately show ‹is-orthogonal x b›
        by (simp add: is-onb-def is-ortho-set-def)
    qed
    have ‹(∑x∈some-onb-of S. selfbutter x) *_V b = (∑x∈some-onb-of S. selfbutter x *_V b)›
      using cblinfun.sum-left by blast
    also have ‹... = 0›
      by (auto intro!: sum.neutral simp: * )
    also have ‹... = Proj S *_V b›
      apply (rule Proj-0-compl[symmetric])
      apply (subst S-ccspan)
      apply (rule mem-ortho-ccspanI)
      using * cinner-zero-right is-orthogonal-sym by blast
    finally show ?thesis
      by −
  qed
 qed
qed


lemma cdim-infinite-0:
 assumes ‹¬ cfinite-dim S›
 shows ‹cdim S = 0›
proof −
 from assms have not-fin-cspan: ‹¬ cfinite-dim (cspan S)›
  using cfinite-dim-def cfinite-dim-subspace-has-basis complex-vector.span-superset by fastforce
 obtain B where ‹cindependent B› and ‹cspan S = cspan B›
  using csubspace-has-basis by blast
 with not-fin-cspan have ‹infinite B›
  by auto
 then have ‹card B = 0›
  by force
 have ‹cdim (cspan S) = 0›
```

apply (*rule complex-vector.dim-unique*[*of B*])
apply (*auto intro*!: *simp add*: ‹*cspan S = cspan B*› *complex-vector.span-superset*)
using ‹*cindependent B*› ‹*card B = 0*› **by** *auto*
then show *?thesis*
by *simp*
qed

lemma *some-onb-of-card*:
fixes *S* :: ‹'*a*::*chilbert-space ccsubspace*›
shows ‹*card* (*some-onb-of S*) = *cdim* (*space-as-set S*)›
proof (*cases* ‹*finite-dim-ccsubspace S*›)
case *True*
show *?thesis*
apply (*rule complex-vector.dim-eq-card*[*symmetric*])
apply (*auto simp*: *is-ortho-set-cindependent*)
apply (*metis True ccspan-finite some-onb-of-ccspan complex-vector.span-clauses*(*1*) *some-onb-of-finite-dim*)
by (*metis True ccspan-finite some-onb-of-ccspan complex-vector.span-eq-iff csubspace-space-as-set some-onb-of-finite-dim*)
next
case *False*
then have ‹*cdim* (*space-as-set S*) = *0*›
by (*simp add*: *cdim-infinite-0 finite-dim-ccsubspace.rep-eq*)
moreover from *False* **have** ‹*infinite* (*some-onb-of S*)›
using *ccspan-finite-dim* **by** *fastforce*
ultimately show *?thesis*
by *simp*
qed

end

9 *Eigenvalues* – Material related to eigenvalues and eigenspaces

theory *Eigenvalues*
imports
Weak-Operator-Topology
Misc-Tensor-Product-TTS
begin

definition *normal-op* :: ‹('*a*::*chilbert-space* ⇒$_{CL}$ '*a*) ⇒ *bool*› **where**
‹*normal-op A* ⟷ *A* o_{CL} *A*∗ = *A*∗ o_{CL} *A*›

definition *eigenvalues* :: ‹('*a*::*complex-normed-vector* ⇒$_{CL}$ '*a*) ⇒ *complex set*› **where**
‹*eigenvalues a* = {*x*. *eigenspace x a* ≠ *0*}›

definition *invariant-subspace* :: ‹'*a*::*complex-inner ccsubspace* ⇒ ('*a* ⇒$_{CL}$ '*a*) ⇒ *bool*› **where**
‹*invariant-subspace S A* ⟷ *A* ∗$_S$ *S* ≤ *S*›

lemma *invariant-subspaceI*: ‹$A *_S S \leq S \Longrightarrow$ *invariant-subspace S A*›
by (*simp add*: *invariant-subspace-def*)

definition *reducing-subspace* :: ‹'*a*::*complex-inner ccsubspace* ⇒ ('*a* ⇒$_{CL}$ '*a*) ⇒ *bool*› **where**
‹*reducing-subspace S A* ⟷ *invariant-subspace S A* ∧ *invariant-subspace* (−*S*) *A*›

lemma *reducing-subspaceI*: ‹$A *_S S \leq S \Longrightarrow A *_S (-S) \leq -S \Longrightarrow$ *reducing-subspace S A*›
by (*simp add*: *reducing-subspace-def invariant-subspace-def*)

lemma *reducing-subspace-ortho*[*simp*]: ‹*reducing-subspace* (−*S*) *A* ⟷ *reducing-subspace S A*›
for *S* :: ‹'*a*::*chilbert-space ccsubspace*›
by (*auto simp*: *reducing-subspace-def*)

lemma *invariant-subspace-bot*[*simp*]: ‹*invariant-subspace* ⊥ *A*›
by (*simp add*: *invariant-subspaceI*)

lemma *invariant-subspace-top*[*simp*]: ‹*invariant-subspace* ⊤ *A*›
by (*simp add*: *invariant-subspaceI*)

lemma *reducing-subspace-bot*[*simp*]: ‹*reducing-subspace* ⊥ *A*›
by (*metis cblinfun-image-bot eq-refl orthogonal-bot orthogonal-spaces-leq-compl reducing-subspaceI*)

lemma *reducing-subspace-top*[*simp*]: ‹*reducing-subspace* ⊤ *A*›
by (*simp add*: *reducing-subspace-def*)

lemma *kernel-uminus*[*simp*]: *kernel* (−*A*) = *kernel A*
for *a* :: *complex* **and** *A* :: (-,-) *cblinfun*
by *transfer auto*

lemma *kernel-scaleC'*: *kernel* (*a* ⋆$_C$ *A*) = (*if a* = *0 then* ⊤ *else kernel A*)
for *a* :: *complex* **and** *A* :: (-,-) *cblinfun*
by (*cases a* = *0*) *auto*

lemma *eigenvalues-0*[*simp*]: ‹*eigenvalues* (*0* :: '*a*::{*not-singleton*,*complex-normed-vector*} ⇒$_{CL}$ '*a*) = {*0*}›
by (*auto simp*: *eigenvalues-def eigenspace-def kernel-scaleC'*)

lemma *nonzero-ccsubspace-contains-unit-vector*:
assumes ‹*S* ≠ *0*›
shows ‹∃ψ. ψ ∈ *space-as-set S* ∧ *norm* ψ = *1*›
proof −
from *assms*
obtain ψ **where** ψ: ‹ψ ∈ *space-as-set S*› ‹ψ ≠ *0*›
by *transfer* (*auto simp*: *complex-vector.subspace-0*)
have ‹*sgn* ψ ∈ *space-as-set S*›
using ψ **by** (*simp add*: *complex-vector.subspace-scale scaleR-scaleC sgn-div-norm*)
moreover have ‹*norm* (*sgn* ψ) = *1*›
by (*simp add*: ‹ψ ≠ *0*› *norm-sgn*)

ultimately show *?thesis*
by *auto*
qed

lemma *unit-eigenvector-ex*:
assumes ‹$x \in$ *eigenvalues a*›
shows ‹$\exists h.$ *norm h* $= 1 \wedge$ *a h* $= x *_C$ *h*›
proof −
from *assms* **have** ‹*eigenspace x a* $\neq 0$›
by (*simp add*: *eigenvalues-def*)
then obtain ψ **where** ψ*-ev*: ‹$\psi \in$ *space-as-set* (*eigenspace x a*)› **and** ‹$\psi \neq 0$›
using *nonzero-ccsubspace-contains-unit-vector* **by** *force*
define *h* **where** ‹*h* = *sgn* ψ›
with ‹$\psi \neq 0$› **have** ‹*norm h* = *1*›
by (*simp add*: *norm-sgn*)
from ψ*-ev* **have** ‹*h* $\in$ *space-as-set* (*eigenspace x a*)›
by (*simp add*: *h-def sgn-in-spaceI*)
then have ‹*a* $*_V$ *h* = *x* $*_C$ *h*›
unfolding *eigenspace-def*
by (*transfer′ fixing*: *x*) *simp*
with ‹*norm h* = *1*› **show** *?thesis*
by *auto*
qed

lemma *eigenvalue-norm-bound*:
assumes ‹*e* $\in$ *eigenvalues a*›
shows ‹*norm e* $\leq$ *norm a*›
proof −
from *assms* **obtain** *h* **where** ‹*norm h* = *1*› **and** *ah-eh*: ‹*a h* = *e* $*_C$ *h*›
using *unit-eigenvector-ex* **by** *blast*
have ‹*cmod e* = *norm* (*e* $*_C$ *h*)›
by (*simp add*: ‹*norm h* = *1*›)
also have ‹... = *norm* (*a h*)›
using *ah-eh* **by** *presburger*
also have ‹... $\leq$ *norm a*›
by (*metis* ‹*norm h* = *1*› *cblinfun.real.bounded-linear-right mult-cancel-left1 norm-cblinfun.rep-eq onorm*)
finally show ‹*cmod e* $\leq$ *norm a*›
by −
qed

lemma *eigenvalue-selfadj-real*:
assumes ‹*e* $\in$ *eigenvalues a*›
assumes ‹*selfadjoint a*›
shows ‹*e* $\in \mathbb{R}$›
proof −
from *assms* **obtain** *h* **where** ‹*norm h* = *1*› **and** *ah-eh*: ‹*a h* = *e* $*_C$ *h*›
using *unit-eigenvector-ex* **by** *blast*

have ‹$e = h \cdot_C (e *_C h)$›
by (*metis* ‹*norm h = 1*› *cinner-simps(6) mult-cancel-left1 norm-one one-cinner-one power2-norm-eq-cinner power2-norm-eq-cinner*)
also have ‹$\ldots = h \cdot_C a\ h$›
by (*simp add*: *ah-eh*)
also from *assms(2)* **have** ‹$\ldots \in \mathbb{R}$›
using *cinner-hermitian-real selfadjoint-def* **by** *blast*
finally show ‹$e \in \mathbb{R}$›
by −
qed

lemma *is-Sup-imp-ex-tendsto*:
fixes X :: ‹'*a*::{*linorder-topology*,*first-countable-topology*} *set*›
assumes *sup*: ‹*is-Sup X l*›
assumes ‹$X \neq \{\}$›
shows ‹$\exists f.$ *range* $f \subseteq X \wedge f \longrightarrow l$›
proof (*cases* ‹$\exists x.\ x < l$›)
case *True*
obtain A :: ‹*nat* ⇒ '*a set*› **where** *openA*: ‹*open* (*A n*)› **and** *lA*: ‹$l \in A\ n$›
and *fl*: ‹$(\bigwedge n.\ f\ n \in A\ n) \Longrightarrow f \longrightarrow l$› **for** *n f*
by (*rule Topological-Spaces.countable-basis*[*of l*]) *blast*
obtain *f* **where** *fAX*: ‹$f\ n \in A\ n \cap X$› **for** *n*
proof (*atomize-elim*, *intro choice allI*)
fix *n* :: *nat*
from *True* **obtain** *x* **where** ‹$x < l$›
by *blast*
from *open-left*[*OF openA lA this*]
obtain *b* **where** ‹$b < l$› **and** *bl-A*: ‹$\{b<..l\} \subseteq A\ n$›
by *blast*
from *sup* ‹$b < l$› **obtain** *x* **where** ‹$x \in X$› **and** ‹$x > b$›
by (*meson is-Sup-def leD leI*)
from ‹$x \in X$› *sup* **have** ‹$x \leq l$›
by (*simp add*: *is-Sup-def*)
from ‹$x \leq l$› **and** ‹$x > b$› **and** *bl-A*
have ‹$x \in A\ n$›
by *fastforce*
with ‹$x \in X$›
show ‹$\exists x.\ x \in A\ n \cap X$›
by *blast*
qed
with *fl* **have** ‹$f \longrightarrow l$›
by *auto*
moreover from *fAX* **have** ‹*range* $f \subseteq X$›
by *auto*
ultimately show *?thesis*
by *blast*
next
case *False*
from ‹$X \neq \{\}$› **obtain** *x* **where** ‹$x \in X$›

```
    by blast
  with ‹is-Sup X l› have ‹x ≤ l›
    by (simp add: is-Sup-def)
  with False have ‹x = l›
    using basic-trans-rules(17) by auto
  with ‹x ∈ X› have ‹l ∈ X›
    by simp
  define f where ‹f n = l› for n :: nat
  then have ‹f ⟶ l›
    by (auto intro!: simp: f-def[abs-def])
  moreover from ‹l ∈ X› have ‹range f ⊆ X›
    by (simp add: f-def)
  ultimately show ?thesis
    by blast
qed

lemma eigenvaluesI:
  assumes ‹A *V h = e *C h›
  assumes ‹h ≠ 0›
  shows ‹e ∈ eigenvalues A›
proof −
  from assms have ‹h ∈ space-as-set (eigenspace e A)›
    by (simp add: eigenspace-def kernel.rep-eq cblinfun.diff-left)
  moreover from ‹h ≠ 0› have ‹h ∉ space-as-set ⊥›
    by transfer simp
  ultimately have ‹eigenspace e A ≠ ⊥›
    by fastforce
  then show ?thesis
    by (simp add: eigenvalues-def)
qed

lemma tendsto-diff-const-left-rewrite:
  fixes c d :: ‹'a::{topological-group-add, ab-group-add}›
  assumes ‹((λx. f x) ⟶ c − d) F›
  shows ‹((λx. c − f x) ⟶ d) F›
  by (auto intro!: assms tendsto-eq-intros)

lemma not-not-singleton-no-eigenvalues:
  fixes a :: ‹'a::complex-normed-vector ⇒CL 'a›
  assumes ‹¬ class.not-singleton TYPE('a)›
  shows ‹eigenvalues a = {}›
proof (rule equals0I)
  fix e assume ‹e ∈ eigenvalues a›
  then have ‹eigenspace e a ≠ ⊥›
    by (simp add: eigenvalues-def)
  then obtain h where ‹norm h = 1› and ‹h ∈ space-as-set (eigenspace e a)›
    using nonzero-ccsubspace-contains-unit-vector by auto
  from assms have ‹h = 0›
    by (rule not-not-singleton-zero)
```

with ‹*norm h = 1*›
show *False*
by *simp*
qed

lemma *cblinfun-cinner-eq0I*:
fixes *a* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*a*›
assumes ‹⋀*h*. *h* •$_C$ *a h* = *0*›
shows ‹*a* = *0*›
by (*rule cblinfun-cinner-eqI*) (*use assms* **in** *simp*)

lemma *normal-op-iff-adj-same-norms*:
— [2], Proposition II.2.16
fixes *a* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*a*›
shows ‹*normal-op a* ⟷ (∀*h*. *norm* (*a h*) = *norm* ((*a**) *h*))›
proof −
have *aux*: ‹(⋀*h*. *a h* = *b h*) ==> (∀*h*. *a h* = (*0*::*complex*)) ⟷ (∀*h*. *b h* = (*0*::*real*))› **for** *a* :: ‹'*a* ⇒ *complex*› **and** *b* :: ‹'*a* ⇒ *real*›
by *simp*
have ‹*normal-op a* ⟷ (*a** *o*$_{CL}$ *a*) − (*a o*$_{CL}$ *a**) = *0*›
using *normal-op-def* **by** *force*
also have ‹... ⟷ (∀*h*. *h* •$_C$ ((*a** *o*$_{CL}$ *a*) − (*a o*$_{CL}$ *a**)) *h* = *0*)›
by (*auto intro*!: *cblinfun-cinner-eqI simp*: *cblinfun.diff-left cinner-diff-right simp flip*: *cblinfun-apply-cblinfun-compose*)
also have ‹... ⟷ (∀*h*. (*norm* (*a h*))^2 − (*norm* ((*a**) *h*))^2 = *0*)›
proof (*rule aux*)
fix *h*
have ‹(*norm* (*a* *$_V$ *h*))^2 − (*norm* (*a** *$_V$ *h*))^2
= (*a* *$_V$ *h*) •$_C$ (*a* *$_V$ *h*) − (*a** *$_V$ *h*) •$_C$ (*a** *$_V$ *h*)›
by (*simp add*: *of-real-diff flip*: *cdot-square-norm of-real-power*)
also have ‹... = *h* •$_C$ ((*a** *o*$_{CL}$ *a*) − (*a o*$_{CL}$ *a**)) *h*›
by (*simp add*: *cblinfun.diff-left cinner-diff-right cinner-adj-left cinner-adj-right flip*: *cinner-adj-left*)
finally show ‹*h* •$_C$ ((*a** *o*$_{CL}$ *a*) − (*a o*$_{CL}$ *a**)) *h* = (*norm* (*a* *$_V$ *h*))^2 − (*norm* (*a** *$_V$ *h*))^2›
by *simp*
qed
also have ‹... ⟷ (∀*h*. *norm* (*a h*) = *norm* ((*a**) *h*))›
by *simp*
finally show *?thesis*.
qed

lemma *normal-op-same-eigenspace-as-adj*:
— Shown inside the proof of [2, Proposition II.5.6]
assumes ‹*normal-op a*›
shows ‹*eigenspace l a* = *eigenspace* (*cnj l*) (*a**)›
proof −
from ‹*normal-op a*›
have ‹*normal-op* (*a* − *l* *$_C$ *id-cblinfun*)›

```
    by (auto intro!: simp: normal-op-def cblinfun-compose-minus-left
        cblinfun-compose-minus-right adj-minus scaleC-diff-right)
  then have ∗: ‹norm ((a − l ∗C id-cblinfun) h) = norm (((a − l ∗C id-cblinfun)∗) h)› for h
    using normal-op-iff-adj-same-norms by blast
  show ?thesis
  proof (rule ccsubspace-eqI)
    fix h
    have ‹h ∈ space-as-set (eigenspace l a) ⟷ norm ((a − l ∗C id-cblinfun) h) = 0›
      by (simp add: eigenspace-def kernel-member-iff)
    also have ‹. . . ⟷ norm (((a∗) − cnj l ∗C id-cblinfun) h) = 0›
      by (simp add: ∗ adj-minus)
    also have ‹. . . ⟷ h ∈ space-as-set (eigenspace (cnj l) (a∗))›
      by (simp add: eigenspace-def kernel-member-iff)
    finally show ‹h ∈ space-as-set (eigenspace l a) ⟷ h ∈ space-as-set (eigenspace (cnj l)
(a∗))›.
  qed
qed

lemma normal-op-adj-eigenvalues:
  assumes ‹normal-op a›
  shows ‹eigenvalues (a∗) = cnj ‘ eigenvalues a›
  by (auto intro!: complex-cnj-cnj[symmetric] image-eqI
      simp: eigenvalues-def assms normal-op-same-eigenspace-as-adj)

lemma invariant-subspace-iff-PAP:
  — [2], Proposition II.3.7 (b)
  ‹invariant-subspace S A ⟷ Proj S oCL A oCL Proj S = A oCL Proj S›
proof −
  define S′ where ‹S′ = space-as-set S›
  have ‹invariant-subspace S A ⟷ (∀ h∈S′. A h ∈ S′)›
  proof safe
    fix h assume A: invariant-subspace S A and h: h ∈ S′
    from h have A ∗V h ∈ space-as-set (A ∗S S)
      using cblinfun-apply-in-image′[of h S A] unfolding S′-def by auto
    also have space-as-set (A ∗S S) ⊆ S′
      using A unfolding S′-def invariant-subspace-def less-eq-ccsubspace-def by auto
    finally show A ∗V h ∈ S′ .
  next
    assume ∗: ∀ h∈S′. A ∗V h ∈ S′
    hence A ∗S S ≤ S
      unfolding S′-def using cblinfun-image-less-eqI by blast
    thus invariant-subspace S A
      unfolding invariant-subspace-def less-eq-ccsubspace-def map-fun-def o-def id-def .
  qed
  also have ‹. . . ⟷ (∀ h. A ∗V Proj S ∗V h ∈ S′)›
    by (metis (no-types, lifting) Proj-fixes-image Proj-range S′-def cblinfun-apply-in-image)
  also have ‹. . . ⟷ (∀ h. Proj S ∗V A ∗V Proj S ∗V h = A ∗V Proj S ∗V h)›
    using Proj-fixes-image S′-def space-as-setI-via-Proj by blast
  also have ‹. . . ⟷ Proj S oCL A oCL Proj S = A oCL Proj S›
```

by (*auto intro*!: *cblinfun-eqI simp*:
simp flip: *cblinfun-apply-cblinfun-compose cblinfun-compose-assoc*)
finally show *?thesis*
by −
qed

lemma *reducing-iff-PA*:
— [2], Proposition II.3.7 (e)
‹*reducing-subspace S A* ⟷ *Proj S* o_{CL} *A* = *A* o_{CL} *Proj S*›
proof (*rule iffI*)
assume *red*: ‹*reducing-subspace S A*›
define *P* **where** ‹*P* = *Proj S*›
from *red* **have** *AP*: ‹*P* o_{CL} *A* o_{CL} *P* = *A* o_{CL} *P*›
by (*simp add*: *invariant-subspace-iff-PAP reducing-subspace-def P-def*)
from *red* **have** ‹*reducing-subspace* (− *S*) *A*›
by *simp*
then have ‹(*id-cblinfun* − *P*) o_{CL} *A* o_{CL} (*id-cblinfun* − *P*) = *A* o_{CL} (*id-cblinfun* − *P*)›
using *invariant-subspace-iff-PAP*[*of* ‹− *S*›] *reducing-subspace-def P-def Proj-ortho-compl*
by *metis*
then have ‹*P* o_{CL} *A* = *P* o_{CL} *A* o_{CL} *P*›
by (*simp add*: *cblinfun-compose-minus-left cblinfun-compose-minus-right*)
with *AP* **show** ‹*P* o_{CL} *A* = *A* o_{CL} *P*›
by *simp*
next
define *P* **where** ‹*P* = *Proj S*›
assume ‹*P* o_{CL} *A* = *A* o_{CL} *P*›
then have ‹*P* o_{CL} *A* o_{CL} *P* = *A* o_{CL} *P* o_{CL} *P*›
by *simp*
then have ‹*P* o_{CL} *A* o_{CL} *P* = *A* o_{CL} *P*›
by (*metis P-def Proj-idempotent cblinfun-assoc-left*(*1*))
then have ‹*invariant-subspace S A*›
by (*simp add*: *P-def invariant-subspace-iff-PAP*)
have ‹(*id-cblinfun* − *P*) o_{CL} *A* o_{CL} (*id-cblinfun* − *P*) = *A* o_{CL} (*id-cblinfun* − *P*)›
by (*metis* (*no-types*, *opaque-lifting*) *P-def Proj-idempotent Proj-ortho-compl* ‹*P* o_{CL} *A* = *A* o_{CL} *P*› *cblinfun-assoc-left*(*1*) *cblinfun-compose-id-left cblinfun-compose-minus-left cblinfun-compose-minus-right*)
then have ‹*invariant-subspace* (− *S*) *A*›
by (*simp add*: *P-def Proj-ortho-compl invariant-subspace-iff-PAP*)
with ‹*invariant-subspace S A*›
show ‹*reducing-subspace S A*›
using *reducing-subspace-def* **by** *blast*
qed

lemma *reducing-iff-also-adj-invariant*:
— [2], Proposition II.3.7 (g)
shows ‹*reducing-subspace S A* ⟷ *invariant-subspace S A* ∧ *invariant-subspace S* (*A*∗)›
proof (*intro iffI conjI*; (*erule conjE*)*?*)
assume ‹*invariant-subspace S A*› **and** ‹*invariant-subspace S* (*A*∗)›
have ‹*invariant-subspace* (− *S*) *A*›

```
  proof (intro invariant-subspaceI cblinfun-image-less-eqI)
    fix h assume ‹h ∈ space-as-set (− S)›
    show ‹A *V h ∈ space-as-set (− S)›
    proof (unfold uminus-ccsubspace.rep-eq, intro orthogonal-complementI)
      fix g assume ‹g ∈ space-as-set S›
      with ‹invariant-subspace S (A*)› have ‹(A*) g ∈ space-as-set S›
        by (metis Proj-compose-cancelI Proj-range cblinfun-apply-in-image' cblinfun-fixes-range
invariant-subspace-def space-as-setI-via-Proj)
      have ‹A h •C g = h •C (A*) g›
        by (simp add: cinner-adj-right)
      also from ‹(A*) g ∈ space-as-set S› and ‹h ∈ space-as-set (− S)›
      have ‹... = 0›
        using orthogonal-spaces-def orthogonal-spaces-leq-compl by blast
      finally show ‹A h •C g = 0›
        by blast
    qed
  qed
  with ‹invariant-subspace S A›
  show ‹reducing-subspace S A›
    using reducing-subspace-def by blast
next
  assume ‹reducing-subspace S A›
  then show ‹invariant-subspace S A›
    using reducing-subspace-def by blast
  show ‹invariant-subspace S (A*)›
     by (metis ‹reducing-subspace S A› adj-Proj adj-cblinfun-compose reducing-iff-PA reduc-
ing-subspace-def)
qed

lemma eigenspace-is-reducing:
  — [2], Proposition II.5.6
  assumes ‹normal-op a›
  shows ‹reducing-subspace (eigenspace l a) a›
proof (unfold reducing-iff-also-adj-invariant invariant-subspace-def,
    intro conjI cblinfun-image-less-eqI subsetI)
  fix h
  assume h-eigen: ‹h ∈ space-as-set (eigenspace l a)›
  then have ‹a h = l *C h›
    by (simp add: eigenspace-memberD)
  also have ‹... ∈ space-as-set (eigenspace l a)›
    by (simp add: Proj-fixes-image cblinfun.scaleC-right h-eigen space-as-setI-via-Proj)
  finally show ‹a h ∈ space-as-set (eigenspace l a)›.
next
  fix h
  assume h-eigen: ‹h ∈ space-as-set (eigenspace l a)›
  then have ‹h ∈ space-as-set (eigenspace (cnj l) (a*))›
    by (simp add: assms normal-op-same-eigenspace-as-adj)
  then have ‹(a*) h = cnj l *C h›
    by (simp add: eigenspace-memberD)
```

also have ‹... ∈ *space-as-set* (*eigenspace l a*)›
by (*simp add*: *Proj-fixes-image cblinfun.scaleC-right h-eigen space-as-setI-via-Proj*)
finally show ‹(*a**) *h* ∈ *space-as-set* (*eigenspace l a*)›**.**
qed

lemma *invariant-subspace-Inf*:
assumes ‹⋀*S*. *S* ∈ *M* ⟹ *invariant-subspace S a*›
shows ‹*invariant-subspace* (⨅ *M*) *a*›
proof (*rule invariant-subspaceI*)
have ‹*a* *$_S$ ⨅ *M* ≤ (⨅ *S*∈*M*. *a* *$_S$ *S*)›
using *cblinfun-image-INF-leq*[**where** *U*=*a* **and** *V*=*id* **and** *X*=*M*] **by** *simp*
also have ‹... ≤ (⨅ *S*∈*M*. *S*)›
by (*rule INF-superset-mono*, *simp*) (*use assms* **in** ‹*auto simp*: *invariant-subspace-def*›)
also have ‹... = ⨅ *M*›
by *simp*
finally show ‹*a* *$_S$ ⨅ *M* ≤ ⨅ *M*› **.**
qed

lemma *invariant-subspace-INF*:
assumes ‹⋀*x*. *x* ∈ *X* ⟹ *invariant-subspace* (*S x*) *a*›
shows ‹*invariant-subspace* (⨅ *x*∈*X*. *S x*) *a*›
by (*smt* (*verit*) *assms imageE invariant-subspace-Inf*)

lemma *invariant-subspace-Sup*:
assumes ‹⋀*S*. *S* ∈ *M* ⟹ *invariant-subspace S a*›
shows ‹*invariant-subspace* (⨆ *M*) *a*›
proof −
have ∗: ‹*a* ‘ *cspan* (⋃ *S*∈*M*. *space-as-set S*) ⊆ *space-as-set* (⨆ *M*)›
proof (*rule image-subsetI*)
fix *h*
assume ‹*h* ∈ *cspan* (⋃ *S*∈*M*. *space-as-set S*)›
then obtain *F r* **where** ‹*h* = (∑ *g*∈*F*. *r g* *$_C$ *g*)› **and** *F-in-union*: ‹*F* ⊆ (⋃ *S*∈*M*. *space-as-set S*)›
by (*auto intro*!: *simp*: *complex-vector.span-explicit*)
then have ‹*a h* = (∑ *g*∈*F*. *r g* *$_C$ *a g*)›
by (*simp add*: *cblinfun.scaleC-right cblinfun.sum-right*)
also have ‹... ∈ *space-as-set* (⨆ *M*)›
proof (*rule complex-vector.subspace-sum*)
show ‹*csubspace* (*space-as-set* (⨆ *M*))›
by *simp*
fix *g* **assume** ‹*g* ∈ *F*›
then obtain *S* **where** ‹*S* ∈ *M*› **and** ‹*g* ∈ *space-as-set S*›
using *F-in-union* **by** *auto*
from *assms*[*OF* ‹*S* ∈ *M*›] ‹*g* ∈ *space-as-set S*›
have ‹*a g* ∈ *space-as-set S*›
by (*simp add*: *Set.basic-monos*(7) *cblinfun-apply-in-image′ invariant-subspace-def less-eq-ccsubspace.rep-eq*)
also from ‹*S* ∈ *M*›**have** ‹... ⊆ *space-as-set* (⨆ *M*)›
by (*meson Sup-upper less-eq-ccsubspace.rep-eq*)
finally show ‹*r g* *$_C$ (*a g*) ∈ *space-as-set* (⨆ *M*)›

```
      by (simp add: complex-vector.subspace-scale)
    qed
    finally show ‹a h ∈ space-as-set (⨆ M)›.
  qed
  have ‹space-as-set (a *S ⨆ M) = closure (a ‘ closure (cspan (⨆ S∈M. space-as-set S)))›
    by (metis Sup-ccsubspace.rep-eq cblinfun-image.rep-eq)
  also have ‹... = closure (a ‘ cspan (⨆ S∈M. space-as-set S))›
    by (rule closure-bounded-linear-image-subset-eq)
       (simp add: cblinfun.real.bounded-linear-right)
  also from * have ‹... ⊆ closure (space-as-set (⨆ M))›
    by (meson closure-mono)
  also have ‹... = space-as-set (⨆ M)›
    by force
  finally have ‹a *S ⨆ M ≤ ⨆ M›
    by (simp add: less-eq-ccsubspace.rep-eq)
  then show ?thesis
    using invariant-subspaceI by blast
qed

lemma invariant-subspace-SUP:
  assumes ‹⋀x. x ∈ X ⟹ invariant-subspace (S x) a›
  shows ‹invariant-subspace (⨆ x∈X. S x) a›
  by (metis (mono-tags, lifting) assms imageE invariant-subspace-Sup)

lemma reducing-subspace-Inf:
  fixes a :: ‹'a::chilbert-space ⇒CL 'a›
  assumes ‹⋀S. S ∈ M ⟹ reducing-subspace S a›
  shows ‹reducing-subspace (⨅ M) a›
  using assms
  by (auto intro!: invariant-subspace-Inf invariant-subspace-SUP
      simp add: reducing-subspace-def uminus-Inf invariant-subspace-Inf)

lemma reducing-subspace-INF:
  fixes a :: ‹'a::chilbert-space ⇒CL 'a›
  assumes ‹⋀x. x ∈ X ⟹ reducing-subspace (S x) a›
  shows ‹reducing-subspace (⨅ x∈X. S x) a›
  by (metis (mono-tags, lifting) assms imageE reducing-subspace-Inf)

lemma reducing-subspace-Sup:
  fixes a :: ‹'a::chilbert-space ⇒CL 'a›
  assumes ‹⋀S. S ∈ M ⟹ reducing-subspace S a›
  shows ‹reducing-subspace (⨆ M) a›
  using assms
  by (auto intro!: invariant-subspace-Sup invariant-subspace-INF
      simp add: reducing-subspace-def uminus-Sup invariant-subspace-Inf)

lemma reducing-subspace-SUP:
  fixes a :: ‹'a::chilbert-space ⇒CL 'a›
  assumes ‹⋀x. x ∈ X ⟹ reducing-subspace (S x) a›
```

shows ‹*reducing-subspace* (⨆ *x*∈*X*. *S x*) *a*›
by (*metis* (*mono-tags*, *lifting*) *assms imageE reducing-subspace-Sup*)

lemma *selfadjoint-imp-normal*: ‹*normal-op a*› **if** ‹*selfadjoint a*›
using *that* **by** (*simp add*: *selfadjoint-def normal-op-def*)

lemma *eigenspaces-orthogonal*:
— [2], Proposition II.5.7
assumes ‹*e* ≠ *f*›
assumes ‹*normal-op a*›
shows ‹*orthogonal-spaces* (*eigenspace e a*) (*eigenspace f a*)›
proof (*intro orthogonal-spaces-def*[*THEN iffD2*] *ballI*)
fix *g h* **assume** *g-eigen*: ‹*g* ∈ *space-as-set* (*eigenspace e a*)› **and** *h-eigen*: ‹*h* ∈ *space-as-set* (*eigenspace f a*)›
with ‹*normal-op a*› **have** ‹*g* ∈ *space-as-set* (*eigenspace* (*cnj e*) (*a**))›
by (*simp add*: *normal-op-same-eigenspace-as-adj*)
then have *a-adj-g*: ‹(*a**) *g* = *cnj e* *$_C$ *g*›
using *eigenspace-memberD* **by** *blast*
from *h-eigen* **have** *a-h*: ‹*a h* = *f* *$_C$ *h*›
by (*simp add*: *eigenspace-memberD*)
have ‹*f* * (*g* ·$_C$ *h*) = *g* ·$_C$ *a h*›
by (*simp add*: *a-h*)
also have ‹... = (*a**) *g* ·$_C$ *h*›
by (*simp add*: *cinner-adj-left*)
also have ‹... = *e* * (*g* ·$_C$ *h*)›
using *a-adj-g* **by** *auto*
finally have ‹(*f* − *e*) * (*g* ·$_C$ *h*) = *0*›
by (*simp add*: *vector-space-over-itself.scale-left-diff-distrib*)
with ‹*e* ≠ *f*› **show** ‹*g* ·$_C$ *h* = *0*›
by *simp*
qed

definition *largest-eigenvalue* :: ‹('*a*::*complex-normed-vector* ⇒$_{CL}$ '*a*) ⇒ *complex*› **where**
‹*largest-eigenvalue a* =
(*if* ∃ *x*. *x* ∈ *eigenvalues a* ∧ (∀ *y* ∈ *eigenvalues a*. *cmod x* ≥ *cmod y*) *then*
SOME x. *x* ∈ *eigenvalues a* ∧ (∀ *y* ∈ *eigenvalues a*. *cmod x* ≥ *cmod y*) *else* 0)›

lemma *largest-eigenvalue-0-aux*:
‹*largest-eigenvalue* (*0* :: '*a*::{*not-singleton*,*complex-normed-vector*} ⇒$_{CL}$ '*a*) = *0*›
proof −
let *?zero* = ‹*0* :: '*a* ⇒$_{CL}$ '*a*›
define *e* **where** ‹*e* = (*SOME x*. *x* ∈ *eigenvalues ?zero* ∧ (∀ *y* ∈ *eigenvalues ?zero*. *cmod x* ≥ *cmod y*))›
have ‹∃ *e*. *e* ∈ *eigenvalues ?zero* ∧ (∀ *y*∈*eigenvalues ?zero*. *cmod y* ≤ *cmod e*)› (**is** ‹∃ *e*. *?P e*›)
by (*auto intro*!: *exI*[*of* - *0*])
then have ‹*?P e*›
unfolding *e-def*

by (*rule someI-ex*)
then have ‹*e* = *0*›
by *simp*
then show ‹*largest-eigenvalue ?zero* = *0*›
by (*simp add*: *largest-eigenvalue-def*)
qed

lemma *largest-eigenvalue-0*[*simp*]:
‹*largest-eigenvalue* (*0* :: *'a*::*complex-normed-vector* ⇒$_{CL}$ *'a*) = *0*›
proof (*cases* ‹*class.not-singleton TYPE*('*a*)›)
case *True*
show *?thesis*
using *complex-normed-vector-axioms True*
by (*rule largest-eigenvalue-0-aux*[*internalize-sort' 'a*])
next
case *False*
then have ‹*eigenvalues* (*0* :: *'a*::*complex-normed-vector* ⇒$_{CL}$ *'a*) = {}›
by (*rule not-not-singleton-no-eigenvalues*)
then show *?thesis*
by (*auto simp add*: *largest-eigenvalue-def*)
qed

hide-fact *largest-eigenvalue-0-aux*

lemma *eigenvalues-nonneg*:
assumes ‹*a* ≥ *0*› **and** ‹*v* ∈ *eigenvalues a*›
shows ‹*v* ≥ *0*›
proof −
from *assms* **obtain** *h* **where** ‹*norm h* = *1*› **and** *ahvh*: ‹*a* *$_V$ *h* = *v* *$_C$ *h*›
using *unit-eigenvector-ex* **by** *blast*
have ‹*0* ≤ *h* ∙$_C$ *a h*›
by (*simp add*: *assms*(*1*) *cinner-pos-if-pos*)
also have ‹... = *v* ∗ (*h* ∙$_C$ *h*)›
by (*simp add*: *ahvh*)
also have ‹... = *v*›
using ‹*norm h* = *1*› *cnorm-eq-1* **by** *auto*
finally show ‹*v* ≥ *0*›
by *blast*
qed

end

10 *Compact-Operators* − Finite rank and compact operators

theory *Compact-Operators*

imports *Misc-Tensor-Product-BO HS2Ell2*
Sqrt-Babylonian.Sqrt-Babylonian-Auxiliary Wlog.Wlog
HOL−Analysis.Abstract-Metric-Spaces
Strong-Operator-Topology
Misc-Tensor-Product-TTS
Eigenvalues
begin

unbundle *cblinfun-notation*

10.1 Finite rank operators

definition *finite-rank* **where** ‹*finite-rank A* ⟷ *A* ∈ *cspan* (*Collect rank1*)›

lemma *finite-rank-0*[*simp*]: ‹*finite-rank 0*›
by (*simp add*: *complex-vector.span-zero finite-rank-def*)

lemma *finite-rank-scaleC*[*simp*]: ‹*finite-rank* (*c* *$_C$ *a*)› **if** ‹*finite-rank a*›
using *complex-vector.span-scale finite-rank-def that* **by** *blast*

lemma *finite-rank-scaleR*[*simp*]: ‹*finite-rank* (*c* *$_R$ *a*)› **if** ‹*finite-rank a*›
by (*simp add*: *scaleR-scaleC that*)

lemma *finite-rank-uminus*[*simp*]: ‹*finite-rank* (−*a*) = *finite-rank a*›
by (*metis add.inverse-inverse complex-vector.span-neg finite-rank-def*)

lemma *finite-rank-plus*[*simp*]: ‹*finite-rank* (*a* + *b*)› **if** ‹*finite-rank a*› **and** ‹*finite-rank b*›
using *that* **by** (*auto simp*: *finite-rank-def complex-vector.span-add-eq2*)

lemma *finite-rank-minus*[*simp*]: ‹*finite-rank* (*a* − *b*)› **if** ‹*finite-rank a*› **and** ‹*finite-rank b*›
using *complex-vector.span-diff finite-rank-def that*(*1*) *that*(*2*) **by** *blast*

lemma *finite-rank-butterfly*[*simp*]: ‹*finite-rank* (*butterfly x y*)›
by (*cases* ‹*x* ≠ *0* ∧ *y* ≠ *0*›)
(*auto intro*: *complex-vector.span-base complex-vector.span-zero simp add*: *finite-rank-def*)

lemma *finite-rank-sum-butterfly*:
fixes *a* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
assumes ‹*finite-rank a*›
shows ‹∃ *x y* (*n*::*nat*). *a* = (∑ *i*<*n*. *butterfly* (*x i*) (*y i*))›
proof −
from *assms*
have ‹*a* ∈ *cspan* (*Collect rank1*)›
by (*simp add*: *finite-rank-def*)
then obtain *r t* **where** ‹*finite t*› **and** *t-rank1*: ‹*t* ⊆ *Collect rank1*›
and *a-sum*: ‹*a* = (∑ *a*∈*t*. *r a* *$_C$ *a*)›
by (*smt* (*verit*, *best*) *complex-vector.span-alt mem-Collect-eq*)
from ‹*finite t*› **obtain** *ι* **and** *n*::*nat* **where** *ι*: ‹*bij-betw ι* {..<*n*} *t*›
using *bij-betw-from-nat-into-finite* **by** *blast*

define c **where** ‹$c\ i = r\ (\iota\ i) *_C \iota\ i$› **for** i
from ι *t-rank1*
have *c-rank1*: ‹*rank1* $(c\ i) \vee c\ i = 0$› **if** ‹$i < n$› **for** i
by (*auto intro*!: *rank1-scaleC simp*: *c-def bij-betw-apply subset-iff that*)
have *ac-sum*: ‹$a = (\sum i{<}n.\ c\ i)$›
by (*smt* (*verit*, *best*) *a-sum* ι *c-def sum.cong sum.reindex-bij-betw*)
from *c-rank1*
obtain $x\ y$ **where** ‹$c\ i =$ *butterfly* $(x\ i)\ (y\ i)$› **if** ‹$i < n$› **for** i
apply *atomize-elim*
apply (*rule SMT-choices*)
using *rank1-iff-butterfly* **by** *fastforce*
with *ac-sum* **show** *?thesis*
by *auto*
qed

lemma *finite-rank-sum*: ‹*finite-rank* $(\sum x{\in}F.\ f\ x)$› **if** ‹$\bigwedge x.\ x{\in}F \Longrightarrow$ *finite-rank* $(f\ x)$›
using *that* **by** (*induction F rule*:*infinite-finite-induct*) (*auto intro*!: *finite-rank-plus*)

lemma *rank1-finite-rank*: ‹*finite-rank a*› **if** ‹*rank1 a*›
by (*simp add*: *complex-vector.span-base finite-rank-def that*)

lemma *finite-rank-compose-left*:
assumes ‹*finite-rank B*›
shows ‹*finite-rank* $(A\ o_{CL}\ B)$›
proof −
from *assms* **have** ‹$B \in$ *cspan* (*Collect rank1*)›
by (*simp add*: *finite-rank-def*)
then obtain $F\ t$ **where** ‹*finite F*› **and** *F-rank1*: ‹$F \subseteq$ *Collect rank1*› **and** ‹$B = (\sum x{\in}F.\ t\ x *_C x)$›
by (*smt* (*verit*, *best*) *complex-vector.span-explicit mem-Collect-eq*)
then have ‹$A\ o_{CL}\ B = (\sum x{\in}F.\ t\ x *_C (A\ o_{CL}\ x))$›
by (*metis* (*mono-tags*, *lifting*) *cblinfun-compose-scaleC-right cblinfun-compose-sum-right sum.cong*)
also have ‹$\ldots \in$ *cspan* (*Collect finite-rank*)›
by (*intro complex-vector.span-sum complex-vector.span-scale*)
(*use F-rank1* **in** ‹*auto intro*!: *complex-vector.span-base rank1-finite-rank rank1-compose-left*›)
also have ‹$\ldots =$ *Collect finite-rank*›
by (*metis* (*no-types*, *lifting*) *complex-vector.span-superset cspan-eqI finite-rank-def mem-Collect-eq subset-antisym subset-iff*)
finally show *?thesis*
by *simp*
qed

lemma *finite-rank-compose-right*:
assumes ‹*finite-rank A*›
shows ‹*finite-rank* $(A\ o_{CL}\ B)$›
proof −

from *assms* **have** ‹$A \in$ *cspan* (*Collect rank1*)›
by (*simp add*: *finite-rank-def*)
then obtain F t **where** ‹*finite* F› **and** *F-rank1*: ‹$F \subseteq$ *Collect rank1*› **and** ‹$A = (\sum x{\in}F.\ t\ x *_C x)$›
by (*smt* (*verit*, *best*) *complex-vector.span-explicit mem-Collect-eq*)
then have ‹$A\ o_{CL}\ B = (\sum x{\in}F.\ t\ x *_C (x\ o_{CL}\ B))$›
by (*metis* (*mono-tags*, *lifting*) *cblinfun-compose-scaleC-left cblinfun-compose-sum-left sum.cong*)
also have ‹$\ldots \in$ *cspan* (*Collect finite-rank*)›
by (*intro complex-vector.span-sum complex-vector.span-scale*)
(*use F-rank1* **in** ‹*auto intro*!: *complex-vector.span-base rank1-finite-rank rank1-compose-right*›)
also have ‹$\ldots =$ *Collect finite-rank*›
by (*metis* (*no-types*, *lifting*) *complex-vector.span-superset cspan-eqI finite-rank-def mem-Collect-eq subset-antisym subset-iff*)
finally show *?thesis*
by *simp*
qed

lemma *rank1-Proj-singleton*[*iff*]: ‹*rank1* (*Proj* (*ccspan* {x}))›
using *Proj-range rank1-def* **by** *blast*

lemma *finite-rank-Proj-singleton*[*iff*]: ‹*finite-rank* (*Proj* (*ccspan* {x}))›
by (*simp add*: *rank1-finite-rank*)

lemma *finite-rank-Proj-finite-dim*:
fixes S :: ‹'a::*chilbert-space ccsubspace*›
assumes ‹*finite-dim-ccsubspace* S›
shows ‹*finite-rank* (*Proj* S)›
proof −
from *assms*
obtain B **where** ‹*is-ortho-set* B› **and** ‹*finite* B› **and** *spanB*: ‹*cspan* B = *space-as-set* S›
unfolding *finite-dim-ccsubspace.rep-eq*
using *cfinite-dim-subspace-has-onb* **by** *force*
have ‹*Proj* S = *Proj* (*ccspan* B)›
by (*metis Proj.rep-eq* ‹*finite* B› *cblinfun-apply-inject ccspan-finite spanB*)
moreover have ‹*finite-rank* (*Proj* (*ccspan* B))›
using ‹*finite* B› ‹*is-ortho-set* B›
proof *induction*
case *empty*
then show *?case*
by *simp*
next
case (*insert* x F)
then have ‹*is-ortho-set* F›
by (*meson is-ortho-set-antimono subset-insertI*)
have ‹*Proj* (*ccspan* (*insert* x F)) = *proj* x + *Proj* (*ccspan* F)›
by (*subst Proj-orthog-ccspan-insert*)
(*use insert* **in** ‹*auto simp*: *is-onb-def is-ortho-set-def*›)
moreover have ‹*finite-rank* ...›
by (*rule finite-rank-plus*)

```
      (auto intro!: ‹is-ortho-set F› insert)
    ultimately show ?case
      by simp
  qed
  ultimately show ?thesis
    by simp
qed

lemma finite-rank-Proj-finite:
  fixes F :: ‹'a::chilbert-space set›
  assumes ‹finite F›
  shows ‹finite-rank (Proj (ccspan F))›
proof −
  obtain B where ‹is-ortho-set B› and ‹finite B› and ‹cspan B = cspan F›
    by (meson assms orthonormal-basis-of-cspan)
  have ‹Proj (ccspan F) = Proj (ccspan B)›
    by (simp add: ‹cspan B = cspan F› ccspan.abs-eq)
  moreover have ‹finite-rank (Proj (ccspan B))›
    using ‹finite B› ‹is-ortho-set B›
  proof induction
    case empty
    then show ?case
      by simp
  next
    case (insert x F)
    then have ‹is-ortho-set F›
      by (meson is-ortho-set-antimono subset-insertI)
    have ‹Proj (ccspan (insert x F)) = proj x + Proj (ccspan F)›
      by (subst Proj-orthog-ccspan-insert)
         (use insert in ‹auto simp: is-onb-def is-ortho-set-def›)
    moreover have ‹finite-rank . . . ›
      by (rule finite-rank-plus) (auto intro!: ‹is-ortho-set F› insert)
    ultimately show ?case
      by simp
  qed
  ultimately show ?thesis
    by simp
qed

lemma finite-rank-cfinite-dim[simp]: ‹finite-rank (a :: 'a :: {cfinite-dim,chilbert-space} ⇒CL 'b
:: complex-normed-vector)›
proof −
  obtain B :: ‹'a set› where ‹is-onb B›
    using is-onb-some-chilbert-basis by blast
  from ‹is-onb B› have [simp]: ‹finite B›
   by (auto intro!: cindependent-cfinite-dim-finite is-ortho-set-cindependent simp add: is-onb-def)
  have [simp]: ‹cspan B = UNIV›
  proof −
    from ‹is-onb B› have ‹ccspan B = ⊤›
```

```
    using is-onb-def by blast
  then have ‹closure (cspan B) = UNIV›
    by (metis ccspan.rep-eq space-as-set-top)
  then show ?thesis
    by simp
qed
have a-sum: ‹a = (∑ b∈B. a o_CL selfbutter b)›
proof (rule cblinfun-eq-on-UNIV-span[OF ‹cspan B = UNIV›])
  fix s assume [simp]: ‹s ∈ B›
  with ‹is-onb B› have ‹norm s = 1›
    by (simp add: is-onb-def)
  have 1: ‹j ≠ s ⟹ j ∈ B ⟹ (a o_CL selfbutter j) *_V s = 0› for j
    using ‹is-onb B› ‹s ∈ B› cblinfun.scaleC-right is-onb-def is-ortho-set-def scaleC-eq-0-iff
    by fastforce
  have 2: ‹a *_V s = (if s ∈ B then (a o_CL selfbutter s) *_V s else 0)›
    using ‹norm s = 1› ‹s ∈ B› by (simp add: cnorm-eq-1)
  show ‹a *_V s = (∑ b∈B. a o_CL selfbutter b) *_V s›
    by (subst cblinfun.sum-left, subst sum-single[where i=s]) (use 1 2 in auto)
qed
have ‹finite-rank (∑ b∈B. a o_CL selfbutter b)›
  by (auto intro!: finite-rank-sum simp: cblinfun-comp-butterfly)
with a-sum show ?thesis
  by simp
qed

lemma finite-rank-cspan-butterflies:
  ‹finite-rank a ⟷ a ∈ cspan (range (case-prod butterfly))›
  for a :: ‹'a::chilbert-space ⇒_CL 'b::chilbert-space›
proof −
  have ‹(Collect finite-rank :: ('a ⇒_CL 'b) set) = cspan (Collect rank1)›
    using finite-rank-def by fastforce
  also have ‹... = cspan (insert 0 (Collect rank1))›
    by force
  also have ‹... = cspan (range (case-prod butterfly))›
    by (rule arg-cong[where f=cspan])
      (use butterfly-0-left in ‹force simp: image-iff rank1-iff-butterfly simp del: butterfly-0-left›)+
  finally show ?thesis
    by auto
qed


lemma finite-rank-comp-left: ‹finite-rank (a o_CL b)› if ‹finite-rank a›
  for a b :: ‹-::chilbert-space ⇒_CL -::chilbert-space›
proof −
  from that
  have ‹a ∈ cspan (range (case-prod butterfly))›
    by (simp add: finite-rank-cspan-butterflies)
  then have ‹a o_CL b ∈ (λa. a o_CL b) ‘ cspan (range (case-prod butterfly))›
    by fast
```

also have ‹... = *cspan* ((λ*a*. *a* o_{CL} *b*) ‘ *range* (*case-prod butterfly*))›
by (*simp add*: *clinear-cblinfun-compose-left complex-vector.linear-span-image*)
also have ‹... ⊆ *cspan* (*range* (*case-prod butterfly*))›
by (*force intro*!: *complex-vector.span-mono*
simp add: *image-image case-prod-unfold butterfly-comp-cblinfun image-def*)
finally show *?thesis*
using *finite-rank-cspan-butterflies* **by** *blast*
qed

lemma *finite-rank-comp-right*: ‹*finite-rank* (*a* o_{CL} *b*)› **if** ‹*finite-rank b*›
for *a b* :: ‹-::*chilbert-space* ⇒$_{CL}$ -::*chilbert-space*›
proof −
from *that*
have ‹*b* ∈ *cspan* (*range* (*case-prod butterfly*))›
by (*simp add*: *finite-rank-cspan-butterflies*)
then have ‹*a* o_{CL} *b* ∈ ((o_{CL}) *a*) ‘ *cspan* (*range* (*case-prod butterfly*))›
by *fast*
also have ‹... = *cspan* (((o_{CL}) *a*) ‘ *range* (*case-prod butterfly*))›
by (*simp add*: *clinear-cblinfun-compose-right complex-vector.linear-span-image*)
also have ‹... ⊆ *cspan* (*range* (*case-prod butterfly*))›
by (*force intro*!: *complex-vector.span-mono*
simp add: *image-image case-prod-unfold cblinfun-comp-butterfly image-def*)
finally show *?thesis*
using *finite-rank-cspan-butterflies* **by** *blast*
qed

10.2 Compact operators

definition *compact-map* **where** ‹*compact-map f* ⟷ *clinear f* ∧ *compact* (*closure* (*f* ‘ *cball 0 1*))›

lemma ‹*bounded-clinear f*› **if** ‹*compact-map f*›
— [2], Proposition II.4.2 (a)
thm *bounded-clinear-def*
proof (*unfold bounded-clinear-def bounded-clinear-axioms-def*, *intro conjI*)
show ‹*clinear f*›
using *compact-map-def that* **by** *blast*
have ‹*compact* (*closure* (*f* ‘ *cball 0 1*))›
using *compact-map-def that* **by** *blast*
then have ‹*bounded* (*f* ‘ *cball 0 1*)›
by (*meson bounded-subset closure-subset compact-imp-bounded*)
then obtain *K* **where** ∗: ‹*norm* (*f x*) ≤ *K*› **if** ‹*norm x* ≤ *1*› **for** *x*
by (*force simp*: *bounded-iff dist-norm ball-def*)
have ‹*norm* (*f x*) ≤ *norm x* ∗ *K*› **for** *x*
proof (*cases* ‹*x* = *0*›)
case *True*
then show *?thesis*
using ‹*clinear f*› *complex-vector.linear-0* **by** *force*

next
case *False*
have ‹*norm* (*f x*) = *norm* (*f* (*norm x* $*_C$ *sgn x*))›
by *simp*
also have ‹... = *norm x* * *norm* (*f* (*sgn x*))›
by (*smt* (*verit*, *best*) ‹*clinear f*› *complex-vector.linear-scale norm-ge-zero norm-of-real norm-scaleC*)
also have ‹... ≤ *norm x* * *K*›
by (*simp add*: * *mult-left-mono norm-sgn*)
finally show *?thesis*
by −
qed
then show ‹∃ *K*. ∀ *x*. *norm* (*f x*) ≤ *norm x* * *K*›
by *blast*
qed

lift-definition *compact-op* :: ‹('*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector*) ⇒ *bool*› **is** *compact-map*.

lemma *compact-op-def2*: ‹*compact-op a* ⟷ *compact* (*closure* (*a* ‘ *cball 0 1*))›
by *transfer* (*use bounded-clinear.clinear compact-map-def* **in** *blast*)

lemma *compact-op-0*[*simp*]: ‹*compact-op 0*›
by (*simp add*: *compact-op-def2 image-constant*[**where** *x*=*0*] *mem-cball-leI*[**where** *x*=*0*])

lemma *compact-op-scaleC*[*simp*]: ‹*compact-op* (*c* $*_C$ *a*)› **if** ‹*compact-op a*›
proof −
have ‹*compact* (*closure* (*a* ‘ *cball 0 1*))›
using *compact-op-def2 that* **by** *blast*
then have *: ‹*compact* (*scaleC c* ‘ *closure* (*a* ‘ *cball 0 1*))›
using *compact-scaleC* **by** *blast*
have ‹*closure* ((*c* $*_C$ *a*) ‘ *cball 0 1*) = *closure* (*scaleC c* ‘ *a* ‘ *cball 0 1*)›
by (*metis* (*no-types*, *lifting*) *cblinfun.scaleC-left image-cong image-image*)
also have ‹... = *scaleC c* ‘ *closure* (*a* ‘ *cball 0 1*)›
using *closure-scaleC* **by** *blast*
finally have ‹*compact* (*closure* ((*c* $*_C$ *a*) ‘ *cball 0 1*))›
using * **by** *simp*
then show *?thesis*
using *compact-op-def2* **by** *blast*
qed

lemma *compact-op-scaleR*[*simp*]: ‹*compact-op* (*c* $*_R$ *a*)› **if** ‹*compact-op a*›
by (*simp add*: *scaleR-scaleC that*)

lemma *compact-op-uminus*[*simp*]: ‹*compact-op* (−*a*) = *compact-op a*›
by (*metis compact-op-scaleC scaleC-minus1-left verit-minus-simplify*(*4*))

lemma *compact-op-plus*[*simp*]: ‹*compact-op* (*a* + *b*)› **if** ‹*compact-op a*› **and** ‹*compact-op b*›
proof −

```
  have ‹compact (closure (a ‘ cball 0 1))›
    using compact-op-def2 that by blast
  moreover have ‹compact (closure (b ‘ cball 0 1))›
    using compact-op-def2 that by blast
  ultimately have compact-sum:
    ‹compact {x + y |x y. x ∈ closure ((*V) a ‘ cball 0 1)
                        ∧ y ∈ closure ((*V) b ‘ cball 0 1)}› (is ‹compact ?sum›)
    by (rule compact-sums)
  have ‹compact (closure ((a + b) ‘ cball 0 1))›
  proof −
    have ‹((*V) (a + b) ‘ cball 0 1) ⊆ ?sum›
      using cblinfun.real.add-left closure-subset image-subset-iff by blast
    then have ‹closure ((*V) (a + b) ‘ cball 0 1) ⊆ closure ?sum›
      by (meson closure-mono)
    also have ‹... = ?sum›
      using compact-sum
      by (auto intro!: closure-closed compact-imp-closed)
    finally show ?thesis
      by (rule compact-closed-subset[rotated 2]) (use compact-sum in auto)
  qed
  then show ?thesis
    using compact-op-def2 by blast
qed

lemma csubspace-compact-op: ‹csubspace (Collect compact-op)›
  — [2], Proposition II.4.2 (b)
  by (simp add: complex-vector.subspace-def)

lemma compact-op-minus[simp]: ‹compact-op (a − b)› if ‹compact-op a› and ‹compact-op b›
  by (metis compact-op-plus compact-op-uminus that(1) that(2) uminus-add-conv-diff)

lemma compact-op-sgn[simp]: ‹compact-op (sgn a) = compact-op a›
proof (cases ‹a = 0›)
  case True
  then show ?thesis
    by simp
next
  case False
  have ‹compact-op (sgn a)› if ‹compact-op a›
    by (simp add: sgn-cblinfun-def that)
  moreover have ‹compact-op (norm a *R sgn a)› if ‹compact-op (sgn a)›
    by (simp add: that)
  moreover have ‹norm a *R sgn a = a›
    by (simp add: False sgn-div-norm)
  ultimately show ?thesis
    by auto
qed

lemma closed-compact-op:
```

shows ‹*closed* (*Collect* (*compact-op* :: ('*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*chilbert-space*) ⇒ *bool*))›
— [2], Proposition II.4.2 (b)
proof (*intro closed-sequential-limits*[*THEN iffD2*] *allI impI conjI*)
fix *T* **and** *A* :: ‹'*a* ⇒$_{CL}$ '*b*›
assume *asm*: ‹(∀ *n*. *T n* ∈ *Collect compact-op*) ∧ *T* ⟶ *A*›
have ‹*Met-TC.mtotally-bounded* (*A* ‘ *cball 0 1*)›
proof (*unfold Met-TC.mtotally-bounded-def*, *intro allI impI*)
fix ε :: *real* **assume** ‹ε > *0*›
define δ **where** ‹δ = ε/*3*›
then have ‹δ > *0*›
using ‹ε > *0*› **by** *simp*
from *asm*[*unfolded LIMSEQ-def*, *THEN conjunct2*, *rule-format*, *OF* ‹δ > *0*›]
obtain *n* **where** *dist-TA*: ‹*dist* (*T n*) *A* < δ›
by *auto*
from *asm* **have** ‹*compact-op* (*T n*)›
by *simp*
then have ‹*Met-TC.mtotally-bounded* (*T n* ‘ *cball 0 1*)›
by (*subst Met-TC.mtotally-bounded-eq-compact-closure-of*)
(*auto intro*!: *simp*: *compact-op-def2 Met-TC.mtotally-bounded-eq-compact-closure-of*)
then obtain *K* **where** ‹*finite K*› **and** *K-T*: ‹*K* ⊆ *T n* ‘ *cball 0 1*› **and**
TK: ‹*T n* ‘ *cball 0 1* ⊆ (⋃*k*∈*K*. *Met-TC.mball k* δ)›
unfolding *Met-TC.mtotally-bounded-def* **using** ‹δ > *0*› **by** *meson*
from ‹*finite K*› **and** *K-T* **obtain** *H* **where** ‹*finite H*› **and** ‹*H* ⊆ *cball 0 1*›
and *KTH*: ‹*K* = *T n* ‘ *H*›
by (*meson finite-subset-image*)
from *TK* **have** *TH*: ‹*T n* ‘ *cball 0 1* ⊆ (⋃*h*∈*H*. *ball* (*T n* ∗$_V$ *h*) δ)›
by (*simp add*: *KTH*)
have ‹*A* ‘ *cball 0 1* ⊆ (⋃*h*∈*H*. *ball* (*A h*) ε)›
proof (*rule subsetI*)
fix *x* **assume** ‹*x* ∈ (∗$_V$) *A* ‘ *cball 0 1*›
then obtain *l* **where** ‹*l* ∈ *cball 0 1*› **and** *xl*: ‹*x* = *A l*›
by *blast*
then have ‹*T n l* ∈ *T n* ‘ *cball 0 1*›
by *auto*
with *TH* **obtain** *h* **where** ‹*h* ∈ *H*› **and** ‹*T n l* ∈ *ball* (*T n h*) δ›
by *blast*
then have *dist-Tlh*: ‹*dist* (*T n l*) (*T n h*) < δ›
by (*simp add*: *dist-commute*)
have ‹*dist* (*A h*) (*A l*) < ε›
proof −
have *norm-h*: ‹*norm h* ≤ *1*›
using ‹*H* ⊆ *cball 0 1*› ‹*h* ∈ *H*› *mem-cball-0* **by** *blast*
have *norm-l*: ‹*norm l* ≤ *1*›
using ‹*l* ∈ *cball 0 1*› **by** *auto*
have ‹*dist* (*A h*) (*T n h*) < δ›
proof −
have ‹*dist* (*T n* ∗$_V$ *h*) (*A* ∗$_V$ *h*) ≤ *norm h* ∗ *dist* (*T n*) *A*›
using *norm-cblinfun*[*of T n* − *A h*] **by** (*simp add*: *dist-norm cblinfun.diff-left mult-ac*)

```
      also have ‹... ≤ 1 * dist (T n) A›
        by (rule mult-right-mono) (use norm-h in auto)
      also have ‹dist (T n) A < δ›
        by fact
      finally show ?thesis
        by (simp add: dist-commute)
    qed
    moreover have ‹dist (T n h) (T n l) < δ›
      using dist-Tlh by (metis dist-commute)
    moreover from dist-TA norm-l have ‹dist (T n l) (A l) < δ›
    proof −
      have ‹dist (T n *V l) (A *V l) ≤ norm l * dist (T n) A›
       using norm-cblinfun[of T n − A l] by (simp add: dist-norm cblinfun.diff-left mult-ac)
      also have ‹... ≤ 1 * dist (T n) A›
        by (rule mult-right-mono) (use norm-l in auto)
      also have ‹dist (T n) A < δ›
        by fact
      finally show ?thesis
        by (simp add: dist-commute)
    qed
    ultimately show ?thesis
      unfolding δ-def
      by (rule dist-triangle-third)
  qed
  then show ‹x ∈ (⋃h∈H. ball (A h) ε) ›
    using ‹h ∈ H› by (auto intro!: simp: xl)
 qed
 then show ‹∃K. finite K ∧ K ⊆ (*V) A ‘ cball 0 1 ∧
        (*V) A ‘ cball 0 1 ⊆ (⋃x∈K. Met-TC.mball x ε)›
  using ‹H ⊆ cball 0 1›
  by (force intro!: exI[of - ‹A ‘ H›] ‹finite H› simp: ball-def)
qed
then have ‹Met-TC.mtotally-bounded (closure (A ‘ cball 0 1))›
  using Met-TC.mtotally-bounded-closure-of by auto
then have ‹compact (closure (A ‘ cball 0 1))›
  by (simp-all add: Met-TC.mtotally-bounded-eq-compact-closure-of complete-UNIV-cuspace)
then show ‹A ∈ Collect compact-op›
  using compact-op-def2 by blast
qed

lemma rank1-compact-op: ‹compact-op a› if ‹rank1 a›
proof −
 wlog ‹a ≠ 0›
  using negation by simp
 with that obtain ψ where im-a: ‹a *S ⊤ = ccspan {ψ}› and ‹ψ ≠ 0›
  using rank1-def by fastforce
 define c where ‹c = norm a / norm ψ›
 have compact-ψc: ‹compact ((λx. x *C ψ) ‘ cball 0 c)›
 proof −
```

```
    have ‹continuous-on (cball 0 c) (λx. x *C ψ)›
      by (auto intro!: continuous-at-imp-continuous-on)
    moreover have ‹compact (cball (0::complex) c)›
      by (simp add: compact-eq-bounded-closed)
    ultimately show ?thesis
      by (rule compact-continuous-image)
  qed
  have ‹a ‘ cball 0 1 ⊆ (λx. x *C ψ) ‘ cball 0 c›
  proof (rule subsetI)
    fix φ
    assume asm: ‹φ ∈ a ‘ cball 0 1›
    then have ‹φ ∈ space-as-set (a *S ⊤)›
      using cblinfun-apply-in-image by blast
    also have ‹... = cspan {ψ}›
      by (simp add: ccspan.rep-eq im-a)
    finally obtain d where d: ‹φ = d *C ψ›
      by (metis complex-vector.span-breakdown-eq complex-vector.span-empty eq-iff-diff-eq-0 sin-
gletonD)
    from asm obtain γ where ‹φ = a γ› and ‹norm γ ≤ 1›
      by force
    have ‹cmod d * norm ψ = norm φ›
      by (simp add: d)
    also have ‹... ≤ norm a * norm γ›
      using ‹φ = a *V γ› complex-of-real-mono norm-cblinfun by blast
    also have ‹... ≤ norm a›
      by (metis ‹norm γ ≤ 1› mult.commute mult-left-le-one-le norm-ge-zero)
    finally have ‹cmod d ≤ c›
    by (smt (verit, ccfv-threshold) ‹ψ ≠ 0› c-def linordered-field-class.pos-divide-le-eq nonzero-eq-divide-eq
norm-le-zero-iff)
    then show ‹φ ∈ (λx. x *C ψ) ‘ cball 0 c›
      by (auto simp: d)
  qed
  with compact-ψc have cl-in-cl: ‹closure (a ‘ cball 0 1) ⊆ ((λx. x *C ψ) ‘ cball 0 c)›
    using closure-mono[of - ‹((λx. x *C ψ) ‘ cball 0 c)›] compact-ψc
    by (simp add: compact-imp-closed)
  with compact-ψc show ‹compact-op a›
    using compact-closed-subset compact-op-def2 by blast
qed

lemma finite-rank-compact-op: ‹compact-op a› if ‹finite-rank a›
proof −
  from that obtain t r where ‹finite t› and ‹t ⊆ Collect rank1›
    and a-decomp: ‹a = (∑x∈t. r x *C x)›
    by (auto simp: finite-rank-def complex-vector.span-explicit)
  from ‹finite t› ‹t ⊆ Collect rank1› show ‹compact-op a›
    by (unfold a-decomp, induction)
       (auto intro!: compact-op-plus compact-op-scaleC intro: rank1-compact-op)
qed
```

lemma *bounded-products-sot-lim-imp-lim*:
— Implicit in the proof of [2], Proposition II.4.4 (c)
fixes A :: ‹'a::*complex-normed-vector* $\Rightarrow_{CL}$ 'b::*chilbert-space*›
assumes *lim-PA*: ‹*limitin cstrong-operator-topology* $(\lambda x.\ P\ x\ o_{CL}\ A)$ A F›
and ‹*compact-op* A›
and *P-leq-B*: ‹$\bigwedge x.$ *norm* $(P\ x) \leq B$›
shows ‹$((\lambda x.\ P\ x\ o_{CL}\ A) \longrightarrow A)$ F›
proof −
wlog ‹$F \neq \bot$›
using *negation* **by** *simp*
wlog ‹$B \neq 0$›
proof −
from *negation assms* **have** *P0*: ‹$P\ x = 0$› **for** x
by *auto*
from *lim-PA* **have** ‹$((\lambda x.\ 0) \longrightarrow$ *Abs-cblinfun-sot* $A)$ F›
unfolding *limitin-canonical-iff* [*symmetric*]
by (*transfer fixing*: P F) (*use P0* **in** *simp*)
moreover have ‹$((\lambda x.\ 0) \longrightarrow 0)$ F›
by *simp*
ultimately have ‹*Abs-cblinfun-sot* $A = 0$›
using ‹$F \neq \bot$› *tendsto-unique* **by** *blast*
then have ‹$A = 0$›
by (*metis Abs-cblinfun-sot-inverse cstrong-operator-topology-topspace lim-PA*
limitin-def zero-cblinfun-sot.rep-eq)
with *P0* **show** *?thesis*
by *simp*
qed
have ‹$B > 0$›
proof −
from *P-leq-B*[*of undefined*] **have** ‹$B \geq 0$›
by (*smt* (*verit*, *del-insts*) *norm-ge-zero*)
with ‹$B \neq 0$›
show *?thesis*
by *simp*
qed

show *?thesis*
proof (*rule metric-space-class.tendstoI*)
fix ε :: *real* **assume** ‹$\varepsilon > 0$›
define δ γ T **where** ‹$\delta = \varepsilon/4$› **and** ‹$\gamma = min\ \delta\ (\delta/B)$› **and** ‹$T\ x = P\ x\ o_{CL}\ A$› **for** x
then have ‹$\delta > 0$›
using ‹$\varepsilon > 0$› **by** *simp*
then have ‹$\gamma > 0$›
using ‹$B > 0$› **by** (*simp add*: γ-*def*)
from ‹*compact-op* A› **have** ‹*Met-TC.mtotally-bounded* $(A$ ‘ *cball 0 1*$)$›
by (*subst Met-TC.mtotally-bounded-eq-compact-closure-of*)
(*auto intro*!: *simp*: *compact-op-def2 Met-TC.mtotally-bounded-eq-compact-closure-of*)
then obtain K **where** ‹*finite* K› **and** K-T: ‹$K \subseteq A$ ‘ *cball 0 1*› **and**
AK: ‹A ‘ *cball 0 1* $\subseteq (\bigcup k{\in}K.$ *Met-TC.mball k* $\gamma)$›

```
    unfolding Met-TC.mtotally-bounded-def using ‹γ > 0› by meson
  from ‹finite K› and K-T obtain H where ‹finite H› and ‹H ⊆ cball 0 1›
    and KAH: ‹K = A ‘ H›
    by (meson finite-subset-image)
  from AK have AH: ‹A ‘ cball 0 1 ⊆ (⋃h∈H. ball (A *V h) γ)›
    by (simp add: KAH)
  have ‹∀F x in F. ∀h∈H. dist (T x h) (A h) < δ›
    using lim-PA ‹δ > 0›
    by (auto intro!: eventually-ball-finite ‹finite H›
        simp: limitin-cstrong-operator-topology T-def metric-space-class.tendsto-iff)
  then show ‹∀F x in F. dist (T x) A < ε›
  proof (rule eventually-mono)
    fix x
    assume asm: ‹∀h∈H. dist (T x *V h) (A *V h) < δ›
    have ‹dist (T x l) (A l) ≤ 3 * δ› if ‹norm l = 1› for l
    proof −
      from that have ‹A l ∈ A ‘ cball 0 1›
        by auto
      with AH obtain h where ‹h ∈ H› and Alγ: ‹A l ∈ ball (A h) γ›
        by blast
      then have dist-Alh: ‹dist (A l) (A h) < γ›
        by (simp add: dist-commute)
      have ‹dist (A l) (A h) < δ›
        using dist-Alh by (simp add: γ-def)
      moreover from asm have ‹dist (A h) (T x h) < δ›
        by (simp add: ‹h ∈ H› dist-commute)
      moreover have ‹dist (T x h) (T x l) < δ›
      proof −
        have ‹dist (T x h) (T x l) ≤ norm (P x) * dist (A h) (A l)›
          by (metis T-def cblinfun.real.diff-right cblinfun-apply-cblinfun-compose
                dist-norm norm-cblinfun)
        also from Alγ P-leq-B have ‹... < B * γ›
         by (smt (verit, ccfv-SIG) ‹B ≠ 0› linordered-semiring-strict-class.mult-le-less-imp-less
linordered-semiring-strict-class.mult-strict-mono' mem-ball norm-ge-zero zero-le-dist)
        also have ‹... ≤ B * (δ / B)›
          by (smt (verit, best) γ-def ‹0 < B› mult-left-mono)
        also have ‹... ≤ δ›
          by (simp add: ‹B ≠ 0›)
        finally show ?thesis
          by −
      qed
      ultimately show ?thesis
        by (smt (verit) dist-commute dist-triangle2)
    qed
    then have ‹dist (T x) A ≤ 3 * δ›
      unfolding dist-norm using ‹δ > 0›
      by (auto intro!: norm-cblinfun-bound-unit simp: cblinfun.diff-left)
    then show ‹dist (T x) A < ε›
      by (rule order.strict-trans1) (use ‹ε > 0› in ‹simp add: δ-def›)
```

```
    qed
  qed
qed


lemma compact-op-finite-rank:
  fixes A :: ‹'a::complex-normed-vector ⇒_CL 'b::chilbert-space›
  shows ‹compact-op A ⟷ A ∈ closure (Collect finite-rank)›
  — [2], Proposition II.4.4 (c)
proof (rule iffI)
  assume ‹A ∈ closure (Collect finite-rank)›
  then have ‹A ∈ closure (Collect compact-op)›
    by (metis closure-sequential finite-rank-compact-op mem-Collect-eq)
  also have ‹... = Collect compact-op›
    by (simp add: closed-compact-op)
  finally show ‹compact-op A›
    by simp
next
  assume ‹compact-op A›
  then have ‹compact (closure (A ‘ cball 0 1))›
    using compact-op-def2 by blast
  then have sep-A-ball: ‹separable (closure (A ‘ cball 0 1))›
    using compact-imp-separable by blast
  define L where ‹L = closure (range A)›
  obtain B :: ‹nat ⇒ -› where ‹L ⊆ closure (range B)›
  proof atomize-elim
    from sep-A-ball obtain B0 where ‹countable B0›
      and A-B0: ‹A ‘ cball 0 1 ⊆ closure B0›
      by (meson closure-subset order-trans separable-def)
    define B1 where ‹B1 = (⋃n::nat. scaleR n ‘ B0)›
    from ‹countable B0› have ‹countable B1›
      by (auto intro!: countable-UN countable-image simp: B1-def)
    have ‹range A = (⋃n::nat. A ‘ scaleR n ‘ cball (0::'a) 1)›
    proof −
      have ‹UNIV = (⋃n::nat. scaleR n ‘ cball (0::'a) 1)›
      proof (intro antisym subsetI UNIV-I)
        fix x :: 'a
        have norm x < 1 + real-of-int ⌈norm x⌉ 1 + real-of-int ⌈norm x⌉ > 0
          using norm-ge-zero[of x] by linarith+
        hence ‹x ∈ scaleR (nat (ceiling (norm x)) + 1) ‘ cball (0::'a) 1›
          by (intro image-eqI[where x=‹x /_R (nat (ceiling (norm x)) + 1)›])
             (auto simp: divide-simps)
        then show ‹x ∈ (⋃x::nat. (*_R) (real x) ‘ cball 0 1)›
          by blast
      qed
      then show ?thesis
        by fastforce
    qed
    also have ‹... = (⋃n::nat. scaleR n ‘ A ‘ cball 0 1)›
```

```
    by (auto simp: cblinfun.scaleR-right image-comp fun-eq-iff)
  also have ‹... ⊆ (⋃n::nat. scaleR n ‘ closure B0)›
    using A-B0 by fastforce
  also have ‹... ⊆ closure (⋃n::nat. scaleR n ‘ B0)›
    by (metis (mono-tags, lifting) SUP-le-iff closure-closure closure-mono closure-scaleR clo-
sure-subset)
  also have ‹... = closure B1›
    using B1-def by fastforce
  finally have ‹L ⊆ closure B1›
    by (simp add: L-def closure-minimal)
  with ‹countable B1›
  show ‹∃B :: nat ⇒ -. L ⊆ closure (range B)›
    by (metis L-def closure-eq-empty empty-not-UNIV image-is-empty range-from-nat-into
subset-empty)
 qed
 define P T where ‹P n = Proj (ccspan (B ‘ {..n}))›
  and ‹T n = P n o_CL A› for n
 have ‹limitin cstrong-operator-topology T A sequentially›
 proof (intro limitin-cstrong-operator-topology[THEN iffD2, rule-format] metric-LIMSEQ-I)

  fix h and ε :: real assume ‹ε > 0›
  define Ah where ‹Ah = A h›
  have ‹Ah ∈ closure (range B)›
    by (metis L-def Ah-def ‹L ⊆ closure (range B)› cblinfun-apply-in-image
          cblinfun-image.rep-eq subsetD top-ccsubspace.rep-eq)
  then obtain x where ‹x ∈ range B› and ‹dist x Ah < ε›
    using ‹ε > 0› unfolding closure-approachable by blast
  then obtain n0 where x-n0: ‹x = B n0›
    by blast
  have ‹dist (P n *_V Ah) Ah < ε› if ‹n ≥ n0› for n
  proof −
    have ‹x ∈ space-as-set (P n *_S ⊤)›
      using ‹n ≥ n0›
      by (auto intro!: ccspan-superset' simp: P-def x-n0)
    from Proj-nearest[OF this, of Ah]
    have ‹dist (P n *_V Ah) Ah ≤ dist x Ah›
      by (simp add: P-def)
    with ‹dist x Ah < ε› show ?thesis
      by auto
  qed
  then show ‹∃n0. ∀n≥n0. dist (T n *_V h) (A *_V h) < ε›
    unfolding T-def Ah-def by auto
 qed
 then have ‹((λx. P x o_CL A) ⟶ A) sequentially›
  unfolding T-def
  by (auto intro!: bounded-products-sot-lim-imp-lim[where B=1] ‹compact-op A› norm-is-Proj
      simp: P-def)
 moreover have ‹finite-rank (P x o_CL A)› for x
  by (auto intro!: finite-rank-compose-right finite-rank-Proj-finite simp: P-def)
```

ultimately show ‹$A \in$ *closure* (*Collect finite-rank*)›
using *closure-sequential* **by** *force*
qed

typedef (**overloaded**) ('*a*::*chilbert-space*,'*b*::*complex-normed-vector*) *compact-op* =
‹*Collect compact-op* :: ('*a* $\Rightarrow_{CL}$ '*b*) *set*›
morphisms *from-compact-op Abs-compact-op*
by (*auto intro*!: *exI*[*of* - *0*])
setup-lifting *type-definition-compact-op*

instantiation *compact-op* :: (*chilbert-space*, *complex-normed-vector*) *complex-normed-vector* **begin**
lift-definition *scaleC-compact-op* :: ‹*complex* ⇒ ('*a*, '*b*) *compact-op* ⇒ ('*a*, '*b*) *compact-op*› **is** *scaleC* **by** *simp*
lift-definition *uminus-compact-op* :: ‹('*a*, '*b*) *compact-op* ⇒ ('*a*, '*b*) *compact-op*› **is** *uminus* **by** *simp*
lift-definition *zero-compact-op* :: ‹('*a*, '*b*) *compact-op*› **is** *0* **by** *simp*
lift-definition *minus-compact-op* :: ‹('*a*, '*b*) *compact-op* ⇒ ('*a*, '*b*) *compact-op* ⇒ ('*a*, '*b*) *compact-op*› **is** *minus* **by** *simp*
lift-definition *plus-compact-op* :: ‹('*a*, '*b*) *compact-op* ⇒ ('*a*, '*b*) *compact-op* ⇒ ('*a*, '*b*) *compact-op*› **is** *plus* **by** *simp*
lift-definition *sgn-compact-op* :: ‹('*a*, '*b*) *compact-op* ⇒ ('*a*, '*b*) *compact-op*› **is** *sgn* **by** *simp*
lift-definition *norm-compact-op* :: ‹('*a*, '*b*) *compact-op* ⇒ *real*› **is** *norm* **.**
lift-definition *scaleR-compact-op* :: ‹*real* ⇒ ('*a*, '*b*) *compact-op* ⇒ ('*a*, '*b*) *compact-op*› **is** *scaleR* **by** *simp*
lift-definition *dist-compact-op* :: ‹('*a*, '*b*) *compact-op* ⇒ ('*a*, '*b*) *compact-op* ⇒ *real*› **is** *dist* **.**
definition [*code del*]:
‹(*uniformity* :: (('*a*, '*b*) *compact-op* × ('*a*, '*b*) *compact-op*) *filter*) = (*INF e*∈{*0* <..}. *principal* {(*x*, *y*). *dist x y* < *e*})›
definition *open-compact-op* :: ('*a*, '*b*) *compact-op set* ⇒ *bool*
where [*code del*]: *open-compact-op S* = (∀ *x*∈*S*. ∀ *F* (*x'*, *y*) *in uniformity*. *x'* = *x* ⟶ *y* ∈ *S*)
instance
proof
show ((∗$_R$) *r* :: ('*a*, '*b*) *compact-op* ⇒ -) = (∗$_C$) (*complex-of-real r*) **for** *r*
by (*rule ext*, *transfer*) (*simp add*: *scaleR-scaleC*)
show *a* + *b* + *c* = *a* + (*b* + *c*)
for *a b c* :: ('*a*, '*b*) *compact-op*
by *transfer simp*
show *a* + *b* = *b* + *a*
for *a b* :: ('*a*, '*b*) *compact-op*
by *transfer simp*
show *0* + *a* = *a*
for *a* :: ('*a*, '*b*) *compact-op*
by *transfer simp*
show − (*a*::('*a*, '*b*) *compact-op*) + *a* = *0*
for *a* :: ('*a*, '*b*) *compact-op*
by *transfer simp*
show *a* − *b* = *a* + − *b*
for *a b* :: ('*a*, '*b*) *compact-op*

by *transfer simp*
show $a *_C (x + y) = a *_C x + a *_C y$
for a :: *complex* **and** $x\ y$ :: $('a, 'b)$ *compact-op*
by *transfer* (*simp add*: *scaleC-add-right*)
show $(a + b) *_C x = a *_C x + b *_C x$
for $a\ b$ :: *complex* **and** x :: $('a, 'b)$ *compact-op*
by *transfer* (*simp add*: *scaleC-left.add*)
show $a *_C b *_C x = (a * b) *_C x$
for $a\ b$ :: *complex* **and** x :: $('a, 'b)$ *compact-op*
by *transfer simp*
show $1 *_C x = x$
for x :: $('a, 'b)$ *compact-op*
by *transfer simp*
show *dist* $x\ y$ = *norm* $(x - y)$
for $x\ y$ :: $('a, 'b)$ *compact-op*
by *transfer* (*simp add*: *dist-norm*)
show $a *_R (x + y) = a *_R x + a *_R y$
for a :: *real* **and** $x\ y$ :: $('a, 'b)$ *compact-op*
by *transfer* (*simp add*: *scaleR-right-distrib*)
show $(a + b) *_R x = a *_R x + b *_R x$
for $a\ b$ :: *real* **and** x :: $('a, 'b)$ *compact-op*
by *transfer* (*simp add*: *scaleR-left.add*)
show $a *_R b *_R x = (a * b) *_R x$
for $a\ b$ :: *real* **and** x :: $('a, 'b)$ *compact-op*
by *transfer simp*
show $1 *_R x = x$
for x :: $('a, 'b)$ *compact-op*
by *transfer simp*
show *sgn* x = *inverse* (*norm* x) $*_R$ x
for x :: $('a, 'b)$ *compact-op*
by *transfer* (*simp add*: *sgn-div-norm*)
show *uniformity* = (*INF* e∈{0<..}. *principal* {(x, y). *dist* (x::$('a, 'b)$ *compact-op*) $y < e$})
using *uniformity-compact-op-def* **by** *blast*
show *open* U = ($\forall x \in U.\ \forall_F\ (x', y)$ *in uniformity*. $x' = x \longrightarrow y \in U$)
for U :: $('a, 'b)$ *compact-op set*
by (*simp add*: *open-compact-op-def*)
show (*norm* $x = 0$) $\longleftrightarrow$ $(x = 0)$
for x :: $('a, 'b)$ *compact-op*
by *transfer simp*
show *norm* $(x + y) \leq$ *norm* x + *norm* y
for $x\ y$ :: $('a, 'b)$ *compact-op*
by *transfer* (*use norm-triangle-ineq* **in** *blast*)
show *norm* $(a *_R x) = |a| *$ *norm* x
for a :: *real* **and** x :: $('a, 'b)$ *compact-op*
by *transfer simp*
show *norm* $(a *_C x)$ = *cmod* $a *$ *norm* x
for a :: *complex* **and** x :: $('a, 'b)$ *compact-op*
by *transfer simp*
qed

end

lemma *from-compact-op-plus*: ‹*from-compact-op* (*a* + *b*) = *from-compact-op a* + *from-compact-op b*›
by *transfer simp*

lemma *from-compact-op-scaleC*: ‹*from-compact-op* (*c* *$_C$ *a*) = *c* *$_C$ *from-compact-op a*›
by *transfer simp*

lemma *from-compact-op-norm*[*simp*]: ‹*norm* (*from-compact-op a*) = *norm a*›
by *transfer simp*

lemma *compact-op-butterfly*[*simp*]: ‹*compact-op* (*butterfly x y*)›
by (*simp add*: *finite-rank-compact-op*)

lift-definition *butterfly-co* :: ‹'*a*::*complex-normed-vector* ⇒ '*b*::*chilbert-space* ⇒ ('*b*,'*a*) *compact-op*› **is** *butterfly*
by *simp*

lemma *butterfly-co-add-left*: ‹*butterfly-co* (*a* + *a'*) *b* = *butterfly-co a b* + *butterfly-co a' b*›
by *transfer* (*rule butterfly-add-left*)

lemma *butterfly-co-add-right*: ‹*butterfly-co a* (*b* + *b'*) = *butterfly-co a b* + *butterfly-co a b'*›
by *transfer* (*rule butterfly-add-right*)

lemma *butterfly-co-scaleR-left*[*simp*]: *butterfly-co* (*r* *$_R$ *ψ*) *φ* = *r* *$_C$ *butterfly-co ψ φ*
by *transfer* (*rule butterfly-scaleR-left*)

lemma *butterfly-co-scaleR-right*[*simp*]: *butterfly-co ψ* (*r* *$_R$ *φ*) = *r* *$_C$ *butterfly-co ψ φ*
by *transfer* (*rule butterfly-scaleR-right*)

lemma *butterfly-co-scaleC-left*[*simp*]: *butterfly-co* (*r* *$_C$ *ψ*) *φ* = *r* *$_C$ *butterfly-co ψ φ*
by *transfer* (*rule butterfly-scaleC-left*)

lemma *butterfly-co-scaleC-right*[*simp*]: *butterfly-co ψ* (*r* *$_C$ *φ*) = *cnj r* *$_C$ *butterfly-co ψ φ*
by *transfer* (*rule butterfly-scaleC-right*)

lemma *finite-rank-separating-on-compact-op*:
fixes *F G* :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*) *compact-op* ⇒ '*c*::*complex-normed-vector*›
assumes ‹⋀*x*. *finite-rank* (*from-compact-op x*) ⟹ *F x* = *G x*›
assumes ‹*bounded-clinear F*›
assumes ‹*bounded-clinear G*›
shows ‹*F* = *G*›
proof −
define *FG* **where** ‹*FG x* = *F x* − *G x*› **for** *x*
from ‹*bounded-clinear F*› **and** ‹*bounded-clinear G*›
have ‹*bounded-clinear FG*›
by (*auto simp*: *FG-def*[*abs-def*] *intro*!: *bounded-clinear-sub*)

then have *contFG'*: ‹*continuous-map euclidean euclidean FG*›
by (*simp add*: *Complex-Vector-Spaces.bounded-clinear.bounded-linear linear-continuous-on*)
have ‹*continuous-on* (*Collect compact-op*) (*FG o Abs-compact-op*)›
proof
fix *a* :: *'a* ⇒$_{CL}$ *'b* **and** *e* :: *real*
assume *0* < *e* **and** *a-compact*: *a* ∈ *Collect compact-op*
have *dist-rw*: ‹*dist x' a* = *dist* (*Abs-compact-op x'*) (*Abs-compact-op a*)› **if** ‹*compact-op x'*› **for** *x'*
by (*metis Abs-compact-op-inverse a-compact dist-compact-op.rep-eq mem-Collect-eq that*)

from ‹*bounded-clinear FG*›
have ‹*continuous-on UNIV FG*›
using *contFG' continuous-map-iff-continuous2* **by** *blast*
then have ‹∃ *d*>*0*. ∀ *x'*. *dist x'* (*Abs-compact-op a*) < *d* ⟶ *dist* (*FG x'*) (*FG* (*Abs-compact-op a*)) ≤ *e*›
using ‹*e* > *0*› **by** (*force simp*: *continuous-on-iff*)
then have ‹∃ *d*>*0*. ∀ *x'*. *compact-op x'* ⟶ *dist* (*Abs-compact-op x'*) (*Abs-compact-op a*) < *d* ⟶
dist (*FG* (*Abs-compact-op x'*)) (*FG* (*Abs-compact-op a*)) ≤ *e*›
by *blast*
then show ∃ *d*>*0*. ∀ *x'*∈*Collect compact-op*. *dist x' a* < *d* ⟶ *dist* ((*FG* ∘ *Abs-compact-op*) *x'*) ((*FG* ∘ *Abs-compact-op*) *a*) ≤ *e*
by (*simp add*: *dist-rw o-def*)
qed
then have *contFG*: ‹*continuous-on* (*closure* (*Collect finite-rank*)) (*FG o Abs-compact-op*)›
by (*auto simp*: *compact-op-finite-rank*[*abs-def*])

have *FG0*: ‹*finite-rank a* ⟹ (*FG o Abs-compact-op*) *a* = *0*› **for** *a*
by (*metis* (*no-types*, *lifting*) *Abs-compact-op-inverse FG-def assms*(*1*) *closure-subset comp-apply compact-op-finite-rank eq-iff-diff-eq-0 mem-Collect-eq subset-eq*)

have ‹(*FG o Abs-compact-op*) *a* = *0*› **if** ‹*compact-op a*› **for** *a*
using *contFG FG0*
by (*rule continuous-constant-on-closure*) (*use that* **in** ‹*auto simp*: *compact-op-finite-rank*›)

then have ‹*FG a* = *0*› **for** *a*
by (*metis Abs-compact-op-cases comp-apply mem-Collect-eq*)

then show ‹*F* = *G*›
by (*auto simp*: *FG-def*[*abs-def*] *fun-eq-iff*)
qed

lemma *trunc-ell2-as-Proj*: ‹*trunc-ell2 S ψ* = *Proj* (*ccspan* (*ket* ‘ *S*)) *ψ*›
proof (*rule cinner-ket-eqI*)
fix *x*
have ∗: ‹*Proj* (*ccspan* (*ket* ‘ *S*)) (*ket x*) = *0*› **if** ‹*x* ∉ *S*›
by (*auto intro*!: *Proj-0-compl mem-ortho-ccspanI simp*: *that*)
have ‹*ket x* •$_C$ *trunc-ell2 S ψ* = *of-bool* (*x*∈*S*) ∗ (*ket x* •$_C$ *ψ*)›
by (*simp add*: *cinner-ket-left trunc-ell2.rep-eq*)

also have ‹... = *Proj* (*ccspan* (*ket* ‘ *S*)) (*ket x*) ⋅$_C$ *ψ*›
by (*cases* ‹*x* ∈ *S*›) (*auto simp add*: ∗ *ccspan-superset′ Proj-fixes-image*)
also have ‹... = *ket x* ⋅$_C$ (*Proj* (*ccspan* (*ket* ‘ *S*)) ∗$_V$ *ψ*)›
by (*simp add*: *adj-Proj flip*: *cinner-adj-left*)
finally show ‹*ket x* ⋅$_C$ *trunc-ell2 S ψ* = *ket x* ⋅$_C$ (*Proj* (*ccspan* (*ket* ‘ *S*)) ∗$_V$ *ψ*)› **.**
qed

lemma *unitary-between-bij-betw*:
assumes ‹*is-onb A*› ‹*is-onb B*›
shows ‹*bij-betw* ((∗$_V$) (*unitary-between A B*)) *A B*›
using *bij-between-bases-bij*[*OF assms*]
by (*rule bij-betw-cong*[*THEN iffD1*, *rotated*])
(*simp add*: *assms*(*1*) *assms*(*2*) *unitary-between-apply*)

lemma *tendsto-finite-subsets-at-top-image*:
assumes ‹*inj-on g X*›
shows ‹(*f* ⟶ *x*) (*finite-subsets-at-top* (*g* ‘ *X*)) ⟷ ((λ*S*. *f* (*g* ‘ *S*)) ⟶ *x*) (*finite-subsets-at-top X*)›
by (*simp add*: *filterlim-def assms o-def*
flip: *filtermap-image-finite-subsets-at-top filtermap-compose*)

lemma *Proj-onb-limit*:
shows ‹*is-onb A* ⟹ ((λ*S*. *Proj* (*ccspan S*) *ψ*) ⟶ *ψ*) (*finite-subsets-at-top A*)›
proof −
have *main*: ‹((λ*S*. *Proj* (*ccspan S*) *ψ*) ⟶ *ψ*) (*finite-subsets-at-top A*)› **if** ‹*is-onb A*›
for *ψ* :: ‹*′b*::{*chilbert-space*,*not-singleton*}› **and** *A*
proof −
define *U* **where** ‹*U* = *unitary-between* (*ell2-to-hilbert*∗ ‘ *A*) (*range ket*)›
have [*simp*]: ‹*unitary U*›
by (*simp add*: *U-def that unitary-between-unitary unitary-image-onb*)
have *lim1*: ‹((λ*S*. *trunc-ell2 S* (*U* ∗$_V$ *ell2-to-hilbert*∗ ∗$_V$ *ψ*)) ⟶ *U* ∗$_V$ *ell2-to-hilbert*∗ ∗$_V$ *ψ*) (*finite-subsets-at-top UNIV*)›
by (*rule trunc-ell2-lim-at-UNIV*)
have *lim2*: ‹((λ*S*. *ell2-to-hilbert* ∗$_V$ *U*∗ ∗$_V$ *trunc-ell2 S* (*U* ∗$_V$ *ell2-to-hilbert*∗ ∗$_V$ *ψ*)) ⟶ *ell2-to-hilbert* ∗$_V$ *U*∗ ∗$_V$ *U* ∗$_V$ *ell2-to-hilbert*∗ ∗$_V$ *ψ*) (*finite-subsets-at-top UNIV*)›
by (*intro cblinfun.tendsto lim1*) *auto*
have ∗: ‹*ell2-to-hilbert* ∗$_V$ *U*∗ ∗$_V$ *trunc-ell2 S* (*U* ∗$_V$ *ell2-to-hilbert*∗ ∗$_V$ *ψ*)
= *Proj* (*ccspan* ((*ell2-to-hilbert o U*∗ *o ket*) ‘ *S*)) *ψ*› (**is** ‹*?lhs* = *?rhs*›) **for** *S*
proof −
have ‹*?lhs* = (*sandwich ell2-to-hilbert* ∗$_V$ *sandwich* (*U*∗) ∗$_V$ *Proj* (*ccspan* (*ket* ‘ *S*))) ∗$_V$ *ψ*›
by (*simp add*: *trunc-ell2-as-Proj sandwich-apply*)
also have ‹... = *Proj* (*ell2-to-hilbert* ∗$_S$ *U*∗ ∗$_S$ *ccspan* (*ket* ‘ *S*)) ∗$_V$ *ψ*›
by (*simp add*: *Proj-sandwich*)
also have ‹... = *Proj* (*ccspan* (*ell2-to-hilbert* ‘ *U*∗ ‘ *ket* ‘ *S*)) ∗$_V$ *ψ*›
by (*simp add*: *cblinfun-image-ccspan*)
also have ‹... = *?rhs*›

```
      by (simp add: image-comp)
    finally show ?thesis
      by −
  qed
  have **: ‹ell2-to-hilbert *V U* *V U *V ell2-to-hilbert* *V ψ = ψ›
    by (simp add: lift-cblinfun-comp[OF unitaryD1] lift-cblinfun-comp[OF unitaryD2])
  have ***: ‹range (ell2-to-hilbert o U* o ket) = A› (is ‹?lhs = -›)
  proof −
    have ‹bij-betw U (ell2-to-hilbert* ‘ A) (range ket)›
      by (auto intro!: unitary-between-bij-betw that unitary-image-onb simp add: U-def)
    then have bijUadj: ‹bij-betw (U*) (range ket) (ell2-to-hilbert* ‘ A)›
    by (metis ‹unitary U› bij-betw-imp-surj-on inj-imp-bij-betw-inv unitary-adj-inv unitary-inj)
    have ‹?lhs = ell2-to-hilbert ‘ U* ‘ range ket›
      by (simp add: image-comp)
    also from this and bijUadj have ‹... = ell2-to-hilbert ‘ (ell2-to-hilbert* ‘ A)›
      by (metis bij-betw-imp-surj-on)
    also have ‹... = A›
      by (metis image-inv-f-f unitary-adj unitary-adj-inv unitary-ell2-to-hilbert unitary-inj)
    finally show ?thesis
      by −
  qed
   from lim2 have lim3: ‹((λS. Proj (ccspan ((ell2-to-hilbert o U* o ket) ‘ S)) ψ) ⟶ ψ)
(finite-subsets-at-top UNIV)›
    unfolding * ** by −
 then have lim4: ‹((λS. Proj (ccspan S) ψ) ⟶ ψ) (finite-subsets-at-top (range (ell2-to-hilbert
o U* o ket)))›
    by (rule tendsto-finite-subsets-at-top-image[THEN iffD2, rotated])
       (intro inj-compose unitary-inj unitary-ell2-to-hilbert unitary-adj[THEN iffD2] ‹unitary
U› inj-ket)
  then show ?thesis
    unfolding *** by −
 qed
 assume ‹is-onb A›
 show ?thesis
 proof (cases ‹class.not-singleton TYPE('a)›)
  case True
  show ?thesis
    using chilbert-space-class.chilbert-space-axioms True ‹is-onb A›
    by (rule main[internalize-sort' 'b2])
 next
  case False
  then have ‹ψ = 0›
    by (rule not-not-singleton-zero)
  then show ?thesis
    by simp
 qed
qed

lemma is-ortho-setD:
```

```
  assumes is-ortho-set S x ∈ S y ∈ S x ≠ y
  shows   x ·_C y = 0
  using assms unfolding is-ortho-set-def by blast

lemma finite-rank-dense-compact:
  fixes A :: ‹'a::chilbert-space set› and B :: ‹'b::chilbert-space set›
  assumes ‹is-onb A› and ‹is-onb B›
  shows ‹closure (cspan ((λ(ξ,η). butterfly ξ η) ‘ (A × B))) = Collect compact-op›
proof (rule Set.equalityI)
  show ‹closure (cspan ((λ(ξ,η). butterfly ξ η) ‘ (A × B))) ⊆ Collect compact-op›
  proof −
    have ‹closure (cspan ((λ(ξ,η). butterfly ξ η) ‘ (A × B))) ⊆ closure (Collect finite-rank)›
    proof (rule closure-mono; safe)
      fix x assume x ∈ cspan ((λ(ξ, η). butterfly ξ η) ‘ (A × B))
      thus finite-rank x
        by (induction rule: complex-vector.span-induct-alt) auto
    qed
    also have ‹... = Collect compact-op›
      by (simp add: Set.set-eqI compact-op-finite-rank)
    finally show ?thesis
      by −
  qed
  show ‹Collect compact-op ⊆ closure (cspan ((λ(ξ,η). butterfly ξ η) ‘ (A × B)))›
  proof −
    have ‹Collect (compact-op :: 'b⇒_CL'a ⇒ -) = closure (cspan (Collect rank1))›
      by (simp add: compact-op-finite-rank[abs-def] finite-rank-def[abs-def])
    also have ‹... ⊆ closure (cspan (closure (cspan ((λ(ξ,η). butterfly ξ η) ‘ (A × B)))))›
    proof (rule closure-mono, rule complex-vector.span-mono, rule subsetI)
      fix x :: ‹'b ⇒_CL 'a› assume ‹x ∈ Collect rank1›
      then obtain a b where xab: ‹x = butterfly a b›
        using rank1-iff-butterfly by fastforce
      define f where ‹f F G = butterfly (Proj (ccspan F) a) (Proj (ccspan G) b)› for F G
      have lim: ‹(case-prod f ⟶⟶ x) (finite-subsets-at-top A ×_F finite-subsets-at-top B)›
      proof (rule tendstoI, subst dist-norm)
        fix e :: real assume ‹e > 0›
        define d where ‹d = (if norm a = 0 ∧ norm b = 0 then 1
                                 else e / (max (norm a) (norm b)) / 4)›
        have [simp]: d > 0
          unfolding d-def using ‹e > 0›
          by (auto intro!: divide-pos-pos simp: less-max-iff-disj)
        have d: ‹norm a * d + norm a * d + norm b * d < e›
        proof −
          have ‹x * d ≤ e / 4› if x: x ∈ {norm a, norm b} for x
          proof (cases x = 0)
            case False
            have d: d = e / (max (norm a) (norm b)) / 4
              using False x by (auto simp: d-def)
            have d ≤ e / x / 4
              unfolding d by (intro divide-left-mono divide-right-mono)
```

```
                (use x ‹d > 0› ‹e > 0› False in ‹auto simp: less-max-iff-disj›)
    thus ?thesis
      using False x by (auto simp: field-simps)
  qed (use ‹e > 0› in auto)
  hence norm a * d ≤ e / 4 norm b * d ≤ e / 4
    by blast+
  hence ‹norm a * d + norm a * d + norm b * d ≤ 3 * e / 4›
    by linarith
  also have ‹... < e›
    by (simp add: ‹0 < e›)
  finally show ?thesis .
qed
from Proj-onb-limit[where ψ=a, OF assms(1)]
have ‹∀F F in finite-subsets-at-top A. norm (Proj (ccspan F) a − a) < d›
  by (metis Lim-null ‹0 < d› order-tendstoD(2) tendsto-norm-zero-iff)
moreover from Proj-onb-limit[where ψ=b, OF assms(2)]
have ‹∀F G in finite-subsets-at-top B. norm (Proj (ccspan G) b − b) < d›
  by (metis Lim-null ‹0 < d› order-tendstoD(2) tendsto-norm-zero-iff)
ultimately have FG-close: ‹∀F (F,G) in finite-subsets-at-top A ×F finite-subsets-at-top
B.
      norm (Proj (ccspan F) a − a) < d ∧ norm (Proj (ccspan G) b − b) < d›
  unfolding case-prod-beta
  by (rule eventually-prodI)
have fFG-dist: ‹norm (f F G − x) < e›
  if ‹norm (Proj (ccspan F) a − a) < d› and ‹norm (Proj (ccspan G) b − b) < d›
    and ‹F ⊆ A› and ‹G ⊆ B› for F G
proof −
  have a-split: ‹a = Proj (ccspan F) *V a + Proj (ccspan (A−F)) *V a›
  proof −
    have A: is-ortho-set A ccspan A = ⊤
      using assms unfolding is-onb-def by auto
    have Proj (ccspan (F ∪ A)) = Proj (ccspan F) + Proj (ccspan (A−F))
      by (subst Proj-orthog-ccspan-union [symmetric])
         (use that in ‹auto intro!: is-ortho-setD[OF A(1)]›)
    also have F ∪ A = A
      using that by blast
    finally show ?thesis
      using A(2) by (simp flip: cblinfun.add-left)
  qed

  have b-split: ‹b = Proj (ccspan G) *V b + Proj (ccspan (B−G)) *V b›
  proof −
    have B: is-ortho-set B ccspan B = ⊤
      using assms unfolding is-onb-def by auto
    have Proj (ccspan (G ∪ B)) = Proj (ccspan G) + Proj (ccspan (B−G))
      by (subst Proj-orthog-ccspan-union [symmetric])
         (use that in ‹auto intro!: is-ortho-setD[OF B(1)]›)
    also have G ∪ B = B
      using that by blast
```

```
      finally show ?thesis
        using B(2) by (simp flip: cblinfun.add-left)
    qed

    have n1: ‹norm (f F (B−G)) ≤ norm a * d› for F
    proof −
      have ‹norm (f F (B−G)) ≤ norm a * norm (Proj (ccspan (B−G)) b)›
      by (auto intro!: mult-right-mono is-Proj-reduces-norm simp add: f-def norm-butterfly)
      also have ‹... ≤ norm a * norm (Proj (ccspan G) b − b)›
        by (metis add-diff-cancel-left' b-split less-eq-real-def norm-minus-commute)
      also have ‹... ≤ norm a * d›
        by (meson less-eq-real-def mult-left-mono norm-ge-zero that(2))
      finally show ?thesis
        by −
    qed
    have n2: ‹norm (f (A−F) G) ≤ norm b * d› for G
    proof −
      have ‹norm (f (A−F) G) ≤ norm b * norm (Proj (ccspan (A−F)) a)›
       by (auto intro!: mult-right-mono is-Proj-reduces-norm simp add: f-def norm-butterfly
mult.commute)
      also have ‹... ≤ norm b * norm (Proj (ccspan F) a − a)›
        by (smt (verit, best) a-split add-diff-cancel-left' minus-diff-eq norm-minus-cancel)
      also have ‹... ≤ norm b * d›
        by (meson less-eq-real-def mult-left-mono norm-ge-zero that(1))
      finally show ?thesis
        by −
    qed
    have ‹norm (f F G − x) = norm (− f F (B−G) − f (A−F) (B−G) − f (A−F) G)›
      unfolding xab
      by (subst a-split, subst b-split)
         (simp add: f-def butterfly-add-right butterfly-add-left)
     also have ‹... ≤ norm (f F (B−G)) + norm (f (A−F) (B−G)) + norm (f (A−F)
G)›
      by (smt (verit, best) norm-minus-cancel norm-triangle-ineq4)
    also have ‹... ≤ norm a * d + norm a * d + norm b * d›
      using n1 n2
      by (meson add-mono-thms-linordered-semiring(1))
    also have ‹... < e›
      by (fact d)
    finally show ?thesis
      by −
  qed
  have ∀F (F, G) in finite-subsets-at-top A ×F finite-subsets-at-top B.
        (finite F ∧ F ⊆ A) ∧ finite G ∧ G ⊆ B
    unfolding case-prod-unfold by (intro eventually-prodI) auto
  thus ‹∀F FG in finite-subsets-at-top A ×F finite-subsets-at-top B.
        norm ((case FG of (F, G) ⇒ f F G) − x) < e›
    using FG-close by eventually-elim (use fFG-dist in auto)
qed
```

```
    have nontriv: ‹finite-subsets-at-top A ×_F finite-subsets-at-top B ≠ ⊥›
      by (simp add: prod-filter-eq-bot)
    have inside: ‹∀_F x in finite-subsets-at-top A ×_F finite-subsets-at-top B.
          case-prod f x ∈ cspan ((λ(ξ,η). butterfly ξ η) ‘ (A × B))›
    proof (rule eventually-mp[where P=‹λ(F,G). finite F ∧ finite G›])
     show ‹∀_F (F,G) in finite-subsets-at-top A ×_F finite-subsets-at-top B. finite F ∧ finite G›
     by (smt (verit) case-prod-conv eventually-finite-subsets-at-top-weakI eventually-prod-filter)
      have f-in-span: ‹f F G ∈ cspan ((λ(ξ,η). butterfly ξ η) ‘ (A × B))› if [simp]: ‹finite F›
‹finite G› and ‹F ⊆ A› ‹G ⊆ B› for F G
      proof −
        have ‹Proj (ccspan F) a ∈ cspan F›
          by (metis Proj-range cblinfun-apply-in-image ccspan-finite that(1))
        then obtain r where ProjFsum: ‹Proj (ccspan F) a = (∑ x∈F. r x *_C x)›
          using complex-vector.span-finite[OF ‹finite F›] by auto
        have ‹Proj (ccspan G) b ∈ cspan G›
          by (metis Proj-range cblinfun-apply-in-image ccspan-finite that(2))
        then obtain s where ProjGsum: ‹Proj (ccspan G) b = (∑ x∈G. s x *_C x)›
          using complex-vector.span-finite[OF ‹finite G›] by auto
        have ‹butterfly ξ η ∈ (λ(ξ, η). butterfly ξ η) ‘ (A × B)›
          if ‹η ∈ G› and ‹ξ ∈ F› for η ξ
          using ‹F ⊆ A› ‹G ⊆ B› that by auto
        then show ?thesis
          by (auto intro!: complex-vector.span-sum complex-vector.span-scale
              complex-vector.span-base[where a=‹butterfly - -›]
              simp add: f-def ProjFsum ProjGsum butterfly-sum-left butterfly-sum-right)
      qed
      have ∀_F (F, G) in finite-subsets-at-top A ×_F finite-subsets-at-top B.
            (finite F ∧ F ⊆ A) ∧ finite G ∧ G ⊆ B
        unfolding case-prod-unfold by (intro eventually-prodI) auto
      thus ‹∀_F x in finite-subsets-at-top A ×_F finite-subsets-at-top B.
                  (case x of (F, G) ⇒ finite F ∧ finite G) ⟶ (case x of (F, G) ⇒ f F G) ∈
cspan ((λ(ξ, η). butterfly ξ η) ‘ (A × B))›
        by eventually-elim (auto intro!: f-in-span)
    qed
    show ‹x ∈ closure (cspan ((λ(ξ, η). butterfly ξ η) ‘ (A × B)))›
      using lim nontriv inside by (rule limit-in-closure)
  qed
  also have ‹... = closure (cspan ((λ(ξ,η). butterfly ξ η) ‘ (A × B)))›
    by (simp add: complex-vector.span-eq-iff[THEN iffD2])
  finally show ?thesis
    by −
 qed
qed

lemma compact-op-comp-left: ‹compact-op (a o_CL b)› if ‹compact-op a›
 for a b :: ‹-::chilbert-space ⇒_CL -::chilbert-space›
proof −
 from that have ‹a ∈ closure (Collect finite-rank)›
 using compact-op-finite-rank by blast
```

then have ‹*a* o_{CL} *b* ∈ (λ*a*. *a* o_{CL} *b*) ‘ *closure* (*Collect finite-rank*)›
by *blast*
also have ‹... ⊆ *closure* ((λ*a*. *a* o_{CL} *b*) ‘ *Collect finite-rank*)›
by (*auto intro*!: *closure-bounded-linear-image-subset bounded-clinear.bounded-linear bounded-clinear-cblinfun-compo*
also have ‹... ⊆ *closure* (*Collect finite-rank*)›
by (*auto intro*!: *closure-mono finite-rank-comp-left*)
finally show ‹*compact-op* (*a* o_{CL} *b*)›
using *compact-op-finite-rank* **by** *blast*
qed

lemma *compact-op-eigenspace-finite-dim*:
fixes *a* :: ‹'*a* ⇒$_{CL}$ '*a*::*chilbert-space*›
assumes ‹*compact-op a*›
assumes ‹*e* ≠ *0*›
shows ‹*finite-dim-ccsubspace* (*eigenspace e a*)›
proof −
define *S* **where** ‹*S* = *space-as-set* (*eigenspace e a*)›
obtain *B* **where** ‹*ccspan B* = *eigenspace e a*› **and** ‹*is-ortho-set B*›
and *norm-B*: ‹*x* ∈ *B* ⟹ *norm x* = *1*› **for** *x*
using *orthonormal-subspace-basis-exists*[**where** *S*=‹{}› **and** *V*=‹*eigenspace e a*›]
by (*auto simp*: *S-def*)
then have *span-BS*: ‹*closure* (*cspan B*) = *S*›
by (*metis S-def ccspan.rep-eq*)
have ‹*finite B*›
proof (*rule ccontr*)
assume ‹*infinite B*›
then obtain *b* :: ‹*nat* ⇒ '*a*› **where** *range-b*: ‹*range b* ⊆ *B*› **and** ‹*inj b*›
by (*meson infinite-countable-subset*)
define *f* **where** ‹*f n* = *a* (*b n*)› **for** *n*
have *range-f*: ‹*range f* ⊆ *closure* (*a* ‘ *cball 0 1*)›
using *norm-B range-b*
by (*auto intro*!: *closure-subset*[*THEN subsetD*] *imageI simp*: *f-def*)
from ‹*compact-op a*› **have** *compact*: ‹*compact* (*closure* (*a* ‘ *cball 0 1*))›
using *compact-op-def2* **by** *blast*
obtain *l r* **where** ‹*strict-mono r*› **and** *fr-lim*: ‹(*f o r*) ⟶ *l*›
using *range-f compact*[*unfolded compact-def*, *rule-format*, *of f*]
by *fast*
define *d* :: *real* **where** ‹*d* = *cmod e* ∗ *sqrt 2*›
from ‹*e* ≠ *0*› **have** ‹*d* > *0*›
by (*auto intro*!: *Rings.linordered-semiring-strict-class.mult-pos-pos simp*: *d-def*)
have *aux*: ‹∃ *n*≥*N*. *P n*› **if** ‹*P* (*Suc N*)› **for** *P N*
using *Suc-n-not-le-n nat-le-linear that* **by** *blast*
have ‹*dist* (*f* (*r n*)) (*f* (*r* (*Suc n*))) = *d*› **for** *n*
proof −
have *ortho*: ‹*is-orthogonal* (*b* (*r n*)) (*b* (*r* (*Suc n*)))›
proof −
have ‹*b* (*r n*) ≠ *b* (*r* (*Suc n*))›
by (*metis Suc-n-not-n* ‹*inj b*› ‹*strict-mono r*› *injD strict-mono-eq*)

```
      moreover from range-b have ‹b (r n) ∈ B› and ‹b (r (Suc n)) ∈ B›
        by fast+
      ultimately show ?thesis
        using ‹is-ortho-set B›
        by (auto intro!: simp: is-ortho-set-def)
    qed
    have normb: ‹norm (b n) = 1› for n
      by (metis ‹inj b› image-subset-iff inj-image-mem-iff norm-B range-b range-eqI)
    have ‹f (r n) = e *C b (r n)› for n
    proof −
      from range-b span-BS
      have ‹b (r n) ∈ S›
        using complex-vector.span-superset closure-subset
        by (blast dest: range-subsetD[where i = ‹b n›])
      then show ?thesis
        by (auto intro!: dest!: eigenspace-memberD simp: S-def f-def)
    qed
    then have ‹(dist (f (r n)) (f (r (Suc n))))^2 = (cmod e * dist (b (r n)) (b (r (Suc n))))^2›
      by (simp add: dist-norm flip: scaleC-diff-right)
    also from ortho have ‹... = (cmod e * sqrt 2)^2›
      by (simp add: dist-norm polar-identity-minus power-mult-distrib normb)
    finally show ?thesis
      by (simp add: d-def)
  qed
  with ‹d > 0› have ‹¬ Cauchy (f o r)›
    by (auto intro!: exI[of - ‹d/2›] aux
        simp: Cauchy-altdef2 dist-commute simp del: less-divide-eq-numeral1)
  with fr-lim show False
    using LIMSEQ-imp-Cauchy by blast
qed
with span-BS show ?thesis
  using S-def cspan-finite-dim finite-dim-ccsubspace.rep-eq by fastforce
qed

lemma eigenvalue-in-the-limit-compact-op:
— [2], Proposition II.4.14
assumes ‹compact-op T›
assumes ‹l ≠ 0›
assumes normh: ‹⋀n. norm (h n) = 1›
assumes Tl-lim: ‹(λn. (T − l *C id-cblinfun) (h n)) ⟶ 0›
shows ‹l ∈ eigenvalues T›
proof −
from assms(1)
have compact-Tball: ‹compact (closure (T ‘ cball 0 1))›
  using compact-op-def2 by blast
have ‹T (h n) ∈ closure (T ‘ cball 0 1)› for n
  by (smt (z3) assms(3) closure-subset image-subset-iff mem-cball-0)
then obtain n f where lim-Thn: ‹(λk. T (h (n k))) ⟶ f› and ‹strict-mono n›
  using compact-Tball[unfolded compact-def, rule-format, where f=‹T o h›, unfolded o-def]
```

```
    by fast
  have lThk-lim: ‹(λk. (l *C id-cblinfun − T) (h (n k))) ⟶ 0›
  proof −
    have ‹(λn. (l *C id-cblinfun − T) (h n)) ⟶ 0›
      using Tl-lim[THEN tendsto-minus]
      by (simp add: cblinfun.diff-left)
    with ‹strict-mono n› show ?thesis
      by (rule LIMSEQ-subseq-LIMSEQ[unfolded o-def, rotated])
  qed
  have ‹h (n k) = inverse l *C ((l *C id-cblinfun − T) (h (n k)) + T (h (n k)))› for k
   by (metis assms(2) cblinfun.real.add-left cblinfun.scaleC-left diff-add-cancel divideC-field-splits-simps-1(5)
id-cblinfun.rep-eq scaleC-zero-right)
  moreover have ‹... ⟶ inverse l *C (0 + f)›
    by (intro tendsto-intros lThk-lim lim-Thn)
  ultimately have lim-hn: ‹(λk. h (n k)) ⟶ inverse l *C f›
    by simp
  have ‹f ≠ 0›
  proof −
    from lim-hn have ‹(λk. norm (h (n k))) ⟶ norm (inverse l *C f)›
      by (rule isCont-tendsto-compose[unfolded o-def, rotated]) fastforce
    moreover have ‹(λ-. 1) ⟶ 1›
      by simp
    ultimately have ‹norm (inverse l *C f) = 1›
      unfolding normh
      using LIMSEQ-unique by blast
    then show ‹f ≠ 0›
      by force
  qed
  from lim-hn have ‹(λk. T (h (n k))) ⟶ T (inverse l *C f)›
    by (rule isCont-tendsto-compose[rotated]) force
  with lim-Thn have ‹f = T (inverse l *C f)›
    using LIMSEQ-unique by blast
  with ‹l ≠ 0› have ‹l *C f = T f›
    by (metis cblinfun.scaleC-right divideC-field-simps(2))
  with ‹f ≠ 0› show ‹l ∈ eigenvalues T›
    by (auto intro!: eigenvaluesI[where h=f])
qed


lemma norm-is-eigenvalue:
  — [2], Proposition II.5.9
  fixes a :: ‹'a ⇒CL 'a::{not-singleton, chilbert-space}›
  assumes ‹compact-op a›
  assumes ‹selfadjoint a›
  shows ‹norm a ∈ eigenvalues a ∨ − norm a ∈ eigenvalues a›
proof −
  wlog ‹a ≠ 0›
    using negation by auto
  obtain h e where h-lim: ‹(λi. h i •C a (h i)) ⟶ e› and normh: ‹norm (h i) = 1›
```

```
    and norme: ‹cmod e = norm a› for i
  proof atomize-elim
    have sgn-cmod: ‹sgn x * cmod x = x› for x
      by (simp add: complex-of-real-cmod sgn-mult-abs)
    from cblinfun-norm-is-Sup-cinner[OF ‹selfadjoint a›]
    obtain f where range-f: ‹range f ⊆ ((λψ. cmod (ψ •C (a *V ψ))) ‘ {ψ. norm ψ = 1})›
      and f-lim: ‹f ⟶ norm a›
      by (atomize-elim, rule is-Sup-imp-ex-tendsto) (auto simp: ex-norm1-not-singleton)
    obtain h0 where normh0: ‹norm (h0 i) = 1› and f-h0: ‹f i = cmod (h0 i •C a (h0 i))›
for i
      by (atomize-elim, rule choice2) (use range-f in auto)
    from f-h0 f-lim have h0lim-cmod: ‹(λi. cmod (h0 i •C a (h0 i))) ⟶ norm a›
      by presburger
    have sgn-sphere: ‹sgn (h0 i •C a (h0 i)) ∈ insert 0 (sphere 0 1)› for i
      using normh0 by (auto intro!: left-inverse simp: sgn-div-norm)
    have compact: ‹compact (insert 0 (sphere (0::complex) 1))›
      by simp
    obtain r l where ‹strict-mono r› and l-sphere: ‹l ∈ insert 0 (sphere 0 1)›
      and h0lim-sgn: ‹((λi. sgn (h0 i •C a (h0 i))) ∘ r) ⟶ l›
      using compact[unfolded compact-def, rule-format, OF sgn-sphere]
      by fast
    define h and e where ‹h i = h0 (r i)› and ‹e = l * norm a› for i
    have hlim-cmod: ‹(λi. cmod (h i •C a (h i))) ⟶ norm a›
      using LIMSEQ-subseq-LIMSEQ[OF h0lim-cmod ‹strict-mono r›]
      unfolding h-def o-def by auto
    with h0lim-sgn have ‹(λi. sgn (h i •C a (h i)) * cmod (h i •C a (h i))) ⟶ e›
      by (auto intro!: tendsto-mult tendsto-of-real simp: o-def h-def e-def)
    then have hlim: ‹(λi. h i •C a (h i)) ⟶ e›
      by (simp add: sgn-cmod)
    have ‹e ≠ 0›
    proof (rule ccontr, unfold not-not)
      assume ‹e = 0›
      from hlim have ‹(λi. cmod (h i •C a (h i))) ⟶ cmod e›
        by (rule tendsto-compose[where g=cmod, rotated])
           (smt (verit, del-insts) ‹e = 0› diff-zero dist-norm metric-LIM-imp-LIM
              norm-ge-zero norm-zero real-norm-def tendsto-ident-at)
      with ‹e = 0› hlim-cmod have ‹norm a = 0›
        using LIMSEQ-unique by fastforce
      with ‹a ≠ 0› show False
        by simp
    qed
    then have norme: ‹norm e = norm a›
      using l-sphere by (simp add: e-def norm-mult)
    show ‹∃h e. (λi. h i •C (a *V h i)) ⟶ e ∧ (∀i. norm (h i) = 1) ∧ cmod e = norm a›
      using norme normh0 hlim
      by (auto intro!: exI[of - h] exI[of - e] simp: h-def)
  qed
  have ‹e ∈ ℝ›
  proof −
```

```
    from h-lim[THEN tendsto-Im]
    have ∗: ‹(λi. Im (h i •C a (h i))) ⟶ Im e›
      by −
    have ∗∗: ‹Im (h i •C a (h i)) = 0› for i
      using assms(2) selfadjoint-def cinner-hermitian-real complex-is-Real-iff by auto
    have ‹Im e = 0›
      using - ∗ by (rule tendsto-unique) (use ∗∗ in auto)
    then show ?thesis
      using complex-is-Real-iff by presburger
  qed
  define e' where ‹e' = Re e›
  with ‹e ∈ ℝ› have ee': ‹e = of-real e'›
    by simp
  have ‹e' ∈ eigenvalues a›
  proof −
    have [trans]: ‹f ⟶ 0› if ‹⋀x. f x ≤ g x› and ‹g ⟶ 0› and ‹⋀x. f x ≥ 0› for f g
:: ‹nat ⇒ real›
      by (rule real-tendsto-sandwich[where h=g and f=‹λ-. 0›]) (use that in auto)
    have [simp]: a∗ = a
      using assms(2) by (simp add: selfadjoint-def)
    have [simp]: Re (h x •C h x) = 1 for x
      using normh[of x] by (simp flip: power2-norm-eq-cinner')
    have ‹(norm ((a − e' ∗R id-cblinfun) (h n)))^2 = (norm (a (h n)))^2 − 2 ∗ e' ∗ Re (h n •C
a (h n)) + e'^2› for n
      by (simp add: power2-norm-eq-cinner' algebra-simps cblinfun.cbilinear-simps
                    cblinfun.scaleR-left power2-eq-square[of e'] flip: cinner-adj-right)
    also have ‹... n ≤ e'^2 − 2 ∗ e' ∗ Re (h n •C a (h n)) + e'^2› for n
    proof −
      from norme have ‹e'^2 = (norm a)^2›
        by (auto simp: ee' power2-eq-iff abs-if split: if-splits)
      then have ‹(norm (a ∗V h n))^2 ≤ e'^2›
        using norm-cblinfun[of a h n] by (simp add: normh)
      then show ?thesis
        by auto
    qed
    also have ‹... ⟶ 0›
     apply (subst asm-rl[of ‹(λn. e'^2 − 2 ∗ e' ∗ Re (h n •C a (h n)) + e'^2) = (λn. 2 ∗ e' ∗ (e'
− Re (h n •C (a ∗V h n))))›])
      subgoal
        by (auto simp: fun-eq-iff right-diff-distrib power2-eq-square)[1]
      subgoal
        using h-lim[THEN tendsto-Re]
        by (auto intro!: tendsto-mult-right-zero tendsto-diff-const-left-rewrite simp: ee')
      done
    finally have ‹(λn. (a − e' ∗R id-cblinfun) (h n)) ⟶ 0›
      by (simp add: tendsto-norm-zero-iff)
    then show ‹e' ∈ eigenvalues a›
      unfolding scaleR-scaleC
      by (rule eigenvalue-in-the-limit-compact-op[rotated −1])
```

```
      (use ‹a ≠ 0› norme in ‹auto intro!: normh simp: assms ee'›)
  qed
  from ‹e ∈ ℝ› norme
  have ‹e = norm a ∨ e = − norm a›
    by (smt (verit, best) in-Reals-norm of-real-Re)
  with ‹e' ∈ eigenvalues a› show ?thesis
    using ee' by presburger
qed

lemma
  fixes a :: ‹'a ⇒CL 'a::{not-singleton, chilbert-space}›
  assumes ‹compact-op a›
  assumes ‹selfadjoint a›
  shows largest-eigenvalue-norm-aux: ‹largest-eigenvalue a ∈ {norm a, − norm a}›
    and largest-eigenvalue-ex: ‹largest-eigenvalue a ∈ eigenvalues a›
proof −
  define l where ‹l = (SOME x. x ∈ eigenvalues a ∧ (∀ y ∈ eigenvalues a. cmod x ≥ cmod y))›
  from norm-is-eigenvalue[OF assms]
  obtain e where ‹e ∈ {of-real (norm a), − of-real (norm a)}› and ‹e ∈ eigenvalues a›
    by auto
  then have norme: ‹norm e = norm a›
    by auto
  have ‹e ∈ eigenvalues a ∧ (∀ y∈eigenvalues a. cmod y ≤ cmod e)› (is ‹?P e›)
    by (auto intro!: ‹e ∈ eigenvalues a› simp: eigenvalue-norm-bound norme)
  then have ∗: ‹l ∈ eigenvalues a ∧ (∀ y∈eigenvalues a. cmod y ≤ cmod l)›
    unfolding l-def largest-eigenvalue-def by (rule someI)
  then have l-def': ‹l = largest-eigenvalue a›
    by (metis (mono-tags, lifting) l-def largest-eigenvalue-def)
  from ∗ have ‹l ∈ eigenvalues a›
    by (simp add: l-def)
  then show ‹largest-eigenvalue a ∈ eigenvalues a›
    by (simp add: l-def')
  have ‹norm l ≥ norm a›
    using ∗ norme ‹e ∈ eigenvalues a› by auto
  moreover have ‹norm l ≤ norm a›
    using ∗ eigenvalue-norm-bound by blast
  ultimately have ‹norm l = norm a›
    by linarith
  moreover have ‹l ∈ ℝ›
    using ‹l ∈ eigenvalues a› assms(2) eigenvalue-selfadj-real by blast
  ultimately have ‹l ∈ {norm a, − norm a}›
    by (smt (verit, ccfv-SIG) in-Reals-norm insertCI l-def of-real-Re)
  then show ‹largest-eigenvalue a ∈ {norm a, − norm a}›
    by (simp add: l-def')
qed

lemma largest-eigenvalue-norm:
  fixes a :: ‹'a ⇒CL 'a::chilbert-space›
  assumes ‹compact-op a›
```

assumes ‹*selfadjoint a*›
shows ‹*largest-eigenvalue a* ∈ {*norm a*, − *norm a*}›
proof (*cases* ‹*class.not-singleton TYPE*('*a*)›)
case *True*
show *?thesis*
using *chilbert-space-class.chilbert-space-axioms True assms*
by (*rule largest-eigenvalue-norm-aux*[*internalize-sort*' '*a*])
next
case *False*
then have ‹*a* = *0*›
by (*rule not-not-singleton-cblinfun-zero*)
then show *?thesis*
by *simp*
qed

hide-fact *largest-eigenvalue-norm-aux*

lemma *cmod-largest-eigenvalue*:
fixes *a* :: ‹'*a* ⇒$_{CL}$ '*a*::*chilbert-space*›
assumes ‹*compact-op a*›
assumes ‹*selfadjoint a*›
shows ‹*cmod* (*largest-eigenvalue a*) = *norm a*›
using *largest-eigenvalue-norm*[*OF assms*] **by** *auto*

lemma *compact-op-comp-right*: ‹*compact-op* (*a* *o*$_{CL}$ *b*)› **if** ‹*compact-op b*›
for *a b* :: ‹-::*chilbert-space* ⇒$_{CL}$ -::*chilbert-space*›
proof −
from *that* **have** ‹*b* ∈ *closure* (*Collect finite-rank*)›
using *compact-op-finite-rank* **by** *blast*
then have ‹*a* *o*$_{CL}$ *b* ∈ *cblinfun-compose a* ' *closure* (*Collect finite-rank*)›
by *blast*
also have ‹... ⊆ *closure* (*cblinfun-compose a* ' *Collect finite-rank*)›
by (*auto intro*!: *closure-bounded-linear-image-subset bounded-clinear.bounded-linear bounded-clinear-cblinfun-compo*
also have ‹... ⊆ *closure* (*Collect finite-rank*)›
by (*auto intro*!: *closure-mono finite-rank-comp-right*)
finally show ‹*compact-op* (*a* *o*$_{CL}$ *b*)›
using *compact-op-finite-rank* **by** *blast*
qed

end

11 *Spectral-Theorem* – The spectral theorem for compact operators

theory *Spectral-Theorem*
imports *Compact-Operators Positive-Operators Eigenvalues*
begin

11.1 Spectral decomp, compact op

fun *spectral-dec-val* :: ‹('a::*chilbert-space* ⇒$_{CL}$ 'a) ⇒ *nat* ⇒ *complex*›
— The eigenvalues in the spectral decomposition
and *spectral-dec-space* :: ‹('a ⇒$_{CL}$ 'a) ⇒ *nat* ⇒ 'a *ccsubspace*›
— The eigenspaces in the spectral decomposition
and *spectral-dec-op* :: ‹('a ⇒$_{CL}$ 'a) ⇒ *nat* ⇒ ('a ⇒$_{CL}$ 'a)›
— A sequence of operators mostly for the proof of spectral composition. But see also *spectral-dec-op-spectral-dec-proj* below.
where ‹*spectral-dec-val a n* = *largest-eigenvalue* (*spectral-dec-op a n*)›
| ‹*spectral-dec-space a n* = (*if spectral-dec-val a n* = *0 then 0 else eigenspace* (*spectral-dec-val a n*) (*spectral-dec-op a n*))›
| ‹*spectral-dec-op a* (*Suc n*) = *spectral-dec-op a n* o$_{CL}$ *Proj* (− *spectral-dec-space a n*)›
| ‹*spectral-dec-op a 0* = *a*›

definition *spectral-dec-proj* :: ‹('a::*chilbert-space* ⇒$_{CL}$ 'a) ⇒ *nat* ⇒ ('a ⇒$_{CL}$ 'a)› **where**
— Projectors in the spectral decomposition
‹*spectral-dec-proj a n* = *Proj* (*spectral-dec-space a n*)›

declare *spectral-dec-val.simps*[*simp del*]
declare *spectral-dec-space.simps*[*simp del*]

lemmas *spectral-dec-def* = *spectral-dec-val.simps*
lemmas *spectral-dec-space-def* = *spectral-dec-space.simps*

lemma *spectral-dec-op-selfadj*:
assumes ‹*selfadjoint a*›
shows ‹*selfadjoint* (*spectral-dec-op a n*)›
proof (*induction n*)
case *0*
with *assms* **show** *?case*
by *simp*
next
case (*Suc n*)
define *E T* **where** ‹*E* = *spectral-dec-space a n*› **and** ‹*T* = *spectral-dec-op a n*›
from *Suc* **have** ‹*normal-op T*›
by (*auto intro*!: *selfadjoint-imp-normal simp*: *T-def*)
then have ‹*reducing-subspace E T*›
by (*auto intro*!: *eigenspace-is-reducing simp*: *spectral-dec-space-def E-def T-def*)
then have ‹*reducing-subspace* (− *E*) *T*›
by *simp*
then have ∗: ‹*Proj* (− *E*) o$_{CL}$ *T* o$_{CL}$ *Proj* (− *E*) = *T* o$_{CL}$ *Proj* (− *E*)›
by (*simp add*: *invariant-subspace-iff-PAP reducing-subspace-def*)
show *?case*
using *Suc*
apply (*simp add*: *flip*: *T-def E-def* ∗)
by (*simp add*: *selfadjoint-def adj-Proj cblinfun-compose-assoc*)
qed

lemma *spectral-dec-op-compact*:
assumes ‹*compact-op a*›
shows ‹*compact-op* (*spectral-dec-op a n*)›
apply (*induction n*)
by (*auto intro*!: *compact-op-comp-left assms*)

lemma *spectral-dec-val-eigenvalue-of-spectral-dec-op*:
fixes *a* :: ‹'*a*::{*chilbert-space*, *not-singleton*} ⇒$_{CL}$ '*a*›
assumes ‹*compact-op a*›
assumes ‹*selfadjoint a*›
shows ‹*spectral-dec-val a n* ∈ *eigenvalues* (*spectral-dec-op a n*)›
by (*auto intro*!: *largest-eigenvalue-ex spectral-dec-op-compact spectral-dec-op-selfadj assms*
simp: *spectral-dec-def*)

lemma *spectral-dec-proj-finite-rank*:
assumes ‹*compact-op a*›
shows ‹*finite-rank* (*spectral-dec-proj a n*)›
apply (*cases* ‹*spectral-dec-val a n* = *0*›)
by (*auto intro*!: *finite-rank-Proj-finite-dim compact-op-eigenspace-finite-dim spectral-dec-op-compact assms*
simp: *spectral-dec-proj-def spectral-dec-space-def*)

lemma *norm-spectral-dec-op*:
assumes ‹*compact-op a*›
assumes ‹*selfadjoint a*›
shows ‹*norm* (*spectral-dec-op a n*) = *cmod* (*spectral-dec-val a n*)›
by (*simp add*: *spectral-dec-def cmod-largest-eigenvalue spectral-dec-op-compact spectral-dec-op-selfadj assms*)

lemma *spectral-dec-op-decreasing-eigenspaces*:
assumes ‹*n* ≥ *m*› **and** ‹*e* ≠ *0*›
assumes ‹*selfadjoint a*›
shows ‹*eigenspace e* (*spectral-dec-op a n*) ≤ *eigenspace e* (*spectral-dec-op a m*)›
proof −
have ∗: ‹*eigenspace e* (*spectral-dec-op a* (*Suc n*)) ≤ *eigenspace e* (*spectral-dec-op a n*)› **for** *n*
proof (*intro ccsubspace-leI subsetI*)
fix *h*
assume *asm*: ‹*h* ∈ *space-as-set* (*eigenspace e* (*spectral-dec-op a* (*Suc n*)))›
have ‹*orthogonal-spaces* (*eigenspace e* (*spectral-dec-op a* (*Suc n*))) (*kernel* (*spectral-dec-op a* (*Suc n*)))›
using *spectral-dec-op-selfadj*[*of a* ‹*Suc n*›] ‹*e* ≠ *0*› ‹*selfadjoint a*›
by (*auto intro*!: *eigenspaces-orthogonal selfadjoint-imp-normal spectral-dec-op-selfadj*
simp: *spectral-dec-space-def simp flip*: *eigenspace-0*)
then have ‹*eigenspace e* (*spectral-dec-op a* (*Suc n*)) ≤ − *kernel* (*spectral-dec-op a* (*Suc n*))›
using *orthogonal-spaces-leq-compl* **by** *blast*
also have ‹... ≤ − *spectral-dec-space a n*›
by (*auto intro*!: *ccsubspace-leI kernel-memberI simp*: *Proj-0-compl*)
finally have ‹*h* ∈ *space-as-set* (− *spectral-dec-space a n*)›

```
    using asm by (simp add: Set.basic-monos(7) less-eq-ccsubspace.rep-eq)
  then have ‹spectral-dec-op a n h = spectral-dec-op a (Suc n) h›
    by (simp add: Proj-fixes-image)
  also have ‹... = e *_C h›
    using asm eigenspace-memberD by blast
  finally show ‹h ∈ space-as-set (eigenspace e (spectral-dec-op a n))›
    by (simp add: eigenspace-memberI)
 qed
 define k where ‹k = n - m›
 from * have ‹eigenspace e (spectral-dec-op a (m + k)) ≤ eigenspace e (spectral-dec-op a m)›
  by (induction k) (auto simp del: spectral-dec-op.simps intro: order.trans)
 then show ?thesis
  using ‹n ≥ m› by (simp add: k-def)
qed

lemma spectral-dec-val-not-not-singleton:
 fixes a :: ‹'a::chilbert-space ⇒_CL 'a›
 assumes ‹¬ class.not-singleton TYPE('a)›
 shows ‹spectral-dec-val a n = 0›
proof -
 from assms have ‹spectral-dec-op a n = 0›
  by (rule not-not-singleton-cblinfun-zero)
 then have ‹largest-eigenvalue (spectral-dec-op a n) = 0›
  by simp
 then show ?thesis
  by (simp add: spectral-dec-def)
qed

lemma spectral-dec-val-eigenvalue-aux:
 — [2], Theorem II.5.1
 fixes a :: ‹'a::{chilbert-space, not-singleton} ⇒_CL 'a›
 assumes ‹compact-op a›
 assumes ‹selfadjoint a›
 assumes eigen-neq0: ‹spectral-dec-val a n ≠ 0›
 shows ‹spectral-dec-val a n ∈ eigenvalues a›
proof -
 define e where ‹e = spectral-dec-val a n›
 with assms have ‹e ≠ 0›
  by fastforce

 from spectral-dec-op-decreasing-eigenspaces[where m=0 and a=a and n=n, OF - ‹e ≠ 0› ‹selfadjoint a›]
 have 1: ‹eigenspace e (spectral-dec-op a n) ≤ eigenspace e a›
  by simp
 have 2: ‹spectral-dec-space a n ≠ ⊥›
 proof -
  have ‹spectral-dec-val a n ∈ eigenvalues (spectral-dec-op a n)›
    by (simp add: assms(1) assms(2) spectral-dec-val.simps spectral-dec-op-compact spectral-dec-op-selfadj largest-eigenvalue-ex)
```

```
    then show ?thesis
      using ‹e ≠ 0› by (simp add: eigenvalues-def spectral-dec-space.simps e-def)
  qed
  from 1 2 have ‹eigenspace e a ≠ ⊥›
    by (auto simp: spectral-dec-space-def bot-unique simp flip: e-def simp: ‹e ≠ 0›)
  then show ‹e ∈ eigenvalues a›
    by (simp add: eigenvalues-def)
qed

lemma spectral-dec-val-eigenvalue:
  — [2], Theorem II.5.1
  fixes a :: ‹('a::chilbert-space ⇒_CL 'a)›
  assumes ‹compact-op a›
  assumes ‹selfadjoint a›
  assumes eigen-neq0: ‹spectral-dec-val a n ≠ 0›
  shows ‹spectral-dec-val a n ∈ eigenvalues a›
proof (cases ‹class.not-singleton TYPE('a)›)
  case True
  show ?thesis
    using chilbert-space-axioms True assms
    by (rule spectral-dec-val-eigenvalue-aux[internalize-sort' 'a])
next
  case False
  then have ‹spectral-dec-val a n = 0›
    by (rule spectral-dec-val-not-not-singleton)
  with assms show ?thesis
    by simp
qed

hide-fact spectral-dec-val-eigenvalue-aux

lemma spectral-dec-val-decreasing:
  assumes ‹compact-op a›
  assumes ‹selfadjoint a›
  assumes ‹n ≥ m›
  shows ‹cmod (spectral-dec-val a n) ≤ cmod (spectral-dec-val a m)›
proof −
  have ‹norm (spectral-dec-op a (Suc n)) ≤ norm (spectral-dec-op a n)› for n
    apply simp
   by (smt (verit) Proj-partial-isometry cblinfun-compose-zero-right mult-cancel-left2 norm-cblinfun-compose
norm-le-zero-iff norm-partial-isometry)
  then have ∗: ‹cmod (spectral-dec-val a (Suc n)) ≤ cmod (spectral-dec-val a n)› for n
    by (simp add: cmod-largest-eigenvalue spectral-dec-op-compact assms spectral-dec-op-selfadj
spectral-dec-def
        del: spectral-dec-op.simps)
  define k where ‹k = n − m›
  have ‹cmod (spectral-dec-val a (m + k)) ≤ cmod (spectral-dec-val a m)›
    apply (induction k arbitrary: m)
     apply simp
```

```
    by (metis * add-Suc-right order-trans-rules(23))
  with ‹n ≥ m› show ?thesis
    by (simp add: k-def)
qed


lemma spectral-dec-val-distinct-aux:
  fixes a :: ‹('a::{chilbert-space, not-singleton} ⇒CL 'a)›
  assumes ‹n ≠ m›
  assumes ‹compact-op a›
  assumes ‹selfadjoint a›
  assumes neq0: ‹spectral-dec-val a n ≠ 0›
  shows ‹spectral-dec-val a n ≠ spectral-dec-val a m›
proof (rule ccontr)
  assume ‹¬ spectral-dec-val a n ≠ spectral-dec-val a m›
  then have eq: ‹spectral-dec-val a n = spectral-dec-val a m›
    by blast
  wlog nm: ‹n > m› goal False generalizing n m keeping eq neq0
    using hypothesis[of n m] negation assms eq neq0
    by auto
  define e where ‹e = spectral-dec-val a n›
  with neq0 have ‹e ≠ 0›
    by simp

  have ‹spectral-dec-space a n ≠ ⊥›
  proof −
    have ‹e ∈ eigenvalues (spectral-dec-op a n)›
      by (auto intro!: spectral-dec-val-eigenvalue-of-spectral-dec-op assms simp: e-def)
    then show ?thesis
      by (simp add: spectral-dec-space-def eigenvalues-def e-def neq0)
  qed
  then obtain h where ‹norm h = 1› and h-En: ‹h ∈ space-as-set (spectral-dec-space a n)›
    using ccsubspace-contains-unit by blast
  have T-Sucm-h: ‹spectral-dec-op a (Suc m) h = 0›
  proof −
    have ‹spectral-dec-space a n = eigenspace e (spectral-dec-op a n)›
      by (simp add: spectral-dec-space-def e-def neq0)
    also have ‹... ≤ eigenspace e (spectral-dec-op a m)›
      using ‹n > m› ‹e ≠ 0› assms
      by (auto intro!: spectral-dec-op-decreasing-eigenspaces simp: )
    also have ‹... = spectral-dec-space a m›
      using neq0 by (simp add: spectral-dec-space-def e-def eq)
    finally have ‹h ∈ space-as-set (spectral-dec-space a m)›
      using h-En
      by (simp add: basic-trans-rules(31) less-eq-ccsubspace.rep-eq)
    then show ‹spectral-dec-op a (Suc m) h = 0›
      by (simp add: Proj-0-compl)
  qed
  have ‹spectral-dec-op a (Suc m + k) h = 0› if ‹k ≤ n − m − 1› for k
```

```
  proof (insert that, induction k)
    case 0
    from T-Sucm-h show ?case
      by simp
  next
    case (Suc k)
    define mk1 where ‹mk1 = Suc (m + k)›
    from Suc.prems have ‹mk1 ≤ n›
      using mk1-def by linarith
    have ‹eigenspace e (spectral-dec-op a n) ≤ eigenspace e (spectral-dec-op a mk1)›
      using ‹mk1 ≤ n› ‹e ≠ 0› ‹selfadjoint a›
      by (rule spectral-dec-op-decreasing-eigenspaces)
    with h-En have h-mk1: ‹h ∈ space-as-set (eigenspace e (spectral-dec-op a mk1))›
      by (auto simp: e-def spectral-dec-space-def less-eq-ccsubspace.rep-eq neq0)
    have ‹Proj (− spectral-dec-space a mk1) *V h = 0 ∨ Proj (− spectral-dec-space a mk1) *V
h = h›
    proof (cases ‹e = spectral-dec-val a mk1›)
      case True
      from h-mk1 have ‹Proj (− spectral-dec-space a mk1) h = 0›
        using ‹e ≠ 0› by (simp add: Proj-0-compl True spectral-dec-space-def)
      then show ?thesis
        by simp
    next
      case False
      have ‹orthogonal-spaces (eigenspace e (spectral-dec-op a mk1)) (spectral-dec-space a mk1)›
      by (simp add: False assms eigenspaces-orthogonal spectral-dec-space.simps spectral-dec-op-selfadj
selfadjoint-imp-normal)
      with h-mk1 have ‹h ∈ space-as-set (− spectral-dec-space a mk1)›
        using less-eq-ccsubspace.rep-eq orthogonal-spaces-leq-compl by blast
      then have ‹Proj (− spectral-dec-space a mk1) h = h›
        by (rule Proj-fixes-image)
      then show ?thesis
        by simp
    qed
    with Suc show ?case
      by (auto simp: mk1-def)
  qed
  from this[where k=‹n − m − 1›]
  have ‹spectral-dec-op a n h = 0›
    using ‹n > m›
    by (simp del: spectral-dec-op.simps)
  moreover from h-En have ‹spectral-dec-op a n h = e *C h›
    by (simp add: neq0 e-def eigenspace-memberD spectral-dec-space-def)
  ultimately show False
    using ‹norm h = 1› ‹e ≠ 0›
    by force
qed

lemma spectral-dec-val-distinct:
```

fixes $a :: \langle 'a :: chilbert\text{-}space \Rightarrow_{CL} 'a \rangle$
assumes $\langle n \neq m \rangle$
assumes $\langle compact\text{-}op\ a \rangle$
assumes $\langle selfadjoint\ a \rangle$
assumes *neq0*: $\langle spectral\text{-}dec\text{-}val\ a\ n \neq 0 \rangle$
shows $\langle spectral\text{-}dec\text{-}val\ a\ n \neq spectral\text{-}dec\text{-}val\ a\ m \rangle$
proof (*cases* $\langle class.not\text{-}singleton\ TYPE('a) \rangle$)
case *True*
show *?thesis*
using *chilbert-space-axioms True assms*
by (*rule spectral-dec-val-distinct-aux*[*internalize-sort'* *'a*])
next
case *False*
then have $\langle spectral\text{-}dec\text{-}val\ a\ n = 0 \rangle$
by (*rule spectral-dec-val-not-not-singleton*)
with *assms* **show** *?thesis*
by *simp*
qed

hide-fact *spectral-dec-val-distinct-aux*

lemma *spectral-dec-val-tendsto-0*:

assumes $\langle compact\text{-}op\ a \rangle$
assumes $\langle selfadjoint\ a \rangle$
shows $\langle spectral\text{-}dec\text{-}val\ a \longrightarrow 0 \rangle$
proof (*cases* $\langle \exists n.\ spectral\text{-}dec\text{-}val\ a\ n = 0 \rangle$)
case *True*
then obtain *n* **where** $\langle spectral\text{-}dec\text{-}val\ a\ n = 0 \rangle$
by *auto*
then have $\langle spectral\text{-}dec\text{-}val\ a\ m = 0 \rangle$ **if** $\langle m \geq n \rangle$ **for** *m*
using *spectral-dec-val-decreasing*[*OF assms that*]
by *simp*
then show $\langle spectral\text{-}dec\text{-}val\ a \longrightarrow 0 \rangle$
by (*auto intro*!: *tendsto-eventually eventually-sequentiallyI*)
next
case *False*
define *E* **where** $\langle E = spectral\text{-}dec\text{-}val\ a \rangle$
from *False* **have** $\langle E\ n \in eigenvalues\ a \rangle$ **for** *n*
by (*simp add*: *spectral-dec-val-eigenvalue assms E-def*)
then have $\langle eigenspace\ (E\ n)\ a \neq 0 \rangle$ **for** *n*
by (*simp add*: *eigenvalues-def*)
then obtain *e* **where** *e-E*: $\langle e\ n \in space\text{-}as\text{-}set\ (eigenspace\ (E\ n)\ a) \rangle$
and *norm-e*: $\langle norm\ (e\ n) = 1 \rangle$ **for** *n*
apply *atomize-elim*
using *ccsubspace-contains-unit*
by (*auto intro*!: *choice2*)
then obtain *h n* **where** $\langle strict\text{-}mono\ n \rangle$ **and** *aen-lim*: $\langle (\lambda j.\ a\ (e\ (n\ j))) \longrightarrow h \rangle$
proof *atomize-elim*

```
  from ‹compact-op a›
  have compact:‹compact (closure (a ‘ cball 0 1))›
    by (simp add: compact-op-def2)
  from norm-e have ‹a (e n) ∈ closure (a ‘ cball 0 1)› for n
    using closure-subset[of ‹a ‘ cball 0 1›] by auto
  with compact[unfolded compact-def, rule-format, of ‹λn. a (e n)›]
  show ‹∃ n h. strict-mono n ∧ (λj. a (e (n j))) ⟶ h›
    by (auto simp: o-def)
qed
have ortho-en: ‹is-orthogonal (e (n j)) (e (n k))› if ‹j ≠ k› for j k
proof −
  have ‹n j ≠ n k›
    by (simp add: ‹strict-mono n› strict-mono-eq that)
  then have ‹E (n j) ≠ E (n k)›
    unfolding E-def
    apply (rule spectral-dec-val-distinct)
    using False assms by auto
  then have ‹orthogonal-spaces (eigenspace (E (n j)) a) (eigenspace (E (n k)) a)›
    apply (rule eigenspaces-orthogonal)
    by (simp add: assms(2) selfadjoint-imp-normal)
  with e-E show ?thesis
    using orthogonal-spaces-def by blast
qed
have aEe: ‹a (e n) = E n *C e n› for n
  by (simp add: e-E eigenspace-memberD)
obtain α where E-lim: ‹(λn. norm (E n)) ⟶ α›
  by (rule decseq-convergent[where X=‹λn. cmod (E n)› and B=0])
     (use spectral-dec-val-decreasing[OF assms] in ‹auto intro!: simp: decseq-def E-def›)
then have ‹α ≥ 0›
  apply (rule LIMSEQ-le-const)
  by simp
have aen-diff: ‹norm (a (e (n j)) − a (e (n k))) ≥ α * sqrt 2› if ‹j ≠ k› for j k
proof −
  from E-lim and spectral-dec-val-decreasing[OF assms, folded E-def]
  have E-geq-α: ‹cmod (E n) ≥ α› for n
    apply (rule-tac decseq-ge[unfolded decseq-def, rotated])
    by auto
  have ‹(norm (a (e (n j)) − a (e (n k))))² = (cmod (E (n j)))² + (cmod (E (n k)))²›
    by (simp add: polar-identity-minus aEe that ortho-en norm-e)
  also have ‹... ≥ α² + α²› (is ‹- ≥ ...›)
    apply (rule add-mono)
    using E-geq-α ‹α ≥ 0› by auto
  also have ‹... = (α * sqrt 2)²›
    by (simp add: algebra-simps)
  finally show ?thesis
    apply (rule power2-le-imp-le)
    by simp
qed
have ‹α = 0›
```

```
  proof −
    have ‹α ∗ sqrt 2 < ε› if ‹ε > 0› for ε
    proof −
      from ‹strict-mono n› have cauchy: ‹Cauchy (λk. a (e (n k)))›
        using LIMSEQ-imp-Cauchy aen-lim by blast
      obtain k where k: ‹∀ m≥k. ∀ na≥k. dist (a *V e (n m)) (a *V e (n na)) < ε›
        apply atomize-elim
        using cauchy[unfolded Cauchy-def, rule-format, OF ‹ε > 0›]
        by simp
      define j where ‹j = Suc k›
      then have ‹j ≠ k›
        by simp
      from k have ‹dist (a (e (n j))) (a (e (n k))) < ε›
        by (simp add: j-def)
      with aen-diff[OF ‹j ≠ k›] show ‹α ∗ sqrt 2 < ε›
        by (simp add: Cauchy-def dist-norm)
    qed
    with ‹α ≥ 0› show ‹α = 0›
      by (smt (verit) linordered-semiring-strict-class.mult-pos-pos real-sqrt-le-0-iff)
  qed
  with E-lim show ?thesis
    by (auto intro!: tendsto-norm-zero-cancel simp: E-def)
qed

lemma spectral-dec-op-tendsto:
  assumes ‹compact-op a›
  assumes ‹selfadjoint a›
  shows ‹spectral-dec-op a ⟶ 0›
  apply (rule tendsto-norm-zero-cancel)
  using spectral-dec-val-tendsto-0[OF assms]
  apply (simp add: norm-spectral-dec-op assms)
  using tendsto-norm-zero by blast

lemma spectral-dec-op-spectral-dec-proj:
  ‹spectral-dec-op a n = a − (∑ i<n. spectral-dec-val a i *C spectral-dec-proj a i)›
proof (induction n)
  case 0
  show ?case
    by simp
next
  case (Suc n)
  have ‹spectral-dec-op a (Suc n) = spectral-dec-op a n o_CL Proj (− spectral-dec-space a n)›
    by simp
  also have ‹... = spectral-dec-op a n − spectral-dec-val a n *C spectral-dec-proj a n› (is ‹?lhs
= ?rhs›)
  proof −
    have ‹?lhs h = ?rhs h› if ‹h ∈ space-as-set (spectral-dec-space a n)› for h
    proof −
      have ‹?lhs h = 0›
```

by (*simp add*: *Proj-0-compl that*)
have ‹*spectral-dec-op a n* $*_V$ *h* = *spectral-dec-val a n* $*_C$ *h*›
by (*smt* (*verit*, *best*) *Proj-fixes-image* ‹(*spectral-dec-op a n* o_{CL} *Proj* (− *spectral-dec-space a n*)) $*_V$ *h* = *0*› *cblinfun-apply-cblinfun-compose complex-vector.scale-eq-0-iff eigenspace-memberD spectral-dec-space.elims kernel-Proj kernel-cblinfun-compose kernel-memberD kernel-memberI ortho-involution that*)
also have ‹... = *spectral-dec-val a n* $*_C$ (*spectral-dec-proj a n* $*_V$ *h*)›
by (*simp add*: *Proj-fixes-image spectral-dec-proj-def that*)
finally
have ‹*?rhs h* = *0*›
by (*simp add*: *cblinfun.diff-left*)
with ‹*?lhs h* = *0*› **show** *?thesis*
by *simp*
qed
moreover have ‹*?lhs h* = *?rhs h*› **if** ‹*h* ∈ *space-as-set* (− *spectral-dec-space a n*)› **for** *h*
by (*simp add*: *Proj-0-compl Proj-fixes-image cblinfun.diff-left spectral-dec-proj-def that*)
ultimately have ‹*?lhs h* = *?rhs h*›
if ‹*h* ∈ *space-as-set* (*spectral-dec-space a n* ⊔ − *spectral-dec-space a n*) › **for** *h*
using *that* **by** (*rule eq-on-ccsubspaces-sup*)
then show ‹*?lhs* = *?rhs*›
by (*auto intro*!: *cblinfun-eqI simp add*:)
qed
also have ‹... = *a* − (∑ *i*<*Suc n*. *spectral-dec-val a i* $*_C$ *spectral-dec-proj a i*)›
by (*simp add*: *Suc.IH*)
finally show *?case*
by −
qed

lemma *sequential-tendsto-reorder*:
assumes ‹*inj g*›
assumes ‹*f* ⟶ *l*›
shows ‹(*f o g*) ⟶ *l*›
proof (*intro lim-explicit*[*THEN iffD2*] *impI allI*)
fix *S* **assume** ‹*open S*› **and** ‹*l* ∈ *S*›
with ‹*f* ⟶ *l*›
obtain *M* **where** *M*: ‹*f m* ∈ *S*› **if** ‹*m* ≥ *M*› **for** *m*
using *tendsto-obtains-N* **by** *blast*
define *N* **where** ‹*N* = *Max* (*g* −‘ {..<*M*}) + *1*›
have *N*: ‹*g n* ≥ *M*› **if** ‹*n* ≥ *N*› **for** *n*
proof −
from ‹*inj g*› **have** ‹*finite* (*g* −‘ {..<*M*})›
using *finite-vimageI* **by** *blast*
then have ‹*N* > *n*› **if** ‹*n* ∈ *g* −‘ {..<*M*}› **for** *n*
using *N-def that*
by (*simp add*: *less-Suc-eq-le*)
then have ‹*N* > *n*› **if** ‹*g n* < *M*› **for** *n*
by (*simp add*: *that*)
with *that* **show** ‹*g n* ≥ *M*›

```
    using linorder-not-less by blast
  qed
  from N M show ‹∃ N. ∀ n≥N. (f ∘ g) n ∈ S›
    by auto
qed
```

```
lemma spectral-dec-sums:
  assumes ‹compact-op a›
  assumes ‹selfadjoint a›
  shows ‹(λn. spectral-dec-val a n *C spectral-dec-proj a n) sums a›
proof −
  from spectral-dec-op-tendsto[OF assms]
  have ‹(λn. a − spectral-dec-op a n) ⟶ a›
    by (simp add: tendsto-diff-const-left-rewrite)
  moreover from spectral-dec-op-spectral-dec-proj[of a]
  have ‹a − spectral-dec-op a n = (∑ i<n. spectral-dec-val a i *C spectral-dec-proj a i)› for n
    by simp
  ultimately show ?thesis
    by (simp add: sums-def)
qed

lemma spectral-dec-val-real:
  assumes ‹compact-op a›
  assumes ‹selfadjoint a›
  shows ‹spectral-dec-val a n ∈ ℝ›
  by (metis Reals-0 assms(1) assms(2) eigenvalue-selfadj-real spectral-dec-val-eigenvalue)


lemma spectral-dec-space-orthogonal:
  assumes ‹compact-op a›
  assumes ‹selfadjoint a›
  assumes ‹n ≠ m›
  shows ‹orthogonal-spaces (spectral-dec-space a n) (spectral-dec-space a m)›
proof (cases ‹spectral-dec-val a n = 0 ∨ spectral-dec-val a m = 0›)
  case True
  then show ?thesis
    by (auto intro!: simp: spectral-dec-space-def)
next
  case False
  have ‹spectral-dec-space a n ≤ eigenspace (spectral-dec-val a n) a›
    using ‹selfadjoint a›
   by (metis False spectral-dec-space.elims spectral-dec-op.simps(2) spectral-dec-op-decreasing-eigenspaces
zero-le)
  moreover have ‹spectral-dec-space a m ≤ eigenspace (spectral-dec-val a m) a›
    using ‹selfadjoint a›
```

by (*metis False spectral-dec-space.elims spectral-dec-op.simps*(*2*) *spectral-dec-op-decreasing-eigenspaces zero-le*)
moreover have ‹*orthogonal-spaces* (*eigenspace* (*spectral-dec-val a n*) *a*) (*eigenspace* (*spectral-dec-val a m*) *a*)›
apply (*intro eigenspaces-orthogonal selfadjoint-imp-normal assms spectral-dec-val-distinct*)
using *False* **by** *simp*
ultimately show *?thesis*
by (*meson order.trans orthocomplemented-lattice-class.compl-mono orthogonal-spaces-leq-compl*)

qed

lemma *spectral-dec-proj-pos*: ‹*spectral-dec-proj a n* ≥ *0*›
by (*auto intro*!: *simp*: *spectral-dec-proj-def*)

lemma
assumes ‹*compact-op a*›
assumes ‹*selfadjoint a*›
shows *spectral-dec-tendsto-pos-op*: ‹(λ*n*. *max 0* (*spectral-dec-val a n*) ∗$_C$ *spectral-dec-proj a n*) *sums pos-op a*› (**is** *?thesis1*)
and *spectral-dec-tendsto-neg-op*: ‹(λ*n*. − *min* (*spectral-dec-val a n*) *0* ∗$_C$ *spectral-dec-proj a n*) *sums neg-op a*› (**is** *?thesis2*)
proof −
define *I J* **where** ‹*I* = {*n*. *spectral-dec-val a n* ≥ *0*}›
and ‹*J* = {*n*. *spectral-dec-val a n* ≤ *0*}›
define *R S* **where** ‹*R* = (⨆ *n*∈*I*. *spectral-dec-space a n*)›
and ‹*S* = (⨆ *n*∈*J*. *spectral-dec-space a n*)›
define *aR aS* **where** ‹*aR* = *a* o$_{CL}$ *Proj R*› **and** ‹*aS* = − *a* o$_{CL}$ *Proj S*›
have *spectral-dec-cases*: ‹(*0* < *spectral-dec-val a n* ⟹ *P*) ⟹
(*spectral-dec-val a n* < *0* ⟹ *P*) ⟹
(*spectral-dec-val a n* = *0* ⟹ *P*) ⟹ *P*› **for** *n P*
apply *atomize-elim*
using *reals-zero-comparable*[*OF spectral-dec-val-real*[*OF assms*, *of n*]]
by *auto*
have *PRP*: ‹*spectral-dec-proj a n* o$_{CL}$ *Proj R* = *spectral-dec-proj a n*› **if** ‹*n* ∈ *I*› **for** *n*
by (*auto intro*!: *Proj-o-Proj-subspace-left simp add*: *R-def SUP-upper that spectral-dec-proj-def*)
have *PR0*: ‹*spectral-dec-proj a n* o$_{CL}$ *Proj R* = *0*› **if** ‹*n* ∉ *I*› **for** *n*
apply (*cases rule*: *spectral-dec-cases*[*of n*])
using *that*
by (*auto intro*!: *orthogonal-spaces-SUP-right spectral-dec-space-orthogonal assms simp*: *spectral-dec-proj-def R-def I-def simp flip*: *orthogonal-projectors-orthogonal-spaces*)
have *PSP*: ‹*spectral-dec-proj a n* o$_{CL}$ *Proj S* = *spectral-dec-proj a n*› **if** ‹*n* ∈ *J*› **for** *n*
by (*auto intro*!: *Proj-o-Proj-subspace-left simp add*: *S-def SUP-upper that spectral-dec-proj-def*)
have *PS0*: ‹*spectral-dec-proj a n* o$_{CL}$ *Proj S* = *0*› **if** ‹*n* ∉ *J*› **for** *n*
apply (*cases rule*: *spectral-dec-cases*[*of n*])
using *that*

by (*auto intro*!: *orthogonal-spaces-SUP-right spectral-dec-space-orthogonal assms*
simp: *spectral-dec-proj-def S-def J-def*
simp flip: *orthogonal-projectors-orthogonal-spaces*)
from *spectral-dec-sums*[*OF assms*]
have ‹(λ*n*. (*spectral-dec-val a n* ∗$_C$ *spectral-dec-proj a n*) o_{CL} *Proj R*) *sums aR*›
unfolding *aR-def*
apply (*rule bounded-linear.sums*[*rotated*])
by (*intro bounded-clinear.bounded-linear bounded-clinear-cblinfun-compose-left*)
then have *sum-aR*: ‹(λ*n*. *max 0* (*spectral-dec-val a n*) ∗$_C$ *spectral-dec-proj a n*) *sums aR*›
apply (*rule sums-cong*[*THEN iffD1*, *rotated*])
by (*simp add*: *I-def PR0 PRP max-def*)
from *sum-aR* **have** ‹*aR* ≥ *0*›
apply (*rule sums-pos-cblinfun*)
by (*auto intro*!: *spectral-dec-proj-pos scaleC-nonneg-nonneg simp*: *max-def*)
from *spectral-dec-sums*[*OF assms*]
have ‹(λ*n*. *spectral-dec-val a n* ∗$_C$ *spectral-dec-proj a n* o_{CL} *Proj S*) *sums* − *aS*›
unfolding *aS-def minus-minus cblinfun-compose-uminus-left*
apply (*rule bounded-linear.sums*[*rotated*])
by (*intro bounded-clinear.bounded-linear bounded-clinear-cblinfun-compose-left*)
then have *sum-aS'*: ‹(λ*n*. *min* (*spectral-dec-val a n*) *0* ∗$_C$ *spectral-dec-proj a n*) *sums* − *aS*›
apply (*rule sums-cong*[*THEN iffD1*, *rotated*])
by (*simp add*: *J-def PS0 PSP min-def*)
then have *sum-aS*: ‹(λ*n*. − *min* (*spectral-dec-val a n*) *0* ∗$_C$ *spectral-dec-proj a n*) *sums aS*›
using *sums-minus* **by** *fastforce*
from *sum-aS* **have** ‹*aS* ≥ *0*›
by (*rule sums-pos-cblinfun*)
(*auto intro*!: *spectral-dec-proj-pos scaleC-nonpos-nonneg simp*: *max-def min-def*)
from *sum-aR sum-aS'*
have ‹(λ*n*. *max 0* (*spectral-dec-val a n*) ∗$_C$ *spectral-dec-proj a n*
+ *min* (*spectral-dec-val a n*) *0* ∗$_C$ *spectral-dec-proj a n*) *sums* (*aR* − *aS*)›
using *sums-add* **by** *fastforce*
then have ‹(λ*n*. *spectral-dec-val a n* ∗$_C$ *spectral-dec-proj a n*) *sums* (*aR* − *aS*)›
proof (*rule sums-cong*[*THEN iffD1*, *rotated*])
fix *n*
have ‹*max 0* (*spectral-dec-val a n*) + *min* (*spectral-dec-val a n*) *0*
= *spectral-dec-val a n*›
apply (*cases rule*: *spectral-dec-cases*[*of n*])
by (*auto intro*!: *simp*: *max-def min-def*)
then
show ‹*max 0* (*spectral-dec-val a n*) ∗$_C$ *spectral-dec-proj a n* +
min (*spectral-dec-val a n*) *0* ∗$_C$ *spectral-dec-proj a n* =
spectral-dec-val a n ∗$_C$ *spectral-dec-proj a n*›
by (*metis scaleC-left.add*)
qed
with *spectral-dec-sums*[*OF assms*]
have ‹*aR* − *aS* = *a*›
using *sums-unique2* **by** *blast*
have ‹*aR* o_{CL} *aS* = *0*›
by (*metis* (*no-types*, *opaque-lifting*) *Proj-idempotent* ‹*0* ≤ *aR*› ‹*aR* − *aS* = *a*› *aR-def*

add-cancel-left-left add-minus-cancel adj-0 adj-Proj adj-cblinfun-compose assms(2) cblinfun-compose-minus-right comparable-hermitean lift-cblinfun-comp(2) selfadjoint-def uminus-add-conv-diff)
have ‹*aR* = *pos-op a*› **and** ‹*aS* = *neg-op a*›
by (*intro pos-op-neg-op-unique*[**where** *b*=*aR* **and** *c*=*aS*]
‹*aR* − *aS* = *a*› ‹0 ≤ *aR*› ‹0 ≤ *aS*› ‹*aR* o_{CL} *aS* = 0›)+
with *sum-aR* **and** *sum-aS*
show *?thesis1* **and** *?thesis2*
by *auto*
qed

lemma *spectral-dec-tendsto-abs-op*:
assumes ‹*compact-op a*›
assumes ‹*selfadjoint a*›
shows ‹(λ*n*. *cmod* (*spectral-dec-val a n*) $*_R$ *spectral-dec-proj a n*) *sums* *abs-op a*›
proof −
from *spectral-dec-tendsto-pos-op*[*OF assms*] *spectral-dec-tendsto-neg-op*[*OF assms*]
have ‹(λ*n*. *max 0* (*spectral-dec-val a n*) $*_C$ *spectral-dec-proj a n*
+ − *min* (*spectral-dec-val a n*) *0* $*_C$ *spectral-dec-proj a n*) *sums* (*pos-op a* + *neg-op a*)›
using *sums-add* **by** *blast*
then have ‹(λ*n*. *cmod* (*spectral-dec-val a n*) $*_R$ *spectral-dec-proj a n*) *sums* (*pos-op a* + *neg-op a*)›
apply (*rule sums-cong*[*THEN iffD1*, *rotated*])
using *spectral-dec-val-real*[*OF assms*]
apply (*simp add*: *complex-is-Real-iff cmod-def max-def min-def less-eq-complex-def scaleR-scaleC flip*: *scaleC-add-right*)
by (*metis complex-surj zero-complex.code*)
then show *?thesis*
by (*simp add*: *pos-op-plus-neg-op*)
qed

definition *spectral-dec-vecs* :: ‹('*a* ⇒$_{CL}$ '*a*) ⇒ '*a*::*chilbert-space set*› **where**
‹*spectral-dec-vecs a* = (⋃ *n*. *scaleC* (*csqrt* (*spectral-dec-val a n*)) ‘ *some-onb-of* (*spectral-dec-space a n*))›

lemma *spectral-dec-vecs-ortho*:
assumes ‹*selfadjoint a*› **and** ‹*compact-op a*›
shows ‹*is-ortho-set* (*spectral-dec-vecs a*)›
proof (*unfold is-ortho-set-def*, *intro conjI ballI impI*)
show ‹0 ∉ *spectral-dec-vecs a*›
proof (*rule notI*)
assume ‹0 ∈ *spectral-dec-vecs a*›
then obtain *n v* **where** *v0*: ‹0 = *csqrt* (*spectral-dec-val a n*) $*_C$ *v*› **and** *v-in*: ‹*v* ∈ *some-onb-of* (*spectral-dec-space a n*)›
by (*auto simp*: *spectral-dec-vecs-def*)
from *v-in* **have** ‹*v* ≠ 0›
using *some-onb-of-norm1* **by** *fastforce*
from *v-in* **have** ‹*spectral-dec-space a n* ≠ 0›
using *some-onb-of-0* **by** *force*
then have ‹*spectral-dec-val a n* ≠ 0›

```
    by (meson spectral-dec-space.elims)
  with v0 ‹v ≠ 0› show False
    by force
 qed
 fix g h assume g: ‹g ∈ spectral-dec-vecs a› and h: ‹h ∈ spectral-dec-vecs a› and ‹g ≠ h›
 from g obtain ng g' where gg': ‹g = csqrt (spectral-dec-val a ng) *_C g'› and g'-in: ‹g' ∈
some-onb-of (spectral-dec-space a ng)›
  by (auto simp: spectral-dec-vecs-def)
 from h obtain nh h' where hh': ‹h = csqrt (spectral-dec-val a nh) *_C h'› and h'-in: ‹h' ∈
some-onb-of (spectral-dec-space a nh)›
  by (auto simp: spectral-dec-vecs-def)
 have ‹is-orthogonal g' h'›
 proof (cases ‹ng = nh›)
  case True
  with h'-in have ‹h' ∈ some-onb-of (spectral-dec-space a nh)›
    by simp
  with g'-in True ‹g ≠ h› gg' hh'
  show ?thesis
    using is-ortho-set-def by fastforce
 next
  case False
  then have ‹orthogonal-spaces (spectral-dec-space a ng) (spectral-dec-space a nh)›
    by (auto intro!: spectral-dec-space-orthogonal assms simp: )
  with h'-in g'-in show ‹is-orthogonal g' h'›
    using orthogonal-spaces-ccspan by force
 qed
 then show ‹is-orthogonal g h›
  by (simp add: gg' hh')
qed

lemma spectral-dec-val-nonneg:
 assumes ‹a ≥ 0›
 assumes ‹compact-op a›
 shows ‹spectral-dec-val a n ≥ 0›
proof −
 define v where ‹v = spectral-dec-val a n›
 wlog non0: ‹spectral-dec-val a n ≠ 0› generalizing v keeping v-def
  using negation by force
 have [simp]: ‹selfadjoint a›
  using adj-0 assms(1) comparable-hermitean selfadjoint-def by blast
 have ‹v ∈ eigenvalues a›
  by (auto intro!: non0 spectral-dec-val-eigenvalue assms simp: v-def)
 then show ‹spectral-dec-val a n ≥ 0›
  using assms(1) eigenvalues-nonneg v-def by blast
qed

lemma spectral-dec-space-finite-dim[intro]:
 assumes ‹compact-op a›
 shows ‹finite-dim-ccsubspace (spectral-dec-space a n)›
```

by (*auto intro*!: *compact-op-eigenspace-finite-dim spectral-dec-op-compact assms simp*: *spectral-dec-space-def*)

lemma *spectral-dec-space-0*:
assumes ‹*spectral-dec-val a n* = *0*›
shows ‹*spectral-dec-space a n* = *0*›
by (*simp add*: *assms spectral-dec-space-def*)

end

12 *Trace-Class* − Trace-class operators

theory *Trace-Class*
imports *Complex-Bounded-Operators.Complex-L2 HS2Ell2*
Weak-Operator-Topology Positive-Operators Compact-Operators
Spectral-Theorem
begin

hide-fact (**open**) *Infinite-Set-Sum.abs-summable-on-Sigma-iff*
hide-fact (**open**) *Infinite-Set-Sum.abs-summable-on-comparison-test*
hide-const (**open**) *Determinants.trace*
hide-fact (**open**) *Determinants.trace-def*

unbundle *cblinfun-notation*

12.1 Auxiliary lemmas

lemma
fixes *h* :: ‹'*a*::{*chilbert-space*}›
assumes ‹*is-onb E*›
shows *parseval-abs-summable*: ‹(λ*e*. (*cmod* (*e* •$_C$ *h*))^2) *abs-summable-on E*›
proof (*cases* ‹*h* = *0*›)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have ‹(∑$_\infty$ *e*∈*E*. (*cmod* (*e* •$_C$ *h*))^2) ≠ *0*›
using *assms* **by** (*simp add*: *parseval-identity is-onb-def*)
then show *?thesis*
using *infsum-not-exists* **by** *auto*
qed

lemma *basis-image-square-has-sum1*:
— Half of [1, Proposition 18.1], other half in *basis-image-square-has-sum1*.
fixes *E* :: ‹'*a*::*complex-inner set*› **and** *F* :: ‹'*b*::*chilbert-space set*›
assumes ‹*is-onb E*› **and** ‹*is-onb F*›

shows ‹((λe. (*norm* (*A* ∗$_V$ *e*))^2) *has-sum t*) *E* ⟷ ((λ(*e*,*f*). (*cmod* (*f* ·$_C$ (*A* ∗$_V$ *e*)))^2) *has-sum t*) (*E*×*F*)›
proof (*rule iffI*)
assume *asm*: ‹((λe. (*norm* (*A* ∗$_V$ *e*))^2) *has-sum t*) *E›*
have *sum1*: ‹*t* = (∑$_\infty$*e*∈*E*. (*norm* (*A* ∗$_V$ *e*))^2)›
using *asm infsumI* **by** *blast*
have *abs1*: ‹(λe. (*norm* (*A* ∗$_V$ *e*))^2) *abs-summable-on E*›
using *asm summable-on-def* **by** *auto*
have *sum2*: ‹*t* = (∑$_\infty$*e*∈*E*. ∑$_\infty$*f*∈*F*. (*cmod* (*f* ·$_C$ (*A* ∗$_V$ *e*)))^2)›
apply (*subst sum1*)
apply (*rule infsum-cong*)
using *assms*(*2*)
by (*simp add*: *is-onb-def flip*: *parseval-identity*)
have *abs2*: ‹(λe. ∑$_\infty$*f*∈*F*. (*cmod* (*f* ·$_C$ (*A* ∗$_V$ *e*)))^2) *abs-summable-on E*›
using - *abs1* **apply** (*rule summable-on-cong*[*THEN iffD2*])
apply (*subst parseval-identity*)
using *assms*(*2*) **by** (*auto simp*: *is-onb-def*)
have *abs3*: ‹(λ(*x*, *y*). (*cmod* (*y* ·$_C$ (*A* ∗$_V$ *x*)))^2) *abs-summable-on E* × *F*›
thm *abs-summable-on-Sigma-iff*
apply (*rule abs-summable-on-Sigma-iff*[*THEN iffD2*], *rule conjI*)
using *abs2* **apply** (*auto simp del*: *real-norm-def*)
using *assms*(*2*) *parseval-abs-summable* **apply** *blast*
by *auto*
have *sum3*: ‹*t* = (∑$_\infty$(*e*,*f*)∈*E*×*F*. (*cmod* (*f* ·$_C$ (*A* ∗$_V$ *e*)))^2)›
apply (*subst sum2*)
apply (*subst infsum-Sigma′-banach*[*symmetric*])
using *abs3 abs-summable-summable* **apply** *blast*
by *auto*
then show ‹((λ(*e*,*f*). (*cmod* (*f* ·$_C$ (*A* ∗$_V$ *e*)))^2) *has-sum t*) (*E*×*F*)›
using *abs3 abs-summable-summable has-sum-infsum* **by** *blast*
next
assume *asm*: ‹((λ(*e*,*f*). (*cmod* (*f* ·$_C$ (*A* ∗$_V$ *e*)))^2) *has-sum t*) (*E*×*F*)›
have *abs3*: ‹(λ(*x*, *y*). (*cmod* (*y* ·$_C$ (*A* ∗$_V$ *x*)))^2) *abs-summable-on E* × *F*›
using *asm summable-on-def summable-on-iff-abs-summable-on-real*
by *blast*
have *sum3*: ‹*t* = (∑$_\infty$(*e*,*f*)∈*E*×*F*. (*cmod* (*f* ·$_C$ (*A* ∗$_V$ *e*)))^2)›
using *asm infsumI* **by** *blast*
have *sum2*: ‹*t* = (∑$_\infty$*e*∈*E*. ∑$_\infty$*f*∈*F*. (*cmod* (*f* ·$_C$ (*A* ∗$_V$ *e*)))^2)›
by (*metis* (*mono-tags*, *lifting*) *asm infsum-Sigma′-banach infsum-cong sum3 summable-iff-has-sum-infsum*)
have *abs2*: ‹(λe. ∑$_\infty$*f*∈*F*. (*cmod* (*f* ·$_C$ (*A* ∗$_V$ *e*)))^2) *abs-summable-on E*›
by (*smt* (*verit*, *del-insts*) *abs3 summable-on-Sigma-banach summable-on-cong summable-on-iff-abs-summable-on-re*
have *sum1*: ‹*t* = (∑$_\infty$*e*∈*E*. (*norm* (*A* ∗$_V$ *e*))^2)›
apply (*subst sum2*)
apply (*rule infsum-cong*)
using *assms*
by (*auto intro*!: *simp*: *parseval-identity is-onb-def*)
have *abs1*: ‹(λe. (*norm* (*A* ∗$_V$ *e*))^2) *abs-summable-on E*›
using *assms abs2*
by (*auto intro*!: *simp*: *parseval-identity is-onb-def*)

show ‹((λe. (*norm* (*A* $*_V$ *e*))^2) *has-sum t*) *E*›
using *abs1 sum1* **by** *auto*
qed

lemma *basis-image-square-has-sum2*:
— Half of [1, Proposition 18.1], other half in *basis-image-square-has-sum1*.
fixes *E* :: ‹'*a*::*chilbert-space set*› **and** *F* :: ‹'*b*::*chilbert-space set*›
assumes ‹*is-onb E*› **and** ‹*is-onb F*›
shows ‹((λe. (*norm* (*A* $*_V$ *e*))^2) *has-sum t*) *E* ⟷ ((λf. (*norm* (*A*∗ $*_V$ *f*))^2) *has-sum t*) *F*›
proof −
have ‹((λe. (*norm* (*A* $*_V$ *e*))^2) *has-sum t*) *E* ⟷ ((λ(e,f). (*cmod* (*f* $\cdot_C$ (*A* $*_V$ *e*)))^2) *has-sum t*) (*E*×*F*)›
using *basis-image-square-has-sum1 assms* **by** *blast*
also have ‹... ⟷ ((λ(e,f). (*cmod* ((*A*∗ $*_V$ *f*) $\cdot_C$ *e*))^2) *has-sum t*) (*E*×*F*)›
apply (*subst cinner-adj-left*)
by (*rule refl*)
also have ‹... ⟷ ((λ(f,e). (*cmod* ((*A*∗ $*_V$ *f*) $\cdot_C$ *e*))^2) *has-sum t*) (*F*×*E*)›
apply (*subst asm-rl*[*of* ‹*F*×*E* = *prod.swap* ‘ (*E*×*F*)›])
apply *force*
apply (*subst has-sum-reindex*)
by (*auto simp*: *o-def*)
also have ‹... ⟷ ((λf. (*norm* (*A*∗ $*_V$ *f*))^2) *has-sum t*) *F*›
apply (*subst cinner-commute*, *subst complex-mod-cnj*)
using *basis-image-square-has-sum1 assms*
by *blast*
finally show *?thesis*
by −
qed

12.2 Trace-norm and trace-class

lemma *trace-norm-basis-invariance*:
assumes ‹*is-onb E*› **and** ‹*is-onb F*›
shows ‹((λe. *cmod* (*e* $\cdot_C$ (*abs-op A* $*_V$ *e*))) *has-sum t*) *E* ⟷ ((λf. *cmod* (*f* $\cdot_C$ (*abs-op A* $*_V$ *f*))) *has-sum t*) *F*›
— [1], Corollary 18.2
proof −
define *B* **where** ‹*B* = *sqrt-op* (*abs-op A*)›
have ‹*complex-of-real* (*cmod* (*e* $\cdot_C$ (*abs-op A* $*_V$ *e*))) = (*B*∗ $*_V$ *B*$*_V$ *e*) $\cdot_C$ *e*› **for** *e*
apply (*simp add*: *B-def positive-hermitianI flip*: *cblinfun-apply-cblinfun-compose*)
by (*metis abs-op-pos abs-pos cinner-commute cinner-pos-if-pos complex-cnj-complex-of-real complex-of-real-cmod*)
also have ‹... *e* = *complex-of-real* ((*norm* (*B* $*_V$ *e*))^2)› **for** *e*
apply (*subst cdot-square-norm*[*symmetric*])
apply (*subst cinner-adj-left*[*symmetric*])
by (*simp add*: *B-def*)
finally have ∗: ‹*cmod* (*e* $\cdot_C$ (*abs-op A* $*_V$ *e*)) = (*norm* (*B* $*_V$ *e*))^2› **for** *e*
by (*metis Re-complex-of-real*)

have ‹((λe. *cmod* (e •$_C$ (*abs-op* A *$*_V$ e))) *has-sum* t) E ⟷ ((λe. (*norm* (B *$*_V$ e))²) *has-sum* t) E›
by (*simp add*: *)
also have ‹... = ((λf. (*norm* (B* *$*_V$ f))²) *has-sum* t) F›
apply (*subst basis-image-square-has-sum2*[**where** F=F])
by (*simp-all add*: *assms*)
also have ‹... = ((λf. (*norm* (B *$*_V$ f))²) *has-sum* t) F›
using *basis-image-square-has-sum2 assms*(2) **by** *blast*
also have ‹... = ((λe. *cmod* (e •$_C$ (*abs-op* A *$*_V$ e))) *has-sum* t) F›
by (*simp add*: *)
finally show *?thesis*
by *simp*
qed

definition *trace-class* :: ‹('a::*chilbert-space* ⇒$_{CL}$ 'b::*complex-inner*) ⇒ *bool*›
where ‹*trace-class* A ⟷ (∃ E. *is-onb* E ∧ (λe. e •$_C$ (*abs-op* A *$*_V$ e)) *abs-summable-on* E)›

lemma *trace-classI*:
assumes ‹*is-onb* E› **and** ‹(λe. e •$_C$ (*abs-op* A *$*_V$ e)) *abs-summable-on* E›
shows ‹*trace-class* A›
using *assms*(1) *assms*(2) *trace-class-def* **by** *blast*

lemma *trace-class-iff-summable*:
assumes ‹*is-onb* E›
shows ‹*trace-class* A ⟷ (λe. e •$_C$ (*abs-op* A *$*_V$ e)) *abs-summable-on* E›
apply (*auto intro*!: *trace-classI assms simp*: *trace-class-def*)
using *assms summable-on-def trace-norm-basis-invariance* **by** *blast*

lemma *trace-class-0*[*simp*]: ‹*trace-class* 0›
unfolding *trace-class-def*
by (*auto intro*!: *exI*[*of* - *some-chilbert-basis*] *simp*: *is-onb-def is-normal-some-chilbert-basis*)

lemma *trace-class-uminus*: ‹*trace-class* t ⟹ *trace-class* (−t)›
by (*auto simp add*: *trace-class-def*)

lemma *trace-class-uminus-iff*[*simp*]: ‹*trace-class* (−a) = *trace-class* a›
by (*auto simp add*: *trace-class-def*)

definition *trace-norm* **where** ‹*trace-norm* A = (*if trace-class* A *then* ($\sum_\infty$e∈*some-chilbert-basis*. *cmod* (e •$_C$ (*abs-op* A *$*_V$ e))) *else* 0)›

definition *trace* **where** ‹*trace* A = (*if trace-class* A *then* ($\sum_\infty$e∈*some-chilbert-basis*. e •$_C$ (A *$*_V$ e)) *else* 0)›

lemma *trace-0*[*simp*]: ‹*trace* 0 = 0›
unfolding *trace-def* **by** *simp*

lemma *trace-class-abs-op*[*simp*]: ‹*trace-class* (*abs-op A*) = *trace-class A*›
unfolding *trace-class-def*
by *simp*

lemma *trace-abs-op*[*simp*]: ‹*trace* (*abs-op A*) = *trace-norm A*›
proof (*cases* ‹*trace-class A*›)
case *True*
have *pos*: ‹*e* •$_C$ (*abs-op A* *$_V$ *e*) ≥ *0*› **for** *e*
by (*simp add*: *cinner-pos-if-pos*)
then have *abs*: ‹*e* •$_C$ (*abs-op A* *$_V$ *e*) = *abs* (*e* •$_C$ (*abs-op A* *$_V$ *e*))› **for** *e*
by (*simp add*: *abs-pos*)

have ‹*trace* (*abs-op A*) = (∑$_\infty$*e*∈*some-chilbert-basis*. *e* •$_C$ (*abs-op A* *$_V$ *e*))›
by (*simp add*: *trace-def True*)
also have ‹... = (∑$_\infty$*e*∈*some-chilbert-basis*. *complex-of-real* (*cmod* (*e* •$_C$ (*abs-op A* *$_V$ *e*))))›
using *pos abs complex-of-real-cmod* **by** *presburger*
also have ‹... = *complex-of-real* (∑$_\infty$*e*∈*some-chilbert-basis*. *cmod* (*e* •$_C$ (*abs-op A* *$_V$ *e*)))›
by (*simp add*: *infsum-of-real*)
also have ‹... = *trace-norm A*›
by (*simp add*: *trace-norm-def True*)
finally show *?thesis*
by −
next
case *False*
then show *?thesis*
by (*simp add*: *trace-def trace-norm-def*)
qed

lemma *trace-norm-pos*: ‹*trace-norm A* = *trace A*› **if** ‹*A* ≥ *0*›
by (*metis abs-op-id-on-pos that trace-abs-op*)

lemma *trace-norm-alt-def*:
assumes ‹*is-onb B*›
shows ‹*trace-norm A* = (*if trace-class A then* (∑$_\infty$*e*∈*B*. *cmod* (*e* •$_C$ (*abs-op A* *$_V$ *e*))) *else* *0*)›
by (*metis* (*mono-tags*, *lifting*) *assms infsum-eqI'* *is-onb-some-chilbert-basis trace-norm-basis-invariance trace-norm-def*)

lemma *trace-class-finite-dim*[*simp*]: ‹*trace-class A*› **for** *A* :: ‹'*a*::{*cfinite-dim*,*chilbert-space*} ⇒$_{CL}$ '*b*::*complex-inner*›
apply (*subst trace-class-iff-summable*[*of some-chilbert-basis*])
by (*auto intro*!: *summable-on-finite*)

lemma *trace-class-scaleC*: ‹*trace-class* (*c* *$_C$ *a*)› **if** ‹*trace-class a*›
proof −
from *that* **obtain** *B* **where** ‹*is-onb B*› **and** ‹(λ*x*. *x* •$_C$ (*abs-op a* *$_V$ *x*)) *abs-summable-on B*›
by (*auto simp*: *trace-class-def*)
then show *?thesis*

by (*auto intro*!: *exI*[*of* - *B*] *summable-on-cmult-right simp*: *trace-class-def* ‹*is-onb B*› *abs-op-scaleC norm-mult*)
qed

lemma *trace-scaleC*: ‹*trace* (*c* *$_C$ *a*) = *c* * *trace a*›
proof −
consider (*trace-class*) ‹*trace-class a*› | (*c0*) ‹*c* = *0*› | (*non-trace-class*) ‹¬ *trace-class a*› ‹*c* ≠ *0*›
by *auto*
then show *?thesis*
proof *cases*
case *trace-class*
then have ‹*trace-class* (*c* *$_C$ *a*)›
by (*rule trace-class-scaleC*)
then have ‹*trace* (*c* *$_C$ *a*) = (∑$_\infty$*e*∈*some-chilbert-basis*. *e* •$_C$ (*c* *$_C$ *a* *$_V$ *e*))›
unfolding *trace-def* **by** *simp*
also have ‹... = *c* * (∑$_\infty$*e*∈*some-chilbert-basis*. *e* •$_C$ (*a* *$_V$ *e*))›
by (*auto simp*: *infsum-cmult-right*′)
also from *trace-class* **have** ‹... = *c* * *trace a*›
by (*simp add*: *Trace-Class.trace-def*)
finally show *?thesis*
by −
next
case *c0*
then show *?thesis*
by *simp*
next
case *non-trace-class*
then have ‹¬ *trace-class* (*c* *$_C$ *a*)›
by (*metis divideC-field-simps*(*2*) *trace-class-scaleC*)
with *non-trace-class* **show** *?thesis*
by (*simp add*: *trace-def*)
qed
qed

lemma *trace-uminus*: ‹*trace* (− *a*) = − *trace a*›
by (*metis mult-minus1 scaleC-minus1-left trace-scaleC*)

lemma *trace-norm-0*[*simp*]: ‹*trace-norm 0* = *0*›
by (*auto simp*: *trace-norm-def*)

lemma *trace-norm-nneg*[*simp*]: ‹*trace-norm a* ≥ *0*›
apply (*cases* ‹*trace-class a*›)
by (*auto simp*: *trace-norm-def infsum-nonneg*)

lemma *trace-norm-scaleC*: ‹*trace-norm* (*c* *$_C$ *a*) = *norm c* * *trace-norm a*›
proof −
consider (*trace-class*) ‹*trace-class a*› | (*c0*) ‹*c* = *0*› | (*non-trace-class*) ‹¬ *trace-class a*› ‹*c* ≠ *0*›

```
    by auto
  then show ?thesis
  proof cases
    case trace-class
    then have ‹trace-class (c *C a)›
      by (rule trace-class-scaleC)
    then have ‹trace-norm (c *C a) = (∑∞e∈some-chilbert-basis. norm (e •C (abs-op (c *C
a) *V e)))›
      unfolding trace-norm-def by simp
    also have ‹... = norm c * (∑∞e∈some-chilbert-basis. norm (e •C (abs-op a *V e)))›
      by (auto simp: infsum-cmult-right' abs-op-scaleC norm-mult)
    also from trace-class have ‹... = norm c * trace-norm a›
      by (simp add: trace-norm-def)
    finally show ?thesis
      by −
  next
    case c0
    then show ?thesis
      by simp
  next
    case non-trace-class
    then have ‹¬ trace-class (c *C a)›
      by (metis divideC-field-simps(2) trace-class-scaleC)
    with non-trace-class show ?thesis
      by (simp add: trace-norm-def)
  qed
qed


lemma trace-norm-nondegenerate: ‹a = 0› if ‹trace-class a› and ‹trace-norm a = 0›
proof (rule ccontr)
  assume ‹a ≠ 0›
  then have ‹abs-op a ≠ 0›
    using abs-op-nondegenerate by blast
  then obtain x where xax: ‹x •C (abs-op a *V x) ≠ 0›
    by (metis cblinfun.zero-left cblinfun-cinner-eqI cinner-zero-right)
  then have ‹norm x ≠ 0›
    by auto
  then have xax': ‹sgn x •C (abs-op a *V sgn x) ≠ 0› and [simp]: ‹norm (sgn x) = 1›
    unfolding sgn-div-norm using xax by (auto simp: cblinfun.scaleR-right)
  obtain B where sgnx-B: ‹{sgn x} ⊆ B› and ‹is-onb B›
    apply atomize-elim apply (rule orthonormal-basis-exists)
    using ‹norm x ≠ 0› by (auto simp: is-ortho-set-def sgn-div-norm)

  from ‹is-onb B› that
  have summable: ‹(λe. e •C (abs-op a *V e)) abs-summable-on B›
    using trace-class-iff-summable by fastforce

  from that have ‹0 = trace-norm a›
```

by *simp*
also from ‹*is-onb B*› **have** ‹*trace-norm a* = $(\sum_\infty e{\in}B.$ *cmod* (*e* $\cdot_C$ (*abs-op a* $*_V$ *e*)))›
by (*smt* (*verit*, *ccfv-SIG*) *abs-norm-cancel infsum-cong infsum-not-exists real-norm-def trace-class-def trace-norm-alt-def*)
also have ‹... ≥ $(\sum_\infty e{\in}\{$*sgn x*$\}.$ *cmod* (*e* $\cdot_C$ (*abs-op a* $*_V$ *e*)))› (**is** ‹- ≥ ...›)
apply (*rule infsum-mono2*)
using *summable sgnx-B* **by** *auto*
also from *xax′* **have** ‹... > *0*›
by (*simp add*: *is-orthogonal-sym xax′*)
finally show *False*
by *simp*
qed

typedef (**overloaded**) (*'a*::*chilbert-space*, *'b*::*chilbert-space*) *trace-class* = ‹*Collect trace-class* :: (*'a* $\Rightarrow_{CL}$ *'b*) *set*›
morphisms *from-trace-class Abs-trace-class*
by (*auto intro*!: *exI*[*of* - *0*])
setup-lifting *type-definition-trace-class*

lemma *trace-class-from-trace-class*[*simp*]: ‹*trace-class* (*from-trace-class t*)›
using *from-trace-class* **by** *blast*

lemma *trace-pos*: ‹*trace a* ≥ *0*› **if** ‹*a* ≥ *0*›
by (*metis abs-op-def complex-of-real-nn-iff sqrt-op-unique that trace-abs-op trace-norm-nneg*)

lemma *trace-adj-prelim*: ‹*trace* (*a**) = *cnj* (*trace a*)› **if** ‹*trace-class a*› **and** ‹*trace-class* (*a**)›
— We will later strengthen this as *trace-adj* and then hide this fact.
by (*simp add*: *trace-def that flip*: *cinner-adj-right infsum-cnj*)

12.3 Hilbert-Schmidt operators

definition *hilbert-schmidt* **where** ‹*hilbert-schmidt a* ⟷ *trace-class* (*a** o_{CL} *a*)›

definition *hilbert-schmidt-norm* **where** ‹*hilbert-schmidt-norm a* = *sqrt* (*trace-norm* (*a** o_{CL} *a*))›

lemma *hilbert-schmidtI*: ‹*hilbert-schmidt a*› **if** ‹*trace-class* (*a** o_{CL} *a*)›
using *that* **unfolding** *hilbert-schmidt-def* **by** *simp*

lemma *hilbert-schmidt-0*[*simp*]: ‹*hilbert-schmidt 0*›
unfolding *hilbert-schmidt-def* **by** *simp*

lemma *hilbert-schmidt-norm-pos*[*simp*]: ‹*hilbert-schmidt-norm a* ≥ *0*›
by (*auto simp*: *hilbert-schmidt-norm-def*)

lemma *has-sum-hilbert-schmidt-norm-square*:
— [1], Proposition 18.6 (a)
assumes ‹*is-onb B*› **and** ‹*hilbert-schmidt a*›
shows ‹$((\lambda x.$ (*norm* (*a* $*_V$ *x*))$^2)$ *has-sum* (*hilbert-schmidt-norm a*)$^2)$ *B*›

proof −
from ‹*hilbert-schmidt a*›
have ‹*trace-class* (*a∗ o*$_{CL}$ *a*)›
using *hilbert-schmidt-def* **by** *blast*
with ‹*is-onb B*› **have** ‹((λ*x*. *cmod* (*x* ·$_C$ ((*a∗ o*$_{CL}$ *a*) ∗$_V$ *x*))) *has-sum trace-norm* (*a∗ o*$_{CL}$ *a*)) *B*›
by (*metis* (*no-types*, *lifting*) *abs-op-def has-sum-cong has-sum-infsum positive-cblinfun-squareI sqrt-op-unique trace-class-def trace-norm-alt-def trace-norm-basis-invariance*)
then show *?thesis*
by (*auto simp*: *cinner-adj-right cdot-square-norm of-real-power norm-power hilbert-schmidt-norm-def*)
qed

lemma *summable-hilbert-schmidt-norm-square*:
— [1], Proposition 18.6 (a)
assumes ‹*is-onb B*› **and** ‹*hilbert-schmidt a*›
shows ‹(λ*x*. (*norm* (*a* ∗$_V$ *x*))^2) *summable-on B*›
using *assms*(*1*) *assms*(*2*) *has-sum-hilbert-schmidt-norm-square summable-on-def* **by** *blast*

lemma *summable-hilbert-schmidt-norm-square-converse*:
assumes ‹*is-onb B*›
assumes ‹(λ*x*. (*norm* (*a* ∗$_V$ *x*))^2) *summable-on B*›
shows ‹*hilbert-schmidt a*›
proof −
from *assms*(*2*)
have ‹(λ*x*. *cmod* (*x* ·$_C$ ((*a∗ o*$_{CL}$ *a*) ∗$_V$ *x*))) *summable-on B*›
by (*metis* (*no-types*, *lifting*) *cblinfun-apply-cblinfun-compose cinner-adj-right cinner-pos-if-pos cmod-Re positive-cblinfun-squareI power2-norm-eq-cinner′ summable-on-cong*)
then have ‹*trace-class* (*a∗ o*$_{CL}$ *a*)›
by (*metis* (*no-types*, *lifting*) *abs-op-def assms*(*1*) *positive-cblinfun-squareI sqrt-op-unique summable-on-cong trace-class-def*)
then show *?thesis*
using *hilbert-schmidtI* **by** *blast*
qed

lemma *infsum-hilbert-schmidt-norm-square*:
— [1], Proposition 18.6 (a)
assumes ‹*is-onb B*› **and** ‹*hilbert-schmidt a*›
shows ‹(∑$_\infty$*x*∈*B*. (*norm* (*a* ∗$_V$ *x*))^2) = ((*hilbert-schmidt-norm a*)^2)›
using *assms has-sum-hilbert-schmidt-norm-square infsumI* **by** *blast*

lemma
— [1], Proposition 18.6 (d)
assumes ‹*hilbert-schmidt b*›
shows *hilbert-schmidt-comp-right*: ‹*hilbert-schmidt* (*a o*$_{CL}$ *b*)›
and *hilbert-schmidt-norm-comp-right*: ‹*hilbert-schmidt-norm* (*a o*$_{CL}$ *b*) ≤ *norm a* ∗ *hilbert-schmidt-norm b*›
proof −
define *B* :: ‹′*a set*› **where** ‹*B* = *some-chilbert-basis*›

have [*simp*]: ‹*is-onb B*›
by (*simp add*: *B-def*)

have *leq*: ‹(*norm* ((*a* o_{CL} *b*) $*_V$ *x*))^2 ≤ (*norm a*)^2 * (*norm* (*b* $*_V$ *x*))^2› **for** *x*
by (*metis cblinfun-apply-cblinfun-compose norm-cblinfun norm-ge-zero power-mono power-mult-distrib*)

have ‹(λ*x*. (*norm* (*b* $*_V$ *x*))^2) *summable-on B*›
using ‹*is-onb B*› *summable-hilbert-schmidt-norm-square assms* **by** *blast*
then have *sum2*: ‹(λ*x*. (*norm a*)^2 * (*norm* (*b* $*_V$ *x*))^2) *summable-on B*›
using *summable-on-cmult-right* **by** *blast*
then have ‹(λ*x*. ((*norm a*)^2 * (*norm* (*b* $*_V$ *x*))^2)) *abs-summable-on B*›
by *auto*
then have ‹(λ*x*. (*norm* ((*a* o_{CL} *b*) $*_V$ *x*))^2) *abs-summable-on B*›
apply (*rule abs-summable-on-comparison-test*)
using *leq* **by** *force*
then have *sum5*: ‹(λ*x*. (*norm* ((*a* o_{CL} *b*) $*_V$ *x*))^2) *summable-on B*›
by *auto*
then show [*simp*]: ‹*hilbert-schmidt* (*a* o_{CL} *b*)›
using ‹*is-onb B*›
by (*rule summable-hilbert-schmidt-norm-square-converse*[*rotated*])

have ‹(*hilbert-schmidt-norm* (*a* o_{CL} *b*))^2 = (∑$_\infty$*x*∈*B*. (*norm* ((*a* o_{CL} *b*) $*_V$ *x*))^2)›
apply (*rule infsum-hilbert-schmidt-norm-square*[*symmetric*])
by *simp-all*
also have ‹... ≤ (∑$_\infty$*x*∈*B*. (*norm a*)^2 * (*norm* (*b* $*_V$ *x*))^2)›
using *sum5 sum2 leq* **by** (*rule infsum-mono*)
also have ‹... = (*norm a*)^2 * (∑$_\infty$*x*∈*B*. (*norm* (*b* $*_V$ *x*))^2)›
by (*simp add*: *infsum-cmult-right'*)
also have ‹... = (*norm a*)^2 * (*hilbert-schmidt-norm b*)^2›
by (*simp add*: *assms infsum-hilbert-schmidt-norm-square*)
finally show ‹*hilbert-schmidt-norm* (*a* o_{CL} *b*) ≤ *norm a* * *hilbert-schmidt-norm b*›
apply (*rule-tac power2-le-imp-le*)
by (*auto intro*!: *mult-nonneg-nonneg simp*: *power-mult-distrib*)
qed

lemma *hilbert-schmidt-adj*[*simp*]:
— Implicit in [1], Proposition 18.6 (b)
assumes ‹*hilbert-schmidt a*›
shows ‹*hilbert-schmidt* (*a**)›
proof −
from *assms*
have ‹(λ*e*. (*norm* (*a* $*_V$ *e*))^2) *summable-on some-chilbert-basis*›
using *is-onb-some-chilbert-basis summable-hilbert-schmidt-norm-square* **by** *blast*
then have ‹(λ*e*. (*norm* (*a** $*_V$ *e*))^2) *summable-on some-chilbert-basis*›
by (*metis basis-image-square-has-sum2 is-onb-some-chilbert-basis summable-on-def*)
then show *?thesis*
using *is-onb-some-chilbert-basis summable-hilbert-schmidt-norm-square-converse* **by** *blast*
qed

lemma *hilbert-schmidt-norm-adj*[*simp*]:
— [1], Proposition 18.6 (b)
shows ‹*hilbert-schmidt-norm* (*a*∗) = *hilbert-schmidt-norm a*›
proof (*cases* ‹*hilbert-schmidt a*›)
case *True*
then have ‹((λ*x*. (*norm* (*a* ∗$_V$ *x*))^2) *has-sum* (*hilbert-schmidt-norm a*)^2) *some-chilbert-basis*›
by (*simp add*: *has-sum-hilbert-schmidt-norm-square*)
then have *1*: ‹((λ*x*. (*norm* (*a*∗ ∗$_V$ *x*))^2) *has-sum* (*hilbert-schmidt-norm a*)^2) *some-chilbert-basis*›
by (*metis basis-image-square-has-sum2 is-onb-some-chilbert-basis*)

from *True*
have ‹*hilbert-schmidt* (*a*∗)›
by *simp*
then have *2*: ‹((λ*x*. (*norm* (*a*∗ ∗$_V$ *x*))^2) *has-sum* (*hilbert-schmidt-norm* (*a*∗))^2) *some-chilbert-basis*›
by (*simp add*: *has-sum-hilbert-schmidt-norm-square*)

from *1 2* **show** *?thesis*
by (*metis abs-of-nonneg hilbert-schmidt-norm-pos infsumI real-sqrt-abs*)
next
case *False*
then have ‹¬ *hilbert-schmidt* (*a*∗)›
using *hilbert-schmidt-adj* **by** *fastforce*

then show *?thesis*
by (*metis False hilbert-schmidt-def hilbert-schmidt-norm-def trace-norm-def*)
qed

lemma
— [1], Proposition 18.6 (d)
fixes *a* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*› **and** *b*
assumes ‹*hilbert-schmidt a*›
shows *hilbert-schmidt-comp-left*: ‹*hilbert-schmidt* (*a* o$_{CL}$ *b*)›
apply (*subst asm-rl*[*of* ‹*a* o$_{CL}$ *b* = (*b*∗ o$_{CL}$ *a*∗)∗›], *simp*)
by (*auto intro*!: *assms hilbert-schmidt-comp-right hilbert-schmidt-adj simp del*: *adj-cblinfun-compose*)

lemma
— [1], Proposition 18.6 (d)
fixes *a* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*› **and** *b*
assumes ‹*hilbert-schmidt a*›
shows *hilbert-schmidt-norm-comp-left*: ‹*hilbert-schmidt-norm* (*a* o$_{CL}$ *b*) ≤ *norm b* ∗ *hilbert-schmidt-norm a*›
apply (*subst asm-rl*[*of* ‹*a* o$_{CL}$ *b* = (*b*∗ o$_{CL}$ *a*∗)∗›], *simp*)
using *hilbert-schmidt-norm-comp-right*[*of* ‹*a*∗› ‹*b*∗›]
by (*auto intro*!: *assms hilbert-schmidt-adj simp del*: *adj-cblinfun-compose*)

lemma *hilbert-schmidt-scaleC*: ‹*hilbert-schmidt* (*c* ∗$_C$ *a*)› **if** ‹*hilbert-schmidt a*›
using *hilbert-schmidt-def that trace-class-scaleC* **by** *fastforce*

lemma *hilbert-schmidt-scaleR*: ‹*hilbert-schmidt* $(r *_R a)$› **if** ‹*hilbert-schmidt a*›
by (*simp add*: *hilbert-schmidt-scaleC scaleR-scaleC that*)

lemma *hilbert-schmidt-uminus*: ‹*hilbert-schmidt* $(- a)$› **if** ‹*hilbert-schmidt a*›
by (*metis hilbert-schmidt-scaleC scaleC-minus1-left that*)

lemma *hilbert-schmidt-plus*: ‹*hilbert-schmidt* $(t + u)$› **if** ‹*hilbert-schmidt t*› **and** ‹*hilbert-schmidt u*›
for *t u* :: ‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*chilbert-space*›
— [1], Proposition 18.6 (e). We use a different proof than Conway: Our proof of *trace-class-plus* below was easy to adapt to Hilbert-Schmidt operators, so we adapted that one. However, Conway's proof would most likely work as well, and possible additionally allow us to weaken the sort of $'b$ to *complex-inner*.
proof −
define *II* :: ‹$'a \Rightarrow_{CL} ('a \times 'a)$› **where** ‹*II* = *cblinfun-left* + *cblinfun-right*›
define *JJ* :: ‹$('b \times 'b) \Rightarrow_{CL} 'b$› **where** ‹*JJ* = *cblinfun-left*∗ + *cblinfun-right*∗›
define *t2 u2* **where** ‹*t2* = *t*∗ o_{CL} *t*› **and** ‹*u2* = *u*∗ o_{CL} *u*›
define *tu* :: ‹$('a \times 'a) \Rightarrow_{CL} ('b \times 'b)$› **where** ‹*tu* = (*cblinfun-left* o_{CL} *t* o_{CL} *cblinfun-left*∗) + (*cblinfun-right* o_{CL} *u* o_{CL} *cblinfun-right*∗)›
define *tu2* :: ‹$('a \times 'a) \Rightarrow_{CL} ('a \times 'a)$› **where** ‹*tu2* = (*cblinfun-left* o_{CL} *t2* o_{CL} *cblinfun-left*∗) + (*cblinfun-right* o_{CL} *u2* o_{CL} *cblinfun-right*∗)›
have *t-plus-u*: ‹*t* + *u* = *JJ* o_{CL} *tu* o_{CL} *II*›
apply (*simp add*: *II-def JJ-def tu-def cblinfun-compose-add-left cblinfun-compose-add-right cblinfun-compose-assoc*)
by (*simp flip*: *cblinfun-compose-assoc*)
have *tu-tu2*: ‹*tu*∗ o_{CL} *tu* = *tu2*›
by (*simp add*: *tu-def tu2-def t2-def u2-def cblinfun-compose-add-left*
cblinfun-compose-add-right cblinfun-compose-assoc adj-plus
isometryD[*THEN simp-a-oCL-b*] *cblinfun-right-left-ortho*[*THEN simp-a-oCL-b*]
cblinfun-left-right-ortho[*THEN simp-a-oCL-b*])
have ‹*trace-class tu2*›
proof (*rule trace-classI*)
define *BL BR B* :: ‹$('a \times 'a)$ *set*› **where** ‹*BL* = *some-chilbert-basis* × {*0*}›
and ‹*BR* = {*0*} × *some-chilbert-basis*›
and ‹*B* = *BL* ∪ *BR*›
have ‹*BL* ∩ *BR* = {}›
using *is-ortho-set-some-chilbert-basis*
by (*auto simp*: *BL-def BR-def is-ortho-set-def*)
show ‹*is-onb B*›
by (*simp add*: *BL-def BR-def B-def is-onb-prod*)
have ‹*tu2* ≥ *0*›
by (*auto intro*!: *positive-cblinfunI simp*: *t2-def u2-def cinner-adj-right tu2-def cblinfun.add-left cinner-pos-if-pos*)
then have *abs-tu2*: ‹*abs-op tu2* = *tu2*›
by (*metis abs-opI*)
have *abs-t2*: ‹*abs-op t2* = *t2*›
by (*metis abs-opI positive-cblinfun-squareI t2-def*)
have *abs-u2*: ‹*abs-op u2* = *u2*›
by (*metis abs-opI positive-cblinfun-squareI u2-def*)

from *that(1)*
have ‹$(\lambda x.\ x \cdot_C (\textit{abs-op}\ t2 *_V x))$ *abs-summable-on some-chilbert-basis*›
by (*simp add*: *hilbert-schmidt-def t2-def trace-class-iff-summable*[*OF is-onb-some-chilbert-basis*])
then have ‹$(\lambda x.\ x \cdot_C (t2 *_V x))$ *abs-summable-on some-chilbert-basis*›
by (*simp add*: *abs-t2*)
then have *sum-BL*: ‹$(\lambda x.\ x \cdot_C (tu2 *_V x))$ *abs-summable-on BL*›
apply (*subst asm-rl*[*of* ‹*BL* = $(\lambda x.\ (x,0))$ ‘ *some-chilbert-basis*›])
by (*auto simp*: *BL-def summable-on-reindex inj-on-def o-def tu2-def cblinfun.add-left*)
from *that(2)*
have ‹$(\lambda x.\ x \cdot_C (\textit{abs-op}\ u2 *_V x))$ *abs-summable-on some-chilbert-basis*›
by (*simp add*: *hilbert-schmidt-def u2-def trace-class-iff-summable*[*OF is-onb-some-chilbert-basis*])
then have ‹$(\lambda x.\ x \cdot_C (u2 *_V x))$ *abs-summable-on some-chilbert-basis*›
by (*simp add*: *abs-u2*)
then have *sum-BR*: ‹$(\lambda x.\ x \cdot_C (tu2 *_V x))$ *abs-summable-on BR*›
apply (*subst asm-rl*[*of* ‹*BR* = $(\lambda x.\ (0,x))$ ‘ *some-chilbert-basis*›])
by (*auto simp*: *BR-def summable-on-reindex inj-on-def o-def tu2-def cblinfun.add-left*)
from *sum-BL sum-BR*
show ‹$(\lambda x.\ x \cdot_C (\textit{abs-op}\ tu2 *_V x))$ *abs-summable-on B*›
using ‹*BL* ∩ *BR* = {}›
by (*auto intro*!: *summable-on-Un-disjoint simp*: *B-def abs-tu2*)
qed
then have ‹*hilbert-schmidt tu*›
by (*auto simp flip*: *tu-tu2 intro*!: *hilbert-schmidtI*)
with *t-plus-u*
show ‹*hilbert-schmidt* $(t + u)$›
by (*auto intro*: *hilbert-schmidt-comp-left hilbert-schmidt-comp-right*)
qed

lemma *hilbert-schmidt-minus*: ‹*hilbert-schmidt* $(a - b)$› **if** ‹*hilbert-schmidt a*› **and** ‹*hilbert-schmidt b*›
for *a b* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
using *hilbert-schmidt-plus hilbert-schmidt-uminus that(1) that(2)* **by** *fastforce*

typedef (**overloaded**) ('*a*::*chilbert-space*,'*b*::*complex-inner*) *hilbert-schmidt* = ‹*Collect hilbert-schmidt* :: ('*a* ⇒$_{CL}$ '*b*) *set*›
by (*auto intro*!: *exI*[*of* - *0*])
setup-lifting *type-definition-hilbert-schmidt*

instantiation *hilbert-schmidt* :: (*chilbert-space*, *chilbert-space*)
{*zero*,*scaleC*,*uminus*,*plus*,*minus*,*dist-norm*,*sgn-div-norm*,*uniformity-dist*,*open-uniformity*} **begin**
lift-definition *zero-hilbert-schmidt* :: ‹('*a*,'*b*) *hilbert-schmidt*› **is** *0* **by** *auto*
lift-definition *norm-hilbert-schmidt* :: ‹('*a*,'*b*) *hilbert-schmidt* ⇒ *real*› **is** *hilbert-schmidt-norm*
.
lift-definition *scaleC-hilbert-schmidt* :: ‹*complex* ⇒ ('*a*,'*b*) *hilbert-schmidt* ⇒ ('*a*,'*b*) *hilbert-schmidt*› **is** *scaleC*
by (*simp add*: *hilbert-schmidt-scaleC*)
lift-definition *scaleR-hilbert-schmidt* :: ‹*real* ⇒ ('*a*,'*b*) *hilbert-schmidt* ⇒ ('*a*,'*b*) *hilbert-schmidt*›

is *scaleR*
by (*simp add*: *hilbert-schmidt-scaleR*)
lift-definition *uminus-hilbert-schmidt* :: ‹('a,'b) *hilbert-schmidt* ⇒ ('a,'b) *hilbert-schmidt*› **is** *uminus*
by (*simp add*: *hilbert-schmidt-uminus*)
lift-definition *minus-hilbert-schmidt* :: ‹('a,'b) *hilbert-schmidt* ⇒ ('a,'b) *hilbert-schmidt* ⇒ ('a,'b) *hilbert-schmidt*› **is** *minus*
by (*simp add*: *hilbert-schmidt-minus*)
lift-definition *plus-hilbert-schmidt* :: ‹('a,'b) *hilbert-schmidt* ⇒ ('a,'b) *hilbert-schmidt* ⇒ ('a,'b) *hilbert-schmidt*› **is** *plus*
by (*simp add*: *hilbert-schmidt-plus*)
definition ‹*dist a b* = *norm* (*a* − *b*)› **for** *a b* :: ‹('a,'b) *hilbert-schmidt*›
definition ‹*sgn x* = *inverse* (*norm x*) ∗$_R$ *x*› **for** *x* :: ‹('a,'b) *hilbert-schmidt*›
definition ‹*uniformity* = (*INF e*∈{*0*<..}. *principal* {(*x*::('a,'b) *hilbert-schmidt*, *y*). *dist x y* < *e*})›
definition ‹*open U* = (∀ *x*∈*U*. ∀$_F$ (*x'*, *y*) *in INF e*∈{*0*<..}. *principal* {(*x*, *y*). *norm* (*x* − *y*) < *e*}. *x'* = *x* ⟶ *y* ∈ *U*)› **for** *U* :: ‹('a,'b) *hilbert-schmidt set*›
instance
proof *intro-classes*
show ‹(∗$_R$) *r* = ((∗$_C$) (*complex-of-real r*) :: ('a,'b) *hilbert-schmidt* ⇒ -)› **for** *r* :: *real*
apply (*rule ext*)
apply *transfer*
by (*auto simp*: *scaleR-scaleC*)
show ‹*dist x y* = *norm* (*x* − *y*)› **for** *x y* :: ‹('a,'b) *hilbert-schmidt*›
by (*simp add*: *dist-hilbert-schmidt-def*)
show ‹*sgn x* = *inverse* (*norm x*) ∗$_R$ *x*› **for** *x* :: ‹('a,'b) *hilbert-schmidt*›
by (*simp add*: *Trace-Class.sgn-hilbert-schmidt-def*)
show ‹*uniformity* = (*INF e*∈{*0*<..}. *principal* {(*x*::('a,'b) *hilbert-schmidt*, *y*). *dist x y* < *e*})›
using *Trace-Class.uniformity-hilbert-schmidt-def* **by** *blast*
show ‹*open U* = (∀ *x*∈*U*. ∀$_F$ (*x'*, *y*) *in uniformity*. *x'* = *x* ⟶ *y* ∈ *U*)› **for** *U* :: ‹('a,'b) *hilbert-schmidt set*›
by (*simp add*: *uniformity-hilbert-schmidt-def open-hilbert-schmidt-def dist-hilbert-schmidt-def*)
qed
end

lift-definition *hs-compose* :: ‹('b::*chilbert-space*,'c::*complex-inner*) *hilbert-schmidt*
⇒ ('a::*chilbert-space*,'b) *hilbert-schmidt* ⇒ ('a,'c) *hilbert-schmidt*› **is**
cblinfun-compose
by (*simp add*: *hilbert-schmidt-comp-right*)

lemma
— [1], 18.8 Proposition
fixes *A* :: ‹'*a* :: *chilbert-space* ⇒$_{CL}$ '*b* :: *chilbert-space*›
shows *trace-class-iff-sqrt-hs*: ‹*trace-class A* ⟷ *hilbert-schmidt* (*sqrt-op* (*abs-op A*))› (**is** *?thesis1*)
and *trace-class-iff-hs-times-hs*: ‹*trace-class A* ⟷ (∃ *B* (*C*::'*a*⇒$_{CL}$'*a*). *hilbert-schmidt B* ∧ *hilbert-schmidt C* ∧ *A* = *B* *o*$_{CL}$ *C*)› (**is** *?thesis2*)
and *trace-class-iff-abs-hs-times-hs*: ‹*trace-class A* ⟷ (∃ *B* (*C*::'*a*⇒$_{CL}$'*a*). *hilbert-schmidt B* ∧ *hilbert-schmidt C* ∧ *abs-op A* = *B* *o*$_{CL}$ *C*)› (**is** *?thesis3*)

proof −
define *Sq W* **where** ‹*Sq* = *sqrt-op* (*abs-op A*)› **and** ‹*W* = *polar-decomposition A*›
have *trace-class-sqrt-hs*: ‹*hilbert-schmidt Sq*› **if** ‹*trace-class A*›
proof (*rule hilbert-schmidtI*)
from *that*
have ‹*trace-class* (*abs-op A*)›
by *simp*
then show ‹*trace-class* (*Sq*∗ o_{CL} *Sq*)›
by (*auto simp*: *Sq-def positive-hermitianI*)
qed
have *sqrt-hs-hs-times-hs*: ‹∃ *B* (*C* :: *'a* ⇒$_{CL}$ *'a*). *hilbert-schmidt B* ∧ *hilbert-schmidt C* ∧ *A* = *B* o_{CL} *C*›
if ‹*hilbert-schmidt Sq*›
proof −
have ‹*A* = *W* o_{CL} *abs-op A*›
by (*simp add*: *polar-decomposition-correct W-def*)
also have ‹... = (*W* o_{CL} *Sq*) o_{CL} *Sq*›
by (*metis Sq-def abs-op-pos cblinfun-compose-assoc positive-hermitianI sqrt-op-pos sqrt-op-square*)
finally have ‹*A* = (*W* o_{CL} *Sq*) o_{CL} *Sq*›
by −
then show *?thesis*
apply (*rule-tac exI*[*of* - ‹*W* o_{CL} *Sq*›], *rule-tac exI*[*of* - *Sq*])
using *that* **by** (*auto simp add*: *hilbert-schmidt-comp-right*)
qed
have *hs-times-hs-abs-hs-times-hs*: ‹∃ *B* (*C* :: *'a* ⇒$_{CL}$ *'a*). *hilbert-schmidt B* ∧ *hilbert-schmidt C* ∧ *abs-op A* = *B* o_{CL} *C*›
if ‹∃ *B* (*C* :: *'a* ⇒$_{CL}$ *'a*). *hilbert-schmidt B* ∧ *hilbert-schmidt C* ∧ *A* = *B* o_{CL} *C*›
proof −
from *that* **obtain** *B* **and** *C* :: ‹*'a* ⇒$_{CL}$ *'a*› **where** ‹*hilbert-schmidt B*› **and** ‹*hilbert-schmidt C*› **and** *ABC*: ‹*A* = *B* o_{CL} *C*›
by *auto*
from ‹*hilbert-schmidt B*›
have *hs-WB*: ‹*hilbert-schmidt* (*W*∗ o_{CL} *B*)›
by (*simp add*: *hilbert-schmidt-comp-right*)
have ‹*abs-op A* = *W*∗ o_{CL} *A*›
by (*simp add*: *W-def polar-decomposition-correct'*)
also have ‹... = (*W*∗ o_{CL} *B*) o_{CL} *C*›
by (*metis ABC cblinfun-compose-assoc*)
finally have ‹*abs-op A* = (*W*∗ o_{CL} *B*) o_{CL} *C*›
by −
with *hs-WB* ‹*hilbert-schmidt C*›
show *?thesis*
by *auto*
qed
have *abs-hs-times-hs-trace-class*: ‹*trace-class A*›
if ‹∃ *B* (*C* :: *'a* ⇒$_{CL}$ *'a*). *hilbert-schmidt B* ∧ *hilbert-schmidt C* ∧ *abs-op A* = *B* o_{CL} *C*›
proof −
from *that* **obtain** *B* **and** *C* :: ‹*'a* ⇒$_{CL}$ *'a*› **where** ‹*hilbert-schmidt B*› **and** ‹*hilbert-schmidt C*› **and** *ABC*: ‹*abs-op A* = *B* o_{CL} *C*›

by *auto*
from ‹*hilbert-schmidt B*›
have ‹*hilbert-schmidt* (*B**)›
by *simp*
then have ‹(λ*e*. (*norm* (*B** *$_V$ e*))^2) *abs-summable-on some-chilbert-basis*›
by (*metis is-onb-some-chilbert-basis summable-hilbert-schmidt-norm-square summable-on-iff-abs-summable-on-rea*
moreover
from ‹*hilbert-schmidt C*›
have ‹(λ*e*. (*norm* (*C* *$_V$ *e*))^2) *abs-summable-on some-chilbert-basis*›
by (*metis is-onb-some-chilbert-basis summable-hilbert-schmidt-norm-square summable-on-iff-abs-summable-on-rea*
ultimately have ‹(λ*e*. *norm* (*B** *$_V$ e*) * *norm* (*C* *$_V$ e*)) *abs-summable-on some-chilbert-basis*›
apply (*rule-tac abs-summable-product*)
by (*metis* (*no-types*, *lifting*) *power2-eq-square summable-on-cong*)+
then have ‹(λ*e*. *cinner e* (*abs-op A* *$_V$ *e*)) *abs-summable-on some-chilbert-basis*›
proof (*rule Infinite-Sum.abs-summable-on-comparison-test*)
fix *e* :: *'a* **assume** ‹*e* ∈ *some-chilbert-basis*›
have ‹*norm* (*e* •$_C$ (*abs-op A* *$_V$ *e*)) = *norm* ((*B** *$_V$ *e*) •$_C$ (*C* *$_V$ *e*))›
by (*simp add*: *ABC cinner-adj-left*)
also have ‹... ≤ *norm* (*B** *$_V$ *e*) * *norm* (*C* *$_V$ *e*)›
by (*rule Cauchy-Schwarz-ineq2*)
also have ‹... = *norm* (*norm* (*B** *$_V$ *e*) * *norm* (*C* *$_V$ *e*))›
by *simp*
finally show ‹*cmod* (*e* •$_C$ (*abs-op A* *$_V$ *e*)) ≤ *norm* (*norm* (*B** *$_V$ *e*) * *norm* (*C* *$_V$ *e*))›
by −
qed
then show ‹*trace-class A*›
apply (*rule trace-classI*[*rotated*]) **by** *simp*
qed
from *trace-class-sqrt-hs sqrt-hs-hs-times-hs hs-times-hs-abs-hs-times-hs abs-hs-times-hs-trace-class*
show *?thesis1* **and** *?thesis2* **and** *?thesis3*
unfolding *Sq-def* **by** *metis*+
qed

lemma *trace-exists*:
— [1], Proposition 18.9
assumes ‹*is-onb B*› **and** ‹*trace-class A*›
shows ‹(λ*e*. *e* •$_C$ (*A* *$_V$ *e*)) *summable-on B*›
proof −
obtain *b c* :: ‹*'a* ⇒$_{CL}$ *'a*› **where** ‹*hilbert-schmidt b*› ‹*hilbert-schmidt c*› **and** *Abc*: ‹*A* = *c** *o*$_{CL}$ *b*›
by (*metis abs-op-pos adj-cblinfun-compose assms*(*2*) *double-adj hilbert-schmidt-comp-left hilbert-schmidt-comp-right polar-decomposition-correct polar-decomposition-correct' positive-hermitianI trace-class-iff-hs-times-hs*)

have ‹(λ*e*. (*norm* (*b* *$_V$ *e*))^2) *summable-on B*›
using ‹*hilbert-schmidt b*› *assms*(*1*) *summable-hilbert-schmidt-norm-square* **by** *auto*
moreover have ‹(λ*e*. (*norm* (*c* *$_V$ *e*))^2) *summable-on B*›

using ‹*hilbert-schmidt c*› *assms*(*1*) *summable-hilbert-schmidt-norm-square* **by** *auto*
ultimately have ‹(λ*e*. (((*norm* (*b* *$_V$* *e*))^2 + (*norm* (*c* *$_V$* *e*))^2)) / *2*) *summable-on B*›
by (*auto intro*!: *summable-on-cdivide summable-on-add*)

then have ‹(λ*e*. (((*norm* (*b* *$_V$* *e*))^2 + (*norm* (*c* *$_V$* *e*))^2)) / *2*) *abs-summable-on B*›
by *simp*

then have ‹(λ*e*. *e* •$_C$ (*A* *$_V$* *e*)) *abs-summable-on B*›
proof (*rule abs-summable-on-comparison-test*)
fix *e* **assume** ‹*e* ∈ *B*›
obtain γ **where** ‹*cmod* γ = *1*› **and** γ: ‹γ * ((*b* *$_V$* *e*) •$_C$ (*c* *$_V$* *e*)) = *abs* ((*b* *$_V$* *e*) •$_C$ (*c* *$_V$* *e*))›
apply *atomize-elim*
apply (*cases* ‹(*b* *$_V$* *e*) •$_C$ (*c* *$_V$* *e*) ≠ *0*›)
apply (*rule exI*[*of* - ‹*cnj* (*sgn* ((*b* *$_V$* *e*) •$_C$ (*c* *$_V$* *e*)))›])
apply (*auto simp add*: *norm-sgn intro*!: *norm-one*)
by (*metis* (*no-types*, *lifting*) *abs-mult-sgn cblinfun.scaleC-right cblinfun-mult-right.rep-eq cdot-square-norm complex-norm-square complex-scaleC-def mult.comm-neutral norm-one norm-sgn one-cinner-one*)

have ‹*cmod* (*e* •$_C$ (*A* *$_V$* *e*)) = *Re* (*abs* (*e* •$_C$ (*A* *$_V$* *e*)))›
by (*metis abs-nn cmod-Re norm-abs*)
also have ‹... = *Re* (*abs* ((*b* *$_V$* *e*) •$_C$ (*c* *$_V$* *e*)))›
by (*metis* (*mono-tags*, *lifting*) *Abc abs-nn cblinfun-apply-cblinfun-compose cinner-adj-left cinner-commute' complex-mod-cnj complex-of-real-cmod norm-abs*)
also have ‹... = *Re* (((*b* *$_V$* *e*) •$_C$ (γ *$_C$* (*c* *$_V$* *e*))))›
by (*simp add*: γ)
also have ‹... ≤ ((*norm* (*b* *$_V$* *e*))^2 + (*norm* (γ *$_C$* (*c* *$_V$* *e*)))^2) / *2*›
by (*smt* (*z3*) *field-sum-of-halves norm-ge-zero polar-identity-minus zero-le-power-eq-numeral*)
also have ‹... = ((*norm* (*b* *$_V$* *e*))^2 + (*norm* (*c* *$_V$* *e*))^2) / *2*›
by (*simp add*: ‹*cmod* γ = *1*›)
also have ‹... ≤ *norm* (((*norm* (*b* *$_V$* *e*))^2 + (*norm* (*c* *$_V$* *e*))^2) / *2*)›
by *simp*
finally show ‹*cmod* (*e* •$_C$ (*A* *$_V$* *e*)) ≤ *norm* (((*norm* (*b* *$_V$* *e*))^2 + (*norm* (*c* *$_V$* *e*))^2) / *2*)›
by −
qed

then show *?thesis*
by (*metis abs-summable-summable*)

qed

lemma *trace-plus-prelim*:
assumes ‹*trace-class a*› ‹*trace-class b*› ‹*trace-class* (*a*+*b*)›
— We will later strengthen this as *trace-plus* and then hide this fact.
shows ‹*trace* (*a* + *b*) = *trace a* + *trace b*›
by (*auto simp add*: *assms infsum-add trace-def cblinfun.add-left cinner-add-right intro*!: *infsum-add trace-exists*)

lemma *hs-times-hs-trace-class*:
fixes $B :: \langle 'b::chilbert\text{-}space \Rightarrow_{CL} 'c::chilbert\text{-}space\rangle$ **and** $C :: \langle 'a::chilbert\text{-}space \Rightarrow_{CL} 'b::chilbert\text{-}space\rangle$
assumes ‹*hilbert-schmidt B*› **and** ‹*hilbert-schmidt C*›
shows ‹*trace-class* $(B\ o_{CL}\ C)$›
— Not an immediate consequence of *trace-class-iff-hs-times-hs* because here the types of B, C are more general.
proof −
define *A Sq W* **where** ‹$A = B\ o_{CL}\ C$› **and** ‹$Sq = sqrt\text{-}op\ (abs\text{-}op\ A)$› **and** ‹$W = polar\text{-}decomposition\ A$›

from ‹*hilbert-schmidt B*›
have *hs-WB*: ‹*hilbert-schmidt* $(W{*}\ o_{CL}\ B)$›
by (*simp add*: *hilbert-schmidt-comp-right*)
have ‹$abs\text{-}op\ A = W{*}\ o_{CL}\ A$›
by (*simp add*: *W-def polar-decomposition-correct'*)
also have ‹$\ldots = (W{*}\ o_{CL}\ B)\ o_{CL}\ C$›
by (*metis A-def cblinfun-compose-assoc*)
finally have *abs-op-A*: ‹$abs\text{-}op\ A = (W{*}\ o_{CL}\ B)\ o_{CL}\ C$›
by −
from ‹*hilbert-schmidt* $(W{*}\ o_{CL}\ B)$›
have ‹*hilbert-schmidt* $(B{*}\ o_{CL}\ W)$›
by (*simp add*: *assms*(*1*) *hilbert-schmidt-comp-left*)
then have ‹$(\lambda e.\ (norm\ ((B{*}\ o_{CL}\ W)\ *_V\ e))^2)$ *abs-summable-on some-chilbert-basis*›
by (*metis is-onb-some-chilbert-basis summable-hilbert-schmidt-norm-square summable-on-iff-abs-summable-on-real*)
moreover from ‹*hilbert-schmidt C*›
have ‹$(\lambda e.\ (norm\ (C\ *_V\ e))^2)$ *abs-summable-on some-chilbert-basis*›
by (*metis is-onb-some-chilbert-basis summable-hilbert-schmidt-norm-square summable-on-iff-abs-summable-on-real*)
ultimately have ‹$(\lambda e.\ norm\ ((B{*}\ o_{CL}\ W)\ *_V\ e) * norm\ (C\ *_V\ e))$ *abs-summable-on some-chilbert-basis*›
apply (*rule-tac abs-summable-product*)
by (*metis* (*no-types*, *lifting*) *power2-eq-square summable-on-cong*)+
then have ‹$(\lambda e.\ cinner\ e\ (abs\text{-}op\ A\ *_V\ e))$ *abs-summable-on some-chilbert-basis*›
proof (*rule Infinite-Sum.abs-summable-on-comparison-test*)
fix $e :: 'a$ **assume** ‹$e \in$ *some-chilbert-basis*›
have ‹$norm\ (e \cdot_C (abs\text{-}op\ A\ *_V\ e)) = norm\ (((B{*}\ o_{CL}\ W)\ *_V\ e) \cdot_C (C\ *_V\ e))$›
by (*simp add*: *abs-op-A cinner-adj-left cinner-adj-right*)
also have ‹$\ldots \leq norm\ ((B{*}\ o_{CL}\ W)\ *_V\ e) * norm\ (C\ *_V\ e)$›
by (*rule Cauchy-Schwarz-ineq2*)
also have ‹$\ldots = norm\ (norm\ ((B{*}\ o_{CL}\ W)\ *_V\ e)) * norm\ (C\ *_V\ e))$›
by *simp*
finally show ‹$cmod\ (e \cdot_C (abs\text{-}op\ A\ *_V\ e)) \leq norm\ (norm\ ((B{*}\ o_{CL}\ W)\ *_V\ e) * norm\ (C\ *_V\ e))$›
by −
qed
then show ‹*trace-class A*›
apply (*rule trace-classI*[*rotated*]) **by** *simp*
qed

instantiation *hilbert-schmidt* :: (*chilbert-space*, *chilbert-space*) *complex-vector* **begin**
instance
proof *intro-classes*
fix *a b c* :: ‹('*a*,'*b*) *hilbert-schmidt*›
show ‹*a* + *b* + *c* = *a* + (*b* + *c*)›
apply *transfer* **by** *auto*
show ‹*a* + *b* = *b* + *a*›
apply *transfer* **by** *auto*
show ‹*0* + *a* = *a*›
apply *transfer* **by** *auto*
show ‹− *a* + *a* = *0*›
apply *transfer* **by** *auto*
show ‹*a* − *b* = *a* + − *b*›
apply *transfer* **by** *auto*
show ‹*r* *$_C$ (*a* + *b*) = *r* *$_C$ *a* + *r* *$_C$ *b*› **for** *r* :: *complex*
apply *transfer*
using *scaleC-add-right*
by *auto*
show ‹(*r* + *r'*) *$_C$ *a* = *r* *$_C$ *a* + *r'* *$_C$ *a*› **for** *r r'* :: *complex*
apply *transfer*
by (*simp add*: *scaleC-add-left*)
show ‹*r* *$_C$ *r'* *$_C$ *a* = (*r* * *r'*) *$_C$ *a*› **for** *r r'*
apply *transfer* **by** *auto*
show ‹*1* *$_C$ *a* = *a*›
apply *transfer* **by** *auto*
qed
end

instantiation *hilbert-schmidt* :: (*chilbert-space*, *chilbert-space*) *complex-inner* **begin**
lift-definition *cinner-hilbert-schmidt* :: ‹('*a*,'*b*) *hilbert-schmidt* ⇒ ('*a*,'*b*) *hilbert-schmidt* ⇒ *complex*› **is**
‹λ*b c*. *trace* (*b** *$_{CL}$ *c*)› .
instance
proof *intro-classes*
fix *x y z* :: ‹('*a*,'*b*) *hilbert-schmidt*›
show ‹*x* ·*$_C$ *y* = *cnj* (*y* ·*$_C$ *x*)›
proof (*transfer*; *unfold mem-Collect-eq*)
fix *x y* :: ‹'*a* ⇒*$_{CL}$ '*b*›
assume *hs-xy*: ‹*hilbert-schmidt x*› ‹*hilbert-schmidt y*›
then have *tc*: ‹*trace-class* ((*y** *$_{CL}$ *x*)*)› ‹*trace-class* (*y** *$_{CL}$ *x*)›
by (*auto intro*!: *hs-times-hs-trace-class*)
have ‹*trace* (*x** *$_{CL}$ *y*) = *trace* ((*y** *$_{CL}$ *x*)*)›
by *simp*
also have ‹... = *cnj* (*trace* (*y** *$_{CL}$ *x*))›
using *tc trace-adj-prelim* **by** *blast*
finally show ‹*trace* (*x** *$_{CL}$ *y*) = *cnj* (*trace* (*y** *$_{CL}$ *x*))›
by −
qed

show ‹$(x + y) \cdot_C z = x \cdot_C z + y \cdot_C z$›
proof (*transfer*; *unfold mem-Collect-eq*)
fix $x\ y\ z :: \langle 'a \Rightarrow_{CL} 'b\rangle$
assume [*simp*]: ‹*hilbert-schmidt x*› ‹*hilbert-schmidt y*› ‹*hilbert-schmidt z*›
have [*simp*]: ‹*trace-class* $((x + y)* o_{CL} z)$› ‹*trace-class* $(x* o_{CL} z)$› ‹*trace-class* $(y* o_{CL} z)$›
by (*auto intro*!: *hs-times-hs-trace-class hilbert-schmidt-adj hilbert-schmidt-plus*)
then have [*simp*]: ‹*trace-class* $((x* o_{CL} z) + (y* o_{CL} z))$›
by (*simp add*: *adj-plus cblinfun-compose-add-left*)
show ‹*trace* $((x + y)* o_{CL} z) =$ *trace* $(x* o_{CL} z) +$ *trace* $(y* o_{CL} z)$›
by (*simp add*: *trace-plus-prelim adj-plus cblinfun-compose-add-left hs-times-hs-trace-class*)
qed
show ‹$r *_C x \cdot_C y =$ *cnj* $r *$ $(x \cdot_C y)$› **for** *r*
apply *transfer*
by (*simp add*: *trace-scaleC*)
show ‹$0 \leq x \cdot_C x$›
apply *transfer*
by (*simp add*: *positive-cblinfun-squareI trace-pos*)
show ‹$(x \cdot_C x = 0) = (x = 0)$›
proof (*transfer*; *unfold mem-Collect-eq*)
fix $x :: \langle 'a \Rightarrow_{CL} 'b\rangle$
assume [*simp*]: ‹*hilbert-schmidt x*›
have ‹*trace* $(x* o_{CL} x) = 0 \longleftrightarrow$ *trace* $($*abs-op* $(x* o_{CL} x)) = 0$›
by (*metis abs-op-def positive-cblinfun-squareI sqrt-op-unique*)
also have ‹$\ldots \longleftrightarrow$ *trace-norm* $(x* o_{CL} x) = 0$›
by *simp*
also have ‹$\ldots \longleftrightarrow x* o_{CL} x = 0$›
by (*metis* ‹*hilbert-schmidt x*› *hilbert-schmidt-def trace-norm-0 trace-norm-nondegenerate*)
also have ‹$\ldots \longleftrightarrow x = 0$›
using *cblinfun-compose-zero-right op-square-nondegenerate* **by** *blast*
finally show ‹*trace* $(x* o_{CL} x) = 0 \longleftrightarrow x = 0$›
by −
qed
show ‹*norm* $x =$ *sqrt* $($*cmod* $(x \cdot_C x))$›
apply *transfer*
apply (*auto simp*: *hilbert-schmidt-norm-def*)
by (*metis Re-complex-of-real cmod-Re positive-cblinfun-squareI trace-norm-pos trace-pos*)
qed
end

lemma *hilbert-schmidt-norm-triangle-ineq*:
— [1], Proposition 18.6 (e). We do not use their proof but get it as a simple corollary of the instantiation of *hilbert-schmidt* as a inner product space. The proof by Conway would probably allow us to weaken the sort of *'b* to *complex-inner*.
fixes *a b* :: ‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*chilbert-space*›
assumes ‹*hilbert-schmidt a*› ‹*hilbert-schmidt b*›
shows ‹*hilbert-schmidt-norm* $(a + b) \leq$ *hilbert-schmidt-norm a* + *hilbert-schmidt-norm b*›
proof −
define a' b' **where** ‹$a' =$ *Abs-hilbert-schmidt a*› **and** ‹$b' =$ *Abs-hilbert-schmidt b*›
have [*transfer-rule*]: ‹*cr-hilbert-schmidt a* a'›

```
    by (simp add: Abs-hilbert-schmidt-inverse a'-def assms(1) cr-hilbert-schmidt-def)
  have [transfer-rule]: ‹cr-hilbert-schmidt b b'›
    by (simp add: Abs-hilbert-schmidt-inverse assms(2) b'-def cr-hilbert-schmidt-def)
  have ‹norm (a' + b') ≤ norm a' + norm b'›
    by (rule norm-triangle-ineq)
  then show ?thesis
    apply transfer
    by –
qed

lift-definition adj-hs :: ‹('a::chilbert-space,'b::chilbert-space) hilbert-schmidt ⇒ ('b,'a) hilbert-schmidt›
is adj
  by auto

lemma adj-hs-plus: ‹adj-hs (x + y) = adj-hs x + adj-hs y›
  apply transfer
  by (simp add: adj-plus)

lemma adj-hs-minus: ‹adj-hs (x - y) = adj-hs x - adj-hs y›
  apply transfer
  by (simp add: adj-minus)

lemma norm-adj-hs[simp]: ‹norm (adj-hs x) = norm x›
  apply transfer
  by simp

lemma hilbert-schmidt-norm-geq-norm:
  — [1], Proposition 18.6 (c)
  assumes ‹hilbert-schmidt a›
  shows ‹norm a ≤ hilbert-schmidt-norm a›
proof –
  have ‹norm (a x) ≤ hilbert-schmidt-norm a› if ‹norm x = 1› for x
  proof –
    obtain B where ‹x ∈ B› and ‹is-onb B›
      using orthonormal-basis-exists[of ‹{x}›] ‹norm x = 1›
      by force
    have ‹(norm (a x))² = (∑∞x∈{x}. (norm (a x))²)›
      by simp
    also have ‹... ≤ (∑∞x∈B. (norm (a x))²)›
      apply (rule infsum-mono-neutral)
      by (auto intro!: summable-hilbert-schmidt-norm-square ‹is-onb B› assms ‹x ∈ B›)
    also have ‹... = (hilbert-schmidt-norm a)²›
      using infsum-hilbert-schmidt-norm-square[OF ‹is-onb B› assms]
      by –
    finally show ?thesis
      by force
  qed
  then show ?thesis
    by (auto intro!: norm-cblinfun-bound-unit)
```

qed

12.4 Trace-norm and trace-class, continued

lemma *trace-class-comp-left*: ‹*trace-class* (*a* o_{CL} *b*)› **if** ‹*trace-class a*› **for** *a* :: ‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*b*::*chilbert-space*›
— [1], Theorem 18.11 (a)
proof −
from ‹*trace-class a*›
obtain *C* :: ‹'*a* $\Rightarrow_{CL}$ '*b*› **and** *B* **where** ‹*hilbert-schmidt C*› **and** ‹*hilbert-schmidt B*› **and** *aCB*: ‹*a* = *C* o_{CL} *B*›
by (*auto simp*: *trace-class-iff-hs-times-hs*)
from ‹*hilbert-schmidt B*› **have** ‹*hilbert-schmidt* (*B* o_{CL} *b*)›
by (*simp add*: *hilbert-schmidt-comp-left*)
with ‹*hilbert-schmidt C*› **have** ‹*trace-class* (*C* o_{CL} (*B* o_{CL} *b*))›
using *hs-times-hs-trace-class* **by** *blast*
then show *?thesis*
by (*simp flip*: *aCB cblinfun-compose-assoc*)
qed

lemma *trace-class-comp-right*: ‹*trace-class* (*a* o_{CL} *b*)› **if** ‹*trace-class b*› **for** *a* :: ‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*b*::*chilbert-space*›
— [1], Theorem 18.11 (a)
proof −
from ‹*trace-class b*›
obtain *C* :: ‹'*c* $\Rightarrow_{CL}$ '*a*› **and** *B* **where** ‹*hilbert-schmidt C*› **and** ‹*hilbert-schmidt B*› **and** *aCB*: ‹*b* = *C* o_{CL} *B*›
by (*auto simp*: *trace-class-iff-hs-times-hs*)
from ‹*hilbert-schmidt C*› **have** ‹*hilbert-schmidt* (*a* o_{CL} *C*)›
by (*simp add*: *hilbert-schmidt-comp-right*)
with ‹*hilbert-schmidt B*› **have** ‹*trace-class* ((*a* o_{CL} *C*) o_{CL} *B*)›
using *hs-times-hs-trace-class* **by** *blast*
then show *?thesis*
by (*simp flip*: *aCB add*: *cblinfun-compose-assoc*)
qed

lemma
fixes *B* :: ‹'*a*::*chilbert-space set*› **and** *A* :: ‹'*a* $\Rightarrow_{CL}$ '*a*› **and** *b* :: ‹'*b*::*chilbert-space* $\Rightarrow_{CL}$ '*c*::*chilbert-space*› **and** *c* :: ‹'*c* $\Rightarrow_{CL}$ '*b*›
shows *trace-alt-def*:
— [1], Proposition 18.9
‹*is-onb B* $\Longrightarrow$ *trace A* = (*if trace-class A then* ($\sum_\infty e\in B.$ *e* $\cdot_C$ (*A* $*_V$ *e*)) *else 0*)›
and *trace-hs-times-hs*: ‹*hilbert-schmidt c* $\Longrightarrow$ *hilbert-schmidt b* $\Longrightarrow$ *trace* (*c* o_{CL} *b*) =
((*of-real* (*hilbert-schmidt-norm* ((*c*∗) + *b*)))^2 − (*of-real* (*hilbert-schmidt-norm* ((*c*∗) − *b*)))^2 −
i ∗ (*of-real* (*hilbert-schmidt-norm* (((*c*∗) + i $*_C$ *b*))))^2 +
i ∗ (*of-real* (*hilbert-schmidt-norm* (((*c*∗) − i $*_C$ *b*))))^2) / *4*›
proof −
have *ecbe-has-sum*: ‹((λ*e*. *e* $\cdot_C$ ((*c* o_{CL} *b*) $*_V$ *e*)) *has-sum*

((*of-real* (*hilbert-schmidt-norm* ((*c*∗) + *b*)))^2 − (*of-real* (*hilbert-schmidt-norm* ((*c*∗) − *b*)))^2 −
i ∗ (*of-real* (*hilbert-schmidt-norm* ((*c*∗) + i ∗$_C$ *b*)))^2 +
i ∗ (*of-real* (*hilbert-schmidt-norm* ((*c*∗) − i ∗$_C$ *b*)))^2) / *4*) *B*›
if ‹*is-onb B*› **and** ‹*hilbert-schmidt c*› **and** ‹*hilbert-schmidt b*› **for** *B* :: ‹'*y*::*chilbert-space set*› **and** *c* :: ‹'*x*::*chilbert-space* ⇒$_{CL}$ '*y*› **and** *b*
apply (*simp flip*: *cinner-adj-left*[*of c*])
apply (*subst cdot-norm*)
using *that* **by** (*auto simp add*: *field-class.field-divide-inverse infsum-cmult-left'*
simp del: *Num.inverse-eq-divide-numeral*
simp flip: *cblinfun.add-left cblinfun.diff-left cblinfun.scaleC-left of-real-power*
intro!: *has-sum-cmult-left has-sum-cmult-right has-sum-add has-sum-diff has-sum-of-real*
has-sum-hilbert-schmidt-norm-square hilbert-schmidt-plus hilbert-schmidt-minus hilbert-schmidt-scaleC)

then have *ecbe-infsum*: ‹(∑$_\infty$*e*∈*B*. *e* ∙$_C$ ((*c* o$_{CL}$ *b*) ∗$_V$ *e*)) =
(((*of-real* (*hilbert-schmidt-norm* ((*c*∗) + *b*)))^2 − (*of-real* (*hilbert-schmidt-norm* ((*c*∗) − *b*)))^2 −
i ∗ (*of-real* (*hilbert-schmidt-norm* ((*c*∗) + i ∗$_C$ *b*)))^2 +
i ∗ (*of-real* (*hilbert-schmidt-norm* ((*c*∗) − i ∗$_C$ *b*)))^2) / *4*)›
if ‹*is-onb B*› **and** ‹*hilbert-schmidt c*› **and** ‹*hilbert-schmidt b*› **for** *B* :: ‹'*y*::*chilbert-space set*› **and** *c* :: ‹'*x*::*chilbert-space* ⇒$_{CL}$ '*y*› **and** *b*
using *infsumI that*(*1*) *that*(*2*) *that*(*3*) **by** *blast*

show ‹*trace* (*c* o$_{CL}$ *b*) =
((*of-real* (*hilbert-schmidt-norm* ((*c*∗) + *b*)))^2 − (*of-real* (*hilbert-schmidt-norm* ((*c*∗) − *b*)))^2 −
i ∗ (*of-real* (*hilbert-schmidt-norm* ((*c*∗) + i ∗$_C$ *b*)))^2 +
i ∗ (*of-real* (*hilbert-schmidt-norm* ((*c*∗) − i ∗$_C$ *b*)))^2) / *4*›
if ‹*hilbert-schmidt c*› **and** ‹*hilbert-schmidt b*›
proof −
from *that* **have** *tc-cb*[*simp*]: ‹*trace-class* (*c* o$_{CL}$ *b*)›
by (*rule hs-times-hs-trace-class*)
show *?thesis*
using *ecbe-infsum*[*OF is-onb-some-chilbert-basis* ‹*hilbert-schmidt c*› ‹*hilbert-schmidt b*›]
apply (*simp only*: *trace-def*)
by *simp*
qed

show ‹*trace A* = (*if trace-class A then* (∑$_\infty$*e*∈*B*. *e* ∙$_C$ (*A* ∗$_V$ *e*)) *else 0*)› **if** ‹*is-onb B*›
proof (*cases* ‹*trace-class A*›)
case *True*
with *that*
obtain *b c* :: ‹'*a* ⇒$_{CL}$ '*a*› **where** *hs-b*: ‹*hilbert-schmidt b*› **and** *hs-c*: ‹*hilbert-schmidt c*› **and** *Acb*: ‹*A* = *c* o$_{CL}$ *b*›
by (*metis trace-class-iff-hs-times-hs*)
have [*simp*]: ‹*trace-class* (*c* o$_{CL}$ *b*)›
using *Acb True* **by** *auto*

show ‹*trace A* = (*if trace-class A then* (∑$_\infty$*e*∈*B*. *e* ∙$_C$ (*A* ∗$_V$ *e*)) *else 0*)›

```
    using ecbe-infsum[OF is-onb-some-chilbert-basis hs-c hs-b]
    using ecbe-infsum[OF ‹is-onb B› hs-c hs-b]
    by (simp only: Acb trace-def)
  next
    case False
    then show ?thesis
      by (simp add: trace-def)
  qed
qed

lemma trace-ket-sum:
  fixes A :: ‹'a ell2 ⇒CL 'a ell2›
  assumes ‹trace-class A›
  shows ‹trace A = (∑∞e. ket e •C (A *V ket e))›
  apply (subst infsum-reindex[where h=ket, unfolded o-def, symmetric])
  by (auto simp: ‹trace-class A› trace-alt-def[OF is-onb-ket] is-onb-ket)

lemma trace-one-dim[simp]: ‹trace A = one-dim-iso A› for A :: ‹'a::one-dim ⇒CL 'a›
proof −
  have onb: ‹is-onb {1 :: 'a}›
    by auto
  have ‹trace A = 1 •C (A *V 1)›
    apply (subst trace-alt-def)
     apply (fact onb)
    by simp
  also have ‹... = one-dim-iso A›
    by (simp add: cinner-cblinfun-def one-dim-iso-def)
  finally show ?thesis
    by −
qed


lemma trace-has-sum:
  assumes ‹is-onb E›
  assumes ‹trace-class t›
  shows ‹((λe. e •C (t *V e)) has-sum trace t) E›
  using assms(1) assms(2) trace-alt-def trace-exists by fastforce


lemma trace-sandwich-isometry[simp]: ‹trace (sandwich U A) = trace A› if ‹isometry U›
proof (cases ‹trace-class A›)
  case True
  note True[simp]
  have ‹is-ortho-set ((*V) U ‘ some-chilbert-basis)›
    unfolding is-ortho-set-def
    apply auto
   apply (metis (no-types, opaque-lifting) cblinfun-apply-cblinfun-compose cblinfun-id-cblinfun-apply
cinner-adj-right is-ortho-set-def is-ortho-set-some-chilbert-basis isometry-def that)
   by (metis is-normal-some-chilbert-basis isometry-preserves-norm norm-zero that zero-neq-one)
```

moreover have ‹$x \in (*_V)\ U$ ‘ *some-chilbert-basis* $\Longrightarrow$ *norm* $x = 1$› **for** x
using *is-normal-some-chilbert-basis isometry-preserves-norm that* **by** *fastforce*
ultimately obtain B **where** BU: ‹$B \supseteq U$ ‘ *some-chilbert-basis*› **and** ‹*is-onb* B›
apply *atomize-elim*
by (*rule orthonormal-basis-exists*)

have xUy: ‹$x \cdot_C\ U\ y = 0$› **if** xBU: ‹$x \in B - U$ ‘ *some-chilbert-basis*› **for** $x\ y$
proof −
from *that* ‹*is-onb* B› ‹*isometry* U›
have ‹$x \cdot_C\ z = 0$› **if** ‹$z \in U$ ‘ *some-chilbert-basis*› **for** z
using *that* **by** (*metis BU Diff-iff in-mono is-onb-def is-ortho-set-def*)
then have ‹$x \in$ *orthogonal-complement* (*closure* (*cspan* (U ‘ *some-chilbert-basis*)))›
by (*metis orthogonal-complementI orthogonal-complement-of-closure orthogonal-complement-of-cspan*)
then have ‹$x \in$ *space-as-set* (− *ccspan* (U ‘ *some-chilbert-basis*))›
by (*simp add*: *ccspan.rep-eq uminus-ccsubspace.rep-eq*)
then have ‹$x \in$ *space-as-set* (− ($U *_S$ *top*))›
by (*metis cblinfun-image-ccspan ccspan-some-chilbert-basis*)
moreover have ‹$U\ y \in$ *space-as-set* ($U *_S$ *top*)›
by *simp*
ultimately show *?thesis*
apply (*transfer fixing*: $x\ y$)
using *orthogonal-complement-orthoI* **by** *blast*
qed

have [*simp*]: ‹*trace-class* (*sandwich* $U\ A$)›
by (*simp add*: *sandwich.rep-eq trace-class-comp-left trace-class-comp-right*)
have ‹*trace* (*sandwich* $U\ A$) = ($\sum_\infty e \in B.\ e \cdot_C ((sandwich\ U *_V A) *_V e)$)›
using ‹*is-onb* B› *trace-alt-def* **by** *fastforce*
also have ‹... = ($\sum_\infty e \in U$ ‘ *some-chilbert-basis*. $e \cdot_C ((sandwich\ U *_V A) *_V e)$)›
apply (*rule infsum-cong-neutral*)
using $BU\ xUy$ **by** (*auto simp*: *sandwich-apply*)
also have ‹... = ($\sum_\infty e \in$ *some-chilbert-basis*. $U\ e \cdot_C ((sandwich\ U *_V A) *_V U\ e)$)›
apply (*subst infsum-reindex*)
apply (*metis cblinfun-apply-cblinfun-compose cblinfun-id-cblinfun-apply inj-on-inverseI isometry-def that*)
by (*auto simp*: *o-def*)
also have ‹... = ($\sum_\infty e \in$ *some-chilbert-basis*. $e \cdot_C\ A\ e$)›
apply (*rule infsum-cong*)
apply (*simp add*: *sandwich-apply flip*: *cinner-adj-right*)
by (*metis cblinfun-apply-cblinfun-compose cblinfun-id-cblinfun-apply isometry-def that*)
also have ‹... = *trace* A›
by (*simp add*: *trace-def*)
finally show *?thesis*
by −
next
case *False*
note *False*[*simp*]
then have [*simp*]: ‹¬ *trace-class* (*sandwich* $U\ A$)›
by (*smt* (*verit*, *ccfv-SIG*) *cblinfun-assoc-left*(1) *cblinfun-compose-id-left cblinfun-compose-id-right*

isometryD sandwich.rep-eq that trace-class-comp-left trace-class-comp-right)
show *?thesis*
by (*simp add*: *trace-def*)
qed

lemma *circularity-of-trace*:
— [1], Theorem 18.11 (e)
fixes $a :: \langle 'a :: chilbert\text{-}space \Rightarrow_{CL} 'b :: chilbert\text{-}space \rangle$ **and** $b :: \langle 'b \Rightarrow_{CL} 'a \rangle$
— The proof from [1] only work for square operators, we generalize it
assumes ‹*trace-class a*›
— Actually, *trace-class* $(a\ o_{CL}\ b) \wedge$ *trace-class* $(b\ o_{CL}\ a)$ is sufficient here, see [3] but the proof is more involved. Only *trace-class* $(a\ o_{CL}\ b)$ is not sufficient, see [4].
shows $\langle trace\ (a\ o_{CL}\ b) = trace\ (b\ o_{CL}\ a) \rangle$
proof −

define $a'\ b' :: \langle ('a \times 'b) \Rightarrow_{CL} ('a \times 'b) \rangle$
where $\langle a' = cblinfun\text{-}right\ o_{CL}\ a\ o_{CL}\ cblinfun\text{-}left* \rangle$
and $\langle b' = cblinfun\text{-}left\ o_{CL}\ b\ o_{CL}\ cblinfun\text{-}right* \rangle$

have ‹*trace-class a'*›
by (*simp add*: *a'-def assms trace-class-comp-left trace-class-comp-right*)

have *circ'*: $\langle trace\ (a'\ o_{CL}\ b') = trace\ (b'\ o_{CL}\ a') \rangle$
proof −
from ‹*trace-class a'*›
obtain $B\ C :: \langle ('a \times 'b) \Rightarrow_{CL} ('a \times 'b) \rangle$ **where** ‹*hilbert-schmidt B*› **and** ‹*hilbert-schmidt C*› **and** *aCB*: $\langle a' = C* \ o_{CL}\ B \rangle$
by (*metis abs-op-pos adj-cblinfun-compose double-adj hilbert-schmidt-comp-left hilbert-schmidt-comp-right polar-decomposition-correct polar-decomposition-correct' positive-hermitianI trace-class-iff-hs-times-hs*)
have *hs-iB*: $\langle hilbert\text{-}schmidt\ (\mathrm{i} *_C B) \rangle$
by (*metis Abs-hilbert-schmidt-inverse Rep-hilbert-schmidt* ‹*hilbert-schmidt B*› *mem-Collect-eq scaleC-hilbert-schmidt.rep-eq*)
have *: $\langle Re\ (trace\ (C* \ o_{CL}\ B)) = Re\ (trace\ (C\ o_{CL}\ B*)) \rangle$ **if** ‹*hilbert-schmidt B*› ‹*hilbert-schmidt C*› **for** $B\ C :: \langle ('a \times 'b) \Rightarrow_{CL} ('a \times 'b) \rangle$
proof −
from *that*
obtain $B'\ C'$ **where** $\langle B = Rep\text{-}hilbert\text{-}schmidt\ B' \rangle$ **and** $\langle C = Rep\text{-}hilbert\text{-}schmidt\ C' \rangle$
by (*meson Rep-hilbert-schmidt-cases mem-Collect-eq*)
then have [*transfer-rule*]: ‹*cr-hilbert-schmidt B B'*› ‹*cr-hilbert-schmidt C C'*›
by (*simp-all add*: *cr-hilbert-schmidt-def*)

have $\langle Re\ (trace\ (C* \ o_{CL}\ B)) = Re\ (C' \cdot_C B') \rangle$
apply *transfer* **by** *simp*
also have $\langle \ldots = (1/4) * ((norm\ (C' + B'))^2 - (norm\ (C' - B'))^2) \rangle$
by (*simp add*: *cdot-norm*)
also have $\langle \ldots = (1/4) * ((norm\ (adj\text{-}hs\ C' + adj\text{-}hs\ B'))^2 - (norm\ (adj\text{-}hs\ C' - adj\text{-}hs\ B'))^2) \rangle$
by (*simp add*: *flip*: *adj-hs-plus adj-hs-minus*)

```
    also have ‹... = Re (adj-hs C' ·C adj-hs B')›
      by (simp add: cdot-norm)
    also have ‹... = Re (trace (C oCL B*))›
      apply transfer by simp
    finally show ?thesis
      by −
  qed
  have **: ‹trace (C* oCL B) = cnj (trace (C oCL B*))› if ‹hilbert-schmidt B› ‹hilbert-schmidt
C› for B C :: ‹('a×'b) ⇒CL ('a×'b)›
    using *[OF ‹hilbert-schmidt B› ‹hilbert-schmidt C›]
    using *[OF hilbert-schmidt-scaleC[of - i, OF ‹hilbert-schmidt B›] ‹hilbert-schmidt C›]
    apply (auto simp: trace-scaleC cblinfun-compose-uminus-right trace-uminus)
    by (smt (verit, best) cnj.code complex.collapse)

  have ‹trace (b' oCL a') = trace ((b' oCL C*) oCL B)›
    by (simp add: aCB cblinfun-assoc-left(1))
  also from ** ‹hilbert-schmidt B› ‹hilbert-schmidt C› have ‹... = cnj (trace ((C oCL b'*)
oCL B*))›
    by (metis adj-cblinfun-compose double-adj hilbert-schmidt-comp-left)
  also have ‹... = cnj (trace (C oCL (B oCL b')*))›
    by (simp add: cblinfun-assoc-left(1))
  also  from ** ‹hilbert-schmidt B› ‹hilbert-schmidt C› have ‹... = trace (C* oCL (B oCL
b'))›
    by (simp add: hilbert-schmidt-comp-left)
  also have ‹... = trace (a' oCL b')›
    by (simp add: aCB cblinfun-compose-assoc)
  finally show ?thesis
    by simp
 qed

 have ‹trace (a oCL b) = trace (sandwich cblinfun-right (a oCL b) :: ('a×'b) ⇒CL ('a×'b))›
   by simp
 also have ‹... = trace (sandwich cblinfun-right (a oCL (cblinfun-left* oCL (cblinfun-left ::
-⇒CL('a×'b))) oCL b) :: ('a×'b) ⇒CL ('a×'b))›
   by simp
 also have ‹... = trace (a' oCL b')›
   by (simp only: a'-def b'-def sandwich-apply cblinfun-compose-assoc)
 also have ‹... = trace (b' oCL a')›
   by (rule circ')
 also have ‹... = trace (sandwich cblinfun-left (b oCL (cblinfun-right* oCL (cblinfun-right ::
-⇒CL('a×'b))) oCL a) :: ('a×'b) ⇒CL ('a×'b))›
   by (simp only: a'-def b'-def sandwich-apply cblinfun-compose-assoc)
 also have ‹... = trace (sandwich cblinfun-left (b oCL a) :: ('a×'b) ⇒CL ('a×'b))›
   by simp
 also have ‹... = trace (b oCL a)›
   by simp
 finally show ‹trace (a oCL b) = trace (b oCL a)›
   by −
qed
```

lemma *trace-butterfly-comp*: ‹*trace* (*butterfly x y* o_{CL} *a*) = *y* $\cdot_C$ (*a* $*_V$ *x*)›
proof −
have ‹*trace* (*butterfly x y* o_{CL} *a*) = *trace* (*vector-to-cblinfun y** o_{CL} (*a* o_{CL} (*vector-to-cblinfun x* :: *complex* $\Rightarrow_{CL}$ -)))›
unfolding *butterfly-def*
by (*metis cblinfun-compose-assoc circularity-of-trace trace-class-finite-dim*)
also have ‹... = *y* $\cdot_C$ (*a* $*_V$ *x*)›
by (*simp add*: *one-dim-iso-cblinfun-comp*)
finally show *?thesis*
by −
qed

lemma *trace-butterfly*: ‹*trace* (*butterfly x y*) = *y* $\cdot_C$ *x*›
using *trace-butterfly-comp*[**where** *a*=*id-cblinfun*] **by** *auto*

lemma *trace-butterfly-comp'*: ‹*trace* (*a* o_{CL} *butterfly x y*) = *y* $\cdot_C$ (*a* $*_V$ *x*)›
by (*simp add*: *cblinfun-comp-butterfly trace-butterfly*)

lemma *trace-norm-adj*[*simp*]: ‹*trace-norm* (*a**) = *trace-norm a*›
— [1], Theorem 18.11 (f)
proof −
have ‹*of-real* (*trace-norm* (*a**)) = *trace* (*sandwich* (*polar-decomposition a*) $*_V$ *abs-op a*)›
by (*metis abs-op-adj trace-abs-op*)
also have ‹... = *trace* ((*polar-decomposition a*)* o_{CL} (*polar-decomposition a*) o_{CL} *abs-op a*)›
by (*metis* (*no-types*, *lifting*) *abs-op-adj cblinfun-compose-assoc circularity-of-trace double-adj polar-decomposition-correct polar-decomposition-correct' sandwich.rep-eq trace-class-abs-op trace-def*)
also have ‹... = *trace* (*abs-op a*)›
by (*simp add*: *cblinfun-compose-assoc polar-decomposition-correct polar-decomposition-correct'*)
also have ‹... = *of-real* (*trace-norm a*)›
by *simp*
finally show *?thesis*
by *simp*
qed

lemma *trace-class-adj*[*simp*]: ‹*trace-class* (*a**)› **if** ‹*trace-class a*›
proof (*rule ccontr*)
assume *asm*: ‹¬ *trace-class* (*a**)›
then have ‹*trace-norm* (*a**) = *0*›
by (*simp add*: *trace-norm-def*)
then have ‹*trace-norm a* = *0*›
by (*metis trace-norm-adj*)
then have ‹*a* = *0*›
using *that trace-norm-nondegenerate* **by** *blast*
then have ‹*trace-class* (*a**)›
by *simp*
with *asm* **show** *False*

by *simp*
qed

lift-definition *adj-tc* :: ‹('*a*::*chilbert-space*, '*b*::*chilbert-space*) *trace-class* ⇒ ('*b*,'*a*) *trace-class*›
is *adj*
by *simp*

lift-definition *selfadjoint-tc* :: ‹('*a*::*chilbert-space*, '*a*) *trace-class* ⇒ *bool*› **is** *selfadjoint*.

lemma *selfadjoint-tc-def'*: ‹*selfadjoint-tc a* ⟷ *adj-tc a* = *a*›
apply *transfer*
using *selfadjoint-def* **by** *blast*

lemma *trace-class-finite-dim'*[*simp*]: ‹*trace-class A*› **for** *A* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::{*cfinite-dim*,*chilbert-space*}›
by (*metis double-adj trace-class-adj trace-class-finite-dim*)

lemma *trace-class-plus*[*simp*]:
fixes *t u* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
assumes ‹*trace-class t*› **and** ‹*trace-class u*›
shows ‹*trace-class* (*t* + *u*)›
— [1], Theorem 18.11 (a). However, we use a completely different proof that does not need the fact that trace class operators can be diagonalized with countably many diagonal elements.
proof −
define *II* :: ‹'*a* ⇒$_{CL}$ ('*a*×'*a*)› **where** ‹*II* = *cblinfun-left* + *cblinfun-right*›
define *JJ* :: ‹('*b*×'*b*) ⇒$_{CL}$ '*b*› **where** ‹*JJ* = *cblinfun-left*∗ + *cblinfun-right*∗›
define *tu* :: ‹('*a*×'*a*) ⇒$_{CL}$ ('*b*×'*b*)› **where** ‹*tu* = (*cblinfun-left* o_{CL} *t* o_{CL} *cblinfun-left*∗) + (*cblinfun-right* o_{CL} *u* o_{CL} *cblinfun-right*∗)›
have *t-plus-u*: ‹*t* + *u* = *JJ* o_{CL} *tu* o_{CL} *II*›
apply (*simp add*: *II-def JJ-def tu-def cblinfun-compose-add-left cblinfun-compose-add-right cblinfun-compose-assoc*)
by (*simp flip*: *cblinfun-compose-assoc*)
have ‹*trace-class tu*›
proof (*rule trace-classI*)
define *BL BR B* :: ‹('*a*×'*a*) *set*› **where** ‹*BL* = *some-chilbert-basis* × {*0*}›
and ‹*BR* = {*0*} × *some-chilbert-basis*›
and ‹*B* = *BL* ∪ *BR*›
have ‹*BL* ∩ *BR* = {}›
using *is-ortho-set-some-chilbert-basis*
by (*auto simp*: *BL-def BR-def is-ortho-set-def*)
show ‹*is-onb B*›
by (*simp add*: *BL-def BR-def B-def is-onb-prod*)
have *abs-tu*: ‹*abs-op tu* = (*cblinfun-left* o_{CL} *abs-op t* o_{CL} *cblinfun-left*∗) + (*cblinfun-right* o_{CL} *abs-op u* o_{CL} *cblinfun-right*∗)›
using [[*show-consts*]]
proof −
have ‹((*cblinfun-left* o_{CL} *abs-op t* o_{CL} *cblinfun-left*∗) + (*cblinfun-right* o_{CL} *abs-op u* o_{CL} *cblinfun-right*∗))∗
o_{CL} ((*cblinfun-left* o_{CL} *abs-op t* o_{CL} *cblinfun-left*∗) + (*cblinfun-right* o_{CL} *abs-op u* o_{CL} *cblinfun-right*∗))

```
    = tu* o_CL tu›
  proof −
    have tt[THEN simp-a-oCL-b, simp]: ‹(abs-op t)* o_CL abs-op t = t* o_CL t›
      by (simp add: abs-op-def positive-cblinfun-squareI positive-hermitianI)
    have uu[THEN simp-a-oCL-b, simp]: ‹(abs-op u)* o_CL abs-op u = u* o_CL u›
      by (simp add: abs-op-def positive-cblinfun-squareI positive-hermitianI)
    note isometryD[THEN simp-a-oCL-b, simp]
    note cblinfun-right-left-ortho[THEN simp-a-oCL-b, simp]
    note cblinfun-left-right-ortho[THEN simp-a-oCL-b, simp]
    show ?thesis
      using tt[of ‹cblinfun-left* :: ('a×'a) ⇒_CL 'a›] uu[of ‹cblinfun-right* :: ('a×'a) ⇒_CL
'a›]
      by (simp add: tu-def cblinfun-compose-add-right cblinfun-compose-add-left adj-plus
          cblinfun-compose-assoc)
  qed
  moreover have ‹(cblinfun-left o_CL abs-op t o_CL cblinfun-left*) + (cblinfun-right o_CL
abs-op u o_CL cblinfun-right*) ≥ 0›
    apply (rule positive-cblinfunI)
    by (auto simp: cblinfun.add-left cinner-pos-if-pos)
  ultimately show ?thesis
    by (rule abs-opI[symmetric])
qed
from assms(1)
have ‹(λx. x •_C (abs-op t *_V x)) abs-summable-on some-chilbert-basis›
by (metis is-onb-some-chilbert-basis summable-on-iff-abs-summable-on-complex trace-class-abs-op
trace-exists)
then have sum-BL: ‹(λx. x •_C (abs-op tu *_V x)) abs-summable-on BL›
  apply (subst asm-rl[of ‹BL = (λx. (x,0)) ‘ some-chilbert-basis›])
  by (auto simp: BL-def summable-on-reindex inj-on-def o-def abs-tu cblinfun.add-left)
from assms(2)
have ‹(λx. x •_C (abs-op u *_V x)) abs-summable-on some-chilbert-basis›
by (metis is-onb-some-chilbert-basis summable-on-iff-abs-summable-on-complex trace-class-abs-op
trace-exists)
then have sum-BR: ‹(λx. x •_C (abs-op tu *_V x)) abs-summable-on BR›
  apply (subst asm-rl[of ‹BR = (λx. (0,x)) ‘ some-chilbert-basis›])
  by (auto simp: BR-def summable-on-reindex inj-on-def o-def abs-tu cblinfun.add-left)
from sum-BL sum-BR
show ‹(λx. x •_C (abs-op tu *_V x)) abs-summable-on B›
  using ‹BL ∩ BR = {}›
  by (auto intro!: summable-on-Un-disjoint simp: B-def)
qed
with t-plus-u
show ‹trace-class (t + u)›
  by (simp add: trace-class-comp-left trace-class-comp-right)
qed

lemma trace-class-minus[simp]: ‹trace-class t ⟹ trace-class u ⟹ trace-class (t − u)›
  for t u :: ‹'a::chilbert-space ⇒_CL 'b::chilbert-space›
  by (metis trace-class-plus trace-class-uminus uminus-add-conv-diff)
```

lemma *trace-plus*:
assumes ‹*trace-class a*› ‹*trace-class b*›
shows ‹*trace* (*a* + *b*) = *trace a* + *trace b*›
by (*simp add*: *assms*(*1*) *assms*(*2*) *trace-plus-prelim*)
hide-fact *trace-plus-prelim*

lemma *trace-class-sum*:
fixes *a* :: ‹'*a* ⇒ '*b*::*chilbert-space* ⇒$_{CL}$ '*c*::*chilbert-space*›
assumes ‹⋀*i*. *i*∈*I* ⟹ *trace-class* (*a i*)›
shows ‹*trace-class* (∑ *i*∈*I*. *a i*)›
using *assms* **apply** (*induction I rule*:*infinite-finite-induct*)
by *auto*

lemma
assumes ‹⋀*i*. *i*∈*I* ⟹ *trace-class* (*a i*)›
shows *trace-sum*: ‹*trace* (∑ *i*∈*I*. *a i*) = (∑ *i*∈*I*. *trace* (*a i*))›
using *assms* **apply** (*induction I rule*:*infinite-finite-induct*)
by (*auto simp*: *trace-plus trace-class-sum*)

lemma *cmod-trace-times*: ‹*cmod* (*trace* (*a* o$_{CL}$ *b*)) ≤ *norm a* ∗ *trace-norm b*› **if** ‹*trace-class b*›
for *b* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
— [1], Theorem 18.11 (e)
proof −
define *W* **where** ‹*W* = *polar-decomposition b*›

have ‹*norm W* ≤ *1*›
by (*metis W-def norm-partial-isometry norm-zero order-refl polar-decomposition-partial-isometry zero-less-one-class.zero-le-one*)
have *hs1*: ‹*hilbert-schmidt* (*sqrt-op* (*abs-op b*))›
using *that trace-class-iff-sqrt-hs* **by** *blast*
then have *hs2*: ‹*hilbert-schmidt* (*sqrt-op* (*abs-op b*) o$_{CL}$ *W*∗ o$_{CL}$ *a*∗)›
by (*simp add*: *hilbert-schmidt-comp-left*)

from ‹*trace-class b*›
have ‹*trace-class* (*a* o$_{CL}$ *b*)›
using *trace-class-comp-right* **by** *blast*
then have *sum1*: ‹(λ*e*. *e* •$_C$ ((*a* o$_{CL}$ *b*) ∗$_V$ *e*)) *abs-summable-on some-chilbert-basis*›
by (*metis is-onb-some-chilbert-basis summable-on-iff-abs-summable-on-complex trace-exists*)

have *sum5*: ‹(λ*x*. (*norm* (*sqrt-op* (*abs-op b*) ∗$_V$ *x*))^2) *summable-on some-chilbert-basis*›
using *summable-hilbert-schmidt-norm-square*[*OF is-onb-some-chilbert-basis hs1*]
by (*simp add*: *power2-eq-square*)

have *sum4*: ‹(λ*x*. (*norm* ((*sqrt-op* (*abs-op b*) o$_{CL}$ *W*∗ o$_{CL}$ *a*∗) ∗$_V$ *x*))^2) *summable-on some-chilbert-basis*›
using *summable-hilbert-schmidt-norm-square*[*OF is-onb-some-chilbert-basis hs2*]
by (*simp add*: *power2-eq-square*)

have *sum3*: ‹(λ*e*. *norm* ((*sqrt-op* (*abs-op b*) o$_{CL}$ *W*∗ o$_{CL}$ *a*∗) ∗$_V$ *e*) ∗ *norm* (*sqrt-op* (*abs-op*

$b) *_V e))$ *summable-on some-chilbert-basis*›
apply (*rule abs-summable-summable*)
apply (*rule abs-summable-product*)
by (*intro sum4 sum5 summable-on-iff-abs-summable-on-real*[*THEN iffD1*])+

have *sum2*: ‹$(\lambda e.\ ((sqrt\text{-}op\ (abs\text{-}op\ b)\ o_{CL}\ W*\ o_{CL}\ a*)\ *_V\ e)\ \cdot_C\ (sqrt\text{-}op\ (abs\text{-}op\ b)\ *_V\ e))$ *abs-summable-on some-chilbert-basis*›
using *sum3*[*THEN summable-on-iff-abs-summable-on-real*[*THEN iffD1*]]
apply (*rule abs-summable-on-comparison-test*)
by (*simp add*: *complex-inner-class.Cauchy-Schwarz-ineq2*)

from ‹*trace-class b*›
have ‹*cmod* (*trace* ($a\ o_{CL}\ b$)) = *cmod* ($\sum_\infty e{\in}$*some-chilbert-basis*. $e\ \cdot_C\ ((a\ o_{CL}\ b)\ *_V\ e)$)›
by (*simp add*: *trace-class-comp-right trace-def*)
also have ‹$\ldots \leq (\sum_\infty e{\in}$*some-chilbert-basis*. *cmod* $(e\ \cdot_C\ ((a\ o_{CL}\ b)\ *_V\ e)))$›
using *sum1* **by** (*rule norm-infsum-bound*)
also have ‹$\ldots = (\sum_\infty e{\in}$*some-chilbert-basis*. *cmod* $(((sqrt\text{-}op\ (abs\text{-}op\ b)\ o_{CL}\ W*\ o_{CL}\ a*)\ *_V\ e)\ \cdot_C\ (sqrt\text{-}op\ (abs\text{-}op\ b)\ *_V\ e)))$›
apply (*simp add*: *positive-hermitianI flip*: *cinner-adj-right cblinfun-apply-cblinfun-compose*)
by (*metis* (*full-types*) *W-def abs-op-def cblinfun-compose-assoc polar-decomposition-correct sqrt-op-pos sqrt-op-square*)
also have ‹$\ldots \leq (\sum_\infty e{\in}$*some-chilbert-basis*. *norm* $((sqrt\text{-}op\ (abs\text{-}op\ b)\ o_{CL}\ W*\ o_{CL}\ a*)\ *_V\ e) * norm\ (sqrt\text{-}op\ (abs\text{-}op\ b)\ *_V\ e))$›
using *sum2 sum3* **apply** (*rule infsum-mono*)
using *complex-inner-class.Cauchy-Schwarz-ineq2* **by** *blast*
also have ‹$\ldots = (\sum_\infty e{\in}$*some-chilbert-basis*. *norm* $(norm\ ((sqrt\text{-}op\ (abs\text{-}op\ b)\ o_{CL}\ W*\ o_{CL}\ a*)\ *_V\ e) * norm\ (sqrt\text{-}op\ (abs\text{-}op\ b)\ *_V\ e)))$›
by *simp*
also have ‹$\ldots \leq sqrt\ (\sum_\infty e{\in}$*some-chilbert-basis*. $(norm\ (norm\ ((sqrt\text{-}op\ (abs\text{-}op\ b)\ o_{CL}\ W*\ o_{CL}\ a*)\ *_V\ e)))^2)$
$* sqrt\ (\sum_\infty e{\in}$*some-chilbert-basis*. $(norm\ (norm\ (sqrt\text{-}op\ (abs\text{-}op\ b)\ *_V\ e)))^2)$›
apply (*rule Cauchy-Schwarz-ineq-infsum*)
using *sum4 sum5* **by** *auto*
also have ‹$\ldots = sqrt\ (\sum_\infty e{\in}$*some-chilbert-basis*. $(norm\ ((sqrt\text{-}op\ (abs\text{-}op\ b)\ o_{CL}\ W*\ o_{CL}\ a*)\ *_V\ e))^2)$
$* sqrt\ (\sum_\infty e{\in}$*some-chilbert-basis*. $(norm\ (sqrt\text{-}op\ (abs\text{-}op\ b)\ *_V\ e))^2)$›
by *simp*
also have ‹$\ldots =$ *hilbert-schmidt-norm* $(sqrt\text{-}op\ (abs\text{-}op\ b)\ o_{CL}\ W*\ o_{CL}\ a*) *$ *hilbert-schmidt-norm* $(sqrt\text{-}op\ (abs\text{-}op\ b))$›
apply (*subst infsum-hilbert-schmidt-norm-square*, *simp*, *fact hs2*)
apply (*subst infsum-hilbert-schmidt-norm-square*, *simp*, *fact hs1*)
by *simp*
also have ‹$\ldots \leq$ *hilbert-schmidt-norm* $(sqrt\text{-}op\ (abs\text{-}op\ b)) * norm\ (W*\ o_{CL}\ a*) *$ *hilbert-schmidt-norm* $(sqrt\text{-}op\ (abs\text{-}op\ b))$›
by (*metis cblinfun-assoc-left*(*1*) *hilbert-schmidt-norm-comp-left hilbert-schmidt-norm-pos mult.commute mult-right-mono that trace-class-iff-sqrt-hs*)
also have ‹$\ldots \leq$ *hilbert-schmidt-norm* $(sqrt\text{-}op\ (abs\text{-}op\ b)) * norm\ (W*) * norm\ (a*) *$ *hilbert-schmidt-norm* $(sqrt\text{-}op\ (abs\text{-}op\ b))$›
by (*metis* (*no-types*, *lifting*) *ab-semigroup-mult-class.mult-ac*(*1*) *hilbert-schmidt-norm-pos*

mult-right-mono norm-cblinfun-compose ordered-comm-semiring-class.comm-mult-left-mono)
also have ‹... ≤ *hilbert-schmidt-norm* (*sqrt-op* (*abs-op b*)) ∗ *norm* (*a*∗) ∗ *hilbert-schmidt-norm* (*sqrt-op* (*abs-op b*))›
by (*metis* ‹*norm W* ≤ *1*› *hilbert-schmidt-norm-pos mult.right-neutral mult-left-mono mult-right-mono norm-adj norm-ge-zero*)
also have ‹... = *norm a* ∗ (*hilbert-schmidt-norm* (*sqrt-op* (*abs-op b*)))^2›
by (*simp add*: *power2-eq-square*)
also have ‹... = *norm a* ∗ *trace-norm b*›
apply (*simp add*: *hilbert-schmidt-norm-def positive-hermitianI*)
by (*metis abs-op-idem of-real-eq-iff trace-abs-op*)
finally show *?thesis*
by −
qed

lemma *trace-leq-trace-norm*[*simp*]: ‹*cmod* (*trace a*) ≤ *trace-norm a*›
proof (*cases* ‹*trace-class a*›)
case *True*
then have ‹*cmod* (*trace a*) ≤ *norm* (*id-cblinfun* :: *'a* ⇒$_{CL}$ *'a*) ∗ *trace-norm a*›
using *cmod-trace-times*[**where** *a*=‹*id-cblinfun* :: *'a* ⇒$_{CL}$ *'a*› **and** *b*=*a*]
by *simp*
also have ‹... ≤ *trace-norm a*›
apply (*rule mult-left-le-one-le*)
by (*auto intro*!: *mult-left-le-one-le simp*: *norm-cblinfun-id-le*)
finally show *?thesis*
by −
next
case *False*
then show *?thesis*
by (*simp add*: *trace-def*)
qed

lemma *trace-norm-triangle*:
fixes *a b* :: ‹*'a*::*chilbert-space* ⇒$_{CL}$ *'b*::*chilbert-space*›
assumes [*simp*]: ‹*trace-class a*› ‹*trace-class b*›
shows ‹*trace-norm* (*a* + *b*) ≤ *trace-norm a* + *trace-norm b*›
— [1], Theorem 18.11 (a)
proof −
define *w* **where** ‹*w* = *polar-decomposition* (*a*+*b*)›
have ‹*norm* (*w*∗) ≤ *1*›
by (*metis dual-order.refl norm-adj norm-partial-isometry norm-zero polar-decomposition-partial-isometry w-def zero-less-one-class.zero-le-one*)
have ‹*trace-norm* (*a* + *b*) = *cmod* (*trace* (*abs-op* (*a*+*b*)))›
by *simp*
also have ‹... = *cmod* (*trace* (*w*∗ o_{CL} (*a*+*b*)))›
by (*simp add*: *polar-decomposition-correct'* *w-def*)
also have ‹... ≤ *cmod* (*trace* (*w*∗ o_{CL} *a*)) + *cmod* (*trace* (*w*∗ o_{CL} *b*))›
by (*simp add*: *cblinfun-compose-add-right norm-triangle-ineq trace-class-comp-right trace-plus*)
also have ‹... ≤ (*norm* (*w*∗) ∗ *trace-norm a*) + (*norm* (*w*∗) ∗ *trace-norm b*)›
by (*smt* (*verit*, *best*) *assms*(*1*) *assms*(*2*) *cmod-trace-times*)

```
  also have ‹... ≤ trace-norm a + trace-norm b›
    using ‹norm (w*) ≤ 1›
    by (smt (verit, ccfv-SIG) mult-le-cancel-right2 trace-norm-nneg)
  finally show ?thesis
    by −
qed

instantiation trace-class :: (chilbert-space, chilbert-space) {complex-vector} begin

lift-definition zero-trace-class :: ‹('a,'b) trace-class› is 0 by auto
lift-definition minus-trace-class :: ‹('a,'b) trace-class ⇒ ('a,'b) trace-class ⇒ ('a,'b) trace-class›
is minus by auto
lift-definition uminus-trace-class :: ‹('a,'b) trace-class ⇒ ('a,'b) trace-class› is uminus by simp
lift-definition plus-trace-class :: ‹('a,'b) trace-class ⇒ ('a,'b) trace-class ⇒ ('a,'b) trace-class›
is plus by auto
lift-definition scaleC-trace-class :: ‹complex ⇒ ('a,'b) trace-class ⇒ ('a,'b) trace-class› is scaleC
  by (metis (no-types, opaque-lifting) cblinfun-compose-id-right cblinfun-compose-scaleC-right
mem-Collect-eq trace-class-comp-left)
lift-definition scaleR-trace-class :: ‹real ⇒ ('a,'b) trace-class ⇒ ('a,'b) trace-class› is scaleR
  by (metis (no-types, opaque-lifting) cblinfun-compose-id-right cblinfun-compose-scaleC-right
mem-Collect-eq scaleR-scaleC trace-class-comp-left)
instance
proof standard
  fix a b c :: ‹('a,'b) trace-class›
  show ‹a + b + c = a + (b + c)›
    apply transfer by auto
  show ‹a + b = b + a›
    apply transfer by auto
  show ‹0 + a = a›
    apply transfer by auto
  show ‹− a + a = 0›
    apply transfer by auto
  show ‹a − b = a + − b›
    apply transfer by auto
  show ‹(*R) r = ((*C) (complex-of-real r) :: - ⇒ ('a,'b) trace-class)› for r :: real
   by (metis (mono-tags, opaque-lifting) Trace-Class.scaleC-trace-class-def Trace-Class.scaleR-trace-class-def
id-apply map-fun-def o-def scaleR-scaleC)
  show ‹r *C (a + b) = r *C a + r *C b› for r :: complex
    apply transfer
    by (metis (no-types, lifting) scaleC-add-right)
  show ‹(r + r') *C a = r *C a + r' *C a› for r r' :: complex
    apply transfer
    by (metis (no-types, lifting) scaleC-add-left)
  show ‹r *C r' *C a = (r * r') *C a› for r r' :: complex
    apply transfer by auto
  show ‹1 *C a = a›
    apply transfer by auto
qed
end
```

lemma *from-trace-class-0*[*simp*]: ‹*from-trace-class 0 = 0*›
by (*simp add*: *zero-trace-class.rep-eq*)

instantiation *trace-class* :: (*chilbert-space*, *chilbert-space*) {*complex-normed-vector*} **begin**

lift-definition *norm-trace-class* :: ‹('*a*,'*b*) *trace-class* ⇒ *real*› **is** *trace-norm* .
definition *sgn-trace-class* :: ‹('*a*,'*b*) *trace-class* ⇒ ('*a*,'*b*) *trace-class*› **where** ‹*sgn-trace-class a* = *a* /$_R$ *norm a*›
definition *dist-trace-class* :: ‹('*a*,'*b*) *trace-class* ⇒ - ⇒ -› **where** ‹*dist-trace-class a b* = *norm* (*a* − *b*)›
definition [*code del*]: *uniformity-trace-class* = (*INF e*∈{*0*<..}. *principal* {(*x*::('*a*,'*b*) *trace-class*, *y*). *dist x y* < *e*})
definition [*code del*]: *open-trace-class U* = (∀ *x*∈*U*. ∀$_F$ (*x'*, *y*) *in INF e*∈{*0*<..}. *principal* {(*x*, *y*). *dist x y* < *e*}. *x'* = *x* ⟶ *y* ∈ *U*) **for** *U* :: ('*a*,'*b*) *trace-class set*
instance
proof *standard*
fix *a b* :: ‹('*a*,'*b*) *trace-class*›
show ‹*dist a b* = *norm* (*a* − *b*)›
by (*metis* (*no-types*, *lifting*) *Trace-Class.dist-trace-class-def*)
show ‹*uniformity* = (*INF e*∈{*0*<..}. *principal* {(*x* :: ('*a*,'*b*) *trace-class*, *y*). *dist x y* < *e*})›
by (*simp add*: *uniformity-trace-class-def*)
show ‹*open U* = (∀ *x*∈*U*. ∀$_F$ (*x'*, *y*) *in uniformity*. *x'* = *x* ⟶ *y* ∈ *U*)› **for** *U* :: ‹('*a*,'*b*) *trace-class set*›
by (*smt* (*verit*, *del-insts*) *case-prod-beta' eventually-mono open-trace-class-def uniformity-trace-class-def*)
show ‹(*norm a* = *0*) = (*a* = *0*)›
apply *transfer*
by (*auto simp add*: *trace-norm-nondegenerate*)
show ‹*norm* (*a* + *b*) ≤ *norm a* + *norm b*›
apply *transfer*
by (*auto simp*: *trace-norm-triangle*)
show ‹*norm* (*r* *$_C$ *a*) = *cmod r* * *norm a*› **for** *r*
apply *transfer*
by (*auto simp*: *trace-norm-scaleC*)
then show ‹*norm* (*r* *$_R$ *a*) = |*r*| * *norm a*› **for** *r*
by (*metis norm-of-real scaleR-scaleC*)
show ‹*sgn a* = *a* /$_R$ *norm a*›
by (*simp add*: *sgn-trace-class-def*)
qed
end

lemma *trace-norm-comp-right*:
fixes *a* :: ‹'*b*::*chilbert-space* ⇒$_{CL}$ '*c*::*chilbert-space*› **and** *b* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*›
assumes ‹*trace-class b*›

shows ‹*trace-norm* (*a* o_{CL} *b*) ≤ *norm a* ∗ *trace-norm b*›
— [1], Theorem 18.11 (g)
proof −
define *w w1 s* **where** ‹*w* = *polar-decomposition b*› **and** ‹*w1* = *polar-decomposition* (*a* o_{CL} *b*)›
and ‹*s* = *w1*∗ o_{CL} *a* o_{CL} *w*›
have *abs-ab*: ‹*abs-op* (*a* o_{CL} *b*) = *s* o_{CL} *abs-op b*›
by (*auto simp*: *w1-def w-def s-def cblinfun-compose-assoc polar-decomposition-correct polar-decomposition-correct′*)
have *norm-s-t*: ‹*norm s* ≤ *norm a*›
proof −
have ‹*norm s* ≤ *norm* (*w1*∗ o_{CL} *a*) ∗ *norm w*›
by (*simp add*: *norm-cblinfun-compose s-def*)
also have ‹. . . ≤ *norm* (*w1*∗) ∗ *norm a* ∗ *norm w*›
by (*metis mult.commute mult-left-mono norm-cblinfun-compose norm-ge-zero*)
also have ‹. . . ≤ *norm a*›
by (*metis* (*no-types*, *opaque-lifting*) *dual-order.refl mult.commute mult.right-neutral mult-zero-left norm-adj norm-ge-zero norm-partial-isometry norm-zero polar-decomposition-partial-isometry w1-def w-def*)
finally show *?thesis*
by −
qed
have ‹*trace-norm* (*a* o_{CL} *b*) = *cmod* (*trace* (*abs-op* (*a* o_{CL} *b*)))›
by *simp*
also have ‹. . . = *cmod* (*trace* (*s* o_{CL} *abs-op b*))›
using *abs-ab* **by** *presburger*
also have ‹. . . ≤ *norm s* ∗ *trace-norm* (*abs-op b*)›
using *assms* **by** (*simp add*: *cmod-trace-times*)
also from *norm-s-t* **have** ‹. . . ≤ *norm a* ∗ *trace-norm b*›
by (*metis abs-op-idem mult-right-mono of-real-eq-iff trace-abs-op trace-norm-nneg*)
finally show *?thesis*
by −
qed

lemma *trace-norm-comp-left*:
— [1], Theorem 18.11 (g)
fixes *a* :: ‹*'b*::*chilbert-space* ⇒$_{CL}$ *'c*::*chilbert-space*› **and** *b* :: ‹*'a*::*chilbert-space* ⇒$_{CL}$ *'b*›
assumes [*simp*]: ‹*trace-class a*›
shows ‹*trace-norm* (*a* o_{CL} *b*) ≤ *trace-norm a* ∗ *norm b*›
proof −
have ‹*trace-norm* (*b*∗ o_{CL} *a*∗) ≤ *norm* (*b*∗) ∗ *trace-norm* (*a*∗)›
apply (*rule trace-norm-comp-right*)
by *simp*
then have ‹*trace-norm* ((*b*∗ o_{CL} *a*∗)∗) ≤ *norm b* ∗ *trace-norm a*›
by (*simp del*: *adj-cblinfun-compose*)
then show *?thesis*
by (*simp add*: *mult.commute*)
qed

lemma *bounded-clinear-trace-duality*: ‹*trace-class t* ⟹ *bounded-clinear* (λ*a*. *trace* (*t* o_{CL} *a*))›
apply (*rule bounded-clinearI*[**where** *K*=‹*trace-norm t*›])
apply (*auto simp add*: *cblinfun-compose-add-right trace-class-comp-left trace-plus trace-scaleC*)[2]
by (*metis circularity-of-trace order-trans trace-leq-trace-norm trace-norm-comp-right*)

lemma *trace-class-butterfly*[*simp*]: ‹*trace-class* (*butterfly x y*)› **for** *x* :: ‹'*a*::*chilbert-space*› **and** *y* :: ‹'*b*::*chilbert-space*›
unfolding *butterfly-def*
apply (*rule trace-class-comp-left*)
by *simp*

lemma *trace-adj*: ‹*trace* (*a*∗) = *cnj* (*trace a*)›
by (*metis Complex-Inner-Product0.complex-inner-1-right cinner-zero-right double-adj is-onb-some-chilbert-basis is-orthogonal-sym trace-adj-prelim trace-alt-def trace-class-adj*)
hide-fact *trace-adj-prelim*

lemma *cmod-trace-times'*: ‹*cmod* (*trace* (*a* o_{CL} *b*)) ≤ *norm b* ∗ *trace-norm a*› **if** ‹*trace-class a*›
— [1], Theorem 18.11 (e)
apply (*subst asm-rl*[*of* ‹*a* o_{CL} *b* = (*b*∗ o_{CL} *a*∗)∗›], *simp*)
apply (*subst trace-adj*)
using *cmod-trace-times*[*of* ‹*a*∗› ‹*b*∗›]
by (*auto intro*!: *that trace-class-adj hilbert-schmidt-comp-right hilbert-schmidt-adj simp del*: *adj-cblinfun-compose*)

lift-definition *iso-trace-class-compact-op-dual'* :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*) *trace-class* ⇒ ('*b*,'*a*) *compact-op* ⇒$_{CL}$ *complex*› **is**
‹λ*t c*. *trace* (*from-compact-op c* o_{CL} *t*)›
proof (*rename-tac t*)
include *lifting-syntax*
fix *t* :: ‹'*a* ⇒$_{CL}$ '*b*›
assume ‹*t* ∈ *Collect trace-class*›
then have [*simp*]: ‹*trace-class t*›
by *simp*
have ‹*cmod* (*trace* (*from-compact-op x* o_{CL} *t*)) ≤ *norm x* ∗ *trace-norm t*› **for** *x*
by (*metis* ‹*trace-class t*› *cmod-trace-times from-compact-op-norm*)
then show ‹*bounded-clinear* (λ*c*. *trace* (*from-compact-op c* o_{CL} *t*))›
apply (*rule-tac bounded-clinearI*[**where** *K*=‹*trace-norm t*›])
by (*auto simp*: *from-compact-op-plus from-compact-op-scaleC cblinfun-compose-add-right cblinfun-compose-add-left trace-plus trace-class-comp-right trace-scaleC*)
qed

lemma *iso-trace-class-compact-op-dual'-apply*: ‹*iso-trace-class-compact-op-dual' t c* = *trace* (*from-compact-op c* o_{CL} *from-trace-class t*)›
by (*simp add*: *iso-trace-class-compact-op-dual'.rep-eq*)

lemma *iso-trace-class-compact-op-dual'-plus*: ‹*iso-trace-class-compact-op-dual'* (*a* + *b*) = *iso-trace-class-compact-op-dual' a* + *iso-trace-class-compact-op-dual' b*›
apply *transfer*

by (*simp add*: *cblinfun-compose-add-right trace-class-comp-right trace-plus*)

lemma *iso-trace-class-compact-op-dual'-scaleC*: ‹*iso-trace-class-compact-op-dual'* (*c* $*_C$ *a*) = *c* $*_C$ *iso-trace-class-compact-op-dual' a*›
apply *transfer*
by (*simp add*: *trace-scaleC*)

lemma *iso-trace-class-compact-op-dual'-bounded-clinear*[*bounded-clinear*, *simp*]:
— [1], Theorem 19.1
‹*bounded-clinear* (*iso-trace-class-compact-op-dual'* :: ('*a*::*chilbert-space*,'*b*::*chilbert-space*) *trace-class* ⇒ -)›
proof −
let *?iso* = ‹*iso-trace-class-compact-op-dual'* :: ('*a*,'*b*) *trace-class* ⇒ -›
have ‹*norm* (*?iso t*) ≤ *norm t*› **for** *t*
proof (*rule norm-cblinfun-bound*)
show ‹*norm t* ≥ *0*› **by** *simp*
fix *c*
show ‹*cmod* (*iso-trace-class-compact-op-dual' t* $*_V$ *c*) ≤ *norm t* ∗ *norm c*›
apply (*transfer fixing*: *c*)
apply *simp*
by (*metis cmod-trace-times from-compact-op-norm ordered-field-class.sign-simps*(*5*))
qed
then show ‹*bounded-clinear ?iso*›
apply (*rule-tac bounded-clinearI*[**where** *K*=*1*])
by (*auto simp*: *iso-trace-class-compact-op-dual'-plus iso-trace-class-compact-op-dual'-scaleC*)
qed

lemma *iso-trace-class-compact-op-dual'-surjective*[*simp*]:
‹*surj* (*iso-trace-class-compact-op-dual'* :: ('*a*::*chilbert-space*,'*b*::*chilbert-space*) *trace-class* ⇒ -)›
proof −
let *?iso* = ‹*iso-trace-class-compact-op-dual'* :: ('*a*,'*b*) *trace-class* ⇒ -›
have ‹∃ *A*. Φ = *?iso A*› **for** Φ :: ‹('*b*, '*a*) *compact-op* ⇒$_{CL}$ *complex*›
proof −
define *p* **where** ‹*p x y* = Φ (*butterfly-co y x*)› **for** *x y*
have *norm-p*: ‹*norm* (*p x y*) ≤ *norm* Φ ∗ *norm x* ∗ *norm y*› **for** *x y*
proof −
have ‹*norm* (*p x y*) ≤ *norm* Φ ∗ *norm* (*butterfly-co y x*)›
by (*auto simp*: *p-def norm-cblinfun*)
also have ‹... = *norm* Φ ∗ *norm* (*butterfly y x*)›
apply *transfer* **by** *simp*
also have ‹... = *norm* Φ ∗ *norm x* ∗ *norm y*›
by (*simp add*: *norm-butterfly*)
finally show *?thesis*
by −
qed
have [*simp*]: ‹*bounded-sesquilinear p*›
apply (*rule bounded-sesquilinear.intro*)
using *norm-p*

by (*auto*
intro!: *exI*[*of* - ‹*norm* Φ›]
simp add: *p-def butterfly-co-add-left butterfly-co-add-right complex-vector.linear-add cblinfun.scaleC-right cblinfun.scaleC-left ab-semigroup-mult-class.mult-ac*)
define *A* **where** ‹*A* = (*the-riesz-rep-sesqui p*)*›
then have *xAy*: ‹*x* •$_C$ (*A y*) = *p x y*› **for** *x y*
by (*simp add*: *cinner-adj-right the-riesz-rep-sesqui-apply*)
have Φ*C*: ‹Φ *C* = *trace* (*from-compact-op C* o_{CL} *A*)› **if** ‹*finite-rank* (*from-compact-op C*)› **for** *C*
proof −
from *that*
obtain *x y* **and** *n* :: *nat* **where** *C-sum*: ‹*from-compact-op C* = (∑ *i*<*n*. *butterfly* (*y i*) (*x i*))›
apply *atomize-elim* **by** (*rule finite-rank-sum-butterfly*)
then have ‹*C* = (∑ *i*<*n*. *butterfly-co* (*y i*) (*x i*))›
apply *transfer* **by** *simp*
then have ‹Φ *C* = (∑ *i*<*n*. Φ *$*$_V$ *butterfly-co* (*y i*) (*x i*))›
using *cblinfun.sum-right* **by** *blast*
also have ‹... = (∑ *i*<*n*. *p* (*x i*) (*y i*))›
using *p-def* **by** *presburger*
also have ‹... = (∑ *i*<*n*. (*x i*) •$_C$ (*A* (*y i*)))›
using *xAy* **by** *presburger*
also have ‹... = (∑ *i*<*n*. *trace* (*butterfly* (*y i*) (*x i*) o_{CL} *A*))›
by (*simp add*: *trace-butterfly-comp*)
also have ‹... = *trace* ((∑ *i*<*n*. *butterfly* (*y i*) (*x i*)) o_{CL} *A*)›
by (*metis* (*mono-tags*, *lifting*) *cblinfun-compose-sum-left sum.cong trace-class-butterfly trace-class-comp-left trace-sum*)
also have ‹... = *trace* (*from-compact-op C* o_{CL} *A*)›
using *C-sum* **by** *presburger*
finally show *?thesis*
by −
qed
have ‹*trace-class A*›
proof (*rule trace-classI*)
show ‹*is-onb some-chilbert-basis*›
by *simp*
define *W* **where** ‹*W* = *polar-decomposition A*›
have ‹*norm* (*W*∗) ≤ *1*›
by (*metis W-def nle-le norm-adj norm-partial-isometry norm-zero not-one-le-zero polar-decomposition-partial-isometry*)
have ‹(∑ *x*∈*E*. *cmod* (*x* •$_C$ (*abs-op A* *$_V$ *x*))) ≤ *norm* Φ› **if** ‹*finite E*› **and** ‹*E* ⊆ *some-chilbert-basis*› **for** *E*
proof −
define *CE* **where** ‹*CE* = (∑ *x*∈*E*. (*butterfly x x*))›
from ‹*E* ⊆ *some-chilbert-basis*›
have ‹*norm CE* ≤ *1*›
by (*auto intro*!: *sum-butterfly-is-Proj norm-is-Proj is-normal-some-chilbert-basis simp*: *CE-def is-ortho-set-antimono*)
have ‹(∑ *x*∈*E*. *cmod* (*x* •$_C$ (*abs-op A* *$_V$ *x*))) = *cmod* (∑ *x*∈*E*. *x* •$_C$ (*abs-op A* *$_V$ *x*))›

```
      apply (rule sum-cmod-pos)
      by (simp add: cinner-pos-if-pos)
    also have ‹... = cmod (∑ x∈E. (W *V x) •C (A *V x))›
      apply (rule arg-cong, rule sum.cong, simp)
   by (metis W-def cblinfun-apply-cblinfun-compose cinner-adj-right polar-decomposition-correct')
    also have ‹... = cmod (∑ x∈E. Φ (butterfly-co x (W x)))›
      apply (rule arg-cong, rule sum.cong, simp)
      by (simp flip: p-def xAy)
    also have ‹... = cmod (Φ (∑ x∈E. butterfly-co x (W x)))›
      by (simp add: cblinfun.sum-right)
    also have ‹... ≤ norm Φ * norm (∑ x∈E. butterfly-co x (W x))›
      using norm-cblinfun by blast
    also have ‹... = norm Φ * norm (∑ x∈E. butterfly x (W x))›
      apply transfer by simp
    also have ‹... = norm Φ * norm (∑ x∈E. (butterfly x x o_CL W*))›
      apply (rule arg-cong, rule sum.cong, simp)
      by (simp add: butterfly-comp-cblinfun)
    also have ‹... = norm Φ * norm (CE o_CL W*)›
      by (simp add: CE-def cblinfun-compose-sum-left)
    also have ‹... ≤ norm Φ›
      apply (rule mult-left-le, simp-all)
      using ‹norm CE ≤ 1› ‹norm (W*) ≤ 1›
      by (metis mult-le-one norm-cblinfun-compose norm-ge-zero order-trans)
    finally show ?thesis
      by −
  qed
  then show ‹(λx. x •C (abs-op A *V x)) abs-summable-on some-chilbert-basis›
    apply (rule-tac nonneg-bdd-above-summable-on)
    by (auto intro!: bdd-aboveI2)
 qed
 then obtain A' where A': ‹A = from-trace-class A'›
  using from-trace-class-cases by blast
 from ΦC have ΦC': ‹Φ C = ?iso A' C› if ‹finite-rank (from-compact-op C)› for C
  by (simp add: that iso-trace-class-compact-op-dual'-apply A')
 have ‹Φ = ?iso A'›
  apply (unfold cblinfun-apply-inject[symmetric])
  apply (rule finite-rank-separating-on-compact-op)
  using ΦC' by (auto intro!: cblinfun.bounded-clinear-right)
 then show ?thesis
  by auto
qed
then show ?thesis
 by auto
qed

lemma iso-trace-class-compact-op-dual'-isometric[simp]:
 — [1], Theorem 19.1
 ‹norm (iso-trace-class-compact-op-dual' t) = norm t› for t :: ‹('a::chilbert-space, 'b::chilbert-space)
trace-class›
```

proof −
let *?iso* = ‹*iso-trace-class-compact-op-dual′* :: (′*a*,′*b*) *trace-class* ⇒ -›
have ‹*norm* (*?iso t*) ≤ *norm t*› **for** *t*
proof (*rule norm-cblinfun-bound*)
show ‹*norm t* ≥ *0*› **by** *simp*
fix *c*
show ‹*cmod* (*iso-trace-class-compact-op-dual′ t* ∗*V c*) ≤ *norm t* ∗ *norm c*›
apply (*transfer fixing*: *c*)
apply *simp*
by (*metis cmod-trace-times from-compact-op-norm ordered-field-class.sign-simps*(*5*))
qed
moreover have ‹*norm* (*?iso t*) ≥ *norm t*› **for** *t*
proof −
define *s* **where** ‹*s E* = (∑ *e*∈*E*. *cmod* (*e* ·*C* (*abs-op* (*from-trace-class t*) ∗*V e*)))› **for** *E*
have *bound*: ‹*norm* (*?iso t*) ≥ *s E*› **if** ‹*finite E*› **and** ‹*E* ⊆ *some-chilbert-basis*› **for** *E*
proof −

Partial duplication from the proof of *iso-trace-class-compact-op-dual′-surjective*. In Conway's text, this subproof occurs only once. However, it did not become clear to use how this works: It seems that Conway's proof only implies that *iso-trace-class-compact-op-dual′* is isometric on the subset of trace-class operators A constructed in that proof, but not necessarily on others (if *iso-trace-class-compact-op-dual′* were non-injective, there might be others)

define *A* Φ **where** ‹*A* = *from-trace-class t*› **and** ‹Φ = *?iso t*›
define *W* **where** ‹*W* = *polar-decomposition A*›
have ‹*norm* (*W*∗) ≤ *1*›
by (*metis W-def nle-le norm-adj norm-partial-isometry norm-zero not-one-le-zero polar-decomposition-partial-isometry*)
define *CE* **where** ‹*CE* = (∑ *x*∈*E*. (*butterfly x x*))›
from ‹*E* ⊆ *some-chilbert-basis*›
have ‹*norm CE* ≤ *1*›
by (*auto intro*!: *sum-butterfly-is-Proj norm-is-Proj is-normal-some-chilbert-basis simp*: *CE-def is-ortho-set-antimono*)
have ‹*s E* = (∑ *x*∈*E*. *cmod* (*x* ·*C* (*abs-op A* ∗*V x*)))›
using *A-def s-def* **by** *blast*
also have ‹... = *cmod* (∑ *x*∈*E*. *x* ·*C* (*abs-op A* ∗*V x*))›
apply (*rule sum-cmod-pos*)
by (*simp add*: *cinner-pos-if-pos*)
also have ‹... = *cmod* (∑ *x*∈*E*. (*W* ∗*V x*) ·*C* (*A* ∗*V x*))›
apply (*rule arg-cong*, *rule sum.cong*, *simp*)
by (*metis W-def cblinfun-apply-cblinfun-compose cinner-adj-right polar-decomposition-correct′*)
also have ‹... = *cmod* (∑ *x*∈*E*. Φ (*butterfly-co x* (*W x*)))›
apply (*rule arg-cong*, *rule sum.cong*, *simp*)
by (*auto simp*: Φ-*def iso-trace-class-compact-op-dual′-apply butterfly-co.rep-eq trace-butterfly-comp simp flip*: *A-def*)
also have ‹... = *cmod* (Φ (∑ *x*∈*E*. *butterfly-co x* (*W x*)))›
by (*simp add*: *cblinfun.sum-right*)
also have ‹... ≤ *norm* Φ ∗ *norm* (∑ *x*∈*E*. *butterfly-co x* (*W x*))›

using *norm-cblinfun* **by** *blast*
also have ‹... = *norm* Φ * *norm* (∑ x∈E. *butterfly x* (*W x*))›
apply *transfer* **by** *simp*
also have ‹... = *norm* Φ * *norm* (∑ x∈E. (*butterfly x x* o_{CL} *W**))›
apply (*rule arg-cong*, *rule sum.cong*, *simp*)
by (*simp add*: *butterfly-comp-cblinfun*)
also have ‹... = *norm* Φ * *norm* (*CE* o_{CL} *W**)›
by (*simp add*: *CE-def cblinfun-compose-sum-left*)
also have ‹... ≤ *norm* Φ›
apply (*rule mult-left-le*, *simp-all*)
using ‹*norm CE* ≤ *1*› ‹*norm* (*W**) ≤ *1*›
by (*metis mult-le-one norm-cblinfun-compose norm-ge-zero order-trans*)
finally show *?thesis*
by (*simp add*: Φ-*def*)
qed
have ‹*trace-class* (*from-trace-class t*)› **and** ‹*norm t* = *trace-norm* (*from-trace-class t*)›
using *from-trace-class*
by (*auto simp add*: *norm-trace-class.rep-eq*)
then have ‹((λe. *cmod* (e •$_C$ (*abs-op* (*from-trace-class t*) *$_V$ e))) *has-sum norm t*) *some-chilbert-basis*›

by (*metis* (*no-types*, *lifting*) *has-sum-cong has-sum-infsum is-onb-some-chilbert-basis trace-class-def trace-norm-alt-def trace-norm-basis-invariance*)
then have *lim*: ‹(*s* ⟶ *norm t*) (*finite-subsets-at-top some-chilbert-basis*)›
by (*simp add*: *filterlim-iff has-sum-def s-def*)
show *?thesis*
using - - *lim* **apply** (*rule tendsto-le*)
by (*auto intro*!: *tendsto-const eventually-finite-subsets-at-top-weakI bound*)
qed
ultimately show *?thesis*
using *nle-le* **by** *blast*
qed

instance *trace-class* :: (*chilbert-space*, *chilbert-space*) *cbanach*
proof
let *?UNIVc* = ‹*UNIV* :: (('*b*,'*a*) *compact-op* ⇒$_{CL}$ *complex*) *set*›
let *?UNIVt* = ‹*UNIV* :: ('*a*,'*b*) *trace-class set*›
let *?iso* = ‹*iso-trace-class-compact-op-dual'* :: ('*a*,'*b*) *trace-class* ⇒ -›
have *lin-inv*[*simp*]: ‹*bounded-clinear* (*inv ?iso*)›
apply (*rule bounded-clinear-inv*[**where** *b*=*1*])
by *auto*
have [*simp*]: ‹*inj ?iso*›
proof (*rule injI*)
fix *x y* **assume** ‹*?iso x* = *?iso y*›
then have ‹*norm* (*?iso* (*x* − *y*)) = *0*›
by (*metis* (*no-types*, *opaque-lifting*) *add-diff-cancel-left diff-self iso-trace-class-compact-op-dual'-isometric iso-trace-class-compact-op-dual'-plus norm-eq-zero ordered-field-class.sign-simps*(*12*))
then have ‹*norm* (*x* − *y*) = *0*›
by *simp*

```
    then show ‹x = y›
      by simp
  qed
  have norm-inv[simp]: ‹norm (inv ?iso x) = norm x› for x
   by (metis iso-trace-class-compact-op-dual'-isometric iso-trace-class-compact-op-dual'-surjective
surj-f-inv-f)
  have ‹complete ?UNIVc›
    by (simp add: complete-UNIV)
  then have ‹complete (inv ?iso ‘ ?UNIVc)›
    apply (rule complete-isometric-image[rotated 4, where e=1])
    by (auto simp: bounded-clinear.bounded-linear)
  then have ‹complete ?UNIVt›
    by (simp add: inj-imp-surj-inv)
  then show ‹Cauchy X ⟹ convergent X› for X :: ‹nat ⇒ ('a, 'b) trace-class›
    by (simp add: complete-def convergent-def)
qed



lemma trace-norm-geq-cinner-abs-op: ‹ψ •C (abs-op t *V ψ) ≤ trace-norm t› if ‹trace-class t›
and ‹norm ψ = 1›
proof −
  have ‹∃B. {ψ} ⊆ B ∧ is-onb B›
    apply (rule orthonormal-basis-exists)
    using ‹norm ψ = 1›
    by auto
  then obtain B where ‹is-onb B› and ‹ψ ∈ B›
    by auto

  have ‹ψ •C (abs-op t *V ψ) = (∑∞ψ∈{ψ}. ψ •C (abs-op t *V ψ))›
    by simp
  also have ‹... ≤ (∑∞ψ∈B. ψ •C (abs-op t *V ψ))›
    apply (rule infsum-mono-neutral-complex)
    using ‹ψ ∈ B› ‹is-onb B› that
    by (auto simp add: trace-exists cinner-pos-if-pos)
  also have ‹... = trace-norm t›
    using ‹is-onb B› that
    by (metis trace-abs-op trace-alt-def trace-class-abs-op)
  finally show ?thesis
    by −
qed

lemma norm-leq-trace-norm: ‹norm t ≤ trace-norm t› if ‹trace-class t›
  for t :: ‹'a::chilbert-space ⇒CL 'b::chilbert-space›
proof −
  wlog not-singleton: ‹class.not-singleton TYPE('a)›
    using not-not-singleton-cblinfun-zero[of t] negation by simp
  note cblinfun-norm-approx-witness' = cblinfun-norm-approx-witness[internalize-sort' 'a, OF
complex-normed-vector-axioms not-singleton]
```

```
show ?thesis
proof (rule field-le-epsilon)
  fix ε :: real assume ‹ε > 0›

  define δ :: real where
    ‹δ = min (sqrt (ε / 2)) (ε / (4 * (norm (sqrt-op (abs-op t)) + 1)))›
  have ‹δ > 0›
    using ‹ε > 0› apply (auto simp add: δ-def)
    by (smt (verit) norm-not-less-zero zero-less-divide-iff)
  have δ-small: ‹δ^2 + 2 * norm (sqrt-op (abs-op t)) * δ ≤ ε›
  proof −
    define n where ‹n = norm (sqrt-op (abs-op t))›
    then have ‹n ≥ 0›
      by simp
    have δ: ‹δ = min (sqrt (ε / 2)) (ε / (4 * (n + 1)))›
      by (simp add: δ-def n-def)

    have ‹δ^2 + 2 * n * δ ≤ ε / 2 + 2 * n * δ›
      apply (rule add-right-mono)
      apply (subst δ) apply (subst min-power-distrib-left)
      using ‹ε > 0› ‹n ≥ 0› by auto
    also have ‹... ≤ ε / 2 + 2 * n * (ε / (4 * (n + 1)))›
      apply (intro add-left-mono mult-left-mono)
      by (simp-all add: δ ‹n ≥ 0›)
    also have ‹... = ε / 2 + 2 * (n / (n+1)) * (ε / 4)›
      by simp
    also have ‹... ≤ ε / 2 + 2 * 1 * (ε / 4)›
      apply (intro add-left-mono mult-left-mono mult-right-mono)
      using ‹n ≥ 0› ‹ε > 0› by auto
    also have ‹... = ε›
      by simp
    finally show ‹δ^2 + 2 * n * δ ≤ ε›
      by −
  qed

  from ‹δ > 0› obtain ψ where ψε: ‹norm (sqrt-op (abs-op t)) − δ ≤ norm (sqrt-op (abs-op
t) *_V ψ)› and ‹norm ψ = 1›
    apply atomize-elim by (rule cblinfun-norm-approx-witness')

  have aux1: ‹2 * complex-of-real x = complex-of-real (2 * x)› for x
    by simp

  have ‹complex-of-real (norm t) = norm (abs-op t)›
    by simp
  also have ‹... = (norm (sqrt-op (abs-op t)))^2›
    by (simp add: positive-hermitianI flip: norm-AadjA)
  also have ‹... ≤ (norm (sqrt-op (abs-op t) *_V ψ) + δ)^2›
      by (smt (verit) ψε complex-of-real-mono norm-triangle-ineq4 norm-triangle-sub pos2
```

power-strict-mono)
also have ‹... = (*norm* (*sqrt-op* (*abs-op t*) $*_V$ ψ))^2 + δ^2 + *2* * *norm* (*sqrt-op* (*abs-op t*) $*_V$ ψ) * δ›
by (*simp add*: *power2-sum*)
also have ‹... ≤ (*norm* (*sqrt-op* (*abs-op t*) $*_V$ ψ))^2 + δ^2 + *2* * *norm* (*sqrt-op* (*abs-op t*)) * δ›
apply (*rule complex-of-real-mono-iff*[*THEN iffD2*])
by (*smt* (*z3*) ‹*0* < δ› ‹*norm* ψ = *1*› *more-arith-simps*(*11*) *mult-less-cancel-right-disj norm-cblinfun one-power2 power2-eq-square*)
also have ‹... ≤ (*norm* (*sqrt-op* (*abs-op t*) $*_V$ ψ))^2 + ε›
apply (*rule complex-of-real-mono-iff*[*THEN iffD2*])
using δ-*small* **by** *auto*
also have ‹... = ((*sqrt-op* (*abs-op t*) $*_V$ ψ) $\cdot_C$ (*sqrt-op* (*abs-op t*) $*_V$ ψ)) + ε›
by (*simp add*: *cdot-square-norm*)
also have ‹... = (ψ $\cdot_C$ (*abs-op t* $*_V$ ψ)) + ε›
by (*simp add*: *positive-hermitianI flip*: *cinner-adj-right cblinfun-apply-cblinfun-compose*)
also have ‹... ≤ *trace-norm t* + ε›
using ‹*norm* ψ = *1*› ‹*trace-class t*› **by** (*auto simp add*: *trace-norm-geq-cinner-abs-op*)
finally show ‹*norm t* ≤ *trace-norm t* + ε›
using *complex-of-real-mono-iff* **by** *blast*
qed
qed

lemma *clinear-from-trace-class*[*iff*]: ‹*clinear from-trace-class*›
apply (*rule clinearI*; *transfer*)
by *auto*

lemma *bounded-clinear-from-trace-class*[*bounded-clinear*]:
‹*bounded-clinear* (*from-trace-class* :: ('*a*::*chilbert-space*,'*b*::*chilbert-space*) *trace-class* ⇒ -)›
proof (*cases* ‹*class.not-singleton TYPE*('*a*)›)
case *True*
show *?thesis*
apply (*rule bounded-clinearI*[**where** *K*=*1*]; *transfer*)
by (*auto intro*!: *norm-leq-trace-norm*[*internalize-sort'* '*a*] *chilbert-space-axioms True*)
next
case *False*
then have *zero*: ‹*A* = *0*› **for** *A* :: ‹'*a* ⇒$_{CL}$ '*b*›
by (*rule not-not-singleton-cblinfun-zero*)
show *?thesis*
apply (*rule bounded-clinearI*[**where** *K*=*1*])
by (*subst zero*, *simp*)+
qed

instantiation *trace-class* :: (*chilbert-space*, *chilbert-space*) *order* **begin**
lift-definition *less-eq-trace-class* :: ‹('*a*, '*b*) *trace-class* ⇒ ('*a*, '*b*) *trace-class* ⇒ *bool*› **is**
less-eq.
lift-definition *less-trace-class* :: ‹('*a*, '*b*) *trace-class* ⇒ ('*a*, '*b*) *trace-class* ⇒ *bool*› **is**
less.

instance
apply *intro-classes*
apply (*auto simp add*: *less-eq-trace-class.rep-eq less-trace-class.rep-eq*)
by (*simp add*: *from-trace-class-inject*)
end

lift-definition *compose-tcl* :: ‹('*a*::*chilbert-space*, '*b*::*chilbert-space*) *trace-class* ⇒ ('*c*::*chilbert-space* ⇒$_{CL}$ '*a*) ⇒ ('*c*,'*b*) *trace-class*› **is**
‹*cblinfun-compose* :: '*a* ⇒$_{CL}$ '*b* ⇒ '*c* ⇒$_{CL}$ '*a* ⇒ '*c* ⇒$_{CL}$ '*b*›
by (*simp add*: *trace-class-comp-left*)

lift-definition *compose-tcr* :: ‹('*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*) ⇒ ('*c*::*chilbert-space*, '*a*) *trace-class* ⇒ ('*c*,'*b*) *trace-class*› **is**
‹*cblinfun-compose* :: '*a* ⇒$_{CL}$ '*b* ⇒ '*c* ⇒$_{CL}$ '*a* ⇒ '*c* ⇒$_{CL}$ '*b*›
by (*simp add*: *trace-class-comp-right*)

lemma *norm-compose-tcl*: ‹*norm* (*compose-tcl a b*) ≤ *norm a* ∗ *norm b*›
by (*auto intro*!: *trace-norm-comp-left simp*: *norm-trace-class.rep-eq compose-tcl.rep-eq*)

lemma *norm-compose-tcr*: ‹*norm* (*compose-tcr a b*) ≤ *norm a* ∗ *norm b*›
by (*auto intro*!: *trace-norm-comp-right simp*: *norm-trace-class.rep-eq compose-tcr.rep-eq*)

interpretation *compose-tcl*: *bounded-cbilinear compose-tcl*
proof (*intro bounded-cbilinear.intro exI*[*of - 1*] *allI*)
fix *a a'* :: ‹('*a*,'*b*) *trace-class*› **and** *b b'* :: ‹'*c* ⇒$_{CL}$ '*a*› **and** *r* :: *complex*
show ‹*compose-tcl* (*a* + *a'*) *b* = *compose-tcl a b* + *compose-tcl a' b*›
apply *transfer*
by (*simp add*: *cblinfun-compose-add-left*)
show ‹*compose-tcl a* (*b* + *b'*) = *compose-tcl a b* + *compose-tcl a b'*›
apply *transfer*
by (*simp add*: *cblinfun-compose-add-right*)
show ‹*compose-tcl* (*r* ∗$_C$ *a*) *b* = *r* ∗$_C$ *compose-tcl a b*›
apply *transfer*
by *simp*
show ‹*compose-tcl a* (*r* ∗$_C$ *b*) = *r* ∗$_C$ *compose-tcl a b*›
apply *transfer*
by *simp*
show ‹*norm* (*compose-tcl a b*) ≤ *norm a* ∗ *norm b* ∗ *1*›
by (*simp add*: *norm-compose-tcl*)
qed

interpretation *compose-tcr*: *bounded-cbilinear compose-tcr*
proof (*intro bounded-cbilinear.intro exI*[*of - 1*] *allI*)
fix *a a'* :: ‹'*a* ⇒$_{CL}$ '*b*› **and** *b b'* :: ‹('*c*,'*a*) *trace-class*› **and** *r* :: *complex*
show ‹*compose-tcr* (*a* + *a'*) *b* = *compose-tcr a b* + *compose-tcr a' b*›
apply *transfer*
by (*simp add*: *cblinfun-compose-add-left*)
show ‹*compose-tcr a* (*b* + *b'*) = *compose-tcr a b* + *compose-tcr a b'*›

apply *transfer*
by (*simp add*: *cblinfun-compose-add-right*)
show ‹*compose-tcr* (*r* $*_C$ *a*) *b* = *r* $*_C$ *compose-tcr a b*›
apply *transfer*
by *simp*
show ‹*compose-tcr a* (*r* $*_C$ *b*) = *r* $*_C$ *compose-tcr a b*›
apply *transfer*
by *simp*
show ‹*norm* (*compose-tcr a b*) ≤ *norm a* ∗ *norm b* ∗ *1*›
by (*simp add*: *norm-compose-tcr*)
qed

lemma *trace-norm-sandwich*: ‹*trace-norm* (*sandwich e t*) ≤ (*norm e*)^2 ∗ *trace-norm t*› **if** ‹*trace-class t*›
apply (*simp add*: *sandwich-apply*)
by (*smt* (*z3*) *Groups.mult-ac*(*2*) *more-arith-simps*(*11*) *mult-left-mono norm-adj norm-ge-zero power2-eq-square that trace-class-comp-right trace-norm-comp-left trace-norm-comp-right*)

lemma *trace-class-sandwich*: ‹*trace-class b* ⟹ *trace-class* (*sandwich a b*)›
by (*simp add*: *sandwich-apply trace-class-comp-right trace-class-comp-left*)

definition ‹*sandwich-tc e t* = *compose-tcl* (*compose-tcr e t*) (*e*∗)›

lemma *sandwich-tc-transfer*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹((=) ===> *cr-trace-class* ===> *cr-trace-class*) (λ*e*. (∗$_V$) (*sandwich e*)) *sandwich-tc*›
by (*auto intro*!: *rel-funI simp*: *sandwich-tc-def cr-trace-class-def compose-tcl.rep-eq compose-tcr.rep-eq sandwich-apply*)

lemma *from-trace-class-sandwich-tc*:
‹*from-trace-class* (*sandwich-tc e t*) = *sandwich e* (*from-trace-class t*)›
apply *transfer*
by (*rule sandwich-apply*)

lemma *norm-sandwich-tc*: ‹*norm* (*sandwich-tc e t*) ≤ (*norm e*)^2 ∗ *norm t*›
by (*simp add*: *norm-trace-class.rep-eq from-trace-class-sandwich-tc trace-norm-sandwich*)

lemma *sandwich-tc-pos*: ‹*sandwich-tc e t* ≥ *0*› **if** ‹*t* ≥ *0*›
using *that* **apply** (*transfer fixing*: *e*)
by (*simp add*: *sandwich-pos*)

lemma *sandwich-tc-scaleC-right*: ‹*sandwich-tc e* (*c* $*_C$ *t*) = *c* $*_C$ *sandwich-tc e t*›
apply (*transfer fixing*: *e c*)
by (*simp add*: *cblinfun.scaleC-right*)

lemma *sandwich-tc-plus*: ‹*sandwich-tc e* (*t* + *u*) = *sandwich-tc e t* + *sandwich-tc e u*›
by (*simp add*: *sandwich-tc-def compose-tcr.add-right compose-tcl.add-left*)

lemma *sandwich-tc-minus*: ‹*sandwich-tc e* (*t* − *u*) = *sandwich-tc e t* − *sandwich-tc e u*›
by (*simp add*: *sandwich-tc-def compose-tcr.diff-right compose-tcl.diff-left*)

lemma *sandwich-tc-uminus-right*: ‹*sandwich-tc e* (− *t*) = − *sandwich-tc e t*›
by (*metis* (*no-types*, *lifting*) *add.right-inverse arith-simps*(*50*) *diff-0 group-cancel.sub1 sandwich-tc-minus*)

lemma *trace-comp-pos*:
fixes *a b* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*a*›
assumes ‹*trace-class b*›
assumes ‹*a* ≥ *0*› **and** ‹*b* ≥ *0*›
shows ‹*trace* (*a* o$_{CL}$ *b*) ≥ *0*›
proof −
obtain *c* :: ‹'*a* ⇒$_{CL}$ '*a*› **where** ‹*a* = *c*∗ o$_{CL}$ *c*›
by (*metis assms*(*2*) *positive-hermitianI sqrt-op-pos sqrt-op-square*)
then have ‹*trace* (*a* o$_{CL}$ *b*) = *trace* (*sandwich c b*)›
by (*simp add*: *sandwich-apply assms*(*1*) *cblinfun-assoc-left*(*1*) *circularity-of-trace trace-class-comp-right*)
also have ‹... ≥ *0*›
by (*auto intro*!: *trace-pos sandwich-pos assms*)
finally show *?thesis*
by −
qed

lemma *trace-norm-one-dim*: ‹*trace-norm x* = *cmod* (*one-dim-iso x*)›
apply (*rule of-real-eq-iff*[**where** '*a*=*complex*, *THEN iffD1*])
apply (*simp add*: *abs-op-one-dim flip*: *trace-abs-op*)
by (*simp add*: *abs-complex-def*)

lemma *trace-norm-bounded*:
fixes *A B* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*a*›
assumes ‹*A* ≥ *0*› **and** ‹*trace-class B*›
assumes ‹*A* ≤ *B*›
shows ‹*trace-class A*›
proof −
have ‹(λ*x*. *x* ∙$_{C}$ (*B* ∗$_{V}$ *x*)) *abs-summable-on some-chilbert-basis*›
by (*metis assms*(*2*) *is-onb-some-chilbert-basis summable-on-iff-abs-summable-on-complex trace-exists*)
then have ‹(λ*x*. *x* ∙$_{C}$ (*A* ∗$_{V}$ *x*)) *abs-summable-on some-chilbert-basis*›
apply (*rule abs-summable-on-comparison-test*)
using ‹*A* ≥ *0*› ‹*A* ≤ *B*›
by (*auto intro*!: *cmod-mono cinner-pos-if-pos simp*: *abs-op-id-on-pos less-eq-cblinfun-def*)
then show *?thesis*
by (*auto intro*!: *trace-classI*[*OF is-onb-some-chilbert-basis*] *simp*: *abs-op-id-on-pos* ‹*A* ≥ *0*›)
qed

lemma *trace-norm-cblinfun-mono*:
fixes *A B* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*a*›

assumes ‹$A \geq 0$› **and** ‹*trace-class* B›
assumes ‹$A \leq B$›
shows ‹*trace-norm* $A \leq$ *trace-norm* B›
proof −
from *assms* **have** ‹*trace-class* A›
by (*rule trace-norm-bounded*)
from ‹$A \leq B$› **and** ‹$A \geq 0$›
have ‹$B \geq 0$›
by *simp*
have ‹*cmod* $(x \cdot_C (\textit{abs-op}\ A *_V x)) \leq$ *cmod* $(x \cdot_C (\textit{abs-op}\ B *_V x))$› **for** x
using ‹$A \leq B$›
unfolding *less-eq-cblinfun-def*
using ‹$A \geq 0$› ‹$B \geq 0$›
by (*auto intro*!: *cmod-mono cinner-pos-if-pos simp*: *abs-op-id-on-pos*)
then show ‹*trace-norm* $A \leq$ *trace-norm* B›
using ‹*trace-class* A› ‹*trace-class* B›
by (*auto intro*!: *infsum-mono*
simp add: *trace-norm-def trace-class-iff-summable*[*OF is-onb-some-chilbert-basis*])
qed

lemma *norm-cblinfun-mono-trace-class*:
fixes $A\ B$ :: ‹('*a*::*chilbert-space*, '*a*) *trace-class*›
assumes ‹$A \geq 0$›
assumes ‹$A \leq B$›
shows ‹*norm* $A \leq$ *norm* B›
using *assms*
apply *transfer*
apply (*rule trace-norm-cblinfun-mono*)
by *auto*

lemma *trace-norm-butterfly*: ‹*trace-norm* (*butterfly a b*) = (*norm a*) ∗ (*norm b*)›
for *a b* :: ‹- :: *chilbert-space*›
proof −
have ‹*trace-norm* (*butterfly a b*) = *trace* (*abs-op* (*butterfly a b*))›
by (*simp flip*: *trace-abs-op*)
also have ‹... = (*norm a* / *norm b*) ∗ *trace* (*selfbutter b*)›
by (*simp add*: *abs-op-butterfly scaleR-scaleC trace-scaleC del*: *trace-abs-op*)
also have ‹... = (*norm a* / *norm b*) ∗ *trace* ((*vector-to-cblinfun b* :: *complex* $\Rightarrow_{CL}$ -)∗ o_{CL} *vector-to-cblinfun b*)›
apply (*subst butterfly-def*)
apply (*subst circularity-of-trace*)
by *simp-all*
also have ‹... = (*norm a* / *norm b*) ∗ (*b* $\cdot_C$ *b*)›
by *simp*
also have ‹... = (*norm a*) ∗ (*norm b*)›
by (*simp add*: *cdot-square-norm power2-eq-square*)
finally show *?thesis*

using *of-real-eq-iff* **by** *blast*
qed

lemma *from-trace-class-sum*:
shows ‹*from-trace-class* $(\sum x{\in}M.\ f\ x) = (\sum x{\in}M.\ \textit{from-trace-class}\ (f\ x))$›
apply (*induction M rule*:*infinite-finite-induct*)
by (*simp-all add*: *plus-trace-class.rep-eq*)

lemma *has-sum-mono-neutral-traceclass*:
fixes $f :: 'a \Rightarrow ('b{::}\textit{chilbert-space}, 'b)$ *trace-class*
assumes ‹$(f$ *has-sum* $a)\ A$› **and** $(g$ *has-sum* $b)\ B$
assumes ‹$\bigwedge x.\ x \in A{\cap}B \Longrightarrow f\ x \leq g\ x$›
assumes ‹$\bigwedge x.\ x \in A{-}B \Longrightarrow f\ x \leq 0$›
assumes ‹$\bigwedge x.\ x \in B{-}A \Longrightarrow g\ x \geq 0$›
shows $a \leq b$
proof −
from *assms*(*1*)
have ‹$((\lambda x.$ *from-trace-class* $(f\ x))$ *has-sum from-trace-class* $a)\ A$›
apply (*rule Infinite-Sum.has-sum-bounded-linear*[*rotated*])
by (*intro bounded-clinear-from-trace-class bounded-clinear.bounded-linear*)
moreover
from *assms*(*2*)
have ‹$((\lambda x.$ *from-trace-class* $(g\ x))$ *has-sum from-trace-class* $b)\ B$›
apply (*rule Infinite-Sum.has-sum-bounded-linear*[*rotated*])
by (*intro bounded-clinear-from-trace-class bounded-clinear.bounded-linear*)
ultimately have ‹*from-trace-class* $a \leq$ *from-trace-class* b›
apply (*rule has-sum-mono-neutral-cblinfun*)
using *assms* **by** (*auto simp*: *less-eq-trace-class.rep-eq*)
then show *?thesis*
by (*auto simp*: *less-eq-trace-class.rep-eq*)
qed

lemma *has-sum-mono-traceclass*:
fixes $f :: 'a \Rightarrow ('b{::}\textit{chilbert-space}, 'b)$ *trace-class*
assumes $(f$ *has-sum* $x)\ A$ **and** $(g$ *has-sum* $y)\ A$
assumes ‹$\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x$›
shows $x \leq y$
using *assms has-sum-mono-neutral-traceclass* **by** *force*

lemma *infsum-mono-traceclass*:
fixes $f :: 'a \Rightarrow ('b{::}\textit{chilbert-space}, 'b)$ *trace-class*
assumes f *summable-on* A **and** g *summable-on* A
assumes ‹$\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x$›
shows *infsum* $f\ A \leq$ *infsum* $g\ A$
by (*meson assms has-sum-infsum has-sum-mono-traceclass*)

lemma *infsum-mono-neutral-traceclass*:

fixes $f :: 'a \Rightarrow ('b::chilbert\text{-}space, 'b)$ *trace-class*
assumes *f summable-on A* **and** *g summable-on B*
assumes $\langle\bigwedge x.\ x \in A \cap B \Longrightarrow f\ x \leq g\ x\rangle$
assumes $\langle\bigwedge x.\ x \in A - B \Longrightarrow f\ x \leq 0\rangle$
assumes $\langle\bigwedge x.\ x \in B - A \Longrightarrow g\ x \geq 0\rangle$
shows *infsum f A* $\leq$ *infsum g B*
using *assms(1) assms(2) assms(3) assms(4) assms(5) has-sum-mono-neutral-traceclass summable-iff-has-sum-infsu*
by *blast*

instance *trace-class* :: (*chilbert-space*, *chilbert-space*) *ordered-complex-vector*
apply (*intro-classes*; *transfer*)
by (*auto intro*!: *scaleC-left-mono scaleC-right-mono*)

lemma *Abs-trace-class-geq0I*: $\langle 0 \leq$ *Abs-trace-class t*$\rangle$ **if** $\langle$*trace-class t*$\rangle$ **and** $\langle t \geq 0\rangle$
using *that* **by** (*simp add*: *zero-trace-class.abs-eq less-eq-trace-class.abs-eq eq-onp-def*)

lift-definition *tc-compose* :: $\langle('b::chilbert\text{-}space, 'c::chilbert\text{-}space)$ *trace-class*
$\Rightarrow$ $('a::chilbert\text{-}space, 'b)$ *trace-class* $\Rightarrow$ $('a,'c)$ *trace-class*$\rangle$ **is**
cblinfun-compose
by (*simp add*: *trace-class-comp-left*)

lemma *norm-tc-compose*:
$\langle$*norm* (*tc-compose a b*) $\leq$ *norm a* $*$ *norm b*$\rangle$
proof *transfer*
fix $a :: \langle 'c \Rightarrow_{CL} 'b\rangle$ **and** $b :: \langle 'a \Rightarrow_{CL} 'c\rangle$
assume $\langle a \in Collect\ trace\text{-}class\rangle$ **and** *tc-b*: $\langle b \in Collect\ trace\text{-}class\rangle$
then have $\langle$*trace-norm* (*a* o_{CL} *b*) $\leq$ *trace-norm a* $*$ *norm b*$\rangle$
by (*simp add*: *trace-norm-comp-left*)
also have $\langle \ldots \leq$ *trace-norm a* $*$ *trace-norm b*$\rangle$
using *tc-b* **by** (*auto intro*!: *mult-left-mono norm-leq-trace-norm*)
finally show $\langle$*trace-norm* (*a* o_{CL} *b*) $\leq$ *trace-norm a* $*$ *trace-norm b*$\rangle$
by −
qed

lift-definition *trace-tc* :: $\langle('a::chilbert\text{-}space, 'a)$ *trace-class* $\Rightarrow$ *complex*$\rangle$ **is** *trace*.

lemma *trace-tc-plus*: $\langle$*trace-tc* (*a* + *b*) = *trace-tc a* + *trace-tc b*$\rangle$
apply *transfer* **by** (*simp add*: *trace-plus*)

lemma *trace-tc-scaleC*: $\langle$*trace-tc* (*c* $*_C$ *a*) = *c* $*_C$ *trace-tc a*$\rangle$
apply *transfer* **by** (*simp add*: *trace-scaleC*)

lemma *trace-tc-norm*: $\langle$*norm* (*trace-tc a*) $\leq$ *norm a*$\rangle$
apply *transfer* **by** *auto*

lemma *bounded-clinear-trace-tc*[*bounded-clinear*, *simp*]: $\langle$*bounded-clinear trace-tc*$\rangle$
apply (*rule bounded-clinearI*[**where** *K*=*1*])
by (*auto simp*: *trace-tc-scaleC trace-tc-plus intro*!: *trace-tc-norm*)

lemma *norm-tc-pos*: ‹*norm A = trace-tc A*› **if** ‹*A ≥ 0*›
  using *that* **apply** *transfer* **by** (*simp add*: *trace-norm-pos*)

lemma *norm-tc-pos-Re*: ‹*norm A = Re* (*trace-tc A*)› **if** ‹*A ≥ 0*›
 using *norm-tc-pos*[*OF that*]
 by (*metis Re-complex-of-real*)

lemma *from-trace-class-pos*: ‹*from-trace-class A ≥ 0 ⟷ A ≥ 0*›
 by (*simp add*: *less-eq-trace-class.rep-eq*)

lemma *infsum-tc-norm-bounded-abs-summable*:
 fixes *A* :: ‹*'a ⇒* (*'b*::*chilbert-space*, *'b*::*chilbert-space*) *trace-class*›
 assumes *pos*: ‹⋀*x*. *x ∈ M ⟹ A x ≥ 0*›
 assumes *bound-B*: ‹⋀*F*. *finite F ⟹ F ⊆ M ⟹ norm* (∑ *x*∈*F*. *A x*) *≤ B*›
 shows ‹*A abs-summable-on M*›
proof −
 have ‹(∑ *x*∈*F*. *norm* (*A x*)) = *norm* (∑ *x*∈*F*. *A x*)› **if** ‹*F ⊆ M*› **for** *F*
 proof −
  have ‹*complex-of-real* (∑ *x*∈*F*. *norm* (*A x*)) = (∑ *x*∈*F*. *complex-of-real* (*trace-norm* (*from-trace-class* (*A x*))))›
    by (*simp add*: *norm-trace-class.rep-eq trace-norm-pos*)
   also have ‹*. . .* = (∑ *x*∈*F*. *trace* (*from-trace-class* (*A x*)))›
    using *that pos* **by** (*auto intro*!: *sum.cong simp add*: *trace-norm-pos less-eq-trace-class.rep-eq*)
   also have ‹*. . .* = *trace* (*from-trace-class* (∑ *x*∈*F*. *A x*))›
    by (*simp add*: *from-trace-class-sum trace-sum*)
   also have ‹*. . .* = *norm* (∑ *x*∈*F*. *A x*)›
    by (*smt* (*verit*, *ccfv-threshold*) *calculation norm-of-real norm-trace-class.rep-eq sum-norm-le trace-leq-trace-norm*)
   finally show *?thesis*
    using *of-real-eq-iff* **by** *blast*
 qed
 with *bound-B* **have** *bound-B'*: ‹(∑ *x*∈*F*. *norm* (*A x*)) *≤ B*› **if** ‹*finite F*› **and** ‹*F ⊆ M*› **for** *F*
  by (*metis that*(*1*) *that*(*2*))
 then show ‹*A abs-summable-on M*›
  apply (*rule-tac nonneg-bdd-above-summable-on*)
  by (*auto intro*!: *bdd-aboveI*)
qed

lemma *trace-norm-uminus*[*simp*]: ‹*trace-norm* (−*a*) = *trace-norm a*›
 by (*metis abs-op-uminus of-real-eq-iff trace-abs-op*)

lemma *trace-norm-triangle-minus*:
 fixes *a b* :: ‹*'a*::*chilbert-space ⇒*$_{CL}$ *'b*::*chilbert-space*›
 assumes [*simp*]: ‹*trace-class a*› ‹*trace-class b*›
 shows ‹*trace-norm* (*a* − *b*) ≤ *trace-norm a* + *trace-norm b*›
 using *trace-norm-triangle*[*of a* ‹−*b*›]
 by *auto*

lemma *trace-norm-abs-op*[*simp*]: ‹*trace-norm* (*abs-op t*) = *trace-norm t*›
by (*simp add*: *trace-norm-def*)

lemma
fixes *t* :: ‹'*a* ⇒$_{CL}$ '*a*::*chilbert-space*›
shows *cblinfun-decomp-4pos*: ‹
∃ *t1 t2 t3 t4*.
t = *t1* − *t2* + i ∗$_C$ *t3* − i ∗$_C$ *t4*
∧ *t1* ≥ *0* ∧ *t2* ≥ *0* ∧ *t3* ≥ *0* ∧ *t4* ≥ *0*› (**is** *?thesis1*)
and *trace-class-decomp-4pos*: ‹*trace-class t* ⟹
∃ *t1 t2 t3 t4*.
t = *t1* − *t2* + i ∗$_C$ *t3* − i ∗$_C$ *t4*
∧ *trace-class t1* ∧ *trace-class t2* ∧ *trace-class t3* ∧ *trace-class t4*
∧ *trace-norm t1* ≤ *trace-norm t* ∧ *trace-norm t2* ≤ *trace-norm t* ∧ *trace-norm t3* ≤ *trace-norm t* ∧ *trace-norm t4* ≤ *trace-norm t*
∧ *t1* ≥ *0* ∧ *t2* ≥ *0* ∧ *t3* ≥ *0* ∧ *t4* ≥ *0*› (**is** ‹- ⟹ *?thesis2*›)
proof −
define *th ta* **where** ‹*th* = (*1*/*2*) ∗$_C$ (*t* + *t*∗)› **and** ‹*ta* = (−i/*2*) ∗$_C$ (*t* − *t*∗)›
have *th-herm*: ‹*th*∗ = *th*›
by (*simp add*: *adj-plus th-def*)
have ‹*ta*∗ = *ta*›
by (*simp add*: *adj-minus ta-def scaleC-diff-right adj-uminus*)
have ‹*t* = *th* + i ∗$_C$ *ta*›
by (*smt* (*verit*, *ccfv-SIG*) *add.commute add.inverse-inverse complex-i-mult-minus complex-vector.vector-space-assms*(*1*) *complex-vector.vector-space-assms*(*3*) *diff-add-cancel group-cancel.add2 i-squared scaleC-half-double ta-def th-def times-divide-eq-right*)
define *t1 t2* **where** ‹*t1* = (*abs-op th* + *th*) /$_R$ *2*› **and** ‹*t2* = (*abs-op th* − *th*) /$_R$ *2*›
have ‹*t1* ≥ *0*›
using *abs-op-geq-neq*[*unfolded selfadjoint-def*, *OF* ‹*th*∗ = *th*›] *ordered-field-class.sign-simps*(*15*)
by (*fastforce simp add*: *t1-def intro*!: *scaleR-nonneg-nonneg*)
have ‹*t2* ≥ *0*›
using *abs-op-geq*[*unfolded selfadjoint-def*, *OF* ‹*th*∗ = *th*›] *ordered-field-class.sign-simps*(*15*)
by (*fastforce simp add*: *t2-def intro*!: *scaleR-nonneg-nonneg*)
have ‹*th* = *t1* − *t2*›
apply (*simp add*: *t1-def t2-def*)
by (*metis* (*no-types*, *opaque-lifting*) *Extra-Ordered-Fields.sign-simps*(*8*) *diff-add-cancel ordered-field-class.sign-simps*(*2*) *ordered-field-class.sign-simps*(*27*) *scaleR-half-double*)
define *t3 t4* **where** ‹*t3* = (*abs-op ta* + *ta*) /$_R$ *2*› **and** ‹*t4* = (*abs-op ta* − *ta*) /$_R$ *2*›
have ‹*t3* ≥ *0*›
using *abs-op-geq-neq*[*unfolded selfadjoint-def*, *OF* ‹*ta*∗ = *ta*›] *ordered-field-class.sign-simps*(*15*)
by (*fastforce simp add*: *t3-def intro*!: *scaleR-nonneg-nonneg*)
have ‹*t4* ≥ *0*›
using *abs-op-geq*[*unfolded selfadjoint-def*, *OF* ‹*ta*∗ = *ta*›] *ordered-field-class.sign-simps*(*15*)
by (*fastforce simp add*: *t4-def intro*!: *scaleR-nonneg-nonneg*)
have ‹*ta* = *t3* − *t4*›
apply (*simp add*: *t3-def t4-def*)
by (*metis* (*no-types*, *opaque-lifting*) *Extra-Ordered-Fields.sign-simps*(*8*) *diff-add-cancel ordered-field-class.sign-simps*(*2*) *ordered-field-class.sign-simps*(*27*) *scaleR-half-double*)

```
  have decomp: ‹t = t1 − t2 + i *C t3 − i *C t4›
    by (simp add: ‹t = th + i *C ta› ‹th = t1 − t2› ‹ta = t3 − t4› scaleC-diff-right)
  from decomp ‹t1 ≥ 0› ‹t2 ≥ 0› ‹t3 ≥ 0› ‹t4 ≥ 0›
  show ?thesis1
    by auto
  show ?thesis2 if ‹trace-class t›
  proof −
    have ‹trace-class th› ‹trace-class ta›
      by (auto simp add: th-def ta-def
       intro!: ‹trace-class t› trace-class-scaleC trace-class-plus trace-class-minus trace-class-uminus
trace-class-adj)
    then have tc: ‹trace-class t1› ‹trace-class t2› ‹trace-class t3› ‹trace-class t4›
      by (auto simp add: t1-def t2-def t3-def t4-def scaleR-scaleC intro!: trace-class-scaleC)
    have tn-th: ‹trace-norm th ≤ trace-norm t›
      using trace-norm-triangle[of t ‹t*›]
      by (auto simp add: that th-def trace-norm-scaleC)
    have tn-ta: ‹trace-norm ta ≤ trace-norm t›
      using trace-norm-triangle-minus[of t ‹t*›]
      by (auto simp add: that ta-def trace-norm-scaleC)
    have tn1: ‹trace-norm t1 ≤ trace-norm t›
      using trace-norm-triangle[of ‹abs-op th› th] tn-th
      by (auto simp add: ‹trace-class th› t1-def trace-norm-scaleC scaleR-scaleC)
    have tn2: ‹trace-norm t2 ≤ trace-norm t›
      using trace-norm-triangle-minus[of ‹abs-op th› th] tn-th
      by (auto simp add: ‹trace-class th› t2-def trace-norm-scaleC scaleR-scaleC)
    have tn3: ‹trace-norm t3 ≤ trace-norm t›
      using trace-norm-triangle[of ‹abs-op ta› ta] tn-ta
      by (auto simp add: ‹trace-class ta› t3-def trace-norm-scaleC scaleR-scaleC)
    have tn4: ‹trace-norm t4 ≤ trace-norm t›
      using trace-norm-triangle-minus[of ‹abs-op ta› ta] tn-ta
      by (auto simp add: ‹trace-class ta› t4-def trace-norm-scaleC scaleR-scaleC)
    from decomp tc ‹t1 ≥ 0› ‹t2 ≥ 0› ‹t3 ≥ 0› ‹t4 ≥ 0› tn1 tn2 tn3 tn4
    show ?thesis2
      by auto
  qed
qed

lemma trace-class-decomp-4pos':
  fixes t :: ‹('a::chilbert-space,'a) trace-class›
  shows ‹∃ t1 t2 t3 t4 .
            t = t1 − t2 + i *C t3 − i *C t4
            ∧ norm t1 ≤ norm t ∧ norm t2 ≤ norm t ∧ norm t3 ≤ norm t ∧ norm t4 ≤ norm
t
             ∧ t1 ≥ 0 ∧ t2 ≥ 0 ∧ t3 ≥ 0 ∧ t4 ≥ 0›
proof −
  from trace-class-decomp-4pos[of ‹from-trace-class t›, OF trace-class-from-trace-class]
  obtain t1' t2' t3' t4'
    where *: ‹from-trace-class t = t1' − t2' + i *C t3' − i *C t4'
             ∧ trace-class t1' ∧ trace-class t2' ∧ trace-class t3' ∧ trace-class t4'
```

$\wedge$ *trace-norm t1'* $\leq$ *trace-norm* (*from-trace-class t*) $\wedge$ *trace-norm t2'* $\leq$ *trace-norm* (*from-trace-class t*) $\wedge$ *trace-norm t3'* $\leq$ *trace-norm* (*from-trace-class t*) $\wedge$ *trace-norm t4'* $\leq$ *trace-norm* (*from-trace-class t*)
$\wedge$ *t1'* $\geq$ *0* $\wedge$ *t2'* $\geq$ *0* $\wedge$ *t3'* $\geq$ *0* $\wedge$ *t4'* $\geq$ *0*›
by *auto*
then obtain *t1 t2 t3 t4* **where**
t1234: ‹*t1'* = *from-trace-class t1*› ‹*t2'* = *from-trace-class t2*› ‹*t3'* = *from-trace-class t3*› ‹*t4'* = *from-trace-class t4*›
by (*metis from-trace-class-cases mem-Collect-eq*)
with ∗ **have** *n1234*: ‹*norm t1* $\leq$ *norm t*› ‹*norm t2* $\leq$ *norm t*› ‹*norm t3* $\leq$ *norm t*› ‹*norm t4* $\leq$ *norm t*›
by (*metis norm-trace-class.rep-eq*)+
have *t-decomp*: ‹*t* = *t1* − *t2* + i ∗$_C$ *t3* − i ∗$_C$ *t4*›
using ∗ **unfolding** *t1234*
by (*auto simp*: *from-trace-class-inject*
simp flip: *scaleC-trace-class.rep-eq plus-trace-class.rep-eq minus-trace-class.rep-eq*)
have *pos1234*: ‹*t1* $\geq$ *0*› ‹*t2* $\geq$ *0*› ‹*t3* $\geq$ *0*› ‹*t4* $\geq$ *0*›
using ∗ **unfolding** *t1234*
by (*auto simp*: *less-eq-trace-class-def*)
from *t-decomp pos1234 n1234*
show *?thesis*
by *blast*
qed

thm *bounded-clinear-trace-duality*
lemma *bounded-clinear-trace-duality'*: ‹*trace-class t* $\Longrightarrow$ *bounded-clinear* (λa. *trace* (*a* o_{CL} *t*))›
for *t* :: ‹-::*chilbert-space* $\Rightarrow_{CL}$ -::*chilbert-space*›
apply (*rule bounded-clinearI*[**where** *K*=‹*trace-norm t*›])
apply (*auto simp add*: *cblinfun-compose-add-left trace-class-comp-right trace-plus trace-scaleC*)[*2*]
by (*metis circularity-of-trace order-trans trace-leq-trace-norm trace-norm-comp-right*)

lemma *infsum-nonneg-traceclass*:
fixes *f* :: *'a* $\Rightarrow$ (*'b*::*chilbert-space*, *'b*) *trace-class*
assumes $\bigwedge x$. *x* $\in$ *M* $\Longrightarrow$ *0* $\leq$ *f x*
shows *infsum f M* $\geq$ *0*
apply (*cases* ‹*f summable-on M*›)
apply (*subst infsum-0-simp*[*symmetric*])
apply (*rule infsum-mono-neutral-traceclass*)
using *assms* **by** (*auto simp*: *infsum-not-exists*)

lemma *sandwich-tc-compose*: ‹*sandwich-tc* (*A* o_{CL} *B*) = *sandwich-tc A o sandwich-tc B*›
apply (*rule ext*)
apply (*rule from-trace-class-inject*[*THEN iffD1*])
apply (*transfer fixing*: *A B*)
by (*simp add*: *sandwich-compose*)

lemma *sandwich-tc-0-left*[*simp*]: ‹*sandwich-tc 0* = *0*›
by (*auto intro*!: *ext simp add*: *sandwich-tc-def compose-tcl.zero-left compose-tcr.zero-left*)

lemma *sandwich-tc-0-right*[*simp*]: ‹*sandwich-tc e 0 = 0*›
by (*auto intro*!: *ext simp add*: *sandwich-tc-def compose-tcl.zero-left compose-tcr.zero-right*)

lemma *sandwich-tc-scaleC-left*: ‹*sandwich-tc* (*c* $*_C$ *e*) *t* = (*cmod c*)^2 $*_C$ *sandwich-tc e t*›
apply (*rule from-trace-class-inject*[*THEN iffD1*])
by (*simp add*: *from-trace-class-sandwich-tc scaleC-trace-class.rep-eq sandwich-scaleC-left*)

lemma *sandwich-tc-scaleR-left*: ‹*sandwich-tc* (*r* $*_R$ *e*) *t* = *r*^2 $*_R$ *sandwich-tc e t*›
by (*simp add*: *scaleR-scaleC sandwich-tc-scaleC-left flip*: *of-real-power*)

lemma *bounded-cbilinear-tc-compose*: ‹*bounded-cbilinear tc-compose*›
unfolding *bounded-cbilinear-def*
apply *transfer*
apply (*auto intro*!: *exI*[*of - 1*] *simp*: *cblinfun-compose-add-left cblinfun-compose-add-right*)
by (*meson norm-leq-trace-norm dual-order.trans mult-right-mono trace-norm-comp-right trace-norm-nneg*)
lemmas *bounded-clinear-tc-compose-left*[*bounded-clinear*] = *bounded-cbilinear.bounded-clinear-left*[*OF bounded-cbilinear-tc-compose*]
lemmas *bounded-clinear-tc-compose-right*[*bounded-clinear*] = *bounded-cbilinear.bounded-clinear-right*[*OF bounded-cbilinear-tc-compose*]

lift-definition *tc-butterfly* :: ‹'*a*::*chilbert-space* ⇒ '*b*::*chilbert-space* ⇒ ('*b*,'*a*) *trace-class*›
is *butterfly*
by *simp*

lemma *norm-tc-butterfly*: ‹*norm* (*tc-butterfly* ψ φ) = *norm* ψ * *norm* φ›
apply (*transfer fixing*: ψ φ)
by (*simp add*: *trace-norm-butterfly*)

lemma *comp-tc-butterfly*[*simp*]: ‹*tc-compose* (*tc-butterfly a b*) (*tc-butterfly c d*) = (*b* $\cdot_C$ *c*) $*_C$ *tc-butterfly a d*›
apply *transfer'*
by *simp*

lemma *tc-butterfly-pos*[*simp*]: ‹*0* ≤ *tc-butterfly* ψ ψ›
apply *transfer*
by *simp*

lift-definition *rank1-tc* :: ‹('*a*::*chilbert-space*, '*b*::*chilbert-space*) *trace-class* ⇒ *bool*› **is** *rank1*.
lift-definition *finite-rank-tc* :: ‹('*a*::*chilbert-space*, '*b*::*chilbert-space*) *trace-class* ⇒ *bool*› **is** *finite-rank*.

lemma *finite-rank-tc-0*[*iff*]: ‹*finite-rank-tc 0*›
apply *transfer* **by** *simp*

lemma *finite-rank-tc-plus*: ‹*finite-rank-tc* (*a* + *b*)›
if ‹*finite-rank-tc a*› **and** ‹*finite-rank-tc b*›
using *that* **apply** *transfer*
by *simp*

lemma *finite-rank-tc-scale*: ‹*finite-rank-tc* (*c* *$_C$ *a*)› **if** ‹*finite-rank-tc a*›
using *that* **apply** *transfer* **by** *simp*

lemma *csubspace-finite-rank-tc*: ‹*csubspace* (*Collect finite-rank-tc*)›
apply (*rule complex-vector.subspaceI*)
by (*auto intro*!: *finite-rank-tc-plus finite-rank-tc-scale*)

lemma *rank1-trace-class*: ‹*trace-class a*› **if** ‹*rank1 a*›
for *a b* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
using *that* **by** (*auto intro*!: *simp*: *rank1-iff-butterfly*)

lemma *finite-rank-trace-class*: ‹*trace-class a*› **if** ‹*finite-rank a*›
for *a* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
proof −
from ‹*finite-rank a*› **obtain** *F f* **where** ‹*finite F*› **and** ‹*F* ⊆ *Collect rank1*›
and *a-def*: ‹*a* = (∑ *x*∈*F*. *f x* *$_C$ *x*)›
by (*smt* (*verit*, *ccfv-threshold*) *complex-vector.span-explicit finite-rank-def mem-Collect-eq*)
then show ‹*trace-class a*›
unfolding *a-def*
apply *induction*
by (*auto intro*!: *trace-class-plus trace-class-scaleC intro*: *rank1-trace-class*)
qed

lemma *trace-minus*:
assumes ‹*trace-class a*› ‹*trace-class b*›
shows ‹*trace* (*a* − *b*) = *trace a* − *trace b*›
by (*metis* (*no-types*, *lifting*) *add-uminus-conv-diff assms*(*1*) *assms*(*2*) *trace-class-uminus trace-plus trace-uminus*)

lemma *trace-cblinfun-mono*:
fixes *A B* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*a*›
assumes ‹*trace-class A*› **and** ‹*trace-class B*›
assumes ‹*A* ≤ *B*›
shows ‹*trace A* ≤ *trace B*›
proof −
have *sumA*: ‹(λ*e*. *e* ∙$_C$ (*A* *$_V$ *e*)) *summable-on some-chilbert-basis*›
by (*auto intro*!: *trace-exists assms*)
moreover have *sumB*: ‹(λ*e*. *e* ∙$_C$ (*B* *$_V$ *e*)) *summable-on some-chilbert-basis*›
by (*auto intro*!: *trace-exists assms*)
moreover have ‹*x* ∙$_C$ (*A* *$_V$ *x*) ≤ *x* ∙$_C$ (*B* *$_V$ *x*)› **for** *x*
using *assms*(*3*) *less-eq-cblinfun-def* **by** *blast*
ultimately have ‹(∑$_\infty$*e*∈*some-chilbert-basis*. *e* ∙$_C$ (*A* *$_V$ *e*)) ≤ (∑$_\infty$*e*∈*some-chilbert-basis*. *e* ∙$_C$ (*B* *$_V$ *e*))›
by (*rule infsum-mono-complex*)
then show *?thesis*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *diff-ge-0-iff-ge trace-minus trace-pos*)
qed

lemma *trace-tc-mono*:
assumes ‹$A \leq B$›
shows ‹*trace-tc* $A \leq$ *trace-tc* B›
using *assms*
apply *transfer*
by (*simp add*: *trace-cblinfun-mono*)

lemma *trace-tc-0*[*simp*]: ‹*trace-tc* $0 = 0$›
apply *transfer'* **by** *simp*

lemma *cspan-tc-transfer*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-set cr-trace-class* ===> *rel-set cr-trace-class*) *cspan cspan*›
proof (*intro rel-funI rel-setI*)
fix B :: ‹('a ⇒$_{CL}$ 'b) *set*› **and** C
assume ‹*rel-set cr-trace-class* B C›
then have BC: ‹B = *from-trace-class* ‘ C›
by (*auto intro*!: *simp*: *cr-trace-class-def image-iff rel-set-def*)

show ‹∃ t∈*cspan* C. *cr-trace-class* a t› **if** ‹$a \in$ *cspan* B› **for** a
proof −
from *that* **obtain** F f **where** ‹*finite* F› **and** ‹$F \subseteq B$› **and** *a-sum*: ‹$a = (\sum x{\in}F.\ f\ x *_C\ x)$›
by (*auto simp*: *complex-vector.span-explicit*)
from ‹$F \subseteq B$›
obtain F' **where** ‹$F' \subseteq C$› **and** FF': ‹F = *from-trace-class* ‘ F'›
by (*auto elim*!: *subset-imageE simp*: BC)
define t **where** ‹$t = (\sum x{\in}F'.\ f$ (*from-trace-class* x) $*_C\ x)$›
have ‹a = *from-trace-class* t›
by (*simp add*: *a-sum t-def from-trace-class-sum scaleC-trace-class.rep-eq* FF'
sum.reindex o-def from-trace-class-inject inj-on-def)
moreover have ‹$t \in$ *cspan* C›
by (*metis* (*no-types*, *lifting*) ‹$F' \subseteq C$› *complex-vector.span-clauses*(*4*) *complex-vector.span-sum*
complex-vector.span-superset subsetD t-def)
ultimately show ‹∃ t∈*cspan* C. *cr-trace-class* a t›
by (*auto simp*: *cr-trace-class-def*)
qed

show ‹∃ a∈*cspan* B. *cr-trace-class* a t› **if** ‹$t \in$ *cspan* C› **for** t
proof −
from *that* **obtain** F f **where** ‹*finite* F› **and** ‹$F \subseteq C$› **and** *t-sum*: ‹$t = (\sum x{\in}F.\ f\ x *_C\ x)$›
by (*auto simp*: *complex-vector.span-explicit*)
define a **where** ‹$a = (\sum x{\in}F.\ f\ x *_C$ *from-trace-class* $x)$›
then have ‹a = *from-trace-class* t›
by (*simp add*: *t-sum a-def from-trace-class-sum scaleC-trace-class.rep-eq*)
moreover have ‹$a \in$ *cspan* B›
using BC ‹$F \subseteq C$›
by (*auto intro*!: *complex-vector.span-base complex-vector.span-sum complex-vector.span-scale*
simp: *a-def*)
ultimately show *?thesis*

```
    by (auto simp: cr-trace-class-def)
  qed
qed

lemma finite-rank-tc-def': ‹finite-rank-tc A ⟷ A ∈ cspan (Collect rank1-tc)›
  apply transfer'
  apply (auto simp: finite-rank-def)
  apply (metis (no-types, lifting) Collect-cong rank1-trace-class)
  by (metis (no-types, lifting) Collect-cong rank1-trace-class)


lemma tc-butterfly-add-left: ‹tc-butterfly (a + a') b = tc-butterfly a b + tc-butterfly a' b›
  apply transfer
  by (rule butterfly-add-left)

lemma tc-butterfly-add-right: ‹tc-butterfly a (b + b') = tc-butterfly a b + tc-butterfly a b'›
  apply transfer
  by (rule butterfly-add-right)

lemma tc-butterfly-sum-left: ‹tc-butterfly (∑ i∈M. ψ i) φ = (∑ i∈M. tc-butterfly (ψ i) φ)›
  apply transfer
  by (rule butterfly-sum-left)

lemma tc-butterfly-sum-right: ‹tc-butterfly ψ (∑ i∈M. φ i) = (∑ i∈M. tc-butterfly ψ (φ i))›
  apply transfer
  by (rule butterfly-sum-right)

lemma tc-butterfly-scaleC-left[simp]: tc-butterfly (c *C ψ) φ = c *C tc-butterfly ψ φ
  apply transfer by simp

lemma tc-butterfly-scaleC-right[simp]: tc-butterfly ψ (c *C φ) = cnj c *C tc-butterfly ψ φ
  apply transfer by simp

lemma bounded-sesquilinear-tc-butterfly[iff]: ‹bounded-sesquilinear (λa b. tc-butterfly b a)›
  by (auto intro!: bounded-sesquilinear.intro exI[of - 1]
      simp: tc-butterfly-add-left tc-butterfly-add-right norm-tc-butterfly)


lemma trace-norm-plus-orthogonal:
  assumes ‹trace-class a› and ‹trace-class b›
  assumes ‹a* oCL b = 0› and ‹a oCL b* = 0›
  shows ‹trace-norm (a + b) = trace-norm a + trace-norm b›
proof −
  have ‹trace-norm (a + b) = trace (abs-op (a + b))›
    by simp
  also have ‹... = trace (abs-op a + abs-op b)›
   by (simp add: abs-op-plus-orthogonal assms)
  also have ‹... = trace (abs-op a) + trace (abs-op b)›
    by (simp add: assms trace-plus)
```

also have ‹... = *trace-norm a* + *trace-norm b*›
by *simp*
finally show *?thesis*
using *of-real-eq-iff* **by** *blast*
qed

lemma *norm-tc-plus-orthogonal*:
assumes ‹*tc-compose* (*adj-tc a*) *b* = *0*› **and** ‹*tc-compose a* (*adj-tc b*) = *0*›
shows ‹*norm* (*a* + *b*) = *norm a* + *norm b*›
using *assms* **apply** *transfer*
by (*auto intro*!: *trace-norm-plus-orthogonal*)

lemma *trace-norm-sum-exchange*:
fixes *t* :: ‹- ⇒ (-::*chilbert-space* ⇒$_{CL}$ -::*chilbert-space*)›
assumes ‹⋀*i*. *i* ∈ *F* ⟹ *trace-class* (*t i*)›
assumes ‹⋀*i j*. *i* ∈ *F* ⟹ *j* ∈ *F* ⟹ *i* ≠ *j* ⟹ (*t i*)* *o*$_{CL}$ *t j* = *0*›
assumes ‹⋀*i j*. *i* ∈ *F* ⟹ *j* ∈ *F* ⟹ *i* ≠ *j* ⟹ *t i o*$_{CL}$ (*t j*)* = *0*›
shows ‹*trace-norm* (∑ *i*∈*F*. *t i*) = (∑ *i*∈*F*. *trace-norm* (*t i*))›
proof (*insert assms*, *induction F rule*:*infinite-finite-induct*)
case (*infinite A*)
then show *?case*
by *simp*
next
case *empty*
show *?case*
by *simp*
next
case (*insert x F*)
have ‹*trace-norm* (∑ *i*∈*insert x F*. *t i*) = *trace-norm* (*t x* + (∑ *x*∈*F*. *t x*))›
by (*simp add*: *insert*)
also have ‹... = *trace-norm* (*t x*) + *trace-norm* (∑ *x*∈*F*. *t x*)›
proof (*rule trace-norm-plus-orthogonal*)
show ‹*trace-class* (*t x*)›
by (*simp add*: *insert.prems*)
show ‹*trace-class* (∑ *x*∈*F*. *t x*)›
by (*simp add*: *trace-class-sum insert.prems*)
show ‹*t x** *o*$_{CL}$ (∑ *x*∈*F*. *t x*) = *0*›
by (*auto intro*!: *sum.neutral insert.prems simp*: *cblinfun-compose-sum-right sum-adj insert.hyps*)
show ‹*t x o*$_{CL}$ (∑ *x*∈*F*. *t x*)* = *0*›
by (*auto intro*!: *sum.neutral insert.prems simp*: *cblinfun-compose-sum-right sum-adj insert.hyps*)
qed
also have ‹... = *trace-norm* (*t x*) + (∑ *x*∈*F*. *trace-norm* (*t x*))›
apply (*subst insert.IH*)
by (*simp-all add*: *insert.prems*)
also have ‹... = (∑ *i*∈*insert x F*. *trace-norm* (*t i*))›
by (*simp add*: *insert*)

```
  finally show ?case
    by −
qed

lemma norm-tc-sum-exchange:
  assumes ‹⋀i j. i ∈ F ⟹ j ∈ F ⟹ i ≠ j ⟹ tc-compose (adj-tc (t i)) (t j) = 0›
  assumes ‹⋀i j. i ∈ F ⟹ j ∈ F ⟹ i ≠ j ⟹ tc-compose (t i) (adj-tc (t j)) = 0›
  shows ‹norm (∑ i∈F. t i) = (∑ i∈F. norm (t i))›
  using assms apply transfer
  by (auto intro!: trace-norm-sum-exchange)


instantiation trace-class :: (one-dim, one-dim) complex-inner begin
lift-definition cinner-trace-class :: ‹('a, 'b) trace-class ⇒ ('a, 'b) trace-class ⇒ complex› is
‹(·C)›.
instance
proof intro-classes
  fix x y z :: ‹('a, 'b) trace-class›
  show ‹x ·C y = cnj (y ·C x)›
    apply transfer'
    by simp
  show ‹(x + y) ·C z = x ·C z + y ·C z›
    apply transfer'
    by (simp add: cinner-simps)
  show ‹r *C x ·C y = cnj r * (x ·C y)› for r
    apply (transfer' fixing: r)
    using cinner-simps by blast
  show ‹0 ≤ x ·C x›
    apply transfer'
    by simp
  show ‹(x ·C x = 0) = (x = 0)›
    apply transfer'
    by auto
  show ‹norm x = sqrt (cmod (x ·C x))›
  proof transfer'
    fix x :: ‹'a ⇒CL 'b›
    define c :: complex where ‹c = one-dim-iso x›
    then have xc: ‹x = c *C 1›
      by simp
    have ‹trace-norm x = norm c›
      by (simp add: trace-norm-one-dim xc)
    also have ‹norm c = sqrt (cmod (x ·C x))›
      by (metis inner-real-def norm-eq-sqrt-cinner norm-one norm-scaleC real-inner-1-right xc)
    finally show ‹trace-norm x = sqrt (cmod (x ·C x)) ›
      by (simp add: cinner-cblinfun-def)
  qed
qed
end
```

instantiation *trace-class* :: (*one-dim*, *one-dim*) *one-dim* **begin**
lift-definition *one-trace-class* :: ‹('*a*, '*b*) *trace-class*› **is** *1*
by *auto*
lift-definition *times-trace-class* :: ‹('*a*, '*b*) *trace-class* ⇒ ('*a*, '*b*) *trace-class* ⇒ ('*a*, '*b*) *trace-class*›
is ‹(∗)›
by *auto*
lift-definition *divide-trace-class* :: ‹('*a*, '*b*) *trace-class* ⇒ ('*a*, '*b*) *trace-class* ⇒ ('*a*, '*b*) *trace-class*›
is ‹(/)›
by *auto*
lift-definition *inverse-trace-class* :: ‹('*a*, '*b*) *trace-class* ⇒ ('*a*, '*b*) *trace-class*› **is** ‹*Fields.inverse*›
by *auto*
definition *canonical-basis-trace-class* :: ‹('*a*, '*b*) *trace-class list*› **where** ‹*canonical-basis-trace-class*
= [*1*]›
definition *canonical-basis-length-trace-class* :: ‹('*a*, '*b*) *trace-class itself* ⇒ *nat*› **where** ‹*canonical-basis-length-trace-class* = *1* ›
instance
proof *intro-classes*
fix *x y z* :: ‹('*a*, '*b*) *trace-class*›
have [*simp*]: ‹*1* ≠ (*0* :: ('*a*, '*b*) *trace-class*)›
using *one-trace-class.rep-eq* **by** *force*
then have [*simp*]: ‹*0* ≠ (*1* :: ('*a*, '*b*) *trace-class*)›
by *force*
show ‹*distinct* (*canonical-basis* :: (-,-) *trace-class list*)›
by (*simp add*: *canonical-basis-trace-class-def*)
show ‹*cindependent* (*set canonical-basis* :: (-,-) *trace-class set*)›
by (*simp add*: *canonical-basis-trace-class-def*)
show ‹*canonical-basis-length TYPE*(('*a*, '*b*) *trace-class*) = *length* (*canonical-basis* :: (-,-) *trace-class*
list)›
by (*simp add*: *canonical-basis-length-trace-class-def canonical-basis-trace-class-def*)
show ‹*x* ∈ *set canonical-basis* ⟹ *norm x* = *1*›
apply (*simp add*: *canonical-basis-trace-class-def*)
by (*smt* (*verit*, *ccfv-threshold*) *one-trace-class-def cinner-trace-class.abs-eq cnorm-eq-1 one-cinner-one*
one-trace-class.rsp)
show ‹*canonical-basis* = [*1* :: ('*a*,'*b*) *trace-class*]›
by (*simp add*: *canonical-basis-trace-class-def*)
show ‹*a* ∗$_C$ *1* ∗ *b* ∗$_C$ *1* = (*a* ∗ *b*) ∗$_C$ (*1* :: ('*a*,'*b*) *trace-class*)› **for** *a b* :: *complex*
apply (*transfer' fixing*: *a b*)
by *simp*
show ‹*x div y* = *x* ∗ *inverse y*›
apply *transfer'*
by (*simp add*: *divide-cblinfun-def*)
show ‹*inverse* (*a* ∗$_C$ *1* :: ('*a*,'*b*) *trace-class*) = *1* /$_C$ *a*› **for** *a* :: *complex*
apply *transfer'*
by *simp*
show ‹*is-ortho-set* (*set canonical-basis* :: ('*a*,'*b*) *trace-class set*)›
by (*simp add*: *is-ortho-set-def canonical-basis-trace-class-def*)
show ‹*cspan* (*set canonical-basis* :: ('*a*,'*b*) *trace-class set*) = *UNIV*›
proof (*intro Set.set-eqI iffI UNIV-I*)

```
    fix x :: ‹('a,'b) trace-class›
    have ‹∃ c. y = c *C 1› for y :: ‹'a ⇒CL 'b›
      apply (rule exI[where x=‹one-dim-iso y›])
      by simp
    then obtain c where ‹x = c *C 1›
      apply transfer'
      by auto
    then show ‹x ∈ cspan (set canonical-basis)›
      by (auto intro!: complex-vector.span-base complex-vector.span-clauses
          simp: canonical-basis-trace-class-def)
  qed
qed
end

lemma from-trace-class-one-dim-iso[simp]: ‹from-trace-class = one-dim-iso›
proof (rule ext)
  fix x:: ‹('a, 'b) trace-class›
  have ‹from-trace-class x = from-trace-class (one-dim-iso x *C 1)›
    by simp
  also have ‹... = one-dim-iso x *C from-trace-class 1›
    using scaleC-trace-class.rep-eq by blast
  also have ‹... = one-dim-iso x *C 1›
    by (simp add: one-trace-class.rep-eq)
  also have ‹... = one-dim-iso x›
    by simp
  finally show ‹from-trace-class x = one-dim-iso x›
    by -
qed

lemma trace-tc-one-dim-iso[simp]: ‹trace-tc = one-dim-iso›
  by (simp add: trace-tc.rep-eq[abs-def])

lemma compose-tcr-id-left[simp]: ‹compose-tcr id-cblinfun t = t›
  by (auto intro!: from-trace-class-inject[THEN iffD1] simp: compose-tcr.rep-eq)

lemma compose-tcl-id-right[simp]: ‹compose-tcl t id-cblinfun = t›
  by (auto intro!: from-trace-class-inject[THEN iffD1] simp: compose-tcl.rep-eq)


lemma sandwich-tc-id-cblinfun[simp]: ‹sandwich-tc id-cblinfun t = t›
  by (simp add: from-trace-class-inverse sandwich-tc-def)

lemma bounded-clinear-sandwich-tc[bounded-clinear]: ‹bounded-clinear (sandwich-tc e)›
  using norm-sandwich-tc[of e]
  by (auto intro!: bounded-clinearI[where K=‹(norm e)²›]
      simp: sandwich-tc-plus sandwich-tc-scaleC-right cross3-simps)

lemma trace-class-Proj: ‹trace-class (Proj S) ⟷ finite-dim-ccsubspace S›
proof -
```

```
  define C where ‹C = some-onb-of S›
  then obtain B where ‹is-onb B› and ‹C ⊆ B›
    using orthonormal-basis-exists some-onb-of-norm1 by blast
  have card-C: ‹card C = cdim (space-as-set S)›
    by (simp add: C-def some-onb-of-card)
  have S-C: ‹S = ccspan C›
    by (simp add: C-def)

  from ‹is-onb B›
  have ‹trace-class (Proj S) ⟷ ((λx. x •C (abs-op (Proj S) *V x)) abs-summable-on B)›
    by (rule trace-class-iff-summable)
  also have ‹... ⟷ ((λx. cmod (x •C (Proj S *V x))) abs-summable-on B)›
    by simp
  also have ‹... ⟷ ((λx. 1::real) abs-summable-on C)›
  proof (rule summable-on-cong-neutral)
    fix x :: 'a
    show ‹norm 1 = 0› if ‹x ∈ C - B›
      using that ‹C ⊆ B› by auto
    show ‹norm (cmod (x •C (Proj S *V x))) = norm (1::real)› if ‹x ∈ B ∩ C›
      apply (subst Proj-fixes-image)
      using C-def Int-absorb1 that ‹is-onb B›
      by (auto simp: is-onb-def cnorm-eq-1)
    show ‹norm (cmod (x •C (Proj S *V x))) = 0› if ‹x ∈ B - C›
      apply (subst Proj-0-compl)
       apply (subst S-C)
       apply (rule mem-ortho-ccspanI)
      using that ‹is-onb B› ‹C ⊆ B›
      by (force simp: is-onb-def is-ortho-set-def)+
  qed
  also have ‹... ⟷ finite C›
    using infsum-diverge-constant[where A=C and c=‹1::real›]
    by auto
  also have ‹... ⟷ finite-dim-ccsubspace S›
    by (metis C-def S-C ccspan-finite-dim some-onb-of-finite-dim)
  finally show ?thesis
    by -
qed

lemma not-trace-class-trace0: ‹trace a = 0› if ‹¬ trace-class a›
  using that by (simp add: trace-def)


lemma trace-Proj: ‹trace (Proj S) = cdim (space-as-set S)›
proof (cases ‹finite-dim-ccsubspace S›)
  case True
  define C where ‹C = some-onb-of S›
  then obtain B where ‹is-onb B› and ‹C ⊆ B›
    using orthonormal-basis-exists some-onb-of-norm1 by blast
  have [simp]: ‹finite C›
```

```
    using C-def True some-onb-of-finite-dim by blast
  have card-C: ‹card C = cdim (space-as-set S)›
    by (simp add: C-def some-onb-of-card)
  have S-C: ‹S = ccspan C›
    by (simp add: C-def)

  from True have ‹trace-class (Proj S)›
    by (simp add: trace-class-Proj)
  with ‹is-onb B› have ‹((λe. e ·C (Proj S *V e)) has-sum trace (Proj S)) B›
    by (rule trace-has-sum)
  then have ‹((λe. 1) has-sum trace (Proj S)) C›
  proof (rule has-sum-cong-neutral[THEN iffD1, rotated -1])
    fix x :: 'a
    show ‹1 = 0› if ‹x ∈ C - B›
      using that ‹C ⊆ B› by auto
    show ‹x ·C (Proj S *V x) = 1› if ‹x ∈ B ∩ C›
      apply (subst Proj-fixes-image)
      using C-def Int-absorb1 that ‹is-onb B›
      by (auto simp: is-onb-def cnorm-eq-1)
    show ‹is-orthogonal x (Proj S *V x)› if ‹x ∈ B - C›
      apply (subst Proj-0-compl)
       apply (subst S-C)
       apply (rule mem-ortho-ccspanI)
      using that ‹is-onb B› ‹C ⊆ B›
      by (force simp: is-onb-def is-ortho-set-def)+
  qed
  then have ‹trace (Proj S) = card C›
    using has-sum-constant[OF ‹finite C›, of 1]
    apply simp
    using has-sum-unique by blast
  also have ‹... = cdim (space-as-set S)›
    using card-C by presburger
  finally show ?thesis
    by -
next
  case False
  then have ‹¬ trace-class (Proj S)›
    using trace-class-Proj by blast
  then have ‹trace (Proj S) = 0›
    by (rule not-trace-class-trace0)
  moreover from False have ‹cdim (space-as-set S) = 0›
    apply transfer
    by (simp add: cdim-infinite-0)
  ultimately show ?thesis
    by simp
qed

lemma trace-tc-pos: ‹t ≥ 0 ⟹ trace-tc t ≥ 0›
  using trace-tc-mono by fastforce
```

lift-definition *tc-apply* :: ‹('a::*chilbert-space*,'b::*chilbert-space*) *trace-class* ⇒ 'a ⇒ 'b› **is** *cblinfun-apply*.

lemma *bounded-cbilinear-tc-apply*: ‹*bounded-cbilinear tc-apply*›
apply (*rule bounded-cbilinear.intro*; *transfer*)
apply (*auto intro*!: *exI*[*of - 1*] *cblinfun.add-left cblinfun.add-right cblinfun.scaleC-right*)
by (*smt* (*verit*, *del-insts*) *mult-right-mono norm-cblinfun norm-ge-zero norm-leq-trace-norm*)

lift-definition *diagonal-operator-tc* :: ‹('a ⇒ *complex*) ⇒ ('a *ell2*, 'a *ell2*) *trace-class*› **is**
‹λf. *if f abs-summable-on UNIV then diagonal-operator f else 0*›
proof (*rule CollectI*)
fix *f* :: ‹'a ⇒ *complex*›
show ‹*trace-class* (*if f abs-summable-on UNIV then diagonal-operator f else 0*)›
proof (*cases* ‹*f abs-summable-on UNIV*›)
case *True*
have *bdd*: ‹*bdd-above* (*range* (λx. *cmod* (*f x*)))›
proof (*rule bdd-aboveI2*)
fix *x*
have ‹*cmod* (*f x*) = (∑∞x∈{*x*}. *cmod* (*f x*))›
by *simp*
also have ‹... ≤ (∑∞x. *cmod* (*f x*))›
apply (*rule infsum-mono-neutral*)
by (*auto intro*!: *True*)
finally show ‹*cmod* (*f x*) ≤ (∑∞x. *cmod* (*f x*))›
by −
qed
have ‹*trace-class* (*diagonal-operator f*)›
by (*auto intro*!: *trace-classI*[*OF is-onb-ket*] *summable-on-reindex*[*THEN iffD2*] *True*
simp: *abs-op-diagonal-operator o-def diagonal-operator-ket bdd*)
with *True* **show** *?thesis*
by *simp*
next
case *False*
then show *?thesis*
by *simp*
qed
qed

lemma *from-trace-class-diagonal-operator-tc*:
assumes ‹*f abs-summable-on UNIV*›
shows ‹*from-trace-class* (*diagonal-operator-tc f*) = *diagonal-operator f*›
apply (*transfer fixing*: *f*)
using *assms* **by** *simp*

lemma *tc-butterfly-scaleC-summable*:
fixes *f* :: ‹'a ⇒ *complex*›
assumes ‹*f abs-summable-on A*›
shows ‹(λx. *f x* ∗$_C$ *tc-butterfly* (*ket x*) (*ket x*)) *summable-on A*›

proof −
define M **where** ‹$M = (\sum_\infty x{\in}A.\ norm\ (f\ x))$›
have ‹$(\sum x{\in}F.\ cmod\ (f\ x) * norm\ (tc\text{-}butterfly\ (ket\ x)\ (ket\ x))) \leq M$› **if** ‹*finite* F› **and** ‹$F \subseteq A$› **for** F
proof −
have ‹$(\sum x{\in}F.\ norm\ (f\ x) * norm\ (tc\text{-}butterfly\ (ket\ x)\ (ket\ x))) = (\sum x{\in}F.\ norm\ (f\ x))$›
by (*simp add*: *norm-tc-butterfly*)
also have ‹$\ldots \leq (\sum_\infty x{\in}A.\ norm\ (f\ x))$›
using *assms finite-sum-le-infsum norm-ge-zero that(1) that(2)* **by** *blast*
also have ‹$\ldots = M$›
by (*simp add*: *M-def*)
finally show *?thesis*
by −
qed
then have ‹$(\lambda x.\ norm\ (f\ x *_C\ tc\text{-}butterfly\ (ket\ x)\ (ket\ x)))\ abs\text{-}summable\text{-}on\ A$›
apply (*intro nonneg-bdd-above-summable-on bdd-aboveI*)
by *auto*
then show *?thesis*
by (*auto intro*: *abs-summable-summable*)
qed

lemma *tc-butterfly-scaleC-has-sum*:
fixes f :: ‹$'a \Rightarrow complex$›
assumes ‹$f\ abs\text{-}summable\text{-}on\ UNIV$›
shows ‹$((\lambda x.\ f\ x *_C\ tc\text{-}butterfly\ (ket\ x)\ (ket\ x))\ has\text{-}sum\ diagonal\text{-}operator\text{-}tc\ f)\ UNIV$›
proof −
define D **where** ‹$D = (\sum_\infty x.\ f\ x *_C\ tc\text{-}butterfly\ (ket\ x)\ (ket\ x))$›
have *bdd-f*: ‹*bdd-above* (*range* ($\lambda x.\ cmod\ (f\ x)$))›
by (*metis assms summable-on-bdd-above-real*)

have ‹$ket\ y \cdot_C from\text{-}trace\text{-}class\ D\ (ket\ z) = ket\ y \cdot_C from\text{-}trace\text{-}class\ (diagonal\text{-}operator\text{-}tc\ f)\ (ket\ z)$› **for** $y\ z$
proof −
have *blin-tc-apply*: ‹*bounded-linear* ($\lambda a.\ tc\text{-}apply\ a\ (ket\ z)$)›
by (*intro bounded-clinear.bounded-linear bounded-cbilinear.bounded-clinear-left bounded-cbilinear-tc-apply*)
have *summ*: ‹$(\lambda x.\ f\ x *_C\ tc\text{-}butterfly\ (ket\ x)\ (ket\ x))\ summable\text{-}on\ UNIV$›
by (*intro tc-butterfly-scaleC-summable assms*)

have ‹$((\lambda x.\ f\ x *_C\ tc\text{-}butterfly\ (ket\ x)\ (ket\ x))\ has\text{-}sum\ D)\ UNIV$›
by (*simp add*: *D-def summ*)
with *blin-tc-apply* **have** ‹$((\lambda x.\ tc\text{-}apply\ (f\ x *_C\ tc\text{-}butterfly\ (ket\ x)\ (ket\ x))\ (ket\ z))\ has\text{-}sum\ tc\text{-}apply\ D\ (ket\ z))\ UNIV$›
by (*rule Infinite-Sum.has-sum-bounded-linear*)
then have ‹$((\lambda x.\ ket\ y \cdot_C\ tc\text{-}apply\ (f\ x *_C\ tc\text{-}butterfly\ (ket\ x)\ (ket\ x))\ (ket\ z))\ has\text{-}sum\ ket\ y \cdot_C\ tc\text{-}apply\ D\ (ket\ z))\ UNIV$›
by (*smt* (*verit*, *best*) *has-sum-cong has-sum-imp-summable has-sum-infsum infsumI infsum-cinner-left summable-on-cinner-left*)

```
    then have ‹((λx. of-bool (x=y) * of-bool (y=z) * f y) has-sum ket y •C tc-apply D (ket z))
UNIV›
      apply (rule has-sum-cong[THEN iffD2, rotated])
      by (auto intro!: simp: tc-apply.rep-eq scaleC-trace-class.rep-eq tc-butterfly.rep-eq)
    then have ‹((λx. of-bool (y=z) * f y) has-sum ket y •C tc-apply D (ket z)) {y}›
      apply (rule has-sum-cong-neutral[THEN iffD2, rotated −1])
      by auto
    then have ‹ket y •C tc-apply D (ket z) = of-bool (y=z) * f y›
      by simp
    also have ‹... = ket y •C from-trace-class (diagonal-operator-tc f) (ket z)›
      by (simp add: diagonal-operator-tc.rep-eq assms diagonal-operator-ket bdd-f)
    finally show ?thesis
      by (simp add: tc-apply.rep-eq)
  qed
  then have ‹from-trace-class D = from-trace-class (diagonal-operator-tc f)›
    by (auto intro!: equal-ket cinner-ket-eqI)
  then have ‹D = diagonal-operator-tc f›
    by (simp add: from-trace-class-inject)
  with tc-butterfly-scaleC-summable[OF assms]
  show ?thesis
    using D-def by force
qed

lemma diagonal-operator-tc-invalid: ‹¬ f abs-summable-on UNIV ⟹ diagonal-operator-tc f =
0›
  apply (transfer fixing: f) by simp



lemma tc-butterfly-scaleC-infsum:
  fixes f :: ‹'a ⇒ complex›
  shows ‹(∑∞x. f x *C tc-butterfly (ket x) (ket x)) = diagonal-operator-tc f›
proof (cases ‹f abs-summable-on UNIV›)
  case True
  then show ?thesis
    using infsumI tc-butterfly-scaleC-has-sum by fastforce
next
  case False
  then have [simp]: ‹diagonal-operator-tc f = 0›
    apply (transfer fixing: f) by simp
  have ‹¬ (λx. f x *C tc-butterfly (ket x) (ket x)) summable-on UNIV›
  proof (rule notI)
    assume ‹(λx. f x *C tc-butterfly (ket x) (ket x)) summable-on UNIV›
    then have ‹(λx. trace-tc (f x *C tc-butterfly (ket x) (ket x))) summable-on UNIV›
      apply (rule summable-on-bounded-linear[rotated])
      by (simp add: bounded-clinear.bounded-linear)
    then have ‹f summable-on UNIV›
      apply (rule summable-on-cong[THEN iffD1, rotated])
      apply (transfer' fixing: f)
```

by (*simp add*: *trace-scaleC trace-butterfly*)
with *False*
show *False*
by (*metis summable-on-iff-abs-summable-on-complex*)
qed
then have [*simp*]: ‹$(\sum_\infty x.\ f\ x *_C$ *tc-butterfly* (*ket x*) (*ket x*)) = *0*›
using *infsum-not-exists* **by** *blast*
show *?thesis*
by *simp*
qed

lemma *from-trace-class-abs-summable*: ‹*f abs-summable-on X* ⟹ (λ*x*. *from-trace-class* (*f x*)) *abs-summable-on X*›
apply (*rule abs-summable-on-comparison-test*[**where** *g*=‹*f*›])
by (*simp-all add*: *norm-leq-trace-norm norm-trace-class.rep-eq*)

lemma *from-trace-class-summable*: ‹*f summable-on X* ⟹ (λ*x*. *from-trace-class* (*f x*)) *summable-on X*›
apply (*rule Infinite-Sum.summable-on-bounded-linear*[**where** *h*=*from-trace-class*])
by (*simp-all add*: *bounded-clinear.bounded-linear bounded-clinear-from-trace-class*)

lemma *from-trace-class-infsum*:
assumes ‹*f summable-on UNIV*›
shows ‹*from-trace-class* ($\sum_\infty x.\ f\ x$) = ($\sum_\infty x.$ *from-trace-class* (*f x*))›
apply (*rule infsum-bounded-linear-strong*[*symmetric*])
using *assms*
by (*auto intro*!: *bounded-clinear.bounded-linear bounded-clinear-from-trace-class from-trace-class-summable*)

lemma *cspan-trace-class*:
‹*cspan* (*Collect trace-class* :: (*'a*::*chilbert-space* ⇒$_{CL}$ *'b*::*chilbert-space*) *set*) = *Collect trace-class*›
proof (*intro Set.set-eqI iffI*)
fix *x* :: ‹*'a* ⇒$_{CL}$ *'b*›
show ‹*x* ∈ *Collect trace-class* ⟹ *x* ∈ *cspan* (*Collect trace-class*)›
by (*simp add*: *complex-vector.span-clauses*)
assume ‹*x* ∈ *cspan* (*Collect trace-class*)›
then obtain *F f* **where** *x-def*: ‹*x* = ($\sum a{\in}F.\ f\ a *_C\ a$)› **and** ‹*F* ⊆ *Collect trace-class*›
by (*auto intro*!: *simp*: *complex-vector.span-explicit*)
then have ‹*trace-class x*›
by (*auto intro*!: *trace-class-sum trace-class-scaleC simp*: *x-def*)
then show ‹*x* ∈ *Collect trace-class* ›
by *simp*
qed

lemma *monotone-convergence-tc*:
fixes *f* :: ‹*'b* ⇒ (*'a*, *'a*::*chilbert-space*) *trace-class*›
assumes *bounded*: ‹∀$_F$ *x in F*. *trace-tc* (*f x*) ≤ *B*›
assumes *pos*: ‹∀$_F$ *x in F*. *f x* ≥ *0*›
assumes *increasing*: ‹*increasing-filter* (*filtermap f F*)›
shows ‹∃ *L*. (*f* ⟶ *L*) *F*›

proof −
wlog ‹$F \neq \bot$›
using *negation* **by** *simp*
then have ‹*filtermap f F* $\neq \bot$›
by (*simp add*: *filtermap-bot-iff*)
have ‹*mono-on* {*t*::('*a*,'*a*) *trace-class*. *t* ≥ *0*} *trace-tc*›
by (*simp add*: *ord.mono-onI trace-tc-mono*)
with *increasing*
have ‹*increasing-filter* (*filtermap* (*trace-tc o f*) *F*)›
unfolding *filtermap-compose*
apply (*rule increasing-filtermap*)
by (*auto intro*!: *pos simp*: *eventually-filtermap*)
then obtain *l* **where** *l*: ‹((λ*x*. *trace-tc* (*f x*)) ⟶ *l*) *F*›
apply *atomize-elim*
apply (*rule monotone-convergence-complex*)
using *bounded* **by** (*simp-all add*: *o-def*)
have ‹*cauchy-filter* (*filtermap f F*)›
proof (*rule cauchy-filter-metricI*)
fix *e* :: *real* **assume** ‹*e* > *0*›
define *d* **where** ‹*d* = *e*/*4*›
have ‹∀$_F$ *x in filtermap f F*. *dist* (*trace-tc x*) *l* < *d*›
unfolding *eventually-filtermap*
using *l* **apply** (*rule tendstoD*)
using ‹*e* > *0*› **by** (*simp add*: *d-def*)
then obtain *P1* **where** *ev-P1*: ‹*eventually P1* (*filtermap f F*)› **and** *P1*: ‹*P1 x* ⟹ *dist* (*trace-tc x*) *l* < *d*› **for** *x*
by *blast*
from *increasing* **obtain** *P2* **where** *ev-P2*: ‹*eventually P2* (*filtermap f F*)› **and**
P2: ‹*P2 x* ⟹ (∀$_F$ *z in filtermap f F*. *z* ≥ *x*)› **for** *x*
using *increasing-filter-def* **by** *blast*
define *P* **where** ‹*P x* ⟷ *P1 x* ∧ *P2 x*› **for** *x*
with *ev-P1 ev-P2* **have** *ev-P*: ‹*eventually P* (*filtermap f F*)›
by (*auto intro*!: *eventually-conj simp*: *P-def*[*abs-def*])
have ‹*dist x y* < *e*› **if** ‹*P x*› **and** ‹*P y*› **for** *x y*
proof −
from ‹*P x*› **have** ‹∀$_F$ *z in filtermap f F*. *z* ≥ *x*›
by (*simp add*: *P-def P2*)
moreover from ‹*P y*› **have** ‹∀$_F$ *z in filtermap f F*. *z* ≥ *y*›
by (*simp add*: *P-def P2*)
ultimately have ‹∀$_F$ *z in filtermap f F*. *z* ≥ *x* ∧ *z* ≥ *y* ∧ *P1 z*›
using *ev-P1* **by** (*auto intro*!: *eventually-conj*)
from *eventually-happens'*[*OF* ‹*filtermap f F* ≠ ⊥› *this*]
obtain *z* **where** ‹*z* ≥ *x*› **and** ‹*z* ≥ *y*› **and** ‹*P1 z*›
by *auto*
have ‹*dist x y* ≤ *norm* (*z* − *x*) + *norm* (*z* − *y*)›
by (*metis* (*no-types*, *lifting*) *diff-add-cancel diff-add-eq-diff-diff-swap dist-trace-class-def norm-minus-commute norm-triangle-sub*)
also from ‹*x* ≤ *z*› ‹*y* ≤ *z*› **have** ‹... = (*trace-tc z* − *trace-tc x*) + (*trace-tc z* − *trace-tc y*)›

by (*metis* (*no-types*, *lifting*) *cross3-simps*(*16*) *diff-left-mono diff-self norm-tc-pos of-real-add trace-tc-plus*)
also from ‹$x \leq z$› ‹$y \leq z$› **have** ‹$\ldots = norm\ (trace\text{-}tc\ z - trace\text{-}tc\ x) + norm\ (trace\text{-}tc\ z - trace\text{-}tc\ y)$›
by (*simp add*: *Extra-Ordered-Fields.complex-of-real-cmod trace-tc-mono abs-pos*)
also have ‹$\ldots = dist\ (trace\text{-}tc\ z)\ (trace\text{-}tc\ x) + dist\ (trace\text{-}tc\ z)\ (trace\text{-}tc\ y)$›
using *dist-complex-def* **by** *presburger*
also have ‹$\ldots \leq (dist\ (trace\text{-}tc\ z)\ l + dist\ (trace\text{-}tc\ x)\ l) + (dist\ (trace\text{-}tc\ z)\ l + dist\ (trace\text{-}tc\ y)\ l)$›
apply (*intro complex-of-real-mono add-mono*)
by (*simp-all add*: *dist-triangle2*)
also from *P1* ‹*P1 z*› *that* **have** ‹$\ldots < 4 * d$›
by (*smt* (*verit*, *best*) *P-def complex-of-real-strict-mono-iff*)
also have ‹$\ldots = e$›
by (*simp add*: *d-def*)
finally show *?thesis*
by *simp*
qed
with *ev-P* **show** ‹$\exists P.\ eventually\ P\ (filtermap\ f\ F) \wedge (\forall x\ y.\ P\ x \wedge P\ y \longrightarrow dist\ x\ y < e)$›
by *blast*
qed
then have ‹*convergent-filter* (*filtermap f F*)›
using *cauchy-filter-convergent* **by** *fastforce*
then show ‹$\exists L.\ (f \longrightarrow L)\ F$›
by (*simp add*: *convergent-filter-iff filterlim-def*)
qed

lemma *nonneg-bdd-above-summable-on-tc*:
fixes *f* :: ‹'*a* ⇒ ('*c*::*chilbert-space*, '*c*) *trace-class*›
assumes *pos*: ‹$\bigwedge x.\ x \in A \Longrightarrow f\ x \geq 0$›
assumes *bdd*: ‹*bdd-above* (*trace-tc* ‘ *sum f* ‘ {*F*. *F*⊆*A* ∧ *finite F*})›
shows ‹*f summable-on A*›
proof −
have *pos'*: ‹$(\sum x \in F.\ f\ x) \geq 0$› **if** ‹*finite F*› **and** ‹$F \subseteq A$› **for** *F*
using *that pos*
by (*simp add*: *basic-trans-rules*(*31*) *sum-nonneg*)
from *pos* **have** *incr*: ‹*increasing-filter* (*filtermap* (*sum f*) (*finite-subsets-at-top A*))›
by (*auto intro*!: *increasing-filtermap*[**where** *X*=‹{*F*. *finite F* ∧ *F* ⊆ *A*}›] *mono-onI sum-mono2*)
from *bdd* **obtain** *B* **where** *B*: ‹*trace-tc* (*sum f X*) ≤ *B*› **if** ‹*finite X*› **and** ‹$X \subseteq A$› **for** *X*
apply *atomize-elim*
by (*auto simp*: *bdd-above-def*)
show *?thesis*
apply (*simp add*: *summable-on-def has-sum-def*)
by (*safe intro*!: *pos' incr monotone-convergence-tc*[**where** *B*=*B*] *B*
eventually-finite-subsets-at-top-weakI)
qed

lemma *summable-Sigma-positive-tc*:

fixes $f :: \langle 'a \Rightarrow 'b \Rightarrow ('c, 'c::chilbert\text{-}space)\ trace\text{-}class\rangle$
assumes $\langle\bigwedge x.\ x{\in}X \Longrightarrow f\ x\ summable\text{-}on\ Y\ x\rangle$
assumes $\langle(\lambda x.\ \sum_\infty y{\in}Y\ x.\ f\ x\ y)\ summable\text{-}on\ X\rangle$
assumes $\langle\bigwedge x\ y.\ x \in X \Longrightarrow y \in Y\ x \Longrightarrow f\ x\ y \geq 0\rangle$
shows $\langle(\lambda(x,\ y).\ f\ x\ y)\ summable\text{-}on\ (SIGMA\ x{:}X.\ Y\ x)\rangle$
proof −
have $\langle trace\text{-}tc\ (\sum(x,y){\in}F.\ f\ x\ y) \leq trace\text{-}tc\ (\sum_\infty x{\in}X.\ \sum_\infty y{\in}Y\ x.\ f\ x\ y)\rangle$ **if** $\langle F \subseteq Sigma\ X\ Y\rangle$ **and** $\langle finite\ F\rangle$ **for** F
proof −
define g **where** $\langle g\ x\ y = (if\ (x,y) \in Sigma\ X\ Y\ then\ f\ x\ y\ else\ 0)\rangle$ **for** $x\ y$
have $g\text{-}pos[iff]$: $\langle g\ x\ y \geq 0\rangle$ **for** $x\ y$
using *assms* **by** (*auto intro*!: *simp*: *g-def*)
have *g-summable*: $\langle g\ x\ summable\text{-}on\ Y\ x\rangle$ **for** x
by (*metis assms*(*1*) *g-def mem-Sigma-iff summable-on-0 summable-on-cong*)
have *sum-g-summable*: $\langle(\lambda x.\ \sum_\infty y{\in}Y\ x.\ g\ x\ y)\ summable\text{-}on\ X\rangle$
by (*metis* (*mono-tags*, *lifting*) *SigmaI g-def assms*(*2*) *infsum-cong summable-on-cong*)
have $\langle(\sum(x,y){\in}F.\ f\ x\ y) = (\sum(x,y){\in}F.\ g\ x\ y)\rangle$
by (*smt* (*verit*, *ccfv-SIG*) *g-def split-cong subsetD sum.cong that*(*1*))
also have $\langle(\sum(x,y){\in}F.\ g\ x\ y) \leq (\sum(x,y){\in}fst\ `\ F \times snd\ `\ F.\ g\ x\ y)\rangle$
using *that assms* **apply** (*auto intro*!: *sum-mono2 assms simp*: *image-iff*)
by *force*+
also have $\langle\ldots = (\sum x{\in}fst\ `\ F.\ \sum y{\in}snd\ `\ F.\ g\ x\ y)\rangle$
by (*metis* (*no-types*, *lifting*) *finite-imageI sum.Sigma sum.cong that*(*2*))
also have $\langle\ldots = (\sum x{\in}fst\ `\ F.\ \sum_\infty y{\in}snd\ `\ F.\ g\ x\ y)\rangle$
by (*metis finite-imageI infsum-finite that*(*2*))
also have $\langle\ldots \leq (\sum x{\in}fst\ `\ F.\ \sum_\infty y{\in}Y\ x.\ g\ x\ y)\rangle$
apply (*intro sum-mono infsum-mono-neutral-traceclass*)
using *assms that*
apply (*auto intro*!: *g-summable*)
by (*simp add*: *g-def*)
also have $\langle\ldots = (\sum_\infty x{\in}fst\ `\ F.\ \sum_\infty y{\in}Y\ x.\ g\ x\ y)\rangle$
using *that* **by** (*auto intro*!: *infsum-finite*[*symmetric*] *simp*:)
also have $\langle\ldots \leq (\sum_\infty x{\in}X.\ \sum_\infty y{\in}Y\ x.\ g\ x\ y)\rangle$
apply (*rule infsum-mono-neutral-traceclass*)
using *that assms* **by** (*auto intro*!: *infsum-nonneg-traceclass sum-g-summable*)
also have $\langle\ldots = (\sum_\infty x{\in}X.\ \sum_\infty y{\in}Y\ x.\ f\ x\ y)\rangle$
by (*smt* (*verit*, *ccfv-threshold*) *g-def infsum-cong mem-Sigma-iff*)
finally show *?thesis*
using *trace-tc-mono* **by** *blast*
qed
then show *?thesis*
apply (*rule-tac nonneg-bdd-above-summable-on-tc*)
by (*auto intro*!: *assms bdd-aboveI2*)
qed

lemma *infsum-Sigma-positive-tc*:
fixes $f :: \langle 'a \Rightarrow 'b \Rightarrow ('c{::}chilbert\text{-}space, 'c)\ trace\text{-}class\rangle$
assumes $\langle\bigwedge x.\ x{\in}X \Longrightarrow f\ x\ summable\text{-}on\ Y\ x\rangle$

assumes ‹$\bigwedge x\ y.\ x \in X \Longrightarrow y \in Y\ x \Longrightarrow f\ x\ y \geq 0$›
shows ‹$(\sum_\infty x{\in}X.\ \sum_\infty y{\in}Y\ x.\ f\ x\ y) = (\sum_\infty (x,y){\in}Sigma\ X\ Y.\ f\ x\ y)$›
proof (*cases* ‹$(\lambda x.\ \sum_\infty y{\in}Y\ x.\ f\ x\ y)$ *summable-on* X›)
case *True*
show *?thesis*
apply (*rule infsum-Sigma'-banach*)
apply (*rule summable-Sigma-positive-tc*)
using *assms True* **by** *auto*
next
case *False*
then have *1*: ‹$(\sum_\infty x{\in}X.\ \sum_\infty y{\in}Y\ x.\ f\ x\ y) = 0$›
using *infsum-not-exists* **by** *blast*
from *False* **have** ‹$\neg$ $(\lambda(x,y).\ f\ x\ y)$ *summable-on Sigma X Y*›
using *summable-on-Sigma-banach* **by** *blast*
then have *2*: ‹$(\sum_\infty (x,\ y){\in}Sigma\ X\ Y.\ f\ x\ y) = 0$›
using *infsum-not-exists* **by** *blast*
from *1 2* **show** *?thesis*
by *simp*
qed

lemma *infsum-swap-positive-tc*:
fixes f :: ‹$'a \Rightarrow 'b \Rightarrow ('c{::}chilbert\text{-}space,\ 'c)$ *trace-class*›
assumes ‹$\bigwedge x.\ x{\in}X \Longrightarrow f\ x$ *summable-on* Y›
assumes ‹$\bigwedge y.\ y{\in}Y \Longrightarrow (\lambda x.\ f\ x\ y)$ *summable-on* X›
assumes ‹$\bigwedge x\ y.\ x \in X \Longrightarrow y \in Y \Longrightarrow f\ x\ y \geq 0$›
shows ‹$(\sum_\infty x{\in}X.\ \sum_\infty y{\in}Y.\ f\ x\ y) = (\sum_\infty y{\in}Y.\ \sum_\infty x{\in}X.\ f\ x\ y)$›
proof −
have ‹$(\sum_\infty x{\in}X.\ \sum_\infty y{\in}Y.\ f\ x\ y) = (\sum_\infty (x,y){\in}X\times Y.\ f\ x\ y)$›
apply (*rule infsum-Sigma-positive-tc*)
using *assms* **by** *auto*
also have ‹$\ldots = (\sum_\infty (y,x){\in}Y\times X.\ f\ x\ y)$›
apply (*subst product-swap*[*symmetric*])
by (*simp add*: *infsum-reindex o-def*)
also have ‹$\ldots = (\sum_\infty y{\in}Y.\ \sum_\infty x{\in}X.\ f\ x\ y)$›
apply (*rule infsum-Sigma-positive-tc*[*symmetric*])
using *assms* **by** *auto*
finally show *?thesis*
by −
qed

lemma *separating-density-ops*:
assumes ‹$B > 0$›
shows ‹*separating-set clinear* $\{t :: ('a{::}chilbert\text{-}space,\ 'a)\ trace\text{-}class.\ 0 \leq t \wedge norm\ t \leq B\}$›
proof −
define S **where** ‹$S = \{t :: ('a,\ 'a)\ trace\text{-}class.\ 0 \leq t \wedge norm\ t \leq B\}$›
have ‹*cspan* $S = UNIV$›
proof (*intro Set.set-eqI iffI UNIV-I*)

```
    fix t :: ‹('a, 'a) trace-class›
    from trace-class-decomp-4pos' 
    obtain t1 t2 t3 t4 where t-decomp: ‹t = t1 - t2 + i *C t3 - i *C t4›
      and pos: ‹t1 ≥ 0› ‹t2 ≥ 0› ‹t3 ≥ 0› ‹t4 ≥ 0›
      by fast
    have ‹t' ∈ cspan S› if ‹t' ≥ 0› for t'
    proof -
      define t'' where ‹t'' = (B / norm t') *R t'›
      have ‹t'' ∈ S›
        using ‹B > 0›
        by (simp add: S-def that zero-le-scaleR-iff t''-def)
      have t'-t'': ‹t' = (norm t' / B) *R t''›
        using ‹B > 0› t''-def by auto
      show ‹t' ∈ cspan S›
        apply (subst t'-t'')
        using ‹t'' ∈ S›
        by (simp add: scaleR-scaleC complex-vector.span-clauses)
    qed
    with pos have ‹t1 ∈ cspan S› and ‹t2 ∈ cspan S› and ‹t3 ∈ cspan S› and ‹t4 ∈ cspan S›
      by auto
    then show ‹t ∈ cspan S›
     by (auto intro!: complex-vector.span-diff complex-vector.span-add complex-vector.span-scale
         intro: complex-vector.span-base simp: t-decomp)
  qed
  then show ‹separating-set clinear S›
    by (rule separating-set-clinear-cspan)
qed

lemma summable-abs-summable-tc:
  fixes f :: ‹'a ⇒ ('b::chilbert-space,'b) trace-class›
  assumes ‹f summable-on X›
  assumes ‹⋀x. x∈X ⟹ f x ≥ 0›
  shows ‹f abs-summable-on X›
proof -
  from assms(1) have ‹(λx. trace-tc (f x)) summable-on X›
    apply (rule summable-on-bounded-linear[rotated])
    by (simp add: bounded-clinear.bounded-linear)
  then have ‹(λx. Re (trace-tc (f x))) summable-on X›
    using summable-on-Re by blast
  then show ‹(λx. norm (f x)) summable-on X›
    by (metis (mono-tags, lifting) assms(2) norm-tc-pos-Re summable-on-cong)
qed
```

12.5 More Hilbert-Schmidt

```
lemma trace-class-hilbert-schmidt: ‹hilbert-schmidt a› if ‹trace-class a›
  for a :: ‹'a::chilbert-space ⇒CL 'b::chilbert-space›
  by (auto intro!: trace-class-comp-right that simp: hilbert-schmidt-def)
```

lemma *finite-rank-hilbert-schmidt*: ‹*hilbert-schmidt a*› **if** ‹*finite-rank a*›
for *a* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
using *finite-rank-comp-right finite-rank-trace-class hilbert-schmidtI that* **by** *blast*

lemma *hilbert-schmidt-compact*: ‹*compact-op a*› **if** ‹*hilbert-schmidt a*›
for *a* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
— [1], Corollary 18.7. (Only the second part. The first part is stated inside this proof though.)
proof −
have ‹∃ *b*. *finite-rank b* ∧ *hilbert-schmidt-norm* (*b* − *a*) < *ε*› **if** ‹*ε* > *0*› **for** *ε*
proof −
have ‹*ε*^{2} > *0*›
using *that* **by** *force*
obtain *B* :: ‹'*a set*› **where** ‹*is-onb B*›
using *is-onb-some-chilbert-basis* **by** *blast*
with ‹*hilbert-schmidt a*› **have** *a-sum-B*: ‹(λ*x*. (*norm* (*a* *$_V$ *x*))^2) *summable-on B*›
by (*auto intro*!: *summable-hilbert-schmidt-norm-square*)
then have ‹((λ*x*. (*norm* (*a* *$_V$ *x*))^2) *has-sum* (∑$_\infty$*x*∈*B*. (*norm* (*a* *$_V$ *x*))^2)) *B*›
using *has-sum-infsum* **by** *blast*
from *tendsto-iff*[*THEN iffD1*, *rule-format*, *OF this*[*unfolded has-sum-def*] ‹*ε*^{2} > *0*›]
obtain *F* **where** [*simp*]: ‹*finite F*› **and** ‹*F* ⊆ *B*›
and *Fbound*: ‹*dist* (∑*x*∈*F*. (*norm* (*a* *$_V$ *x*))^2) (∑$_\infty$*x*∈*B*. (*norm* (*a* *$_V$ *x*))^2) < *ε*^{2}›
apply *atomize-elim*
by (*auto intro*!: *simp*: *eventually-finite-subsets-at-top*)
define *p b* **where** ‹*p* = (∑*x*∈*F*. *selfbutter x*)› **and** ‹*b* = *a* *$_{CL}$ *p*›
have [*simp*]: ‹*p x* = *x*› **if** ‹*x* ∈ *F*› **for** *x*
apply (*simp add*: *p-def cblinfun.sum-left*)
apply (*subst sum-single*[**where** *i*=*x*])
using ‹*F* ⊆ *B*› *that* ‹*is-onb B*›
by (*auto intro*!: *simp*: *cnorm-eq-1 is-onb-def is-ortho-set-def*)
have [*simp*]: ‹*p x* = *0*› **if** ‹*x* ∈ *B* − *F*› **for** *x*
using ‹*F* ⊆ *B*› *that* ‹*is-onb B*›
apply (*auto intro*!: *sum.neutral simp add*: *p-def cblinfun.sum-left is-onb-def is-ortho-set-def*)
by *auto*
have ‹*finite-rank p*›
by (*simp add*: *finite-rank-sum p-def*)
then have ‹*finite-rank b*›
by (*simp add*: *b-def finite-rank-comp-right*)
with ‹*hilbert-schmidt a*› **have** ‹*hilbert-schmidt* (*b* − *a*)›
by (*auto intro*!: *hilbert-schmidt-minus intro*: *finite-rank-hilbert-schmidt*)
then have ‹(*hilbert-schmidt-norm* (*b* − *a*))^2 = (∑$_\infty$*x*∈*B*. (*norm* ((*b* − *a*) *$_V$ *x*))^2)›
by (*simp add*: *infsum-hilbert-schmidt-norm-square* ‹*is-onb B*›)
also have ‹... = (∑$_\infty$*x*∈*B*−*F*. (*norm* (*a* *$_V$ *x*))^2)›
by (*auto intro*!: *infsum-cong-neutral*
simp: *b-def cblinfun.diff-left*)
also have ‹... = (∑$_\infty$*x*∈*B*. (*norm* (*a* *$_V$ *x*))^2) − (∑*x*∈*F*. (*norm* (*a* *$_V$ *x*))^2)›
apply (*subst infsum-Diff*)
using ‹*F* ⊆ *B*› *a-sum-B* **by** *auto*
also have ‹... < *ε*^{2}›
using *Fbound*

by (*simp add*: *dist-norm*)
finally show *?thesis*
using ‹*finite-rank b*›
using *power-less-imp-less-base that* **by** *fastforce*
qed
then have ‹∃ *b*. *finite-rank b* ∧ *dist b a* < *ε*› **if** ‹*ε* > *0*› **for** *ε*
apply (*rule ex-mono*[*rule-format*, *rotated*])
apply (*auto intro*!: *that simp*: *dist-norm*)
using *hilbert-schmidt-minus* ‹*hilbert-schmidt a*› *finite-rank-hilbert-schmidt hilbert-schmidt-norm-geq-norm*
by *fastforce*
then show *?thesis*
by (*simp add*: *compact-op-finite-rank closure-approachable*)
qed

lemma *trace-class-compact*: ‹*compact-op a*› **if** ‹*trace-class a*›
for *a* :: ‹*'a*::*chilbert-space* ⇒$_{CL}$ *'b*::*chilbert-space*›
by (*simp add*: *hilbert-schmidt-compact that trace-class-hilbert-schmidt*)

12.6 Spectral Theorem

The spectral theorem for trace class operators. A corollary of the one for compact operators (*Hilbert-Space-Tensor-Product.Spectral-Theorem*) but not an immediate one.

lift-definition *spectral-dec-proj-tc* :: ‹(*'a*::*chilbert-space*, *'a*) *trace-class* ⇒ *nat* ⇒ (*'a*, *'a*) *trace-class*›
is
spectral-dec-proj
using *finite-rank-trace-class spectral-dec-proj-finite-rank trace-class-compact* **by** *blast*

lift-definition *spectral-dec-val-tc* :: ‹(*'a*::*chilbert-space*, *'a*) *trace-class* ⇒ *nat* ⇒ *complex*› **is**
spectral-dec-val.

lemma *spectral-dec-proj-tc-finite-rank*:
assumes ‹*adj-tc a* = *a*›
shows ‹*finite-rank-tc* (*spectral-dec-proj-tc a n*)›
using *assms* **apply** *transfer*
by (*simp add*: *spectral-dec-proj-finite-rank trace-class-compact*)

lemma *spectral-dec-summable-tc*:
assumes ‹*selfadjoint-tc a*›
shows ‹(λ*n*. *spectral-dec-val-tc a n* ∗$_C$ *spectral-dec-proj-tc a n*) *abs-summable-on UNIV*›
proof (*intro nonneg-bounded-partial-sums-imp-summable-on norm-ge-zero eventually-finite-subsets-at-top-weakI*)
define *a'* **where** ‹*a'* = *from-trace-class a*›
then have [*transfer-rule*]: ‹*cr-trace-class a' a*›
by (*simp add*: *cr-trace-class-def*)

have ‹*compact-op a'*›
by (*auto intro*!: *trace-class-compact simp*: *a'-def*)
have ‹*selfadjoint a'*›
using *a'-def assms selfadjoint-tc.rep-eq* **by** *blast*

fix F :: ‹*nat set*› **assume** ‹*finite F*›
define R **where** ‹$R = (\bigsqcup n{\in}F.$ *spectral-dec-space* a' $n)$›
have ‹$(\sum x{\in}F.$ *norm* (*spectral-dec-val-tc a x* $*_C$ *spectral-dec-proj-tc a x*))
$=$ *norm* $(\sum x{\in}F.$ *spectral-dec-val-tc a x* $*_C$ *spectral-dec-proj-tc a x*)›
proof (*rule norm-tc-sum-exchange*[*symmetric*]; *transfer*; *rename-tac n m F*)
fix n m :: *nat* **assume** ‹$n \neq m$›
then have $*$: ‹*Proj* (*spectral-dec-space* a' n) o_{CL} *Proj* (*spectral-dec-space* a' m) $= 0$› **if** ‹*spectral-dec-val* a' $n \neq 0$› **and** ‹*spectral-dec-val* a' $m \neq 0$›
by (*auto intro*!: *orthogonal-projectors-orthogonal-spaces*[*THEN iffD1*] *spectral-dec-space-orthogonal* ‹*compact-op* a'› ‹*selfadjoint* a'›*simp*:)
show ‹(*spectral-dec-val* a' n $*_C$ *spectral-dec-proj* a' n)$*$ o_{CL} *spectral-dec-val* a' m $*_C$ *spectral-dec-proj* a' $m = 0$›
by (*auto intro*!: $*$ *simp*: *spectral-dec-proj-def adj-Proj*)
show ‹*spectral-dec-val* a' n $*_C$ *spectral-dec-proj* a' n o_{CL} (*spectral-dec-val* a' m $*_C$ *spectral-dec-proj* a' m)$* = 0$›
by (*auto intro*!: $*$ *simp*: *spectral-dec-proj-def adj-Proj*)
qed
also have ‹$\ldots$ = *trace-norm* $(\sum x{\in}F.$ *spectral-dec-val* a' x $*_C$ *spectral-dec-proj* a' $x)$›
by (*metis* (*no-types*, *lifting*) a'*-def spectral-dec-proj-tc.rep-eq spectral-dec-val-tc.rep-eq from-trace-class-sum norm-trace-class.rep-eq scaleC-trace-class.rep-eq sum.cong*)
also have ‹$\ldots$ = *trace-norm* $(\sum x.$ *if* $x{\in}F$ *then spectral-dec-val* a' x $*_C$ *spectral-dec-proj* a' x *else* $0)$›
by (*simp add*: ‹*finite F*› *suminf-If-finite-set*)
also have ‹$\ldots$ = *trace-norm* $(\sum x.$ (*spectral-dec-val* a' x $*_C$ *spectral-dec-proj* a' x) o_{CL} *Proj R*)›
proof −
have ‹*spectral-dec-proj* a' n = *spectral-dec-proj* a' n o_{CL} *Proj R*› **if** ‹$n \in F$› **for** n
by (*auto intro*!: *Proj-o-Proj-subspace-left*[*symmetric*] *SUP-upper that simp*: *spectral-dec-proj-def R-def*)
moreover have ‹*spectral-dec-proj* a' n o_{CL} *Proj R* $= 0$› **if** ‹$n \notin F$› **for** n
using *that*
by (*auto intro*!: *orthogonal-spaces-SUP-right spectral-dec-space-orthogonal* ‹*compact-op* a'› ‹*selfadjoint* a'›
simp: *spectral-dec-proj-def R-def*
simp flip: *orthogonal-projectors-orthogonal-spaces*)
ultimately show *?thesis*
by (*auto intro*!: *arg-cong*[**where** *f*=*trace-norm*] *suminf-cong*)
qed
also have ‹$\ldots$ = *trace-norm* $((\sum x.$ *spectral-dec-val* a' x $*_C$ *spectral-dec-proj* a' x) o_{CL} *Proj R*)›
apply (*intro arg-cong*[**where** *f*=*trace-norm*] *bounded-linear.suminf*[*symmetric*] *bounded-clinear.bounded-linear bounded-clinear-cblinfun-compose-left sums-summable*)
using ‹*compact-op* a'› ‹*selfadjoint* a'› *spectral-dec-sums* **by** *blast*
also have ‹$\ldots$ = *trace-norm* $(a'$ o_{CL} *Proj R*)›
using *spectral-dec-sums*[*OF* ‹*compact-op* a'› ‹*selfadjoint* a'›] *sums-unique* **by** *fastforce*
also have ‹$\ldots \leq$ *trace-norm* $a' *$ *norm* (*Proj R*)›
by (*auto intro*!: *trace-norm-comp-left simp*: a'*-def*)
also have ‹$\ldots \leq$ *trace-norm* a'›
by (*simp add*: *mult-left-le norm-Proj-leq1*)

finally show ‹$(\sum x{\in}F.\ norm\ (spectral\text{-}dec\text{-}val\text{-}tc\ a\ x *_C spectral\text{-}dec\text{-}proj\text{-}tc\ a\ x)) \leq trace\text{-}norm\ a'$›
by −
qed

lemma *spectral-dec-has-sum-tc*:
assumes ‹*selfadjoint-tc a*›
shows ‹$((\lambda n.\ spectral\text{-}dec\text{-}val\text{-}tc\ a\ n *_C spectral\text{-}dec\text{-}proj\text{-}tc\ a\ n)\ \ has\text{-}sum\ \ a)\ UNIV$›
proof −
define $a'\ b\ b'$ **where** ‹$a' = from\text{-}trace\text{-}class\ a$›
and ‹$b = (\sum_\infty n.\ spectral\text{-}dec\text{-}val\text{-}tc\ a\ n *_C spectral\text{-}dec\text{-}proj\text{-}tc\ a\ n)$› **and** ‹$b' = from\text{-}trace\text{-}class\ b$›
have [*simp*]: ‹*compact-op a'*›
by (*auto intro*!: *trace-class-compact simp*: *a'-def*)
have [*simp*]: ‹*selfadjoint a'*›
using *a'-def assms selfadjoint-tc.rep-eq* **by** *blast*
have [*simp*]: ‹*trace-class b'*›
by (*simp add*: *b'-def*)
from *spectral-dec-summable-tc*[*OF assms*]
have *has-sum-b*: ‹$((\lambda n.\ spectral\text{-}dec\text{-}val\text{-}tc\ a\ n *_C spectral\text{-}dec\text{-}proj\text{-}tc\ a\ n)\ \ has\text{-}sum\ \ b)\ UNIV$›
by (*metis abs-summable-summable b-def summable-iff-has-sum-infsum*)
then have ‹$((\lambda F.\ \sum n{\in}F.\ spectral\text{-}dec\text{-}val\text{-}tc\ a\ n *_C spectral\text{-}dec\text{-}proj\text{-}tc\ a\ n) \longrightarrow b)\ (finite\text{-}subsets\text{-}at\text{-}top\ UNIV)$›
by (*simp add*: *has-sum-def*)
then have ‹$((\lambda F.\ norm\ ((\sum n{\in}F.\ spectral\text{-}dec\text{-}val\text{-}tc\ a\ n *_C spectral\text{-}dec\text{-}proj\text{-}tc\ a\ n) - b)) \longrightarrow 0)\ (finite\text{-}subsets\text{-}at\text{-}top\ UNIV)$›
using *LIM-zero tendsto-norm-zero* **by** *blast*
then have ‹$((\lambda F.\ norm\ ((\sum n{\in}F.\ spectral\text{-}dec\text{-}val\text{-}tc\ a\ n *_C spectral\text{-}dec\text{-}proj\text{-}tc\ a\ n) - b)) \longrightarrow 0)\ (filtermap\ (\lambda n.\ \{..{<}n\})\ sequentially)$›
by (*meson filterlim-compose filterlim-filtermap filterlim-lessThan-at-top*)
then have ‹$((\lambda m.\ norm\ ((\sum n{\in}\{..{<}m\}.\ spectral\text{-}dec\text{-}val\text{-}tc\ a\ n *_C spectral\text{-}dec\text{-}proj\text{-}tc\ a\ n) - b)) \longrightarrow 0)\ sequentially$›
by (*simp add*: *filterlim-filtermap*)
then have ‹$((\lambda m.\ trace\text{-}norm\ ((\sum n{\in}\{..{<}m\}.\ spectral\text{-}dec\text{-}val\ a'\ n *_C spectral\text{-}dec\text{-}proj\ a'\ n) - b')) \longrightarrow 0)\ sequentially$›
unfolding *a'-def b'-def*
by *transfer*
then have ‹$((\lambda m.\ norm\ ((\sum n{\in}\{..{<}m\}.\ spectral\text{-}dec\text{-}val\ a'\ n *_C spectral\text{-}dec\text{-}proj\ a'\ n) - b')) \longrightarrow 0)\ sequentially$›
apply (*rule tendsto-0-le*[**where** *K=1*])
by (*auto intro*!: *eventually-sequentiallyI norm-leq-trace-norm trace-class-minus trace-class-sum trace-class-scaleC spectral-dec-proj-finite-rank intro*: *finite-rank-trace-class*)
then have ‹$(\lambda n.\ spectral\text{-}dec\text{-}val\ a'\ n *_C spectral\text{-}dec\text{-}proj\ a'\ n)\ sums\ b'$›
using *LIM-zero-cancel sums-def tendsto-norm-zero-iff* **by** *blast*
moreover have ‹$(\lambda n.\ spectral\text{-}dec\text{-}val\ a'\ n *_C spectral\text{-}dec\text{-}proj\ a'\ n)\ sums\ a'$›
using ‹*compact-op a'*› ‹*selfadjoint a'*› **by** (*rule spectral-dec-sums*)
ultimately have ‹$a = b$›

using *a′-def b′-def from-trace-class-inject sums-unique2* **by** *blast*
with *has-sum-b* **show** *?thesis*
by *simp*
qed

lemma *spectral-dec-sums-tc*:
assumes ‹*selfadjoint-tc a*›
shows ‹(λ*n*. *spectral-dec-val-tc a n* $*_C$ *spectral-dec-proj-tc a n*) *sums a*›
using *assms has-sum-imp-sums spectral-dec-has-sum-tc* **by** *blast*

lift-definition *spectral-dec-vecs-tc* :: ‹('*a*,'*a*) *trace-class* ⇒ '*a*::*chilbert-space set*› **is**
spectral-dec-vecs.

lemma *compact-from-trace-class*[*iff*]: ‹*compact-op* (*from-trace-class t*)›
by (*auto intro*!: *simp*: *trace-class-compact*)

lemma *sum-some-onb-of-tc-butterfly*:
assumes ‹*finite-dim-ccsubspace S*›
shows ‹($\sum$ *x*∈*some-onb-of S*. *tc-butterfly x x*) = *Abs-trace-class* (*Proj S*)›
by (*metis* (*mono-tags*, *lifting*) *assms from-trace-class-inverse from-trace-class-sum sum.cong sum-some-onb-of-butterfly tc-butterfly.rep-eq*)

lemma *butterfly-spectral-dec-vec-tc-has-sum*:
assumes ‹*t* ≥ *0*›
shows ‹((λ*v*. *tc-butterfly v v*) *has-sum t*) (*spectral-dec-vecs-tc t*)›
proof −
define *t′* **where** ‹*t′* = *from-trace-class t*›
note *power2-csqrt*[*unfolded power2-eq-square*, *simp*]
note *Reals-cnj-iff*[*simp*]
have [*simp*]: ‹*compact-op t′*›
by (*simp add*: *t′-def*)
from *assms* **have** ‹*selfadjoint-tc t*›
apply *transfer*
apply (*rule comparable-hermitean*[*of 0*])
by *simp-all*
have *spectral-real*[*simp*]: ‹*spectral-dec-val t′ n* ∈ ℝ› **for** *n*
apply (*rule spectral-dec-val-real*)
using ‹*selfadjoint-tc t*› **by** (*auto intro*!: *trace-class-compact simp*: *selfadjoint-tc.rep-eq t′-def*)

have ∗: ‹((λ(*n*,*v*). *tc-butterfly v v*) *has-sum t*) (*SIGMA n*:*UNIV*. ($*_C$) (*csqrt* (*spectral-dec-val t′ n*)) ‘ *some-onb-of* (*spectral-dec-space t′ n*))›
proof (*rule has-sum-SigmaI*[**where** *g*=‹λ*n*. *spectral-dec-val t′ n* $*_C$ *spectral-dec-proj-tc t n*›])
have ‹*spectral-dec-val t′ n* ≥ *0*› **for** *n*
by (*simp add*: *assms from-trace-class-pos spectral-dec-val-nonneg t′-def*)
then have [*simp*]: ‹*cnj* (*csqrt* (*spectral-dec-val t′ n*)) ∗ *csqrt* (*spectral-dec-val t′ n*) = *spectral-dec-val t′ n*› **for** *n*
apply (*auto simp add*: *csqrt-of-real-nonneg less-eq-complex-def*)
by (*metis of-real-Re of-real-mult spectral-real sqrt-sqrt*)

have *sum*: ‹($\sum$ *y*∈(λ*x*. *csqrt* (*spectral-dec-val t′ n*) $*_C$ *x*) ‘ *some-onb-of* (*spectral-dec-space t′ n*). *tc-butterfly y y*) = *spectral-dec-val t′ n* $*_C$ *spectral-dec-proj-tc t n*› **for** *n*
proof (*cases* ‹*spectral-dec-val t′ n* = *0*›)
case *True*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *csqrt-0 imageE scaleC-eq-0-iff sum.neutral tc-butterfly-scaleC-left*)
next
case *False*
then have ‹*inj-on* (λ*x*. *csqrt* (*spectral-dec-val t′ n*) $*_C$ *x*) *X*› **for** *X* :: ‹*'a set*›
by (*meson csqrt-eq-0 inj-scaleC*)
then show *?thesis*
by (*simp add*: *sum.reindex False spectral-dec-space-finite-dim sum-some-onb-of-tc-butterfly spectral-dec-proj-def spectral-dec-proj-tc-def flip*: *scaleC-sum-right t′-def*)
qed
then show ‹((λ*y*. *case* (*n*, *y*) *of* (*n*, *v*) ⇒ *tc-butterfly v v*) *has-sum spectral-dec-val t′ n* $*_C$ *spectral-dec-proj-tc t n*)
(($*_C$) (*csqrt* (*spectral-dec-val t′ n*)) ‘ *some-onb-of* (*spectral-dec-space t′ n*))› **for** *n*
by (*auto intro*!: *has-sum-finiteI finite-imageI some-onb-of-finite-dim spectral-dec-space-finite-dim simp*: *t′-def*)
show ‹((λ*n*. *spectral-dec-val t′ n* $*_C$ *spectral-dec-proj-tc t n*) *has-sum t*) *UNIV*›
by (*auto intro*!: *spectral-dec-has-sum-tc* ‹*selfadjoint-tc t*› *simp*: *t′-def simp flip*: *spectral-dec-val-tc.rep-eq*)
show ‹(λ(*n*, *v*). *tc-butterfly v v*) *summable-on* (*SIGMA n*:*UNIV*. ($*_C$) (*csqrt* (*spectral-dec-val t′ n*)) ‘ *some-onb-of* (*spectral-dec-space t′ n*))›
proof −
have *inj*: ‹*inj-on* (($*_C$) (*csqrt* (*spectral-dec-val t′ n*))) (*some-onb-of* (*spectral-dec-space t′ n*))› **for** *n*
proof (*cases* ‹*spectral-dec-val t′ n* = *0*›)
case *True*
then have ‹*spectral-dec-space t′ n* = *0*›
using *spectral-dec-space-0* **by** *blast*
then have ‹*some-onb-of* (*spectral-dec-space t′ n*) = {}›
using *some-onb-of-0* **by** *auto*
then show *?thesis*
by *simp*
next
case *False*
then show *?thesis*
by (*auto intro*!: *inj-scaleC*)
qed
have *1*: ‹(λ*x*. *tc-butterfly x x*) *abs-summable-on* (λ*xa*. *csqrt* (*spectral-dec-val t′ n*) $*_C$ *xa*) ‘ *some-onb-of* (*spectral-dec-space t′ n*)› **for** *n*
by (*auto intro*!: *summable-on-finite some-onb-of-finite-dim spectral-dec-space-finite-dim simp*: *t′-def*)

have ‹(λ*n*. *cmod* (*spectral-dec-val t′ n*) * ($\sum_\infty$*h*∈*some-onb-of* (*spectral-dec-space t′ n*). *norm* (*tc-butterfly h h*))) *abs-summable-on UNIV*›
proof −
have *∗*: ‹($\sum_\infty$*h*∈*some-onb-of* (*spectral-dec-space t′ n*). *norm* (*tc-butterfly h h*)) = *norm*

(*spectral-dec-proj-tc t n*)› **for** *n*
proof −
have ‹(∑∞*h*∈*some-onb-of* (*spectral-dec-space t′ n*). *norm* (*tc-butterfly h h*))
= (∑∞*h*∈*some-onb-of* (*spectral-dec-space t′ n*). *1*)›
by (*simp add*: *infsum-cong norm-tc-butterfly some-onb-of-norm1*)
also have ‹... = *card* (*some-onb-of* (*spectral-dec-space t′ n*))›
by *simp*
also have ‹... = *cdim* (*space-as-set* (*spectral-dec-space t′ n*))›
by (*simp add*: *some-onb-of-card*)
also have ‹... = *norm* (*spectral-dec-proj-tc t n*)›
unfolding *t′-def*
apply *transfer*
by (*metis of-real-eq-iff of-real-of-nat-eq spectral-dec-proj-def spectral-dec-proj-pos trace-Proj trace-norm-pos*)
finally show *?thesis*
by −
qed
show *?thesis*
apply (*simp add*: ∗)
by (*metis* (*no-types*, *lifting*) ‹*selfadjoint-tc t*› *norm-scaleC spectral-dec-summable-tc spectral-dec-val-tc.rep-eq summable-on-cong t′-def*)
qed
then have *2*: ‹(*λn*. ∑∞*v*∈(∗*C*) (*csqrt* (*spectral-dec-val t′ n*)) ‘ *some-onb-of* (*spectral-dec-space t′ n*).
norm (*tc-butterfly v v*)) *abs-summable-on UNIV*›
apply (*subst infsum-reindex*)
by (*auto intro*!: *inj simp*: *o-def infsum-cmult-right′ norm-mult simp del*: *real-norm-def*)
show *?thesis*
apply (*rule abs-summable-summable*)
apply (*rule abs-summable-on-Sigma-iff*[*THEN iffD2*])
using *1 2* **by** *auto*
qed
qed
have ‹((*λv*. *tc-butterfly v v*) *has-sum t*) (⋃*n*. (∗*C*) (*csqrt* (*spectral-dec-val t′ n*)) ‘ *some-onb-of* (*spectral-dec-space t′ n*))›
proof −
have ∗∗: ‹(⋃*n*. (∗*C*) (*csqrt* (*spectral-dec-val t′ n*)) ‘ *some-onb-of* (*spectral-dec-space t′ n*)) =
snd ‘ (*SIGMA n*:*UNIV*. (∗*C*) (*csqrt* (*spectral-dec-val t′ n*)) ‘ *some-onb-of* (*spectral-dec-space t′ n*))›
by *force*
have *inj*: ‹*inj-on snd* (*SIGMA n*:*UNIV*. (*λx*. *csqrt* (*spectral-dec-val t′ n*) ∗*C x*) ‘ *some-onb-of* (*spectral-dec-space t′ n*))›
proof (*rule inj-onI*)
fix *nh* **assume** *nh*: ‹*nh* ∈ (*SIGMA n*:*UNIV*. (*λx*. *csqrt* (*spectral-dec-val t′ n*) ∗*C x*) ‘ *some-onb-of* (*spectral-dec-space t′ n*))›
fix *mg* **assume** *mg*: ‹*mg* ∈ (*SIGMA m*:*UNIV*. (*λx*. *csqrt* (*spectral-dec-val t′ m*) ∗*C x*) ‘ *some-onb-of* (*spectral-dec-space t′ m*))›
assume ‹*snd nh* = *snd mg*›
from *nh* **obtain** *n h* **where** *nh′*: ‹*nh* = (*n*, *csqrt* (*spectral-dec-val t′ n*) ∗*C h*)› **and** *h*: ‹*h*

```
∈ some-onb-of (spectral-dec-space t' n)›
      by blast
    from mg obtain m g where mg': ‹mg = (m, csqrt (spectral-dec-val t' m) *C g)› and g:
‹g ∈ some-onb-of (spectral-dec-space t' m)›
      by blast
    have ‹n = m›
    proof (rule ccontr)
      assume [simp]: ‹n ≠ m›
      from h have val-not-0: ‹spectral-dec-val t' n ≠ 0›
        using some-onb-of-0 spectral-dec-space-0 by fastforce
       from ‹snd nh = snd mg› nh' mg' have eq: ‹csqrt (spectral-dec-val t' n) *C h = csqrt
(spectral-dec-val t' m) *C g›
        by simp
      from ‹n ≠ m› have ‹orthogonal-spaces (spectral-dec-space t' n) (spectral-dec-space t' m)›
        apply (rule spectral-dec-space-orthogonal[rotated −1])
            using ‹selfadjoint-tc t› by (auto intro!: trace-class-compact simp: t'-def selfad-
joint-tc.rep-eq)
      with h g have ‹is-orthogonal h g›
        using orthogonal-spaces-ccspan by fastforce
      then have ‹is-orthogonal (csqrt (spectral-dec-val t' n) *C h) (csqrt (spectral-dec-val t' m)
*C g)›
        by force
      with eq have val-h-0: ‹csqrt (spectral-dec-val t' n) *C h = 0›
        by simp
      with val-not-0 have ‹h = 0›
        by fastforce
      with h show False
        using some-onb-of-is-ortho-set
        by (auto simp: is-ortho-set-def)
    qed
    with ‹snd nh = snd mg› nh' mg' show ‹nh = mg›
      by simp
  qed
  from ∗ show ?thesis
    apply (subst ** )
    apply (rule has-sum-reindex[THEN iffD2, rotated])
    by (auto intro!: inj simp: o-def case-prod-unfold)
 qed
 then show ?thesis
   by (simp add: spectral-dec-vecs-tc.rep-eq spectral-dec-vecs-def flip: t'-def)
qed


lemma spectral-dec-vec-tc-norm-summable:
 assumes ‹t ≥ 0›
 shows ‹(λv. (norm v)^2) summable-on (spectral-dec-vecs-tc t)›
proof −
 from butterfly-spectral-dec-vec-tc-has-sum[OF assms]
 have ‹(λv. tc-butterfly v v) summable-on (spectral-dec-vecs-tc t)›
```

using *has-sum-imp-summable* **by** *blast*
then have ‹(λ*v*. *trace-tc* (*tc-butterfly v v*)) *summable-on* (*spectral-dec-vecs-tc t*)›
apply (*rule summable-on-bounded-linear*[*rotated*])
by (*simp add*: *bounded-clinear.bounded-linear*)
moreover have ∗: ‹*trace-tc* (*tc-butterfly v v*) = *of-real* ((*norm v*)^2)› **for** *v* :: *'a*
by (*metis norm-tc-butterfly norm-tc-pos power2-eq-square tc-butterfly-pos*)
ultimately have ‹(λ*v*. *complex-of-real* ((*norm v*)^2)) *summable-on* (*spectral-dec-vecs-tc t*)›
by *simp*
then show *?thesis*
by (*smt* (*verit*, *ccfv-SIG*) *norm-of-real summable-on-cong summable-on-iff-abs-summable-on-complex zero-le-power2*)
qed

12.7 More Trace-Class

lemma *finite-rank-tc-dense-aux*: ‹*closure* (*Collect finite-rank-tc* :: (*'a*::*chilbert-space*, *'a*) *trace-class set*) = *UNIV*›
proof (*intro order-top-class.top-le subsetI*)
fix *a* :: ‹(*'a*,*'a*) *trace-class*›
wlog *selfadj*: ‹*selfadjoint-tc a*› **goal** ‹*a* ∈ *closure* (*Collect finite-rank-tc*)› **generalizing** *a*
proof −
define *b c* **where** ‹*b* = *a* + *adj-tc a*› **and** ‹*c* = i ∗$_C$ (*a* − *adj-tc a*)›
have ‹*adj-tc b* = *b*›
unfolding *b-def*
apply *transfer*
by (*simp add*: *adj-plus*)
have ‹*adj-tc c* = *c*›
unfolding *c-def*
apply *transfer*
apply (*simp add*: *adj-minus*)
by (*metis minus-diff-eq scaleC-right.minus*)
have *abc*: ‹*a* = (*1*/*2*) ∗$_C$ *b* + (−i/*2*) ∗$_C$ *c*›
apply (*simp add*: *b-def c-def*)
by (*metis* (*no-types*, *lifting*) *cross3-simps*(*8*) *diff-add-cancel group-cancel.add2 scaleC-add-right scaleC-half-double*)
have ‹*b* ∈ *closure* (*Collect finite-rank-tc*)› **and** ‹*c* ∈ *closure* (*Collect finite-rank-tc*)›
using ‹*adj-tc b* = *b*› ‹*adj-tc c* = *c*› *hypothesis selfadjoint-tc-def'* **by** *auto*
with *abc* **have** ‹*a* ∈ *cspan* (*closure* (*Collect finite-rank-tc*))›
by (*metis complex-vector.span-add complex-vector.span-clauses*(*1*) *complex-vector.span-clauses*(*4*))
also have ‹... ⊆ *closure* (*cspan* (*Collect finite-rank-tc*))›
by (*simp add*: *closure-mono complex-vector.span-minimal complex-vector.span-superset*)
also have ‹... = *closure* (*Collect finite-rank-tc*)›
by (*metis Set.basic-monos*(*1*) *complex-vector.span-minimal complex-vector.span-superset csubspace-finite-rank-tc subset-antisym*)
finally show *?thesis*
by −
qed
then have ‹(λ*n*. *spectral-dec-val-tc a n* ∗$_C$ *spectral-dec-proj-tc a n*) *sums a*›
by (*simp add*: *spectral-dec-sums-tc*)

```
  moreover from selfadj
  have ‹finite-rank-tc (∑ i<n. spectral-dec-val-tc a i *C spectral-dec-proj-tc a i)› for n
    apply (induction n)
     by (auto intro!: finite-rank-tc-plus spectral-dec-proj-tc-finite-rank finite-rank-tc-scale
        simp: selfadjoint-tc-def')
  ultimately show ‹a ∈ closure (Collect finite-rank-tc)›
    unfolding sums-def closure-sequential
    apply (auto intro!: simp: sums-def closure-sequential)
    by meson
qed


lemma finite-rank-tc-dense: ‹closure (Collect finite-rank-tc :: ('a::chilbert-space, 'b::chilbert-space)
trace-class set) = UNIV›
proof −
  have ‹UNIV = closure (Collect finite-rank-tc :: ('a×'b, 'a×'b) trace-class set)›
    by (rule finite-rank-tc-dense-aux[symmetric])
  define l r and corner :: ‹('a×'b, 'a×'b) trace-class ⇒ -› where
    ‹l = cblinfun-left› and ‹r = cblinfun-right› and
    ‹corner t = compose-tcl (compose-tcr (r*) t) l› for t
  have [iff]: ‹bounded-clinear corner›
   by (auto intro: bounded-clinear-compose compose-tcl.bounded-clinear-left compose-tcr.bounded-clinear-right

        simp: corner-def[abs-def])
  have ‹UNIV = corner ‘ UNIV›
  proof (intro UNIV-eq-I range-eqI)
    fix t
    have ‹from-trace-class (corner (compose-tcl (compose-tcr r t) (l*)))
          = (r* oCL r) oCL from-trace-class t oCL (l* oCL l)›
      by (simp add: corner-def compose-tcl.rep-eq compose-tcr.rep-eq cblinfun-compose-assoc)
    also have ‹... = from-trace-class t›
      by (simp add: l-def r-def)
    finally show ‹t = corner (compose-tcl (compose-tcr r t) (l*))›
      by (metis from-trace-class-inject)
  qed
  also have ‹... = corner ‘ closure (Collect finite-rank-tc)›
    by (simp add: finite-rank-tc-dense-aux)
  also have ‹... ⊆ closure (corner ‘ Collect finite-rank-tc)›
    by (auto intro!: bounded-clinear.bounded-linear closure-bounded-linear-image-subset)
  also have ‹... ⊆ closure (Collect finite-rank-tc)›
  proof (intro closure-mono subsetI CollectI)
    fix t assume ‹t ∈ corner ‘ Collect finite-rank-tc›
    then obtain u where ‹finite-rank-tc u› and tu: ‹t = corner u›
      by blast
    show ‹finite-rank-tc t›
      using ‹finite-rank-tc u›
      by (auto intro!: finite-rank-compose-right[of - l] finite-rank-compose-left[of - ‹r*›]
          simp add: corner-def tu finite-rank-tc.rep-eq compose-tcl.rep-eq compose-tcr.rep-eq)
  qed
```

```
  finally show ?thesis
    by blast
qed


hide-fact finite-rank-tc-dense-aux


lemma onb-butterflies-span-trace-class:
  fixes A :: ‹'a::chilbert-space set› and B :: ‹'b::chilbert-space set›
  assumes ‹is-onb A› and ‹is-onb B›
  shows ‹ccspan ((λ(x, y). tc-butterfly x y) ‘ (A×B)) = ⊤›
proof −
  have ‹closure (cspan ((λ(x, y). tc-butterfly x y) ‘ (A×B))) ⊇ Collect rank1-tc›
  proof (rule subsetI)
    — This subproof is almost identical to the corresponding one in finite-rank-dense-compact,
and lengthy. Can they be merged into one subproof?
    fix x :: ‹('b, 'a) trace-class› assume ‹x ∈ Collect rank1-tc›
    then obtain a b where xab: ‹x = tc-butterfly a b›
      apply transfer using rank1-iff-butterfly by fastforce
    define f where ‹f F G = tc-butterfly (Proj (ccspan F) a) (Proj (ccspan G) b)› for F G
    have lim: ‹(case-prod f ⟶⟶ x) (finite-subsets-at-top A ×_F finite-subsets-at-top B)›
    proof (rule tendstoI, subst dist-norm)
      fix e :: real assume ‹e > 0›
      define d where ‹d = (if norm a = 0 ∧ norm b = 0 then 1
                                  else e / (max (norm a) (norm b)) / 4)›
      have d: ‹norm a * d + norm a * d + norm b * d < e›
      proof −
        have ‹norm a * d ≤ e/4›
          using ‹e > 0› apply (auto simp: d-def)
           apply (simp add: divide-le-eq)
         by (smt (z3) Extra-Ordered-Fields.mult-sign-intros(3) ‹0 < e› antisym-conv divide-le-eq
less-imp-le linordered-field-class.mult-imp-div-pos-le mult-left-mono nice-ordered-field-class.dense-le
nice-ordered-field-class.divide-nonneg-neg nice-ordered-field-class.divide-nonpos-pos nle-le nonzero-mult-div-cancel-left
norm-imp-pos-and-ge ordered-field-class.sign-simps(5) split-mult-pos-le)
        moreover have ‹norm b * d ≤ e/4›
          using ‹e > 0› apply (auto simp: d-def)
           apply (simp add: divide-le-eq)
         by (smt (verit) linordered-field-class.mult-imp-div-pos-le mult-left-mono norm-le-zero-iff
ordered-field-class.sign-simps(5))
        ultimately have ‹norm a * d + norm a * d + norm b * d ≤ 3 * e / 4›
          by linarith
        also have ‹... < e›
          by (simp add: ‹0 < e›)
        finally show ?thesis
          by −
      qed
      have [simp]: ‹d > 0›
        using ‹e > 0› apply (auto simp: d-def)
```

apply (*smt* (*verit*, *best*) *nice-ordered-field-class.divide-pos-pos norm-eq-zero norm-not-less-zero*)
by (*smt* (*verit*) *linordered-field-class.divide-pos-pos zero-less-norm-iff*)
from *Proj-onb-limit*[**where** ψ=*a*, *OF assms*(*1*)]
have ‹$\forall_F$ *F in finite-subsets-at-top A. norm* (*Proj* (*ccspan F*) *a* − *a*) < *d*›
by (*metis Lim-null* ‹*0* < *d*› *order-tendstoD*(*2*) *tendsto-norm-zero-iff*)
moreover from *Proj-onb-limit*[**where** ψ=*b*, *OF assms*(*2*)]
have ‹$\forall_F$ *G in finite-subsets-at-top B. norm* (*Proj* (*ccspan G*) *b* − *b*) < *d*›
by (*metis Lim-null* ‹*0* < *d*› *order-tendstoD*(*2*) *tendsto-norm-zero-iff*)
ultimately have *FG-close*: ‹$\forall_F$ (*F*,*G*) *in finite-subsets-at-top A* ×$_F$ *finite-subsets-at-top B*.
norm (*Proj* (*ccspan F*) *a* − *a*) < *d* ∧ *norm* (*Proj* (*ccspan G*) *b* − *b*) < *d*›
unfolding *case-prod-beta*
by (*rule eventually-prodI*)
have *fFG-dist*: ‹*norm* (*f F G* − *x*) < *e*›
if ‹*norm* (*Proj* (*ccspan F*) *a* − *a*) < *d*› **and** ‹*norm* (*Proj* (*ccspan G*) *b* − *b*) < *d*›
and ‹*F* ⊆ *A*› **and** ‹*G* ⊆ *B*› **for** *F G*
proof −
have *a-split*: ‹*a* = *Proj* (*ccspan F*) *a* + *Proj* (*ccspan* (*A*−*F*)) *a*›
using *assms* **apply** (*simp add*: *is-onb-def is-ortho-set-def that Proj-orthog-ccspan-union flip*: *cblinfun.add-left*)
apply (*subst Proj-orthog-ccspan-union*[*symmetric*])
apply (*metis DiffD1 DiffD2 in-mono that*(*3*))
using ‹*F* ⊆ *A*› **by** (*auto intro*!: *simp*: *Un-absorb1*)
have *b-split*: ‹*b* = *Proj* (*ccspan G*) *b* + *Proj* (*ccspan* (*B*−*G*)) *b*›
using *assms* **apply** (*simp add*: *is-onb-def is-ortho-set-def that Proj-orthog-ccspan-union flip*: *cblinfun.add-left*)
apply (*subst Proj-orthog-ccspan-union*[*symmetric*])
apply (*metis DiffD1 DiffD2 in-mono that*(*4*))
using ‹*G* ⊆ *B*› **by** (*auto intro*!: *simp*: *Un-absorb1*)
have *n1*: ‹*norm* (*f F* (*B*−*G*)) ≤ *norm a* ∗ *d*› **for** *F*
proof −
have ‹*norm* (*f F* (*B*−*G*)) ≤ *norm a* ∗ *norm* (*Proj* (*ccspan* (*B*−*G*)) *b*)›
by (*auto intro*!: *mult-right-mono is-Proj-reduces-norm simp add*: *f-def norm-tc-butterfly*)
also have ‹... ≤ *norm a* ∗ *norm* (*Proj* (*ccspan G*) *b* − *b*)›
by (*metis add-diff-cancel-left*′ *b-split less-eq-real-def norm-minus-commute*)
also have ‹... ≤ *norm a* ∗ *d*›
by (*meson less-eq-real-def mult-left-mono norm-ge-zero that*(*2*))
finally show *?thesis*
by −
qed
have *n2*: ‹*norm* (*f* (*A*−*F*) *G*) ≤ *norm b* ∗ *d*› **for** *G*
proof −
have ‹*norm* (*f* (*A*−*F*) *G*) ≤ *norm b* ∗ *norm* (*Proj* (*ccspan* (*A*−*F*)) *a*)›
by (*auto intro*!: *mult-right-mono is-Proj-reduces-norm simp add*: *f-def norm-tc-butterfly mult.commute*)
also have ‹... ≤ *norm b* ∗ *norm* (*Proj* (*ccspan F*) *a* − *a*)›
by (*smt* (*verit*, *best*) *a-split add-diff-cancel-left*′ *minus-diff-eq norm-minus-cancel*)
also have ‹... ≤ *norm b* ∗ *d*›
by (*meson less-eq-real-def mult-left-mono norm-ge-zero that*(*1*))

finally show *?thesis*
by −
qed
have ‹*norm* (*f F G* − *x*) = *norm* (− *f F* (*B*−*G*) − *f* (*A*−*F*) (*B*−*G*) − *f* (*A*−*F*) *G*)›
unfolding *xab*
apply (*subst a-split*, *subst b-split*)
by (*simp add*: *f-def tc-butterfly-add-right tc-butterfly-add-left*)
also have ‹... ≤ *norm* (*f F* (*B*−*G*)) + *norm* (*f* (*A*−*F*) (*B*−*G*)) + *norm* (*f* (*A*−*F*) *G*)›
by (*smt* (*verit*, *best*) *norm-minus-cancel norm-triangle-ineq4*)
also have ‹... ≤ *norm a* ∗ *d* + *norm a* ∗ *d* + *norm b* ∗ *d*›
using *n1 n2*
by (*meson add-mono-thms-linordered-semiring*(*1*))
also have ‹... < *e*›
by (*fact d*)
finally show *?thesis*
by −
qed
show ‹∀$_F$ *FG in finite-subsets-at-top A* ×$_F$ *finite-subsets-at-top B. norm* (*case-prod f FG* − *x*) < *e*›
apply (*rule eventually-elim2*)
apply (*rule eventually-prodI*[**where** *P*=‹λ*F. finite F* ∧ *F* ⊆ *A*› **and** *Q*=‹λ*G. finite G* ∧ *G* ⊆ *B*›])
apply *auto*[*2*]
apply (*rule FG-close*)
using *fFG-dist* **by** *fastforce*
qed
have *nontriv*: ‹*finite-subsets-at-top A* ×$_F$ *finite-subsets-at-top B* ≠ ⊥›
by (*simp add*: *prod-filter-eq-bot*)
have *inside*: ‹∀$_F$ *x in finite-subsets-at-top A* ×$_F$ *finite-subsets-at-top B.*
case-prod f x ∈ *cspan* ((λ(ξ,η). *tc-butterfly* ξ η) ‘ (*A* × *B*))›
proof (*rule eventually-mp*[**where** *P*=‹λ(*F*,*G*). *finite F* ∧ *finite G*›])
show ‹∀$_F$ (*F*,*G*) *in finite-subsets-at-top A* ×$_F$ *finite-subsets-at-top B. finite F* ∧ *finite G*›
by (*smt* (*verit*) *case-prod-conv eventually-finite-subsets-at-top-weakI eventually-prod-filter*)
have *f-in-span*: ‹*f F G* ∈ *cspan* ((λ(ξ,η). *tc-butterfly* ξ η) ‘ (*A* × *B*))› **if** [*simp*]: ‹*finite F*› ‹*finite G*› **and** ‹*F* ⊆ *A*› ‹*G* ⊆ *B*› **for** *F G*
proof −
have ‹*Proj* (*ccspan F*) *a* ∈ *cspan F*›
by (*metis Proj-range cblinfun-apply-in-image ccspan-finite that*(*1*))
then obtain *r* **where** *ProjFsum*: ‹*Proj* (*ccspan F*) *a* = (∑ *x*∈*F*. *r x* ∗$_C$ *x*)›
apply *atomize-elim*
using *complex-vector.span-finite*[*OF* ‹*finite F*›]
by *auto*
have ‹*Proj* (*ccspan G*) *b* ∈ *cspan G*›
by (*metis Proj-range cblinfun-apply-in-image ccspan-finite that*(*2*))
then obtain *s* **where** *ProjGsum*: ‹*Proj* (*ccspan G*) *b* = (∑ *x*∈*G*. *s x* ∗$_C$ *x*)›
apply *atomize-elim*
using *complex-vector.span-finite*[*OF* ‹*finite G*›]
by *auto*
have ‹*tc-butterfly* ξ η ∈ (λ(ξ, η). *tc-butterfly* ξ η) ‘ (*A* × *B*)›

if ‹$\eta \in G$› **and** ‹$\xi \in F$› **for** η ξ
using ‹$F \subseteq A$› ‹$G \subseteq B$› *that* **by** (*auto intro*!: *pair-imageI*)
then show *?thesis*
by (*auto intro*!: *complex-vector.span-sum complex-vector.span-scale*
intro: *complex-vector.span-base*[**where** *a*=‹*tc-butterfly - -*›]
simp add: *f-def ProjFsum ProjGsum tc-butterfly-sum-left tc-butterfly-sum-right*)
qed
show ‹$\forall_{F}$ *x in finite-subsets-at-top A* $\times_{F}$ *finite-subsets-at-top B*.
(*case x of* (*F*, *G*) ⇒ *finite F* ∧ *finite G*) ⟶ (*case x of* (*F*, *G*) ⇒ *f F G*) ∈ *cspan* ((λ(ξ, η). *tc-butterfly* ξ η) ‘ (*A* × *B*))›
apply (*rule eventually-mono*)
apply (*rule eventually-prodI*[**where** *P*=‹λ*F*. *finite F* ∧ *F* ⊆ *A*› **and** *Q*=‹λ*G*. *finite G* ∧ *G* ⊆ *B*›])
by (*auto intro*!: *f-in-span*)
qed
show ‹*x* ∈ *closure* (*cspan* ((λ(ξ, η). *tc-butterfly* ξ η) ‘ (*A* × *B*)))›
using *lim nontriv inside* **by** (*rule limit-in-closure*)
qed
moreover have ‹*cspan* (*Collect rank1-tc* :: (′*b*,′*a*) *trace-class set*) = *Collect finite-rank-tc*›
using *finite-rank-tc-def* ′ **by** *fastforce*
moreover have ‹*closure* (*Collect finite-rank-tc* :: (′*b*,′*a*) *trace-class set*) = *UNIV*›
by (*rule finite-rank-tc-dense*)
ultimately have ‹*closure* (*cspan* ((λ(*x*, *y*). *tc-butterfly x y*) ‘ (*A*×*B*))) ⊇ *UNIV*›
by (*smt* (*verit*, *del-insts*) *Un-UNIV-left closed-sum-closure-left closed-sum-cspan closure-closure closure-is-csubspace complex-vector.span-eq-iff complex-vector.subspace-span subset-Un-eq*)
then show *?thesis*
by (*metis ccspan.abs-eq ccspan-UNIV closure-UNIV complex-vector.span-UNIV top.extremum-uniqueI*)
qed

lemma *separating-set-tc-butterfly*: ‹*separating-set bounded-clinear* ((λ(*g*,*h*). *tc-butterfly g h*) ‘ (*UNIV* × *UNIV*))›
apply (*rule separating-set-mono*[**where** *S*=‹(λ(*g*, *h*). *tc-butterfly g h*) ‘ (*some-chilbert-basis* × *some-chilbert-basis*)›])
by (*auto intro*!: *separating-set-bounded-clinear-dense onb-butterflies-span-trace-class*)

lemma *separating-set-tc-butterfly-nested*:
assumes ‹*separating-set* (*bounded-clinear* :: (- ⇒ ′*c*::*complex-normed-vector*) ⇒ -) *A*›
assumes ‹*separating-set* (*bounded-clinear* :: (- ⇒ ′*c conjugate-space*) ⇒ -) *B*›
shows ‹*separating-set* (*bounded-clinear* :: (- ⇒ ′*c*) ⇒ -) ((λ(*g*,*h*). *tc-butterfly g h*) ‘ (*A* × *B*))›
proof −
from *separating-set-tc-butterfly*
have ‹*separating-set bounded-clinear* ((λ(*g*,*h*). *tc-butterfly g h*) ‘ *prod.swap* ‘ (*UNIV* × *UNIV*))›
by *simp*
then have ‹*separating-set bounded-clinear* ((λ(*g*,*h*). *tc-butterfly h g*) ‘ (*UNIV* × *UNIV*))›
unfolding *image-image* **by** *simp*
then have ‹*separating-set* (*bounded-clinear* :: (- ⇒ ′*c*) ⇒ -) ((λ(*g*,*h*). *tc-butterfly h g*) ‘ (*B* × *A*))›
apply (*rule separating-set-bounded-sesquilinear-nested*)
apply (*rule bounded-sesquilinear-tc-butterfly*)

using *assms* **by** *auto*
then have ‹*separating-set* (*bounded-clinear* :: (- ⇒ 'c) ⇒ -) ((λ(g,h). *tc-butterfly h g*) ‘ *prod.swap* ‘ (*A* × *B*))›
by (*smt* (*verit*, *del-insts*) *SigmaE SigmaI eq-from-separatingI image-iff pair-in-swap-image separating-setI*)
then show *?thesis*
unfolding *image-image* **by** *simp*
qed

end

13 *Weak-Star-Topology* – Weak* topology on complex bounded operators

theory *Weak-Star-Topology*
imports *Trace-Class Weak-Operator-Topology Misc-Tensor-Product-TTS*
begin

definition *weak-star-topology* :: ‹('a::*chilbert-space* ⇒$_{CL}$ 'b::*chilbert-space*) *topology*›
where ‹*weak-star-topology* = *pullback-topology UNIV* (λx. λt∈*Collect trace-class*. *trace* (*t* o_{CL} *x*))
(*product-topology* (λ-. *euclidean*) (*Collect trace-class*))›

lemma *open-map-product-topology-reindex*:
fixes π :: ‹'b ⇒ 'a›
assumes *bij-π*: ‹*bij-betw* π *B A*› **and** *ST*: ‹⋀x. x∈*B* ⟹ *S x* = *T* (π *x*)›
assumes *g-def*: ‹⋀f. *g f* = *restrict* (*f o* π) *B*›
shows ‹*open-map* (*product-topology T A*) (*product-topology S B*) *g*›
proof −
define π' *g'* **where** ‹π' = *inv-into B* π› **and** ‹*g' f* = *restrict* (*f* ∘ π') *A*› **for** *f* :: ‹'b ⇒ 'c›
have *bij-g*: ‹*bij-betw g* (Pi_E *A V*) (Pi_E *B* (*V* ∘ π))› **for** *V*
apply (*rule bij-betw-byWitness*[**where** *f'*=*g'*])
subgoal
unfolding *g'-def g-def* π'*-def*
by (*smt* (*verit*, *best*) *PiE-restrict bij-π bij-betw-imp-surj-on bij-betw-inv-into-right comp-eq-dest-lhs inv-into-into restrict-def restrict-ext*)
subgoal
unfolding *g'-def g-def* π'*-def*
by (*smt* (*verit*, *ccfv-SIG*) *PiE-restrict bij-π bij-betwE bij-betw-inv-into-left comp-apply restrict-apply restrict-ext*)
subgoal
unfolding *g'-def g-def* π'*-def*
using *PiE-mem bij-π bij-betw-imp-surj-on* **by** *fastforce*
subgoal
unfolding *g'-def g-def* π'*-def*

by (*smt* (*verit*, *best*) *PiE-mem bij-π bij-betw-iff-bijections bij-betw-inv-into-left comp-def image-subset-iff restrict-PiE-iff*)
done
have *open-gU*: ‹*openin* (*product-topology S B*) (*g* ‘ *U*)› **if** ‹*openin* (*product-topology T A*) *U*› **for** *U*
proof −
from *product-topology-open-contains-basis*[*OF that*]
obtain *V* **where** *xAV*: ‹*x* ∈ *PiE A* (*V x*)› **and** *openV*: ‹*openin* (*T a*) (*V x a*)› **and** *finiteV*: ‹*finite* {*a*. *V x a* ≠ *topspace* (*T a*)}›
and *AVU*: ‹*Pi*$_E$ *A* (*V x*) ⊆ *U*› **if** ‹*x* ∈ *U*› **for** *x a*
apply *atomize-elim*
apply (*rule choice4*)
by *meson*
define *V′* **where** ‹*V′ x b* = (*if b* ∈ *B then V x* (*π b*) *else topspace* (*S b*))› **for** *b x*
have *PiEV′*: ‹*Pi*$_E$ *B* (*V x* ∘ *π*) = *Pi*$_E$ *B* (*V′ x*)› **for** *x*
by (*metis* (*mono-tags*, *opaque-lifting*) *PiE-cong V′-def comp-def*)
from *xAV AVU* **have** *AVU′*: ‹(⋃*x*∈*U*. *Pi*$_E$ *A* (*V x*)) = *U*›
by *blast*
have *openVb*: ‹*openin* (*S b*) (*V′ x b*)› **if** [*simp*]: ‹*x* ∈ *U*› **for** *x b*
by (*auto simp*: *ST V′-def intro*!: *openV*)
have ‹*bij-betw π′* {*a*∈*A*. *V x a* ≠ *topspace* (*T a*)} {*b*∈*B*. (*V x* ∘ *π*) *b* ≠ *topspace* (*S b*)}› **for** *x*
apply (*rule bij-betw-byWitness*[**where** *f′*=*π*])
apply *simp*
apply (*metis π′-def bij-π bij-betw-inv-into-right*)
using *π′-def bij-π bij-betw-imp-inj-on* **apply** *fastforce*
apply (*smt* (*verit*, *best*) *ST π′-def bij-π bij-betw-imp-surj-on comp-apply f-inv-into-f image-Collect-subsetI inv-into-into mem-Collect-eq*)
using *ST bij-π bij-betwE* **by** *fastforce*

then have ‹*finite* {*b*∈*B*. (*V x* ∘ *π*) *b* ≠ *topspace* (*S b*)}› **if** ‹*x* ∈ *U*› **for** *x*
apply (*rule bij-betw-finite*[*THEN iffD1*])
using *that finiteV*
by *simp*
also have ‹{*b*∈*B*. (*V x* ∘ *π*) *b* ≠ *topspace* (*S b*)} = {*b*. *V′ x b* ≠ *topspace* (*S b*)}› **if** ‹*x* ∈ *U*› **for** *x*
by (*auto simp*: *V′-def*)
finally have *finiteVπ*: ‹*finite* {*b*. *V′ x b* ≠ *topspace* (*S b*)}› **if** ‹*x* ∈ *U*› **for** *x*
using *that* **by** −
from *openVb finiteVπ*
have ‹*openin* (*product-topology S B*) (*Pi*$_E$ *B* (*V′ x*))› **if** [*simp*]: ‹*x* ∈ *U*› **for** *x*
by (*auto intro*!: *product-topology-basis*)
with *bij-g PiEV′* **have** ‹*openin* (*product-topology S B*) (*g* ‘ (*Pi*$_E$ *A* (*V x*)))› **if** ‹*x* ∈ *U*› **for** *x*
by (*metis bij-betw-imp-surj-on that*)
then have ‹*openin* (*product-topology S B*) (⋃*x*∈*U*. (*g* ‘ (*Pi*$_E$ *A* (*V x*))))›
by *blast*
with *AVU′* **show** ‹*openin* (*product-topology S B*) (*g* ‘ *U*)›
by (*metis image-UN*)
qed

show ‹*open-map* (*product-topology T A*) (*product-topology S B*) *g*›
by (*simp add*: *open-gU open-map-def*)
qed

lemma *homeomorphic-map-product-topology-reindex*:
fixes π :: ‹$'b \Rightarrow 'a$›
assumes *big-π*: ‹*bij-betw π B A*› **and** *ST*: ‹$\bigwedge x.\ x \in B \Longrightarrow S\ x = T\ (\pi\ x)$›
assumes *g-def*: ‹$\bigwedge f.\ g\ f = restrict\ (f\ o\ \pi)\ B$›
shows ‹*homeomorphic-map* (*product-topology T A*) (*product-topology S B*) *g*›
proof (*rule bijective-open-imp-homeomorphic-map*)
show *open-map*: ‹*open-map* (*product-topology T A*) (*product-topology S B*) *g*›
using *assms* **by** (*rule open-map-product-topology-reindex*)
define π' g' **where** ‹$\pi' = inv\text{-}into\ B\ \pi$› **and** ‹$g'\ f = restrict\ (f \circ \pi')\ A$› **for** f :: ‹$'b \Rightarrow 'c$›
have ‹*bij-betw* π' *A B*›
by (*simp add*: π'*-def big-π bij-betw-inv-into*)

have *l1*: ‹$x \in (\lambda x.\ restrict\ (x \circ \pi)\ B)\ `\ (\Pi_E\ i \in A.\ topspace\ (T\ i))$› **if** ‹$x \in (\Pi_E\ i \in B.\ topspace\ (S\ i))$› **for** *x*
proof −
have ‹$g'\ x \in (\Pi_E\ i \in A.\ topspace\ (T\ i))$›
by (*smt* (*z3*) g'*-def PiE-mem* π'*-def assms*(*1*) *assms*(*2*) *bij-betw-imp-surj-on bij-betw-inv-into-right comp-apply inv-into-into restrict-PiE-iff that*)
moreover have ‹$x = restrict\ (g'\ x \circ \pi)\ B$›
by (*smt* (*verit*) *PiE-restrict* π'*-def assms*(*1*) *bij-betwE bij-betw-inv-into-left comp-apply restrict-apply restrict-ext that* g'*-def*)
ultimately show *?thesis*
by (*intro rev-image-eqI*)
qed
show *topspace*: ‹$g\ `\ topspace\ (product\text{-}topology\ T\ A) = topspace\ (product\text{-}topology\ S\ B)$›
using *l1 assms* **unfolding** *g-def* [*abs-def*] *topspace-product-topology*
by (*auto simp*: *bij-betw-def*)

show ‹*inj-on g* (*topspace* (*product-topology T A*))›
apply (*simp add*: *g-def*[*abs-def*])
by (*smt* (*verit*) *PiE-ext assms*(*1*) *bij-betw-iff-bijections comp-apply inj-on-def restrict-apply*′)

have *open-map-g*′: ‹*open-map* (*product-topology S B*) (*product-topology T A*) g'›
using ‹*bij-betw* π' *A B*› **apply** (*rule open-map-product-topology-reindex*)
apply (*metis ST* π'*-def big-π bij-betw-imp-surj-on bij-betw-inv-into-right inv-into-into*)
using g'*-def* **by** *blast*
have $g'g$: ‹$g'\ (g\ x) = x$› **if** ‹$x \in topspace\ (product\text{-}topology\ T\ A)$› **for** *x*
using *that* **unfolding** g'*-def g-def topspace-product-topology*
by (*smt* (*verit*) *PiE-restrict* ‹*bij-betw* π' *A B*› π'*-def big-π bij-betwE bij-betw-inv-into-right comp-def restrict-apply*′ *restrict-ext*)
have *gg*′: ‹$g\ (g'\ x) = x$› **if** ‹$x \in topspace\ (product\text{-}topology\ S\ B)$› **for** *x*
unfolding g'*-def g-def*
by (*metis* (*no-types*, *lifting*) g'*-def f-inv-into-f* $g'g$ *g-def inv-into-into that topspace*)

from *open-map-g′*
have ‹*openin* (*product-topology T A*) (*g′ ‘ U*)› **if** ‹*openin* (*product-topology S B*) *U*› **for** *U*
using *open-map-def that* **by** *blast*
also have ‹*g′ ‘ U* = (*g* −‘ *U*) ∩ (*topspace* (*product-topology T A*))› **if** ‹*openin* (*product-topology S B*) *U*› **for** *U*
proof −
from *that*
have *U-top*: ‹*U* ⊆ *topspace* (*product-topology S B*)›
using *openin-subset* **by** *blast*
from *topspace*
have *topspace′*: ‹*topspace* (*product-topology T A*) = *g′ ‘ topspace* (*product-topology S B*)›
by (*metis bij-betw-byWitness bij-betw-def calculation g′g gg′ openin-subset openin-topspace*)
show *?thesis*
unfolding *topspace′*
using *U-top gg′*
by *auto*
qed
finally have *open-gU2*: ‹*openin* (*product-topology T A*) ((*g* −‘ *U*) ∩ (*topspace* (*product-topology T A*)))›
if ‹*openin* (*product-topology S B*) *U*› **for** *U*
using *that* **by** *blast*

then show ‹*continuous-map* (*product-topology T A*) (*product-topology S B*) *g*›
by (*smt* (*verit*, *best*) *g′g image-iff open-eq-continuous-inverse-map open-map-g′ topspace*)
qed

lemma *weak-star-topology-def′*:
‹*weak-star-topology* = *pullback-topology UNIV* (λ*x t*. *trace* (*from-trace-class t* o_{CL} *x*)) *euclidean*›
proof −
define *f g* **where** ‹*f x* = (λ*t*∈*Collect trace-class*. *trace* (*t* o_{CL} *x*))› **and** ‹*g f′* = *f′ o from-trace-class*›
for *x* :: ‹*′a* ⇒$_{CL}$ *′b*› **and** *f′* :: ‹*′b* ⇒$_{CL}$ *′a* ⇒ *complex*›
have ‹*homeomorphic-map* (*product-topology* (λ-. *euclidean*) (*Collect trace-class*)) (*product-topology* (λ-. *euclidean*) *UNIV*) *g*›
unfolding *g-def*[*abs-def*]
apply (*rule homeomorphic-map-product-topology-reindex*[**where** π=*from-trace-class*])
subgoal
by (*smt* (*verit*, *best*) *UNIV-I bij-betwI′ from-trace-class from-trace-class-cases from-trace-class-inject*)
by *auto*
then have *homeo-g*: ‹*homeomorphic-map* (*product-topology* (λ-. *euclidean*) (*Collect trace-class*)) *euclidean g*›
by (*simp add*: *euclidean-product-topology*)
have ‹*weak-star-topology* = *pullback-topology UNIV f* (*product-topology* (λ-. *euclidean*) (*Collect trace-class*))›
by (*simp add*: *weak-star-topology-def pullback-topology-homeo-cong homeo-g f-def*[*abs-def*])
also have ‹... = *pullback-topology UNIV* (*g o f*) *euclidean*›
by (*subst pullback-topology-homeo-cong*)
(*auto simp add*: *homeo-g f-def*[*abs-def*] *split*: *if-splits*)
also have ‹... = *pullback-topology UNIV* (λ*x t*. *trace* (*from-trace-class t* o_{CL} *x*)) *euclidean*›

by (*auto simp*: *f-def*[*abs-def*] *g-def*[*abs-def*] *o-def*)
finally show *?thesis*
by −
qed

lemma *weak-star-topology-topspace*[*simp*]:
topspace weak-star-topology = *UNIV*
unfolding *weak-star-topology-def topspace-pullback-topology topspace-euclidean* **by** *auto*

lemma *weak-star-topology-basis'*:
fixes *f*::('*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*) **and** *U*::'*i* ⇒ *complex set* **and** *t*::'*i* ⇒ ('*b*,'*a*) *trace-class*
assumes *finite I* ⋀*i*. *i* ∈ *I* ⟹ *open* (*U i*)
shows *openin weak-star-topology* {*f*. ∀ *i*∈*I*. *trace* (*from-trace-class* (*t i*) *o*$_{CL}$ *f*) ∈ *U i*}
proof −
have *1*: *open* {*g*. ∀ *i*∈*I*. *g* (*t i*) ∈ *U i*}
using *assms* **by** (*rule product-topology-basis'*)
show *?thesis*
unfolding *weak-star-topology-def'*
apply (*subst openin-pullback-topology*)
apply (*intro exI conjI*)
using *1* **by** *auto*
qed

lemma *weak-star-topology-basis*:
fixes *f*::('*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*) **and** *U*::'*i* ⇒ *complex set* **and** *t*::'*i* ⇒ ('*b* ⇒$_{CL}$ '*a*)
assumes *finite I* ⋀*i*. *i* ∈ *I* ⟹ *open* (*U i*)
assumes *tc*: ‹⋀*i*. *i* ∈ *I* ⟹ *trace-class* (*t i*)›
shows *openin weak-star-topology* {*f*. ∀ *i*∈*I*. *trace* (*t i o*$_{CL}$ *f*) ∈ *U i*}
proof −
obtain *t'* **where** *tt'*: ‹*t i* = *from-trace-class* (*t' i*)› **if** ‹*i* ∈ *I*› **for** *i*
by (*atomize-elim*, *rule choice*) (*use tc from-trace-class-cases* **in** *blast*)
show *?thesis*
using *assms* **by** (*auto simp*: *tt' o-def intro*!: *weak-star-topology-basis'*)
qed

lemma *wot-weaker-than-weak-star*:
continuous-map weak-star-topology cweak-operator-topology (λ*f*. *f*)
unfolding *weak-star-topology-def cweak-operator-topology-def*
proof (*rule continuous-map-pullback-both*)
define *g'* :: ‹('*b* ⇒$_{CL}$ '*a* ⇒ *complex*) ⇒ '*b* × '*a* ⇒ *complex*› **where**
‹*g' f* = (λ(*x*,*y*). *f* (*butterfly y x*))› **for** *f*
show ‹(λ*x*. λ*t*∈*Collect trace-class*. *trace* (*t o*$_{CL}$ *x*)) −‘ *topspace* (*product-topology* (λ-. *euclidean*) (*Collect trace-class*)) ∩ *UNIV*
⊆ (λ*f*. *f*) −‘ *UNIV*›
by *simp*
show ‹*g'* (λ*t*∈*Collect trace-class*. *trace* (*t o*$_{CL}$ *x*)) = (λ(*xa*, *y*). *xa* ⋆$_C$ (*x* ∗$_V$ *y*))›
if ‹(λ*t*∈*Collect trace-class*. *trace* (*t o*$_{CL}$ *x*)) ∈ *topspace* (*product-topology* (λ-. *euclidean*)

(*Collect trace-class*))›
  for *x*
  by (*auto intro*!: *ext simp*: *g′-def trace-butterfly-comp*)
 show ‹*continuous-map* (*product-topology* (λ-. *euclidean*) (*Collect trace-class*)) *euclidean g′*›
  apply (*subst euclidean-product-topology*[*symmetric*])
  apply (*rule continuous-map-coordinatewise-then-product*)
  subgoal for *i*
   unfolding *g′-def case-prod-unfold*
   by (*metis continuous-map-product-projection mem-Collect-eq trace-class-butterfly*)
  subgoal
   by (*auto simp*: *g′-def*[*abs-def*])
  done
qed

lemma *wot-weaker-than-weak-star′*:
 ‹*openin cweak-operator-topology U* ⟹ *openin weak-star-topology U*›
 using *wot-weaker-than-weak-star*[**where** *′a*=*′a* **and** *′b*=*′b*]
 by (*auto simp*: *continuous-map-def weak-star-topology-topspace*)

lemma *weak-star-topology-continuous-duality′*:
 shows *continuous-map weak-star-topology euclidean* (λ*x*. *trace* (*from-trace-class t* o_{CL} *x*))
proof −
 have *continuous-map weak-star-topology euclidean* ((λ*f*. *f t*) *o* (λ*x t*. *trace* (*from-trace-class t* o_{CL} *x*)))
  unfolding *weak-star-topology-def′* **apply** (*rule continuous-map-pullback*)
  using *continuous-on-product-coordinates* **by** *fastforce*
 then show *?thesis* **unfolding** *comp-def* **by** *simp*
qed

lemma *weak-star-topology-continuous-duality*:
 assumes ‹*trace-class t*›
 shows *continuous-map weak-star-topology euclidean* (λ*x*. *trace* (*t* o_{CL} *x*))
 by (*metis assms from-trace-class-cases mem-Collect-eq weak-star-topology-continuous-duality′*)

lemma *continuous-on-weak-star-topo-iff-coordinatewise*:
 fixes *f* :: ‹*′a* ⇒ *′b*::*chilbert-space* ⇒$_{CL}$ *′c*::*chilbert-space*›
 shows *continuous-map T weak-star-topology f*
  ⟷ (∀ *t*. *trace-class t* ⟶ *continuous-map T euclidean* (λ*x*. *trace* (*t* o_{CL} *f x*)))
proof (*intro iffI allI impI*)
 fix *t* :: ‹*′c* ⇒$_{CL}$ *′b*›
 assume ‹*trace-class t*›
 assume *continuous-map T weak-star-topology f*
 with *continuous-map-compose*[*OF this weak-star-topology-continuous-duality*, *OF* ‹*trace-class t*›]
 have *continuous-map T euclidean* ((λ*x*. *trace* (*t* o_{CL} *x*)) *o f*)
  by *simp*
 then show *continuous-map T euclidean* (λ*x*. *trace* (*t* o_{CL} *f x*))
  unfolding *comp-def* **by** *auto*
next

assume ‹∀ *t*. *trace-class t* ⟶ *continuous-map T euclidean* (λ*x*. *trace* (*t* o_{CL} *f x*))›
then have ‹*continuous-map T euclidean* (λ*x*. *trace* (*from-trace-class t* o_{CL} *f x*))› **for** *t*
by *auto*
then have ∗: *continuous-map T euclidean* ((λ*x t*. *trace* (*from-trace-class t* o_{CL} *x*)) *o f*)
by (*auto simp flip*: *euclidean-product-topology simp*: *o-def*)
show *continuous-map T weak-star-topology f*
unfolding *weak-star-topology-def'*
apply (*rule continuous-map-pullback'*)
by (*auto simp add*: ∗)
qed

lemma *weak-star-topology-weaker-than-euclidean*:
continuous-map euclidean weak-star-topology (λ*f*. *f*)
apply (*subst continuous-on-weak-star-topo-iff-coordinatewise*)
by (*auto intro*!: *linear-continuous-on bounded-clinear.bounded-linear bounded-clinear-trace-duality*)

typedef (**overloaded**) ('*a*,'*b*) *cblinfun-weak-star* = ‹*UNIV* :: ('*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector*) *set*›
morphisms *from-weak-star to-weak-star* ..
setup-lifting *type-definition-cblinfun-weak-star*

lift-definition *id-weak-star* :: ‹('*a*::*complex-normed-vector*, '*a*) *cblinfun-weak-star*› **is** *id-cblinfun*
.

instantiation *cblinfun-weak-star* :: (*complex-normed-vector*, *complex-normed-vector*) *complex-vector*
begin
lift-definition *scaleC-cblinfun-weak-star* :: ‹*complex* ⇒ ('*a*, '*b*) *cblinfun-weak-star* ⇒ ('*a*, '*b*) *cblinfun-weak-star*›
is ‹*scaleC*› .
lift-definition *uminus-cblinfun-weak-star* :: ‹('*a*, '*b*) *cblinfun-weak-star* ⇒ ('*a*, '*b*) *cblinfun-weak-star*›
is *uminus* .
lift-definition *zero-cblinfun-weak-star* :: ‹('*a*, '*b*) *cblinfun-weak-star*› **is** *0* .
lift-definition *minus-cblinfun-weak-star* :: ‹('*a*, '*b*) *cblinfun-weak-star* ⇒ ('*a*, '*b*) *cblinfun-weak-star* ⇒ ('*a*, '*b*) *cblinfun-weak-star*› **is** *minus* .
lift-definition *plus-cblinfun-weak-star* :: ‹('*a*, '*b*) *cblinfun-weak-star* ⇒ ('*a*, '*b*) *cblinfun-weak-star* ⇒ ('*a*, '*b*) *cblinfun-weak-star*› **is** *plus* .
lift-definition *scaleR-cblinfun-weak-star* :: ‹*real* ⇒ ('*a*, '*b*) *cblinfun-weak-star* ⇒ ('*a*, '*b*) *cblinfun-weak-star*› **is** *scaleR* .
instance
by (*intro-classes*; *transfer*) (*auto simp add*: *scaleR-scaleC scaleC-add-right scaleC-add-left*)
end

instantiation *cblinfun-weak-star* :: (*chilbert-space*, *chilbert-space*) *topological-space* **begin**
lift-definition *open-cblinfun-weak-star* :: ‹('*a*, '*b*) *cblinfun-weak-star set* ⇒ *bool*› **is** ‹*openin weak-star-topology*› .
instance
proof *intro-classes*
show ‹*open* (*UNIV* :: ('*a*,'*b*) *cblinfun-weak-star set*)›

by *transfer* (*metis weak-star-topology-topspace openin-topspace*)
show ‹*open S* ⟹ *open T* ⟹ *open* (*S* ∩ *T*)› **for** *S T* :: ‹('*a*,'*b*) *cblinfun-weak-star set*›
by *transfer auto*
show ‹∀ *S*∈*K*. *open S* ⟹ *open* (⋃ *K*)› **for** *K* :: ‹('*a*,'*b*) *cblinfun-weak-star set set*›
by *transfer auto*
qed
end

lemma *transfer-nhds-weak-star-topology*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*cr-cblinfun-weak-star* ===> *rel-filter cr-cblinfun-weak-star*) (*nhdsin weak-star-topology*) *nhds*›
proof −
have (*cr-cblinfun-weak-star* ===> *rel-filter cr-cblinfun-weak-star*)
(λ*a*. ∏ (*principal* ‘ {*S*. *openin weak-star-topology S* ∧ *a* ∈ *S*}))
(λ*a*. ∏ (*principal* ‘ {*S*. *open S* ∧ *a* ∈ *S*}))
by *transfer-prover*
thus *?thesis*
unfolding *nhds-def nhdsin-def weak-star-topology-topspace* **by** *simp*
qed

lemma *limitin-weak-star-topology'*:
‹*limitin weak-star-topology f l F* ⟷ (∀ *t*. ((λ*j*. *trace* (*from-trace-class t* o_{CL} *f j*)) ⟶ *trace* (*from-trace-class t* o_{CL} *l*)) *F*)›
by (*simp add*: *weak-star-topology-def' limitin-pullback-topology tendsto-coordinatewise*)

lemma *limitin-weak-star-topology*:
‹*limitin weak-star-topology f l F* ⟷ (∀ *t*. *trace-class t* ⟶ ((λ*j*. *trace* (*t* o_{CL} *f j*)) ⟶ *trace* (*t* o_{CL} *l*)) *F*)›
by (*smt* (*z3*) *eventually-mono from-trace-class from-trace-class-cases limitin-weak-star-topology' mem-Collect-eq tendsto-def*)

lemma *filterlim-weak-star-topology*:
‹*filterlim f* (*nhdsin weak-star-topology l*) = *limitin weak-star-topology f l*›
by (*auto simp*: *weak-star-topology-topspace simp flip*: *filterlim-nhdsin-iff-limitin*)

lemma *openin-weak-star-topology'*: ‹*openin weak-star-topology U* ⟷ (∃ *V*. *open V* ∧ *U* = (λ*x t*. *trace* (*from-trace-class t* o_{CL} *x*)) −‘ *V*)›
by (*simp add*: *weak-star-topology-def' openin-pullback-topology*)

lemma *hausdorff-weak-star*[*simp*]: ‹*Hausdorff-space weak-star-topology*›
by (*metis cweak-operator-topology-topspace hausdorff-cweak-operator-topology Hausdorff-space-def weak-star-topology-topspace wot-weaker-than-weak-star'*)

lemma *Domainp-cr-cblinfun-weak-star*[*simp*]: ‹*Domainp cr-cblinfun-weak-star* = (λ-. *True*)›

by (*metis* (*no-types*, *opaque-lifting*) *DomainPI cblinfun-weak-star.left-total left-totalE*)

lemma *Rangep-cr-cblinfun-weak-star*[*simp*]: ‹*Rangep cr-cblinfun-weak-star* = (λ-. *True*)›
by (*meson RangePI cr-cblinfun-weak-star-def*)

lemma *transfer-euclidean-weak-star-topology*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹(*rel-topology cr-cblinfun-weak-star*) *weak-star-topology euclidean*›
proof (*unfold rel-topology-def*, *intro conjI allI impI*)
show ‹(*rel-set cr-cblinfun-weak-star* ===> (=)) (*openin weak-star-topology*) (*openin euclidean*)›
unfolding *rel-fun-def rel-set-def open-openin* [*symmetric*] *cr-cblinfun-weak-star-def*
by (*transfer*, *intro allI impI arg-cong*[*of* - - *openin x* **for** *x*]) *blast*
next
fix *U* :: ‹('*a* ⇒$_{CL}$ '*b*) *set*›
assume ‹*openin weak-star-topology U*›
show ‹*Domainp* (*rel-set cr-cblinfun-weak-star*) *U*›
by (*simp add*: *Domainp-set*)
next
fix *U* :: ‹('*a*, '*b*) *cblinfun-weak-star set*›
assume ‹*openin euclidean U*›
show ‹*Rangep* (*rel-set cr-cblinfun-weak-star*) *U*›
by (*simp add*: *Rangep-set*)
qed

instance *cblinfun-weak-star* :: (*chilbert-space*, *chilbert-space*) *t2-space*
apply (*rule hausdorff-OFCLASS-t2-space*)
apply *transfer*
by (*rule hausdorff-weak-star*)

lemma *weak-star-topology-plus-cont*: ‹*LIM* (*x*,*y*) *nhdsin weak-star-topology a* ×$_F$ *nhdsin weak-star-topology b*.
x + *y* :> *nhdsin weak-star-topology* (*a* + *b*)›
proof −
have *trace-plus*: ‹*trace* (*t* o$_{CL}$ (*a* + *b*)) = *trace* (*t* o$_{CL}$ *a*) + *trace* (*t* o$_{CL}$ *b*)› **if** ‹*trace-class t*›
for *t* :: ‹'*b* ⇒$_{CL}$ '*a*› **and** *a b*
by (*auto simp*: *cblinfun-compose-add-right trace-plus that trace-class-comp-left*)
show *?thesis*
unfolding *weak-star-topology-def'*
by (*rule pullback-topology-bi-cont*[**where** *f'*=*plus*])
(*auto simp*: *trace-plus case-prod-unfold tendsto-add-Pair*)
qed

instance *cblinfun-weak-star* :: (*chilbert-space*, *chilbert-space*) *topological-group-add*
proof *intro-classes*

show ‹((λx. *fst* x + *snd* x) ⟶ a + b) (*nhds* a ×$_F$ *nhds* b)› **for** a b :: ‹('a,'b) *cblinfun-weak-star*›
apply *transfer*
using *weak-star-topology-plus-cont*
by (*auto simp*: *case-prod-unfold*)

have ‹*continuous-map weak-star-topology euclidean* (λx. *trace* (t o$_{CL}$ − x))› **if** ‹*trace-class t*› **for** t :: ‹'b ⇒$_{CL}$ 'a›
using *weak-star-topology-continuous-duality*[*of* ‹−t›]
by (*auto simp*: *cblinfun-compose-uminus-left cblinfun-compose-uminus-right intro*!: *that trace-class-uminus*)
then have ∗: ‹*continuous-map weak-star-topology weak-star-topology* (*uminus* :: ('a ⇒$_{CL}$ 'b) ⇒ -)›
by (*auto simp*: *continuous-on-weak-star-topo-iff-coordinatewise*)
show ‹(*uminus* ⟶ − a) (*nhds* a)› **for** a :: ‹('a,'b) *cblinfun-weak-star*›
apply (*subst tendsto-at-iff-tendsto-nhds*[*symmetric*])
apply (*subst isCont-def*[*symmetric*])
apply (*rule continuous-on-interior*[**where** *S*=*UNIV*])
apply (*subst continuous-map-iff-continuous2*[*symmetric*])
apply *transfer*
using ∗ **by** *auto*
qed

lemma *continuous-map-left-comp-weak-star*:
‹*continuous-map weak-star-topology weak-star-topology* (λa::'a::*chilbert-space* ⇒$_{CL}$ -. b o$_{CL}$ a)›

for b :: ‹'b::*chilbert-space* ⇒$_{CL}$ 'c::*chilbert-space*›
proof (*unfold weak-star-topology-def*, *rule continuous-map-pullback-both*)
define g' :: ‹('b ⇒$_{CL}$ 'a ⇒ *complex*) ⇒ ('c ⇒$_{CL}$ 'a ⇒ *complex*)› **where**
‹g' f = (λt∈*Collect trace-class*. f (t o$_{CL}$ b))› **for** f
show ‹(λx. λt∈*Collect trace-class*. *trace* (t o$_{CL}$ x)) −‘ *topspace* (*product-topology* (λ-. *euclidean*) (*Collect trace-class*)) ∩ *UNIV*
⊆ (o$_{CL}$) b −‘ *UNIV*›
by *simp*
show ‹g' (λt∈*Collect trace-class*. *trace* (t o$_{CL}$ x)) = (λt∈*Collect trace-class*. *trace* (t o$_{CL}$ (b o$_{CL}$ x)))› **for** x
by (*auto intro*!: *ext simp*: *g'-def*[*abs-def*] *cblinfun-compose-assoc trace-class-comp-left*)
show ‹*continuous-map* (*product-topology* (λ-. *euclidean*) (*Collect trace-class*)) (*product-topology* (λ-. *euclidean*) (*Collect trace-class*)) g'›
apply (*rule continuous-map-coordinatewise-then-product*)
subgoal for *i*
unfolding *g'-def*
apply (*subst restrict-apply'*)
subgoal by *simp*
subgoal by (*metis continuous-map-product-projection mem-Collect-eq trace-class-comp-left*)
done
subgoal by (*auto simp*: *g'-def*[*abs-def*])
done
qed

lemma *continuous-map-right-comp-weak-star*:
‹*continuous-map weak-star-topology weak-star-topology* (λ*b*::'*b*::*chilbert-space* ⇒$_{CL}$ -. *b* o_{CL} *a*)›

for *a* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
proof (*subst weak-star-topology-def*, *subst weak-star-topology-def*, *rule continuous-map-pullback-both*)
define *g'* :: ‹('*c* ⇒$_{CL}$ '*b* ⇒ *complex*) ⇒ ('*c* ⇒$_{CL}$ '*a* ⇒ *complex*)› **where**
‹*g' f* = (λ*t*∈*Collect trace-class*. *f* (*a* o_{CL} *t*))› **for** *f*
show ‹(λ*x*. λ*t*∈*Collect trace-class*. *trace* (*t* o_{CL} *x*)) −‘ *topspace* (*product-topology* (λ-. *euclidean*) (*Collect trace-class*)) ∩ *UNIV*
⊆ (λ*b*. *b* o_{CL} *a*) −‘ *UNIV*›
by *simp*
have ∗: *trace* (*a* o_{CL} *y* o_{CL} *x*) = *trace* (*y* o_{CL} (*x* o_{CL} *a*)) **if** *trace-class y* **for** *x* :: '*b* ⇒$_{CL}$ '*c* **and** *y* :: '*c* ⇒$_{CL}$ '*a*
by (*simp add*: *circularity-of-trace simp-a-oCL-b that trace-class-comp-left*)
show ‹*g'* (λ*t*∈*Collect trace-class*. *trace* (*t* o_{CL} *x*)) = (λ*t*∈*Collect trace-class*. *trace* (*t* o_{CL} (*x* o_{CL} *a*)))› **for** *x*
by (*auto intro*!: *ext simp*: *g'-def*[*abs-def*] *trace-class-comp-right* ∗)
show ‹*continuous-map* (*product-topology* (λ-. *euclidean*) (*Collect trace-class*))
(*product-topology* (λ-. *euclidean*) (*Collect trace-class*)) *g'*›
apply (*rule continuous-map-coordinatewise-then-product*)
subgoal for *i*
unfolding *g'-def mem-Collect-eq*
apply (*subst restrict-apply'*)
subgoal by *simp*
subgoal
by (*metis continuous-map-product-projection mem-Collect-eq trace-class-comp-right*)
done
subgoal by (*auto simp*: *g'-def*[*abs-def*])
done
qed

lemma *continuous-map-scaleC-weak-star*: ‹*continuous-map weak-star-topology weak-star-topology* (*scaleC c*)›
apply (*subst asm-rl*[*of* ‹*scaleC c* = (*o*$_{CL}$) (*c* ∗$_C$ *id-cblinfun*)›])
subgoal by *auto*
subgoal by (*rule continuous-map-left-comp-weak-star*)
done

lemma *continuous-scaleC-weak-star*: ‹*continuous-on X* (*scaleC c* :: (-,-) *cblinfun-weak-star* ⇒ -)›
apply (*rule continuous-on-subset*[*rotated*, **where** *s*=*UNIV*])
subgoal by *simp*
subgoal
apply (*subst continuous-map-iff-continuous2*[*symmetric*])
apply *transfer*
by (*rule continuous-map-scaleC-weak-star*)
done

lemma *weak-star-closure-is-csubspace*[*simp*]:

fixes *A*::(*'a*::*chilbert-space*, *'b*::*chilbert-space*) *cblinfun-weak-star set*
assumes ‹*csubspace A*›
shows ‹*csubspace* (*closure A*)›
proof (*rule complex-vector.subspaceI*)
include *lattice-syntax*
show *0*: ‹*0* ∈ *closure A*›
by (*simp add*: *assms closure-def complex-vector.subspace-0*)
show ‹*x* + *y* ∈ *closure A*› **if** ‹*x* ∈ *closure A*› ‹*y* ∈ *closure A*› **for** *x y*
proof −
define *FF* **where** ‹*FF* = ((*nhds x* ⊓ *principal A*) ×$_F$ (*nhds y* ⊓ *principal A*))›
have *nt*: ‹*FF* ≠ *bot*›
by (*simp add*: *prod-filter-eq-bot that*(*1*) *that*(*2*) *FF-def flip*: *closure-nhds-principal*)
have ‹∀$_F$ *x in FF*. *fst x* ∈ *A*›
unfolding *FF-def*
by (*smt* (*verit*, *ccfv-SIG*) *eventually-prod-filter fst-conv inf-sup-ord*(*2*) *le-principal*)
moreover have ‹∀$_F$ *x in FF*. *snd x* ∈ *A*›
unfolding *FF-def*
by (*smt* (*verit*, *ccfv-SIG*) *eventually-prod-filter snd-conv inf-sup-ord*(*2*) *le-principal*)
ultimately have *FF-plus*: ‹∀$_F$ *x in FF*. *fst x* + *snd x* ∈ *A*›
by (*smt* (*verit*, *best*) *assms complex-vector.subspace-add eventually-elim2*)

have ‹(*fst* ⟶ *x*) ((*nhds x* ⊓ *principal A*) ×$_F$ (*nhds y* ⊓ *principal A*))›
apply (*simp add*: *filterlim-def*)
using *filtermap-fst-prod-filter*
using *le-inf-iff* **by** *blast*
moreover have ‹(*snd* ⟶ *y*) ((*nhds x* ⊓ *principal A*) ×$_F$ (*nhds y* ⊓ *principal A*))›
apply (*simp add*: *filterlim-def*)
using *filtermap-snd-prod-filter*
using *le-inf-iff* **by** *blast*
ultimately have ‹(*id* ⟶ (*x*,*y*)) *FF*›
by (*simp add*: *filterlim-def nhds-prod prod-filter-mono FF-def*)

moreover note *tendsto-add-Pair*[*of x y*]
ultimately have ‹(((λ*x*. *fst x* + *snd x*) *o id*) ⟶ (λ*x*. *fst x* + *snd x*) (*x*,*y*)) *FF*›
unfolding *filterlim-def nhds-prod*
by (*smt* (*verit*, *best*) *filterlim-compose filterlim-def filterlim-filtermap fst-conv snd-conv tendsto-compose-filtermap*)

then have ‹((λ*x*. *fst x* + *snd x*) ⟶ (*x*+*y*)) *FF*›
by *simp*
then show ‹*x* + *y* ∈ *closure A*›
using *nt FF-plus* **by** (*rule limit-in-closure*)
qed
show ‹*c* ∗$_C$ *x* ∈ *closure A*› **if** ‹*x* ∈ *closure A*› **for** *x c*
proof (*cases c* = *0*)
case *False*
have (∗$_C$) *c* ‘ *closure A* ⊆ *closure A*
using *csubspace-scaleC-invariant*[*of c A*] ‹*csubspace A*› *False closure-subset*[*of A*]
by (*intro image-closure-subset continuous-scaleC-weak-star closed-closure*) *auto*

```
    thus ?thesis
      using that by blast
  qed (use 0 in auto)
qed


lemma transfer-csubspace-cblinfun-weak-star[transfer-rule]:
  includes lifting-syntax
  shows ‹(rel-set cr-cblinfun-weak-star ===> (=)) csubspace csubspace›
  unfolding complex-vector.subspace-def
  by transfer-prover

lemma transfer-closed-cblinfun-weak-star[transfer-rule]:
  includes lifting-syntax
  shows ‹(rel-set cr-cblinfun-weak-star ===> (=)) (closedin weak-star-topology) closed›
proof −
  have (rel-set cr-cblinfun-weak-star ===> (=))
        (λS. openin weak-star-topology (UNIV − S))
        (λS. open (UNIV − S))
    by transfer-prover
  thus ?thesis
    by (simp add: closed-def[abs-def] closedin-def[abs-def] Compl-eq-Diff-UNIV)
qed

lemma transfer-closure-cblinfun-weak-star[transfer-rule]:
  includes lifting-syntax
 shows ‹(rel-set cr-cblinfun-weak-star ===> rel-set cr-cblinfun-weak-star) (Abstract-Topology.closure-of
weak-star-topology) closure›
 apply (subst closure-of-hull[where X=weak-star-topology, unfolded weak-star-topology-topspace,
simplified, abs-def])
  apply (subst closure-hull[abs-def])
  unfolding hull-def
  by transfer-prover

lemma weak-star-closure-is-csubspace'[simp]:
  fixes A::('a::chilbert-space ⇒CL 'b::chilbert-space) set
  assumes ‹csubspace A›
  shows ‹csubspace (weak-star-topology closure-of A)›
  using weak-star-closure-is-csubspace[of ‹to-weak-star ‘ A›] assms
  apply (transfer fixing: A)
  by simp

lemma has-sum-closed-weak-star-topology:
  assumes aA: ‹⋀i. a i ∈ A›
  assumes closed: ‹closedin weak-star-topology A›
  assumes subspace: ‹csubspace A›
  assumes has-sum: ‹⋀t. trace-class t ⟹ ((λi. trace (t oCL a i)) has-sum trace (t oCL b)) I›
  shows ‹b ∈ A›
proof −
```

```
  have 1: ‹range (sum a) ⊆ A›
  proof −
    have ‹sum a X ∈ A› for X
      apply (induction X rule:infinite-finite-induct)
      by (auto simp add: subspace complex-vector.subspace-0 aA complex-vector.subspace-add)
    then show ?thesis
      by auto
  qed

  from has-sum
   have ‹((λF. ∑ i∈F. trace (t o_CL a i)) ⟶ trace (t o_CL b)) (finite-subsets-at-top I)› if
‹trace-class t› for t
    by (auto intro: that simp: has-sum-def)
  then have ‹limitin weak-star-topology (λF. ∑ i∈F. a i) b (finite-subsets-at-top I)›
   by (auto simp add: limitin-weak-star-topology cblinfun-compose-sum-right trace-sum trace-class-comp-left)
  then show ‹b ∈ A›
    using 1 closed apply (rule limitin-closedin)
    by simp
qed

lemma has-sum-in-weak-star:
  ‹has-sum-in weak-star-topology f A l ⟷
     (∀ t. trace-class t ⟶ ((λi. trace (t o_CL f i)) has-sum trace (t o_CL l)) A)›
proof −
  have ∗: ‹trace (t o_CL sum f F) = sum (λi. trace (t o_CL f i)) F› if ‹trace-class t›
    for t F
    by (simp-all add: cblinfun-compose-sum-right that trace-class-comp-left trace-sum)
  show ?thesis
    by (simp add: ∗ has-sum-def has-sum-in-def limitin-weak-star-topology)
qed

lemma has-sum-butterfly-ket: ‹has-sum-in weak-star-topology (λi. butterfly (ket i) (ket i)) UNIV
id-cblinfun›
proof (rule has-sum-in-weak-star[THEN iffD2, rule-format])
  fix t :: ‹'a ell2 ⇒_CL 'a ell2›
  assume [simp]: ‹trace-class t›
  from trace-has-sum[OF is-onb-ket ‹trace-class t›]
  have ‹((λi. ket i •_C (t *_V ket i)) has-sum trace t) UNIV›
    apply (subst (asm) has-sum-reindex)
    by (auto simp: o-def)
  then show ‹((λi. trace (t o_CL butterfly (ket i) (ket i))) has-sum trace (t o_CL id-cblinfun))
UNIV›
    by (simp add: trace-butterfly-comp')
qed

lemma sandwich-weak-star-cont[simp]:
  ‹continuous-map weak-star-topology weak-star-topology (sandwich A)›
 using continuous-map-compose[OF continuous-map-left-comp-weak-star continuous-map-right-comp-weak-star]
  by (auto simp: o-def sandwich-apply[abs-def])
```

lemma *has-sum-butterfly-ket-a*: ‹*has-sum-in weak-star-topology* (λ*i*. *butterfly* (*a* $*_V$ *ket i*) (*ket i*)) *UNIV a*›
proof −
have ‹*has-sum-in weak-star-topology* ((λ*b*. *a* o_{CL} *b*) ∘ (λ*i*. *butterfly* (*ket i*) (*ket i*))) *UNIV* (*a* o_{CL} *id-cblinfun*)›
apply (*rule has-sum-in-comm-additive*)
by (*auto intro*!: *has-sum-butterfly-ket continuous-map-is-continuous-at-point limitin-continuous-map continuous-map-left-comp-weak-star cblinfun-compose-add-right simp*: *Modules.additive-def*)
then show *?thesis*
by (*auto simp*: *o-def cblinfun-comp-butterfly*)
qed

lemma *finite-rank-weak-star-dense*[*simp*]: ‹*weak-star-topology closure-of* (*Collect finite-rank*) = (*UNIV* :: (*'a ell2* ⇒$_{CL}$ *'b*::*chilbert-space*) *set*)›
proof −
have ‹*x* ∈ *weak-star-topology closure-of* (*Collect finite-rank*)› **for** *x* :: ‹*'a ell2* ⇒$_{CL}$ *'b*›
proof (*rule limitin-closure-of*)
define *f* :: ‹*'a* ⇒ *'a ell2* ⇒$_{CL}$ *'b*› **where** ‹*f* = (λ*i*. *butterfly* (*x* $*_V$ *ket i*) (*ket i*))›
have ‹*has-sum-in weak-star-topology f UNIV x*›
using *f-def has-sum-butterfly-ket-a* **by** *blast*
then show ‹*limitin weak-star-topology* (*sum f*) *x* (*finite-subsets-at-top UNIV*)›
using *has-sum-in-def* **by** *blast*
show ‹∀$_F$ *F in finite-subsets-at-top UNIV*. (∑ *i*∈*F*. *butterfly* (*x* $*_V$ *ket i*) (*ket i*)) ∈ *Collect finite-rank*›
by (*auto intro*!: *finite-rank-sum simp*: *f-def*)
show ‹*finite-subsets-at-top UNIV* ≠ ⊥›
by *simp*
qed
then show *?thesis*
by *auto*
qed

lemma *butterkets-weak-star-dense*[*simp*]:
‹*weak-star-topology closure-of cspan* ((λ(ξ,η). *butterfly* (*ket* ξ) (*ket* η)) ‘ *UNIV*) = *UNIV*›

proof −
from *continuous-map-image-closure-subset*[*OF weak-star-topology-weaker-than-euclidean*]
have ‹*weak-star-topology closure-of* (*cspan* ((λ(ξ,η). *butterfly* (*ket* ξ) (*ket* η)) ‘ *UNIV*)) ⊇ *closure* (*cspan* ((λ(ξ,η). *butterfly* (*ket* ξ) (*ket* η)) ‘ *UNIV*))› (**is** ‹- ⊇ ...›)
by *auto*
moreover
have ‹... = *Collect compact-op*›
unfolding *finite-rank-dense-compact*[*OF is-onb-ket is-onb-ket*, *symmetric*]
by (*simp add*: *image-image case-prod-beta flip*: *map-prod-image*)
moreover have ‹... ⊇ *Collect finite-rank*›

by (*metis closure-subset compact-op-finite-rank mem-Collect-eq subsetI subset-antisym*)
ultimately have ∗: ‹*weak-star-topology closure-of* (*cspan* ((λ(ξ,η). *butterfly* (*ket* ξ) (*ket* η)) ‘ *UNIV*)) ⊇ *Collect finite-rank*›
by *blast*
have ‹*weak-star-topology closure-of cspan* ((λ(ξ,η). *butterfly* (*ket* ξ) (*ket* η)) ‘ *UNIV*)
= *weak-star-topology closure-of* (*weak-star-topology closure-of cspan* ((λ(ξ,η). *butterfly* (*ket* ξ) (*ket* η)) ‘ *UNIV*))›
by *simp*
also have ‹... ⊇ *weak-star-topology closure-of Collect finite-rank*› (**is** ‹- ⊇ ...›)
using ∗ *closure-of-mono* **by** *blast*
also have ‹... = *UNIV*›
by *simp*
finally show *?thesis*
by *auto*
qed

lemma *weak-star-clinear-eq-butterfly-ketI*:
fixes *F G* :: ‹('*a ell2* ⇒$_{CL}$ '*b ell2*) ⇒ '*c*::*complex-vector*›
assumes *clinear F* **and** *clinear G*
and ‹*continuous-map weak-star-topology T F*› **and** ‹*continuous-map weak-star-topology T G*›
and ‹*Hausdorff-space T*›
assumes ⋀*i j*. *F* (*butterfly* (*ket i*) (*ket j*)) = *G* (*butterfly* (*ket i*) (*ket j*))
shows *F* = *G*
proof −
have *FG*: ‹*F x* = *G x*› **if** ‹*x* ∈ *cspan* ((λ(ξ,η). *butterfly* (*ket* ξ) (*ket* η)) ‘ *UNIV*)› **for** *x*
by (*smt* (*verit*) *assms*(*1*) *assms*(*2*) *assms*(*6*) *complex-vector.linear-eq-on imageE split-def that*)
show *?thesis*
apply (*rule ext*)
using ‹*Hausdorff-space T*› *FG*
apply (*rule closure-of-eqI*[**where** *f*=*F* **and** *g*=*G* **and** *S*=‹*cspan* ((λ(ξ,η). *butterfly* (*ket* ξ) (*ket* η)) ‘ *UNIV*)›])
using *assms butterkets-weak-star-dense* **by** *auto*
qed

lemma *continuous-map-scaleC-weak-star'*[*continuous-intros*]:
assumes ‹*continuous-map T weak-star-topology f*›
shows ‹*continuous-map T weak-star-topology* (λ*x*. *scaleC c* (*f x*))›
using *continuous-map-compose*[*OF assms continuous-map-scaleC-weak-star*]
by (*simp add*: *o-def*)

lemma *continuous-map-uminus-weak-star*[*continuous-intros*]:
assumes ‹*continuous-map T weak-star-topology f*›
shows ‹*continuous-map T weak-star-topology* (λ*x*. − *f x*)›
apply (*subst scaleC-minus1-left*[*abs-def*,*symmetric*])
by (*intro continuous-map-scaleC-weak-star' assms*)

lemma *continuous-map-add-weak-star*[*continuous-intros*]:
assumes ‹*continuous-map T weak-star-topology f*›
assumes ‹*continuous-map T weak-star-topology g*›
shows ‹*continuous-map T weak-star-topology* (λ*x*. *f x* + *g x*)›
proof −
have ‹*continuous-map T euclidean* (λ*x*. *trace* (*t* o_{CL} *f x*))› **if** ‹*trace-class t*› **for** *t*
using *assms*(*1*) *continuous-on-weak-star-topo-iff-coordinatewise that* **by** *auto*
moreover have ‹*continuous-map T euclidean* (λ*x*. *trace* (*t* o_{CL} *g x*))› **if** ‹*trace-class t*› **for** *t*
using *assms*(*2*) *continuous-on-weak-star-topo-iff-coordinatewise that* **by** *auto*
ultimately show *?thesis*
by (*auto intro*!: *continuous-map-add simp add*: *continuous-on-weak-star-topo-iff-coordinatewise cblinfun-compose-add-right trace-class-comp-left trace-plus*)
qed

lemma *continuous-map-minus-weak-star*[*continuous-intros*]:
assumes ‹*continuous-map T weak-star-topology f*›
assumes ‹*continuous-map T weak-star-topology g*›
shows ‹*continuous-map T weak-star-topology* (λ*x*. *f x* − *g x*)›
by (*subst diff-conv-add-uminus*) (*intro assms continuous-intros*)

lemma *weak-star-topology-is-norm-topology-fin-dim*[*simp*]:
‹(*weak-star-topology* :: ('*a*::{*cfinite-dim*,*chilbert-space*} ⇒$_{CL}$ '*b*::{*cfinite-dim*,*chilbert-space*}) *topology*) = *euclidean*›
proof −
have *1*: ‹*continuous-map euclidean weak-star-topology* (*id* :: '*a*⇒$_{CL}$'*b* ⇒ -)›
by (*simp add*: *id-def weak-star-topology-weaker-than-euclidean*)
have ‹*continuous-map weak-star-topology cweak-operator-topology* (*id* :: '*a*⇒$_{CL}$'*b* ⇒ -)›
by (*simp only*: *id-def wot-weaker-than-weak-star*)
then have *2*: ‹*continuous-map weak-star-topology euclidean* (*id* :: '*a*⇒$_{CL}$'*b* ⇒ -)›
by (*simp only*: *wot-is-norm-topology-findim*)
from *1 2*
show *?thesis*
by (*auto simp*: *topology-finer-continuous-id*[*symmetric*] *simp flip*: *openin-inject*)
qed

lemma *infsum-mono-wot*:
fixes *f* :: '*a* ⇒ ('*b*::*chilbert-space* ⇒$_{CL}$ '*b*)
assumes *summable-on-in cweak-operator-topology f A* **and** *summable-on-in cweak-operator-topology g A*
assumes ‹⋀*x*. *x* ∈ *A* ⟹ *f x* ≤ *g x*›
shows *infsum-in cweak-operator-topology f A* ≤ *infsum-in cweak-operator-topology g A*
by (*meson assms has-sum-in-infsum-in has-sum-mono-wot hausdorff-cweak-operator-topology*)

unbundle *no-cblinfun-notation*

end

14 *Hilbert-Space-Tensor-Product* − Tensor product of Hilbert Spaces

theory *Hilbert-Space-Tensor-Product*
imports *Complex-Bounded-Operators.Complex-L2 Misc-Tensor-Product*
Strong-Operator-Topology Polynomial-Interpolation.Ring-Hom
Positive-Operators Weak-Star-Topology Spectral-Theorem Trace-Class
begin

unbundle *cblinfun-notation*
hide-const (**open**) *Determinants.trace*
hide-fact (**open**) *Determinants.trace-def*

14.1 Tensor product on - *ell2*

lift-definition *tensor-ell2* :: ‹'*a ell2* ⇒ '*b ell2* ⇒ ('*a*×'*b*) *ell2*› (**infixr** ⊗$_s$ *70*) **is**
‹λψ φ (*i*,*j*). ψ *i* * φ *j*›
proof −
fix ψ :: ‹'*a* ⇒ *complex*› **and** φ :: ‹'*b* ⇒ *complex*›
assume ‹*has-ell2-norm* ψ› ‹*has-ell2-norm* φ›
from ‹*has-ell2-norm* φ› **have** φ-*sum*: ‹(λ*j*. (ψ *i* * φ *j*)^2) *abs-summable-on UNIV*› **for** *i*
by (*metis ell2-norm-smult*(*1*) *has-ell2-norm-def*)
have *double-sum*: ‹(λ*i*. ∑$_\infty$*j*. *cmod* ((ψ *i* * φ *j*)^2)) *abs-summable-on UNIV*›
unfolding *norm-mult power-mult-distrib infsum-cmult-right'*
by (*rule summable-on-cmult-left*) (*use* ‹*has-ell2-norm* ψ› **in** ‹*auto simp*: *has-ell2-norm-def*›)
have ‹(λ(*i*,*j*). (ψ *i* * φ *j*)^2) *abs-summable-on UNIV* × *UNIV*›
by (*rule abs-summable-on-Sigma-iff*[*THEN iffD2*]) (*use* φ-*sum double-sum* **in** *auto*)
then show ‹*has-ell2-norm* (λ(*i*, *j*). ψ *i* * φ *j*)›
by (*auto simp add*: *has-ell2-norm-def case-prod-beta*)
qed

lemma *tensor-ell2-add1*: ‹*tensor-ell2* (*a* + *b*) *c* = *tensor-ell2 a c* + *tensor-ell2 b c*›
by *transfer* (*auto simp*: *case-prod-beta vector-space-over-itself.scale-left-distrib*)

lemma *tensor-ell2-add2*: ‹*tensor-ell2 a* (*b* + *c*) = *tensor-ell2 a b* + *tensor-ell2 a c*›
by *transfer* (*auto simp*: *case-prod-beta algebra-simps*)

lemma *tensor-ell2-scaleC1*: ‹*tensor-ell2* (*c* *$_C$ *a*) *b* = *c* *$_C$ *tensor-ell2 a b*›
by *transfer* (*auto simp*: *case-prod-beta*)

lemma *tensor-ell2-scaleC2*: ‹*tensor-ell2 a* (*c* *$_C$ *b*) = *c* *$_C$ *tensor-ell2 a b*›
by *transfer* (*auto simp*: *case-prod-beta*)

lemma *tensor-ell2-diff1*: ‹*tensor-ell2* (*a* − *b*) *c* = *tensor-ell2 a c* − *tensor-ell2 b c*›
by *transfer* (*auto simp*: *case-prod-beta ordered-field-class.sign-simps*)

lemma *tensor-ell2-diff2*: ‹*tensor-ell2 a* (*b* − *c*) = *tensor-ell2 a b* − *tensor-ell2 a c*›
by *transfer* (*auto simp*: *case-prod-beta ordered-field-class.sign-simps*)

lemma *tensor-ell2-inner-prod*[*simp*]: ‹*tensor-ell2 a b* ∙$_C$ *tensor-ell2 c d* = (*a* ∙$_C$ *c*) ∗ (*b* ∙$_C$ *d*)›
apply (*rule local-defE*[**where** *y*=‹*tensor-ell2 a b*›], *rename-tac ab*)
apply (*rule local-defE*[**where** *y*=‹*tensor-ell2 c d*›], *rename-tac cd*)
proof (*transfer*, *hypsubst-thin*)
fix *a c* :: ‹'*a* ⇒ *complex*› **and** *b d* :: ‹'*b* ⇒ *complex*›

assume *assms*: ‹*has-ell2-norm* (λ(*i*, *j*). *a i* ∗ *b j*)› ‹*has-ell2-norm* (λ(*i*, *j*). *c i* ∗ *d j*)›

have ∗: ‹(λ*xy*. *cnj* (*a* (*fst xy*) ∗ *b* (*snd xy*)) ∗ (*c* (*fst xy*) ∗ *d* (*snd xy*))) *abs-summable-on UNIV*›
apply (*rule abs-summable-product*)
subgoal
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *complex-mod-cnj has-ell2-norm-def norm-power split-def summable-on-cong*)
subgoal
by (*metis* (*mono-tags*, *lifting*) *assms*(*2*) *case-prod-unfold has-ell2-norm-def summable-on-cong*)
done

then have ∗: ‹(λ(*x*, *y*). *cnj* (*a x* ∗ *b y*) ∗ (*c x* ∗ *d y*)) *summable-on UNIV* × *UNIV*›
using *abs-summable-summable* **by** (*auto simp*: *case-prod-unfold*)

have ‹(∑$_\infty$*i*. *cnj* (*case i of* (*i*, *j*) ⇒ *a i* ∗ *b j*) ∗ (*case i of* (*i*, *j*) ⇒ *c i* ∗ *d j*))
= (∑$_\infty$(*i*,*j*)∈*UNIV*×*UNIV*. *cnj* (*a i* ∗ *b j*) ∗ (*c i* ∗ *d j*))› (**is** ‹*?lhs* = -›)
by (*simp add*: *case-prod-unfold*)
also have ‹... = (∑$_\infty$*i*. ∑$_\infty$*j*. *cnj* (*a i* ∗ *b j*) ∗ (*c i* ∗ *d j*))›
by (*subst infsum-Sigma'-banach*[*symmetric*]) (*use* ∗ **in** *auto*)
also have ‹... = (∑$_\infty$*i*. *cnj* (*a i*) ∗ *c i*) ∗ (∑$_\infty$*j*. *cnj* (*b j*) ∗ (*d j*))› (**is** ‹- = *?rhs*›)
by (*subst infsum-cmult-left'*[*symmetric*])
(*auto intro*!: *infsum-cong simp flip*: *infsum-cmult-right'*)
finally show ‹*?lhs* = *?rhs*› **.**
qed

lemma *norm-tensor-ell2*: ‹*norm* (*a* ⊗$_s$ *b*) = *norm a* ∗ *norm b*›
by (*simp add*: *norm-eq-sqrt-cinner*[**where** 'a=‹(-::*type*) *ell2*›] *norm-mult real-sqrt-mult*)

lemma *clinear-tensor-ell21*: *clinear* (λ*b*. *a* ⊗$_s$ *b*)
by (*rule clinearI*; *transfer*)
(*auto simp add*: *case-prod-beta cond-case-prod-eta algebra-simps fun-eq-iff*)

lemma *bounded-clinear-tensor-ell21*: *bounded-clinear* (λ*b*. *a* ⊗$_s$ *b*)
by (*auto intro*!: *bounded-clinear.intro clinear-tensor-ell21*
simp: *bounded-clinear-axioms-def norm-tensor-ell2 mult.commute*[*of norm a*])

lemma *clinear-tensor-ell22*: *clinear* (λ*a*. *a* ⊗$_s$ *b*)
by (*rule clinearI*; *transfer*) (*auto simp*: *case-prod-beta algebra-simps*)

lemma *bounded-clinear-tensor-ell22*: *bounded-clinear* (λ*a*. *tensor-ell2 a b*)
by (*auto intro*!: *bounded-clinear.intro clinear-tensor-ell22*
simp: *bounded-clinear-axioms-def norm-tensor-ell2*)

lemma *tensor-ell2-ket*: *tensor-ell2* (*ket i*) (*ket j*) = *ket* (*i*,*j*)
by *transfer auto*

lemma *tensor-ell2-0-left*[*simp*]: ‹*0* ⊗$_s$ *x* = *0*›
by *transfer auto*

lemma *tensor-ell2-0-right*[*simp*]: ‹*x* ⊗$_s$ *0* = *0*›
by *transfer auto*

lemma *tensor-ell2-sum-left*: ‹(∑ *x*∈*X*. *a x*) ⊗$_s$ *b* = (∑ *x*∈*X*. *a x* ⊗$_s$ *b*)›
by (*induction X rule*:*infinite-finite-induct*) (*auto simp*: *tensor-ell2-add1*)

lemma *tensor-ell2-sum-right*: ‹*a* ⊗$_s$ (∑ *x*∈*X*. *b x*) = (∑ *x*∈*X*. *a* ⊗$_s$ *b x*)›
by (*induction X rule*:*infinite-finite-induct*) (*auto simp*: *tensor-ell2-add2*)

lemma *tensor-ell2-dense*:
fixes *S* :: ‹'*a ell2 set*› **and** *T* :: ‹'*b ell2 set*›
assumes ‹*closure* (*cspan S*) = *UNIV*› **and** ‹*closure* (*cspan T*) = *UNIV*›
shows ‹*closure* (*cspan* {*a*⊗$_s$*b* | *a b*. *a*∈*S* ∧ *b*∈*T*}) = *UNIV*›
proof −
define *ST* **where** ‹*ST* = {*a*⊗$_s$*b* | *a b*. *a*∈*S* ∧ *b*∈*T*}›
from *assms* **have** *1*: ‹*bounded-clinear F* ⟹ *bounded-clinear G* ⟹ (∀ *x*∈*S*. *F x* = *G x*) ⟹ *F* = *G*› **for** *F G* :: ‹'*a ell2* ⇒ *complex*›
using *bounded-clinear-eq-on-closure*[*of F G S*] **by** *auto*
from *assms* **have** *2*: ‹*bounded-clinear F* ⟹ *bounded-clinear G* ⟹ (∀ *x*∈*T*. *F x* = *G x*) ⟹ *F* = *G*› **for** *F G* :: ‹'*b ell2* ⇒ *complex*›
using *bounded-clinear-eq-on-closure*[*of F G T*] **by** *auto*
have ‹*F* = *G*›
if [*simp*]: ‹*bounded-clinear F*› ‹*bounded-clinear G*› **and** *eq*: ‹∀ *x*∈*ST*. *F x* = *G x*›
for *F G* :: ‹('*a*×'*b*) *ell2* ⇒ *complex*›
proof −
from *eq* **have** *eq'*: ‹*F* (*s* ⊗$_s$ *t*) = *G* (*s* ⊗$_s$ *t*)› **if** ‹*s* ∈ *S*› **and** ‹*t* ∈ *T*› **for** *s t*
using *ST-def that* **by** *blast*
have *eq''*: ‹*F* (*s* ⊗$_s$ *ket t*) = *G* (*s* ⊗$_s$ *ket t*)› **if** ‹*s* ∈ *S*› **for** *s t*
by (*rule fun-cong*[**where** *x*=‹*ket t*›], *rule 2*)
(*use eq' that* **in** ‹*auto simp*: *bounded-clinear-compose bounded-clinear-tensor-ell21*›)
have *eq'''*: ‹*F* (*ket s* ⊗$_s$ *ket t*) = *G* (*ket s* ⊗$_s$ *ket t*)› **for** *s t*
by (*rule fun-cong*[**where** *x*=‹*ket s*›], *rule 1*)
(*use eq''* **in** ‹*auto simp*: *bounded-clinear-compose bounded-clinear-tensor-ell21*
intro: *bounded-clinear-compose*[*OF* - *bounded-clinear-tensor-ell22*]›)
show *F* = *G*
by (*rule bounded-clinear-equal-ket*) (*use eq'''* **in** ‹*auto simp*: *tensor-ell2-ket*›)
qed
then show ‹*closure* (*cspan ST*) = *UNIV*›
using *separating-dense-span* **by** *blast*

qed

definition *assoc-ell2* :: ‹(('a×'b)×'c) *ell2* ⇒$_{CL}$ ('a×('b×'c)) *ell2*› **where**
‹*assoc-ell2* = *classical-operator* (*Some o* (λ((a,b),c). (a,(b,c))))›

lemma *unitary-assoc-ell2*[*simp*]: ‹*unitary assoc-ell2*›
unfolding *assoc-ell2-def*
by (*rule unitary-classical-operator*, *rule o-bij*[*of* ‹(λ(a,(b,c)). ((a,b),c))›]) *auto*

lemma *assoc-ell2-tensor*: ‹*assoc-ell2* *$_V$ ((a ⊗$_s$ b) ⊗$_s$ c) = (a ⊗$_s$ (b ⊗$_s$ c))›
proof −
note [*simp*] = *bounded-clinear-compose*[*OF bounded-clinear-tensor-ell21*]
bounded-clinear-compose[*OF bounded-clinear-tensor-ell22*]
bounded-clinear-cblinfun-apply
have ‹*assoc-ell2* *$_V$ ((*ket a* ⊗$_s$ *ket b*) ⊗$_s$ *ket c*) = (*ket a* ⊗$_s$ (*ket b* ⊗$_s$ *ket c*))› **for** *a* :: 'a **and** *b* :: 'b **and** *c* :: 'c
by (*simp add*: *inj-def assoc-ell2-def classical-operator-ket classical-operator-exists-inj tensor-ell2-ket*)
then have ‹*assoc-ell2* *$_V$ ((*ket a* ⊗$_s$ *ket b*) ⊗$_s$ *c*) = (*ket a* ⊗$_s$ (*ket b* ⊗$_s$ *c*))› **for** *a* :: 'a **and** *b* :: 'b
apply −
apply (*rule fun-cong*[**where** *x*=*c*])
apply (*rule bounded-clinear-equal-ket*)
by *auto*
then have ‹*assoc-ell2* *$_V$ ((*ket a* ⊗$_s$ *b*) ⊗$_s$ *c*) = (*ket a* ⊗$_s$ (*b* ⊗$_s$ *c*))› **for** *a* :: 'a
apply −
apply (*rule fun-cong*[**where** *x*=*b*])
apply (*rule bounded-clinear-equal-ket*)
by *auto*
then show ‹*assoc-ell2* *$_V$ ((*a* ⊗$_s$ *b*) ⊗$_s$ *c*) = (*a* ⊗$_s$ (*b* ⊗$_s$ *c*))›
apply −
apply (*rule fun-cong*[**where** *x*=*a*])
apply (*rule bounded-clinear-equal-ket*)
by *auto*
qed

lemma *assoc-ell2'-tensor*: ‹*assoc-ell2** *$_V$ *tensor-ell2 a* (*tensor-ell2 b c*) = *tensor-ell2* (*tensor-ell2 a b*) *c*›
by (*metis* (*no-types*, *opaque-lifting*) *assoc-ell2-tensor cblinfun-apply-cblinfun-compose id-cblinfun.rep-eq unitaryD1 unitary-assoc-ell2*)

lemma *assoc-ell2'-inv*: *assoc-ell2* *o$_{CL}$* *assoc-ell2** = *id-cblinfun*
by (*auto intro*: *equal-ket*)

lemma *assoc-ell2-inv*: *assoc-ell2** *o$_{CL}$* *assoc-ell2* = *id-cblinfun*
by (*auto intro*: *equal-ket*)

definition *swap-ell2* :: ‹('a×'b) *ell2* ⇒$_{CL}$ ('b×'a) *ell2*› **where**

‹*swap-ell2* = *classical-operator* (*Some* o *prod.swap*)›

lemma *unitary-swap-ell2*[*simp*]: ‹*unitary swap-ell2*›
unfolding *swap-ell2-def* **by** (*rule unitary-classical-operator*) *auto*

lemma *swap-ell2-tensor*[*simp*]: ‹*swap-ell2* $*_V$ (*a* $\otimes_s$ *b*) = *b* $\otimes_s$ *a*› **for** *a* :: ‹'*a ell2*› **and** *b* :: ‹'*b ell2*›
proof −
note [*simp*] = *bounded-clinear-compose*[*OF bounded-clinear-tensor-ell21*]
bounded-clinear-compose[*OF bounded-clinear-tensor-ell22*]
bounded-clinear-cblinfun-apply
have ‹*swap-ell2* $*_V$ (*ket a* $\otimes_s$ *ket b*) = (*ket b* $\otimes_s$ *ket a*)› **for** *a* :: '*a* **and** *b* :: '*b*
by (*simp add*: *inj-def swap-ell2-def classical-operator-ket classical-operator-exists-inj tensor-ell2-ket*)
then have ‹*swap-ell2* $*_V$ (*ket a* $\otimes_s$ *b*) = (*b* $\otimes_s$ *ket a*)› **for** *a* :: '*a*
apply −
apply (*rule fun-cong*[**where** *x*=*b*])
apply (*rule bounded-clinear-equal-ket*)
by *auto*
then show ‹*swap-ell2* $*_V$ (*a* $\otimes_s$ *b*) = (*b* $\otimes_s$ *a*)›
apply −
apply (*rule fun-cong*[**where** *x*=*a*])
apply (*rule bounded-clinear-equal-ket*)
by *auto*
qed

lemma *swap-ell2-ket*[*simp*]: ‹(*swap-ell2* :: ('*a*×'*b*) *ell2* $\Rightarrow_{CL}$ -)$*_V$ *ket* (*x*,*y*) = *ket* (*y*,*x*)›
by (*metis swap-ell2-tensor tensor-ell2-ket*)

lemma *adjoint-swap-ell2*[*simp*]: ‹*swap-ell2** = *swap-ell2*›
by (*simp add*: *swap-ell2-def inv-map-total*)

lemma *tensor-ell2-extensionality*:
assumes ($\bigwedge$*s t*. *a* $*_V$ (*s* $\otimes_s$ *t*) = *b* $*_V$ (*s* $\otimes_s$ *t*))
shows *a* = *b*
using *assms* **by** (*auto intro*: *equal-ket simp flip*: *tensor-ell2-ket*)

lemma *tensor-ell2-nonzero*: ‹*a* $\otimes_s$ *b* ≠ *0*› **if** ‹*a* ≠ *0*› **and** ‹*b* ≠ *0*›
by (*use that* **in** *transfer*) (*auto simp*: *fun-eq-iff*)

lemma *swap-ell2-selfinv*[*simp*]: ‹*swap-ell2* o_{CL} *swap-ell2* = *id-cblinfun*›
by (*metis adjoint-swap-ell2 unitary-def unitary-swap-ell2*)

lemma *bounded-cbilinear-tensor-ell2*[*bounded-cbilinear*]: ‹*bounded-cbilinear* ($\otimes_s$)›
proof *standard*
fix *a a'* :: '*a ell2* **and** *b b'* :: '*b ell2* **and** *r* :: *complex*
show ‹*tensor-ell2* (*a* + *a'*) *b* = *tensor-ell2 a b* + *tensor-ell2 a' b*›
by (*meson tensor-ell2-add1*)
show ‹*tensor-ell2 a* (*b* + *b'*) = *tensor-ell2 a b* + *tensor-ell2 a b'*›

by (*simp add*: *tensor-ell2-add2*)
show ‹*tensor-ell2* (*r* $*_C$ *a*) *b* = *r* $*_C$ *tensor-ell2 a b*›
by (*simp add*: *tensor-ell2-scaleC1*)
show ‹*tensor-ell2 a* (*r* $*_C$ *b*) = *r* $*_C$ *tensor-ell2 a b*›
by (*simp add*: *tensor-ell2-scaleC2*)
show ‹∃ *K*. ∀ *a b*. *norm* (*tensor-ell2 a b*) ≤ *norm a* ∗ *norm b* ∗ *K* ›
by (*rule exI*[*of* - *1*]) (*simp add*: *norm-tensor-ell2*)
qed

lemma *ket-pair-split*: ‹*ket x* = *tensor-ell2* (*ket* (*fst x*)) (*ket* (*snd x*))›
by (*simp add*: *tensor-ell2-ket*)

lemma *tensor-ell2-is-ortho-set*:
assumes ‹*is-ortho-set A*› ‹*is-ortho-set B*›
shows ‹*is-ortho-set* {*a* ⊗$_s$ *b* |*a b*. *a* ∈ *A* ∧ *b* ∈ *B*}›
unfolding *is-ortho-set-def*
proof *safe*
fix *a a′ b b′*
assume *ab*: *a* ∈ *A a′* ∈ *A b* ∈ *B b′* ∈ *B a* ⊗$_s$ *b* ≠ *a′* ⊗$_s$ *b′*
hence *a* ≠ *a′* ∨ *b* ≠ *b′*
by *auto*
hence *is-orthogonal a a′* ∨ *is-orthogonal b b′*
using *assms is-ortho-setD ab* **by** *metis*
thus *is-orthogonal* (*a* ⊗$_s$ *b*) (*a′* ⊗$_s$ *b′*)
by *auto*
next
fix *a b*
assume *ab*: *a* ∈ *A b* ∈ *B* 0 = *a* ⊗$_s$ *b*
hence *a* ≠ 0 *b* ≠ 0
using *assms* **unfolding** *is-ortho-set-def* **by** *blast*+
thus *False* **using** *ab*
using *tensor-ell2-nonzero*[*of a b*] **by** *simp*
qed

lemma *tensor-ell2-dense′*: ‹*ccspan* {*a* ⊗$_s$ *b* |*a b*. *a* ∈ *A* ∧ *b* ∈ *B*} = ⊤› **if** ‹*ccspan A* = ⊤› **and** ‹*ccspan B* = ⊤›
proof −
from *that* **have** *Adense*: ‹*closure* (*cspan A*) = *UNIV*›
by (*transfer′ fixing*: *A*) *simp*
from *that* **have** *Bdense*: ‹*closure* (*cspan B*) = *UNIV*›
by (*transfer′ fixing*: *B*) *simp*
show ‹*ccspan* {*a* ⊗$_s$ *b* |*a b*. *a* ∈ *A* ∧ *b* ∈ *B*} = ⊤›
by (*transfer fixing*: *A B*) (*use Adense Bdense* **in** ‹*rule tensor-ell2-dense*›)
qed

lemma *tensor-ell2-is-onb*:
assumes ‹*is-onb A*› ‹*is-onb B*›

shows ‹*is-onb* $\{a \otimes_s b \mid a\ b.\ a \in A \wedge b \in B\}$›
proof (*subst is-onb-def*, *intro conjI ballI*)
show ‹*is-ortho-set* $\{a \otimes_s b \mid a\ b.\ a \in A \wedge b \in B\}$›
by (*rule tensor-ell2-is-ortho-set*) (*use assms* **in** ‹*auto simp*: *is-onb-def*›)
show ‹*ccspan* $\{a \otimes_s b \mid a\ b.\ a \in A \wedge b \in B\} = \top$›
by (*rule tensor-ell2-dense'*) (*use* ‹*is-onb A*› ‹*is-onb B*› **in** ‹*simp-all add*: *is-onb-def*›)
show ‹$ab \in \{a \otimes_s b \mid a\ b.\ a \in A \wedge b \in B\} \Longrightarrow$ *norm ab* = *1*› **for** *ab*
using ‹*is-onb A*› ‹*is-onb B*› **by** (*auto simp*: *is-onb-def norm-tensor-ell2*)
qed

lemma *continuous-tensor-ell2*: ‹*continuous-on UNIV* $(\lambda(x{::}'a\ ell2,\ y{::}'b\ ell2).\ x \otimes_s y)$›
proof −
have *cont*: ‹*continuous-on UNIV* $(\lambda t.\ t \otimes_s x)$› **for** $x :: ‹'b\ ell2›$
by (*intro linear-continuous-on bounded-clinear.bounded-linear bounded-clinear-tensor-ell22*)
have *lip*: ‹*local-lipschitz* (*UNIV* :: *'a ell2 set*) (*UNIV* :: *'b ell2 set*) $(\otimes_s)$›
proof (*rule local-lipschitzI*)
fix $t :: ‹'a\ ell2›$ **and** $x :: ‹'b\ ell2›$
define *u L* :: *real* **where** ‹*u* = *1*› **and** ‹*L* = *norm t* + *u*›
have ‹*u* > *0*›
by (*simp add*: *u-def*)
have [*simp*]: ‹*L* ≥ *0*›
by (*simp add*: *L-def u-def*)
have ∗: ‹*norm s* ≤ *L*› **if** ‹*s*∈*cball t u*› **for** $s :: ‹'a\ ell2›$
using *that* **unfolding** *L-def mem-cball* **by** *norm*
have ‹*L*−*lipschitz-on* (*cball x u*) $((\otimes_s)\ s)$› **if** ‹*s*∈*cball t u*› **for** $s :: ‹'a\ ell2›$
by (*rule lipschitz-onI*)
(*auto intro*!: *mult-right-mono* ∗[*OF that*]
simp add: *dist-norm norm-tensor-ell2 simp flip*: *tensor-ell2-diff2*)
with ‹*u* > *0*› **show** ‹∃ *u*>*0*. ∃ *L*. ∀ *s*∈*cball t u* ∩ *UNIV*. *L*−*lipschitz-on* (*cball x u* ∩ *UNIV*) $((\otimes_s)\ s)$›
by *force*
qed
show *?thesis*
by (*subst UNIV-Times-UNIV*[*symmetric*]) (*use lip cont* **in** ‹*rule Lipschitz.continuous-on-TimesI*›)
qed

lemma *summable-on-tensor-ell2-right*: ‹φ *summable-on A* $\Longrightarrow$ $(\lambda x.\ \psi \otimes_s \varphi\ x)$ *summable-on A*›
by (*rule summable-on-bounded-linear*[**where** *h*=‹$\lambda x.\ \psi \otimes_s x$›]) (*intro bounded-linear-intros*)

lemma *summable-on-tensor-ell2-left*: ‹φ *summable-on A* $\Longrightarrow$ $(\lambda x.\ \varphi\ x \otimes_s \psi)$ *summable-on A*›
by (*rule summable-on-bounded-linear*[**where** *h*=‹$\lambda x.\ x \otimes_s \psi$›]) (*intro bounded-linear-intros*)

lift-definition *tensor-ell2-left* :: $‹'a\ ell2 \Rightarrow ('b\ ell2 \Rightarrow_{CL} ('a \times 'b)\ ell2)›$ **is**
‹$\lambda\psi\ \varphi.\ \psi \otimes_s \varphi$›
by (*simp add*: *bounded-cbilinear.bounded-clinear-right bounded-cbilinear-tensor-ell2*)

lemma *tensor-ell2-left-apply*[*simp*]: ‹*tensor-ell2-left* $\psi *_V \varphi = \psi \otimes_s \varphi$›
by (*transfer fixing*: $\psi\ \varphi$) *simp*

lift-definition *tensor-ell2-right* :: ‹'a *ell2* ⇒ ('b *ell2* ⇒$_{CL}$ ('b×'a) *ell2*)› **is**
‹λψ φ. φ ⊗$_s$ ψ›
by (*simp add*: *bounded-clinear-tensor-ell22*)

lemma *tensor-ell2-right-apply*[*simp*]: ‹*tensor-ell2-right* ψ *$_V$ φ = φ ⊗$_s$ ψ›
by (*transfer fixing*: ψ φ) *simp*

lemma *isometry-tensor-ell2-right*: ‹*isometry* (*tensor-ell2-right* ψ)› **if** ‹*norm* ψ = *1*›
by (*rule norm-preserving-isometry*) (*simp add*: *norm-tensor-ell2 that*)

lemma *isometry-tensor-ell2-left*: ‹*isometry* (*tensor-ell2-left* ψ)› **if** ‹*norm* ψ = *1*›
by (*rule norm-preserving-isometry*) (*simp add*: *norm-tensor-ell2 that*)

lemma *tensor-ell2-right-scale*: ‹*tensor-ell2-right* (a *$_C$ ψ) = a *$_C$ *tensor-ell2-right* ψ›
by *transfer* (*auto simp*: *tensor-ell2-scaleC2*)
lemma *tensor-ell2-left-scale*: ‹*tensor-ell2-left* (a *$_C$ ψ) = a *$_C$ *tensor-ell2-left* ψ›
by *transfer* (*auto simp*: *tensor-ell2-scaleC1*)

lemma *tensor-ell2-right-0*[*simp*]: ‹*tensor-ell2-right 0 = 0*›
by (*auto intro*!: *cblinfun-eqI*)
lemma *tensor-ell2-left-0*[*simp*]: ‹*tensor-ell2-left 0 = 0*›
by (*auto intro*!: *cblinfun-eqI*)

lemma *tensor-ell2-right-adj-apply*[*simp*]: ‹(*tensor-ell2-right* ψ*) *$_V$ (α ⊗$_s$ β) = (ψ ·$_C$ β) *$_C$ α›
by (*rule cinner-extensionality*) (*simp add*: *cinner-adj-right*)
lemma *tensor-ell2-left-adj-apply*[*simp*]: ‹(*tensor-ell2-left* ψ*) *$_V$ (α ⊗$_s$ β) = (ψ ·$_C$ α) *$_C$ β›
by (*rule cinner-extensionality*) (*simp add*: *cinner-adj-right*)

lemma *infsum-tensor-ell2-right*: ‹ψ ⊗$_s$ (∑$_\infty$x∈A. φ x) = (∑$_\infty$x∈A. ψ ⊗$_s$ φ x)›
proof −
consider (*summable*) ‹φ *summable-on A*› | (*summable'*) ‹ψ ≠ *0*› ‹(λx. ψ ⊗$_s$ φ x) *summable-on A*›
| (*ψ0*) ‹ψ = *0*›
| (*not-summable*) ‹¬ φ *summable-on A*› ‹¬ (λx. ψ ⊗$_s$ φ x) *summable-on A*›
by *auto*
then show *?thesis*
proof *cases*
case *summable*
then show *?thesis*
by (*rule infsum-bounded-linear*[*symmetric*, *unfolded o-def*, *rotated*])
(*intro bounded-linear-intros*)
next
case *summable'*
then have *: ‹(ψ /$_R$ (*norm* ψ)^2) ·$_C$ ψ = *1*›
by (*simp add*: *scaleR-scaleC cdot-square-norm*)
from *summable'*(*2*) **have** ‹(λx. (*tensor-ell2-left* (ψ /$_R$ (*norm* ψ)^2))* *$_V$ (ψ ⊗$_s$ φ x)) *summable-on A*›
by (*rule summable-on-bounded-linear*[*unfolded o-def*, *rotated*])

```
      (intro bounded-linear-intros)
    with ∗ have ‹φ summable-on A›
      by simp
    then show ?thesis
      by (rule infsum-bounded-linear[symmetric, unfolded o-def, rotated])
        (intro bounded-linear-intros)
  next
    case ψ0
    then show ?thesis
      by simp
  next
    case not-summable
    then show ?thesis
      by (simp add: infsum-not-exists)
  qed
qed

lemma infsum-tensor-ell2-left: ‹(∑∞x∈A. φ x) ⊗s ψ = (∑∞x∈A. φ x ⊗s ψ)›
proof −
  from infsum-tensor-ell2-right
  have ‹swap-ell2 ∗V (ψ ⊗s (∑∞x∈A. φ x)) = swap-ell2 ∗V (∑∞x∈A. ψ ⊗s φ x)›
    by metis
  then show ?thesis
    by (simp add: invertible-cblinfun-isometry flip: infsum-cblinfun-apply-invertible)
qed

lemma tensor-ell2-extensionality3:
  assumes (⋀s t u. a ∗V (s ⊗s t ⊗s u) = b ∗V (s ⊗s t ⊗s u))
  shows a = b
  by (rule equal-ket) (use assms in ‹auto simp flip: tensor-ell2-ket›)

lemma cblinfun-cinner-tensor-eqI:
  assumes ‹⋀ψ φ. (ψ ⊗s φ) •C (A ∗V (ψ ⊗s φ)) = (ψ ⊗s φ) •C (B ∗V (ψ ⊗s φ))›
  shows ‹A = B›
proof −
  define C where ‹C = A − B›
  from assms have assmC: ‹(ψ ⊗s φ) •C (C ∗V (ψ ⊗s φ)) = 0› for ψ φ
    by (simp add: C-def cblinfun.diff-left cinner-simps(3))

  have ‹(x ⊗s y) •C (C ∗V (z ⊗s w)) = 0› for x y z w

  proof −
    define d e f g h j k l m n p q
      where defs: ‹d = (x ⊗s y) •C (C ∗V z ⊗s w)›
        ‹e = (z ⊗s y) •C (C ∗V x ⊗s y)›
        ‹f = (x ⊗s w) •C (C ∗V x ⊗s y)›
        ‹g = (z ⊗s w) •C (C ∗V x ⊗s y)›
        ‹h = (x ⊗s y) •C (C ∗V z ⊗s y)›
        ‹j = (x ⊗s w) •C (C ∗V z ⊗s y)›
```

```
    ‹k = (z ⊗s w) ·C (C *V z ⊗s y)›
    ‹l = (z ⊗s w) ·C (C *V x ⊗s w)›
    ‹m = (x ⊗s y) ·C (C *V x ⊗s w)›
    ‹n = (z ⊗s y) ·C (C *V x ⊗s w)›
    ‹p = (z ⊗s y) ·C (C *V z ⊗s w)›
    ‹q = (x ⊗s w) ·C (C *V z ⊗s w)›

have constraint: ‹cnj α * e + cnj β * f + cnj β * cnj α * g + α * h + α * cnj β * j +
      α * cnj β * cnj α * k + β * m + β * cnj α * n + β * cnj β * cnj α * l +
      β * α * d + β * α * cnj α * p + β * α * cnj β * q = 0›
  (is ‹?lhs = -›) for α β
proof −
  from assms
  have ‹0 = ((x + α *C z) ⊗s (y + β *C w)) ·C (C *V ((x + α *C z) ⊗s (y + β *C w)))›
    by (simp add: assmC)
  also have ‹... = ?lhs›
    by (simp add: tensor-ell2-add1 tensor-ell2-add2 cinner-add-right cinner-add-left
        cblinfun.add-right tensor-ell2-scaleC1 tensor-ell2-scaleC2 semiring-class.distrib-left
        cblinfun.scaleC-right assmC defs flip: add.assoc mult.assoc)
  finally show ?thesis
    by simp
qed

have aux1: ‹a = 0 ⟹ b = 0 ⟹ a + b = 0› for a b :: complex
  by auto
have aux2: ‹a = 0 ⟹ b = 0 ⟹ a − b = 0› for a b :: complex
  by auto
have aux4: ‹2 * a = 0 ⟷ a = 0› for a :: complex
  by auto
have aux5: ‹8 = 2 * 2 * (2::complex)›
  by simp

from constraint[of 1 0]
have 1: ‹e + h = 0›
  by simp
from constraint[of i 0]
have 2: ‹h = e›
  by simp
from 1 2
have [simp]: ‹e = 0› ‹h = 0›
  by auto
from constraint[of 0 1]
have 3: ‹f + m = 0›
  by simp
from constraint[of 0 i]
have 4: ‹m = f›
  by simp
from 3 4
have [simp]: ‹m = 0› ‹f = 0›
```

by *auto*
from *constraint*[*of 1 1*]
have *5*: ‹$g + j + k + n + l + d + p + q = 0$›
by *simp*
from *constraint*[*of 1* ‹-1›]
have *6*: ‹$- g - j - k - n + l - d - p + q = 0$›
by *simp*
from *aux1*[*OF 5 6*]
have *7*: ‹$l + q = 0$›
by *algebra*
from *aux2*[*OF 5 7*]
have *8*: ‹$g + j + k + n + d + p = 0$›
by (*simp add*: *algebra-simps*)
from *constraint*[*of 1* i]
have *9*: ‹$- (\mathrm{i} * g) - \mathrm{i} * j - \mathrm{i} * k + \mathrm{i} * n + l + \mathrm{i} * d + \mathrm{i} * p + q = 0$›
by *simp*
from *constraint*[*of 1* ‹$-$i›]
have *10*: ‹$\mathrm{i} * g + \mathrm{i} * j + \mathrm{i} * k - \mathrm{i} * n + l - \mathrm{i} * d - \mathrm{i} * p + q = 0$›
by *simp*
from *aux2*[*OF 9 10*]
have *11*: ‹$n + d + p - k - j - g = 0$›
using *i-squared* **by** *algebra*
from *aux2*[*OF 8 11*]
have *12*: ‹$g + j + k = 0$›
by *algebra*
from *aux1*[*OF 8 11*]
have *13*: ‹$n + d + p = 0$›
by *algebra*
from *constraint*[*of* i *1*]
have *14*: ‹$\mathrm{i} * j - \mathrm{i} * g + k - \mathrm{i} * n - \mathrm{i} * l + \mathrm{i} * d + p + \mathrm{i} * q = 0$›
by *simp*
from *constraint*[*of* i ‹-1›]
have *15*: ‹$\mathrm{i} * g - \mathrm{i} * j - k + \mathrm{i} * n - \mathrm{i} * l - \mathrm{i} * d - p + \mathrm{i} * q = 0$›
by *simp*
from *aux1*[*OF 14 15*]
have [*simp*]: ‹$q = l$›
by *simp*
from *7*
have [*simp*]: ‹$q = 0$› ‹$l = 0$›
by *auto*
from *14*
have *16*: ‹$\mathrm{i} * j - \mathrm{i} * g + k - \mathrm{i} * n + \mathrm{i} * d + p = 0$›
by *simp*
from *constraint*[*of* ‹$-$i› *1*]
have *17*: ‹$\mathrm{i} * g - \mathrm{i} * j + k + \mathrm{i} * n - \mathrm{i} * d + p = 0$›
by *simp*
from *aux1*[*OF 16 17*]
have [*simp*]: ‹$k = - p$›
by *algebra*

```
  from aux2[OF 16 17]
  have 18: ‹j + d − n − g = 0›
    using i-squared by algebra
  from constraint[of ‹−i› 1]
  have 19: ‹i ∗ g − i ∗ j + i ∗ n − i ∗ d = 0›
    by (simp add: algebra-simps)
  from constraint[of ‹−i› ‹−1›]
  have 20: ‹i ∗ j − i ∗ g − i ∗ n + i ∗ d = 0›
    by (simp add: algebra-simps)
  from constraint[of i i]
  have 21: ‹j − g + n − d + 2 ∗ i ∗ p = 0›
    by (simp add: algebra-simps)
  from constraint[of i ‹−i›]
  have 22: ‹g − j − n + d − 2 ∗ i ∗ p = 0›
    by (simp add: algebra-simps)
  from constraint[of 2 1]
  have 23: ‹g + j + n + d = 0›
    using 12 13 ‹k = −p› by algebra
  from aux2[OF 23 18]
  have [simp]: ‹g = − n›
    by algebra
  from 23
  have [simp]: ‹j = − d›
    by (simp add: add-eq-0-iff2)
  have 8 ∗ (i ∗ p) + (4 ∗ (i ∗ d) + 4 ∗ (i ∗ n)) = 0
    using constraint[of 2 i] by simp
  hence 24: ‹2 ∗ p + d + n = 0›
    using complex-i-not-zero by algebra
  from aux2[OF 24 13]
  have [simp]: ‹p = 0›
    by simp
  then have [simp]: ‹k = 0›
    by auto
  from 12
  have ‹g = − j›
    by simp
  from 21
  have ‹d = − g›
    by auto

  show ‹d = 0›
    using refl[of d]
    apply (subst (asm) ‹d = − g›)
    apply (subst (asm) ‹g = − j›)
    apply (subst (asm) ‹j = − d›)
    by simp
qed
then show ?thesis
  by (auto intro!: equal-ket cinner-ket-eqI
```

simp: *C-def cblinfun.diff-left cinner-diff-right*
simp flip: *tensor-ell2-ket*)
qed

lemma *unitary-tensor-ell2-right-CARD-1*:
fixes ψ :: ‹'*a* :: {*CARD-1*,*enum*} *ell2*›
assumes ‹*norm* ψ = *1*›
shows ‹*unitary* (*tensor-ell2-right* ψ)›
proof (*rule unitaryI*)
show ‹*tensor-ell2-right* $\psi*$ o_{CL} *tensor-ell2-right* ψ = *id-cblinfun*›
by (*simp add*: *assms isometry-tensor-ell2-right*)
have $*$: ‹(ψ $\cdot_C$ φ) $*$ (φ $\cdot_C$ ψ) = φ $\cdot_C$ φ› **for** φ
proof −
define ψ' φ' **where** ‹ψ' = *1* $\cdot_C$ ψ› **and** ‹φ' = *1* $\cdot_C$ φ›
have ψ: ‹ψ = ψ' $*_C$ *1*›
by (*metis* ψ'*-def one-cinner-a-scaleC-one*)
have φ: ‹φ = φ' $*_C$ *1*›
by (*metis* φ'*-def one-cinner-a-scaleC-one*)
show *?thesis*
unfolding ψ φ
by (*metis* (*no-types*, *lifting*) *Groups.mult-ac*(*1*) ψ *assms cinner-simps*(*5*) *cinner-simps*(*6*)
norm-one of-complex-def of-complex-inner-1 power2-norm-eq-cinner)
qed
show ‹*tensor-ell2-right* ψ o_{CL} *tensor-ell2-right* $\psi*$ = *id-cblinfun*›
by (*rule cblinfun-cinner-tensor-eqI*) (*simp add*: $*$)
qed

14.2 Tensor product of operators on - *ell2*

definition *tensor-op* :: ‹('*a ell2*, '*b ell2*) *cblinfun* ⇒ ('*c ell2*, '*d ell2*) *cblinfun*
⇒ (('*a*×'*c*) *ell2*, ('*b*×'*d*) *ell2*) *cblinfun*› (**infixr** $\otimes_o$ *70*) **where**
‹*tensor-op M N* = *cblinfun-extension* (*range ket*) (λ*k*. *case* (*inv ket k*) *of* (*x*,*y*) ⇒ *tensor-ell2*
(*M* $*_V$ *ket x*) (*N* $*_V$ *ket y*))›

lemma
— Loosely following [7, Section IV.1]
fixes *a* :: ‹'*a*› **and** *b* :: ‹'*b*› **and** *c* :: ‹'*c*› **and** *d* :: ‹'*d*› **and** *M* :: ‹'*a ell2* ⇒$_{CL}$ '*b ell2*› **and** *N*
:: ‹'*c ell2* ⇒$_{CL}$ '*d ell2*›
shows *tensor-op-ell2*: ‹(*M* $\otimes_o$ *N*) $*_V$ (ψ $\otimes_s$ φ) = (*M* $*_V$ ψ) $\otimes_s$ (*N* $*_V$ φ)›
and *tensor-op-norm*: ‹*norm* (*M* $\otimes_o$ *N*) = *norm M* $*$ *norm N*›
proof −
define *S1* :: ‹('*a*×'*d*) *ell2 set*› **and** *f1 g1 extg1*
where ‹*S1* = *range ket*›
and ‹*f1 k* = (*case* (*inv ket k*) *of* (*x*,*y*) ⇒ *tensor-ell2* (*M* $*_V$ *ket x*) (*ket y*))›
and ‹*g1* = *cconstruct S1 f1*› **and** ‹*extg1* = *cblinfun-extension* (*cspan S1*) *g1*›
for *k*
define *S2* :: ‹('*a*×'*c*) *ell2 set*› **and** *f2 g2 extg2*
where ‹*S2* = *range ket*›
and ‹*f2 k* = (*case* (*inv ket k*) *of* (*x*,*y*) ⇒ *tensor-ell2* (*ket x*) (*N* $*_V$ *ket y*))›

```
    and ‹g2 = cconstruct S2 f2› and ‹extg2 = cblinfun-extension (cspan S2) g2›
   for k
  define tensorMN where ‹tensorMN = extg1 o_CL extg2›

  have extg1-ket: ‹extg1 *_V ket (x,y) = (M *_V ket x) ⊗_s ket y›
    and norm-extg1: ‹norm extg1 ≤ norm M› for x y
  proof −
    have [simp]: ‹cindependent S1›
      using S1-def cindependent-ket by blast
    have [simp]: ‹closure (cspan S1) = UNIV›
      by (simp add: S1-def)
    have [simp]: ‹ket (x, y) ∈ cspan S1› for x y
      by (simp add: S1-def complex-vector.span-base)
    have g1-f1: ‹g1 (ket (x,y)) = f1 (ket (x,y))› for x y
      by (metis S1-def ‹cindependent S1› complex-vector.construct-basis g1-def rangeI)
    have [simp]: ‹clinear g1›
      unfolding g1-def using ‹cindependent S1› by (rule complex-vector.linear-construct)
    then have g1-add: ‹g1 (x + y) = g1 x + g1 y› if ‹x ∈ cspan S1› and ‹y ∈ cspan S1› for
x y
      using clinear-iff by blast
    from ‹clinear g1› have g1-scale: ‹g1 (c *_C x) = c *_C g1 x› if ‹x ∈ cspan S1› for x c
      by (simp add: complex-vector.linear-scale)

    have g1-bounded: ‹norm (g1 ψ) ≤ norm M * norm ψ› if ‹ψ ∈ cspan S1› for ψ
    proof −
      from that obtain t r where ‹finite t› and ‹t ⊆ range ket› and ψ-tr: ‹ψ = (∑ a∈t. r a
*_C a)›
        by (smt (verit) complex-vector.span-explicit mem-Collect-eq S1-def)
      define X Y where ‹X = fst ` inv ket ` t› and ‹Y = snd ` inv ket ` t›
      have g1-ket: ‹g1 (ket (x,y)) = (M *_V ket x) ⊗_s ket y› for x y
        by (simp add: g1-def S1-def complex-vector.construct-basis f1-def)
      define ξ where ‹ξ y = (∑ x∈X. if (ket (x,y) ∈ t) then r (ket (x,y)) *_C ket x else 0)› for
y
      have ψξ: ‹ψ = (∑ y∈Y. ξ y ⊗_s ket y)›
      proof −
        have ‹(∑ y∈Y. ξ y ⊗_s ket y) = (∑ xy∈X × Y. if ket xy ∈ t then r (ket xy) *_C ket xy
else 0)›
          unfolding ξ-def tensor-ell2-sum-left
          by (subst sum.swap)
            (auto simp: sum.cartesian-product tensor-ell2-scaleC1 tensor-ell2-ket intro!: sum.cong)
        also have ‹... = (∑ xy∈ket ` (X × Y). if xy ∈ t then r xy *_C xy else 0)›
          by (subst sum.reindex) (auto simp add: inj-on-def)
        also have ‹... = ψ›
          unfolding ψ-tr
        proof (rule sum.mono-neutral-cong-right, goal-cases)
          case 2
          show t ⊆ ket ` (X × Y)
          proof
            fix x assume x ∈ t
```

```
        with ‹t ⊆ range ket› obtain a b where ab: x = ket (a, b)
          by fast
        also have ket (a, b) ∈ ket ‘ (X × Y)
          by (metis X-def Y-def ‹x ∈ t› ab f-inv-into-f fst-conv image-eqI
                ket-injective mem-Sigma-iff rangeI snd-conv)
        finally show x ∈ ket ‘ (X × Y) .
      qed
    qed (auto simp add: X-def Y-def ‹finite t›)
    finally show ?thesis
      by simp
  qed
  have ‹(norm (g1 ψ))² = (norm (∑ y∈Y. (M *V ξ y) ⊗s ket y))²›
    by (auto simp: ψξ complex-vector.linear-sum ξ-def tensor-ell2-sum-left
        complex-vector.linear-scale g1-ket tensor-ell2-scaleC1
        complex-vector.linear-0 tensor-ell2-ket
        intro!: sum.cong arg-cong[where f=norm])
  also have ‹... = (∑ y∈Y. (norm ((M *V ξ y) ⊗s ket y))²)›
    unfolding Y-def by (rule pythagorean-theorem-sum) (use ‹finite t› in auto)
  also have ‹... = (∑ y∈Y. (norm (M *V ξ y))²)›
    by (simp add: norm-tensor-ell2)
  also have ‹... ≤ (∑ y∈Y. (norm M * norm (ξ y))²)›
    by (meson norm-cblinfun norm-ge-zero power-mono sum-mono)
  also have ‹... = (norm M)² * (∑ y∈Y. (norm (ξ y ⊗s ket y))²)›
    by (simp add: power-mult-distrib norm-tensor-ell2 flip: sum-distrib-left)
  also have ‹... = (norm M)² * (norm (∑ y∈Y. ξ y ⊗s ket y))²›
    unfolding Y-def
    by (subst pythagorean-theorem-sum) (use ‹finite t› in auto)
  also have ‹... = (norm M)² * (norm ψ)²›
    using ψξ by fastforce
  finally show ‹norm (g1 ψ) ≤ norm M * norm ψ›
    by (metis mult-nonneg-nonneg norm-ge-zero power2-le-imp-le power-mult-distrib)
qed

have extg1-exists: ‹cblinfun-extension-exists (cspan S1) g1›
  by (rule cblinfun-extension-exists-bounded-dense[where B=‹norm M›])
    (use g1-add g1-scale g1-bounded in auto)

then show ‹extg1 *V ket (x,y) = (M *V ket x) ⊗s ket y› for x y
  by (simp add: extg1-def cblinfun-extension-apply g1-f1 f1-def)

from g1-add g1-scale g1-bounded
show ‹norm extg1 ≤ norm M›
  by (auto simp: extg1-def intro!: cblinfun-extension-norm-bounded-dense)
qed

have extg1-apply: ‹extg1 *V (ψ ⊗s φ) = (M *V ψ) ⊗s φ› for ψ φ
proof −
  have 1: ‹bounded-clinear (λa. extg1 *V (a ⊗s ket y))› for y
    by (intro bounded-clinear-cblinfun-apply bounded-clinear-tensor-ell22)
```

```
  have 2: ‹bounded-clinear (λa. (M *V a) ⊗s ket y)› for y :: 'd
  by (auto intro!: bounded-clinear-tensor-ell22[THEN bounded-clinear-compose] bounded-clinear-cblinfun-apply)
  have 3: ‹bounded-clinear (λa. extg1 *V (ψ ⊗s a))›
   by (intro bounded-clinear-cblinfun-apply bounded-clinear-tensor-ell21)
  have 4: ‹bounded-clinear ((⊗s) (M *V ψ))›
  by (auto intro!: bounded-clinear-tensor-ell21[THEN bounded-clinear-compose] bounded-clinear-cblinfun-apply)

  have eq-ket: ‹extg1 *V tensor-ell2 ψ (ket y) = tensor-ell2 (M *V ψ) (ket y)› for y
   by (rule bounded-clinear-eq-on-closure[where t=ψ and G=‹range ket›])
     (use 1 2 extg1-ket in ‹auto simp: tensor-ell2-ket›)
  show ?thesis
   by (rule bounded-clinear-eq-on-closure[where t=φ and G=‹range ket›])
     (use 3 4 eq-ket in auto)
 qed

 have extg2-ket: ‹extg2 *V ket (x,y) = ket x ⊗s (N *V ket y)›
  and norm-extg2: ‹norm extg2 ≤ norm N› for x y
 proof −
  have [simp]: ‹cindependent S2›
   using S2-def cindependent-ket by blast
  have [simp]: ‹closure (cspan S2) = UNIV›
   by (simp add: S2-def)
  have [simp]: ‹ket (x, y) ∈ cspan S2› for x y
   by (simp add: S2-def complex-vector.span-base)
  have g2-f2: ‹g2 (ket (x,y)) = f2 (ket (x,y))› for x y
   by (metis S2-def ‹cindependent S2› complex-vector.construct-basis g2-def rangeI)
  have [simp]: ‹clinear g2›
   unfolding g2-def using ‹cindependent S2› by (rule complex-vector.linear-construct)
  then have g2-add: ‹g2 (x + y) = g2 x + g2 y› if ‹x ∈ cspan S2› and ‹y ∈ cspan S2› for
x y
   using clinear-iff by blast
  from ‹clinear g2› have g2-scale: ‹g2 (c *C x) = c *C g2 x› if ‹x ∈ cspan S2› for x c
   by (simp add: complex-vector.linear-scale)

  have g2-bounded: ‹norm (g2 ψ) ≤ norm N * norm ψ› if ‹ψ ∈ cspan S2› for ψ
  proof −
   from that obtain t r where ‹finite t› and ‹t ⊆ range ket› and ψ-tr: ‹ψ = (∑ a∈t. r a
*C a)›
    by (smt (verit) complex-vector.span-explicit mem-Collect-eq S2-def)
   define X Y where ‹X = fst ‘ inv ket ‘ t› and ‹Y = snd ‘ inv ket ‘ t›
   have g2-ket: ‹g2 (ket (x,y)) = ket x ⊗s (N *V ket y)› for x y
    by (auto simp add: f2-def complex-vector.construct-basis g2-def S2-def)
   define ξ where ‹ξ x = (∑ y∈Y. if (ket (x,y) ∈ t) then r (ket (x,y)) *C ket y else 0)› for
x
   have ψξ: ‹ψ = (∑ x∈X. ket x ⊗s ξ x)›
   proof −
    have ‹(∑ x∈X. ket x ⊗s ξ x) = (∑ xy∈X × Y. if ket xy ∈ t then r (ket xy) *C ket xy
else 0)›
     by (auto simp: ξ-def tensor-ell2-sum-right sum.cartesian-product tensor-ell2-scaleC2
```

```
tensor-ell2-ket intro!: sum.cong)
      also have ‹... = (∑ xy∈ket ‘ (X × Y). if xy ∈ t then r xy *C xy else 0)›
        by (subst sum.reindex) (auto simp add: inj-on-def)
      also have ‹... = ψ›
        unfolding ψ-tr
      proof (rule sum.mono-neutral-cong-right, goal-cases)
        case 2
        show t ⊆ ket ‘ (X × Y)
        proof
          fix x assume x ∈ t
          with ‹t ⊆ range ket› obtain a b where ab: x = ket (a, b)
            by fast
          also have ket (a, b) ∈ ket ‘ (X × Y)
            by (metis X-def Y-def ‹x ∈ t› ab f-inv-into-f fst-conv image-eqI
                  ket-injective mem-Sigma-iff rangeI snd-conv)
          finally show x ∈ ket ‘ (X × Y) .
        qed
      qed (auto simp add: X-def Y-def ‹finite t›)
      finally show ?thesis
        by simp
    qed
    have ‹(norm (g2 ψ))² = (norm (∑ x∈X. ket x ⊗s (N *V ξ x)))²›
      by (auto simp: ψξ complex-vector.linear-sum ξ-def tensor-ell2-sum-right
          complex-vector.linear-scale g2-ket tensor-ell2-scaleC2
          complex-vector.linear-0 tensor-ell2-ket
          intro!: sum.cong arg-cong[where f=norm])
    also have ‹... = (∑ x∈X. (norm (ket x ⊗s (N *V ξ x)))²)›
      unfolding X-def by (rule pythagorean-theorem-sum) (use ‹finite t› in auto)
    also have ‹... = (∑ x∈X. (norm (N *V ξ x))²)›
      by (simp add: norm-tensor-ell2)
    also have ‹... ≤ (∑ x∈X. (norm N * norm (ξ x))²)›
      by (meson norm-cblinfun norm-ge-zero power-mono sum-mono)
    also have ‹... = (norm N)² * (∑ x∈X. (norm (ket x ⊗s ξ x))²)›
      by (simp add: power-mult-distrib norm-tensor-ell2 flip: sum-distrib-left)
    also have ‹... = (norm N)² * (norm (∑ x∈X. ket x ⊗s ξ x))²›
      unfolding X-def by (subst pythagorean-theorem-sum) (use ‹finite t› in auto)
    also have ‹... = (norm N)² * (norm ψ)²›
      using ψξ by fastforce
    finally show ‹norm (g2 ψ) ≤ norm N * norm ψ›
      by (metis mult-nonneg-nonneg norm-ge-zero power2-le-imp-le power-mult-distrib)
  qed

  have extg2-exists: ‹cblinfun-extension-exists (cspan S2) g2›
    by (rule cblinfun-extension-exists-bounded-dense[where B=‹norm N›])
      (use g2-add g2-scale g2-bounded in auto)

  then show ‹extg2 *V ket (x,y) = ket x ⊗s N *V ket y› for x y
    by (simp add: extg2-def cblinfun-extension-apply g2-f2 f2-def)
```

```
  from g2-add g2-scale g2-bounded
  show ‹norm extg2 ≤ norm N›
    by (auto simp: extg2-def intro!: cblinfun-extension-norm-bounded-dense)
qed

have extg2-apply: ‹extg2 *V (ψ ⊗s φ) = ψ ⊗s (N *V φ)› for ψ φ
proof −
  have 1: ‹bounded-clinear (λa. extg2 *V (ket x ⊗s a))› for x
    by (intro bounded-clinear-cblinfun-apply bounded-clinear-tensor-ell21)
  have 2: ‹bounded-clinear (λa. ket x ⊗s (N *V a))› for x :: 'a
   by (auto intro!: bounded-clinear-tensor-ell21[THEN bounded-clinear-compose] bounded-clinear-cblinfun-apply)
  have 3: ‹bounded-clinear (λa. extg2 *V (a ⊗s φ))›
    by (intro bounded-clinear-cblinfun-apply bounded-clinear-tensor-ell22)
  have 4: ‹ bounded-clinear (λa. a ⊗s (N *V φ))›
   by (auto intro!: bounded-clinear-tensor-ell22[THEN bounded-clinear-compose] bounded-clinear-cblinfun-apply)

  have eq-ket: ‹extg2 *V (ket x ⊗s φ) = ket x ⊗s (N *V φ)› for x
    by (rule bounded-clinear-eq-on-closure[where t=φ and G=‹range ket›])
       (use 1 2 extg2-ket in ‹auto simp: tensor-ell2-ket›)
  show ?thesis
    by (rule bounded-clinear-eq-on-closure[where t=ψ and G=‹range ket›])
       (use 3 4 eq-ket in auto)
qed

have tensorMN-apply: ‹tensorMN *V (ψ ⊗s φ) = (M *V ψ) ⊗s (N *V φ)› for ψ φ
  by (simp add: extg1-apply extg2-apply tensorMN-def)

have ‹cblinfun-extension-exists (range ket) (λk. case inv ket k of (x, y) ⇒ (M *V ket x) ⊗s
(N *V ket y))›
  by (rule cblinfun-extension-existsI[where B=tensorMN])
     (use tensorMN-apply[of ‹ket -› ‹ket -›] in ‹auto simp: tensor-ell2-ket›)

then have otimes-ket: ‹(M ⊗o N) *V (ket (a,c)) = (M *V ket a) ⊗s (N *V ket c)› for a c
  by (simp add: tensor-op-def cblinfun-extension-apply)

have tensorMN-otimes: ‹M ⊗o N = tensorMN›
  by (rule equal-ket)
     (use tensorMN-apply[of ‹ket -› ‹ket -›] in ‹auto simp: otimes-ket tensor-ell2-ket›)

show otimes-apply: ‹(M ⊗o N) *V (ψ ⊗s φ) = (M *V ψ) ⊗s (N *V φ)› for ψ φ
  by (simp add: tensorMN-apply tensorMN-otimes)

show ‹norm (M ⊗o N) = norm M * norm N›
proof (rule order.antisym)
  show ‹norm (M ⊗o N) ≤ norm M * norm N›
    unfolding tensorMN-otimes tensorMN-def
   by (smt (verit, best) mult-mono norm-cblinfun-compose norm-extg1 norm-extg2 norm-ge-zero)
  have ‹norm (M ⊗o N) ≥ norm M * norm N * ε› if ‹ε < 1› and ‹ε > 0› for ε
  proof −
```

obtain ψa **where** *1*: ‹*norm* $(M *_V \psi a) \geq$ *norm* $M *$ *sqrt* ε› **and** ‹*norm* $\psi a = 1$›
by (*atomize-elim*, *rule cblinfun-norm-approx-witness-mult*[**where** ε=‹*sqrt* ε› **and** A=M])
(*use* ‹$\varepsilon < 1$› **in** *auto*)
obtain ψb **where** *2*: ‹*norm* $(N *_V \psi b) \geq$ *norm* $N *$ *sqrt* ε› **and** ‹*norm* $\psi b = 1$›
by (*atomize-elim*, *rule cblinfun-norm-approx-witness-mult*[**where** ε=‹*sqrt* ε› **and** A=N])
(*use* ‹$\varepsilon < 1$› **in** *auto*)
have ‹*norm* $((M \otimes_o N) *_V (\psi a \otimes_s \psi b)) /$ *norm* $(\psi a \otimes_s \psi b) =$ *norm* $((M \otimes_o N) *_V (\psi a \otimes_s \psi b))$›
using ‹*norm* $\psi a = 1$› ‹*norm* $\psi b = 1$›
by (*simp add*: *norm-tensor-ell2*)
also have ‹$\ldots =$ *norm* $(M *_V \psi a) *$ *norm* $(N *_V \psi b)$›
by (*simp add*: *norm-tensor-ell2 otimes-apply*)
also from *1 2* **have** ‹$\ldots \geq$ (*norm* $M *$ *sqrt* ε) $*$ (*norm* $N *$ *sqrt* ε)› (**is** ‹- $\geq \ldots$›)
by (*rule mult-mono'*) (*use* ‹$\varepsilon > 0$› **in** *auto*)
also have ‹$\ldots =$ *norm* $M *$ *norm* $N * \varepsilon$›
using ‹$\varepsilon > 0$› **by** *force*
finally show *?thesis*
using *cblinfun-norm-geqI* **by** *blast*
qed
then show ‹*norm* $(M \otimes_o N) \geq$ *norm* $M *$ *norm* N›
by (*metis field-le-mult-one-interval mult.commute*)
qed
qed

lemma *tensor-op-ket*: ‹*tensor-op* $M\ N *_V$ (*ket* (a,c)) = *tensor-ell2* $(M *_V$ *ket* $a)$ $(N *_V$ *ket* $c)$›
by (*metis tensor-ell2-ket tensor-op-ell2*)

lemma *comp-tensor-op*: (*tensor-op* $a\ b$) o_{CL} (*tensor-op* $c\ d$) = *tensor-op* $(a\ o_{CL}\ c)$ $(b\ o_{CL}\ d)$
for a :: $'e$ *ell2* $\Rightarrow_{CL}$ $'c$ *ell2* **and** b :: $'f$ *ell2* $\Rightarrow_{CL}$ $'d$ *ell2* **and**
c :: $'a$ *ell2* $\Rightarrow_{CL}$ $'e$ *ell2* **and** d :: $'b$ *ell2* $\Rightarrow_{CL}$ $'f$ *ell2*
by (*rule equal-ket*) (*auto simp flip*: *tensor-ell2-ket simp*: *tensor-op-ell2*)

lemma *tensor-op-left-add*: ‹$(x + y) \otimes_o b = x \otimes_o b + y \otimes_o b$›
for $x\ y$ :: ‹$'a$ *ell2* $\Rightarrow_{CL}$ $'c$ *ell2*› **and** b :: ‹$'b$ *ell2* $\Rightarrow_{CL}$ $'d$ *ell2*›
by (*auto intro*!: *equal-ket simp*: *tensor-op-ket plus-cblinfun.rep-eq tensor-ell2-add1*)

lemma *tensor-op-right-add*: ‹$b \otimes_o (x + y) = b \otimes_o x + b \otimes_o y$›
for $x\ y$ :: ‹$'a$ *ell2* $\Rightarrow_{CL}$ $'c$ *ell2*› **and** b :: ‹$'b$ *ell2* $\Rightarrow_{CL}$ $'d$ *ell2*›
by (*auto intro*!: *equal-ket simp*: *tensor-op-ket plus-cblinfun.rep-eq tensor-ell2-add2*)

lemma *tensor-op-scaleC-left*: ‹$(c *_C x) \otimes_o b = c *_C (x \otimes_o b)$›
for x :: ‹$'a$ *ell2* $\Rightarrow_{CL}$ $'c$ *ell2*› **and** b :: ‹$'b$ *ell2* $\Rightarrow_{CL}$ $'d$ *ell2*›
by (*auto intro*!: *equal-ket simp*: *tensor-op-ket tensor-ell2-scaleC1*)

lemma *tensor-op-scaleC-right*: ‹$b \otimes_o (c *_C x) = c *_C (b \otimes_o x)$›
for x :: ‹$'a$ *ell2* $\Rightarrow_{CL}$ $'c$ *ell2*› **and** b :: ‹$'b$ *ell2* $\Rightarrow_{CL}$ $'d$ *ell2*›
by (*auto intro*!: *equal-ket simp*: *tensor-op-ket tensor-ell2-scaleC2*)

lemma *tensor-op-bounded-cbilinear*[*simp*]: ‹*bounded-cbilinear tensor-op*›

by (*auto intro*!: *bounded-cbilinear.intro exI*[*of - 1*]
simp: *tensor-op-left-add tensor-op-right-add tensor-op-scaleC-left tensor-op-scaleC-right tensor-op-norm*)

lemma *tensor-op-cbilinear*[*simp*]: ‹*cbilinear tensor-op*›
by (*simp add*: *bounded-cbilinear.add-left bounded-cbilinear.add-right cbilinear-def clinearI tensor-op-scaleC-left tensor-op-scaleC-right*)

lemma *tensor-butter*: ‹*butterfly* (*ket i*) (*ket j*) ⊗$_o$ *butterfly* (*ket k*) (*ket l*) = *butterfly* (*ket* (*i*,*k*)) (*ket* (*j*,*l*))›
by (*rule equal-ket*)
(*auto simp flip*: *tensor-ell2-ket simp*: *tensor-op-ell2 butterfly-def tensor-ell2-scaleC1 tensor-ell2-scaleC2*)

lemma *cspan-tensor-op-butter*: ‹*cspan* {*tensor-op* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*))| (*i*::-::*finite*) (*j*::-::*finite*) (*k*::-::*finite*) (*l*::-::*finite*). *True*} = *UNIV*›
unfolding *tensor-butter*
by (*subst cspan-butterfly-ket*[*symmetric*]) (*metis surj-pair*)

lemma *cindependent-tensor-op-butter*: ‹*cindependent* {*tensor-op* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*))| *i j k l*. *True*}›
unfolding *tensor-butter*
using *cindependent-butterfly-ket*
by (*smt* (*z3*) *Collect-mono-iff complex-vector.independent-mono*)

lift-definition *right-amplification* :: ‹('*a ell2* ⇒$_{CL}$ '*b ell2*) ⇒$_{CL}$ (('*a*×'*c*) *ell2* ⇒$_{CL}$ ('*b*×'*c*) *ell2*)› **is**
‹λ*a*. *a* ⊗$_o$ *id-cblinfun*›
by (*simp add*: *bounded-cbilinear.bounded-clinear-left*)

lift-definition *left-amplification* :: ‹('*a ell2* ⇒$_{CL}$ '*b ell2*) ⇒$_{CL}$ (('*c*×'*a*) *ell2* ⇒$_{CL}$ ('*c*×'*b*) *ell2*)› **is**
‹λ*a*. *id-cblinfun* ⊗$_o$ *a*›
by (*simp add*: *bounded-cbilinear.bounded-clinear-right*)

lemma *sandwich-tensor-ell2-right*: ‹*sandwich* (*tensor-ell2-right* ψ*) *$_V$ *a* ⊗$_o$ *b* = (ψ ·$_C$ (*b* *$_V$ ψ)) *$_C$ *a*›
by (*rule cblinfun-eqI*) (*simp add*: *sandwich-apply tensor-op-ell2*)
lemma *sandwich-tensor-ell2-left*: ‹*sandwich* (*tensor-ell2-left* ψ*) *$_V$ *a* ⊗$_o$ *b* = (ψ ·$_C$ (*a* *$_V$ ψ)) *$_C$ *b*›
by (*rule cblinfun-eqI*) (*simp add*: *sandwich-apply tensor-op-ell2*)

lemma *tensor-op-adjoint*: ‹(*tensor-op a b*)* = *tensor-op* (*a**) (*b**)›
by (*rule cinner-ket-adjointI*[*symmetric*])
(*auto simp flip*: *tensor-ell2-ket simp*: *tensor-op-ell2 cinner-adj-left*)

lemma *has-sum-id-tensor-butterfly-ket*: ‹((λ*i*. (*id-cblinfun* ⊗$_o$ *butterfly* (*ket i*) (*ket i*)) *$_V$ ψ)

has-sum ψ) *UNIV*›
proof −
have ∗: ‹($\sum$ *i*∈*F*. (*id-cblinfun* ⊗$_o$ *butterfly* (*ket i*) (*ket i*)) ∗$_V$ ψ) = *trunc-ell2* (*UNIV* × *F*) ψ› **if** ‹*finite F*› **for** *F*
proof (*rule Rep-ell2-inject*[*THEN iffD1*], *rule ext*, *rename-tac xy*)
fix *xy* :: ‹'*b* × '*a*›
obtain *x y* **where** *xy*: ‹*xy* = (*x*,*y*)›
by *fastforce*
have ‹*Rep-ell2* ($\sum$ *i*∈*F*. (*id-cblinfun* ⊗$_o$ *butterfly* (*ket i*) (*ket i*)) ∗$_V$ ψ) *xy*
= *ket xy* ·$_C$ ($\sum$ *i*∈*F*. (*id-cblinfun* ⊗$_o$ *butterfly* (*ket i*) (*ket i*)) ∗$_V$ ψ)›
by (*simp add*: *cinner-ket-left*)
also have ‹... = ($\sum$ *i*∈*F*. *ket xy* ·$_C$ ((*id-cblinfun* ⊗$_o$ *butterfly* (*ket i*) (*ket i*)) ∗$_V$ ψ))›
using *cinner-sum-right* **by** *blast*
also have ‹... = ($\sum$ *i*∈*F*. *ket xy* ·$_C$ ((*id-cblinfun* ⊗$_o$ *butterfly* (*ket i*) (*ket i*))∗ ∗$_V$ ψ))›
by (*simp add*: *tensor-op-adjoint*)
also have ‹... = ($\sum$ *i*∈*F*. ((*id-cblinfun* ⊗$_o$ *butterfly* (*ket i*) (*ket i*)) ∗$_V$ *ket xy*) ·$_C$ ψ)›
by (*meson cinner-adj-right*)
also have ‹... = *of-bool* (*y*∈*F*) ∗ (*ket xy* ·$_C$ ψ)›
by (*subst sum-single*[**where** *i*=*y*])
(*auto simp*: *xy tensor-op-ell2 cinner-ket that simp flip*: *tensor-ell2-ket*)
also have ‹... = *of-bool* (*y*∈*F*) ∗ (*Rep-ell2* ψ *xy*)›
by (*simp add*: *cinner-ket-left*)
also have ‹... = *Rep-ell2* (*trunc-ell2* (*UNIV* × *F*) ψ) *xy*›
by (*simp add*: *trunc-ell2.rep-eq xy*)
finally show ‹*Rep-ell2* ($\sum$ *i*∈*F*. (*id-cblinfun* ⊗$_o$ *butterfly* (*ket i*) (*ket i*)) ∗$_V$ ψ) *xy* = ...›
by −
qed

have ‹((λ*F*. *trunc-ell2 F* ψ) ⟶ *trunc-ell2 UNIV* ψ) (*filtermap* ((×)*UNIV*) (*finite-subsets-at-top UNIV*))›
by (*rule trunc-ell2-lim-general*)
(*auto simp add*: *filterlim-def le-filter-def eventually-finite-subsets-at-top eventually-filtermap intro*!: *exI*[**where** *x*=‹*snd* ‘ -›])
then have ‹((λ*F*. *trunc-ell2* (*UNIV*×*F*) ψ) ⟶ ψ) (*finite-subsets-at-top UNIV*)›
by (*simp add*: *filterlim-def filtermap-filtermap*)
then have ‹((λ*F*. ($\sum$ *i*∈*F*. (*id-cblinfun* ⊗$_o$ *butterfly* (*ket i*) (*ket i*)) ∗$_V$ ψ)) ⟶ ψ) (*finite-subsets-at-top UNIV*)›
by (*rule Lim-transform-eventually*)
(*simp add*: ∗ *eventually-finite-subsets-at-top-weakI*)
then show *?thesis*
by (*simp add*: *has-sum-def*)
qed

lemma *tensor-op-dense*: ‹*cstrong-operator-topology closure-of* (*cspan* {*a* ⊗$_o$ *b* | *a b*. *True*}) = *UNIV*›
— [7, p.185 (10)], but we prove it directly.
proof (*intro order.antisym subset-UNIV subsetI*)
fix *c* :: ‹('*a* × '*b*) *ell2* ⇒$_{CL}$ ('*c* × '*d*) *ell2*›

define c' **where** ‹$c'\ i\ j = (\mathit{tensor\text{-}ell2\text{-}right}\ (\mathit{ket}\ i))* \circ_{CL} c \circ_{CL} \mathit{tensor\text{-}ell2\text{-}right}\ (\mathit{ket}\ j)$› **for** $i\ j$
define $AB :: ‹((\prime a \times \prime b)\ \mathit{ell2} \Rightarrow_{CL} (\prime c \times \prime d)\ \mathit{ell2})\ \mathit{set}›$ **where**
‹$AB = \mathit{cstrong\text{-}operator\text{-}topology\ closure\text{-}of}\ (\mathit{cspan}\ \{a \otimes_o b \mid a\ b.\ \mathit{True}\})$›

have [*simp*]: ‹*closedin cstrong-operator-topology AB*›
by (*simp add*: *AB-def*)
have [*simp*]: ‹*csubspace AB*›
using *AB-def sot-closure-is-csubspace'* **by** *blast*

have $*$: ‹$c'\ i\ j \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ j) = (\mathit{id\text{-}cblinfun} \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ i)) \circ_{CL} c \circ_{CL} (\mathit{id\text{-}cblinfun} \otimes_o \mathit{butterfly}\ (\mathit{ket}\ j)\ (\mathit{ket}\ j))$› **for** $i\ j$
proof (*rule equal-ket*, *rule cinner-ket-eqI*, *rename-tac a b*)
fix $a :: ‹\prime a \times \prime b›$ **and** $b :: ‹\prime c \times \prime d›$
obtain *bi bj ai aj* **where** *b*: ‹$b = (bi,bj)$› **and** *a*: ‹$a = (ai,aj)$›
by (*meson surj-pair*)
have ‹$\mathit{ket}\ b \cdot_C ((c'\ i\ j \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ j)) *_V \mathit{ket}\ a) = \mathit{of\text{-}bool}\ (j = aj \wedge bj = i) * ((\mathit{ket}\ bi \otimes_s \mathit{ket}\ i) \cdot_C (c *_V \mathit{ket}\ ai \otimes_s \mathit{ket}\ aj))$›
by (*auto simp add*: *a b tensor-op-ell2 cinner-ket c'-def cinner-adj-right*
simp flip: *tensor-ell2-ket*)
also have ‹$\ldots = \mathit{ket}\ b \cdot_C ((\mathit{id\text{-}cblinfun} \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ i) \circ_{CL} c \circ_{CL} \mathit{id\text{-}cblinfun} \otimes_o \mathit{butterfly}\ (\mathit{ket}\ j)\ (\mathit{ket}\ j)) *_V \mathit{ket}\ a)$›
apply (*subst asm-rl*[*of* ‹$\mathit{id\text{-}cblinfun} \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ i) = (\mathit{id\text{-}cblinfun} \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ i))*$›])
subgoal
by (*simp add*: *tensor-op-adjoint*)
subgoal
by (*auto simp*: *a b tensor-op-ell2 cinner-adj-right cinner-ket*
simp flip: *tensor-ell2-ket*)
done
finally show ‹$\mathit{ket}\ b \cdot_C ((c'\ i\ j \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ j)) *_V \mathit{ket}\ a) =$
$\mathit{ket}\ b \cdot_C ((\mathit{id\text{-}cblinfun} \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ i) \circ_{CL} c \circ_{CL} \mathit{id\text{-}cblinfun} \otimes_o \mathit{butterfly}\ (\mathit{ket}\ j)\ (\mathit{ket}\ j)) *_V \mathit{ket}\ a)$›
by −
qed

have ‹$c'\ i\ j \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ j) \in AB$› **for** $i\ j$
proof −
have ‹$c'\ i\ j \otimes_o \mathit{butterfly}\ (\mathit{ket}\ i)\ (\mathit{ket}\ j) \in \{a \otimes_o b \mid a\ b.\ \mathit{True}\}$›
by *auto*
also have ‹$\ldots \subseteq \mathit{cspan}\ \ldots$›
by (*simp add*: *complex-vector.span-superset*)
also have ‹$\ldots \subseteq \mathit{cstrong\text{-}operator\text{-}topology\ closure\text{-}of}\ \ldots$›
by (*rule closure-of-subset*) *simp*
also have ‹$\ldots = AB$›
by (*simp add*: *AB-def*)
finally show *?thesis*
by *simp*
qed

with $*$ **have** *AB1*: ‹(*id-cblinfun* $\otimes_o$ *butterfly* (*ket i*) (*ket i*)) o_{CL} *c* o_{CL} (*id-cblinfun* $\otimes_o$ *butterfly* (*ket j*) (*ket j*)) $\in$ *AB*› **for** *i j*
by *simp*
have ‹((λ*i*. ((*id-cblinfun* $\otimes_o$ *butterfly* (*ket i*) (*ket i*)) o_{CL} *c* o_{CL} (*id-cblinfun* $\otimes_o$ *butterfly* (*ket j*) (*ket j*))) $*_V$ ψ)
has-sum (*c* o_{CL} (*id-cblinfun* $\otimes_o$ *butterfly* (*ket j*) (*ket j*))) $*_V$ ψ) *UNIV*› **for** *j* ψ
by (*simp add*: *has-sum-id-tensor-butterfly-ket*)
then have *AB2*: ‹(*c* o_{CL} (*id-cblinfun* $\otimes_o$ *butterfly* (*ket j*) (*ket j*))) $\in$ *AB*› **for** *j*
by (*rule has-sum-closed-cstrong-operator-topology*[*rotated* −*1*]) (*use AB1* **in** *auto*)

have ‹((λ*j*. (*c* o_{CL} (*id-cblinfun* $\otimes_o$ *butterfly* (*ket j*) (*ket j*))) $*_V$ ψ) *has-sum c* $*_V$ ψ) *UNIV*›
for ψ
by (*simp add*: *has-sum-cblinfun-apply has-sum-id-tensor-butterfly-ket*)
then show *AB3*: ‹*c* $\in$ *AB*›
by (*rule has-sum-closed-cstrong-operator-topology*[*rotated* −*1*]) (*use AB2* **in** *auto*)
qed

lemma *tensor-extensionality-finite*:
fixes *F G* :: ‹(('*a*::*finite* × '*b*::*finite*) *ell2*) ⇒$_{CL}$ (('*c*::*finite* × '*d*::*finite*) *ell2*)) ⇒ '*e*::*complex-vector*›
assumes [*simp*]: *clinear F clinear G*
assumes *tensor-eq*: (⋀*a b*. *F* (*tensor-op a b*) = *G* (*tensor-op a b*))
shows *F* = *G*
proof (*rule ext*, *rule complex-vector.linear-eq-on-span*[**where** *f*=*F* **and** *g*=*G*])
show ‹*clinear F*› **and** ‹*clinear G*›
using *assms* **by** (*simp-all add*: *cbilinear-def*)
show ‹*x* ∈ *cspan* {*tensor-op* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*))| *i j k l*. *True*}›

for *x* :: ‹('*a* × '*b*) *ell2* ⇒$_{CL}$ ('*c* × '*d*) *ell2*›
using *cspan-tensor-op-butter* **by** *auto*
show ‹*F x* = *G x*› **if** ‹*x* ∈ {*tensor-op* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*)) |*i j k l*. *True*}› **for** *x*
using *that* **by** (*auto simp*: *tensor-eq*)
qed

lemma *tensor-id*[*simp*]: ‹*tensor-op id-cblinfun id-cblinfun* = *id-cblinfun*›
by (*rule equal-ket*) (*auto simp flip*: *tensor-ell2-ket simp*: *tensor-op-ell2*)

lemma *tensor-butterfly*: *tensor-op* (*butterfly* ψ ψ') (*butterfly* φ φ') = *butterfly* (*tensor-ell2* ψ φ) (*tensor-ell2* ψ' φ')
by (*rule equal-ket*)
(*auto simp flip*: *tensor-ell2-ket simp*: *tensor-op-ell2 butterfly-def tensor-ell2-scaleC1 tensor-ell2-scaleC2*)

definition *tensor-lift* :: ‹(('*a1*::*finite ell2* ⇒$_{CL}$ '*a2*::*finite ell2*) ⇒ ('*b1*::*finite ell2* ⇒$_{CL}$ '*b2*::*finite ell2*) ⇒ '*c*)
⇒ ((('*a1*×'*b1*) *ell2* ⇒$_{CL}$ ('*a2*×'*b2*) *ell2*) ⇒ '*c*::*complex-normed-vector*)›
where
tensor-lift F2 = (*SOME G*. *clinear G* ∧ (∀ *a b*. *G* (*tensor-op a b*) = *F2 a b*))

lemma
fixes *F2* :: *'a::finite ell2* $\Rightarrow_{CL}$ *'b::finite ell2*
$\Rightarrow$ *'c::finite ell2* $\Rightarrow_{CL}$ *'d::finite ell2*
$\Rightarrow$ *'e::complex-normed-vector*
assumes *cbilinear F2*
shows *tensor-lift-clinear*: *clinear* (*tensor-lift F2*)
and *tensor-lift-correct*: ‹(λ*a b*. *tensor-lift F2* (*a* ⊗$_o$ *b*)) = *F2*›
proof −
define *F2'* *t4* φ **where**
‹*F2'* = *tensor-lift F2*› **and**
‹*t4* = (λ(*i*,*j*,*k*,*l*). *tensor-op* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*)))› **and**
‹φ *m* = (*let* (*i*,*j*,*k*,*l*) = *inv t4 m in F2* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*)))›
for *m*
have *t4inj*: *x* = *y* **if** *t4 x* = *t4 y* **for** *x y*
proof (*rule ccontr*)
obtain *i j k l* **where** *x*: *x* = (*i*,*j*,*k*,*l*) **by** (*meson prod-cases4*)
obtain *i' j' k' l'* **where** *y*: *y* = (*i'*,*j'*,*k'*,*l'*) **by** (*meson prod-cases4*)
have *1*: *bra* (*i*,*k*) *$*_V$* *t4 x* *$*_V$* *ket* (*j*,*l*) = *1*
by (*auto simp*: *t4-def x tensor-op-ell2 butterfly-def cinner-ket simp flip*: *tensor-ell2-ket*)
assume ‹*x* ≠ *y*›
then have *2*: *bra* (*i*,*k*) *$*_V$* *t4 y* *$*_V$* *ket* (*j*,*l*) = *0*
by (*auto simp*: *t4-def x y tensor-op-ell2 butterfly-def cinner-ket*
simp flip: *tensor-ell2-ket*)
from *1 2 that*
show *False*
by *auto*
qed
have ‹φ (*tensor-op* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*))) = *F2* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*))› **for** *i j k l*
apply (*subst asm-rl*[*of* ‹*tensor-op* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*)) = *t4* (*i*,*j*,*k*,*l*)›])
subgoal by (*simp add*: *t4-def*)
subgoal by (*auto simp add*: *injI t4inj inv-f-f* φ*-def*)
done

have ∗: ‹*range t4* = {*tensor-op* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*)) |*i j k l*. *True*}›
by (*force simp*: *case-prod-beta t4-def image-iff*)

have *cblinfun-extension-exists* (*range t4*) φ
by (*rule cblinfun-extension-exists-finite-dim*)
(*use* ∗ *cindependent-tensor-op-butter cspan-tensor-op-butter* **in** *auto*)

then obtain *G* **where** *G*: ‹*G* *$*_V$* (*t4* (*i*,*j*,*k*,*l*)) = *F2* (*butterfly* (*ket i*) (*ket j*)) (*butterfly* (*ket k*) (*ket l*))› **for** *i j k l*
unfolding *cblinfun-extension-exists-def*
by (*metis* (*no-types*, *lifting*) *t4inj* φ*-def f-inv-into-f rangeI split-conv*)

```
  have ∗: ‹G ∗V tensor-op (butterfly (ket i) (ket j)) (butterfly (ket k) (ket l)) = F2 (butterfly
(ket i) (ket j)) (butterfly (ket k) (ket l))› for i j k l
    using G by (auto simp: t4-def)
  have ∗: ‹G ∗V tensor-op a (butterfly (ket k) (ket l)) = F2 a (butterfly (ket k) (ket l))› for a
k l
    apply (rule complex-vector.linear-eq-on-span[where g=‹λa. F2 a -› and B=‹{butterfly (ket
k) (ket l)|k l. True}›])
    unfolding cspan-butterfly-ket
     using ∗ apply (auto intro!: clinear-compose[unfolded o-def, where f=‹λa. tensor-op a -›
and g=‹(∗V) G›])
     apply (metis cbilinear-def tensor-op-cbilinear)
    using assms unfolding cbilinear-def by blast
  have G-F2: ‹G ∗V tensor-op a b = F2 a b› for a b
     apply (rule complex-vector.linear-eq-on-span[where g=‹F2 a› and B=‹{butterfly (ket k)
(ket l)|k l. True}›])
    unfolding cspan-butterfly-ket
    using ∗ apply (auto simp: cblinfun.add-right clinearI
                      intro!: clinear-compose[unfolded o-def, where f=‹tensor-op a› and g=‹(∗V)
G›])
    apply (meson cbilinear-def tensor-op-cbilinear)
    using assms unfolding cbilinear-def by blast

  have ‹clinear F2' ∧ (∀ a b. F2' (tensor-op a b) = F2 a b)›
    unfolding F2'-def tensor-lift-def
    apply (rule someI[where x=‹(∗V) G› and P=‹λG. clinear G ∧ (∀ a b. G (tensor-op a b)
= F2 a b)›])
    using G-F2 by (simp add: cblinfun.add-right clinearI)

  then show ‹clinear F2'› and ‹(λa b. tensor-lift F2 (tensor-op a b)) = F2›
    unfolding F2'-def by auto
qed


lemma tensor-op-nonzero:
  fixes a :: ‹'a ell2 ⇒CL 'c ell2› and b :: ‹'b ell2 ⇒CL 'd ell2›
  assumes ‹a ≠ 0› and ‹b ≠ 0›
  shows ‹a ⊗o b ≠ 0›
proof −
  from ‹a ≠ 0› obtain i where i: ‹a ∗V ket i ≠ 0›
    by (metis cblinfun.zero-left equal-ket)
  from ‹b ≠ 0› obtain j where j: ‹b ∗V ket j ≠ 0›
    by (metis cblinfun.zero-left equal-ket)
  from i j have ijneq0: ‹(a ∗V ket i) ⊗s (b ∗V ket j) ≠ 0›
    by (simp add: tensor-ell2-nonzero)
  have ‹(a ∗V ket i) ⊗s (b ∗V ket j) = (a ⊗o b) ∗V ket (i,j)›
    by (simp add: tensor-op-ket)
  with ijneq0 show ‹a ⊗o b ≠ 0›
    by force
qed
```

lemma *inj-tensor-ell2-left*: ‹*inj* (λ*a*::′*a ell2*. *a* ⊗*s b*)› **if** ‹*b* ≠ *0*› **for** *b* :: ‹′*b ell2*›
proof (*rule injI*, *rule ccontr*)
fix *x y* :: ‹′*a ell2*›
assume *eq*: ‹*x* ⊗*s b* = *y* ⊗*s b*›
assume *neq*: ‹*x* ≠ *y*›
define *a* **where** ‹*a* = *x* − *y*›
from *neq a-def* **have** *neq0*: ‹*a* ≠ *0*›
by *auto*
with ‹*b* ≠ *0*› **have** ‹*a* ⊗*s b* ≠ *0*›
by (*simp add*: *tensor-ell2-nonzero*)
then have ‹*x* ⊗*s b* ≠ *y* ⊗*s b*›
unfolding *a-def*
by (*metis add-cancel-left-left diff-add-cancel tensor-ell2-add1*)
with *eq* **show** *False*
by *auto*
qed

lemma *inj-tensor-ell2-right*: ‹*inj* (λ*b*::′*b ell2*. *a* ⊗*s b*)› **if** ‹*a* ≠ *0*› **for** *a* :: ‹′*a ell2*›
proof (*rule injI*, *rule ccontr*)
fix *x y* :: ‹′*b ell2*›
assume *eq*: ‹*a* ⊗*s x* = *a* ⊗*s y*›
assume *neq*: ‹*x* ≠ *y*›
define *b* **where** ‹*b* = *x* − *y*›
from *neq b-def* **have** *neq0*: ‹*b* ≠ *0*›
by *auto*
with ‹*a* ≠ *0*› **have** ‹*a* ⊗*s b* ≠ *0*›
by (*simp add*: *tensor-ell2-nonzero*)
then have ‹*a* ⊗*s x* ≠ *a* ⊗*s y*›
unfolding *b-def*
by (*metis add-cancel-left-left diff-add-cancel tensor-ell2-add2*)
with *eq* **show** *False*
by *auto*
qed

lemma *inj-tensor-left*: ‹*inj* (λ*a*::′*a ell2* ⇒*CL* ′*c ell2*. *a* ⊗*o b*)› **if** ‹*b* ≠ *0*› **for** *b* :: ‹′*b ell2* ⇒*CL* ′*d ell2*›
proof (*rule injI*, *rule ccontr*)
fix *x y* :: ‹′*a ell2* ⇒*CL* ′*c ell2*›
assume *eq*: ‹*x* ⊗*o b* = *y* ⊗*o b*›
assume *neq*: ‹*x* ≠ *y*›
define *a* **where** ‹*a* = *x* − *y*›
from *neq a-def* **have** *neq0*: ‹*a* ≠ *0*›
by *auto*
with ‹*b* ≠ *0*› **have** ‹*a* ⊗*o b* ≠ *0*›
by (*simp add*: *tensor-op-nonzero*)
then have ‹*x* ⊗*o b* ≠ *y* ⊗*o b*›
unfolding *a-def*
by (*metis add-cancel-left-left diff-add-cancel tensor-op-left-add*)

with *eq* **show** *False*
by *auto*
qed

lemma *inj-tensor-right*: ‹*inj* (λ*b*::'*b ell2* ⇒$_{CL}$ '*c ell2*. *a* ⊗$_o$ *b*)› **if** ‹*a* ≠ *0*› **for** *a* :: ‹'*a ell2* ⇒$_{CL}$ '*d ell2*›
proof (*rule injI*, *rule ccontr*)
fix *x y* :: ‹'*b ell2* ⇒$_{CL}$ '*c ell2*›
assume *eq*: ‹*a* ⊗$_o$ *x* = *a* ⊗$_o$ *y*›
assume *neq*: ‹*x* ≠ *y*›
define *b* **where** ‹*b* = *x* − *y*›
from *neq b-def* **have** *neq0*: ‹*b* ≠ *0*›
by *auto*
with ‹*a* ≠ *0*› **have** ‹*a* ⊗$_o$ *b* ≠ *0*›
by (*simp add*: *tensor-op-nonzero*)
then have ‹*a* ⊗$_o$ *x* ≠ *a* ⊗$_o$ *y*›
unfolding *b-def*
by (*metis add-cancel-left-left diff-add-cancel tensor-op-right-add*)
with *eq* **show** *False*
by *auto*
qed

lemma *tensor-ell2-almost-injective*:
assumes ‹*tensor-ell2 a b* = *tensor-ell2 c d*›
assumes ‹*a* ≠ *0*›
shows ‹∃γ. *b* = γ ∗$_C$ *d*›
proof −
from ‹*a* ≠ *0*› **obtain** *i* **where** *i*: ‹*cinner* (*ket i*) *a* ≠ *0*›
by (*metis cinner-eq-zero-iff cinner-ket-left ell2-pointwise-ortho*)
have ‹*cinner* (*ket i* ⊗$_s$ *ket j*) (*a* ⊗$_s$ *b*) = *cinner* (*ket i* ⊗$_s$ *ket j*) (*c* ⊗$_s$ *d*)› **for** *j*
using *assms* **by** *simp*
then have *eq2*: ‹(*cinner* (*ket i*) *a*) ∗ (*cinner* (*ket j*) *b*) = (*cinner* (*ket i*) *c*) ∗ (*cinner* (*ket j*) *d*)› **for** *j*
by (*metis tensor-ell2-inner-prod*)
then obtain γ **where** ‹*cinner* (*ket i*) *c* = γ ∗ *cinner* (*ket i*) *a*›
by (*metis i eq-divide-eq*)
with *eq2* **have** ‹(*cinner* (*ket i*) *a*) ∗ (*cinner* (*ket j*) *b*) = (*cinner* (*ket i*) *a*) ∗ (γ ∗ *cinner* (*ket j*) *d*)› **for** *j*
by *simp*
then have ‹*cinner* (*ket j*) *b* = *cinner* (*ket j*) (γ ∗$_C$ *d*)› **for** *j*
using *i* **by** *force*
then have ‹*b* = γ ∗$_C$ *d*›
by (*simp add*: *cinner-ket-eqI*)
then show *?thesis*
by *auto*
qed

lemma *tensor-op-almost-injective*:

fixes $a\ c :: \langle 'a\ ell2 \Rightarrow_{CL} 'b\ ell2 \rangle$
and $b\ d :: \langle 'c\ ell2 \Rightarrow_{CL} 'd\ ell2 \rangle$
assumes $\langle tensor\text{-}op\ a\ b = tensor\text{-}op\ c\ d \rangle$
assumes $\langle a \neq 0 \rangle$
shows $\langle \exists \gamma.\ b = \gamma *_C d \rangle$
proof (*cases* $\langle d = 0 \rangle$)
case *False*
from $\langle a \neq 0 \rangle$ **obtain** ψ **where** ψ: $\langle a *_V \psi \neq 0 \rangle$
by (*metis cblinfun.zero-left cblinfun-eqI*)
have $\langle (a \otimes_o b)\ (\psi \otimes_s \varphi) = (c \otimes_o d)\ (\psi \otimes_s \varphi) \rangle$ **for** φ
using *assms* **by** *simp*
then have *eq2*: $\langle (a\ \psi) \otimes_s (b\ \varphi) = (c\ \psi) \otimes_s (d\ \varphi) \rangle$ **for** φ
by (*simp add*: *tensor-op-ell2*)
then have *eq2'*: $\langle (d\ \varphi) \otimes_s (c\ \psi) = (b\ \varphi) \otimes_s (a\ \psi) \rangle$ **for** φ
by (*metis swap-ell2-tensor*)
from *False* **obtain** $\varphi 0$ **where** $\varphi 0$: $\langle d\ \varphi 0 \neq 0 \rangle$
by (*metis cblinfun.zero-left cblinfun-eqI*)
obtain γ **where** $\langle c\ \psi = \gamma *_C a\ \psi \rangle$
apply *atomize-elim*
using *eq2'* $\varphi 0$ **by** (*rule tensor-ell2-almost-injective*)
with *eq2* **have** $\langle (a\ \psi) \otimes_s (b\ \varphi) = (a\ \psi) \otimes_s (\gamma *_C d\ \varphi) \rangle$ **for** φ
by (*simp add*: *tensor-ell2-scaleC1 tensor-ell2-scaleC2*)
then have $\langle b\ \varphi = \gamma *_C d\ \varphi \rangle$ **for** φ
by (*smt* (*verit*, *best*) ψ *complex-vector.scale-cancel-right tensor-ell2-almost-injective tensor-ell2-nonzero tensor-ell2-scaleC2*)
then have $\langle b = \gamma *_C d \rangle$
by (*simp add*: *cblinfun-eqI*)
then show *?thesis*
by *auto*
next
case *True*
then have $\langle c \otimes_o d = 0 \rangle$
by (*metis add-cancel-right-left tensor-op-right-add*)
then have $\langle a \otimes_o b = 0 \rangle$
using *assms*(*1*) **by** *presburger*
with $\langle a \neq 0 \rangle$ **have** $\langle b = 0 \rangle$
by (*meson tensor-op-nonzero*)
then show *?thesis*
by *auto*
qed

lemma *clinear-tensor-left*[*simp*]: $\langle clinear\ (\lambda a.\ a \otimes_o b :: \text{-}\ ell2 \Rightarrow_{CL} \text{-}\ ell2) \rangle$
apply (*rule clinearI*)
apply (*rule tensor-op-left-add*)
by (*rule tensor-op-scaleC-left*)

lemma *clinear-tensor-right*[*simp*]: $\langle clinear\ (\lambda b.\ a \otimes_o b :: \text{-}\ ell2 \Rightarrow_{CL} \text{-}\ ell2) \rangle$
apply (*rule clinearI*)
apply (*rule tensor-op-right-add*)

by (*rule tensor-op-scaleC-right*)

lemma *tensor-op-0-left*[*simp*]: ‹*tensor-op 0 x* = (*0* :: (*'a*∗*'b*) *ell2* ⇒$_{CL}$ (*'c*∗*'d*) *ell2*)›
apply (*rule equal-ket*)
by (*auto simp flip*: *tensor-ell2-ket simp*: *tensor-op-ell2*)

lemma *tensor-op-0-right*[*simp*]: ‹*tensor-op x 0* = (*0* :: (*'a*∗*'b*) *ell2* ⇒$_{CL}$ (*'c*∗*'d*) *ell2*)›
apply (*rule equal-ket*)
by (*auto simp flip*: *tensor-ell2-ket simp*: *tensor-op-ell2*)

lemma *bij-tensor-ell2-one-dim-left*:
assumes ‹$\psi \neq 0$›
shows ‹*bij* (λ*x*::*'b ell2*. (ψ :: *'a*::*CARD-1 ell2*) ⊗$_s$ *x*)›
proof (*rule bijI*)
show ‹*inj* (λ*x*::*'b ell2*. (ψ :: *'a*::*CARD-1 ell2*) ⊗$_s$ *x*)›
using *assms* **by** (*rule inj-tensor-ell2-right*)
have ‹∃ *x*. ψ ⊗$_s$ *x* = φ› **for** φ :: ‹(*'a*∗*'b*) *ell2*›
proof (*use assms* **in** *transfer*)
fix ψ :: ‹*'a* ⇒ *complex*› **and** φ :: ‹*'a*∗*'b* ⇒ *complex*›
assume ‹*has-ell2-norm* φ› **and** ‹$\psi \neq$ (λ-. *0*)›
define *c* **where** ‹*c* = ψ *undefined*›
then have ‹ψ *a* = *c*› **for** *a*
by (*subst everything-the-same*[*of - undefined*]) *simp*
with ‹$\psi \neq$ (λ-. *0*)› **have** ‹*c* ≠ *0*›
by *auto*

define *x* **where** ‹*x j* = φ (*undefined*, *j*) / *c*› **for** *j*
have ‹(λ(*i*, *j*). ψ *i* ∗ *x j*) = φ›
proof (*rule ext*, *safe*)
fix *a* :: *'a* **and** *b* :: *'b*
show ψ *a* ∗ *x b* = φ (*a*, *b*)
using ‹*c* ≠ *0*› **by** (*simp add*: *c-def x-def everything-the-same*[*of a undefined*])
qed
moreover have ‹*has-ell2-norm x*›
proof −
have ‹(λ(*i*,*j*). (φ (*i*,*j*))^2) *abs-summable-on UNIV*›
using ‹*has-ell2-norm* φ› *has-ell2-norm-def* **by** *auto*
then have ‹(λ(*i*,*j*). (φ (*i*,*j*))^2) *abs-summable-on Pair undefined* ‘ *UNIV*›
using *summable-on-subset-banach* **by** *blast*
then have ‹(λ*j*. (φ (*undefined*,*j*))^2) *abs-summable-on UNIV*›
by (*subst* (*asm*) *summable-on-reindex*) (*auto simp*: *o-def inj-def*)
then have ‹(λ*j*. (φ (*undefined*,*j*) / *c*)^2) *abs-summable-on UNIV*›
by (*simp add*: *divide-inverse power-mult-distrib norm-mult summable-on-cmult-left*)
then have ‹(λ*j*. (*x j*)^2) *abs-summable-on UNIV*›
by (*simp add*: *x-def*)
then show *?thesis*
using *has-ell2-norm-def* **by** *blast*
qed
ultimately show ‹∃ *x*∈*Collect has-ell2-norm*. (λ(*i*, *j*). ψ *i* ∗ *x j*) = φ›

```
    by (intro bexI[where x=x]) auto
  qed

  then show ‹surj (λx::'b ell2. (ψ :: 'a::CARD-1 ell2) ⊗s x)›
    by (metis surj-def)
qed

lemma bij-tensor-op-one-dim-left:
  fixes a :: ‹'a::{CARD-1,enum} ell2 ⇒CL 'b::{CARD-1,enum} ell2›
  assumes ‹a ≠ 0›
  shows ‹bij (λx::'c ell2 ⇒CL 'd ell2. a ⊗o x)›
proof −
  have [simp]: ‹bij (Pair (undefined::'a))›
    by (rule o-bij[of snd]) auto
  have [simp]: ‹bij (Pair (undefined::'b))›
    by (rule o-bij[of snd]) auto
  define t where ‹t x = a ⊗o x› for x :: ‹'c ell2 ⇒CL 'd ell2›
  define u :: ‹'c ell2 ⇒CL ('a×'c) ell2› where ‹u = classical-operator (Some o Pair undefined)›
  define v :: ‹'d ell2 ⇒CL ('b×'d) ell2› where ‹v = classical-operator (Some o Pair undefined)›
  have [simp]: ‹unitary u› ‹unitary v›
    by (simp-all add: u-def v-def)
  have u-ket[simp]: ‹u *V ket x = ket (undefined, x)› for x
    by (simp add: u-def classical-operator-ket classical-operator-exists-inj inj-def)
  have uadj-ket[simp]: ‹u* *V ket (z, x) = ket x› for x z
    by (subst everything-the-same[of - undefined])
       (metis (no-types, opaque-lifting) u-ket cinner-adj-right cinner-ket-eqI cinner-ket-same
orthogonal-ket prod.inject)
  have v-ket[simp]: ‹v *V ket x = ket (undefined, x)› for x
    by (simp add: v-def classical-operator-ket classical-operator-exists-inj inj-def)
  have [simp]: ‹v *V x = ket undefined ⊗s x› for x
    by (rule fun-cong[where x=x], rule bounded-clinear-equal-ket)
      (auto simp add: bounded-clinear-tensor-ell21 cblinfun.bounded-clinear-right tensor-ell2-ket)
  define a' :: complex where ‹a' = one-dim-iso a›
  from assms have ‹a' ≠ 0›
    using a'-def one-dim-iso-of-zero' by auto
  have a-a': ‹a = of-complex a'›
    by (simp add: a'-def)
  have ‹t x *V ket (i,j) = (a' *C v o_CL x o_CL u*) *V ket (i,j)› for x i j
    apply (simp add: t-def)
    apply (simp add: ket-CARD-1-is-1 tensor-op-ell2 flip: tensor-ell2-ket)
   by (metis a'-def one-cblinfun-apply-one one-dim-scaleC-1 scaleC-cblinfun.rep-eq tensor-ell2-scaleC1)

  then have t: ‹t x = (a' *C v o_CL x o_CL u*)› for x
    by (intro equal-ket) auto
  define s where ‹s y = (inverse a' *C (v)* o_CL y o_CL u)› for y
  have ‹s (t x) = (a' * inverse a') *C (((v)* o_CL v) o_CL x o_CL (u* o_CL u))› for x
    apply (simp add: s-def t cblinfun-compose-assoc)
    by (simp flip: cblinfun-compose-assoc)?
  also have ‹... x = x› for x
```

```
    using ‹a' ≠ 0› by simp
  finally have ‹s o t = id›
    by auto
  have ‹t (s y) = (a' * inverse a') *C ((v oCL (v)*) oCL y oCL (u oCL u*))› for y
    apply (simp add: s-def t cblinfun-compose-assoc)
    by (simp flip: cblinfun-compose-assoc)?
  also have ‹... y = y› for y
    using ‹a' ≠ 0› by simp
  finally have ‹t o s = id›
    by auto
  from ‹s o t = id› ‹t o s = id›
  show ‹bij t›
    using o-bij by blast
qed

lemma bij-tensor-op-one-dim-right:
  assumes ‹b ≠ 0›
 shows ‹bij (λx::'c ell2 ⇒CL 'd ell2. x ⊗o (b :: 'a::{CARD-1,enum} ell2 ⇒CL 'b::{CARD-1,enum}
ell2))›
    (is ‹bij ?f›)
proof −
  let ?sf = ‹(λx. swap-ell2 oCL (?f x) oCL swap-ell2)›
  let ?s = ‹(λx. swap-ell2 oCL x oCL swap-ell2)›
  let ?g = ‹(λx::'c ell2 ⇒CL 'd ell2. (b :: 'a::{CARD-1,enum} ell2 ⇒CL 'b::{CARD-1,enum}
ell2) ⊗o x)›
  have ‹?sf = ?g›
    by (auto intro!: ext tensor-ell2-extensionality simp add: swap-ell2-tensor tensor-op-ell2)
  have ‹bij ?g›
    using assms by (rule bij-tensor-op-one-dim-left)
  have ‹?s o ?sf = ?f›
    apply (auto intro!: ext simp: cblinfun-assoc-left)
    by (auto simp: cblinfun-assoc-right)?
  also have ‹bij ?s›
    apply (rule o-bij[where g=‹(λx. swap-ell2 oCL x oCL swap-ell2)›])
     apply (auto intro!: ext simp: cblinfun-assoc-left)
    by (auto simp: cblinfun-assoc-right)?
  show ‹bij ?f›
    apply (subst ‹?s o ?sf = ?f›[symmetric], subst ‹?sf = ?g›)
    using ‹bij ?g› ‹bij ?s› by (rule bij-comp)
qed

lemma overlapping-tensor:
  fixes a23 :: ‹('a2*'a3) ell2 ⇒CL ('b2*'b3) ell2›
    and b12 :: ‹('a1*'a2) ell2 ⇒CL ('b1*'b2) ell2›
  assumes eq: ‹butterfly ψ ψ' ⊗o a23 = assoc-ell2 oCL (b12 ⊗o butterfly φ φ') oCL assoc-ell2*›
  assumes ‹ψ ≠ 0› ‹ψ' ≠ 0› ‹φ ≠ 0› ‹φ' ≠ 0›
  shows ‹∃ c. butterfly ψ ψ' ⊗o a23 = butterfly ψ ψ' ⊗o c ⊗o butterfly φ φ'›
proof −
  let ?id1 = ‹id-cblinfun :: unit ell2 ⇒CL unit ell2›
```

note *id-cblinfun-eq-1*[*simp del*]
define d **where** ‹d = *butterfly* ψ ψ' $\otimes_o$ $a23$›

define ψ_n ψ_n' $a23_n$ **where** ‹$\psi_n = \psi$ /$_C$ *norm* ψ› **and** ‹$\psi_n' = \psi'$ /$_C$ *norm* ψ'› **and** ‹$a23_n$ = *norm* ψ *$_C$ *norm* ψ' *$_C$ $a23$›
have [*simp*]: ‹*norm* $\psi_n = 1$› ‹*norm* $\psi_n' = 1$›
using ‹$\psi \neq 0$› ‹$\psi' \neq 0$› **by** (*auto simp*: ψ_n-*def* ψ_n'-*def norm-inverse*)
have *n1*: ‹*butterfly* ψ_n ψ_n' $\otimes_o$ $a23_n$ = *butterfly* ψ ψ' $\otimes_o$ $a23$›
by (*auto simp*: ψ_n-*def* ψ_n'-*def* $a23_n$-*def tensor-op-scaleC-left tensor-op-scaleC-right field-simps*)

define φ_n φ_n' $b12_n$ **where** ‹$\varphi_n = \varphi$ /$_C$ *norm* φ› **and** ‹$\varphi_n' = \varphi'$ /$_C$ *norm* φ'› **and** ‹$b12_n$ = *norm* φ *$_C$ *norm* φ' *$_C$ $b12$›
have [*simp*]: ‹*norm* $\varphi_n = 1$› ‹*norm* $\varphi_n' = 1$›
using ‹$\varphi \neq 0$› ‹$\varphi' \neq 0$› **by** (*auto simp*: φ_n-*def* φ_n'-*def norm-inverse*)
have *n2*: ‹$b12_n$ $\otimes_o$ *butterfly* φ_n φ_n' = $b12$ $\otimes_o$ *butterfly* φ φ'›
by (*auto simp*: φ_n-*def* φ_n'-*def* $b12_n$-*def tensor-op-scaleC-left tensor-op-scaleC-right field-simps*)

define c' :: ‹(*unit**'*a2***unit*) *ell2* ⇒$_{CL}$ (*unit**'*b2***unit*) *ell2*›
where ‹c' = (*vector-to-cblinfun* ψ_n $\otimes_o$ *id-cblinfun* $\otimes_o$ *vector-to-cblinfun* φ_n)* o$_{CL}$ d
o$_{CL}$ (*vector-to-cblinfun* ψ_n' $\otimes_o$ *id-cblinfun* $\otimes_o$ *vector-to-cblinfun* φ_n')›

define c'' :: ‹'*a2 ell2* ⇒$_{CL}$ '*b2 ell2*›
where ‹c'' = *inv* (λc''. *id-cblinfun* $\otimes_o$ c'' $\otimes_o$ *id-cblinfun*) c'›

have *: ‹*bij* (λc''::'*a2 ell2* ⇒$_{CL}$ '*b2 ell2*. *?id1* $\otimes_o$ c'' $\otimes_o$ *?id1*)›
by (*subst asm-rl*[*of* ‹- = (λ*x*. *id-cblinfun* $\otimes_o$ *x*) *o* (λc''. c'' $\otimes_o$ *id-cblinfun*)›])
(*auto intro*!: *bij-comp bij-tensor-op-one-dim-left bij-tensor-op-one-dim-right*)

have c'-c'': ‹c' = *?id1* $\otimes_o$ c'' $\otimes_o$ *?id1*›
unfolding c''-*def*
by (*rule surj-f-inv-f*[**where** *y*=c', *symmetric*]) (*use* * **in** ‹*rule bij-is-surj*›)

define c :: ‹'*a2 ell2* ⇒$_{CL}$ '*b2 ell2*›
where ‹c = c'' /$_C$ *norm* ψ /$_C$ *norm* ψ' /$_C$ *norm* φ /$_C$ *norm* φ'›

have *aux*: ‹*assoc-ell2** o$_{CL}$ (*assoc-ell2* o$_{CL}$ *x* o$_{CL}$ *assoc-ell2**) o$_{CL}$ *assoc-ell2* = *x*› **for** *x*
apply (*simp add*: *cblinfun-assoc-left*)
by (*simp add*: *cblinfun-assoc-right*)*?*
have *aux2*: ‹(*assoc-ell2* o$_{CL}$ ((*x* $\otimes_o$ *y*) $\otimes_o$ *z*) o$_{CL}$ *assoc-ell2**) = *x* $\otimes_o$ (*y* $\otimes_o$ *z*)› **for** *x y z*
by (*intro equal-ket*) (*auto simp flip*: *tensor-ell2-ket simp*: *assoc-ell2'-tensor assoc-ell2-tensor tensor-op-ell2*)

have ‹d = (*butterfly* ψ_n ψ_n $\otimes_o$ *id-cblinfun*) o$_{CL}$ d o$_{CL}$ (*butterfly* ψ_n' ψ_n' $\otimes_o$ *id-cblinfun*)›
by (*auto simp*: *d-def n1*[*symmetric*] *comp-tensor-op cnorm-eq-1*[*THEN iffD1*])
also have ‹... = (*butterfly* ψ_n ψ_n $\otimes_o$ *id-cblinfun*) o$_{CL}$ *assoc-ell2* o$_{CL}$ ($b12_n$ $\otimes_o$ *butterfly* φ_n φ_n')
o$_{CL}$ *assoc-ell2** o$_{CL}$ (*butterfly* ψ_n' ψ_n' $\otimes_o$ *id-cblinfun*)›
by (*auto simp*: *d-def eq n2 cblinfun-assoc-left*)
also have ‹... = (*butterfly* ψ_n ψ_n $\otimes_o$ *id-cblinfun*) o$_{CL}$ *assoc-ell2* o$_{CL}$

((*id-cblinfun* $\otimes_o$ *butterfly* φ_n φ_n) o_{CL} (*b12*$_n$ $\otimes_o$ *butterfly* φ_n $\varphi_n{}'$) o_{CL} (*id-cblinfun* $\otimes_o$ *butterfly* $\varphi_n{}'$ $\varphi_n{}'$))
o_{CL} *assoc-ell2*$*$ o_{CL} (*butterfly* $\psi_n{}'$ $\psi_n{}'$ $\otimes_o$ *id-cblinfun*)›
by (*auto simp*: *comp-tensor-op cnorm-eq-1*[*THEN iffD1*])
also have ‹... = (*butterfly* ψ_n ψ_n $\otimes_o$ *id-cblinfun*) o_{CL} *assoc-ell2* o_{CL}
((*id-cblinfun* $\otimes_o$ *butterfly* φ_n φ_n) o_{CL} (*assoc-ell2*$*$ o_{CL} *d* o_{CL} *assoc-ell2*) o_{CL} (*id-cblinfun* $\otimes_o$ *butterfly* $\varphi_n{}'$ $\varphi_n{}'$))
o_{CL} *assoc-ell2*$*$ o_{CL} (*butterfly* $\psi_n{}'$ $\psi_n{}'$ $\otimes_o$ *id-cblinfun*)›
by (*auto simp*: *d-def n2 eq aux*)
also have ‹... = ((*butterfly* ψ_n ψ_n $\otimes_o$ *id-cblinfun*) o_{CL} (*assoc-ell2* o_{CL} (*id-cblinfun* $\otimes_o$ *butterfly* φ_n φ_n) o_{CL} *assoc-ell2*$*$))
o_{CL} *d* o_{CL} ((*assoc-ell2* o_{CL} (*id-cblinfun* $\otimes_o$ *butterfly* $\varphi_n{}'$ $\varphi_n{}'$) o_{CL} *assoc-ell2*$*$) o_{CL} (*butterfly* $\psi_n{}'$ $\psi_n{}'$ $\otimes_o$ *id-cblinfun*))›
by (*auto simp*: *sandwich-def cblinfun-assoc-left*)
also have ‹... = (*butterfly* ψ_n ψ_n $\otimes_o$ *id-cblinfun* $\otimes_o$ *butterfly* φ_n φ_n)
o_{CL} *d* o_{CL} (*butterfly* $\psi_n{}'$ $\psi_n{}'$ $\otimes_o$ *id-cblinfun* $\otimes_o$ *butterfly* $\varphi_n{}'$ $\varphi_n{}'$)›
apply (*simp only*: *tensor-id*[*symmetric*] *comp-tensor-op aux2*)
by (*simp add*: *cnorm-eq-1*[*THEN iffD1*])
also have ‹... = (*vector-to-cblinfun* ψ_n $\otimes_o$ *id-cblinfun* $\otimes_o$ *vector-to-cblinfun* φ_n)
o_{CL} *c'* o_{CL} (*vector-to-cblinfun* $\psi_n{}'$ $\otimes_o$ *id-cblinfun* $\otimes_o$ *vector-to-cblinfun* $\varphi_n{}'$)$*$›
apply (*simp add*: *c'-def butterfly-def-one-dim*[**where** *'c=unit ell2*] *cblinfun-assoc-left comp-tensor-op tensor-op-adjoint cnorm-eq-1*[*THEN iffD1*])
by (*simp add*: *cblinfun-assoc-right comp-tensor-op*)
also have ‹... = *butterfly* ψ_n $\psi_n{}'$ $\otimes_o$ *c''* $\otimes_o$ *butterfly* φ_n $\varphi_n{}'$›
by (*simp add*: *c'-c'' comp-tensor-op tensor-op-adjoint butterfly-def-one-dim*[*symmetric*])
also have ‹... = *butterfly* ψ ψ' $\otimes_o$ *c* $\otimes_o$ *butterfly* φ φ'›
by (*simp add*: ψ_n*-def* $\psi_n{}'$*-def* φ_n*-def* $\varphi_n{}'$*-def c-def tensor-op-scaleC-left tensor-op-scaleC-right*)
finally have *d-c*: ‹*d* = *butterfly* ψ ψ' $\otimes_o$ *c* $\otimes_o$ *butterfly* φ φ'›
by −
then show *?thesis*
by (*auto simp*: *d-def*)
qed

lemma *tensor-op-pos*: ‹*a* $\otimes_o$ *b* $\geq$ *0*› **if** [*simp*]: ‹*a* $\geq$ *0*› ‹*b* $\geq$ *0*›
for *a* :: ‹*'a ell2* $\Rightarrow_{CL}$ *'a ell2*› **and** *b* :: ‹*'b ell2* $\Rightarrow_{CL}$ *'b ell2*›
— [8, Lemma 18]
proof −
have ‹(*sqrt-op a* $\otimes_o$ *sqrt-op b*)$*$ o_{CL} (*sqrt-op a* $\otimes_o$ *sqrt-op b*) = *a* $\otimes_o$ *b*›
by (*simp add*: *tensor-op-adjoint comp-tensor-op positive-hermitianI*)
then show ‹*a* $\otimes_o$ *b* $\geq$ *0*›
by (*metis positive-cblinfun-squareI*)
qed

lemma *abs-op-tensor*: ‹*abs-op* (*a* $\otimes_o$ *b*) = *abs-op a* $\otimes_o$ *abs-op b*›
— [8, Lemma 18]
proof −
have ‹(*abs-op a* $\otimes_o$ *abs-op b*)$*$ o_{CL} (*abs-op a* $\otimes_o$ *abs-op b*) = (*a* $\otimes_o$ *b*)$*$ o_{CL} (*a* $\otimes_o$ *b*)›
by (*simp add*: *tensor-op-adjoint comp-tensor-op abs-op-def positive-cblinfun-squareI posi-*

tive-hermitianI)
then show *?thesis*
by (*metis abs-opI abs-op-pos tensor-op-pos*)
qed

lemma *trace-class-tensor*: ‹*trace-class* (*a* ⊗*o* *b*)› **if** ‹*trace-class a*› **and** ‹*trace-class b*›
— [8, Lemma 32]
proof −
from ‹*trace-class a*›
have *a*: ‹(λ*x*. *ket x* •*C* (*abs-op a* ∗*V* *ket x*)) *abs-summable-on UNIV*›
by (*auto simp add*: *trace-class-iff-summable*[*OF is-onb-ket*] *summable-on-reindex o-def*)
from ‹*trace-class b*›
have *b*: ‹(λ*y*. *ket y* •*C* (*abs-op b* ∗*V* *ket y*)) *abs-summable-on UNIV*›
by (*auto simp add*: *trace-class-iff-summable*[*OF is-onb-ket*] *summable-on-reindex o-def*)
from *a b* **have** ‹(λ(*x*,*y*). (*ket x* •*C* (*abs-op a* ∗*V* *ket x*)) ∗ (*ket y* •*C* (*abs-op b* ∗*V* *ket y*)))
abs-summable-on UNIV × *UNIV*›
by (*rule abs-summable-times*)
then have ‹(λ(*x*,*y*). (*ket x* ⊗*s* *ket y*) •*C* ((*abs-op a* ⊗*o* *abs-op b*) ∗*V* (*ket x* ⊗*s* *ket y*)))
abs-summable-on UNIV × *UNIV*›
by (*simp add*: *tensor-op-ell2 case-prod-unfold flip*: *tensor-ell2-ket*)
then have ‹(λ*xy*. *ket xy* •*C* ((*abs-op a* ⊗*o* *abs-op b*) ∗*V* *ket xy*)) *abs-summable-on UNIV*›
by (*simp add*: *case-prod-beta tensor-ell2-ket*)
then have ‹(λ*xy*. *ket xy* •*C* (*abs-op* (*a* ⊗*o* *b*) ∗*V* *ket xy*)) *abs-summable-on UNIV*›
by (*simp add*: *abs-op-tensor*)
then show ‹*trace-class* (*a* ⊗*o* *b*)›
by (*auto simp add*: *trace-class-iff-summable*[*OF is-onb-ket*] *summable-on-reindex o-def*)
qed

lemma *swap-tensor-op*[*simp*]: ‹*swap-ell2* *oCL* (*a* ⊗*o* *b*) *oCL* *swap-ell2* = *b* ⊗*o* *a*›
by (*auto intro*!: *equal-ket simp add*: *tensor-op-ell2 simp flip*: *tensor-ell2-ket*)

lemma *swap-tensor-op-sandwich*[*simp*]: ‹*sandwich swap-ell2* (*a* ⊗*o* *b*) = *b* ⊗*o* *a*›
by (*simp add*: *sandwich-apply*)

lemma *swap-ell2-commute-tensor-op*:
‹*swap-ell2* *oCL* (*a* ⊗*o* *b*) = (*b* ⊗*o* *a*) *oCL* *swap-ell2*›
by (*auto intro*!: *tensor-ell2-extensionality simp*: *tensor-op-ell2*)

lemma *trace-class-tensor-op-swap*: ‹*trace-class* (*a* ⊗*o* *b*) ⟷ *trace-class* (*b* ⊗*o* *a*)›
proof (*rule iffI*)
assume ‹*trace-class* (*a* ⊗*o* *b*)›
then have ‹*trace-class* (*swap-ell2* *oCL* (*a* ⊗*o* *b*) *oCL* *swap-ell2*)›
using *trace-class-comp-left trace-class-comp-right* **by** *blast*
then show ‹*trace-class* (*b* ⊗*o* *a*)›
by *simp*
next
assume ‹*trace-class* (*b* ⊗*o* *a*)›
then have ‹*trace-class* (*swap-ell2* *oCL* (*b* ⊗*o* *a*) *oCL* *swap-ell2*)›
using *trace-class-comp-left trace-class-comp-right* **by** *blast*

then show ‹*trace-class* $(a \otimes_o b)$›
by *simp*
qed

lemma *trace-class-tensor-iff*: ‹*trace-class* $(a \otimes_o b) \longleftrightarrow$ (*trace-class a* $\wedge$ *trace-class b*) $\vee$ $a = 0$ $\vee$ $b = 0$›
proof (*intro iffI*)
show ‹*trace-class a* $\wedge$ *trace-class b* $\vee$ $a = 0$ $\vee$ $b = 0 \Longrightarrow$ *trace-class* $(a \otimes_o b)$›
by (*auto simp add*: *trace-class-tensor*)
show ‹*trace-class a* $\wedge$ *trace-class b* $\vee$ $a = 0$ $\vee$ $b = 0$› **if** ‹*trace-class* $(a \otimes_o b)$›
proof (*cases* ‹$a = 0 \vee b = 0$›)
case *True*
then show *?thesis*
by *simp*
next
case *False*
then have ‹$a \neq 0$› **and** ‹$b \neq 0$›
by *auto*
have $*$: ‹*trace-class a*› **if** ‹*trace-class* $(a \otimes_o b)$› **and** ‹$b \neq 0$› **for** $a :: ‹'e\ ell2 \Rightarrow_{CL} 'g\ ell2›$ **and** $b :: ‹'f\ ell2 \Rightarrow_{CL} 'h\ ell2›$
proof −
from ‹$b \neq 0$› **have** ‹*abs-op* $b \neq 0$›
using *abs-op-nondegenerate* **by** *blast*
then obtain $\psi 0$ **where** $\psi 0$: ‹$\psi 0 \cdot_C$ (*abs-op* $b *_V \psi 0$) $\neq 0$›
by (*metis cblinfun.zero-left cblinfun-cinner-eqI cinner-zero-right*)
define ψ **where** ‹$\psi = sgn\ \psi 0$›
with $\psi 0$ **have** ‹$\psi \cdot_C$ (*abs-op* $b *_V \psi$) $\neq 0$› **and** ‹*norm* $\psi = 1$›
by (*auto simp add*: ψ-*def norm-sgn sgn-div-norm cblinfun.scaleR-right field-simps*)
then have ‹$\exists B.\ \{\psi\} \subseteq B \wedge$ *is-onb* B›
by (*intro orthonormal-basis-exists*) *auto*
then obtain B **where** [*simp*]: ‹*is-onb* B› **and** ‹$\psi \in B$›
by *auto*
define $A :: ‹'e\ ell2\ set›$ **where** ‹A = *range ket*›
then have [*simp*]: ‹*is-onb* A› **by** *simp*
with ‹*is-onb* B› **have** ‹*is-onb* $\{\alpha \otimes_s \beta\ |\alpha\ \beta.\ \alpha \in A \wedge \beta \in B\}$›
by (*simp add*: *tensor-ell2-is-onb*)
with ‹*trace-class* $(a \otimes_o b)$›
have ‹$(\lambda\gamma.\ \gamma \cdot_C$ (*abs-op* $(a \otimes_o b) *_V \gamma)$) *abs-summable-on* $\{\alpha \otimes_s \beta\ |\alpha\ \beta.\ \alpha \in A \wedge \beta \in B\}$›
using *trace-class-iff-summable* **by** *auto*
then have ‹$(\lambda\gamma.\ \gamma \cdot_C$ (*abs-op* $(a \otimes_o b) *_V \gamma)$) *abs-summable-on* $(\lambda\alpha.\ \alpha \otimes_s \psi)$ ‘ A›
by (*rule summable-on-subset-banach*) (*use* ‹$\psi \in B$› **in** *blast*)
then have ‹$(\lambda\alpha.\ (\alpha \otimes_s \psi) \cdot_C$ (*abs-op* $(a \otimes_o b) *_V (\alpha \otimes_s \psi)$)) *abs-summable-on* A›
proof (*subst* (*asm*) *summable-on-reindex*)
show *inj-on* $(\lambda\alpha.\ \alpha \otimes_s \psi)$ A
by (*metis UNIV-I* ‹*norm* $\psi = 1$› *inj-on-subset inj-tensor-ell2-left norm-le-zero-iff not-one-le-zero subset-iff*)
qed (*simp-all add*: *o-def*)
then have ‹$(\lambda\alpha.$ *norm* $(\alpha \cdot_C$ (*abs-op* $a *_V \alpha$)) $*$ *norm* $(\psi \cdot_C$ (*abs-op* $b *_V \psi$))) *summable-on*

```
A›
      by (simp add: abs-op-tensor tensor-op-ell2 norm-mult)
    then have ‹(λα. α •C (abs-op a *V α)) abs-summable-on A›
      by (rule summable-on-cmult-left'[THEN iffD1, rotated])
        (use ‹ψ •C (abs-op b *V ψ) ≠ 0› norm-eq-zero in ‹blast›)
    then show ‹trace-class a›
      using ‹is-onb A› trace-classI by blast
  qed
  from *[of a b] ‹b ≠ 0› ‹trace-class (a ⊗o b)› have ‹trace-class a›
    by simp
  have ‹trace-class (b ⊗o a)›
    using that trace-class-tensor-op-swap by blast
  from *[of b a] ‹a ≠ 0› ‹trace-class (b ⊗o a)› have ‹trace-class b›
    by simp
  from ‹trace-class a› ‹trace-class b› show ?thesis
    by simp
 qed
qed


lemma trace-tensor: ‹trace (a ⊗o b) = trace a * trace b›
 — [8, Lemma 32]
proof −
 consider (tc) ‹trace-class a› ‹trace-class b› | (zero) ‹a = 0 ∨ b = 0› | (nota) ‹a ≠ 0› ‹b ≠
0› ‹¬ trace-class a› | (notb) ‹a ≠ 0› ‹b ≠ 0› ‹¬ trace-class b›
   by blast
 then show ?thesis
 proof cases
   case tc
   then have *: ‹trace-class (a ⊗o b)›
     by (simp add: trace-class-tensor)
   have sum: ‹(λ(x, y). ket (x, y) •C ((a ⊗o b) *V ket (x, y))) summable-on UNIV›
     using trace-exists[OF is-onb-ket *]
     by (simp-all add: o-def case-prod-unfold summable-on-reindex)

   have ‹trace a * trace b = (∑∞x. ∑∞y. ket x •C (a *V ket x) * (ket y •C (b *V ket y)))›
     apply (simp add: trace-ket-sum tc flip: infsum-cmult-left')
     by (simp flip: infsum-cmult-right')
   also have ‹... = (∑∞x. ∑∞y. ket (x,y) •C ((a ⊗o b) *V ket (x,y)))›
     by (simp add: tensor-op-ell2 flip: tensor-ell2-ket)
   also have ‹... = (∑∞xy∈UNIV. ket xy •C ((a ⊗o b) *V ket xy))›
     apply (simp add: sum infsum-Sigma'-banach)
     by (simp add: case-prod-unfold)
   also have ‹... = trace (a ⊗o b)›
     by (simp add: * trace-ket-sum)
   finally show ?thesis
     by simp
 next
   case zero
```

```
    then show ?thesis by auto
  next
    case nota
    then have [simp]: ‹trace a = 0›
      unfolding trace-def by simp
    from nota have ‹¬ trace-class (a ⊗o b)›
      by (simp add: trace-class-tensor-iff)
    then have [simp]: ‹trace (a ⊗o b) = 0›
      unfolding trace-def by simp
    show ?thesis
      by simp
  next
    case notb
    then have [simp]: ‹trace b = 0›
      unfolding trace-def by simp
    from notb have ‹¬ trace-class (a ⊗o b)›
      by (simp add: trace-class-tensor-iff)
    then have [simp]: ‹trace (a ⊗o b) = 0›
      unfolding trace-def by simp
    show ?thesis
      by simp
  qed
qed

lemma isometry-tensor-op: ‹isometry (U ⊗o V)› if ‹isometry U› and ‹isometry V›
  unfolding isometry-def using that by (simp add: tensor-op-adjoint comp-tensor-op)

lemma is-Proj-tensor-op: ‹is-Proj a ⟹ is-Proj b ⟹ is-Proj (a ⊗o b)›
  by (simp add: comp-tensor-op is-Proj-algebraic tensor-op-adjoint)

lemma isometry-tensor-id-right[simp]:
  fixes U :: ‹'a ell2 ⇒CL 'b ell2›
  shows ‹isometry (U ⊗o (id-cblinfun :: 'c ell2 ⇒CL -)) ⟷ isometry U›
proof (rule iffI)
  assume ‹isometry U›
  then show ‹isometry (U ⊗o id-cblinfun)›
    unfolding isometry-def
    by (auto simp add: tensor-op-adjoint comp-tensor-op)
next
  let ?id = ‹id-cblinfun :: 'c ell2 ⇒CL -›
  assume asm: ‹isometry (U ⊗o ?id)›
  then have ‹(U* oCL U) ⊗o ?id = id-cblinfun ⊗o ?id›
    by (simp add: isometry-def tensor-op-adjoint comp-tensor-op)
  then have ‹U* oCL U = id-cblinfun›
    by (rule inj-tensor-left[of ?id, unfolded inj-def, rule-format, rotated]) simp
  then show ‹isometry U›
    by (simp add: isometry-def)
qed
```

lemma *isometry-tensor-id-left*[*simp*]:
fixes $U :: \langle 'a\ ell2 \Rightarrow_{CL} 'b\ ell2 \rangle$
shows $\langle isometry\ ((id\text{-}cblinfun :: 'c\ ell2 \Rightarrow_{CL} -) \otimes_o U) \longleftrightarrow isometry\ U \rangle$
proof (*rule iffI*)
assume $\langle isometry\ U \rangle$
then show $\langle isometry\ (id\text{-}cblinfun \otimes_o U) \rangle$
unfolding *isometry-def*
by (*auto simp add*: *tensor-op-adjoint comp-tensor-op*)
next
let *?id* = $\langle id\text{-}cblinfun :: 'c\ ell2 \Rightarrow_{CL} - \rangle$
assume *asm*: $\langle isometry\ (?id \otimes_o U) \rangle$
then have $\langle ?id \otimes_o (U* \circ_{CL} U) = ?id \otimes_o id\text{-}cblinfun \rangle$
by (*simp add*: *isometry-def tensor-op-adjoint comp-tensor-op*)
then have $\langle U* \circ_{CL} U = id\text{-}cblinfun \rangle$
by (*rule inj-tensor-right*[*of ?id*, *unfolded inj-def*, *rule-format*, *rotated*]) *simp*
then show $\langle isometry\ U \rangle$
by (*simp add*: *isometry-def*)
qed

lemma *unitary-tensor-id-right*[*simp*]: $\langle unitary\ (U \otimes_o id\text{-}cblinfun) \longleftrightarrow unitary\ U \rangle$
unfolding *unitary-twosided-isometry*
by (*simp add*: *tensor-op-adjoint*)

lemma *unitary-tensor-id-left*[*simp*]: $\langle unitary\ (id\text{-}cblinfun \otimes_o U) \longleftrightarrow unitary\ U \rangle$
unfolding *unitary-twosided-isometry*
by (*simp add*: *tensor-op-adjoint*)

lemma *sandwich-tensor-op*: $\langle sandwich\ (a \otimes_o b)\ (c \otimes_o d) = sandwich\ a\ c \otimes_o sandwich\ b\ d \rangle$
by (*simp add*: *sandwich-apply tensor-op-adjoint flip*: *cblinfun-compose-assoc comp-tensor-op*)

lemma *sandwich-assoc-ell2-tensor-op*[*simp*]: $\langle sandwich\ assoc\text{-}ell2\ ((a \otimes_o b) \otimes_o c) = a \otimes_o (b \otimes_o c) \rangle$
by (*auto intro*!: *tensor-ell2-extensionality3*
simp: *sandwich-apply assoc-ell2'-tensor assoc-ell2-tensor tensor-op-ell2*)

lemma *unitary-tensor-op*: $\langle unitary\ (a \otimes_o b) \rangle$ **if** [*simp*]: $\langle unitary\ a \rangle$ $\langle unitary\ b \rangle$
by (*auto intro*!: *unitaryI simp add*: *tensor-op-adjoint comp-tensor-op*)

lemma *tensor-ell2-right-butterfly*: $\langle tensor\text{-}ell2\text{-}right\ \psi \circ_{CL} tensor\text{-}ell2\text{-}right\ \varphi* = id\text{-}cblinfun \otimes_o butterfly\ \psi\ \varphi \rangle$
by (*auto intro*!: *equal-ket cinner-ket-eqI simp*: *tensor-op-ell2 simp flip*: *tensor-ell2-ket*)
lemma *tensor-ell2-left-butterfly*: $\langle tensor\text{-}ell2\text{-}left\ \psi \circ_{CL} tensor\text{-}ell2\text{-}left\ \varphi* = butterfly\ \psi\ \varphi \otimes_o id\text{-}cblinfun \rangle$
by (*auto intro*!: *equal-ket cinner-ket-eqI simp*: *tensor-op-ell2 simp flip*: *tensor-ell2-ket*)

lift-definition *tc-tensor* :: $\langle ('a\ ell2, 'b\ ell2)\ trace\text{-}class \Rightarrow ('c\ ell2, 'd\ ell2)\ trace\text{-}class \Rightarrow ((('a \times 'c)\ ell2, ('b \times 'd)\ ell2)\ trace\text{-}class \rangle$ **is**
tensor-op

by (*auto intro*!: *trace-class-tensor*)

lemma *trace-norm-tensor*: ‹*trace-norm* (*a* $\otimes_o$ *b*) = *trace-norm a* ∗ *trace-norm b*›
by (*rule of-real-hom.injectivity*[**where** *'a*=*complex*])
(*simp add*: *abs-op-tensor trace-tensor flip*: *trace-abs-op*)

lemma *bounded-cbilinear-tc-tensor*: ‹*bounded-cbilinear tc-tensor*›
unfolding *bounded-cbilinear-def*
by *transfer*
(*auto intro*!: *exI*[*of - 1*]
simp: *trace-norm-tensor tensor-op-left-add tensor-op-right-add tensor-op-scaleC-left tensor-op-scaleC-right*)
lemmas *bounded-clinear-tc-tensor-left*[*bounded-clinear*] = *bounded-cbilinear.bounded-clinear-left*[*OF bounded-cbilinear-tc-tensor*]
lemmas *bounded-clinear-tc-tensor-right*[*bounded-clinear*] = *bounded-cbilinear.bounded-clinear-right*[*OF bounded-cbilinear-tc-tensor*]

lemma *tc-tensor-scaleC-left*: ‹*tc-tensor* (*c* $*_C$ *a*) *b* = *c* $*_C$ *tc-tensor a b*›
by *transfer'* (*simp add*: *tensor-op-scaleC-left*)
lemma *tc-tensor-scaleC-right*: ‹*tc-tensor a* (*c* $*_C$ *b*) = *c* $*_C$ *tc-tensor a b*›
by *transfer'* (*simp add*: *tensor-op-scaleC-right*)

lemma *comp-tc-tensor*: ‹*tc-compose* (*tc-tensor a b*) (*tc-tensor c d*) = *tc-tensor* (*tc-compose a c*) (*tc-compose b d*)›
by *transfer'* (*rule comp-tensor-op*)

lemma *norm-tc-tensor*: ‹*norm* (*tc-tensor a b*) = *norm a* ∗ *norm b*›
by (*transfer'*, *rule of-real-hom.injectivity*[**where** *'a*=*complex*])
(*simp add*: *abs-op-tensor trace-tensor flip*: *trace-abs-op*)

lemma *tc-tensor-pos*: ‹*tc-tensor a b* ≥ *0*› **if** ‹*a* ≥ *0*› **and** ‹*b* ≥ *0*›
for *a* :: ‹(*'a ell2*,*'a ell2*) *trace-class*› **and** *b* :: ‹(*'b ell2*,*'b ell2*) *trace-class*›
using *that* **by** *transfer'* (*rule tensor-op-pos*)

interpretation *tensor-op-cbilinear*: *bounded-cbilinear tensor-op*
by *simp*

lemma *tensor-op-mono-left*:
fixes *a* :: ‹*'a ell2* $\Rightarrow_{CL}$ *'a ell2*› **and** *c* :: ‹*'b ell2* $\Rightarrow_{CL}$ *'b ell2*›
assumes ‹*a* ≤ *b*› **and** ‹*c* ≥ *0*›
shows ‹*a* $\otimes_o$ *c* ≤ *b* $\otimes_o$ *c*›
proof −
have ‹*b* − *a* ≥ *0*›
by (*simp add*: *assms*(*1*))
with ‹*c* ≥ *0*› **have** ‹(*b* − *a*) $\otimes_o$ *c* ≥ *0*›
by (*intro tensor-op-pos*)
then have ‹*b* $\otimes_o$ *c* − *a* $\otimes_o$ *c* ≥ *0*›

by (*simp add*: *tensor-op-cbilinear.diff-left*)
then show *?thesis*
by *simp*
qed

lemma *tensor-op-mono-right*:
fixes $a :: \langle 'a\ ell2 \Rightarrow_{CL} 'a\ ell2 \rangle$ **and** $b :: \langle 'b\ ell2 \Rightarrow_{CL} 'b\ ell2 \rangle$
assumes $\langle b \leq c \rangle$ **and** $\langle a \geq 0 \rangle$
shows $\langle a \otimes_o b \leq a \otimes_o c \rangle$
proof −
have $\langle c - b \geq 0 \rangle$
by (*simp add*: *assms*(*1*))
with $\langle a \geq 0 \rangle$ **have** $\langle a \otimes_o (c - b) \geq 0 \rangle$
by (*intro tensor-op-pos*)
then have $\langle a \otimes_o c - a \otimes_o b \geq 0 \rangle$
by (*simp add*: *tensor-op-cbilinear.diff-right*)
then show *?thesis*
by *simp*
qed

lemma *tensor-op-mono*:
fixes $a :: \langle 'a\ ell2 \Rightarrow_{CL} 'a\ ell2 \rangle$ **and** $c :: \langle 'b\ ell2 \Rightarrow_{CL} 'b\ ell2 \rangle$
assumes $\langle a \leq b \rangle$ **and** $\langle c \leq d \rangle$ **and** $\langle b \geq 0 \rangle$ **and** $\langle c \geq 0 \rangle$
shows $\langle a \otimes_o c \leq b \otimes_o d \rangle$
proof −
have $\langle a \otimes_o c \leq b \otimes_o c \rangle$
using $\langle a \leq b \rangle$ **and** $\langle c \geq 0 \rangle$
by (*rule tensor-op-mono-left*)
also have $\langle \ldots \leq b \otimes_o d \rangle$
using $\langle c \leq d \rangle$ **and** $\langle b \geq 0 \rangle$
by (*rule tensor-op-mono-right*)
finally show *?thesis*
by −
qed

lemma *sandwich-tc-tensor*: $\langle$*sandwich-tc* (*E* $\otimes_o$ *F*) (*tc-tensor t u*) = *tc-tensor* (*sandwich-tc E t*) (*sandwich-tc F u*)$\rangle$
by (*transfer fixing*: *E F*) (*simp add*: *sandwich-tensor-op*)

lemma *tensor-tc-butterfly*: *tc-tensor* (*tc-butterfly* ψ ψ') (*tc-butterfly* φ φ') = *tc-butterfly* (*tensor-ell2* ψ φ) (*tensor-ell2* ψ' φ')
by (*transfer fixing*: φ φ' ψ ψ') (*simp add*: *tensor-butterfly*)

lemma *separating-set-bounded-clinear-tc-tensor*:

shows ‹*separating-set bounded-clinear* ((λ(ϱ,σ). *tc-tensor* ϱ σ) ‘ (*UNIV* × *UNIV*))›
proof −
have ‹⊤ = *ccspan* ((λ(*x*, *y*). *tc-butterfly x y*) ‘ (*range ket* × *range ket*))›
by (*simp add*: *onb-butterflies-span-trace-class*)
also have ‹... = *ccspan* ((λ(*x*, *y*, *z*, *w*). *tc-butterfly* (*x* ⊗*s* *y*) (*z* ⊗*s* *w*)) ‘ (*range ket* × *range ket* × *range ket* × *range ket*))›
by (*auto intro*!: *arg-cong*[**where** *f*=*ccspan*] *image-eqI simp*: *tensor-ell2-ket*)
also have ‹... = *ccspan* ((λ(*x*, *y*, *z*, *w*). *tc-tensor* (*tc-butterfly x z*) (*tc-butterfly y w*)) ‘ (*range ket* × *range ket* × *range ket* × *range ket*))›
by (*simp add*: *tensor-tc-butterfly*)
also have ‹... ≤ *ccspan* ((λ(ϱ, σ). *tc-tensor* ϱ σ) ‘ (*UNIV* × *UNIV*))›
by (*auto intro*!: *ccspan-mono*)
finally show *?thesis*
by (*intro separating-set-bounded-clinear-dense*) (*use top-le* **in** *blast*)
qed

lemma *separating-set-bounded-clinear-tc-tensor-nested*:
assumes ‹*separating-set* (*bounded-clinear* :: (- => ′*e*::*complex-normed-vector*) ⇒ -) *A*›
assumes ‹*separating-set* (*bounded-clinear* :: (- => ′*e*::*complex-normed-vector*) ⇒ -) *B*›
shows ‹*separating-set* (*bounded-clinear* :: (- => ′*e*::*complex-normed-vector*) ⇒ -) ((λ(ϱ,σ). *tc-tensor* ϱ σ) ‘ (*A* × *B*))›
using *separating-set-bounded-clinear-tc-tensor bounded-cbilinear-tc-tensor assms*
by (*rule separating-set-bounded-cbilinear-nested*)

lemma *tc-tensor-0-left*[*simp*]: ‹*tc-tensor 0 x* = *0*›
by *transfer*′ *simp*
lemma *tc-tensor-0-right*[*simp*]: ‹*tc-tensor x 0* = *0*›
by *transfer*′ *simp*

14.3 Tensor product of subspaces

definition *tensor-ccsubspace* (**infixr** ⊗*S* *70*) **where**
‹*tensor-ccsubspace A B* = *ccspan* {ψ ⊗*s* φ | ψ φ. ψ ∈ *space-as-set A* ∧ φ ∈ *space-as-set B*}›

lemma *tensor-ccsubspace-via-Proj*: ‹*A* ⊗*S* *B* = (*Proj A* ⊗*o* *Proj B*) ∗*S* ⊤›
proof (*rule antisym*)
have ‹ψ ⊗*s* φ ∈ *space-as-set* ((*Proj A* ⊗*o* *Proj B*) ∗*S* ⊤)› **if** ‹ψ ∈ *space-as-set A*› **and** ‹φ ∈ *space-as-set B*› **for** ψ φ
by (*metis Proj-fixes-image cblinfun-apply-in-image tensor-op-ell2 that*(*1*) *that*(*2*))
then show ‹*A* ⊗*S* *B* ≤ (*Proj A* ⊗*o* *Proj B*) ∗*S* ⊤›
by (*auto intro*!: *ccspan-leqI simp*: *tensor-ccsubspace-def*)
have ∗: ‹*ccspan* {ψ ⊗*s* φ | (ψ::′*a ell2*) (φ::′*b ell2*). *True*} = ⊤›
using *tensor-ell2-dense*′[**where** *A*=‹*UNIV* :: ′*a ell2 set*› **and** *B*=‹*UNIV* :: ′*b ell2 set*›]
by *auto*
have ‹(*Proj A* ⊗*o* *Proj B*) ∗*V* ψ ⊗*s* φ ∈ *space-as-set* (*A* ⊗*S* *B*)› **for** ψ φ
unfolding *tensor-op-ell2 tensor-ccsubspace-def*
by (*smt* (*verit*) *Proj-range cblinfun-apply-in-image ccspan-superset mem-Collect-eq subsetD*)
then show ‹(*Proj A* ⊗*o* *Proj B*) ∗*S* ⊤ ≤ *A* ⊗*S* *B*›

by (*auto intro*!: *ccspan-leqI simp*: *cblinfun-image-ccspan simp flip*: ∗)
qed

lemma *tensor-ccsubspace-top*[*simp*]: ‹⊤ ⊗$_S$ ⊤ = ⊤›
by (*simp add*: *tensor-ccsubspace-via-Proj*)

lemma *tensor-ccsubspace-0-left*[*simp*]: ‹*0* ⊗$_S$ *X* = *0*›
by (*simp add*: *tensor-ccsubspace-via-Proj*)

lemma *tensor-ccsubspace-0-right*[*simp*]: ‹*X* ⊗$_S$ *0* = *0*›
by (*simp add*: *tensor-ccsubspace-via-Proj*)

lemma *tensor-ccsubspace-image*: ‹(*A* ∗$_S$ *T*) ⊗$_S$ (*B* ∗$_S$ *U*) = (*A* ⊗$_o$ *B*) ∗$_S$ (*T* ⊗$_S$ *U*)›
proof −
have ‹(*A* ∗$_S$ *T*) ⊗$_S$ (*B* ∗$_S$ *U*) = *ccspan* ((λ(ψ, φ). ψ ⊗$_s$ φ) ‘ (*space-as-set* (*A* ∗$_S$ *T*) × *space-as-set* (*B* ∗$_S$ *U*)))›
by (*simp add*: *tensor-ccsubspace-def set-compr-2-image-collect ccspan.rep-eq*)
also have ‹... = *ccspan* ((λ(ψ, φ). ψ ⊗$_s$ φ) ‘ *closure* ((*A* ‘ *space-as-set T*) × (*B* ‘ *space-as-set U*)))›
by (*simp add*: *cblinfun-image.rep-eq closure-Times*)
also have ‹... = *ccspan* (*closure* ((λ(ψ, φ). ψ ⊗$_s$ φ) ‘ ((*A* ‘ *space-as-set T*) × (*B* ‘ *space-as-set U*))))›
by (*subst closure-image-closure*[*symmetric*])
(*use continuous-on-subset continuous-tensor-ell2* **in** *auto*)
also have ‹... = *ccspan* ((λ(ψ, φ). ψ ⊗$_s$ φ) ‘ ((*A* ‘ *space-as-set T*) × (*B* ‘ *space-as-set U*)))›
by *simp*
also have ‹... = (*A* ⊗$_o$ *B*) ∗$_S$ *ccspan* ((λ(ψ, φ). ψ ⊗$_s$ φ) ‘ (*space-as-set T* × *space-as-set U*))›
by (*simp add*: *cblinfun-image-ccspan image-image tensor-op-ell2 case-prod-beta*
flip: *map-prod-image*)
also have ‹... = (*A* ⊗$_o$ *B*) ∗$_S$ (*T* ⊗$_S$ *U*)›
by (*simp add*: *tensor-ccsubspace-def set-compr-2-image-collect*)
finally show *?thesis*
by −
qed

lemma *tensor-ccsubspace-bot-left*[*simp*]: ‹⊥ ⊗$_S$ *S* = ⊥›
by (*simp add*: *tensor-ccsubspace-via-Proj*)
lemma *tensor-ccsubspace-bot-right*[*simp*]: ‹*S* ⊗$_S$ ⊥ = ⊥›
by (*simp add*: *tensor-ccsubspace-via-Proj*)

lemma *swap-ell2-tensor-ccsubspace*: ‹*swap-ell2* ∗$_S$ (*S* ⊗$_S$ *T*) = *T* ⊗$_S$ *S*›
by (*force intro*!: *arg-cong*[**where** *f*=*ccspan*]
simp add: *tensor-ccsubspace-def cblinfun-image-ccspan image-image set-compr-2-image-collect*)

lemma *tensor-ccsubspace-right1dim-member*:
assumes ‹ψ ∈ *space-as-set* (*S* ⊗$_S$ *ccspan*{φ})›
shows ‹∃ψ′. ψ = ψ′ ⊗$_s$ φ›

proof (*cases* ‹$\varphi = 0$›)
case *True*
with *assms* **show** *?thesis*
by *simp*
next
case *False*
have ‹$\{\psi \otimes_s \varphi' \,|\psi\ \varphi'.\ \psi \in$ *space-as-set* $S \wedge \varphi' \in$ *space-as-set* (*ccspan* $\{\varphi\})\}$
$= \{\psi \otimes_s \varphi \mid \psi.\ \psi \in$ *space-as-set* $S\}$›
proof −
have ‹$\psi \in$ *space-as-set* $S \Longrightarrow \exists \psi'.\ \psi \otimes_s c *_C \varphi = \psi' \otimes_s \varphi \wedge \psi' \in$ *space-as-set* S› **for** $c\ \psi$
by (*rule exI*[**where** x=‹$c *_C \psi$›])
(*auto simp*: *tensor-ell2-scaleC2 tensor-ell2-scaleC1*
complex-vector.subspace-scale)
moreover have ‹$\psi \in$ *space-as-set* $S \Longrightarrow$
$\exists \psi'\ \varphi'.\ \psi \otimes_s \varphi = \psi' \otimes_s \varphi' \wedge \psi' \in$ *space-as-set* $S \wedge \varphi' \in$ *range* $(\lambda k.\ k *_C \varphi)$› **for** ψ
by (*rule exI*[**where** x=ψ], *rule exI*[**where** x=φ])
(*auto intro*!: *range-eqI*[**where** x=‹*1*::*complex*›])
ultimately show *?thesis*
by (*auto simp*: *ccspan-finite complex-vector.span-singleton*)
qed
moreover have ‹*csubspace* $\{\psi \otimes_s \varphi \,|\psi.\ \psi \in$ *space-as-set* $S\}$›
proof (*rule complex-vector.subspaceI*)
show ‹*0* $\in \{\psi \otimes_s \varphi \,|\psi.\ \psi \in$ *space-as-set* $S\}$›
by (*auto intro*!: *exI*[**where** x=*0*])
show ‹$x + y \in \{\psi \otimes_s \varphi \,|\psi.\ \psi \in$ *space-as-set* $S\}$›
if x: ‹$x \in \{\psi \otimes_s \varphi \,|\psi.\ \psi \in$ *space-as-set* $S\}$›
and y: ‹$y \in \{\psi \otimes_s \varphi \,|\psi.\ \psi \in$ *space-as-set* $S\}$› **for** $x\ y$
using *that complex-vector.subspace-add tensor-ell2-add1*
unfolding *mem-Collect-eq* **by** (*metis csubspace-space-as-set*)
show ‹$c *_C x \in \{\psi \otimes_s \varphi \,|\psi.\ \psi \in$ *space-as-set* $S\}$›
if x: ‹$x \in \{\psi \otimes_s \varphi \,|\psi.\ \psi \in$ *space-as-set* $S\}$› **for** $c\ x$
using *that complex-vector.subspace-scale tensor-ell2-scaleC2 tensor-ell2-scaleC1*
unfolding *mem-Collect-eq* **by** (*metis csubspace-space-as-set*)
qed
moreover have ‹*closed* $\{\psi \otimes_s \varphi \,|\psi.\ \psi \in$ *space-as-set* $S\}$›
proof (*rule closed-sequential-limits*[*THEN iffD2*, *rule-format*])
fix $x\ l$
assume *asm*: ‹$(\forall n.\ x\ n \in \{\psi \otimes_s \varphi \,|\psi.\ \psi \in$ *space-as-set* $S\}) \wedge x \longrightarrow l$›
then obtain ψ' **where** *x-def*: ‹$x\ n = \psi'\ n \otimes_s \varphi$› **and** ψ'-*S*: ‹$\psi'\ n \in$ *space-as-set* S› **for** n
unfolding *mem-Collect-eq* **by** *metis*
from *asm* **have** ‹$x \longrightarrow l$›
by *simp*
have ‹*Cauchy* ψ'›
proof (*rule CauchyI*)
fix e :: *real* **assume** ‹$e > 0$›
define d **where** ‹$d = e * norm\ \varphi$›
with *False* ‹$e > 0$› **have** ‹$d > 0$›
by *auto*
from ‹$x \longrightarrow l$›

```
    have ‹Cauchy x›
      using LIMSEQ-imp-Cauchy by blast
    then obtain M where ‹∀ m≥M. ∀ n≥M. norm (x m − x n) < d›
      using Cauchy-iff ‹0 < d› by blast
    then show ‹∃ M. ∀ m≥M. ∀ n≥M. norm (ψ' m − ψ' n) < e›
      by (intro exI[of - M])
        (use False in ‹auto simp add: x-def norm-tensor-ell2 d-def simp flip: tensor-ell2-diff1›)
  qed
  then obtain l' where ‹ψ' ⟶ l'›
    using convergent-eq-Cauchy by blast
  with ψ'-S have l'-S: ‹l' ∈ space-as-set S›
    by (metis ‹Cauchy ψ'› completeE complete-space-as-set limI)
  from ‹ψ' ⟶ l'› have ‹x ⟶ l' ⊗s φ›
    by (auto intro: tendsto-eq-intros simp: x-def[abs-def])
  with ‹x ⟶ l› have ‹l = l' ⊗s φ›
    using LIMSEQ-unique by blast
  then show ‹l ∈ {ψ ⊗s φ |ψ. ψ ∈ space-as-set S}›
    using l'-S by auto
qed
ultimately have ‹space-as-set (ccspan {ψ ⊗s φ' |ψ φ'. ψ ∈ space-as-set S ∧ φ' ∈ space-as-set
(ccspan {φ})})
    = {ψ ⊗s φ |ψ. ψ ∈ space-as-set S}›
  by (simp add: ccspan.rep-eq complex-vector.span-eq-iff[THEN iffD2])
with assms have ‹ψ ∈ {ψ ⊗s φ |ψ. ψ ∈ space-as-set S}›
  by (simp add: tensor-ccsubspace-def)
then show ‹∃ψ'. ψ = ψ' ⊗s φ›
  by auto
qed

lemma tensor-ccsubspace-left1dim-member:
assumes ‹ψ ∈ space-as-set (ccspan{φ} ⊗S S)›
shows ‹∃ψ'. ψ = φ ⊗s ψ'›
proof −
from assms
have ‹swap-ell2 *V ψ ∈ space-as-set (swap-ell2 *S (ccspan {φ} ⊗S S))›
by (metis rev-image-eqI space-as-set-image-commute swap-ell2-selfinv)
then have ‹swap-ell2 ψ ∈ space-as-set (S ⊗S ccspan{φ})›
  by (simp add: swap-ell2-tensor-ccsubspace)
then obtain ψ' where ψ': ‹swap-ell2 ψ = ψ' ⊗s φ›
  using tensor-ccsubspace-right1dim-member by blast
have ‹ψ = swap-ell2 *V swap-ell2 *V ψ›
  by (simp flip: cblinfun-apply-cblinfun-compose)
also have ‹... = swap-ell2 *V (ψ' ⊗s φ)›
  by (simp add: ψ')
also have ‹... = φ ⊗s ψ'›
  by simp
finally show ?thesis
  by auto
qed
```

lemma *tensor-ell2-mem-tensor-ccsubspace-left*:
assumes ‹$a \otimes_s b \in$ *space-as-set* $(S \otimes_S T)$› **and** ‹$b \neq 0$›
shows ‹$a \in$ *space-as-set* S›
proof (*cases* ‹$a = 0$›)
case *True*
then show *?thesis*
by *simp*
next
case *False*
have ‹*norm* (*Proj* S a) $*$ *norm* (*Proj* T b) = *norm* ((*Proj* S a) $\otimes_s$ (*Proj* T b))›
by (*simp add*: *norm-tensor-ell2*)
also have ‹$\ldots$ = *norm* (*Proj* $(S \otimes_S T)$ $(a \otimes_s b)$)›
by (*simp add*: *tensor-ccsubspace-via-Proj Proj-on-own-range is-Proj-tensor-op tensor-op-ell2*)
also from *assms* **have** ‹$\ldots$ = *norm* $(a \otimes_s b)$›
by (*simp add*: *Proj-fixes-image*)
also have ‹$\ldots$ = *norm* a $*$ *norm* b›
by (*simp add*: *norm-tensor-ell2*)
finally have *prod-eq*: ‹*norm* (*Proj* S $*_V$ a) $*$ *norm* (*Proj* T $*_V$ b) = *norm* a $*$ *norm* b›
by −
with *False* ‹$b \neq 0$› **have** *Tb-non0*: ‹*norm* (*Proj* T $*_V$ b) $\neq 0$›
by *fastforce*
have ‹*norm* (*Proj* S a) = *norm* a›
proof (*rule ccontr*)
assume *asm*: ‹*norm* (*Proj* S $*_V$ a) $\neq$ *norm* a›
have *Sa-leq*: ‹*norm* (*Proj* S $*_V$ a) $\leq$ *norm* a›
by (*simp add*: *is-Proj-reduces-norm*)
have *Tb-leq*: ‹*norm* (*Proj* T $*_V$ b) $\leq$ *norm* b›
by (*simp add*: *is-Proj-reduces-norm*)
from *asm Sa-leq* **have** ‹*norm* (*Proj* S $*_V$ a) $<$ *norm* a›
by *simp*
then have ‹*norm* (*Proj* S $*_V$ a) $*$ *norm* (*Proj* T $*_V$ b) $<$ *norm* a $*$ *norm* (*Proj* T $*_V$ b)›
using *Tb-non0* **by** *auto*
also from *Tb-leq* **have** ‹$\ldots \leq$ *norm* a $*$ *norm* b›
using *False* **by** *force*
also note *prod-eq*
finally show *False*
by *simp*
qed
then show ‹$a \in$ *space-as-set* S›
using *norm-Proj-apply* **by** *blast*
qed

lemma *tensor-ell2-mem-tensor-ccsubspace-right*:
assumes ‹$a \otimes_s b \in$ *space-as-set* $(S \otimes_S T)$› **and** ‹$a \neq 0$›
shows ‹$b \in$ *space-as-set* T›
proof −
have ‹*swap-ell2* $*_V$ $(a \otimes_s b) \in$ *space-as-set* (*swap-ell2* $*_S$ $(S \otimes_S T)$)›

```
    using assms(1) cblinfun-apply-in-image' by blast
  then have ‹b ⊗ₛ a ∈ space-as-set (T ⊗S S)›
    by (simp add: swap-ell2-tensor-ccsubspace)
  then show ‹b ∈ space-as-set T›
    using ‹a ≠ 0› by (rule tensor-ell2-mem-tensor-ccsubspace-left)
qed

lemma tensor-ell2-in-tensor-ccsubspace: ‹a ⊗ₛ b ∈ space-as-set (A ⊗S B)› if ‹a ∈ space-as-set
A› and ‹b ∈ space-as-set B›
  — Converse is tensor-ell2-mem-tensor-ccsubspace-left and ...-right.
  using that by (auto intro!: ccspan-superset[THEN subsetD] simp add: tensor-ccsubspace-def)

lemma tensor-ccsubspace-INF-left-top:
  fixes S :: ‹'a ⇒ 'b ell2 ccsubspace›
  shows ‹(INF x∈X. S x) ⊗S (⊤::'c ell2 ccsubspace) = (INF x∈X. S x ⊗S ⊤)›
proof (rule antisym[rotated])
  let ?top = ‹⊤ :: 'c ell2 ccsubspace›
  have ∗: ‹ψ ⊗ₛ φ ∈ space-as-set (⨅x∈X. S x ⊗S ?top)›
    if ‹ψ ∈ space-as-set (⨅x∈X. S x)› for ψ φ
  proof −
    from that(1) have ‹ψ ∈ space-as-set (S x)› if ‹x ∈ X› for x
      using that by (simp add: Inf-ccsubspace.rep-eq)
    then have ‹ψ ⊗ₛ φ ∈ space-as-set (S x ⊗S ⊤)› if ‹x ∈ X› for x
      using ccspan-superset that by (force simp: tensor-ccsubspace-def)
    then show ?thesis
      by (simp add: Inf-ccsubspace.rep-eq)
  qed
  show ‹(INF x∈X. S x) ⊗S ?top ≤ (INF x∈X. S x ⊗S ?top)›
    by (subst tensor-ccsubspace-def, rule ccspan-leqI) (use ∗ in auto)

  show ‹(⨅x∈X. S x ⊗S ?top) ≤ (⨅x∈X. S x) ⊗S ?top›
  proof (rule ccsubspace-leI-unit)
    fix ψ
    assume asm: ‹ψ ∈ space-as-set (⨅x∈X. S x ⊗S ?top)›
    obtain ψ' where ψ'b-b: ‹ψ' b ⊗ₛ ket b = (id-cblinfun ⊗o butterfly (ket b) (ket b)) ∗V ψ› for
b
    proof (atomize-elim, rule choice, intro allI)
      fix b :: 'c
      have ‹(id-cblinfun ⊗o butterfly (ket b) (ket b)) ∗V ψ ∈ space-as-set (⊤ ⊗S ccspan {ket b})›
        by (simp add: butterfly-eq-proj tensor-ccsubspace-via-Proj)
      then show ‹∃ψ'. ψ' ⊗ₛ ket b = (id-cblinfun ⊗o butterfly (ket b) (ket b)) ∗V ψ›
        by (metis tensor-ccsubspace-right1dim-member)
    qed

    have ‹ψ' b ∈ space-as-set (S x)› if ‹x ∈ X› for x b
    proof −
      from asm have ψ-ST: ‹ψ ∈ space-as-set (S x ⊗S ?top)›
        by (meson INF-lower Set.basic-monos(7) less-eq-ccsubspace.rep-eq that)
      have ‹ψ' b ⊗ₛ ket b = (id-cblinfun ⊗o butterfly (ket b) (ket b)) ∗V ψ›
```

```
      by (simp add: ψ'b-b)
    also from ψ-ST
    have ‹... ∈ space-as-set (((id-cblinfun ⊗o butterfly (ket b) (ket b))) *S (S x ⊗S ?top))›
      by (meson cblinfun-apply-in-image')
    also have ‹... = space-as-set (((id-cblinfun ⊗o butterfly (ket b) (ket b)) oCL (Proj (S x)
⊗o id-cblinfun)) *S ⊤)›
      by (simp add: cblinfun-compose-image tensor-ccsubspace-via-Proj)
    also have ‹... = space-as-set ((Proj (S x) ⊗o (butterfly (ket b) (ket b) oCL id-cblinfun))
*S ⊤)›
      by (simp add: comp-tensor-op)
    also have ‹... = space-as-set ((Proj (S x) ⊗o (id-cblinfun oCL butterfly (ket b) (ket b)))
*S ⊤)›
      by simp
    also have ‹... = space-as-set (((Proj (S x) ⊗o id-cblinfun) oCL (id-cblinfun ⊗o butterfly
(ket b) (ket b))) *S ⊤)›
      by (simp add: comp-tensor-op)
    also have ‹... ⊆ space-as-set ((Proj (S x) ⊗o id-cblinfun) *S ⊤)›
    by (metis cblinfun-compose-image cblinfun-image-mono less-eq-ccsubspace.rep-eq top-greatest)
    also have ‹... = space-as-set (S x ⊗S ?top)›
      by (simp add: tensor-ccsubspace-via-Proj)
    finally have ‹ψ' b ⊗s ket b ∈ space-as-set (S x ⊗S ?top)›
      by −
    then show ‹ψ' b ∈ space-as-set (S x)›
      using tensor-ell2-mem-tensor-ccsubspace-left
      by (metis ket-nonzero)
  qed

  then have ‹ψ' b ∈ space-as-set (⨅x∈X. S x)› if ‹x ∈ X› for x b
    using that by (simp add: Inf-ccsubspace.rep-eq)

  then have *: ‹ψ' b ⊗s ket b ∈ space-as-set ((⨅x∈X. S x) ⊗S ?top)› for b
    by (auto intro!: ccspan-superset[THEN set-mp]
        simp add: tensor-ccsubspace-def Inf-ccsubspace.rep-eq)

  have ‹ψ ∈ space-as-set (ccspan (range (λb. ψ' b ⊗s ket b)))› (is ‹ψ ∈ ?rhs›)
  proof −
    define γ where ‹γ F = (∑ b∈F. (id-cblinfun ⊗o butterfly (ket b) (ket b)) *V ψ)› for F
    have γ-rhs: ‹γ F ∈ ?rhs› for F
      using ccspan-superset  by (force intro!: complex-vector.subspace-sum simp add: γ-def
ψ'b-b)
    have γ-trunc: ‹γ F = trunc-ell2 (UNIV × F) ψ› if ‹finite F› for F
    proof (rule cinner-ket-eqI)
      fix x :: ‹'b × 'c› obtain x1 x2 where x-def: ‹x = (x1,x2)›
        by force
      have *: ‹ket x •C ((id-cblinfun ⊗o butterfly (ket j) (ket j)) *V ψ) = of-bool (j=x2) *
Rep-ell2 ψ x› for j
        apply (simp add: x-def tensor-op-ell2 tensor-op-adjoint cinner-ket
            flip: tensor-ell2-ket cinner-adj-left)
        by (simp add: tensor-ell2-ket cinner-ket-left)
```

```
    have ‹ket x •C γ F = of-bool (x2∈F) *C Rep-ell2 ψ x›
      using that
    apply (simp add: x-def γ-def complex-vector.linear-sum[of ‹cinner -›] bounded-clinear-cinner-right
          bounded-clinear.clinear sum-single[where i=x2] tensor-op-adjoint tensor-op-ell2
cinner-ket
        flip: tensor-ell2-ket cinner-adj-left)
      by (simp add: tensor-ell2-ket cinner-ket-left)
    moreover have ‹ket x •C trunc-ell2 (UNIV × F) ψ = of-bool (x2∈F) *C Rep-ell2 ψ x›
      by (simp add: trunc-ell2.rep-eq cinner-ket-left x-def)
    ultimately show ‹ket x •C γ F = ket x •C trunc-ell2 (UNIV × F) ψ›
      by simp
  qed
  have ‹(γ ⟶ ψ) (finite-subsets-at-top UNIV)›
  proof (rule tendsto-iff[THEN iffD2, rule-format])
    fix e :: real assume ‹e > 0›
    from trunc-ell2-lim-at-UNIV[of ψ]
    have ‹∀F F in finite-subsets-at-top UNIV. dist (trunc-ell2 F ψ) ψ < e›
      by (simp add: ‹0 < e› tendstoD)
    then obtain M where ‹finite M› and less-e: ‹finite F ⟹ F ⊇ M ⟹ dist (trunc-ell2
F ψ) ψ < e› for F
      by (metis (mono-tags, lifting) eventually-finite-subsets-at-top subset-UNIV)
    define M' where ‹M' = snd ‘ M›
    have ‹finite M'›
      using M'-def ‹finite M› by blast
    have ‹dist (γ F') ψ < e› if ‹finite F'› and ‹F' ⊇ M'› for F'
    proof −
      have ‹dist (γ F') ψ = norm (trunc-ell2 (− (UNIV × F')) ψ)›
       using that by (simp only: γ-trunc dist-norm trunc-ell2-uminus norm-minus-commute)
      also have ‹... ≤ norm (trunc-ell2 (− ((fst ‘ M) × F')) ψ)›
     by (meson Compl-anti-mono Set.basic-monos(1) Sigma-mono subset-UNIV trunc-ell2-norm-mono)
      also have ‹... = dist (trunc-ell2 ((fst ‘ M) × F') ψ) ψ›
        apply (simp add: trunc-ell2-uminus dist-norm)
        using norm-minus-commute by blast
      also have ‹... < e›
        apply (rule less-e)
        subgoal
          using ‹finite F'› ‹finite M› by force
        subgoal
          using ‹F' ⊇ M'› M'-def by force
        done
      finally show ?thesis
        by −
    qed
    then show ‹∀F F' in finite-subsets-at-top UNIV. dist (γ F') ψ < e›
      using ‹finite M'› by (auto simp add: eventually-finite-subsets-at-top)
  qed
  then show ‹ψ ∈ ?rhs›
    by (rule Lim-in-closed-set[rotated −1]) (use γ-rhs in auto)
```

```
    qed
    also from * have ‹... ⊆ space-as-set ((⨅x∈X. S x) ⊗S ?top)›
      by (meson ccspan-leqI image-subset-iff less-eq-ccsubspace.rep-eq)

    finally show ‹ψ ∈ space-as-set ((⨅x∈X. S x) ⊗S ?top)›
      by −
  qed
qed

lemma tensor-ccsubspace-INF-right-top:
  fixes S :: ‹'a ⇒ 'b ell2 ccsubspace›
  shows ‹(⊤::'c ell2 ccsubspace) ⊗S (INF x∈X. S x) = (INF x∈X. ⊤ ⊗S S x)›
proof −
  have ‹(INF x∈X. S x) ⊗S (⊤::'c ell2 ccsubspace) = (INF x∈X. S x ⊗S ⊤)›
    by (rule tensor-ccsubspace-INF-left-top)
  then have ‹swap-ell2 *S ((INF x∈X. S x) ⊗S (⊤::'c ell2 ccsubspace)) = swap-ell2 *S (INF
x∈X. S x ⊗S ⊤)›
    by simp
  then show ?thesis
    by (cases ‹X = {}›)
       (simp-all add: swap-ell2-tensor-ccsubspace)
qed

lemma tensor-ccsubspace-INF-left: ‹(INF x∈X. S x) ⊗S T = (INF x∈X. S x ⊗S T)› if ‹X ≠
{}›
proof (cases ‹T=0›)
  case True
  then show ?thesis
    using that by simp
next
  case False
  from ccsubspace-as-whole-type[OF False]
  have ‹let 't::type = some-onb-of T in
       (INF x∈X. S x) ⊗S T = (INF x∈X. S x ⊗S T)›
  proof with-type-mp
    with-type-case
    from with-type-mp.premise
    obtain U :: ‹'t ell2 ⇒CL 'c ell2› where [simp]: ‹isometry U› and imU: ‹U *S ⊤ = T›
      by auto
    have ‹(id-cblinfun ⊗o U) *S ((⨅x∈X. S x) ⊗S ⊤) = (id-cblinfun ⊗o U) *S (⨅x∈X. S x
⊗S ⊤)›
      by (rule arg-cong[where f=‹λx. - *S x›], rule tensor-ccsubspace-INF-left-top)
    then show ‹(⨅x∈X. S x) ⊗S T = (⨅x∈X. S x ⊗S T)›
      using that by (simp add: imU flip: tensor-ccsubspace-image)
  qed
  from this[cancel-with-type]
  show ?thesis
    by −
qed
```

lemma *tensor-ccsubspace-INF-right*: ‹(*INF* $x{\in}X.\ T \otimes_S S\ x) = (INF\ x{\in}X.\ T \otimes_S S\ x)$› **if** ‹$X \neq \{\}$›
proof −
from *that* **have** ‹$(INF\ x{\in}X.\ S\ x) \otimes_S T = (INF\ x{\in}X.\ S\ x \otimes_S T)$›
by (*rule tensor-ccsubspace-INF-left*)
then have ‹*swap-ell2* $*_S$ $((INF\ x{\in}X.\ S\ x) \otimes_S T) =$ *swap-ell2* $*_S$ $(INF\ x{\in}X.\ S\ x \otimes_S T)$›
by *simp*
then show *?thesis*
by (*cases* ‹$X = \{\}$›)
(*simp-all add*: *swap-ell2-tensor-ccsubspace*)
qed

lemma *tensor-ccsubspace-ccspan*: ‹*ccspan* $X \otimes_S$ *ccspan* $Y =$ *ccspan* $\{x \otimes_s y \mid x\ y.\ x \in X \wedge y \in Y\}$›
proof (*rule antisym*)
show ‹*ccspan* $\{x \otimes_s y \mid x\ y.\ x \in X \wedge y \in Y\} \leq$ *ccspan* $X \otimes_S$ *ccspan* Y›
using *ccspan-superset*[*of X*] *ccspan-superset*[*of Y*]
by (*auto intro*!: *ccspan-mono Collect-mono ex-mono simp add*: *tensor-ccsubspace-def*)
next
have ‹$\{\psi \otimes_s \varphi \mid \psi\ \varphi.\ \psi \in$ *space-as-set* (*ccspan* $X) \wedge \varphi \in$ *space-as-set* (*ccspan* $Y)\}$
$\subseteq$ *closure* $\{x \otimes_s y \mid x\ y.\ x \in$ *cspan* $X \wedge y \in$ *cspan* $Y\}$›
proof (*rule subsetI*)
fix γ
assume ‹$\gamma \in \{\psi \otimes_s \varphi \mid \psi\ \varphi.\ \psi \in$ *space-as-set* (*ccspan* $X) \wedge \varphi \in$ *space-as-set* (*ccspan* $Y)\}$›
then obtain $\psi\ \varphi$ **where** ψ: ‹$\psi \in$ *space-as-set* (*ccspan* $X)$› **and** φ: ‹$\varphi \in$ *space-as-set* (*ccspan* $Y)$› **and** γ-*def*: ‹$\gamma = \psi \otimes_s \varphi$›
by *blast*
from ψ
obtain ψ' **where** *lim1*: ‹$\psi' \longrightarrow \psi$› **and** $\psi' X$: ‹$\psi'\ n \in$ *cspan* X› **for** n
using *closure-sequential* **unfolding** *ccspan.rep-eq* **by** *blast*
from φ
obtain φ' **where** *lim2*: ‹$\varphi' \longrightarrow \varphi$› **and** $\varphi' Y$: ‹$\varphi'\ n \in$ *cspan* Y› **for** n
using *closure-sequential* **unfolding** *ccspan.rep-eq* **by** *blast*
interpret *tensor*: *bounded-cbilinear tensor-ell2*
by (*rule bounded-cbilinear-tensor-ell2*)
from *lim1 lim2* **have** ‹$(\lambda n.\ \psi'\ n \otimes_s \varphi'\ n) \longrightarrow \psi \otimes_s \varphi$›
by (*rule tensor.tendsto*)
moreover have ‹$\psi'\ n \otimes_s \varphi'\ n \in \{x \otimes_s y \mid x\ y.\ x \in$ *cspan* $X \wedge y \in$ *cspan* $Y\}$› **for** n
using $\psi' X\ \varphi' Y$ **by** *auto*
ultimately show ‹$\gamma \in$ *closure* $\{x \otimes_s y \mid x\ y.\ x \in$ *cspan* $X \wedge y \in$ *cspan* $Y\}$›
unfolding γ-*def*
by (*meson closure-sequential*)
qed
also have ‹*closure* $\{x \otimes_s y \mid x\ y.\ x \in$ *cspan* $X \wedge y \in$ *cspan* $Y\}$
$\subseteq$ *closure* (*cspan* $\{x \otimes_s y \mid x\ y.\ x \in X \wedge y \in Y\})$›
proof (*intro closure-mono subsetI*)
fix γ
assume ‹$\gamma \in \{x \otimes_s y \mid x\ y.\ x \in$ *cspan* $X \wedge y \in$ *cspan* $Y\}$›

then obtain $x\ y$ **where** γ*-def*: ‹$\gamma = x \otimes_s y$› **and** ‹$x \in$ *cspan* X› **and** ‹$y \in$ *cspan* Y›
by *blast*
from ‹$x \in$ *cspan* X›
obtain $X'\ x'$ **where** ‹*finite* X'› **and** ‹$X' \subseteq X$› **and** *x-def*: ‹$x = (\sum i{\in}X'.\ x'\ i *_C i)$›
using *complex-vector.span-explicit*[*of* X] **by** *auto*
from ‹$y \in$ *cspan* Y›
obtain $Y'\ y'$ **where** ‹*finite* Y'› **and** ‹$Y' \subseteq Y$› **and** *y-def*: ‹$y = (\sum j{\in}Y'.\ y'\ j *_C j)$›
using *complex-vector.span-explicit*[*of* Y] **by** *auto*
from *x-def y-def* γ*-def*
have ‹$\gamma = (\sum i{\in}X'.\ x'\ i *_C i) \otimes_s (\sum j{\in}Y'.\ y'\ j *_C j)$›
by *simp*
also have ‹$\ldots = (\sum i{\in}X'.\ \sum j{\in}Y'.\ (x'\ i *_C i) \otimes_s (y'\ j *_C j))$›
by (*smt* (*verit*) *sum.cong tensor-ell2-sum-left tensor-ell2-sum-right*)
also have ‹$\ldots = (\sum i{\in}X'.\ \sum j{\in}Y'.\ (x'\ i * y'\ j) *_C (i \otimes_s j))$›
by (*metis* (*no-types*, *lifting*) *scaleC-scaleC sum.cong tensor-ell2-scaleC1 tensor-ell2-scaleC2*)
also have ‹$\ldots \in$ *cspan* $\{x \otimes_s y\ |x\ y.\ x \in X \wedge y \in Y\}$›
using ‹$X' \subseteq X$› ‹$Y' \subseteq Y$›
by (*auto intro*!: *complex-vector.span-sum complex-vector.span-scale*
complex-vector.span-base[*of* ‹- $\otimes_s$ -›])
finally show ‹$\gamma \in$ *cspan* $\{x \otimes_s y\ |x\ y.\ x \in X \wedge y \in Y\}$›
by −
qed
also have ‹$\ldots =$ *space-as-set* (*ccspan* $\{x \otimes_s y\ |x\ y.\ x \in X \wedge y \in Y\}$)›
using *ccspan.rep-eq* **by** *blast*
finally show ‹*ccspan* $X \otimes_S$ *ccspan* $Y \leq$ *ccspan* $\{x \otimes_s y\ |x\ y.\ x \in X \wedge y \in Y\}$›
by (*auto intro*!: *ccspan-leqI simp add*: *tensor-ccsubspace-def*)
qed

lemma *tensor-ccsubspace-mono*: ‹$A \otimes_S B \leq C \otimes_S D$› **if** ‹$A \leq C$› **and** ‹$B \leq D$›
apply (*auto intro*!: *ccspan-mono simp add*: *tensor-ccsubspace-def*)
using *that*
by (*auto simp add*: *less-eq-ccsubspace-def*)

lemma *tensor-ccsubspace-element-as-infsum*:
fixes A :: ‹*'a ell2 ccsubspace*› **and** B :: ‹*'b ell2 ccsubspace*›
assumes ‹$\psi \in$ *space-as-set* $(A \otimes_S B)$›
shows ‹$\exists \varphi\ \delta.\ (\forall n{::}nat.\ \varphi\ n \in$ *space-as-set* $A) \wedge (\forall n.\ \delta\ n \in$ *space-as-set* $B)$
$\wedge\ ((\lambda n.\ \varphi\ n \otimes_s \delta\ n)$ *has-sum* $\psi)$ *UNIV*›
proof −
obtain A' **where** *spanA'*: ‹*ccspan* $A' = A$› **and** *orthoA'*: ‹*is-ortho-set* A'› **and** *normA'*: ‹$a \in A' \Longrightarrow$ *norm* $a = 1$› **for** a
using *some-onb-of-ccspan some-onb-of-is-ortho-set some-onb-of-norm1*
by *blast*
obtain B' **where** *spanB'*: ‹*ccspan* $B' = B$› **and** *orthoB'*: ‹*is-ortho-set* B'› **and** *normB'*: ‹$b \in B' \Longrightarrow$ *norm* $b = 1$› **for** b
using *some-onb-of-ccspan some-onb-of-is-ortho-set some-onb-of-norm1*
by *blast*
define AB' **where** ‹$AB' = \{a \otimes_s b \mid a\ b.\ a \in A' \wedge b \in B'\}$›
define *ABnon0* **where** ‹$ABnon0 = \{ab \in AB'.\ (ab \cdot_C \psi) *_C ab \neq 0\}$›

have *ABnon0-def'*: ‹*ABnon0* = {*ab* ∈ *AB'*. (*norm* (*ab* ·$_C$ *ψ*))^2 ≠ *0*}›
by (*auto simp*: *ABnon0-def*)
have ‹*is-ortho-set AB'*›
by (*simp add*: *AB'-def orthoA' orthoB' tensor-ell2-is-ortho-set*)
have *normAB'*: ‹*ab* ∈ *AB'* ⟹ *norm ab* = *1*› **for** *ab*
by (*auto simp add*: *AB'-def norm-tensor-ell2 normA' normB'*)
have *spanAB'*: ‹*ccspan AB'* = *A* ⊗$_S$ *B*›
by (*simp add*: *tensor-ccsubspace-ccspan AB'-def flip*: *spanA' spanB'*)
have *sum1*: ‹((λ*ab*. (*ab* ·$_C$ *ψ*) *$*$_C$ *ab*) *has-sum ψ*) *AB'*›
apply (*rule basis-projections-reconstruct-has-sum*)
by (*simp-all add*: *spanAB'* ‹*is-ortho-set AB'*› *normAB' assms*)
have ‹(λ*ab*. (*norm* (*ab* ·$_C$ *ψ*))^2) *summable-on AB'*›
by (*rule parseval-identity-summable*)
(*simp-all add*: *spanAB'* ‹*is-ortho-set AB'*› *normAB' assms*)
then have ‹*countable ABnon0*›
using *ABnon0-def' summable-countable-real* **by** *blast*
obtain *f* **and** *N* :: ‹*nat set*› **where** *bij-f*: ‹*bij-betw f N ABnon0*›
using ‹*countable ABnon0*› *countableE-bij* **by** *blast*
then obtain *φ0 δ0* **where** *f-def*: ‹*f n* = *φ0 n* ⊗$_s$ *δ0 n*› **and** *φ0A'*: ‹*φ0 n* ∈ *A'*› **and** *δ0B'*: ‹*δ0 n* ∈ *B'*› **if** ‹*n* ∈ *N*› **for** *n*
apply *atomize-elim*
apply (*subst all-conj-distrib*[*symmetric*] *choice-iff*[*symmetric*])+
apply (*simp add*: *bij-betw-def ABnon0-def*)
using *AB'-def* ‹*bij-betw f N ABnon0*› *bij-betwE mem-Collect-eq* **by** *blast*
define *c* **where** ‹*c n* = (*φ0 n* ⊗$_s$ *δ0 n*) ·$_C$ *ψ*› **for** *n*
from *sum1* **have** ‹((λ*ab*. (*ab* ·$_C$ *ψ*) *$*$_C$ *ab*) *has-sum ψ*) *ABnon0*›
by (*rule has-sum-cong-neutral*[*THEN iffD1*, *rotated* −*1*]) (*auto simp*: *ABnon0-def*)
then have ‹((λ*n*. (*f n* ·$_C$ *ψ*) *$*$_C$ *f n*) *has-sum ψ*) *N*›
by (*rule has-sum-reindex-bij-betw*[*OF bij-f*, *THEN iffD2*])
then have *sum2*: ‹((λ*n*. *c n* *$*$_C$ (*φ0 n* ⊗$_s$ *δ0 n*)) *has-sum ψ*) *N*›
by (*rule has-sum-cong*[*THEN iffD1*, *rotated*]) (*simp add*: *f-def c-def*)
define *φ δ* **where** ‹*φ n* = (*if n*∈*N then c n* *$*$_C$ *φ0 n else 0*)› **and** ‹*δ n* = (*if n*∈*N then δ0 n else 0*)› **for** *n*
then have *1*: ‹*φ n* ∈ *space-as-set A*› **and** *2*: ‹*δ n* ∈ *space-as-set B*› **for** *n*
using *φ0A' δ0B' spanA' spanB' ccspan-superset*
by (*auto intro*!: *complex-vector.subspace-scale simp*: *φ-def δ-def*)
from *sum2* **have** *sum3*: ‹((λ*n*. *φ n* ⊗$_s$ *δ n*) *has-sum ψ*) *UNIV*›
by (*rule has-sum-cong-neutral*[*THEN iffD2*, *rotated* −*1*])
(*auto simp*: *φ-def δ-def tensor-ell2-scaleC1*)
from *1 2 sum3* **show** *?thesis*
by *auto*
qed

lemma *ortho-tensor-ccsubspace-right*: ‹− (⊤ ⊗$_S$ *A*) = ⊤ ⊗$_S$ (− *A*)›
proof −
have [*simp*]: ‹*is-Proj* (*id-cblinfun* ⊗$_o$ *Proj X*)› **for** *X*
by (*metis Proj-is-Proj Proj-top is-Proj-tensor-op*)

have ‹*Proj* (− (⊤ ⊗$_S$ *A*)) = *id-cblinfun* − *Proj* (⊤ ⊗$_S$ *A*)›

by (*simp add*: *Proj-ortho-compl*)
also have ‹... = *id-cblinfun* − (*id-cblinfun* ⊗$_o$ *Proj A*)›
by (*simp add*: *tensor-ccsubspace-via-Proj Proj-on-own-range*)
also have ‹... = *id-cblinfun* ⊗$_o$ (*id-cblinfun* − *Proj A*)›
by (*metis cblinfun.diff-right left-amplification.rep-eq tensor-id*)
also have ‹... = *Proj* ⊤ ⊗$_o$ *Proj* (− *A*)›
by (*simp add*: *Proj-ortho-compl*)
also have ‹... = *Proj* (⊤ ⊗$_S$ (− *A*))›
by (*simp add*: *tensor-ccsubspace-via-Proj Proj-on-own-range*)
finally show *?thesis*
using *Proj-inj* **by** *blast*
qed

lemma *ortho-tensor-ccsubspace-left*: ‹− (*A* ⊗$_S$ ⊤) = (− *A*) ⊗$_S$ ⊤›
proof −
have ‹− (*A* ⊗$_S$ ⊤) = *swap-ell2* ∗$_S$ (− (⊤ ⊗$_S$ *A*))›
by (*simp add*: *unitary-image-ortho-compl swap-ell2-tensor-ccsubspace*)
also have ‹... = *swap-ell2* ∗$_S$ (⊤ ⊗$_S$ (− *A*))›
by (*simp add*: *ortho-tensor-ccsubspace-right*)
also have ‹... = (− *A*) ⊗$_S$ ⊤›
by (*simp add*: *swap-ell2-tensor-ccsubspace*)
finally show *?thesis*
by −
qed

lemma *kernel-tensor-id-left*: ‹*kernel* (*id-cblinfun* ⊗$_o$ *A*) = ⊤ ⊗$_S$ *kernel A*›
proof −
have ‹*kernel* (*id-cblinfun* ⊗$_o$ *A*) = − ((*id-cblinfun* ⊗$_o$ *A*)∗ ∗$_S$ ⊤)›
by (*rule kernel-compl-adj-range*)
also have ‹... = − (*id-cblinfun* ∗$_S$ ⊤ ⊗$_S$ *A*∗ ∗$_S$ ⊤)›
by (*metis cblinfun-image-id id-cblinfun-adjoint tensor-ccsubspace-image tensor-ccsubspace-top tensor-op-adjoint*)
also have ‹... = ⊤ ⊗$_S$ (− (*A*∗ ∗$_S$ ⊤))›
by (*simp add*: *ortho-tensor-ccsubspace-right*)
also have ‹... = ⊤ ⊗$_S$ *kernel A*›
by (*simp add*: *kernel-compl-adj-range*)
finally show *?thesis*
by −
qed

lemma *kernel-tensor-id-right*: ‹*kernel* (*A* ⊗$_o$ *id-cblinfun*) = *kernel A* ⊗$_S$ ⊤›
proof −
have *ker-swap*: ‹*kernel swap-ell2* = *0*›
by (*simp add*: *kernel-isometry*)
have ‹*kernel* (*id-cblinfun* ⊗$_o$ *A*) = ⊤ ⊗$_S$ *kernel A*›
by (*rule kernel-tensor-id-left*)
from *this*[*THEN arg-cong*, *of* ‹*cblinfun-image swap-ell2*›]
show *?thesis*
by (*simp add*: *swap-ell2-tensor-ccsubspace cblinfun-image-kernel-unitary*

flip: *swap-ell2-commute-tensor-op kernel-cblinfun-compose*[*OF ker-swap*])
qed

lemma *eigenspace-tensor-id-left*: ‹*eigenspace c* (*id-cblinfun* $\otimes_o$ *A*) = $\top$ $\otimes_S$ *eigenspace c A*›
proof −
have ‹*eigenspace c* (*id-cblinfun* $\otimes_o$ *A*) = *kernel* (*id-cblinfun* $\otimes_o$ (*A* − *c* $*_C$ *id-cblinfun*))›
unfolding *eigenspace-def*
by (*metis* (*no-types*, *lifting*) *complex-vector.scale-minus-left tensor-id tensor-op-right-add tensor-op-scaleC-right uminus-add-conv-diff*)
also have ‹*kernel* (*id-cblinfun* $\otimes_o$ (*A* − *c* $*_C$ *id-cblinfun*)) = $\top$ $\otimes_S$ *kernel* (*A* − *c* $*_C$ *id-cblinfun*)›
by (*simp add*: *kernel-tensor-id-left*)
also have ‹... = $\top$ $\otimes_S$ *eigenspace c A*›
by (*simp add*: *eigenspace-def*)
finally show *?thesis*
by −
qed

lemma *eigenspace-tensor-id-right*: ‹*eigenspace c* (*A* $\otimes_o$ *id-cblinfun*) = *eigenspace c A* $\otimes_S$ $\top$›
proof −
have ‹*eigenspace c* (*id-cblinfun* $\otimes_o$ *A*) = $\top$ $\otimes_S$ *eigenspace c A*›
by (*rule eigenspace-tensor-id-left*)
from *this*[*THEN arg-cong*, *of* ‹*cblinfun-image swap-ell2*›]
show *?thesis*
by (*simp add*: *swap-ell2-commute-tensor-op cblinfun-image-eigenspace-unitary swap-ell2-tensor-ccsubspace*)
qed

end

15 *Partial-Trace* − The partial trace

theory *Partial-Trace*
imports *Trace-Class Hilbert-Space-Tensor-Product*
begin

hide-fact (**open**) *Infinite-Set-Sum.abs-summable-on-Sigma-iff*
hide-fact (**open**) *Infinite-Set-Sum.abs-summable-on-comparison-test*
hide-const (**open**) *Determinants.trace*
hide-fact (**open**) *Determinants.trace-def*

definition *partial-trace* :: ‹(('*a* × '*c*) *ell2*, ('*b* × '*c*) *ell2*) *trace-class* ⇒ ('*a ell2*, '*b ell2*) *trace-class*› **where**
‹*partial-trace t* = ($\sum_\infty$*j*. *compose-tcl* (*compose-tcr* ((*tensor-ell2-right* (*ket j*))*) *t*) (*tensor-ell2-right* (*ket j*)))›

lemma *partial-trace-def'*: ‹*partial-trace t* = ($\sum_\infty$*j*. *sandwich-tc* ((*tensor-ell2-right* (*ket j*))*) *t*)›
— We cannot use this as the definition of *partial-trace* because this definition has a more restricted

type (t is a square operator).
by (*auto intro*!: *simp*: *partial-trace-def sandwich-tc-def*)

lemma *partial-trace-abs-summable*:
‹(λ*j*. *compose-tcl* (*compose-tcr* ((*tensor-ell2-right* (*ket j*))∗) *t*) (*tensor-ell2-right* (*ket j*))) *abs-summable-on UNIV*›
and *partial-trace-has-sum*:
‹((λ*j*. *compose-tcl* (*compose-tcr* ((*tensor-ell2-right* (*ket j*))∗) *t*) (*tensor-ell2-right* (*ket j*))) *has-sum partial-trace t*) *UNIV*›
and *partial-trace-norm-reducing*: ‹*norm* (*partial-trace t*) ≤ *norm t*›
proof −
define t' **where** ‹t' = *from-trace-class t*›
define *s* **where** ‹*s k* = *compose-tcl* (*compose-tcr* ((*tensor-ell2-right* (*ket k*))∗) *t*) (*tensor-ell2-right* (*ket k*))› **for** *k*

have *bound*: ‹(∑ *k*∈*F*. *norm* (*s k*)) ≤ *norm t*›
if ‹*F* ∈ {*F*. *F* ⊆ *UNIV* ∧ *finite F*}›
for *F* :: ‹'*a set*›
proof −
from *that* **have** [*simp*]: ‹*finite F*›
by *force*
define *tk* **where** ‹*tk k* = *tensor-ell2-right* (*ket k*)∗ *o*$_{CL}$ t' *o*$_{CL}$ *tensor-ell2-right* (*ket k*)› **for** *k*
have *tc-t'*[*simp*]: ‹*trace-class t'*›
by (*simp add*: t'*-def*)
then have *tc-tk*[*simp*]: ‹*trace-class* (*tk k*)› **for** *k*
by (*simp add*: *tk-def trace-class-comp-left trace-class-comp-right*)
define *uk* **where** ‹*uk k* = (*polar-decomposition* (*tk k*))∗› **for** *k*
define *u* **where** ‹*u* = (∑ *k*∈*F*. *uk k* ⊗$_o$ *butterfly* (*ket k*) (*ket k*))›
define *B* :: ‹'*b ell2 set*› **where** ‹*B* = *range ket*›

have *aux1*: ‹*tensor-ell2-right* (*ket x*)∗ ∗$_V$ *u* ∗$_V$ *a* = *0*› **if** ‹*x* ∉ *F*› **for** *x a*
proof −
have ∗: ‹*u*∗ *o*$_{CL}$ *tensor-ell2-right* (*ket x*) = *0*›
by (*auto intro*!: *equal-ket simp*: *u-def sum-adj tensor-op-adjoint tensor-ell2-right-apply cblinfun.sum-left tensor-op-ell2 cinner-ket sum-single*[**where** *i*=*x*] ‹*x* ∉ *F*›)
have ‹*tensor-ell2-right* (*ket x*)∗ *o*$_{CL}$ *u* = *0*›
by (*rule adj-inject*[*THEN iffD1*]) (*use* ∗ **in** *simp*)
then show *?thesis*
by (*simp flip*: *cblinfun-apply-cblinfun-compose*)
qed

have *aux2*: ‹*uk x* ∗$_V$ *tensor-ell2-right* (*ket x*)∗ ∗$_V$ *a* = *tensor-ell2-right* (*ket x*)∗ ∗$_V$ *u* ∗$_V$ *a*›
if ‹*x* ∈ *F*› **for** *x a*
proof −
have ∗: ‹*tensor-ell2-right* (*ket x*) *o*$_{CL}$ (*uk x*)∗ = *u*∗ *o*$_{CL}$ *tensor-ell2-right* (*ket x*)›
by (*auto intro*!: *equal-ket simp*: *u-def sum-adj tensor-op-adjoint tensor-ell2-right-apply cblinfun.sum-left tensor-op-ell2* ‹*x* ∈ *F*› *cinner-ket sum-single*[**where** *i*=*x*])
have ‹*uk x* *o*$_{CL}$ *tensor-ell2-right* (*ket x*)∗ = *tensor-ell2-right* (*ket x*)∗ *o*$_{CL}$ *u*›

```
    by (rule adj-inject[THEN iffD1]) (use ∗ in simp)
  then show ?thesis
    by (simp flip: cblinfun-apply-cblinfun-compose)
qed

have sum1: ‹(λ(x, y). ket (y, x) ·C (u ∗V t′ ∗V ket (y, x))) summable-on UNIV›
proof −
  have ‹trace-class (u oCL t′)›
    by (simp add: trace-class-comp-right)
  then have ‹(λyx. yx ·C ((u oCL t′) ∗V yx)) summable-on (range ket)›
    using is-onb-ket trace-exists by blast
  then have ‹(λyx. ket yx ·C ((u oCL t′) ∗V ket yx)) summable-on UNIV›
    apply (subst summable-on-reindex-bij-betw[where g=ket and A=UNIV and B=‹range
ket›])
     using bij-betw-def inj-ket by blast
  then show ?thesis
   by (subst summable-on-reindex-bij-betw[where g=prod.swap and A=UNIV, symmetric])
auto
qed

have norm-u: ‹norm u ≤ 1›
proof −
  define u2 uk2 where ‹u2 = u∗ oCL u› and ‹uk2 k = (uk k)∗ oCL uk k› for k
  have ∗: ‹(∑ i∈F. (uk i∗ oCL uk k) ⊗o (ket i ·C ket k) ∗C butterfly (ket i) (ket k))
      = (uk k∗ oCL uk k) ⊗o butterfly (ket k) (ket k)› if [simp]: ‹k ∈ F› for k
    apply (subst sum-single[where i=k])
    by (auto simp: cinner-ket)
  have ∗∗: ‹(∑ ka∈F. (uk2 ka oCL uk2 k) ⊗o (ket ka ·C ket k) ∗C butterfly (ket ka) (ket k))
      = (uk2 k oCL uk2 k) ⊗o butterfly (ket k) (ket k)› if [simp]: ‹k ∈ F› for k
    apply (subst sum-single[where i=k])
    by (auto simp: cinner-ket)
  have proj-uk2: ‹is-Proj (uk2 k)› for k
    unfolding uk2-def
    apply (rule partial-isometry-square-proj)
    by (auto intro!: partial-isometry-square-proj partial-isometry-adj simp: uk-def)
  have u2-explicit: ‹u2 = (∑ k∈F. uk2 k ⊗o butterfly (ket k) (ket k))›
    by (simp add: u2-def u-def sum-adj tensor-op-adjoint cblinfun-compose-sum-right
        cblinfun-compose-sum-left tensor-butter comp-tensor-op ∗ uk2-def)
  have ‹u2∗ = u2›
    by (simp add: u2-def)
  moreover have ‹u2 oCL u2 = u2›
    by (simp add: u2-explicit cblinfun-compose-sum-right cblinfun-compose-sum-left
        comp-tensor-op ∗∗ proj-uk2 is-Proj-idempotent)
  ultimately have ‹is-Proj u2›
    by (simp add: is-Proj-I)
  then have ‹norm u2 ≤ 1›
    using norm-is-Proj by blast
  then show ‹norm u ≤ 1›
    by (simp add: power-le-one-iff norm-AAadj u2-def)
```

qed

have ‹$(\sum k{\in}F.\ norm\ (s\ k))$
$= (\sum k{\in}F.\ trace\text{-}norm\ (tk\ k))$›
by (*simp add*: *s-def tk-def norm-trace-class.rep-eq compose-tcl.rep-eq compose-tcr.rep-eq t′-def*)
also have ‹$\ldots = cmod\ (\sum k{\in}F.\ trace\ (uk\ k\ o_{CL}\ tk\ k))$›
by (*smt* (*verit*, *best*) *norm-of-real of-real-hom.hom-sum polar-decomposition-correct′ sum.cong sum-nonneg trace-abs-op trace-norm-nneg uk-def*)
also have ‹$\ldots = cmod\ (\sum k{\in}F.\ trace\ (tensor\text{-}ell2\text{-}right\ (ket\ k){*}\ o_{CL}\ u\ o_{CL}\ t'\ o_{CL}\ tensor\text{-}ell2\text{-}right\ (ket\ k)))$›
apply (*rule arg-cong*[**where** *f*=*cmod*], *rule sum.cong*[*OF refl*], *rule arg-cong*[**where** *f*=*trace*])
by (*auto intro*!: *equal-ket simp*: *tk-def aux2*)
also have ‹$\ldots = cmod\ (\sum k{\in}F.\ \sum_\infty j.\ ket\ j \cdot_C ((tensor\text{-}ell2\text{-}right\ (ket\ k){*}\ o_{CL}\ u\ o_{CL}\ t'\ o_{CL}\ tensor\text{-}ell2\text{-}right\ (ket\ k))\ *_V\ ket\ j))$›
by (*auto intro*!: *sum.cong simp*: *trace-ket-sum trace-class-comp-left trace-class-comp-right*)
also have ‹$\ldots = cmod\ (\sum_\infty k{\in}F.\ \sum_\infty j.\ ket\ j \cdot_C ((tensor\text{-}ell2\text{-}right\ (ket\ k){*}\ o_{CL}\ u\ o_{CL}\ t'\ o_{CL}\ tensor\text{-}ell2\text{-}right\ (ket\ k))\ *_V\ ket\ j))$›
by (*simp add*: ‹*finite F*›)
also have ‹$\ldots = cmod\ (\sum_\infty k.\ \sum_\infty j.\ ket\ j \cdot_C ((tensor\text{-}ell2\text{-}right\ (ket\ k){*}\ o_{CL}\ u\ o_{CL}\ t'\ o_{CL}\ tensor\text{-}ell2\text{-}right\ (ket\ k))\ *_V\ ket\ j))$›
apply (*rule arg-cong*[**where** *f*=*cmod*])
apply (*rule infsum-cong-neutral*)
by (*auto simp*: *aux1*)
also have ‹$\ldots = cmod\ (\sum_\infty k.\ \sum_\infty j.\ ket\ (j,k) \cdot_C ((u\ o_{CL}\ t')\ *_V\ ket\ (j,k)))$›
apply (*rule arg-cong*[**where** *f*=*cmod*], *rule infsum-cong*, *rule infsum-cong*)
by (*simp add*: *tensor-ell2-right-apply cinner-adj-right tensor-ell2-ket*)
also have ‹$\ldots = cmod\ (\sum_\infty (k,j).\ ket\ (j,k) \cdot_C ((u\ o_{CL}\ t')\ *_V\ ket\ (j,k)))$›
apply (*rule arg-cong*[**where** *f*=*cmod*])
apply (*subst infsum-Sigma′-banach*)
using *sum1* **by** *auto*
also have ‹$\ldots = cmod\ (\sum_\infty jk.\ ket\ jk \cdot_C ((u\ o_{CL}\ t')\ *_V\ ket\ jk))$›
apply (*subst infsum-reindex-bij-betw*[**where** *g*=*prod.swap* **and** *A*=*UNIV*, *symmetric*])
by *auto*
also have ‹$\ldots = cmod\ (trace\ (u\ o_{CL}\ t'))$›
by (*simp add*: *trace-ket-sum trace-class-comp-right*)
also have ‹$\ldots \leq trace\text{-}norm\ (u\ o_{CL}\ t')$›
using *trace-leq-trace-norm* **by** *blast*
also have ‹$\ldots \leq norm\ u * trace\text{-}norm\ t'$›
by (*simp add*: *trace-norm-comp-right*)
also have ‹$\ldots \leq trace\text{-}norm\ t'$›
using *norm-u*
by (*metis more-arith-simps*(5) *mult-right-mono trace-norm-nneg*)
also have ‹$\ldots = norm\ t$›
by (*simp add*: *norm-trace-class.rep-eq t′-def*)
finally show ‹$(\sum k{\in}F.\ norm\ (s\ k)) \leq norm\ t$›
by −
qed

show *abs-summable*: ‹*s abs-summable-on UNIV*›
by (*intro nonneg-bdd-above-summable-on bdd-aboveI2*[**where** *M*=‹*norm t*›] *norm-ge-zero bound*)

from *abs-summable*
show *has-sum*: ‹(*s has-sum partial-trace t*) *UNIV*›
by (*simp add*: *abs-summable-summable partial-trace-def s-def*[*abs-def*] *t'-def*)

show ‹*norm* (*partial-trace t*) ≤ *norm t*›
proof −
have ‹*norm* (*partial-trace t*) ≤ (∑∞*k*. *norm* (*s k*))›
using - *has-sum* **apply** (*rule norm-has-sum-bound*)
using *abs-summable has-sum-infsum* **by** *blast*
also from *bound* **have** ‹(∑∞*k*. *norm* (*s k*)) ≤ *norm t*›
by (*simp add*: *abs-summable infsum-le-finite-sums*)
finally show *?thesis*
by −
qed
qed

lemma *partial-trace-abs-summable'*:
‹(λ*j*. *sandwich-tc* ((*tensor-ell2-right* (*ket j*))*) *t*) *abs-summable-on UNIV*›
and *partial-trace-has-sum'*:
‹((λ*j*. *sandwich-tc* ((*tensor-ell2-right* (*ket j*))*) *t*) *has-sum partial-trace t*) *UNIV*›
using *partial-trace-abs-summable partial-trace-has-sum*
by (*auto intro*!: *simp*: *sandwich-tc-def sandwich-apply*)

lemma *trace-partial-trace-compose-eq-trace-compose-tensor-id*:
‹*trace* (*from-trace-class* (*partial-trace t*) *o*$_{CL}$ *x*) = *trace* (*from-trace-class t* *o*$_{CL}$ (*x* ⊗*o* *id-cblinfun*))›
proof −
define *s* **where** ‹*s* = *trace* (*from-trace-class* (*partial-trace t*) *o*$_{CL}$ *x*)›
define *s'* **where** ‹*s' e* = *ket e* •$_C$ ((*from-trace-class* (*partial-trace t*) *o*$_{CL}$ *x*) *$_V$ *ket e*)› **for** *e*
define *u* **where** ‹*u j* = *compose-tcl* (*compose-tcr* ((*tensor-ell2-right* (*ket j*))*) *t*) (*tensor-ell2-right* (*ket j*))› **for** *j*
define *u'* **where** ‹*u' e j* = *ket e* •$_C$ (*from-trace-class* (*u j*) *$_V$ *x* *$_V$ *ket e*)› **for** *e j*
have ‹(*u has-sum partial-trace t*) *UNIV*›
using *partial-trace-has-sum*[*of t*]
by (*simp add*: *u-def*[*abs-def*])
then have ‹((λ*u*. *from-trace-class u* *$_V$ *x* *$_V$ *ket e*) *o u has-sum from-trace-class* (*partial-trace t*) *$_V$ *x* *$_V$ *ket e*) *UNIV*› **for** *e*
proof (*rule has-sum-comm-additive*[*rotated* −*1*])
show ‹*Modules.additive* (λ*u*. *from-trace-class u* *$_V$ *x* *$_V$ *ket e*)›
by (*simp add*: *Modules.additive-def cblinfun.add-left plus-trace-class.rep-eq*)
have *bounded-clinear*: ‹*bounded-clinear* (λ*u*. *from-trace-class u* *$_V$ *x* *$_V$ *ket e*)›
proof (*rule bounded-clinearI*[**where** *K*=‹*norm* (*x* *$_V$ *ket e*)›])
show ‹*from-trace-class* (*b1* + *b2*) *$_V$ *x* *$_V$ *ket e* = *from-trace-class b1* *$_V$ *x* *$_V$ *ket e* +

from-trace-class b2 $*_V$ *x* $*_V$ *ket e*› **for** *b1 b2*
by (*simp add*: *plus-cblinfun.rep-eq plus-trace-class.rep-eq*)
show ‹*from-trace-class* (*r* $*_C$ *b*) $*_V$ *x* $*_V$ *ket e* = *r* $*_C$ (*from-trace-class b* $*_V$ *x* $*_V$ *ket e*)› **for** *b r*
by (*simp add*: *scaleC-trace-class.rep-eq*)
show ‹*norm* (*from-trace-class t* $*_V$ *x* $*_V$ *ket e*) ≤ *norm t* ∗ *norm* (*x* $*_V$ *ket e*)› **for** *t*
proof −
have ‹*norm* (*from-trace-class t* $*_V$ *x* $*_V$ *ket e*) ≤ *norm* (*from-trace-class t*) ∗ *norm* (*x* $*_V$ *ket e*)›
by (*simp add*: *norm-cblinfun*)
also have ‹. . . ≤ *norm t* ∗ *norm* (*x* $*_V$ *ket e*)›
by (*auto intro*!: *mult-right-mono simp add*: *norm-leq-trace-norm norm-trace-class.rep-eq*)
finally show *?thesis*
by −
qed
qed
have ‹*isCont* (λ*u*. *from-trace-class u* $*_V$ *x* $*_V$ *ket e*) (*partial-trace t*)›
using *bounded-clinear clinear-continuous-at* **by** *auto*
then show ‹(λ*u*. *from-trace-class u* $*_V$ *x* $*_V$ *ket e*) −*partial-trace t*→ *from-trace-class* (*partial-trace t*) $*_V$ *x* $*_V$ *ket e*›
by (*simp add*: *isCont-def*)
qed
then have ‹((λ*v*. *ket e* $•_C$ *v*) *o* ((λ*u*. *from-trace-class u* $*_V$ *x* $*_V$ *ket e*) *o u*) *has-sum ket e* $•_C$ (*from-trace-class* (*partial-trace t*) $*_V$ *x* $*_V$ *ket e*)) *UNIV*› **for** *e*
proof (*rule has-sum-comm-additive*[*rotated* −1])
show ‹*Modules.additive* (λ*v*. *ket e* $•_C$ *v*)›
by (*simp add*: *Modules.additive-def cinner-simps*(2))
have *bounded-clinear*: ‹*bounded-clinear* (λ*v*. *ket e* $•_C$ *v*)›
using *bounded-clinear-cinner-right* **by** *auto*
then have ‹*isCont* (λ*v*. *ket e* $•_C$ *v*) *l*› **for** *l*
by *simp*
then show ‹(λ*v*. *ket e* $•_C$ *v*) −*l*→ *ket e* $•_C$ *l*› **for** *l*
by (*simp add*: *isContD*)
qed
then have *has-sum-u'*: ‹((λ*j*. *u' e j*) *has-sum s' e*) *UNIV*› **for** *e*
by (*simp add*: *o-def u'-def s'-def*)
then have *infsum-u'*: ‹*s' e* = *infsum* (*u' e*) *UNIV*› **for** *e*
by (*metis infsumI*)
have *tc-u-x*[*simp*]: ‹*trace-class* (*from-trace-class* (*u j*) o_{CL} *x*)› **for** *j*
by (*simp add*: *trace-class-comp-left*)

have *summable-u'-pairs*: ‹(λ(*e*, *j*). *u' e j*) *summable-on UNIV* × *UNIV*›
proof −
have ‹*trace-class* (*from-trace-class t* o_{CL} (*x* ⊗$_o$ *id-cblinfun*))›
by (*simp add*: *trace-class-comp-left*)
from *trace-exists*[*OF is-onb-ket this*]
have ‹(λ*ej*. *ket ej* $•_C$ (*from-trace-class t* $*_V$ (*x* ⊗$_o$ *id-cblinfun*) $*_V$ *ket ej*)) *summable-on UNIV*›
by (*simp-all add*: *summable-on-reindex o-def*)

```
    then show ?thesis
      by (simp-all add: o-def u'-def[abs-def] u-def
        trace-class-comp-left trace-class-comp-right Abs-trace-class-inverse tensor-ell2-right-apply

         ket-pair-split tensor-op-ell2 case-prod-unfold cinner-adj-right
         compose-tcl.rep-eq compose-tcr.rep-eq)
  qed

  have u'-tensor: ‹u' e j = ket (e,j) •C ((from-trace-class t oCL (x ⊗o id-cblinfun)) *V ket (e,j))›
for e j
    by (simp add: u'-def u-def tensor-op-ell2 tensor-ell2-right-apply  Abs-trace-class-inverse
      trace-class-comp-left trace-class-comp-right cinner-adj-right compose-tcl.rep-eq compose-tcr.rep-eq
       flip: tensor-ell2-ket)

  have ‹((λe. e •C ((from-trace-class (partial-trace t) oCL x) *V e)) has-sum s) (range ket)›
    unfolding s-def
    apply (rule trace-has-sum)
    by (auto simp: trace-class-comp-left)
  then have ‹(s' has-sum s) UNIV›
    apply (subst (asm) has-sum-reindex)
    by (auto simp: o-def s'-def[abs-def])
  then have ‹s = infsum s' UNIV›
    by (simp add: infsumI)
  also have ‹... = infsum (λe. infsum (u' e) UNIV) UNIV›
    using infsum-u' by presburger
  also have ‹... = (∑∞(e, j)∈UNIV. u' e j)›
    apply (subst infsum-Sigma'-banach)
     apply (rule summable-u'-pairs)
    by simp
  also have ‹... = trace (from-trace-class t oCL (x ⊗o id-cblinfun))›
    unfolding u'-tensor
    by (simp add: trace-ket-sum cond-case-prod-eta trace-class-comp-left)
  finally show ?thesis
    by (simp add: s-def)
qed



lemma right-amplification-weak-star-cont[simp]:
  ‹continuous-map weak-star-topology weak-star-topology (λa. a ⊗o id-cblinfun)›
  — Logically does not belong in this theory but uses the partial trace in the proof.
proof (unfold weak-star-topology-def', rule continuous-map-pullback-both)
  show ‹S ⊆ f -‘ UNIV› for S :: ‹'x set› and f :: ‹'x ⇒ 'y›
    by simp
  define g' :: ‹(('b ell2, 'a ell2) trace-class ⇒ complex) ⇒ (('b × 'c) ell2, ('a × 'c) ell2)
trace-class ⇒ complex› where
    ‹g' τ t = τ (partial-trace t)› for τ t
  have ‹continuous-on UNIV g'›
    by (simp add: continuous-on-coordinatewise-then-product g'-def)
```

then show ‹*continuous-map euclidean euclidean g'*›
using *continuous-map-iff-continuous2* **by** *blast*
show ‹*g'* (λ*t*. *trace* (*from-trace-class t* o_{CL} *x*)) =
(λ*t*. *trace* (*from-trace-class t* o_{CL} *x* ⊗$_o$ *id-cblinfun*))› **for** *x*
by (*auto intro*!: *ext simp*: *g'-def trace-partial-trace-compose-eq-trace-compose-tensor-id*)
qed

lemma *left-amplification-weak-star-cont*[*simp*]:
‹*continuous-map weak-star-topology weak-star-topology* (λ*b*. *id-cblinfun* ⊗$_o$ *b* :: (′*c*×′*a*) *ell2* ⇒$_{CL}$ (′*c*×′*b*) *ell2*)›
— Logically does not belong in this theory but uses the partial trace in the proof.
proof −
have ‹*continuous-map weak-star-topology weak-star-topology* (
(λ*x*. *x* o_{CL} *swap-ell2*) *o* (λ*x*. *swap-ell2* o_{CL} *x*) *o* (λ*a*. *a* ⊗$_o$ *id-cblinfun* :: (′*a*×′*c*) *ell2* ⇒$_{CL}$ (′*b*×′*c*) *ell2*))›
by (*auto intro*!: *continuous-map-compose*[**where** *X'*=*weak-star-topology*]
continuous-map-left-comp-weak-star continuous-map-right-comp-weak-star)
then show *?thesis*
by (*auto simp*: *o-def*)
qed

lemma *partial-trace-plus*: ‹*partial-trace* (*t* + *u*) = *partial-trace t* + *partial-trace u*›
proof −
from *partial-trace-has-sum*[*of t*] **and** *partial-trace-has-sum*[*of u*]
have ‹((λ*j*. *compose-tcl* (*compose-tcr* ((*tensor-ell2-right* (*ket j*))∗) *t*) (*tensor-ell2-right* (*ket j*))
+ *compose-tcl* (*compose-tcr* ((*tensor-ell2-right* (*ket j*))∗) *u*) (*tensor-ell2-right* (*ket j*)))
has-sum
partial-trace t + *partial-trace u*) *UNIV*› (**is** ‹(*?f has-sum* -) -›)
by (*rule has-sum-add*)
moreover have ‹*?f j* = *compose-tcl* (*compose-tcr* ((*tensor-ell2-right* (*ket j*))∗) (*t* + *u*))
(*tensor-ell2-right* (*ket j*))› (**is** ‹*?f j* = *?g j*›) **for** *j*
by (*simp add*: *compose-tcl.add-left compose-tcr.add-right*)
ultimately have ‹(*?g has-sum partial-trace t* + *partial-trace u*) *UNIV*›
by *simp*
moreover have ‹(*?g has-sum partial-trace* (*t* + *u*)) *UNIV*›
by (*simp add*: *partial-trace-has-sum*)
ultimately show *?thesis*
using *has-sum-unique* **by** *blast*
qed

lemma *partial-trace-scaleC*: ‹*partial-trace* (*c* ∗$_C$ *t*) = *c* ∗$_C$ *partial-trace t*›
by (*simp add*: *partial-trace-def infsum-scaleC-right compose-tcr.scaleC-right compose-tcl.scaleC-left*)

lemma *partial-trace-tensor*: ‹*partial-trace* (*tc-tensor t u*) = *trace-tc u* ∗$_C$ *t*›
proof −
define *t' u'* **where** ‹*t'* = *from-trace-class t*› **and** ‹*u'* = *from-trace-class u*›
have *1*: ‹(λ*j*. *ket j* •$_C$ (*from-trace-class u* ∗$_V$ *ket j*)) *summable-on UNIV*›
using *trace-exists*[**where** *B*=‹*range ket*› **and** *A*=‹*from-trace-class u*›]

```
    by (simp add: summable-on-reindex o-def)
  have ‹partial-trace (tc-tensor t u) =
    (∑∞j. compose-tcl (compose-tcr (tensor-ell2-right (ket j)∗) (tc-tensor t u)) (tensor-ell2-right
(ket j)))›
    by (simp add: partial-trace-def)
  also have ‹... = (∑∞j. (ket j •C (from-trace-class u *V ket j)) *C t)›
  proof −
    have ∗: ‹tensor-ell2-right (ket j)∗ oCL t' ⊗o u' oCL tensor-ell2-right (ket j) =
        (ket j •C (u' *V ket j)) *C t'› for j
      by (auto intro!: cblinfun-eqI simp: tensor-op-ell2)
    show ?thesis
    apply (rule infsum-cong)
      by (auto intro!: from-trace-class-inject[THEN iffD1] simp flip: t'-def u'-def
        simp: ∗ compose-tcl.rep-eq compose-tcr.rep-eq tc-tensor.rep-eq scaleC-trace-class.rep-eq)
  qed
  also have ‹... = trace-tc u *C t›
   by (auto intro!: infsum-scaleC-left simp: trace-tc-def trace-alt-def[OF is-onb-ket] infsum-reindex
o-def 1)
  finally show ?thesis
    by −
qed

lemma bounded-clinear-partial-trace[bounded-clinear, iff]: ‹bounded-clinear partial-trace›
  apply (rule bounded-clinearI[where K=1])
  by (auto simp add: partial-trace-plus partial-trace-scaleC partial-trace-norm-reducing)

lemma vector-sandwich-partial-trace-has-sum:
  ‹((λz. ((x ⊗s ket z) •C (from-trace-class ϱ *V (y ⊗s ket z))))
      has-sum x •C (from-trace-class (partial-trace ϱ) *V y)) UNIV›
proof −
  define xϱy where ‹xϱy = x •C (from-trace-class (partial-trace ϱ) *V y)›
   have ‹((λj. compose-tcl (compose-tcr ((tensor-ell2-right (ket j))∗) ϱ) (tensor-ell2-right (ket
j)))
       has-sum partial-trace ϱ) UNIV›
    using partial-trace-has-sum by force
  then have ‹((λj. x •C (from-trace-class (compose-tcl (compose-tcr ((tensor-ell2-right (ket j))∗)
ϱ) (tensor-ell2-right (ket j))) *V y))
       has-sum xϱy) UNIV›
    unfolding xϱy-def
    apply (rule Infinite-Sum.has-sum-bounded-linear[rotated])
    by (intro bounded-clinear.bounded-linear bounded-linear-intros)
  then have ‹((λj. x •C (tensor-ell2-right (ket j)∗ *V from-trace-class ϱ *V y ⊗s ket j)) has-sum
    xϱy) UNIV›
    by (simp add: compose-tcl.rep-eq compose-tcr.rep-eq)
  then show ?thesis
    by (metis (no-types, lifting) cinner-adj-right has-sum-cong tensor-ell2-right-apply xϱy-def)
qed

lemma vector-sandwich-partial-trace:
```

‹x $\cdot_C$ (*from-trace-class* (*partial-trace* ϱ) $*_V$ y) =
($\sum_\infty z$. ((x $\otimes_s$ *ket* z) $\cdot_C$ (*from-trace-class* ϱ $*_V$ (y $\otimes_s$ *ket* z))))›
by (*metis* (*mono-tags*, *lifting*) *infsumI vector-sandwich-partial-trace-has-sum*)

end

16 *Von-Neumann-Algebras* – Von Neumann algebras and the double commutant theorem

theory *Von-Neumann-Algebras*
imports *Hilbert-Space-Tensor-Product*
begin

16.1 Commutants

definition ‹*commutant* F = {x. $\forall$ y∈F. x o_{CL} y = y o_{CL} x}›

lemma *sandwich-unitary-commutant*:
fixes U :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
assumes [*simp*]: ‹*unitary* U›
shows ‹*sandwich* U ‘ *commutant* X = *commutant* (*sandwich* U ‘ X)›
proof (*rule Set.set-eqI*)
fix x
let *?comm* = ‹λa b. a o_{CL} b = b o_{CL} a›
have ‹x ∈ *sandwich* U ‘ *commutant* X ⟷ *sandwich* ($U*$) x ∈ *commutant* X›
apply (*subst inj-image-mem-iff*[*symmetric*, **where** f=‹*sandwich* ($U*$)›])
by (*auto intro*!: *inj-sandwich-isometry simp*: *image-image*
simp flip: *cblinfun-apply-cblinfun-compose sandwich-compose*)
also have ‹... ⟷ (∀ y∈X. *?comm* (*sandwich* ($U*$) x) y)›
by (*simp add*: *commutant-def*)
also have ‹... ⟷ (∀ y∈X. *?comm* x (*sandwich* U y))›
apply (*rule ball-cong*, *simp*)
apply (*simp add*: *sandwich-apply*)
by (*smt* (*verit*) *assms cblinfun-assoc-left*(1) *cblinfun-compose-id-left cblinfun-compose-id-right unitaryD1 unitaryD2*)
also have ‹... ⟷ (∀ y∈*sandwich* U ‘ X. *?comm* x y)›
by *fast*
also have ‹... ⟷ x ∈ *commutant* (*sandwich* U ‘ X)›
by (*simp add*: *commutant-def*)
finally show ‹(x ∈ ($*_V$) (*sandwich* U) ‘ *commutant* X) ⟷ (x ∈ *commutant* (($*_V$) (*sandwich* U) ‘ X))›
by −
qed

lemma *commutant-tensor1*: ‹*commutant* (*range* (λa. a $\otimes_o$ *id-cblinfun*)) = *range* (λb. *id-cblinfun* $\otimes_o$ b)›

proof (*rule Set.set-eqI*, *rule iffI*)
fix $x :: \langle('a \times 'b)\ ell2 \Rightarrow_{CL} ('a \times 'b)\ ell2\rangle$
fix $\gamma :: 'a$
assume $\langle x \in commutant\ (range\ (\lambda a.\ a \otimes_o id\text{-}cblinfun))\rangle$
then have *comm*: $\langle(a \otimes_o id\text{-}cblinfun) *_V x *_V \psi = x *_V (a \otimes_o id\text{-}cblinfun) *_V \psi\rangle$ **for** $a\ \psi$
by (*metis* (*mono-tags*, *lifting*) *commutant-def mem-Collect-eq rangeI cblinfun-apply-cblinfun-compose*)

define *op* **where** $\langle op = classical\text{-}operator\ (\lambda i.\ Some\ (\gamma,i::'b))\rangle$
have [*simp*]: $\langle classical\text{-}operator\text{-}exists\ (\lambda i.\ Some\ (\gamma,i))\rangle$
apply (*rule classical-operator-exists-inj*)
using *inj-map-def* **by** *blast*
define x' **where** $\langle x' = op* \ o_{CL}\ x\ o_{CL}\ op\rangle$
have x': $\langle cinner\ (ket\ j)\ (x' *_V ket\ l) = cinner\ (ket\ (\gamma,j))\ (x *_V ket\ (\gamma,l))\rangle$ **for** $j\ l$
by (*simp add*: x'*-def op-def classical-operator-ket cinner-adj-right*)

have $\langle cinner\ (ket\ (i,j))\ (x *_V ket\ (k,l)) = cinner\ (ket\ (i,j))\ ((id\text{-}cblinfun \otimes_o x') *_V ket\ (k,l))\rangle$
for $i\ j\ k\ l$
proof −
have $\langle cinner\ (ket\ (i,j))\ (x *_V ket\ (k,l))$
$= cinner\ ((butterfly\ (ket\ i)\ (ket\ \gamma) \otimes_o id\text{-}cblinfun) *_V ket\ (\gamma,j))\ (x *_V (butterfly\ (ket\ k)\ (ket\ \gamma) \otimes_o id\text{-}cblinfun) *_V ket\ (\gamma,l))\rangle$
by (*auto simp*: *tensor-op-ket tensor-ell2-ket*)
also have $\langle\ldots = cinner\ (ket\ (\gamma,j))\ ((butterfly\ (ket\ \gamma)\ (ket\ i) \otimes_o id\text{-}cblinfun) *_V x *_V (butterfly\ (ket\ k)\ (ket\ \gamma) \otimes_o id\text{-}cblinfun) *_V ket\ (\gamma,l))\rangle$
by (*metis* (*no-types*, *lifting*) *cinner-adj-left butterfly-adjoint id-cblinfun-adjoint tensor-op-adjoint*)
also have $\langle\ldots = cinner\ (ket\ (\gamma,j))\ (x *_V (butterfly\ (ket\ \gamma)\ (ket\ i) \otimes_o id\text{-}cblinfun\ o_{CL}\ butterfly\ (ket\ k)\ (ket\ \gamma) \otimes_o id\text{-}cblinfun) *_V ket\ (\gamma,l))\rangle$
unfolding *comm* **by** (*simp add*: *cblinfun-apply-cblinfun-compose*)
also have $\langle\ldots = cinner\ (ket\ i)\ (ket\ k) * cinner\ (ket\ (\gamma,j))\ (x *_V ket\ (\gamma,l))\rangle$
by (*simp add*: *comp-tensor-op tensor-op-ket tensor-op-scaleC-left cinner-ket tensor-ell2-ket*)
also have $\langle\ldots = cinner\ (ket\ i)\ (ket\ k) * cinner\ (ket\ j)\ (x' *_V ket\ l)\rangle$
by (*simp add*: x')
also have $\langle\ldots = cinner\ (ket\ (i,j))\ ((id\text{-}cblinfun \otimes_o x') *_V ket\ (k,l))\rangle$
apply (*simp add*: *tensor-op-ket*)
by (*simp flip*: *tensor-ell2-ket*)
finally show *?thesis* **by** −
qed
then have $\langle x = (id\text{-}cblinfun \otimes_o x')\rangle$
by (*auto intro*!: *equal-ket cinner-ket-eqI*)
then show $\langle x \in range\ (\lambda b.\ id\text{-}cblinfun \otimes_o b)\rangle$
by *auto*
next
fix $x :: \langle('a \times 'b)\ ell2 \Rightarrow_{CL} ('a \times 'b)\ ell2\rangle$
assume $\langle x \in range\ (\lambda b.\ id\text{-}cblinfun \otimes_o b)\rangle$
then obtain b **where** x: $\langle x = id\text{-}cblinfun \otimes_o b\rangle$
by *auto*
then show $\langle x \in commutant\ (range\ (\lambda a.\ a \otimes_o id\text{-}cblinfun))\rangle$
by (*auto simp*: x *commutant-def comp-tensor-op*)
qed

lemma *csubspace-commutant*[*simp*]: ‹*csubspace* (*commutant X*)›
by (*auto simp add*: *complex-vector.subspace-def commutant-def cblinfun-compose-add-right cblinfun-compose-add-left*)

lemma *closed-commutant*[*simp*]: ‹*closed* (*commutant X*)›
proof (*subst closed-sequential-limits*, *intro allI impI*, *erule conjE*)
fix *s* :: ‹*nat* ⇒ -› **and** *l*
assume *s-comm*: ‹∀ *n*. *s n* ∈ *commutant X*›
assume ‹*s* ⟶ *l*›
have ‹*l* o_{CL} *x* − *x* o_{CL} *l* = *0*› **if** ‹*x* ∈ *X*› **for** *x*
proof −
from ‹*s* ⟶ *l*›
have ‹(λ*n*. *s n* o_{CL} *x* − *x* o_{CL} *s n*) ⟶ *l* o_{CL} *x* − *x* o_{CL} *l*›
apply (*rule isCont-tendsto-compose*[*rotated*])
by (*intro continuous-intros*)
then have ‹(λ-. *0*) ⟶ *l* o_{CL} *x* − *x* o_{CL} *l*›
using *s-comm that* **by** (*auto simp add*: *commutant-def*)
then show *?thesis*
by (*simp add*: *LIMSEQ-const-iff that*)
qed
then show ‹*l* ∈ *commutant X*›
by (*simp add*: *commutant-def*)
qed

lemma *closed-csubspace-commutant*[*simp*]: ‹*closed-csubspace* (*commutant X*)›
apply (*rule closed-csubspace.intro*) **by** *simp-all*

lemma *commutant-mult*: ‹*a* o_{CL} *b* ∈ *commutant X*› **if** ‹*a* ∈ *commutant X*› **and** ‹*b* ∈ *commutant X*›
using *that*
apply (*auto simp*: *commutant-def cblinfun-compose-assoc*)
by (*simp flip*: *cblinfun-compose-assoc*)

lemma *double-commutant-grows*[*simp*]: ‹*X* ⊆ *commutant* (*commutant X*)›
by (*auto simp add*: *commutant-def*)

lemma *commutant-antimono*: ‹*X* ⊆ *Y* ⟹ *commutant X* ⊇ *commutant Y*›
by (*auto simp add*: *commutant-def*)

lemma *triple-commutant*[*simp*]: ‹*commutant* (*commutant* (*commutant X*)) = *commutant X*›
by (*auto simp*: *commutant-def*)

lemma *commutant-adj*: ‹*adj* ‘ *commutant X* = *commutant* (*adj* ‘ *X*)›
apply (*auto intro*!: *image-eqI double-adj*[*symmetric*] *simp*: *commutant-def simp flip*: *adj-cblinfun-compose*)
by (*metis adj-cblinfun-compose double-adj*)

lemma *commutant-empty*[*simp*]: ‹*commutant* {} = *UNIV*›
by (*simp add*: *commutant-def*)

lemma *commutant-weak-star-closed*[*simp*]: ‹*closedin weak-star-topology* (*commutant X*)›
proof −
have *comm-inter*: ‹*commutant X* = (⋂*x*∈*X*. *commutant* {*x*})›
by (*auto simp*: *commutant-def*)
have *comm-x*: ‹*commutant* {*x*} = (λ*y*. *x* o_{CL} *y* − *y* o_{CL} *x*) −‘ {*0*}› **for** *x* :: ‹*'a* ⇒$_{CL}$ *'a*›
by (*auto simp add*: *commutant-def vimage-def*)
have *cont*: ‹*continuous-map weak-star-topology weak-star-topology* (λ*y*. *x* o_{CL} *y* − *y* o_{CL} *x*)›
for *x* :: ‹*'a* ⇒$_{CL}$ *'a*›
apply (*rule continuous-intros*)
by (*simp-all add*: *continuous-map-left-comp-weak-star continuous-map-right-comp-weak-star*)
have ‹*closedin weak-star-topology* ((λ*y*. *x* o_{CL} *y* − *y* o_{CL} *x*) −‘ {*0*})› **for** *x* :: ‹*'a* ⇒$_{CL}$ *'a*›
using *closedin-vimage*[**where** *U*=‹*weak-star-topology*› **and** *S*=‹{*0*}› **and** *T*=*weak-star-topology*]
using *cont* **by** (*auto simp add*: *closedin-Hausdorff-singleton*)
then show *?thesis*
apply (*cases* ‹*X* = {}›)
using *closedin-topspace*[*of weak-star-topology*]
by (*auto simp add*: *comm-inter comm-x*)
qed

lemma *cspan-in-double-commutant*: ‹*cspan X* ⊆ *commutant* (*commutant X*)›
by (*simp add*: *complex-vector.span-minimal*)

lemma *weak-star-closure-in-double-commutant*: ‹*weak-star-topology closure-of X* ⊆ *commutant* (*commutant X*)›
by (*simp add*: *closure-of-minimal*)

lemma *weak-star-closure-cspan-in-double-commutant*: ‹*weak-star-topology closure-of cspan X* ⊆ *commutant* (*commutant X*)›
by (*simp add*: *closure-of-minimal cspan-in-double-commutant*)

lemma *commutant-memberI*:
assumes ‹⋀*y*. *y* ∈ *X* ⟹ *x* o_{CL} *y* = *y* o_{CL} *x*›
shows ‹*x* ∈ *commutant X*›
using *assms* **by** (*simp add*: *commutant-def*)

lemma *commutant-sot-closed*: ‹*closedin cstrong-operator-topology* (*commutant A*)›
— [2], Exercise IX.6.2
proof (*cases* ‹*A* = {}›)
case *True*
then show *?thesis*
apply *simp*
by (*metis closedin-topspace cstrong-operator-topology-topspace*)
next

case *False*
have *closed-a*: ‹*closedin cstrong-operator-topology* (*commutant* {*a*})› **for** *a* :: ‹'*a* ⇒$_{CL}$ '*a*›
proof −
have *comm-a*: ‹*commutant* {*a*} = (λ*b*. *a* *o*$_{CL}$ *b* − *b* *o*$_{CL}$ *a*) −‘ {*0*}›
by (*auto simp*: *commutant-def*)
have *closed-0*: ‹*closedin cstrong-operator-topology* {*0*}›
apply (*rule closedin-Hausdorff-singleton*)
by *simp-all*
have *cont*: ‹*continuous-map cstrong-operator-topology cstrong-operator-topology* (λ*b*. *a* *o*$_{CL}$ *b* − *b* *o*$_{CL}$ *a*)›
by (*intro continuous-intros continuous-map-left-comp-sot continuous-map-right-comp-sot*)
show *?thesis*
using *closedin-vimage*[*OF closed-0 cont*]
by (*simp add*: *comm-a*)
qed
have ∗: ‹*commutant A* = (⋂ *a*∈*A*. *commutant* {*a*})›
by (*auto simp add*: *commutant-def*)
show *?thesis*
by (*auto intro*!: *closedin-Inter simp*: ∗ *False closed-a*)
qed

lemma *commutant-tensor1'*: ‹*commutant* (*range* (λ*a*. *id-cblinfun* ⊗$_o$ *a*)) = *range* (λ*b*. *b* ⊗$_o$ *id-cblinfun*)›
proof −
have ‹*commutant* (*range* (λ*a*. *id-cblinfun* ⊗$_o$ *a*)) = *commutant* (*sandwich swap-ell2* ‘ *range* (λ*a*. *a* ⊗$_o$ *id-cblinfun*))›
by (*metis* (*no-types*, *lifting*) *image-cong range-composition swap-tensor-op-sandwich*)
also have ‹... = *sandwich swap-ell2* ‘ *commutant* (*range* (λ*a*. *a* ⊗$_o$ *id-cblinfun*))›
by (*simp add*: *sandwich-unitary-commutant*)
also have ‹... = *sandwich swap-ell2* ‘ *range* (λ*a*. *id-cblinfun* ⊗$_o$ *a*)›
by (*simp add*: *commutant-tensor1*)
also have ‹... = *range* (λ*b*. *b* ⊗$_o$ *id-cblinfun*)›
by *force*
finally show *?thesis*
by −
qed

lemma *closed-map-sot-tensor-op-id-right*:
‹*closed-map cstrong-operator-topology cstrong-operator-topology* (λ*a*. *a* ⊗$_o$ *id-cblinfun* :: ('*a* × '*b*) *ell2* ⇒$_{CL}$ ('*a* × '*b*) *ell2*)›
proof (*unfold closed-map-def*, *intro allI impI*)
fix *U* :: ‹('*a ell2* ⇒$_{CL}$ '*a ell2*) *set*›
assume *closed-U*: ‹*closedin cstrong-operator-topology U*›

have *aux1*: ‹*range f* ⊆ *X* ⟷ (∀ *x*. *f x* ∈ *X*)› **for** *f* :: ‹'*x* ⇒ '*y*› **and** *X*
by *blast*

have ‹*l* ∈ (λ*a*. *a* ⊗$_o$ *id-cblinfun*) ‘ *U*› **if** *range*: ‹*range* (λ*x*. *f x*) ⊆ (λ*a*. *a* ⊗$_o$ *id-cblinfun*) ‘ *U*›

```
  and limit: ‹limitin cstrong-operator-topology f l F› and ‹F ≠ ⊥›
 for f and l :: ‹('a × 'b) ell2 ⇒CL ('a × 'b) ell2› and F :: ‹(('a × 'b) ell2 ⇒CL ('a × 'b)
ell2) filter›
 proof −
  from range obtain f' where f'U: ‹range f' ⊆ U› and f-def: ‹f y = f' y ⊗o id-cblinfun›
for y
   apply atomize-elim
   apply (subst aux1)
   apply (rule choice2)
   by auto
  have ‹l ∈ commutant (range (λa. id-cblinfun ⊗o a))›
  proof (rule commutant-memberI)
   fix c :: ‹('a × 'b) ell2 ⇒CL ('a × 'b) ell2›
   assume ‹c ∈ range (λa. id-cblinfun ⊗o a)›
   then obtain c' where c-def: ‹c = id-cblinfun ⊗o c'›
    by blast
   from limit have 1: ‹limitin cstrong-operator-topology ((λz. z oCL c) o f) (l oCL c) F›
    apply(rule continuous-map-limit[rotated])
    by (simp add: continuous-map-right-comp-sot)
   from limit have 2: ‹limitin cstrong-operator-topology ((λz. c oCL z) o f) (c oCL l) F›
    apply(rule continuous-map-limit[rotated])
    by (simp add: continuous-map-left-comp-sot)
   have 3: ‹f x oCL c = c oCL f x› for x
    by (simp add: f-def c-def comp-tensor-op)
   from 1 2 show ‹l oCL c = c oCL l›
    unfolding 3 o-def
    by (meson hausdorff-sot limitin-Hausdorff-unique that(3))
  qed
  then have ‹l ∈ range (λa. a ⊗o id-cblinfun)›
   by (simp add: commutant-tensor1')
  then obtain l' where l-def: ‹l = l' ⊗o id-cblinfun›
   by blast
  have ‹limitin cstrong-operator-topology f' l' F›
  proof (rule limitin-cstrong-operator-topology[THEN iffD2], rule allI)
   fix ψ fix b :: 'b
   have ‹((λx. f x *V (ψ ⊗s ket b)) ⟶ l *V (ψ ⊗s ket b)) F›
    using limitin-cstrong-operator-topology that(2) by auto
   then have ‹((λx. (f' x *V ψ) ⊗s ket b) ⟶ (l' *V ψ) ⊗s ket b) F›
    by (simp add: f-def l-def tensor-op-ell2)
   then have ‹((λx. (tensor-ell2-right (ket b))* *V ((f' x *V ψ) ⊗s ket b))
            ⟶ (tensor-ell2-right (ket b))* *V ((l' *V ψ) ⊗s ket b)) F›
    apply (rule cblinfun.tendsto[rotated])
    by simp
   then show ‹((λx. f' x *V ψ) ⟶ l' *V ψ) F›
    by (simp add: tensor-ell2-right-adj-apply)
  qed
  with closed-U f'U ‹F ≠ ⊥› have ‹l' ∈ U›
   by (simp add: Misc-Tensor-Product.limitin-closedin)
  then show ‹l ∈ (λa. a ⊗o id-cblinfun) ‘ U›
```

by (*simp add*: *l-def*)
qed
then show ‹*closedin cstrong-operator-topology* ((λ*a*. *a* ⊗*o* *id-cblinfun* :: (′*a* × ′*b*) *ell2* ⇒*CL* (′*a* × ′*b*) *ell2*) ‘ *U*)›
apply (*rule-tac closedin-if-converge-inside*)
by *simp-all*
qed

lemma *id-in-commutant*[*iff*]: ‹*id-cblinfun* ∈ *commutant A*›
by (*simp add*: *commutant-memberI*)

lemma *double-commutant-hull*: ‹*commutant* (*commutant X*) = (λ*X*. *commutant* (*commutant X*) = *X*) *hull X*›
by (*smt* (*verit*) *commutant-antimono double-commutant-grows hull-unique triple-commutant*)

lemma *commutant-adj-closed*: ‹(⋀*x*. *x* ∈ *X* ⟹ *x*∗ ∈ *X*) ⟹ *x* ∈ *commutant X* ⟹ *x*∗ ∈ *commutant X*›
by (*metis* (*no-types*, *opaque-lifting*) *commutant-adj commutant-antimono double-adj imageI subset-iff*)

lemma *double-commutant-Un-left*: ‹*commutant* (*commutant* (*commutant* (*commutant X*) ∪ *Y*)) = *commutant* (*commutant* (*X* ∪ *Y*))›
apply (*simp add*: *double-commutant-hull cong*: *arg-cong*[**where** *f*=‹*Hull.hull* -›])
using *hull-Un-left* **by** *fastforce*

lemma *double-commutant-Un-right*: ‹*commutant* (*commutant* (*X* ∪ *commutant* (*commutant Y*))) = *commutant* (*commutant* (*X* ∪ *Y*))›
by (*metis Un-ac*(3) *double-commutant-Un-left*)

lemma *amplification-double-commutant-commute*:
‹*commutant* (*commutant* ((λ*a*. *a* ⊗*o* *id-cblinfun*) ‘ *X*))
= (λ*a*. *a* ⊗*o* *id-cblinfun*) ‘ *commutant* (*commutant X*)›
— [7], Corollary IV.1.5
proof −
define π :: ‹(′*a ell2* ⇒*CL* ′*a ell2*) ⇒ ((′*a* × ′*b*) *ell2* ⇒*CL* (′*a* × ′*b*) *ell2*)› **where**
‹π *a* = *a* ⊗*o* *id-cblinfun*› **for** *a*
define *U* :: ‹′*b* ⇒ ′*a ell2* ⇒*CL* (′*a* × ′*b*) *ell2*› **where** ‹*U i* = *tensor-ell2-right* (*ket i*)› **for** *i* :: ′*b*
write *commutant* (‹-′′› [120] 120)
— Notation *X*′ for *X*′
write *id-cblinfun* (‹**1**›)
have ∗: ‹(π ‘ *X*)′′ ⊆ *range* π› **for** *X*
proof (*rule subsetI*)
fix *x* **assume** *asm*: ‹*x* ∈ (π ‘ *X*)′′›
fix *t*
define *y* **where** ‹*y* = *U t*∗ *oCL x oCL U t*›
have ‹*ket* (*k*,*l*) ·*C* (*x* ∗*V* *ket* (*m*,*n*)) = *ket* (*k*,*l*) ·*C* (π *y* ∗*V* *ket* (*m*,*n*))› **for** *k l m n*
proof −

```
    have comm: ‹x o_CL (U i o_CL U j*) = (U i o_CL U j*) o_CL x› for i j
    proof −
      have ‹U i o_CL U j* = id-cblinfun ⊗_o butterfly (ket i) (ket j)›
        by (simp add: U-def tensor-ell2-right-butterfly)
      also have ‹... ∈ (π ‘ X)'›
        by (simp add: π-def commutant-def comp-tensor-op)
      finally show ?thesis
        using asm
        by (simp add: commutant-def)
    qed
    have ‹ket (k,l) •_C (x *_V ket (m,n)) = ket k •_C (U l* *_V x *_V U n *_V ket m)›
      by (simp add: cinner-adj-right U-def tensor-ell2-ket)
    also have ‹... = ket k •_C (U l* *_V x *_V U n *_V U t* *_V U t *_V ket m)›
      using U-def by fastforce
    also have ‹... = ket k •_C (U l* *_V U n *_V U t* *_V x *_V U t *_V ket m)›
      using simp-a-oCL-b'[OF comm]
      by simp
    also have ‹... = of-bool (l=n) * (ket k •_C (U t* *_V x *_V U t *_V ket m))›
      using U-def by fastforce
    also have ‹... = of-bool (l=n) * (ket k •_C (y *_V ket m))›
      using y-def by force
    also have ‹... = ket (k,l) •_C (π y *_V ket (m,n))›
      by (simp add: π-def tensor-op-ell2 flip: tensor-ell2-ket)
    finally show ?thesis
      by −
  qed
  then have ‹x = π y›
    by (metis cinner-ket-eqI equal-ket surj-pair)
  then show ‹x ∈ range π›
    by simp
qed
have **: ‹π ‘ (Y ') = (π ‘ Y)' ∩ range π› for Y
  using inj-tensor-left[of id-cblinfun]
  apply (auto simp add: commutant-def π-def comp-tensor-op
      intro!: image-eqI)
  using injD by fastforce
have 1: ‹(π ‘ X)'' ⊆ π ‘ (X '')› for X
proof −
  have ‹(π ‘ X)'' ⊆ (π ‘ X)'' ∩ range π›
    by (simp add: *)
  also have ‹... ⊆ ((π ‘ X)' ∩ range π)' ∩ range π›
    by (simp add: commutant-antimono inf.coboundedI1)
  also have ‹... = π ‘ (X '')›
    by (simp add: ** )
  finally show ?thesis
    by −
qed

have ‹x o_CL y = y o_CL x› if ‹x ∈ π ‘ (X '')› and ‹y ∈ (π ‘ X)'› for x y
```

```
  proof (intro equal-ket cinner-ket-eqI)
    fix i j :: ‹'a × 'b›
    from that obtain w where ‹w ∈ X ''› and x-def: ‹x = w ⊗o 1›
      by (auto simp: π-def)
    obtain i1 i2 where i-def: ‹i = (i1, i2)› by force
    obtain j1 j2 where j-def: ‹j = (j1, j2)› by force
    define y₀ where ‹y₀ = U i2* oCL y oCL U j2›

    have ‹y₀ ∈ X '›
    proof (rule commutant-memberI)
      fix z assume ‹z ∈ X›
      then have ‹z ⊗o 1 ∈ π ‘ X›
        by (auto simp: π-def)
      have ‹y₀ oCL z = U i2* oCL y oCL (z ⊗o 1) oCL U j2›
        by (auto intro!: equal-ket simp add: y₀-def U-def tensor-op-ell2)
      also have ‹... = U i2* oCL (z ⊗o 1) oCL y oCL U j2›
        using ‹z ⊗o 1 ∈ π ‘ X› and ‹y ∈ (π ‘ X)'›
        apply (auto simp add: commutant-def)
        by (simp add: cblinfun-compose-assoc)
      also have ‹... = z oCL y₀›
        by (auto intro!: equal-ket cinner-ket-eqI
            simp add: y₀-def U-def tensor-op-ell2 tensor-op-adjoint simp flip: cinner-adj-left)
      finally show ‹y₀ oCL z = z oCL y₀›
        by −
    qed
    have ‹ket i •C ((x oCL y) *V ket j) = ket i1 •C (U i2* *V (w ⊗o 1) *V y *V U j2 *V ket
j1)›
      by (simp add: U-def i-def j-def tensor-ell2-ket cinner-adj-right x-def)
    also have ‹... = ket i1 •C (U i2* *V (w ⊗o 1) *V (U i2 oCL U i2*) *V y *V U j2 *V ket
j1)›
      by (simp add: U-def tensor-ell2-right-butterfly tensor-op-adjoint tensor-op-ell2
          flip: cinner-adj-left)
    also have ‹... = ket i1 •C (w *V y₀ *V ket j1)›
      by (simp add: y₀-def tensor-op-adjoint tensor-op-ell2 U-def flip: cinner-adj-left)
    also have ‹... = ket i1 •C (y₀ *V w *V ket j1)›
      using ‹y₀ ∈ X '› ‹w ∈ X ''›
      apply (subst (asm) (2) commutant-def)
      using lift-cblinfun-comp(4) by force
    also have ‹... = ket i1 •C (U i2* *V y *V (U j2 oCL U j2*) *V (w ⊗o 1) *V U j2 *V ket
j1)›
      by (simp add: y₀-def tensor-op-adjoint tensor-op-ell2 U-def flip: cinner-adj-left)
    also have ‹... = ket i1 •C (U i2* *V y *V (w ⊗o 1) *V U j2 *V ket j1)›
      by (simp add: U-def tensor-ell2-right-butterfly tensor-op-adjoint tensor-op-ell2
          flip: cinner-adj-left)
    also have ‹... = ket i •C ((y oCL x) *V ket j)›
      by (simp add: U-def i-def j-def tensor-ell2-ket cinner-adj-right x-def)
    finally show ‹ket i •C ((x oCL y) *V ket j) = ket i •C ((y oCL x) *V ket j)›
      by −
  qed
```

then have *2*: ‹$(\pi\ `\ X)'' \supseteq \pi\ `\ (X\ '')$›
by (*auto intro*!: *commutant-memberI*)
from *1 2* **show** *?thesis*
by (*auto simp flip*: π*-def*)
qed

lemma *amplification-double-commutant-commute'*:
‹*commutant* (*commutant* ((λ*a*. *id-cblinfun* ⊗$_o$ *a*) ` *X*))
= (λ*a*. *id-cblinfun* ⊗$_o$ *a*) ` *commutant* (*commutant X*)›
proof −
have ‹*commutant* (*commutant* ((λ*a*. *id-cblinfun* ⊗$_o$ *a*) ` *X*))
= *commutant* (*commutant* (*sandwich swap-ell2* ` (λ*a*. *a* ⊗$_o$ *id-cblinfun*) ` *X*))›
by (*simp add*: *swap-tensor-op-sandwich image-image*)
also have ‹... = *sandwich swap-ell2* ` *commutant* (*commutant* ((λ*a*. *a* ⊗$_o$ *id-cblinfun*) ` *X*))›
by (*simp add*: *sandwich-unitary-commutant*)
also have ‹... = *sandwich swap-ell2* ` (λ*a*. *a* ⊗$_o$ *id-cblinfun*) ` *commutant* (*commutant X*)›
by (*simp add*: *amplification-double-commutant-commute*)
also have ‹... = (λ*a*. *id-cblinfun* ⊗$_o$ *a*) ` *commutant* (*commutant X*)›
by (*simp add*: *swap-tensor-op-sandwich image-image*)
finally show *?thesis*
by −
qed

lemma *commutant-cspan*: ‹*commutant* (*cspan A*) = *commutant A*›
by (*meson basic-trans-rules*(*24*) *commutant-antimono complex-vector.span-superset cspan-in-double-commutant dual-order.trans*)

lemma *double-commutant-grows'*: ‹$x \in X \Longrightarrow x \in$ *commutant* (*commutant X*)›
using *double-commutant-grows* **by** *blast*

16.2 Double commutant theorem

fun *inflation-op'* :: ‹*nat* ⇒ ('*a ell2* ⇒$_{CL}$ '*b ell2*) *list* ⇒ ('*a*×*nat*) *ell2* ⇒$_{CL}$ ('*b*×*nat*) *ell2*›
where
‹*inflation-op' n Nil* = *0*›
| ‹*inflation-op' n* (*a*#*as*) = (*a* ⊗$_o$ *butterfly* (*ket n*) (*ket n*)) + *inflation-op'* (*n*+*1*) *as*›

abbreviation ‹*inflation-op* ≡ *inflation-op' 0*›

fun *inflation-state'* :: ‹*nat* ⇒ '*a ell2 list* ⇒ ('*a*×*nat*) *ell2*› **where**
‹*inflation-state' n Nil* = *0*›
| ‹*inflation-state' n* (*a*#*as*) = (*a* ⊗$_s$ *ket n*) + *inflation-state'* (*n*+*1*) *as*›

abbreviation ‹*inflation-state* ≡ *inflation-state' 0*›

fun *inflation-space'* :: ‹*nat* ⇒ '*a ell2 ccsubspace list* ⇒ ('*a*×*nat*) *ell2 ccsubspace*› **where**
‹*inflation-space' n Nil* = *0*›
| ‹*inflation-space' n* (*S*#*Ss*) = (*S* ⊗$_S$ *ccspan* {*ket n*}) + *inflation-space'* (*n*+*1*) *Ss*›

abbreviation ‹*inflation-space* ≡ *inflation-space′ 0*›

definition *inflation-carrier* :: ‹*nat* ⇒ (*′a*×*nat*) *ell2 ccsubspace*› **where**
‹*inflation-carrier n* = *inflation-space* (*replicate n* ⊤)›

definition *inflation-op-carrier* :: ‹*nat* ⇒ ((*′a*×*nat*) *ell2* ⇒$_{CL}$ (*′b*×*nat*) *ell2*) *set*› **where**
‹*inflation-op-carrier n* = { *Proj* (*inflation-carrier n*) o_{CL} *a* o_{CL} *Proj* (*inflation-carrier n*) | *a*. *True* }›

lemma *inflation-op-compose-outside*: ‹*inflation-op′ m ops* o_{CL} (*a* ⊗$_o$ *butterfly* (*ket n*) (*ket n*)) = *0*› **if** ‹*n* < *m*›
using *that* **apply** (*induction ops arbitrary*: *m*)
by (*auto simp*: *cblinfun-compose-add-left comp-tensor-op cinner-ket*)

lemma *inflation-op-compose-outside-rev*: ‹(*a* ⊗$_o$ *butterfly* (*ket n*) (*ket n*)) o_{CL} *inflation-op′ m ops* = *0*› **if** ‹*n* < *m*›
using *that* **apply** (*induction ops arbitrary*: *m*)
by (*auto simp*: *cblinfun-compose-add-right comp-tensor-op cinner-ket*)

lemma *Proj-inflation-carrier*: ‹*Proj* (*inflation-carrier n*) = *inflation-op* (*replicate n id-cblinfun*)›
proof −
have ‹*Proj* (*inflation-space′ m* (*replicate n* ⊤)) = *inflation-op′ m* (*replicate n id-cblinfun*)› **for** *m*
proof (*induction n arbitrary*: *m*)
case *0*
then show *?case*
by *simp*
next
case (*Suc n*)
have ∗: ‹*orthogonal-spaces* ((⊤ :: *′b ell2 ccsubspace*) ⊗$_S$ *ccspan* {*ket m*}) (*inflation-space′* (*Suc m*) (*replicate n* ⊤))›
by (*auto simp add*: *orthogonal-projectors-orthogonal-spaces Suc tensor-ccsubspace-via-Proj Proj-on-own-range is-Proj-tensor-op inflation-op-compose-outside-rev butterfly-is-Proj simp flip*: *butterfly-eq-proj*)
show *?case*
apply (*simp add*: *Suc* ∗ *Proj-sup*)
by (*metis* (*no-types*, *opaque-lifting*) *Proj-is-Proj Proj-on-own-range Proj-top butterfly-eq-proj is-Proj-tensor-op norm-ket tensor-ccsubspace-via-Proj*)
qed
then show *?thesis*
by (*force simp add*: *inflation-carrier-def*)
qed

lemma *inflation-op-carrierI*:
assumes ‹*Proj* (*inflation-carrier n*) o_{CL} *a* o_{CL} *Proj* (*inflation-carrier n*) = *a*›
shows ‹*a* ∈ *inflation-op-carrier n*›
using *assms* **by** (*auto intro*!: *exI*[*of* - *a*] *simp add*: *inflation-op-carrier-def*)

lemma *inflation-op-compose*: ‹*inflation-op' n ops1* o_{CL} *inflation-op' n ops2 = inflation-op' n (map2 cblinfun-compose ops1 ops2)*›
proof (*induction ops2 arbitrary*: *ops1 n*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons op ops2*)
note *IH = this*
fix *ops1* :: ‹('*c ell2* ⇒$_{CL}$ '*b ell2*) *list*›
show ‹*inflation-op' n ops1* o_{CL} *inflation-op' n* (*op # ops2*) =
inflation-op' n (*map2* (o_{CL}) *ops1* (*op # ops2*))›
proof (*cases ops1*)
case *Nil*
then show *?thesis*
by *simp*
next
case (*Cons a list*)
then show *?thesis*
by (*simp add*: *cblinfun-compose-add-right cblinfun-compose-add-left tensor-op-ell2 inflation-op-compose-outside comp-tensor-op inflation-op-compose-outside-rev flip*: *IH*)
qed
qed

lemma *inflation-op-in-carrier*: ‹*inflation-op ops* ∈ *inflation-op-carrier n*› **if** ‹*length ops* ≤ *n*›
apply (*rule inflation-op-carrierI*)
using *that*
by (*simp add*: *Proj-inflation-carrier inflation-op-carrier-def inflation-op-compose zip-replicate1 zip-replicate2 o-def*)

lemma *inflation-op'-apply-tensor-outside*: ‹*n* < *m* ⟹ *inflation-op' m as* ∗$_V$ (*v* ⊗$_s$ *ket n*) = *0*›
apply (*induction as arbitrary*: *m*)
by (*auto simp*: *cblinfun.add-left tensor-op-ell2 cinner-ket*)

lemma *inflation-op'-compose-tensor-outside*: ‹*n* < *m* ⟹ *inflation-op' m as* o_{CL} *tensor-ell2-right* (*ket n*) = *0*›
apply (*rule cblinfun-eqI*)
by (*simp add*: *inflation-op'-apply-tensor-outside*)

lemma *inflation-state'-apply-tensor-outside*: ‹*n* < *m* ⟹ (*a* ⊗$_o$ *butterfly ψ* (*ket n*)) ∗$_V$ *inflation-state' m vs* = *0*›
apply (*induction vs arbitrary*: *m*)
by (*auto simp*: *cblinfun.add-right tensor-op-ell2 cinner-ket*)

lemma *inflation-op-apply-inflation-state*: ‹*inflation-op' n ops* ∗$_V$ *inflation-state' n vecs = inflation-state' n* (*map2 cblinfun-apply ops vecs*)›
proof (*induction vecs arbitrary*: *ops n*)
case *Nil*
then show *?case* **by** *simp*

next
case (*Cons v vecs*)
note *IH* = *this*
fix *ops* :: ‹(*'b ell2* ⇒$_{CL}$ *'a ell2*) *list*›
show ‹*inflation-op' n ops* ∗$_V$ *inflation-state' n* (*v* # *vecs*) =
inflation-state' n (*map2* (∗$_V$) *ops* (*v* # *vecs*))›
proof (*cases ops*)
case *Nil*
then show *?thesis*
by *simp*
next
case (*Cons a list*)
then show *?thesis*
by (*simp add*: *cblinfun.add-right cblinfun.add-left tensor-op-ell2*
inflation-op'-apply-tensor-outside inflation-state'-apply-tensor-outside
flip: *IH*)
qed
qed

lemma *inflation-state-in-carrier*: ‹*inflation-state vecs* ∈ *space-as-set* (*inflation-carrier n*)› **if** ‹*length vecs* + *m* ≤ *n*›
apply (*rule space-as-setI-via-Proj*)
using *that*
by (*simp add*: *Proj-inflation-carrier inflation-op-apply-inflation-state zip-replicate1 o-def*)

lemma *inflation-op'-apply-tensor-outside'*: ‹*n* ≥ *length as* + *m* ⟹ *inflation-op' m as* ∗$_V$ (*v* ⊗$_s$ *ket n*) = *0*›
apply (*induction as arbitrary*: *m*)
by (*auto simp*: *cblinfun.add-left tensor-op-ell2 cinner-ket*)

lemma *Proj-inflation-carrier-outside*: ‹*Proj* (*inflation-carrier n*) ∗$_V$ (*ψ* ⊗$_s$ *ket i*) = *0*› **if** ‹*i* ≥ *n*›
by (*simp add*: *Proj-inflation-carrier inflation-op'-apply-tensor-outside' that*)

lemma *inflation-state'-is-orthogonal-outside*: ‹*n* < *m* ⟹ *is-orthogonal* (*a* ⊗$_s$ *ket n*) (*inflation-state' m vs*)›
apply (*induction vs arbitrary*: *m*)
by (*auto simp*: *cinner-add-right*)

lemma *inflation-op-adj*: ‹(*inflation-op' n ops*)∗ = *inflation-op' n* (*map adj ops*)›
apply (*induction ops arbitrary*: *n*)
by (*simp-all add*: *adj-plus tensor-op-adjoint*)

lemma *inflation-state0*:
assumes ‹⋀*v*. *v* ∈ *set f* ⟹ *v* = *0*›
shows ‹*inflation-state' n f* = *0*›
using *assms* **apply** (*induction f arbitrary*: *n*)
apply *simp*

using *tensor-ell2-0-left* **by** *force*

lemma *inflation-state-plus*:
assumes ‹*length f = length g*›
shows ‹*inflation-state' n f + inflation-state' n g = inflation-state' n (map2 plus f g)*›
using *assms* **apply** (*induction f g arbitrary*: *n rule*: *list-induct2*)
by (*auto simp*: *algebra-simps tensor-ell2-add1*)

lemma *inflation-state-minus*:
assumes ‹*length f = length g*›
shows ‹*inflation-state' n f − inflation-state' n g = inflation-state' n (map2 minus f g)*›
using *assms* **apply** (*induction f g arbitrary*: *n rule*: *list-induct2*)
by (*auto simp*: *algebra-simps tensor-ell2-diff1*)

lemma *inflation-state-scaleC*:
shows ‹*c* $*_C$ *inflation-state' n f = inflation-state' n (map (scaleC c) f)*›
apply (*induction f arbitrary*: *n*)
by (*auto simp*: *algebra-simps tensor-ell2-scaleC1*)

lemma *inflation-op-compose-tensor-ell2-right*:
assumes ‹*i ≥ n*› **and** ‹*i < n + length f*›
shows ‹*inflation-op' n f* o_{CL} *tensor-ell2-right* (*ket i*) = *tensor-ell2-right* (*ket i*) o_{CL} (*f*!(*i*−*n*))›
proof (*insert assms*, *induction f arbitrary*: *n*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a f*)
show *?case*
proof (*cases* ‹*i = n*›)
case *True*
have ‹*a* $\otimes_o$ *butterfly* (*ket n*) (*ket n*) o_{CL} *tensor-ell2-right* (*ket n*) = *tensor-ell2-right* (*ket n*) o_{CL} *a*›
apply (*rule cblinfun-eqI*)
by (*simp add*: *tensor-op-ell2 cinner-ket*)
with *True* **show** *?thesis*
by (*simp add*: *cblinfun-compose-add-left inflation-op'-compose-tensor-outside*)
next
case *False*
with *Cons.prems* **have** *1*: ‹*Suc n ≤ i*›
by *presburger*
have *2*: ‹*a* $\otimes_o$ *butterfly* (*ket n*) (*ket n*) o_{CL} *tensor-ell2-right* (*ket i*) = *0*›
apply (*rule cblinfun-eqI*)
using *False* **by** (*simp add*: *tensor-op-ell2 cinner-ket*)
show *?thesis*
using *Cons.prems 1*
by (*simp add*: *cblinfun-compose-add-left Cons.IH*[**where** *n*=‹*Suc n*›] *2*)
qed
qed

lemma *inflation-op-apply*:
assumes ‹$i \geq n$› **and** ‹$i < n + \mathit{length}\ f$›
shows ‹$\mathit{inflation\text{-}op'}\ n\ f *_V (\psi \otimes_s \mathit{ket}\ i) = (f!(i-n) *_V \psi) \otimes_s \mathit{ket}\ i$›
by (*simp add*: *inflation-op-compose-tensor-ell2-right assms*
flip: *tensor-ell2-right-apply cblinfun-apply-cblinfun-compose*)

lemma *norm-inflation-state*:
‹$\mathit{norm}\ (\mathit{inflation\text{-}state'}\ n\ f) = \mathit{sqrt}\ (\sum v{\leftarrow}f.\ (\mathit{norm}\ v)^2)$›
proof −
have ‹$(\mathit{norm}\ (\mathit{inflation\text{-}state'}\ n\ f))^2 = (\sum v{\leftarrow}f.\ (\mathit{norm}\ v)^2)$›
proof (*induction f arbitrary*: *n*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons v f*)
have ‹$(\mathit{norm}\ (\mathit{inflation\text{-}state'}\ n\ (v\ \#\ f)))^2 = (\mathit{norm}\ (v \otimes_s \mathit{ket}\ n + \mathit{inflation\text{-}state'}\ (\mathit{Suc}\ n)\ f))^2$›
by *simp*
also have ‹$\ldots = (\mathit{norm}\ (v \otimes_s \mathit{ket}\ n))^2 + (\mathit{norm}\ (\mathit{inflation\text{-}state'}\ (\mathit{Suc}\ n)\ f))^2$›
apply (*rule pythagorean-theorem*)
apply (*rule inflation-state'-is-orthogonal-outside*)
by *simp*
also have ‹$\ldots = (\mathit{norm}\ (v \otimes_s \mathit{ket}\ n))^2 + (\sum v{\leftarrow}f.\ (\mathit{norm}\ v)^2)$›
by (*simp add*: *Cons.IH*)
also have ‹$\ldots = (\mathit{norm}\ v)^2 + (\sum v{\leftarrow}f.\ (\mathit{norm}\ v)^2)$›
by (*simp add*: *norm-tensor-ell2*)
also have ‹$\ldots = (\sum v{\leftarrow}v\#f.\ (\mathit{norm}\ v)^2)$›
by *simp*
finally show *?case*
by −
qed
then show *?thesis*
by (*simp add*: *real-sqrt-unique*)
qed

lemma *cstrong-operator-topology-in-closure-algebraicI*:
— [2], Proposition IX.5.3
assumes *space*: ‹*csubspace A*›
assumes *mult*: ‹$\bigwedge a\ a'.\ a \in A \Longrightarrow a' \in A \Longrightarrow a\ o_{CL}\ a' \in A$›
assumes *one*: ‹*id-cblinfun* $\in A$›
assumes *main*: ‹$\bigwedge n\ S.\ S \leq \mathit{inflation\text{-}carrier}\ n \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow \mathit{inflation\text{-}op}\ (\mathit{replicate}\ n\ a) *_S S \leq S) \Longrightarrow$
$\mathit{inflation\text{-}op}\ (\mathit{replicate}\ n\ b) *_S S \leq S$›
shows ‹$b \in$ *cstrong-operator-topology closure-of A*›
proof (*rule cstrong-operator-topology-in-closureI*)
fix $F :: \langle 'a\ \mathit{ell2}\ \mathit{set}\rangle$ **and** $\varepsilon :: \mathit{real}$
assume ‹*finite F*› **and** ‹$\varepsilon > 0$›

obtain f **where** ‹*set f = F*› **and** ‹*distinct f*›
using ‹*finite F*› *finite-distinct-list* **by** *blast*
define *n M' M* **where** ‹*n = length f*›
and ‹*M' = ((λa. inflation-state (map (cblinfun-apply a) f)) ‘ A)*›
and ‹*M = ccspan M'*›
have *M-carrier*: ‹*M ≤ inflation-carrier n*›
proof −
have ‹*M' ⊆ space-as-set (inflation-carrier n)*›
by (*auto intro*!: *inflation-state-in-carrier simp add*: *M'-def n-def*)
then show *?thesis*
by (*simp add*: *M-def ccspan-leqI*)
qed

have ‹*inflation-op (replicate n a) ∗S M ≤ M*› **if** ‹*a ∈ A*› **for** *a*
proof (*unfold M-def*, *rule cblinfun-image-ccspan-leqI*)
fix *v* **assume** ‹*v ∈ M'*›
then obtain *a'* **where** ‹*a' ∈ A*› **and** *v-def*: ‹*v = inflation-state (map (cblinfun-apply a') f)*›
using *M'-def* **by** *blast*
then have ‹*inflation-op (replicate n a) ∗V v = inflation-state (map ((∗V) (a oCL a')) f)*›
by (*simp add*: *v-def n-def inflation-op-apply-inflation-state map2-map-map*
flip: *cblinfun-apply-cblinfun-compose map-replicate-const*)
also have ‹*... ∈ M'*›
using *M'-def* ‹*a' ∈ A*› ‹*a ∈ A*› *mult*
by *simp*
also have ‹*... ⊆ space-as-set (ccspan M')*›
by (*simp add*: *ccspan-superset*)
finally show ‹*inflation-op (replicate n a) ∗V v ∈ space-as-set (ccspan M')*›
by −
qed
then have *b-invariant*: ‹*inflation-op (replicate n b) ∗S M ≤ M*›
using *M-carrier* **by** (*simp add*: *main*)
have *f-M*: ‹*inflation-state f ∈ space-as-set M*›
proof −
have ‹*inflation-state f = inflation-state (map (cblinfun-apply id-cblinfun) f)*›
by *simp*
also have ‹*... ∈ M'*›
using *M'-def one* **by** *blast*
also have ‹*... ⊆ space-as-set M*›
by (*simp add*: *M-def ccspan-superset*)
finally show *?thesis*
by −
qed
have ‹*csubspace M'*›
proof (*rule complex-vector.subspaceI*)
fix *c x y*
show ‹*0 ∈ M'*›
apply (*auto intro*!: *image-eqI*[**where** *x=0*] *simp add*: *M'-def*)
apply (*subst inflation-state0*)
by (*auto simp add*: *space complex-vector.subspace-0*)

```
    show ‹x ∈ M' ⟹ y ∈ M' ⟹ x + y ∈ M'›
      by (auto intro!: image-eqI[where x=‹- + -›]
          simp add: M'-def inflation-state-plus map2-map-map
          cblinfun.add-left[abs-def] space complex-vector.subspace-add)
    show ‹c *C x ∈ M' › if ‹x ∈ M'›
    proof −
      from that
      obtain a where ‹a ∈ A› and ‹x = inflation-state (map ((*V) a) f)›
        by (auto simp add: M'-def)
      then have ‹c *C x = inflation-state (map ((*V) (c *C a)) f)›
        by (simp add: inflation-state-scaleC o-def scaleC-cblinfun.rep-eq)
      moreover have ‹c *C a ∈ A›
        by (simp add: ‹a ∈ A› space complex-vector.subspace-scale)
      ultimately show ?thesis
        unfolding M'-def
        by (rule image-eqI)
    qed
  qed
  then have M-closure-M': ‹space-as-set M = closure M'›
    by (metis M-def ccspan.rep-eq complex-vector.span-eq-iff)
  have ‹inflation-state (map (cblinfun-apply b) f) ∈ space-as-set M›
  proof −
    have ‹map2 (*V) (replicate n b) f = map ((*V) b) f›
      using map2-map-map[where h=cblinfun-apply and g=id and f=‹λ-. b› and xs=f]
      by (simp add: n-def flip: map-replicate-const)
    then have ‹inflation-state (map (cblinfun-apply b) f) = inflation-op (replicate n b) *V
inflation-state f›
      by (simp add: inflation-op-apply-inflation-state)
    also have ‹... ∈ space-as-set (inflation-op (replicate n b) *S M)›
      by (simp add: f-M cblinfun-apply-in-image')
    also have ‹... ⊆ space-as-set M›
      using b-invariant less-eq-ccsubspace.rep-eq by blast
    finally show ?thesis
      by −
  qed
  then obtain m where ‹m ∈ M'› and m-close: ‹norm (m − inflation-state (map (cblinfun-apply
b) f)) ≤ ε›
    apply atomize-elim
    apply (simp add: M-closure-M' closure-approachable dist-norm)
    using ‹ε > 0› by fastforce
  from ‹m ∈ M'›
  obtain a where ‹a ∈ A› and m-def: ‹m = inflation-state (map (cblinfun-apply a) f)›
    by (auto simp add: M'-def)
  have ‹(∑ v←f. (norm ((a − b) *V v))²) ≤ ε²›
  proof −
    have ‹(∑ v←f. (norm ((a − b) *V v))²) = (norm (inflation-state (map (cblinfun-apply (a
− b)) f)))²›
      apply (simp add: norm-inflation-state o-def)
      apply (subst real-sqrt-pow2)
```

apply (*rule sum-list-nonneg*)
by (*auto simp*: *sum-list-nonneg*)
also have ‹... = (*norm* (*m* − *inflation-state* (*map* (*cblinfun-apply b*) *f*)))^2›
by (*simp add*: *m-def inflation-state-minus map2-map-map cblinfun.diff-left*[*abs-def*])
also have ‹... ≤ ε^2›
by (*simp add*: *m-close power-mono*)
finally show *?thesis*
by −
qed
then have ‹(*norm* ((*a* − *b*) $*_V$ *v*))^2 ≤ ε^2› **if** ‹*v* ∈ *F*› **for** *v*
using *that* **apply** (*simp flip*: *sum.distinct-set-conv-list add*: ‹*distinct f*›)
by (*smt* (*verit*) ‹*finite F*› ‹*set f* = *F*› *sum-nonneg-leq-bound zero-le-power2*)
then show ‹∃ *a*∈*A*. ∀ *f*∈*F*. *norm* ((*b* − *a*) $*_V$ *f*) ≤ ε›
using ‹*0* < ε› ‹*a* ∈ *A*›
by (*metis cblinfun.real.diff-left norm-minus-commute power2-le-imp-le power-eq-0-iff power-zero-numeral realpow-pos-nth-unique zero-compare-simps*(*12*))
qed

lemma *commutant-inflation*:
— One direction of [2], Proposition IX.6.2.
fixes *n*
defines ‹⋀*X*. *commutant′ X* ≡ *commutant X* ∩ *inflation-op-carrier n*›
shows ‹(λ*a*. *inflation-op* (*replicate n a*)) ‘ *commutant* (*commutant A*)
⊆ *commutant′* (*commutant′* ((λ*a*. *inflation-op* (*replicate n a*)) ‘ *A*))›
proof (*unfold commutant′-def*, *rule subsetI*, *rule IntI*)
fix *b*
assume ‹*b* ∈ (λ*a*. *inflation-op* (*replicate n a*)) ‘ *commutant* (*commutant A*)›
then obtain *b0* **where** *b-def*: ‹*b* = *inflation-op* (*replicate n b0*)› **and** *b0-A″*: ‹*b0* ∈ *commutant* (*commutant A*)›
by *auto*
show ‹*b* ∈ *inflation-op-carrier n*›
by (*simp add*: *b-def inflation-op-in-carrier*)
show ‹*b* ∈ *commutant* (*commutant* ((λ*a*. *inflation-op* (*replicate n a*)) ‘ *A*) ∩ *inflation-op-carrier n*)›
proof (*rule commutant-memberI*)
fix *c*
assume ‹*c* ∈ *commutant* ((λ*a*. *inflation-op* (*replicate n a*)) ‘ *A*) ∩ *inflation-op-carrier n*›
then have *c-comm*: ‹*c* ∈ *commutant* ((λ*a*. *inflation-op* (*replicate n a*)) ‘ *A*)›
and *c-carr*: ‹*c* ∈ *inflation-op-carrier n*›
by *auto*
define *c′* **where** ‹*c′ i j* = (*tensor-ell2-right* (*ket i*))* o_{CL} *c* o_{CL} *tensor-ell2-right* (*ket j*)› **for** *i j*
have ‹*c′ i j* o_{CL} *a* = *a* o_{CL} *c′ i j*› **if** ‹*a* ∈ *A*› **and** ‹*i* < *n*› **and** ‹*j* < *n*› **for** *a i j*
proof −
from *c-comm* **have** ‹*c* o_{CL} *inflation-op* (*replicate n a*) = *inflation-op* (*replicate n a*) o_{CL} *c*›
using *that* **by** (*auto simp*: *commutant-def*)
then have ‹(*tensor-ell2-right* (*ket i*))* o_{CL} *c* o_{CL} (*inflation-op* (*replicate n a*) o_{CL} *tensor-ell2-right* (*ket j*))

```
            = (inflation-op (replicate n (a*)) o_CL (tensor-ell2-right (ket i)))* o_CL c o_CL
tensor-ell2-right (ket j)›
      apply (simp add: inflation-op-adj)
      by (metis (no-types, lifting) lift-cblinfun-comp(2))
    then show ?thesis
      apply (subst (asm) inflation-op-compose-tensor-ell2-right)
        apply (simp, simp add: that)
      apply (subst (asm) inflation-op-compose-tensor-ell2-right)
        apply (simp, simp add: that)
      by (simp add: that c'-def cblinfun-compose-assoc)
  qed
  then have ‹c' i j ∈ commutant A› if ‹i < n› and ‹j < n› for i j
    using that by (simp add: commutant-memberI)
  with b0-A'' have b0-c': ‹b0 o_CL c' i j = c' i j o_CL b0› if ‹i < n› and ‹j < n› for i j
    using that by (simp add: commutant-def)

    from c-carr obtain c'' where c'': ‹c = Proj (inflation-carrier n) o_CL c'' o_CL Proj
(inflation-carrier n)›
      by (auto simp add: inflation-op-carrier-def)

  have c0: ‹c *_V (ψ ⊗_s ket i) = 0› if ‹i ≥ n› for i ψ
    using that by (simp add: c'' Proj-inflation-carrier-outside)
  have cadj0: ‹c* *_V (ψ ⊗_s ket j) = 0› if ‹j ≥ n› for j ψ
    using that by (simp add: c'' adj-Proj Proj-inflation-carrier-outside)

  have ‹inflation-op (replicate n b0) o_CL c = c o_CL inflation-op (replicate n b0)›
  proof (rule equal-ket, rule cinner-ket-eqI)
    fix ii jj
    obtain i' j' :: 'a and i j :: nat where ii-def: ‹ii = (i',i)› and jj-def: ‹jj = (j',j)›
      by force
    show ‹ket ii •_C ((inflation-op (replicate n b0) o_CL c) *_V ket jj) =
            ket ii •_C ((c o_CL inflation-op (replicate n b0)) *_V ket jj)›
    proof (cases ‹i < n ∧ j < n›)
      case True
       have ‹ket ii •_C ((inflation-op (replicate n b0) o_CL c) *_V ket jj) = ((b0* *_V ket i') ⊗_s
ket i) •_C (c *_V ket j' ⊗_s ket j)›
          using True by (simp add: ii-def jj-def inflation-op-adj inflation-op-apply flip: ten-
sor-ell2-inner-prod
            flip: tensor-ell2-ket cinner-adj-left[where G=‹inflation-op -›])
      also have ‹... = (ket i' ⊗_s ket i) •_C (c *_V (b0 *_V ket j') ⊗_s ket j)›
        using b0-c' apply (simp add: c'-def flip: tensor-ell2-right-apply cinner-adj-right)
        by (metis (no-types, lifting) True simp-a-oCL-b')
      also have ‹... = ket ii •_C ((c o_CL inflation-op (replicate n b0)) *_V ket jj)›
     by (simp add: True ii-def jj-def inflation-op-adj inflation-op-apply flip: tensor-ell2-inner-prod
          flip: tensor-ell2-ket cinner-adj-left[where G=‹inflation-op -›])
      finally show ?thesis
        by −
    next
      case False
```

then show *?thesis*
apply (*auto simp add*: *ii-def jj-def inflation-op-adj c0 inflation-op'-apply-tensor-outside'*
simp flip: *tensor-ell2-ket cinner-adj-left*[**where** *G*=‹*inflation-op* -›])
by (*simp add*: *cadj0 flip*: *cinner-adj-left*[**where** *G*=*c*])
qed
qed
then show ‹*b* o_{CL} *c* = *c* o_{CL} *b*›
by (*simp add*: *b-def*)
qed
qed

lemma *double-commutant-theorem-aux*:
— Basically the double commutant theorem, except that we restricted to spaces of the form *'a ell2*
— [2], Proposition IX.6.4
fixes *A* :: ‹('*a ell2* ⇒$_{CL}$ '*a ell2*) *set*›
assumes ‹*csubspace A*›
assumes ‹⋀*a a'*. *a* ∈ *A* ⟹ *a'* ∈ *A* ⟹ *a* o_{CL} *a'* ∈ *A*›
assumes ‹*id-cblinfun* ∈ *A*›
assumes ‹⋀*a*. *a* ∈ *A* ⟹ *a*∗ ∈ *A*›
shows ‹*commutant* (*commutant A*) = *cstrong-operator-topology closure-of A*›
proof (*intro Set.set-eqI iffI*)
show ‹*x* ∈ *commutant* (*commutant A*)› **if** ‹*x* ∈ *cstrong-operator-topology closure-of A*› **for** *x*
using *closure-of-minimal commutant-sot-closed double-commutant-grows that* **by** *blast*
next
show ‹*b* ∈ *cstrong-operator-topology closure-of A*› **if** *b-A''*: ‹*b* ∈ *commutant* (*commutant A*)›
for *b*
proof (*rule cstrong-operator-topology-in-closure-algebraicI*)
show ‹*csubspace A*› **and** ‹*a* ∈ *A* ⟹ *a'* ∈ *A* ⟹ *a* o_{CL} *a'* ∈ *A*› **and** ‹*id-cblinfun* ∈ *A*› **for**
a a'
using *assms* **by** *auto*
fix *n M*
assume *asm*: ‹*a* ∈ *A* ⟹ *inflation-op* (*replicate n a*) ∗$_S$ *M* ≤ *M*› **for** *a*
assume *M-carrier*: ‹*M* ≤ *inflation-carrier n*›
define *commutant'* **where** ‹*commutant' X* = *commutant X* ∩ *inflation-op-carrier n*› **for** *X*
:: ‹((*'a* × *nat*) *ell2* ⇒$_{CL}$ (*'a* × *nat*) *ell2*) *set*›
define *An* **where** ‹*An* = (λ*a*. *inflation-op* (*replicate n a*)) ‘ *A*›
have ∗: ‹*Proj M* o_{CL} (*inflation-op* (*replicate n a*) o_{CL} *Proj M*) = *inflation-op* (*replicate n*
a) o_{CL} *Proj M*› **if** ‹*a* ∈ *A*› **for** *a*
apply (*rule Proj-compose-cancelI*)
using *asm that* **by** (*simp add*: *cblinfun-compose-image*)
have ‹*Proj M* o_{CL} *inflation-op* (*replicate n a*) = *inflation-op* (*replicate n a*) o_{CL} *Proj M*› **if**
‹*a* ∈ *A*› **for** *a*
proof −
have ‹*Proj M* o_{CL} *inflation-op* (*replicate n a*) = (*inflation-op* (*replicate n* (*a*∗)) o_{CL} *Proj*
M)∗›
by (*simp add*: *inflation-op-adj adj-Proj*)
also have ‹. . . = (*Proj M* o_{CL} *inflation-op* (*replicate n* (*a*∗)) o_{CL} *Proj M*)∗›
apply (*subst* ∗[*symmetric*])

```
      by (simp-all add: that assms flip: cblinfun-compose-assoc)
    also have ‹... = Proj M o_CL inflation-op (replicate n a) o_CL Proj M›
      by (simp add: inflation-op-adj adj-Proj cblinfun-compose-assoc)
    also have ‹... = inflation-op (replicate n a) o_CL Proj M›
      apply (subst *[symmetric])
      by (simp-all add: that flip: cblinfun-compose-assoc)
    finally show ?thesis
      by -
  qed
  then have ‹Proj M ∈ commutant' An›
    using M-carrier
    apply (auto intro!: inflation-op-carrierI simp add: An-def commutant-def commutant'-def)
    by (metis Proj-compose-cancelI Proj-range adj-Proj adj-cblinfun-compose)
  from b-A'' have ‹inflation-op (replicate n b) ∈ commutant' (commutant' An)›
    using commutant-inflation[of n A, folded commutant'-def]
    by (auto simp add: An-def commutant'-def)
  with ‹Proj M ∈ commutant' An›
  have *: ‹inflation-op (replicate n b) o_CL Proj M = Proj M o_CL inflation-op (replicate n b)›
    by (simp add: commutant-def commutant'-def)
  show ‹inflation-op (replicate n b) *_S M ≤ M›
  proof -
    have ‹inflation-op (replicate n b) *_S M = (inflation-op (replicate n b) o_CL Proj M) *_S ⊤›
      by (metis lift-cblinfun-comp(3) Proj-range)
    also have ‹... = (Proj M o_CL inflation-op (replicate n b)) *_S ⊤›
      by (simp add: * )
    also have ‹... ≤ M›
      by (metis lift-cblinfun-comp(3) Proj-image-leq)
    finally show ?thesis
      by -
  qed
qed
qed

lemma double-commutant-theorem-aux2:
  — Basically the double commutant theorem, except that we restricted to spaces of typeclass not-singleton
  — [2], Proposition IX.6.4
  fixes A :: ‹('a::{chilbert-space,not-singleton} ⇒_CL 'a) set›
  assumes subspace: ‹csubspace A›
  assumes mult: ‹⋀a a'. a ∈ A ⟹ a' ∈ A ⟹ a o_CL a' ∈ A›
  assumes id: ‹id-cblinfun ∈ A›
  assumes adj: ‹⋀a. a ∈ A ⟹ a* ∈ A›
  shows ‹commutant (commutant A) = cstrong-operator-topology closure-of A›
proof -
  define A' :: ‹('a chilbert2ell2 ell2 ⇒_CL 'a chilbert2ell2 ell2) set›
    where ‹A' = sandwich (ell2-to-hilbert*) ` A›
  have subspace: ‹csubspace A'›
    using subspace by (auto intro!: complex-vector.linear-subspace-image simp: A'-def)
  have mult: ‹⋀a a'. a ∈ A' ⟹ a' ∈ A' ⟹ a o_CL a' ∈ A'›
```

```
    using mult by (auto simp add: A'-def sandwich-arg-compose unitary-ell2-to-hilbert)
  have id: ‹id-cblinfun ∈ A'›
   using id by (auto intro!: image-eqI simp add: A'-def sandwich-isometry-id unitary-ell2-to-hilbert)
  have adj: ‹⋀a. a ∈ A' ⟹ a* ∈ A'›
    using adj by (auto intro!: image-eqI simp: A'-def simp flip: sandwich-apply-adj)
  have homeo: ‹homeomorphic-map cstrong-operator-topology cstrong-operator-topology
     ((*V) (sandwich ell2-to-hilbert))›
   by (auto intro!: continuous-intros homeomorphic-maps-imp-map[where g=‹sandwich (ell2-to-hilbert*)›]
       simp: homeomorphic-maps-def unitary-ell2-to-hilbert
       simp flip: cblinfun-apply-cblinfun-compose sandwich-compose)
  have ‹commutant (commutant A') = cstrong-operator-topology closure-of A'›
    using subspace mult id adj by (rule double-commutant-theorem-aux)
  then have ‹sandwich ell2-to-hilbert ‘ commutant (commutant A') = sandwich ell2-to-hilbert ‘
(cstrong-operator-topology closure-of A')›
    by simp
  then show ?thesis
   by (simp add: A'-def unitary-ell2-to-hilbert sandwich-unitary-commutant image-image homeo
       flip: cblinfun-apply-cblinfun-compose sandwich-compose
       homeomorphic-map-closure-of[where Y=cstrong-operator-topology])
qed

lemma double-commutant-theorem:
  — [2], Proposition IX.6.4
  fixes A :: ‹('a::{chilbert-space} ⇒CL 'a) set›
  assumes subspace: ‹csubspace A›
  assumes mult: ‹⋀a a'. a ∈ A ⟹ a' ∈ A ⟹ a oCL a' ∈ A›
  assumes id: ‹id-cblinfun ∈ A›
  assumes adj: ‹⋀a. a ∈ A ⟹ a* ∈ A›
  shows ‹commutant (commutant A) = cstrong-operator-topology closure-of A›
proof (cases ‹UNIV = {0::'a}›)
  case True
  then have ‹(x :: 'a) = 0› for x
    by auto
  then have UNIV-0: ‹UNIV = {0 :: 'a⇒CL'a}›
    by (auto intro!: cblinfun-eqI)
  have ‹0 ∈ commutant (commutant A)›
    using complex-vector.subspace-0 csubspace-commutant by blast
  then have 1: ‹commutant (commutant A) = UNIV›
    using UNIV-0
    by force
  have ‹0 ∈ A›
    by (simp add: assms(1) complex-vector.subspace-0)
  then have ‹0 ∈ cstrong-operator-topology closure-of A›
    by (simp add: assms(1) complex-vector.subspace-0)
  then have 2: ‹cstrong-operator-topology closure-of A = UNIV›
    using UNIV-0
    by force
  from 1 2 show ?thesis
    by simp
```

next
case *False*
note *aux2* = *double-commutant-theorem-aux2*[**where** *'a*=‹*'z*::{*chilbert-space*,*not-singleton*}›, *rule-format*, *internalize-sort* ‹*'z*::{*chilbert-space*,*not-singleton*}›]
have *hilbert*: ‹*class.chilbert-space* (∗$_R$) (∗$_C$) (+) (*0*::*'a*) (−) *uminus dist norm sgn uniformity open* (·$_C$)›
by (*rule chilbert-space-class.chilbert-space-axioms*)
from *False*
have *not-singleton*: ‹*class.not-singleton TYPE*(*'a*)›
by (*rule class-not-singletonI-monoid-add*)
show *?thesis*
apply (*rule aux2*)
using *assms hilbert not-singleton* **by** *auto*
qed

hide-fact *double-commutant-theorem-aux double-commutant-theorem-aux2*

lemma *double-commutant-theorem-span*:
fixes *A* :: ‹(*'a*::{*chilbert-space*} ⇒$_{CL}$ *'a*) *set*›
assumes *mult*: ‹⋀*a a'*. *a* ∈ *A* ⟹ *a'* ∈ *A* ⟹ *a* *o*$_{CL}$ *a'* ∈ *A*›
assumes *id*: ‹*id-cblinfun* ∈ *A*›
assumes *adj*: ‹⋀*a*. *a* ∈ *A* ⟹ *a*∗ ∈ *A*›
shows ‹*commutant* (*commutant A*) = *cstrong-operator-topology closure-of* (*cspan A*)›
proof −
have ‹*commutant* (*commutant A*) = *commutant* (*commutant* (*cspan A*))›
by (*simp add*: *commutant-cspan*)
also have ‹. . . = *cstrong-operator-topology closure-of* (*cspan A*)›
apply (*rule double-commutant-theorem*)
using *assms*
apply (*auto simp*: *cspan-compose-closed cspan-adj-closed*)
using *complex-vector.span-clauses*(*1*) **by** *blast*
finally show *?thesis*
by −
qed

16.3 Von Neumann Algebras

definition *one-algebra* :: ‹(*'a* ⇒$_{CL}$ *'a*::*chilbert-space*) *set*› **where**
‹*one-algebra* = *range* (λ*c*. *c* ∗$_C$ *id-cblinfun*)›

definition *von-neumann-algebra* **where** ‹*von-neumann-algebra A* ⟷ (∀ *a*∈*A*. *a*∗ ∈ *A*) ∧ *commutant* (*commutant A*) = *A*›
definition *von-neumann-factor* **where** ‹*von-neumann-factor A* ⟷ *von-neumann-algebra A* ∧ *A* ∩ *commutant A* = *one-algebra*›

lemma *von-neumann-algebraI*: ‹(⋀*a*. *a*∈*A* ⟹ *a*∗ ∈ *A*) ⟹ *commutant* (*commutant A*) ⊆ *A* ⟹ *von-neumann-algebra A*› **for** 𝔉
apply (*auto simp*: *von-neumann-algebra-def*)

using *double-commutant-grows* **by** *blast*

lemma *von-neumann-factorI*:
assumes ‹*von-neumann-algebra A*›
assumes ‹*A* ∩ *commutant A* ⊆ *one-algebra*›
shows ‹*von-neumann-factor A*›
proof −
have *1*: ‹*A* ⊇ *one-algebra*›
apply (*subst asm-rl*[*of* ‹*A* = *commutant* (*commutant A*)›])
using *assms*(*1*) *von-neumann-algebra-def* **apply** *blast*
by (*auto simp*: *commutant-def one-algebra-def*)
have *2*: ‹*commutant A* ⊇ *one-algebra*›
by (*auto simp*: *commutant-def one-algebra-def*)
from *1 2 assms* **show** *?thesis*
by (*auto simp add*: *von-neumann-factor-def*)
qed

lemma *commutant-UNIV*: ‹*commutant* (*UNIV* :: ('*a* ⇒$_{CL}$ '*a*::*chilbert-space*) *set*) = *one-algebra*›

proof −
have *1*: ‹*c* ∗$_C$ *id-cblinfun* ∈ *commutant UNIV*› **for** *c*
by (*simp add*: *commutant-def*)
moreover have *2*: ‹*x* ∈ *range* (λ*c*. *c* ∗$_C$ *id-cblinfun*)› **if** *x-comm*: ‹*x* ∈ *commutant UNIV*› **for** *x* :: ‹'*a* ⇒$_{CL}$ '*a*›
proof −
obtain *B* :: ‹'*a set*› **where** ‹*is-onb B*›
using *is-onb-some-chilbert-basis* **by** *blast*
have ‹∃ *c*. *x* ∗$_V$ ψ = *c* ∗$_C$ ψ› **for** ψ
proof −
have ‹*norm* (*x* ∗$_V$ ψ) = *norm* ((*x* *o*$_{CL}$ *selfbutter* (*sgn* ψ)) ∗$_V$ ψ)›
by (*simp add*: *cnorm-eq-1*)
also have ‹... = *norm* ((*selfbutter* (*sgn* ψ) *o*$_{CL}$ *x*) ∗$_V$ ψ)›
using *x-comm* **by** (*simp add*: *commutant-def del*: *butterfly-apply*)
also have ‹... = *norm* (*proj* ψ ∗$_V$ (*x* ∗$_V$ ψ))›
by (*simp add*: *butterfly-sgn-eq-proj*)
finally have ‹*x* ∗$_V$ ψ ∈ *space-as-set* (*ccspan* {ψ})›
by (*metis norm-Proj-apply*)
then show *?thesis*
by (*auto simp add*: *ccspan-finite complex-vector.span-singleton*)
qed
then obtain *f* **where** *f*: ‹*x* ∗$_V$ ψ = *f* ψ ∗$_C$ ψ› **for** ψ
apply *atomize-elim* **apply** (*rule choice*) **by** *auto*

have ‹*f* ψ = *f* φ› **if** ‹ψ ∈ *B*› **and** ‹φ ∈ *B*› **for** ψ φ
proof (*cases* ‹ψ = φ›)
case *True*
then show *?thesis* **by** *simp*
next
case *False*

```
    with that ‹is-onb B›
    have [simp]: ‹ψ ·C φ = 0›
      by (auto simp: is-onb-def is-ortho-set-def)
    then have [simp]: ‹φ ·C ψ = 0›
      using is-orthogonal-sym by blast
    from that ‹is-onb B› have [simp]: ‹ψ ≠ 0›
      by (auto simp: is-onb-def)
    from that ‹is-onb B› have [simp]: ‹φ ≠ 0›
      by (auto simp: is-onb-def)

    have ‹f (ψ+φ) *C ψ + f (ψ+φ) *C φ = f (ψ+φ) *C (ψ + φ)›
      by (simp add: complex-vector.vector-space-assms(1))
    also have ‹... = x *V (ψ + φ)›
      by (simp add: f)
    also have ‹... = x *V ψ + x *V φ›
      by (simp add: cblinfun.add-right)
    also have ‹... = f ψ *C ψ + f φ *C φ›
      by (simp add: f)
    finally have *: ‹f (ψ + φ) *C ψ + f (ψ + φ) *C φ = f ψ *C ψ + f φ *C φ›
      by −
    have ‹f (ψ + φ) = f ψ›
      using *[THEN arg-cong[where f=‹cinner ψ›]]
      by (simp add: cinner-add-right)
    moreover have ‹f (ψ + φ) = f φ›
      using *[THEN arg-cong[where f=‹cinner φ›]]
      by (simp add: cinner-add-right)
    ultimately show ‹f ψ = f φ›
      by simp
  qed
  then obtain c where ‹f ψ = c› if ‹ψ ∈ B› for ψ
    by meson
  then have ‹x *V ψ = (c *C id-cblinfun) *V ψ› if ‹ψ ∈ B› for ψ
    by (simp add: f that)
  then have ‹x = c *C id-cblinfun›
    apply (rule cblinfun-eq-gen-eqI[where G=B])
    using ‹is-onb B› by (auto simp: is-onb-def)
  then show ‹x ∈ range (λc. c *C id-cblinfun)›
    by (auto)
qed

from 1 2 show ?thesis
  by (auto simp: one-algebra-def)
qed


lemma von-neumann-algebra-UNIV: ‹von-neumann-algebra UNIV›
  by (auto simp: von-neumann-algebra-def commutant-def)

lemma von-neumann-factor-UNIV: ‹von-neumann-factor UNIV›
```

by (*simp add*: *von-neumann-factor-def commutant-UNIV von-neumann-algebra-UNIV*)

lemma *von-neumann-algebra-UNION*:
assumes ‹$\bigwedge x.\ x \in X \Longrightarrow$ *von-neumann-algebra* (*A x*)›
shows ‹*von-neumann-algebra* (*commutant* (*commutant* ($\bigcup x{\in}X.$ *A x*)))›
proof (*rule von-neumann-algebraI*)
show ‹*commutant* (*commutant* (*commutant* (*commutant* ($\bigcup x{\in}X.$ *A x*))))
$\subseteq$ *commutant* (*commutant* ($\bigcup x{\in}X.$ *A x*))›
by (*meson commutant-antimono double-commutant-grows*)
next
fix *a*
assume ‹*a* $\in$ *commutant* (*commutant* ($\bigcup x{\in}X.$ *A x*))›
then have ‹*a*∗ $\in$ *adj* ‘ *commutant* (*commutant* ($\bigcup x{\in}X.$ *A x*))›
by *simp*
also have ‹... = *commutant* (*commutant* ($\bigcup x{\in}X.$ *adj* ‘ *A x*))›
by (*simp add*: *commutant-adj image-UN*)
also have ‹... $\subseteq$ *commutant* (*commutant* ($\bigcup x{\in}X.$ *A x*))›
using *assms* **by** (*auto simp*: *von-neumann-algebra-def intro*!: *commutant-antimono*)
finally show ‹*a*∗ $\in$ *commutant* (*commutant* ($\bigcup x{\in}X.$ *A x*))›
by −
qed

lemma *von-neumann-algebra-union*:
assumes ‹*von-neumann-algebra A*›
assumes ‹*von-neumann-algebra B*›
shows ‹*von-neumann-algebra* (*commutant* (*commutant* (*A* ∪ *B*)))›
using *von-neumann-algebra-UNION*[**where** *X*=‹{*True*,*False*}› **and** *A*=‹$\lambda x.$ *if x then A else B*›]
by (*auto simp*: *assms Un-ac*(*3*))

lemma *von-neumann-algebra-commutant*: ‹*von-neumann-algebra* (*commutant A*)› **if** ‹*von-neumann-algebra A*›
proof (*rule von-neumann-algebraI*)
show ‹*a*∗ $\in$ *commutant A*› **if** ‹*a* $\in$ *commutant A*› **for** *a*
by (*smt* (*verit*) *Set.basic-monos*(*7*) ‹*von-neumann-algebra A*› *commutant-adj commutant-antimono double-adj image-iff image-subsetI that von-neumann-algebra-def*)
show ‹*commutant* (*commutant* (*commutant A*)) $\subseteq$ *commutant A* ›
by *simp*
qed

lemma *von-neumann-algebra-def-sot*:
‹*von-neumann-algebra* $\mathfrak{F}$ $\longleftrightarrow$
($\forall\, a{\in}\mathfrak{F}.\ a* \in \mathfrak{F}$) $\wedge$ *csubspace* $\mathfrak{F}$ $\wedge$ ($\forall\, a{\in}\mathfrak{F}.\ \forall\, b{\in}\mathfrak{F}.\ a\ o_{CL}\ b \in \mathfrak{F}$) $\wedge$ *id-cblinfun* $\in \mathfrak{F}$ $\wedge$
closedin cstrong-operator-topology $\mathfrak{F}$›
proof (*unfold von-neumann-algebra-def*, *intro iffI conjI*; *elim conjE*; *assumption?*)
assume *comm*: ‹*commutant* (*commutant* $\mathfrak{F}$) = $\mathfrak{F}$›
from *comm* **show** ‹*closedin cstrong-operator-topology* $\mathfrak{F}$›
by (*metis commutant-sot-closed*)

from *comm* **show** ‹*csubspace* $\mathfrak{F}$›
by (*metis csubspace-commutant*)
from *comm* **show** ‹$\forall\, a{\in}\mathfrak{F}.\ \forall\, b{\in}\mathfrak{F}.\ a\ o_{CL}\ b \in \mathfrak{F}$›
using *commutant-mult* **by** *blast*
from *comm* **show** ‹*id-cblinfun* $\in \mathfrak{F}$›
by *blast*
next
assume *adj*: ‹$\forall\, a{\in}\mathfrak{F}.\ a* \in \mathfrak{F}$›
assume *subspace*: ‹*csubspace* $\mathfrak{F}$›
assume *closed*: ‹*closedin cstrong-operator-topology* $\mathfrak{F}$›
assume *mult*: ‹$\forall\, a{\in}\mathfrak{F}.\ \forall\, b{\in}\mathfrak{F}.\ a\ o_{CL}\ b \in \mathfrak{F}$›
assume *id*: ‹*id-cblinfun* $\in \mathfrak{F}$›
have ‹*commutant* (*commutant* $\mathfrak{F}$) = *cstrong-operator-topology closure-of* $\mathfrak{F}$›
apply (*rule double-commutant-theorem*)
thm *double-commutant-theorem*[*of* $\mathfrak{F}$]
using *subspace subspace mult id adj*
by *simp-all*
also from *closed* **have** ‹... = $\mathfrak{F}$›
by (*simp add*: *closure-of-eq*)
finally show ‹*commutant* (*commutant* $\mathfrak{F}$) = $\mathfrak{F}$›
by −
qed

lemma *double-commutant-hull'*:
assumes ‹$\bigwedge x.\ x \in X \Longrightarrow x* \in X$›
shows ‹*commutant* (*commutant X*) = *von-neumann-algebra hull X*›
proof (*rule antisym*)
show ‹*commutant* (*commutant X*) ⊆ *von-neumann-algebra hull X*›
apply (*subst double-commutant-hull*)
apply (*rule hull-antimono*)
by (*simp add*: *von-neumann-algebra-def*)
show ‹*von-neumann-algebra hull X* ⊆ *commutant* (*commutant X*)›
apply (*rule hull-minimal*)
by (*simp-all add*: *von-neumann-algebra-def assms commutant-adj-closed*)
qed

lemma *commutant-one-algebra*: ‹*commutant one-algebra* = *UNIV*›
by (*metis commutant-UNIV commutant-empty triple-commutant*)

definition *tensor-vn* (**infixr** $\otimes_{vN}$ *70*) **where**
‹*tensor-vn X Y* = *commutant* (*commutant* ((λ*a*. *a* $\otimes_o$ *id-cblinfun*) ‘ *X* ∪ (λ*a*. *id-cblinfun* $\otimes_o$ *a*) ‘ *Y*))›

lemma *von-neumann-algebra-adj-image*: ‹*von-neumann-algebra X* ⟹ *adj* ‘ *X* = *X*›
by (*auto simp*: *von-neumann-algebra-def intro*!: *image-eqI*[**where** *x*=‹-*›])

lemma *von-neumann-algebra-tensor-vn*:

assumes ‹*von-neumann-algebra X*›
assumes ‹*von-neumann-algebra Y*›
shows ‹*von-neumann-algebra* (*X* $\otimes_{vN}$ *Y*)›
proof (*rule von-neumann-algebraI*)
have ‹*adj* ‘ (*X* $\otimes_{vN}$ *Y*) = *commutant* (*commutant* ((λ*a*. *a* $\otimes_o$ *id-cblinfun*) ‘ *adj* ‘ *X* ∪ (λ*a*. *id-cblinfun* $\otimes_o$ *a*) ‘ *adj* ‘ *Y*))›
by (*simp add*: *tensor-vn-def commutant-adj image-image image-Un tensor-op-adjoint*)
also have ‹... = *commutant* (*commutant* ((λ*a*. *a* $\otimes_o$ *id-cblinfun*) ‘ *X* ∪ (λ*a*. *id-cblinfun* $\otimes_o$ *a*) ‘ *Y*))›
using *assms* **by** (*simp add*: *von-neumann-algebra-adj-image*)
also have ‹... = *X* $\otimes_{vN}$ *Y*›
by (*simp add*: *tensor-vn-def*)
finally show ‹*a*∗ ∈ *X* $\otimes_{vN}$ *Y*› **if** ‹*a* ∈ *X* $\otimes_{vN}$ *Y*› **for** *a*
using *that* **by** *blast*
show ‹*commutant* (*commutant* (*X* $\otimes_{vN}$ *Y*)) ⊆ *X* $\otimes_{vN}$ *Y*›
by (*simp add*: *tensor-vn-def*)
qed

lemma *tensor-vn-one-one*[*simp*]: ‹*one-algebra* $\otimes_{vN}$ *one-algebra* = *one-algebra*›
apply (*simp add*: *tensor-vn-def one-algebra-def image-image*
tensor-op-scaleC-left tensor-op-scaleC-right)
by (*simp add*: *commutant-one-algebra commutant-UNIV flip*: *one-algebra-def*)

lemma *sandwich-swap-tensor-vn*: ‹*sandwich swap-ell2* ‘ (*X* $\otimes_{vN}$ *Y*) = *Y* $\otimes_{vN}$ *X*›
by (*simp add*: *tensor-vn-def sandwich-unitary-commutant image-Un image-image Un-commute*)

lemma *tensor-vn-one-left*: ‹*one-algebra* $\otimes_{vN}$ *X* = (λ*x*. *id-cblinfun* $\otimes_o$ *x*) ‘ *X*› **if** ‹*von-neumann-algebra X*›
proof −
have ‹*one-algebra* $\otimes_{vN}$ *X* = *commutant*
(*commutant* ((λ*a*. *id-cblinfun* $\otimes_o$ *a*) ‘ *X*))›
apply (*simp add*: *tensor-vn-def one-algebra-def image-image*)
by (*metis* (*no-types*, *lifting*) *Un-commute Un-empty-right commutant-UNIV commutant-empty double-commutant-Un-right image-cong one-algebra-def tensor-id tensor-op-scaleC-left*)
also have ‹... = (λ*a*. *id-cblinfun* $\otimes_o$ *a*) ‘ *commutant* (*commutant X*)›
by (*simp add*: *amplification-double-commutant-commute*′)
also have ‹... = (λ*a*. *id-cblinfun* $\otimes_o$ *a*) ‘ *X*›
using *that von-neumann-algebra-def* **by** *blast*
finally show *?thesis*
by −
qed
lemma *tensor-vn-one-right*: ‹*X* $\otimes_{vN}$ *one-algebra* = (λ*x*. *x* $\otimes_o$ *id-cblinfun*) ‘ *X*› **if** ‹*von-neumann-algebra X*›
proof −
have ‹*X* $\otimes_{vN}$ *one-algebra* = *sandwich swap-ell2* ‘ (*one-algebra* $\otimes_{vN}$ *X*)›
by (*simp add*: *sandwich-swap-tensor-vn*)
also have ‹... = *sandwich swap-ell2* ‘ (λ*x*. *id-cblinfun* $\otimes_o$ *x*) ‘ *X*›
by (*simp add*: *tensor-vn-one-left that*)
also have ‹... = (λ*x*. *x* $\otimes_o$ *id-cblinfun*) ‘ *X*›

```
    by (simp add: image-image)
  finally show ?thesis
    by −
qed

lemma double-commutant-in-vn-algI: ‹commutant (commutant X) ⊆ Y›
  if ‹von-neumann-algebra Y› and ‹X ⊆ Y›
  by (metis commutant-antimono that(1) that(2) von-neumann-algebra-def)

lemma von-neumann-algebra-compose:
  assumes ‹von-neumann-algebra M›
  assumes ‹x ∈ M› and ‹y ∈ M›
  shows ‹x o_CL y ∈ M›
  using assms apply (auto simp: von-neumann-algebra-def commutant-def)
  by (metis (no-types, lifting) assms(1) commutant-mult von-neumann-algebra-def)

lemma von-neumann-algebra-id:
  assumes ‹von-neumann-algebra M›
  shows ‹id-cblinfun ∈ M›
  using assms by (auto simp: von-neumann-algebra-def)

lemma tensor-vn-UNIV[simp]: ‹UNIV ⊗_vN UNIV = (UNIV :: (('a×'b) ell2 ⇒_CL -) set)›
proof −
  have ‹(UNIV ⊗_vN UNIV :: (('a×'b) ell2 ⇒_CL -) set) =
        commutant (commutant (range (λa. a ⊗_o id-cblinfun) ∪ range (λa. id-cblinfun ⊗_o a)))›
(is ‹- = ?rhs›)
    by (simp add: tensor-vn-def commutant-cspan)
  also have ‹... ⊇ commutant (commutant {a ⊗_o b |a b. True})› (is ‹- ⊇ ...›)
  proof (rule double-commutant-in-vn-algI)
    show vn: ‹von-neumann-algebra ?rhs›
      by (metis calculation von-neumann-algebra-UNIV von-neumann-algebra-tensor-vn)
    show ‹{a ⊗_o b |(a :: 'a ell2 ⇒_CL -) (b :: 'b ell2 ⇒_CL -). True} ⊆ ?rhs›
    proof (rule subsetI)
      fix x :: ‹('a × 'b) ell2 ⇒_CL ('a × 'b) ell2›
      assume ‹x ∈ {a ⊗_o b |a b. True}›
      then obtain a b where ‹x = a ⊗_o b›
        by auto
      then have ‹x = (a ⊗_o id-cblinfun) o_CL (id-cblinfun ⊗_o b)›
        by (simp add: comp-tensor-op)
      also have ‹... ∈ ?rhs›
      proof −
        have ‹a ⊗_o id-cblinfun ∈ ?rhs›
          by (auto intro!: double-commutant-grows')
        moreover have ‹id-cblinfun ⊗_o b ∈ ?rhs›
          by (auto intro!: double-commutant-grows')
        ultimately show ?thesis
          using commutant-mult by blast
      qed
      finally show ‹x ∈ ?rhs›
```

by −
qed
qed
also have ‹... = *cstrong-operator-topology closure-of* (*cspan* {*a* ⊗$_o$ *b* |*a b*. *True*})›
apply (*rule double-commutant-theorem-span*)
apply (*auto simp*: *comp-tensor-op tensor-op-adjoint*)
using *tensor-id*[*symmetric*] **by** *blast*+
also have ‹... = *UNIV*›
using *tensor-op-dense* **by** *blast*
finally show *?thesis*
by *auto*
qed

end

17 *Tensor-Product-Code* − Support for code generation

theory *Tensor-Product-Code*
imports *Hilbert-Space-Tensor-Product*
Complex-Bounded-Operators.Cblinfun-Code
begin

Automatic evaluation of formulas involving finite dimensional tensor products. Builds upon *Complex-Bounded-Operators.Cblinfun-Code* and reduces computations to the existing procedures from `Jordan_Normal_Form`.

unbundle *cblinfun-notation Finite-Cartesian-Product.no-vec-syntax jnf-notation*
hide-const (**open**) *Finite-Cartesian-Product.vec*
hide-const (**open**) *Finite-Cartesian-Product.mat*

definition *tensor-pack* :: *nat* ⇒ *nat* ⇒ (*nat* × *nat*) ⇒ *nat*
where *tensor-pack X Y* = (λ(*x*, *y*). *x* ∗ *Y* + *y*)

definition *tensor-unpack* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ (*nat* × *nat*)
where *tensor-unpack X Y xy* = (*xy div Y*, *xy mod Y*)

lemma *tensor-unpack-inj*:
assumes *i* < *A* ∗ *B* **and** *j* < *A* ∗ *B*
shows *tensor-unpack A B i* = *tensor-unpack A B j* ⟷ *i* = *j*
by (*metis div-mult-mod-eq prod.sel*(*1*) *prod.sel*(*2*) *tensor-unpack-def*)

lemma *tensor-unpack-bound1*[*simp*]: *i* < *A* ∗ *B* ⟹ *fst* (*tensor-unpack A B i*) < *A*
unfolding *tensor-unpack-def*
by (*auto intro*!: *less-mult-imp-div-less*)
lemma *tensor-unpack-bound2*[*simp*]: *i* < *A* ∗ *B* ⟹ *snd* (*tensor-unpack A B i*) < *B*
unfolding *tensor-unpack-def*
by (*auto intro*!: *mod-less-divisor Nat.gr0I*)

lemma *tensor-unpack-fstfst*: ‹*fst* (*tensor-unpack A B* (*fst* (*tensor-unpack* (*A* ∗ *B*) *C i*)))
= *fst* (*tensor-unpack A* (*B* ∗ *C*) *i*)›
unfolding *tensor-unpack-def* **by** (*auto simp flip*: *div-mult2-eq simp*: *mult.commute*)
lemma *tensor-unpack-sndsnd*: ‹*snd* (*tensor-unpack B C* (*snd* (*tensor-unpack A* (*B* ∗ *C*) *i*)))
= *snd* (*tensor-unpack* (*A* ∗ *B*) *C i*)›
unfolding *tensor-unpack-def* **by** (*auto simp*: *mod-mod-cancel*)
lemma *tensor-unpack-fstsnd*: ‹*fst* (*tensor-unpack B C* (*snd* (*tensor-unpack A* (*B* ∗ *C*) *i*)))
= *snd* (*tensor-unpack A B* (*fst* (*tensor-unpack* (*A* ∗ *B*) *C i*)))›
unfolding *tensor-unpack-def*
by (*cases* ‹*C* = *0*›) (*simp-all add*: *mult.commute* [*of B C*] *mod-mult2-eq* [*of i C B*])

definition *tensor-state-jnf* ψ φ = (*let d1* = *dim-vec* ψ *in let d2* = *dim-vec* φ *in*
vec (*d1*∗*d2*) (λ*i*. *let* (*i1*,*i2*) = *tensor-unpack d1 d2 i in* (*vec-index* ψ *i1*) ∗ (*vec-index* φ *i2*)))

lemma *tensor-state-jnf-dim*[*simp*]: ‹*dim-vec* (*tensor-state-jnf* ψ φ) = *dim-vec* ψ ∗ *dim-vec* φ›
unfolding *tensor-state-jnf-def Let-def* **by** *simp*

lemma *enum-prod-nth-tensor-unpack*:
assumes ‹*i* < *CARD*('*a*) ∗ *CARD*('*b*)›
shows (*Enum.enum* ! *i* :: '*a*::*enum*×'*b*::*enum*) =
(*let* (*i1*,*i2*) = *tensor-unpack CARD*('*a*) *CARD*('*b*) *i in*
(*Enum.enum* ! *i1*, *Enum.enum* ! *i2*))
using *assms*
by (*simp add*: *enum-prod-def product-nth tensor-unpack-def*)

lemma *vec-of-basis-enum-tensor-state-index*:
fixes ψ :: ‹'*a*::*enum ell2*› **and** φ :: ‹'*b*::*enum ell2*›
assumes [*simp*]: ‹*i* < *CARD*('*a*) ∗ *CARD*('*b*)›
shows ‹*vec-of-basis-enum* (ψ ⊗$_s$ φ) $ *i* = (*let* (*i1*,*i2*) = *tensor-unpack CARD*('*a*) *CARD*('*b*)
i in
vec-of-basis-enum ψ $ *i1* ∗ *vec-of-basis-enum* φ $ *i2*)›
proof −
define *i1 i2* **where** *i1* = *fst* (*tensor-unpack CARD*('*a*) *CARD*('*b*) *i*)
and *i2* = *snd* (*tensor-unpack CARD*('*a*) *CARD*('*b*) *i*)
have [*simp*]: *i1* < *CARD*('*a*) *i2* < *CARD*('*b*)
using *assms i1-def tensor-unpack-bound1* **apply** *presburger*
using *assms i2-def tensor-unpack-bound2* **by** *presburger*

have ‹*vec-of-basis-enum* (ψ ⊗$_s$ φ) $ *i* = *Rep-ell2* (ψ ⊗$_s$ φ) (*enum-class.enum* ! *i*)›
by (*simp add*: *vec-of-basis-enum-ell2-component*)
also have ‹... = *Rep-ell2* ψ (*Enum.enum*!*i1*) ∗ *Rep-ell2* φ (*Enum.enum*!*i2*)›
apply (*transfer fixing*: *i i1 i2*)
by (*simp add*: *enum-prod-nth-tensor-unpack case-prod-beta i1-def i2-def*)
also have ‹... = *vec-of-basis-enum* ψ $ *i1* ∗ *vec-of-basis-enum* φ $ *i2*›
by (*simp add*: *vec-of-basis-enum-ell2-component*)

finally show *?thesis*
by (*simp add*: *case-prod-beta i1-def i2-def*)
qed

lemma *vec-of-basis-enum-tensor-state*:
fixes ψ :: ‹'*a*::*enum ell2*› **and** φ :: ‹'*b*::*enum ell2*›
shows ‹*vec-of-basis-enum* ($\psi \otimes_s \varphi$) = *tensor-state-jnf* (*vec-of-basis-enum* ψ) (*vec-of-basis-enum* φ)›
apply (*rule eq-vecI*, *simp-all*)
apply (*subst vec-of-basis-enum-tensor-state-index*, *simp-all*)
by (*simp add*: *tensor-state-jnf-def case-prod-beta Let-def*)

lemma *mat-of-cblinfun-tensor-op-index*:
fixes *a* :: ‹'*a*::*enum ell2* $\Rightarrow_{CL}$ '*b*::*enum ell2*› **and** *b* :: ‹'*c*::*enum ell2* $\Rightarrow_{CL}$ '*d*::*enum ell2*›
assumes [*simp*]: ‹*i* < *CARD*('*b*) * *CARD*('*d*)›
assumes [*simp*]: ‹*j* < *CARD*('*a*) * *CARD*('*c*)›
shows ‹*mat-of-cblinfun* (*tensor-op a b*) $$ (*i*,*j*) =
(*let* (*i1*,*i2*) = *tensor-unpack CARD*('*b*) *CARD*('*d*) *i* *in*
let (*j1*,*j2*) = *tensor-unpack CARD*('*a*) *CARD*('*c*) *j* *in*
mat-of-cblinfun a $$ (*i1*,*j1*) * *mat-of-cblinfun b* $$ (*i2*,*j2*))›
proof −
define *i1 i2 j1 j2*
where *i1* = *fst* (*tensor-unpack CARD*('*b*) *CARD*('*d*) *i*)
and *i2* = *snd* (*tensor-unpack CARD*('*b*) *CARD*('*d*) *i*)
and *j1* = *fst* (*tensor-unpack CARD*('*a*) *CARD*('*c*) *j*)
and *j2* = *snd* (*tensor-unpack CARD*('*a*) *CARD*('*c*) *j*)
have [*simp*]: *i1* < *CARD*('*b*) *i2* < *CARD*('*d*) *j1* < *CARD*('*a*) *j2* < *CARD*('*c*)
using *assms i1-def tensor-unpack-bound1* **apply** *presburger*
using *assms i2-def tensor-unpack-bound2* **apply** *blast*
using *assms*(2) *j1-def tensor-unpack-bound1* **apply** *blast*
using *assms*(2) *j2-def tensor-unpack-bound2* **by** *presburger*

have ‹*mat-of-cblinfun* (*tensor-op a b*) $$ (*i*,*j*)
= *Rep-ell2* (*tensor-op a b* $*_V$ *ket* (*Enum.enum*!*j*)) (*Enum.enum* ! *i*)›
by (*simp add*: *mat-of-cblinfun-ell2-component*)
also have ‹... = *Rep-ell2* ((*a* $*_V$ *ket* (*Enum.enum*!*j1*)) $\otimes_s$ (*b* $*_V$ *ket* (*Enum.enum*!*j2*))) (*Enum.enum*!*i*)›
by (*simp add*: *tensor-op-ell2 enum-prod-nth-tensor-unpack*[**where** *i*=*j*] *Let-def case-prod-beta j1-def*[*symmetric*] *j2-def*[*symmetric*] *flip*: *tensor-ell2-ket*)
also have ‹... = *vec-of-basis-enum* ((*a* $*_V$ *ket* (*Enum.enum*!*j1*)) $\otimes_s$ *b* $*_V$ *ket* (*Enum.enum*!*j2*)) $ *i*›
by (*simp add*: *vec-of-basis-enum-ell2-component*)
also have ‹... = *vec-of-basis-enum* (*a* $*_V$ *ket* (*enum-class.enum* ! *j1*)) $ *i1* *
vec-of-basis-enum (*b* $*_V$ *ket* (*enum-class.enum* ! *j2*)) $ *i2*›
by (*simp add*: *case-prod-beta vec-of-basis-enum-tensor-state-index i1-def*[*symmetric*] *i2-def*[*symmetric*])
also have ‹... = *Rep-ell2* (*a* $*_V$ *ket* (*enum-class.enum* ! *j1*)) (*enum-class.enum* ! *i1*) *
Rep-ell2 (*b* $*_V$ *ket* (*enum-class.enum* ! *j2*)) (*enum-class.enum* ! *i2*)›

by (*simp add*: *vec-of-basis-enum-ell2-component*)
also have ‹... = *mat-of-cblinfun a* $$ (*i1*, *j1*) * *mat-of-cblinfun b* $$ (*i2*, *j2*)›
by (*simp add*: *mat-of-cblinfun-ell2-component*)
finally show *?thesis*
by (*simp add*: *i1-def*[*symmetric*] *i2-def*[*symmetric*] *j1-def*[*symmetric*] *j2-def*[*symmetric*] *case-prod-beta*)
qed

definition *tensor-op-jnf A B* =
(*let r1* = *dim-row A in*
let c1 = *dim-col A in*
let r2 = *dim-row B in*
let c2 = *dim-col B in*
mat (*r1* * *r2*) (*c1* * *c2*)
(λ(*i*,*j*). *let* (*i1*,*i2*) = *tensor-unpack r1 r2 i in*
let (*j1*,*j2*) = *tensor-unpack c1 c2 j in*
(*A* $$ (*i1*,*j1*)) * (*B* $$ (*i2*,*j2*))))

lemma *tensor-op-jnf-dim*[*simp*]:
‹*dim-row* (*tensor-op-jnf a b*) = *dim-row a* * *dim-row b*›
‹*dim-col* (*tensor-op-jnf a b*) = *dim-col a* * *dim-col b*›
unfolding *tensor-op-jnf-def Let-def* **by** *simp-all*

lemma *mat-of-cblinfun-tensor-op*:
fixes *a* :: ‹'*a*::*enum ell2* ⇒$_{CL}$ '*b*::*enum ell2*› **and** *b* :: ‹'*c*::*enum ell2* ⇒$_{CL}$ '*d*::*enum ell2*›
shows ‹*mat-of-cblinfun* (*tensor-op a b*) = *tensor-op-jnf* (*mat-of-cblinfun a*) (*mat-of-cblinfun b*)›
apply (*rule eq-matI*, *simp-all add*: *canonical-basis-length*)
apply (*subst mat-of-cblinfun-tensor-op-index*, *simp-all*)
by (*simp add*: *tensor-op-jnf-def case-prod-beta Let-def canonical-basis-length*)

lemma *mat-of-cblinfun-assoc-ell2'*[*simp*]:
‹*mat-of-cblinfun* (*assoc-ell2** :: (('*a*::*enum*×('*b*::*enum*×'*c*::*enum*)) *ell2* ⇒$_{CL}$ -)) = *one-mat* (*CARD*('*a*)*CARD*('*b*)*CARD*('*c*))›
(**is** *mat-of-cblinfun ?assoc* = -)
proof (*rule mat-eq-iff*[*THEN iffD2*], *intro conjI allI impI*)

show ‹*dim-row* (*mat-of-cblinfun ?assoc*) =
dim-row (*1*$_m$ (*CARD*('*a*) * *CARD*('*b*) * *CARD*('*c*)))›
by (*simp add*: *canonical-basis-length*)
show ‹*dim-col* (*mat-of-cblinfun ?assoc*) =
dim-col (*1*$_m$ (*CARD*('*a*) * *CARD*('*b*) * *CARD*('*c*)))›
by (*simp add*: *canonical-basis-length*)

fix *i j*
let *?i* = *Enum.enum* ! *i* :: (('*a*×'*b*)×'*c*) **and** *?j* = *Enum.enum* ! *j* :: ('*a*×('*b*×'*c*))

assume ‹i < *dim-row* (1_m (*CARD*($'a$) * *CARD*($'b$) * *CARD*($'c$)))›
then have *iB*[*simp*]: ‹i < *CARD*($'a$) * *CARD*($'b$) * *CARD*($'c$)› **by** *simp*
then have *iB'*[*simp*]: ‹i < *CARD*($'a$) * (*CARD*($'b$) * *CARD*($'c$))› **by** *linarith*
assume ‹j < *dim-col* (1_m (*CARD*($'a$) * *CARD*($'b$) * *CARD*($'c$)))›
then have *jB*[*simp*]: ‹j < *CARD*($'a$) * *CARD*($'b$) * *CARD*($'c$)› **by** *simp*
then have *jB'*[*simp*]: ‹j < *CARD*($'a$) * (*CARD*($'b$) * *CARD*($'c$))› **by** *linarith*

define *i1 i23 i2 i3*
where *i1* = *fst* (*tensor-unpack CARD*($'a$) (*CARD*($'b$)**CARD*($'c$)) *i*)
and *i23* = *snd* (*tensor-unpack CARD*($'a$) (*CARD*($'b$)**CARD*($'c$)) *i*)
and *i2* = *fst* (*tensor-unpack CARD*($'b$) *CARD*($'c$) *i23*)
and *i3* = *snd* (*tensor-unpack CARD*($'b$) *CARD*($'c$) *i23*)
define *j12 j1 j2 j3*
where *j12* = *fst* (*tensor-unpack* (*CARD*($'a$)**CARD*($'b$)) *CARD*($'c$) *j*)
and *j1* = *fst* (*tensor-unpack CARD*($'a$) *CARD*($'b$) *j12*)
and *j2* = *snd* (*tensor-unpack CARD*($'a$) *CARD*($'b$) *j12*)
and *j3* = *snd* (*tensor-unpack* (*CARD*($'a$)**CARD*($'b$)) *CARD*($'c$) *j*)

have [*simp*]: *j12* < *CARD*($'a$)**CARD*($'b$) *i23* < *CARD*($'b$)**CARD*($'c$)
using *j12-def jB tensor-unpack-bound1* **apply** *presburger*
using *i23-def iB' tensor-unpack-bound2* **by** *blast*

have *j1'*: ‹*fst* (*tensor-unpack CARD*($'a$) (*CARD*($'b$) * *CARD*($'c$)) *j*) = *j1*›
by (*simp add*: *j1-def j12-def tensor-unpack-fstfst*)

let *?i1* = *Enum.enum* ! *i1* :: $'a$ **and** *?i2* = *Enum.enum* ! *i2* :: $'b$ **and** *?i3* = *Enum.enum* ! *i3* :: $'c$
let *?j1* = *Enum.enum* ! *j1* :: $'a$ **and** *?j2* = *Enum.enum* ! *j2* :: $'b$ **and** *?j3* = *Enum.enum* ! *j3* :: $'c$

have *i*: ‹*?i* = ((*?i1*,*?i2*),*?i3*)›
by (*auto simp add*: *enum-prod-nth-tensor-unpack case-prod-beta*
tensor-unpack-fstfst tensor-unpack-fstsnd tensor-unpack-sndsnd i1-def i2-def i23-def i3-def)
have *j*: ‹*?j* = (*?j1*,(*?j2*,*?j3*))›
by (*auto simp add*: *enum-prod-nth-tensor-unpack case-prod-beta*
tensor-unpack-fstfst tensor-unpack-fstsnd tensor-unpack-sndsnd j1-def j2-def j12-def j3-def)
have *ijeq*: ‹(*?i1*,*?i2*,*?i3*) = (*?j1*,*?j2*,*?j3*) ⟷ *i* = *j*›
unfolding *i1-def i2-def i3-def j1-def j2-def j3-def* **apply** *simp*
apply (*subst enum-inj*, *simp*, *simp*)
apply (*subst enum-inj*, *simp*, *simp*)
apply (*subst enum-inj*, *simp*, *simp*)
apply (*subst tensor-unpack-inj*[*symmetric*, **where** *i*=*i* **and** *j*=*j* **and** *A*=*CARD*($'a$) **and** *B*=*CARD*($'b$)**CARD*($'c$)], *simp*, *simp*)
unfolding *prod-eq-iff*
apply (*subst tensor-unpack-inj*[*symmetric*, **where** *i*=‹*snd* (*tensor-unpack CARD*($'a$) (*CARD*($'b$) * *CARD*($'c$)) *i*)› **and** *A*=*CARD*($'b$) **and** *B*=*CARD*($'c$)], *simp*, *simp*)
by (*simp add*: *i1-def*[*symmetric*] *j1-def*[*symmetric*] *i2-def*[*symmetric*] *j2-def*[*symmetric*]

i3-def[*symmetric*] *j3-def*[*symmetric*]
i23-def[*symmetric*] *j12-def*[*symmetric*] *j1'*
prod-eq-iff tensor-unpack-fstsnd tensor-unpack-sndsnd)

have ‹*mat-of-cblinfun ?assoc* $$ (*i*, *j*) = *Rep-ell2* (*assoc-ell2∗* $*_V$ *ket ?j*) *?i*›
by (*subst mat-of-cblinfun-ell2-component*, *auto*)
also have ‹... = *Rep-ell2* ((*ket ?j1* $\otimes_s$ *ket ?j2*) $\otimes_s$ *ket ?j3*) *?i*›
by (*simp add*: *j assoc-ell2'-tensor flip*: *tensor-ell2-ket*)
also have ‹... = (*if* (*?i1*, *?i2*, *?i3*) = (*?j1*, *?j2*, *?j3*) *then 1 else 0*)›
by (*auto simp add*: *ket.rep-eq i tensor-ell2-ket*)
also have ‹... = (*if i=j then 1 else 0*)›
using *ijeq* **by** *simp*
finally
show ‹*mat-of-cblinfun ?assoc* $$ (*i*, *j*) =
1$_m$ (*CARD*('*a*) ∗ *CARD*('*b*) ∗ *CARD*('*c*)) $$ (*i*, *j*)›
by *auto*
qed

lemma *mat-of-cblinfun-assoc-ell2*[*simp*]:
‹*mat-of-cblinfun* (*assoc-ell2* :: ((('*a*::*enum*×'*b*::*enum*)×'*c*::*enum*) *ell2* ⇒$_{CL}$ -)) = *one-mat* (*CARD*('*a*)∗*CARD*('*b*)∗*CARD*('*c*))›
(**is** *mat-of-cblinfun ?assoc* = -)
proof −
let *?assoc'* = *assoc-ell2∗* :: (('*a*::*enum*×('*b*::*enum*×'*c*::*enum*)) *ell2* ⇒$_{CL}$ -)
have *one-mat* (*CARD*('*a*)∗*CARD*('*b*)∗*CARD*('*c*)) = *mat-of-cblinfun* (*?assoc* o_{CL} *?assoc'*)
by (*simp add*: *mult.assoc mat-of-cblinfun-id*)
also have ‹... = *mat-of-cblinfun ?assoc* ∗ *mat-of-cblinfun ?assoc'*›
using *mat-of-cblinfun-compose* **by** *blast*
also have ‹... = *mat-of-cblinfun ?assoc* ∗ *one-mat* (*CARD*('*a*)∗*CARD*('*b*)∗*CARD*('*c*))›
by *simp*
also have ‹... = *mat-of-cblinfun ?assoc*›
apply (*rule right-mult-one-mat'*)
by (*simp add*: *canonical-basis-length*)
finally show *?thesis*
by *simp*
qed

end

References

[1] J. B. Conway. *A course in operator theory.* Number 21 in Graduate studies in mathematics. American Mathematical Society, Providence, RI, 2000.

[2] J. B. Conway. *A course in functional analysis*, volume 96. Springer Science & Business Media, 2013.

[3] Faisal and Y. Choi. tr(ab) = tr(ba)? Mathoverflow answer, https://mathoverflow.net/a/76389/101775, 2011–2014.

[4] A. Henriques and A. Taghavi. tr(ab) = tr(ba)? Mathoverflow question, https://mathoverflow.net/questions/76386/trab-trba, 2011–2014.

[5] E. W. (https://math.stackexchange.com/users/86856/eric wofsey). A bounded increasing net converges formulated using filters. Mathematics Stack Exchange (Answer). URL:https://math.stackexchange.com/q/4749216 (version: 2023-08-07).

[6] O. Kunar and A. Popescu. From types to sets by local type definition in higher-order logic. *Journal of Automated Reasoning*, 62(2):237260, June 2018.

[7] M. Takesaki. *Theory of operator algebras I.* Springer, New York, NY, Nov. 2011.

[8] D. Unruh. Quantum references. arXiv:2105.10914v3 [cs.LO], 2024.

[9] F. Wecken. Zur theorie linearer operatoren. *Math. Ann.*, 110:722–725, 1935.