Complex Bounded Operators*

Jose Manuel Rodriguez Caballero[1] and Dominique Unruh[1]

[1]University of Tartu

October 13, 2025

Abstract

We present a formalization of bounded operators on complex vector spaces. Our formalization contains material on complex vector spaces (normed spaces, Banach spaces, Hilbert spaces) that complements and goes beyond the developments of real vectors spaces in the Isabelle/HOL standard library. We define the type of bounded operators between complex vector spaces (*cblinfun*) and develop the theory of unitaries, projectors, extension of bounded linear functions (BLT theorem), adjoints, Loewner order, closed subspaces and more. For the finite-dimensional case, we provide code generation support by identifying finite-dimensional operators with matrices as formalized in the *Jordan_Normal_Form* AFP entry.

Contents

*Supported by the ERC consolidator grant CerQuS (819317), the PRG team grant Secure Quantum Technology (PRG946) from the Estonian Research Council, the Estonian Centre of Exellence in IT (EXCITE) funded by ERDF, and the Estonian Cluster of Excellence "Foundations of the Universe" (TK202).

Theories whose names end with *0* are complex analogues of the similarly named theories concerning real vector spaces in the Isabelle/HOL standard library. They are kept in sync with their real counterparts. The theories without *0* contain material that goes beyond the material in the Isabelle/HOL standard library. This separation allows to keep the material in sync more easily when the Isabelle/HOL standard library is updated.

1 *Extra-Pretty-Code-Examples* – Setup for nicer output of *value*

theory *Extra-Pretty-Code-Examples*
imports
HOL−Examples.Sqrt
Real-Impl.Real-Impl
HOL−Library.Code-Target-Numeral
Jordan-Normal-Form.Matrix-Impl
begin

Some setup that makes the output of the *value* command more readable if matrices and complex numbers are involved.

It is not recommended to import this theory in theories that get included in actual developments (because of the changes to the code generation setup).

It is meant for inclusion in example theories only.

lemma *two-sqrt-irrat*[*simp*]: $2 \in$ *sqrt-irrat*
using *sqrt-prime-irrational*[*OF two-is-prime-nat*]
unfolding *Rats-def sqrt-irrat-def image-def* **apply** *auto*
proof −
fix p :: *rat*
assume $p * p = 2$
hence *f1*: $(Ratreal\ p)^2 = real\ 2$

by (*metis Ratreal-def of-nat-numeral-eq of-rat-numeral-eq power2-eq-square real-times-code*)
have $\forall r.$ *if* $0 \leq r$ *then sqrt* $(r^2) = r$ *else* $r +$ *sqrt* $(r^2) = 0$
by *simp*
hence *f2*: *Ratreal* $p +$ *sqrt* $((Ratreal\ p)^2) = 0$
using *f1* **by** (*metis Ratreal-def Rats-def* ‹*sqrt* (*real 2*) $\notin \mathbb{Q}$› *range-eqI*)
have *f3*: *sqrt* (*real 2*) $+ -$ *1* $*$ *sqrt* $((Ratreal\ p)^2) \leq 0$
using *f1* **by** *fastforce*
have *f4*: $0 \leq$ *sqrt* (*real 2*) $+ -$ *1* $*$ *sqrt* $((Ratreal\ p)^2)$
using *f1* **by** *force*
have *f5*: $(-$ *1* $*$ *sqrt* (*real 2*) $=$ *real-of-rat p*$) = ($*sqrt* (*real 2*) $+$ *real-of-rat p* $=$ *0*$)$
by *linarith*
have $\forall$ *x0*. $-$ (*x0*::*real*) $= -$ *1* $*$ *x0*
by *auto*
hence *sqrt* (*real 2*) $+$ *real-of-rat p* $\neq$ *0*
using *f5* **by** (*metis* (*no-types*) *Rats-def Rats-minus-iff* ‹*sqrt* (*real 2*) $\notin \mathbb{Q}$› *range-eqI*)
thus *False*
using *f4 f3 f2* **by** *simp*
qed

lemma *complex-number-code-post*[*code-post*]:
shows *Complex a 0* $=$ *complex-of-real a*
and *complex-of-real 0* $=$ *0*
and *complex-of-real 1* $=$ *1*
and *complex-of-real* (a/b) $=$ *complex-of-real a* $/$ *complex-of-real b*
and *complex-of-real* (*numeral n*) $=$ *numeral n*
and *complex-of-real* $(-r) = -$ *complex-of-real r*
using *complex-eq-cancel-iff2* **by** *auto*

lemma *real-number-code-post*[*code-post*]:
shows *real-of* (*Abs-mini-alg* $(p,\ 0,\ b)$) $=$ *real-of-rat p*
and *real-of* (*Abs-mini-alg* $(p,\ q,\ 2)$) $=$ *real-of-rat p* $+$ *real-of-rat q* $*$ *sqrt 2*
and *sqrt 0* $=$ *0*
and *sqrt* (*real 0*) $=$ *0*
and $x *$ (*0*::*real*) $=$ *0*
and (*0*::*real*) $* x =$ *0*
and (*0*::*real*) $+ x = x$
and $x +$ (*0*::*real*) $= x$
and (*1*::*real*) $* x = x$
and $x *$ (*1*::*real*) $= x$
by (*auto simp add*: *eq-onp-same-args real-of.abs-eq*)

translations $x \rightharpoonup$ *CONST IArray x*

end

2 *Extra-General* – General missing things

theory *Extra-General*
imports
HOL−Library.Cardinality
HOL−Analysis.Elementary-Topology
HOL−Analysis.Uniform-Limit
HOL−Library.Set-Algebras
HOL−Types-To-Sets.Types-To-Sets
HOL−Library.Complex-Order
HOL−Analysis.Infinite-Sum
HOL−Cardinals.Cardinals
HOL−Library.Complemented-Lattices
HOL−Analysis.Abstract-Topological-Spaces
begin

2.1 Misc

lemma *reals-zero-comparable*:
fixes x::*complex*
assumes $x \in \mathbb{R}$
shows $x \leq 0 \vee x \geq 0$
using *assms* **unfolding** *complex-is-real-iff-compare0* **by** *assumption*

lemma *unique-choice*: $\forall x.\ \exists! y.\ Q\ x\ y \Longrightarrow \exists! f.\ \forall x.\ Q\ x\ (f\ x)$
apply (*auto intro*!: *choice ext*) **by** *metis*

lemma *image-set-plus*:
assumes ‹*linear U*›
shows ‹*U* ‘ (*A* + *B*) = *U* ‘ *A* + *U* ‘ *B*›
unfolding *image-def set-plus-def*
using *assms* **by** (*force simp*: *linear-add*)

consts *heterogenous-identity* :: ‹'*a* ⇒ '*b*›
overloading *heterogenous-identity-id* ≡ *heterogenous-identity* :: '*a* ⇒ '*a* **begin**
definition *heterogenous-identity-def*[*simp*]: ‹*heterogenous-identity-id* = *id*›
end

lemma *L2-set-mono2*:
assumes *a1*: *finite K* **and** *a2*: $K \leq L$
shows *L2-set f K* ≤ *L2-set f L*
proof−
have $(\sum i{\in}K.\ (f\ i)^2) \leq (\sum i{\in}L.\ (f\ i)^2)$
apply (*rule sum-mono2*)
using *assms* **by** *auto*
hence *sqrt* $(\sum i{\in}K.\ (f\ i)^2) \leq$ *sqrt* $(\sum i{\in}L.\ (f\ i)^2)$
by (*rule real-sqrt-le-mono*)

```
  thus ?thesis
    unfolding L2-set-def.
qed

lemma Sup-real-close:
  fixes e :: real
  assumes 0 < e
    and S: bdd-above S S ≠ {}
  shows ∃x∈S. Sup S − e < x
proof −
  have ‹Sup (ereal ' S) ≠ ∞›
   by (metis assms(2) bdd-above-def ereal-less-eq(3) less-SUP-iff less-ereal.simps(4)
not-le)
  moreover have ‹Sup (ereal ' S) ≠ −∞›
    by (simp add: SUP-eq-iff assms(3))
  ultimately have Sup-bdd: ‹|Sup (ereal ' S)| ≠ ∞›
    by auto
  then have ‹∃x'∈ereal ' S. Sup (ereal ' S) − ereal e < x'›
    apply (rule-tac Sup-ereal-close)
    using assms by auto
  then obtain x where ‹x ∈ S› and Sup-x: ‹Sup (ereal ' S) − ereal e < ereal x›
    by auto
  have ‹Sup (ereal ' S) = ereal (Sup S)›
    using Sup-bdd by (rule ereal-Sup[symmetric])
  with Sup-x have ‹ereal (Sup S − e) < ereal x›
    by auto
  then have ‹Sup S − e < x›
    by auto
  with ‹x ∈ S› show ?thesis
    by auto
qed
```

Improved version of *internalize-sort*: It is not necessary to specify the sort of the type variable.

```
attribute-setup internalize-sort' = ‹let
fun find-tvar thm v = let
  val tvars = Term.add-tvars (Thm.prop-of thm) []
  val tv = case find-first (fn (n,sort) => n=v) tvars of
                  SOME tv => tv | NONE => raise THM (Type variable ^
string-of-indexname v ^ not found, 0, [thm])
in
TVar tv
end

fun internalize-sort-attr (tvar:indexname) =
  Thm.rule-attribute [] (fn context => fn thm =>
    (snd (Internalize-Sort.internalize-sort (Thm.ctyp-of (Context.proof-of context)
(find-tvar thm tvar)) thm)));
in
```

Scan.lift Args.var >> internalize-sort-attr
end›
internalize a sort

lemma *card-prod-omega*: ‹*X* ∗*c natLeq* =*o X*› **if** ‹*Cinfinite X*›
by (*simp add*: *Cinfinite-Cnotzero cprod-infinite1' natLeq-Card-order natLeq-cinfinite natLeq-ordLeq-cinfinite that*)

lemma *countable-leq-natLeq*: ‹|*X*| ≤*o natLeq*› **if** ‹*countable X*›
using *subset-range-from-nat-into*[*OF that*]
by (*meson card-of-nat ordIso-iff-ordLeq ordLeq-transitive surj-imp-ordLeq*)

lemma *set-Times-plus-distrib*: ‹(*A* × *B*) + (*C* × *D*) = (*A* + *C*) × (*B* + *D*)›
by (*auto simp*: *Sigma-def set-plus-def*)

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-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

2.2 Not singleton

class *not-singleton* =
assumes *not-singleton-card*: ∃ *x y*. *x* ≠ *y*

lemma *not-singleton-existence*[*simp*]:
‹∃ *x*::('*a*::*not-singleton*). *x* ≠ *t*›

using *not-singleton-card*[**where** *?'a* = *'a*] **by** (*metis* (*full-types*))

lemma *not-not-singleton-zero*:
‹*x* = *0*› **if** ‹¬ *class.not-singleton TYPE*('*a*)› **for** *x* :: ‹'*a*::*zero*›
using *that* **unfolding** *class.not-singleton-def* **by** *auto*

lemma *UNIV-not-singleton*[*simp*]: (*UNIV*::-::*not-singleton set*) ≠ {*x*}
using *not-singleton-existence*[*of x*] **by** *blast*

lemma *UNIV-not-singleton-converse*:
assumes⋀*x*::'*a*. *UNIV* ≠ {*x*}
shows ∃ *x*::'*a*. ∃ *y*. *x* ≠ *y*
using *assms*
by *fastforce*

subclass (**in** *card2*) *not-singleton*
apply *standard* **using** *two-le-card*
by (*meson card-2-iff' obtain-subset-with-card-n*)

subclass (**in** *perfect-space*) *not-singleton*
apply *intro-classes*
by (*metis* (*mono-tags*) *Collect-cong Collect-mem-eq UNIV-I local.UNIV-not-singleton local.not-open-singleton local.open-subopen*)

lemma *class-not-singletonI-monoid-add*:
assumes (*UNIV*::'*a set*) ≠ {*0*}
shows *class.not-singleton TYPE*('*a*::*monoid-add*)
proof *intro-classes*
let *?univ* = *UNIV* :: '*a set*
from *assms* **obtain** *x*::'*a* **where** *x* ≠ *0*
by *auto*
thus ∃ *x y* :: '*a*. *x* ≠ *y*
by *auto*
qed

lemma *not-singleton-vs-CARD-1*:
assumes ‹¬ *class.not-singleton TYPE*('*a*)›
shows ‹*class.CARD-1 TYPE*('*a*)›
using *assms* **unfolding** *class.not-singleton-def class.CARD-1-def*
by (*metis* (*full-types*) *One-nat-def UNIV-I card.empty card.insert empty-iff equalityI finite.intros*(*1*) *insert-iff subsetI*)

2.3 *CARD-1*

context *CARD-1* **begin**

lemma *everything-the-same*[*simp*]: (*x*::'*a*)=*y*
by (*metis* (*full-types*) *UNIV-I card-1-singletonE empty-iff insert-iff local.CARD-1*)

lemma *CARD-1-UNIV*: *UNIV* = {*x*::'*a*}
by (*metis* (*full-types*) *UNIV-I card-1-singletonE local.CARD-1 singletonD*)

lemma *CARD-1-ext*: *x* (*a*::'*a*) = *y b* ⟹ *x* = *y*
proof (*rule ext*)
show *x t* = *y t*
if *x a* = *y b*
for *t* :: '*a*
using *that* **apply** (*subst* (*asm*) *everything-the-same*[**where** *x*=*a*])
apply (*subst* (*asm*) *everything-the-same*[**where** *x*=*b*])
by *simp*
qed

end

instance *unit* :: *CARD-1*
apply *standard* **by** *auto*

instance *prod* :: (*CARD-1*, *CARD-1*) *CARD-1*
apply *intro-classes*
by (*simp add*: *CARD-1*)

instance *fun* :: (*CARD-1*, *CARD-1*) *CARD-1*
apply *intro-classes*
by (*auto simp add*: *card-fun CARD-1*)

lemma *enum-CARD-1*: (*Enum.enum* :: '*a*::{*CARD-1*,*enum*} *list*) = [*a*]
proof −
let *?enum* = *Enum.enum* :: '*a*::{*CARD-1*,*enum*} *list*
have *length ?enum* = *1*
apply (*subst card-UNIV-length-enum*[*symmetric*])
by (*rule CARD-1*)
then obtain *b* **where** *?enum* = [*b*]
apply *atomize-elim*
apply (*cases ?enum*, *auto*)
by (*metis length-0-conv length-Cons nat.inject*)
thus *?enum* = [*a*]
by (*subst everything-the-same*[*of* - *b*], *simp*)
qed

lemma *card-not-singleton*: ‹*CARD*('*a*::*not-singleton*) ≠ *1*›
by (*simp add*: *card-1-singleton-iff*)

2.4 Topology

lemma *cauchy-filter-metricI*:
fixes *F* :: '*a*::*metric-space filter*
assumes ⋀*e*. *e*>*0* ⟹ ∃ *P*. *eventually P F* ∧ (∀ *x y*. *P x* ∧ *P y* ⟶ *dist x y* <

e)
shows *cauchy-filter F*
proof (*unfold cauchy-filter-def le-filter-def*, *auto*)
fix $P :: 'a \times 'a \Rightarrow bool$
assume *eventually P uniformity*
then obtain *e* **where** *e*: $e > 0$ **and** *P*: $dist\ x\ y < e \Longrightarrow P\ (x, y)$ **for** *x y*
unfolding *eventually-uniformity-metric* **by** *auto*

obtain P' **where** *evP'*: *eventually* $P'\ F$ **and** P'*-dist*: $P'\ x \wedge P'\ y \Longrightarrow dist\ x\ y < e$ **for** *x y*
apply *atomize-elim* **using** *assms e* **by** *auto*

from *evP' P'-dist P*
show *eventually P* $(F \times_F F)$
unfolding *eventually-uniformity-metric eventually-prod-filter eventually-filtermap*
by *metis*
qed

lemma *cauchy-filter-metric-filtermapI*:
fixes *F* :: *'a filter* **and** *f* :: *'a⇒'b::metric-space*
assumes $\bigwedge e.\ e{>}0 \Longrightarrow \exists P.\ eventually\ P\ F \wedge (\forall x\ y.\ P\ x \wedge P\ y \longrightarrow dist\ (f\ x)\ (f\ y) < e)$
shows *cauchy-filter* (*filtermap f F*)
proof (*rule cauchy-filter-metricI*)
fix *e* :: *real* **assume** *e*: $e > 0$
with *assms* **obtain** *P* **where** *evP*: *eventually P F* **and** *dist*: $P\ x \wedge P\ y \Longrightarrow dist\ (f\ x)\ (f\ y) < e$ **for** *x y*
by *atomize-elim auto*
define P' **where** $P'\ y = (\exists x.\ P\ x \wedge y = f\ x)$ **for** *y*
have *eventually* P' (*filtermap f F*)
unfolding *eventually-filtermap P'-def*
using *evP*
by (*smt eventually-mono*)
moreover have $P'\ x \wedge P'\ y \longrightarrow dist\ x\ y < e$ **for** *x y*
unfolding P'*-def* **using** *dist* **by** *metis*
ultimately show $\exists P.\ eventually\ P\ (filtermap\ f\ F) \wedge (\forall x\ y.\ P\ x \wedge P\ y \longrightarrow dist\ x\ y < e)$
by *auto*
qed

lemma *tendsto-add-const-iff*:
— This is a generalization of *Limits.tendsto-add-const-iff*, the only difference is that the sort here is more general.
$((\lambda x.\ c + f\ x :: 'a{::}topological\text{-}group\text{-}add) \longrightarrow c + d)\ F \longleftrightarrow (f \longrightarrow d)\ F$
using *tendsto-add*[*OF tendsto-const*[*of c*], *of f d*]
and *tendsto-add*[*OF tendsto-const*[*of* $-c$], *of* $\lambda x.\ c + f\ x\ c + d$] **by** *auto*

lemma *finite-subsets-at-top-minus*:

assumes $A \subseteq B$
shows *finite-subsets-at-top* $(B - A) \leq$ *filtermap* $(\lambda F.\ F - A)$ (*finite-subsets-at-top* B)
proof (*rule filter-leI*)
fix P **assume** *eventually* P (*filtermap* $(\lambda F.\ F - A)$ (*finite-subsets-at-top* B))
then obtain X **where** *finite* X **and** $X \subseteq B$
and P: *finite* $Y \wedge X \subseteq Y \wedge Y \subseteq B \longrightarrow P\ (Y - A)$ **for** Y
unfolding *eventually-filtermap eventually-finite-subsets-at-top* **by** *auto*

hence *finite* $(X - A)$ **and** $X - A \subseteq B - A$
by *auto*
moreover have *finite* $Y \wedge X - A \subseteq Y \wedge Y \subseteq B - A \longrightarrow P\ Y$ **for** Y
using P[**where** $Y = Y \cup X$] ‹*finite* X› ‹$X \subseteq B$›
by (*metis Diff-subset Int-Diff Un-Diff finite-Un inf.orderE le-sup-iff sup.orderE sup-ge2*)
ultimately show *eventually* P (*finite-subsets-at-top* $(B - A)$)
unfolding *eventually-finite-subsets-at-top* **by** *meson*
qed

lemma *finite-subsets-at-top-inter*:
assumes $A \subseteq B$
shows *filtermap* $(\lambda F.\ F \cap A)$ (*finite-subsets-at-top* B) = *finite-subsets-at-top* A
proof (*subst filter-eq-iff*, *intro allI iffI*)
fix $P :: 'a\ set \Rightarrow bool$
assume *eventually* P (*finite-subsets-at-top* A)
then show *eventually* P (*filtermap* $(\lambda F.\ F \cap A)$ (*finite-subsets-at-top* B))
unfolding *eventually-filtermap*
unfolding *eventually-finite-subsets-at-top*
by (*metis Int-subset-iff assms finite-Int inf-le2 subset-trans*)
next
fix $P :: 'a\ set \Rightarrow bool$
assume *eventually* P (*filtermap* $(\lambda F.\ F \cap A)$ (*finite-subsets-at-top* B))
then obtain X **where** ‹*finite* X› ‹$X \subseteq B$› **and** P: ‹*finite* $Y \Longrightarrow X \subseteq Y \Longrightarrow Y \subseteq B \Longrightarrow P\ (Y \cap A)$› **for** Y
unfolding *eventually-filtermap eventually-finite-subsets-at-top* **by** *metis*
have $*$: ‹*finite* $Y \Longrightarrow X \cap A \subseteq Y \Longrightarrow Y \subseteq A \Longrightarrow P\ Y$› **for** Y
using P[**where** $Y = ‹Y \cup (B - A)›$]
apply (*subgoal-tac* ‹$(Y \cup (B - A)) \cap A = Y$›)
apply (*smt* (*verit*, *best*) *Int-Un-distrib2 Int-Un-eq*(4) *P Un-subset-iff* ‹$X \subseteq B$› ‹*finite* X› *assms finite-UnI inf.orderE sup-ge2*)
by *auto*
show *eventually* P (*finite-subsets-at-top* A)
unfolding *eventually-finite-subsets-at-top*
apply (*rule exI*[*of* - ‹$X \cap A$›])
by (*auto simp*: ‹*finite* X› *intro*!: $*$)
qed

lemma *tendsto-principal-singleton*:
shows $(f \longrightarrow f\ x)$ (*principal* $\{x\}$)

unfolding *tendsto-def eventually-principal* **by** *simp*

lemma *complete-singleton*:
complete {*s*::*'a*::*uniform-space*}
proof−
have $F \leq$ *principal* $\{s\} \Longrightarrow$
$F \neq$ *bot* $\Longrightarrow$ *cauchy-filter* $F \Longrightarrow F \leq$ *nhds* s **for** F
by (*metis eventually-nhds eventually-principal le-filter-def singletonD*)
thus *?thesis*
unfolding *complete-uniform*
by *simp*
qed

lemma *on-closure-eqI*:
fixes $f\ g$ :: ‹*'a*::*topological-space* ⇒ *'b*::*t2-space*›
assumes *eq*: ‹$\bigwedge x.\ x \in S \Longrightarrow f\ x = g\ x$›
assumes *xS*: ‹$x \in$ *closure* S›
assumes *cont*: ‹*continuous-on UNIV f*› ‹*continuous-on UNIV g*›
shows ‹$f\ x = g\ x$›
proof −
define X **where** ‹$X = \{x.\ f\ x = g\ x\}$›
have ‹*closed X*›
using *cont* **by** (*simp add*: *X-def closed-Collect-eq*)
moreover have ‹$S \subseteq X$›
by (*simp add*: *X-def eq subsetI*)
ultimately have ‹*closure* $S \subseteq X$›
using *closure-minimal* **by** *blast*
with *xS* **have** ‹$x \in X$›
by *auto*
then show *?thesis*
using *X-def* **by** *blast*
qed

lemma *on-closure-leI*:
fixes $f\ g$ :: ‹*'a*::*topological-space* ⇒ *'b*::*linorder-topology*›
assumes *eq*: ‹$\bigwedge x.\ x \in S \Longrightarrow f\ x \leq g\ x$›
assumes *xS*: ‹$x \in$ *closure* S›
assumes *cont*: ‹*continuous-on UNIV f*› ‹*continuous-on UNIV g*›
shows ‹$f\ x \leq g\ x$›
proof −
define X **where** ‹$X = \{x.\ f\ x \leq g\ x\}$›
have ‹*closed X*›
using *cont* **by** (*simp add*: *X-def closed-Collect-le*)
moreover have ‹$S \subseteq X$›
by (*simp add*: *X-def eq subsetI*)
ultimately have ‹*closure* $S \subseteq X$›
using *closure-minimal* **by** *blast*
with *xS* **have** ‹$x \in X$›
by *auto*

then show *?thesis*
using *X-def* **by** *blast*
qed

lemma *tendsto-compose-at-within*:
assumes *f*: $(f \longrightarrow y)\ F$ **and** *g*: $(g \longrightarrow z)\ (at\ y\ within\ S)$
and *fg*: *eventually* $(\lambda w.\ f\ w = y \longrightarrow g\ y = z)\ F$
and *fS*: $\langle \forall_F\ w\ in\ F.\ f\ w \in S \rangle$
shows $((g \circ f) \longrightarrow z)\ F$
proof (*cases* $\langle g\ y = z \rangle$)
case *False*
then have *1*: $(\forall_F\ a\ in\ F.\ f\ a \neq y)$
using *fg* **by** *force*
have *2*: $(g \longrightarrow z)\ (filtermap\ f\ F) \vee \neg\ (\forall_F\ a\ in\ F.\ f\ a \neq y)$
by (*smt* (*verit*, *best*) *eventually-elim2 f fS filterlim-at filterlim-def g tendsto-mono*)
show *?thesis*
using *1 2 tendsto-compose-filtermap* **by** *blast*
next
case *True*
have $*$: *?thesis* **if** $\langle (g \longrightarrow z)\ (filtermap\ f\ F) \rangle$
using *that* **by** (*simp add*: *tendsto-compose-filtermap*)
from *g*
have $\langle (g \longrightarrow g\ y)\ (inf\ (nhds\ y)\ (principal\ (S - \{y\}))) \rangle$
by (*simp add*: *True at-within-def*)
then have g': $\langle (g \longrightarrow g\ y)\ (inf\ (nhds\ y)\ (principal\ S)) \rangle$
using *True g tendsto-at-iff-tendsto-nhds-within* **by** *blast*
from *f* **have** $\langle filterlim\ f\ (nhds\ y)\ F \rangle$
by −
then have f': $\langle filterlim\ f\ (inf\ (nhds\ y)\ (principal\ S))\ F \rangle$
using *fS*
by (*simp add*: *filterlim-inf filterlim-principal*)
from f' g' **show** *?thesis*
by (*simp add*: $*$ *True filterlim-compose filterlim-filtermap*)
qed

2.5 Sums

lemma *sum-single*:
assumes *finite A*
assumes $\bigwedge j.\ j \neq i \Longrightarrow j \in A \Longrightarrow f\ j = 0$
shows *sum f A* = (*if* $i \in A$ *then f i else 0*)
apply (*subst sum.mono-neutral-cong-right*[**where** $S = \langle A \cap \{i\} \rangle$ **and** $h = f$])
using *assms* **by** *auto*

lemma *has-sum-comm-additive-general*:
— This is a strengthening of *has-sum-comm-additive-general*.
fixes $f :: \langle 'b :: \{comm\text{-}monoid\text{-}add, topological\text{-}space\} \Rightarrow 'c :: \{comm\text{-}monoid\text{-}add, topological\text{-}space\} \rangle$

assumes *f-sum*: ‹$\bigwedge F$. *finite* $F \Longrightarrow F \subseteq S \Longrightarrow$ *sum* $(f \circ g)$ $F = f$ $(sum\ g\ F)$›
— Not using *additive* because it would add sort constraint *ab-group-add*
assumes *inS*: ‹$\bigwedge F$. *finite* $F \Longrightarrow$ *sum* g $F \in T$›
assumes *cont*: ‹$(f \longrightarrow f\ x)$ $(at\ x\ within\ T)$›
— For *t2-space* and $T = UNIV$, this is equivalent to *isCont f x* by *isCont-def*.
assumes *infsum*: ‹$(g\ has\text{-}sum\ x)$ S›
shows ‹$((f \circ g)\ has\text{-}sum\ (f\ x))$ S›
proof −
have ‹$(sum\ g \longrightarrow x)$ $(finite\text{-}subsets\text{-}at\text{-}top\ S)$›
using *infsum has-sum-def* **by** *blast*
then have ‹$((f \circ sum\ g) \longrightarrow f\ x)$ $(finite\text{-}subsets\text{-}at\text{-}top\ S)$›
apply (*rule tendsto-compose-at-within*[**where** $S{=}T$])
using *assms* **by** *auto*
then have ‹$(sum\ (f \circ g) \longrightarrow f\ x)$ $(finite\text{-}subsets\text{-}at\text{-}top\ S)$›
apply (*rule tendsto-cong*[*THEN iffD1*, *rotated*])
using *f-sum* **by** *fastforce*
then show ‹$((f \circ g)\ has\text{-}sum\ (f\ x))$ S›
using *has-sum-def* **by** *blast*
qed

lemma *summable-on-comm-additive-general*:
— This is a strengthening of *summable-on-comm-additive-general*.
fixes g :: ‹$'a \Rightarrow 'b$ :: {*comm-monoid-add*,*topological-space*}› **and** f :: ‹$'b \Rightarrow 'c$ :: {*comm-monoid-add*,*topological-space*}›
assumes ‹$\bigwedge F$. *finite* $F \Longrightarrow F \subseteq S \Longrightarrow$ *sum* $(f \circ g)$ $F = f$ $(sum\ g\ F)$›
— Not using *additive* because it would add sort constraint *ab-group-add*
assumes *inS*: ‹$\bigwedge F$. *finite* $F \Longrightarrow$ *sum* g $F \in T$›
assumes *cont*: ‹$\bigwedge x$. $(g\ has\text{-}sum\ x)$ $S \Longrightarrow (f \longrightarrow f\ x)$ $(at\ x\ within\ T)$›
— For *t2-space* and $T = UNIV$, this is equivalent to *isCont f x* by *isCont-def*.
assumes ‹g *summable-on* S›
shows ‹$(f \circ g)$ *summable-on* S›
by (*meson assms summable-on-def has-sum-comm-additive-general has-sum-def infsum-tendsto*)

lemma *has-sum-metric*:
fixes l :: ‹$'a$ :: {*metric-space*, *comm-monoid-add*}›
shows ‹$(f\ has\text{-}sum\ l)$ $A \longleftrightarrow (\forall e.\ e > 0 \longrightarrow (\exists X.\ finite\ X \wedge X \subseteq A \wedge (\forall Y.\ finite\ Y \wedge X \subseteq Y \wedge Y \subseteq A \longrightarrow dist\ (sum\ f\ Y)\ l < e)))$›
unfolding *has-sum-def*
apply (*subst tendsto-iff*)
unfolding *eventually-finite-subsets-at-top*
by *simp*

lemma *summable-on-product-finite-left*:
fixes f :: ‹$'a \times 'b \Rightarrow 'c$::{*topological-comm-monoid-add*}›
assumes *sum*: ‹$\bigwedge x.\ x{\in}X \Longrightarrow (\lambda y.\ f(x,y))$ *summable-on* Y›
assumes ‹*finite* X›
shows ‹f *summable-on* $(X \times Y)$›
using ‹*finite* X› *subset-refl*[*of* X]

proof (*induction rule*: *finite-subset-induct'*)
case *empty*
then show *?case*
by *simp*
next
case (*insert x F*)
have ∗: ‹*bij-betw* (*Pair x*) *Y* ({*x*} × *Y*)›
apply (*rule bij-betwI'*)
by *auto*
from *sum*[*of x*]
have ‹*f summable-on* {*x*} × *Y*›
apply (*rule summable-on-reindex-bij-betw*[*THEN iffD1*, *rotated*])
by (*simp-all add*: ∗ *insert.hyps*(*2*))
then have ‹*f summable-on* {*x*} × *Y* ∪ *F* × *Y*›
apply (*rule summable-on-Un-disjoint*)
using *insert* **by** *auto*
then show *?case*
by (*metis Sigma-Un-distrib1 insert-is-Un*)
qed

lemma *summable-on-product-finite-right*:
fixes *f* :: ‹'*a*×'*b* ⇒ '*c*::{*topological-comm-monoid-add*}›
assumes *sum*: ‹⋀*y*. *y*∈*Y* ⟹ (λ*x*. *f*(*x*,*y*)) *summable-on X*›
assumes ‹*finite Y*›
shows ‹*f summable-on* (*X*×*Y*)›
proof −
have ‹(λ(*y*,*x*). *f*(*x*,*y*)) *summable-on* (*Y*×*X*)›
apply (*rule summable-on-product-finite-left*)
using *assms* **by** *auto*
then show *?thesis*
apply (*subst summable-on-reindex-bij-betw*[**where** *g*=*prod.swap* **and** *A*=‹*Y*×*X*›, *symmetric*])
apply (*simp add*: *bij-betw-def product-swap*)
by (*metis* (*mono-tags*, *lifting*) *case-prod-unfold prod.swap-def summable-on-cong*)
qed

2.6 Complex numbers

lemma *cmod-Re*:
assumes *x* ≥ *0*
shows *cmod x* = *Re x*
using *assms* **unfolding** *less-eq-complex-def cmod-def*
by *auto*

lemma *abs-complex-real*[*simp*]: *abs x* ∈ ℝ **for** *x* :: *complex*
by (*simp add*: *abs-complex-def*)

lemma *Im-abs*[*simp*]: *Im* (*abs x*) = *0*
using *abs-complex-real complex-is-Real-iff* **by** *blast*

lemma *cnj-x-x*: *cnj x * x = (abs x)²*
proof (*cases x*)
  show *cnj x * x = |x|²*
    if *x = Complex x1 x2*
    for *x1 :: real*
      and *x2 :: real*
    using *that*
    by (*auto simp*: *complex-cnj complex-mult abs-complex-def*
        complex-norm power2-eq-square complex-of-real-def)
qed

lemma *cnj-x-x-geq0*[*simp*]: ‹*cnj x * x ≥ 0*›
  by (*simp add*: *less-eq-complex-def*)

lemma *complex-of-real-leq-1-iff*[*iff*]: ‹*complex-of-real x ≤ 1 ⟷ x ≤ 1*›
  by (*simp add*: *less-eq-complex-def*)

lemma *x-cnj-x*: ‹*x * cnj x = (abs x)²*›
  by (*metis cnj-x-x mult.commute*)

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 ‹((λ*x*. *Re* (*f x*)) *has-sum Re a*) *A*›
    using *assms*(*1*) *has-sum-Re has-sum-cong* **by** *blast*
  moreover have ‹((λ*x*. *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-complex-def* **by** *blast*
qed

lemma *Re-mono*: $x \leq y \Longrightarrow Re\ x \leq Re\ y$
unfolding *less-eq-complex-def* **by** *simp*

lemma *comp-Im-same*: $x \leq y \Longrightarrow Im\ x = Im\ y$
unfolding *less-eq-complex-def* **by** *simp*

lemma *Re-strict-mono*: $x < y \Longrightarrow Re\ x < Re\ y$
unfolding *less-complex-def* **by** *simp*

lemma *complex-of-real-cmod*: ‹*complex-of-real* (*cmod x*) = *abs x*›
by (*simp add*: *abs-complex-def*)

lemma *less-eq-complexI*: $Re\ x \leq Re\ y \Longrightarrow Im\ x = Im\ y \Longrightarrow x \leq y$ **unfolding** *less-eq-complex-def*
by *simp*
lemma *less-complexI*: $Re\ x < Re\ y \Longrightarrow Im\ x = Im\ y \Longrightarrow x < y$ **unfolding** *less-complex-def*
by *simp*

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

2.7 List indices and enum

fun *index-of* **where**
  index-of x [] = (*0*::*nat*)
| *index-of x* (*y*#*ys*) = (*if x*=*y then 0 else* (*index-of x ys* + *1*))

definition *enum-idx* (*x*::'*a*::*enum*) = *index-of x* (*enum-class.enum* :: '*a list*)

lemma *index-of-length*: *index-of x y* ≤ *length y*
  apply (*induction y*) **by** *auto*

lemma *index-of-correct*:
  assumes *x* ∈ *set y*
  shows *y* ! *index-of x y* = *x*
  using *assms* **apply** (*induction y arbitrary*: *x*)
  by *auto*

lemma *enum-idx-correct*:
  Enum.enum ! *enum-idx i* = *i*
proof−
  have *i* ∈ *set enum-class.enum*
    using *UNIV-enum* **by** *blast*
  thus *?thesis*
    unfolding *enum-idx-def*
    using *index-of-correct* **by** *metis*
qed

lemma *index-of-bound*:
  assumes *y* ≠ [] **and** *x* ∈ *set y*
  shows *index-of x y* < *length y*
  using *assms* **proof**(*induction y arbitrary*: *x*)
  case *Nil*
  thus *?case* **by** *auto*
next
  case (*Cons a y*)
  show *?case*
  proof(*cases a* = *x*)
    case *True*
    thus *?thesis* **by** *auto*
  next
    case *False*
    moreover have *a* ≠ *x* ⟹ *index-of x y* < *length y*
      using *Cons.IH Cons.prems*(*2*) **by** *fastforce*
    ultimately show *?thesis* **by** *auto*
  qed
qed

lemma *enum-idx-bound*[*simp*]: *enum-idx x* < *CARD*('*a*) **for** *x* :: '*a*::*enum*
proof−
  have *p1*: *False*

if (*Enum.enum* :: *'a list*) = []
proof−
have (*UNIV*::*'a set*) = *set* ([]::*'a list*)
using *that UNIV-enum* **by** *metis*
also have ... = {}
by *blast*
finally have (*UNIV*::*'a set*) = {}**.**
thus *?thesis* **by** *simp*
qed
have *p2*: *x* ∈ *set* (*Enum.enum* :: *'a list*)
using *UNIV-enum* **by** *auto*
moreover have (*enum-class.enum*::*'a list*) ≠ []
using *p2* **by** *auto*
ultimately show *?thesis*
unfolding *enum-idx-def card-UNIV-length-enum*
using *index-of-bound* [**where** *x* = *x* **and** *y* = (*Enum.enum* :: *'a list*)]
by *auto*
qed

lemma *index-of-nth*:
assumes *distinct xs*
assumes *i* < *length xs*
shows *index-of* (*xs* ! *i*) *xs* = *i*
using *assms*
by (*metis gr-implies-not-zero index-of-bound index-of-correct length-0-conv nth-eq-iff-index-eq nth-mem*)

lemma *enum-idx-enum*:
assumes ‹*i* < *CARD*(*'a*::*enum*)›
shows ‹*enum-idx* (*enum-class.enum* ! *i* :: *'a*) = *i*›
unfolding *enum-idx-def* **apply** (*rule index-of-nth*)
using *assms* **by** (*simp-all add*: *card-UNIV-length-enum enum-distinct*)

2.8 Filtering lists/sets

lemma *map-filter-map*: *List.map-filter f* (*map g l*) = *List.map-filter* (*f o g*) *l*
by (*induction l*) (*simp-all split*: *option.split*)

lemma *map-filter-Some* [*simp*]: *List.map-filter* (λ*x*. *Some* (*f x*)) *l* = *map f l*
by (*simp add*: *List.map-filter-def*)

lemma *filter-Un*: *Set.filter f* (*x* ∪ *y*) = *Set.filter f x* ∪ *Set.filter f y*
by *auto*

lemma *Set-filter-unchanged*: *Set.filter P X* = *X* **if** ⋀*x*. *x*∈*X* ⟹ *P x* **for** *P* **and** *X* :: *'z set*
using *that* **by** *auto*

2.9 Maps

definition *inj-map* $\pi = (\forall\, x\ y.\ \pi\ x = \pi\ y \wedge \pi\ x \neq None \longrightarrow x = y)$

definition *inv-map* $\pi = (\lambda y.\ if\ Some\ y \in range\ \pi\ then\ Some\ (inv\ \pi\ (Some\ y))\ else\ None)$

lemma *inj-map-total*[*simp*]: *inj-map* (*Some* $o\ \pi$) = *inj* π
  unfolding *inj-map-def inj-def* **by** *simp*

lemma *inj-map-Some*[*simp*]: *inj-map Some*
  by (*simp add*: *inj-map-def*)

lemma *inv-map-total*:
  assumes *surj* π
  shows *inv-map* (*Some* $o\ \pi$) = *Some* o *inv* π
proof−
  have (*if Some* $y \in$ *range* (λx. *Some* ($\pi\ x$))
        then Some (*SOME x*. *Some* ($\pi\ x$) = *Some y*)
        else None) =
       Some (*SOME b*. $\pi\ b = y$)
    if *surj* π
    for y
    using *that* **by** *auto*
  hence *surj* $\pi \Longrightarrow$
    (λy. *if Some* $y \in$ *range* (λx. *Some* ($\pi\ x$))
      then Some (*SOME x*. *Some* ($\pi\ x$) = *Some y*) *else None*) =
    (λx. *Some* (*SOME xa*. $\pi\ xa = x$))
    by (*rule ext*)
  thus *?thesis*
    unfolding *inv-map-def o-def inv-def*
    using *assms* **by** *linarith*
qed

lemma *inj-map-map-comp*[*simp*]:
  assumes *a1*: *inj-map f* **and** *a2*: *inj-map g*
  shows *inj-map* ($f \circ_m g$)
  using *a1 a2*
  unfolding *inj-map-def*
  by (*metis* (*mono-tags*, *lifting*) *map-comp-def option.case-eq-if option.expand*)

lemma *inj-map-inv-map*[*simp*]: *inj-map* (*inv-map* π)
proof (*unfold inj-map-def*, *rule allI*, *rule allI*, *rule impI*, *erule conjE*)
  fix $x\ y$
  assume *same*: *inv-map* $\pi\ x$ = *inv-map* $\pi\ y$
    and *pix-not-None*: *inv-map* $\pi\ x \neq None$
  have *x-pi*: *Some* $x \in$ *range* π
    using *pix-not-None* **unfolding** *inv-map-def* **apply** *auto*
    by (*meson option.distinct*(*1*))
  have *y-pi*: *Some* $y \in$ *range* π

using *pix-not-None* **unfolding** *same* **unfolding** *inv-map-def* **apply** *auto*
by (*meson option.distinct*(*1*))
have *inv-map* π x = *Some* (*Hilbert-Choice.inv* π (*Some* x))
unfolding *inv-map-def* **using** *x-pi* **by** *simp*
moreover have *inv-map* π y = *Some* (*Hilbert-Choice.inv* π (*Some* y))
unfolding *inv-map-def* **using** *y-pi* **by** *simp*
ultimately have *Hilbert-Choice.inv* π (*Some* x) = *Hilbert-Choice.inv* π (*Some* y)
using *same* **by** *simp*
thus $x = y$
by (*meson inv-into-injective option.inject x-pi y-pi*)
qed

2.10 Lattices

unbundle *lattice-syntax*

The following lemma is identical to *Complete-Lattices.uminus-Inf* except for the more general sort.

lemma *uminus-Inf*: $-(\bigsqcap A) = \bigsqcup(\mathit{uminus}\ `\ A)$ **for** A :: ‹'*a*::*complete-orthocomplemented-lattice set*›
proof (*rule order.antisym*)
show $-\bigsqcap A \leq \bigsqcup(\mathit{uminus}\ `\ A)$
by (*rule compl-le-swap2*, *rule Inf-greatest*, *rule compl-le-swap2*, *rule Sup-upper*) *simp*
show $\bigsqcup(\mathit{uminus}\ `\ A) \leq -\bigsqcap A$
by (*rule Sup-least*, *rule compl-le-swap1*, *rule Inf-lower*) *auto*
qed

The following lemma is identical to *Complete-Lattices.uminus-INF* except for the more general sort.

lemma *uminus-INF*: $-(\mathit{INF}\ x{\in}A.\ B\ x) = (\mathit{SUP}\ x{\in}A.\ -\ B\ x)$ **for** B :: ‹'*a* ⇒ '*b*::*complete-orthocomplemented-lattice*›
by (*simp add*: *uminus-Inf image-image*)

The following lemma is identical to *Complete-Lattices.uminus-Sup* except for the more general sort.

lemma *uminus-Sup*: $-(\bigsqcup A) = \bigsqcap(\mathit{uminus}\ `\ A)$ **for** A :: ‹'*a*::*complete-orthocomplemented-lattice set*›
by (*metis* (*no-types*, *lifting*) *uminus-INF image-cong image-ident ortho-involution*)

The following lemma is identical to *Complete-Lattices.uminus-SUP* except for the more general sort.

lemma *uminus-SUP*: $-(\mathit{SUP}\ x{\in}A.\ B\ x) = (\mathit{INF}\ x{\in}A.\ -\ B\ x)$ **for** B :: ‹'*a* ⇒ '*b*::*complete-orthocomplemented-lattice*›
by (*simp add*: *uminus-Sup image-image*)

lemma *has-sumI-metric*:

fixes $l :: \langle 'a :: \{metric\text{-}space, comm\text{-}monoid\text{-}add\}\rangle$
assumes $\langle \bigwedge e.\ e > 0 \Longrightarrow \exists X.\ finite\ X \wedge X \subseteq A \wedge (\forall Y.\ finite\ Y \wedge X \subseteq Y \wedge Y \subseteq A \longrightarrow dist\ (sum\ f\ Y)\ l < e)\rangle$
shows $\langle (f\ has\text{-}sum\ l)\ A\rangle$
unfolding *has-sum-metric* **using** *assms* **by** *simp*

lemma *limitin-pullback-topology*:
$\langle limitin\ (pullback\text{-}topology\ A\ g\ T)\ f\ l\ F \longleftrightarrow l \in A \wedge (\forall_F\ x\ in\ F.\ f\ x \in A) \wedge limitin\ T\ (g\ o\ f)\ (g\ l)\ F\rangle$
apply (*simp add*: *topspace-pullback-topology limitin-def openin-pullback-topology imp-ex flip*: *ex-simps*(*1*))
apply *rule*
apply *simp*
apply *safe*
using *eventually-mono* **apply** *fastforce*
apply (*simp add*: *eventually-conj-iff*)
by (*simp add*: *eventually-conj-iff*)

lemma *tendsto-coordinatewise*: $\langle (f \longrightarrow l)\ F \longleftrightarrow (\forall x.\ ((\lambda i.\ f\ i\ x) \longrightarrow l\ x)\ F)\rangle$
proof (*intro iffI allI*)
assume *asm*: $\langle (f \longrightarrow l)\ F\rangle$
then show $\langle ((\lambda i.\ f\ i\ x) \longrightarrow l\ x)\ F\rangle$ **for** x
apply (*rule continuous-on-tendsto-compose*[**where** *s*=*UNIV*, *rotated*])
by *auto*
next
assume *asm*: $\langle (\forall x.\ ((\lambda i.\ f\ i\ x) \longrightarrow l\ x)\ F)\rangle$
show $\langle (f \longrightarrow l)\ F\rangle$
proof (*unfold tendsto-def*, *intro allI impI*)
fix S **assume** $\langle open\ S\rangle$ **and** $\langle l \in S\rangle$
from *product-topology-open-contains-basis*[*OF* $\langle open\ S\rangle$[*unfolded open-fun-def*] $\langle l \in S\rangle$]
obtain U **where** *lU*: $\langle l \in Pi\ UNIV\ U\rangle$ **and** *openU*: $\langle \bigwedge x.\ open\ (U\ x)\rangle$ **and** *finiteD*: $\langle finite\ \{x.\ U\ x \neq UNIV\}\rangle$ **and** *US*: $\langle Pi\ UNIV\ U \subseteq S\rangle$
by (*auto simp add*: *PiE-UNIV-domain*)

define D **where** $\langle D = \{x.\ U\ x \neq UNIV\}\rangle$
with *finiteD* **have** *finiteD*: $\langle finite\ D\rangle$
by *simp*
have *PiUNIV*: $\langle t \in Pi\ UNIV\ U \longleftrightarrow (\forall x \in D.\ t\ x \in U\ x)\rangle$ **for** t
using *D-def* **by** *blast*

have *f-Ui*: $\langle \forall_F\ i\ in\ F.\ f\ i\ x \in U\ x\rangle$ **for** x
using *asm*[*rule-format*, *of x*] *openU*[*of x*]
using *lU topological-tendstoD* **by** *fastforce*

have $\langle \forall_F\ x\ in\ F.\ \forall i \in D.\ f\ x\ i \in U\ i\rangle$
using *finiteD*
proof *induction*
case *empty*

```
    then show ?case
      by simp
  next
    case (insert x F)
    with f-Ui show ?case
      by (simp add: eventually-conj-iff)
  qed

  then show ‹∀⇩F x in F. f x ∈ S›
    using US by (simp add: PiUNIV eventually-mono in-mono)
 qed
qed

lemma limitin-closure-of:
 assumes limit: ‹limitin T f c F›
 assumes in-S: ‹∀⇩F x in F. f x ∈ S›
 assumes nontrivial: ‹¬ trivial-limit F›
 shows ‹c ∈ T closure-of S›
proof (intro in-closure-of[THEN iffD2] conjI impI allI)
 from limit show ‹c ∈ topspace T›
   by (simp add: limitin-topspace)
 fix U
 assume ‹c ∈ U ∧ openin T U›
 with limit have ‹∀⇩F x in F. f x ∈ U›
   by (simp add: limitin-def)
 with in-S have ‹∀⇩F x in F. f x ∈ U ∧ f x ∈ S›
   by (simp add: eventually-frequently-simps)
 with nontrivial
 show ‹∃ y. y ∈ S ∧ y ∈ U›
   using eventually-happens' by blast
qed
```

2.11 Separating sets

```
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-separatingIx*:
— When using this as a rule, best instantiate x explicitly.
assumes ‹*separating-set P S*›
assumes ‹*P f*› **and** ‹*P g*›
assumes ‹$\bigwedge x.\ x \in S \Longrightarrow f\ x = g\ x$›
shows ‹$f\ x = g\ x$›
using *assms eq-from-separatingI* **by** *blast*

lemma *eq-from-separatingI2*:
assumes ‹*separating-set P* ($(\lambda(x,y).\ h\ x\ y)$ ‘ $(S \times T)$)›
assumes ‹*P f*› **and** ‹*P g*›
assumes ‹$\bigwedge x\ y.\ x \in S \Longrightarrow y \in T \Longrightarrow f\ (h\ x\ y) = g\ (h\ x\ y)$›
shows ‹$f = g$›
apply (*rule eq-from-separatingI*[*OF assms*(*1*)])
using *assms*(*2*−*4*) **by** *auto*

lemma *eq-from-separatingI2x*:
— When using this as a rule, best instantiate x explicitly.
assumes ‹*separating-set P* ($(\lambda(x,y).\ h\ x\ y)$ ‘ $(S \times T)$)›
assumes ‹*P f*› **and** ‹*P g*›
assumes ‹$\bigwedge x\ y.\ x \in S \Longrightarrow y \in T \Longrightarrow f\ (h\ x\ y) = g\ (h\ x\ y)$›
shows ‹$f\ x = g\ x$›
using *assms eq-from-separatingI2* **by** *blast*

lemma *separating-setI*:
assumes ‹$\bigwedge f\ g.\ P\ f \Longrightarrow P\ g \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow f\ x = g\ x) \Longrightarrow f = g$›
shows ‹*separating-set P S*›
by (*simp add*: *assms separating-set-def*)

end

3 *Extra-Vector-Spaces* − Additional facts about vector spaces

theory *Extra-Vector-Spaces*
imports
HOL−*Analysis.Inner-Product*
HOL−*Analysis.Euclidean-Space*
HOL−*Library.Indicator-Function*
HOL−*Analysis.Topology-Euclidean-Space*
HOL−*Analysis.Line-Segment*
HOL−*Analysis.Bounded-Linear-Function*
Extra-General
begin

3.1 Euclidean spaces

typedef *'a euclidean-space = UNIV :: ('a ⇒ real) set* **..**
setup-lifting *type-definition-euclidean-space*

instantiation *euclidean-space* :: (*type*) *real-vector* **begin**
lift-definition *plus-euclidean-space* ::
'a euclidean-space ⇒ 'a euclidean-space ⇒ 'a euclidean-space
is *λf g x. f x + g x* **.**
lift-definition *zero-euclidean-space* :: *'a euclidean-space* **is** *λ-. 0* **.**
lift-definition *uminus-euclidean-space* ::
'a euclidean-space ⇒ 'a euclidean-space
is *λf x. − f x* **.**
lift-definition *minus-euclidean-space* ::
'a euclidean-space ⇒ 'a euclidean-space ⇒ 'a euclidean-space
is *λf g x. f x − g x*.
lift-definition *scaleR-euclidean-space* ::
real ⇒ 'a euclidean-space ⇒ 'a euclidean-space
is *λc f x. c ∗ f x* **.**
instance
apply *intro-classes*
by (*transfer*; *auto intro*: *distrib-left distrib-right*)+
end

instantiation *euclidean-space* :: (*finite*) *real-inner* **begin**
lift-definition *inner-euclidean-space* :: *'a euclidean-space ⇒ 'a euclidean-space ⇒ real*
is *λf g. ∑x∈UNIV. f x ∗ g x :: real* **.**
definition *norm-euclidean-space* (*x::'a euclidean-space*) = *sqrt* (*inner x x*)
definition *dist-euclidean-space* (*x::'a euclidean-space*) *y* = *norm* (*x−y*)
definition *sgn x* = *x* /*R norm x* **for** *x::'a euclidean-space*
definition *uniformity* = (*INF e∈{0<..}. principal {*(*x::'a euclidean-space*, *y*). *dist x y < e}*)
definition *open U* = (*∀ x∈U. ∀F* (*x'::'a euclidean-space*, *y*) *in uniformity. x'* = *x ⟶ y ∈ U*)
instance
proof *intro-classes*
fix *x* :: *'a euclidean-space*
and *y* :: *'a euclidean-space*
and *z* :: *'a euclidean-space*
show *dist* (*x::'a euclidean-space*) *y* = *norm* (*x − y*)
and *sgn* (*x::'a euclidean-space*) = *x* /*R norm x*
and *uniformity* = (*INF e∈{0<..}. principal {*(*x*, *y*). *dist* (*x::'a euclidean-space*) *y < e}*)
and *open U* = (*∀ x∈U. ∀F* (*x'*, *y*) *in uniformity.* (*x'::'a euclidean-space*) = *x ⟶ y ∈ U*)
and *norm x* = *sqrt* (*inner x x*) **for** *U*
unfolding *dist-euclidean-space-def norm-euclidean-space-def sgn-euclidean-space-def uniformity-euclidean-space-def open-euclidean-space-def*
by *simp-all*

```
  show inner x y = inner y x
    apply transfer
    by (simp add: mult.commute)
  show inner (x + y) z = inner x z + inner y z
  proof transfer
    fix x y z::'a ⇒ real
    have (∑ i∈UNIV. (x i + y i) * z i) = (∑ i∈UNIV. x i * z i + y i * z i)
      by (simp add: distrib-left mult.commute)
    thus (∑ i∈UNIV. (x i + y i) * z i) = (∑ j∈UNIV. x j * z j) + (∑ k∈UNIV.
y k * z k)
      by (subst sum.distrib[symmetric])
  qed

  show inner (r *R x) y = r * (inner x y) for r
  proof transfer
    fix r and x y::'a⇒real
    have (∑ i∈UNIV. r * x i * y i) = (∑ i∈UNIV. r * (x i * y i))
      by (simp add: mult.assoc)
    thus (∑ i∈UNIV. r * x i * y i) = r * (∑ j∈UNIV. x j * y j)
      by (subst sum-distrib-left)
  qed
  show 0 ≤ inner x x
    apply transfer
    by (simp add: sum-nonneg)
  show (inner x x = 0) = (x = 0)
  proof (transfer, rule)
    fix f :: 'a ⇒ real
    assume (∑ i∈UNIV. f i * f i) = 0
    hence f x * f x = 0 for x
    apply (rule-tac sum-nonneg-eq-0-iff[THEN iffD1, rule-format, where A=UNIV
and x=x])
      by auto
    thus f = (λ-. 0)
      by auto
  qed auto
qed
end

instantiation euclidean-space :: (finite) euclidean-space begin
lift-definition euclidean-space-basis-vector :: 'a ⇒ 'a euclidean-space is
  λx. indicator {x} .
definition Basis-euclidean-space == (euclidean-space-basis-vector ` UNIV)
instance
proof intro-classes
  fix u :: 'a euclidean-space
    and v :: 'a euclidean-space
  show (Basis::'a euclidean-space set) ≠ {}
    unfolding Basis-euclidean-space-def by simp
```

show *finite* (*Basis*::*'a euclidean-space set*)
unfolding *Basis-euclidean-space-def* **by** *simp*
show *inner u v* = (*if u* = *v then 1 else 0*)
if *u* ∈ *Basis* **and** *v* ∈ *Basis*
using *that* **unfolding** *Basis-euclidean-space-def*
apply *transfer* **apply** *auto*
by (*auto simp*: *indicator-def*)
show (∀ *v*∈*Basis*. *inner u v* = *0*) = (*u* = *0*)
unfolding *Basis-euclidean-space-def*
apply *transfer*
by *auto*
qed
end

3.2 Misc

lemma *closure-bounded-linear-image-subset-eq*:
assumes *f*: *bounded-linear f*
shows *closure* (*f* ' *closure S*) = *closure* (*f* ' *S*)
by (*meson closed-closure closure-bounded-linear-image-subset closure-minimal closure-mono closure-subset f image-mono subset-antisym*)

lemma *not-singleton-real-normed-is-perfect-space*[*simp*]: ‹*class.perfect-space* (*open* :: *'a*::{*not-singleton*,*real-normed-vector*} *set* ⇒ *bool*)›
apply *standard*
by (*metis UNIV-not-singleton clopen closed-singleton empty-not-insert*)

lemma *infsum-bounded-linear*:
assumes ‹*bounded-linear h*›
assumes ‹*f summable-on A*›
shows ‹*infsum* (λ*x*. *h* (*f x*)) *A* = *h* (*infsum f A*)›
by (*auto intro*!: *infsum-bounded-linear-strong assms summable-on-bounded-linear*[**where** *h*=*h*])

lemma *abs-summable-on-bounded-linear*:
fixes *h f A*
assumes ‹*bounded-linear h*›
assumes ‹*f abs-summable-on A*›
shows ‹(*h o f*) *abs-summable-on A*›
proof −
have *bound*: ‹*norm* (*h* (*f x*)) ≤ *onorm h* ∗ *norm* (*f x*)› **for** *x*
apply (*rule onorm*)
by (*simp add*: *assms*(*1*))

from *assms*(*2*) **have** ‹(λ*x*. *onorm h* ∗*R f x*) *abs-summable-on A*›
by (*auto intro*!: *summable-on-cmult-right*)
then have ‹(λ*x*. *h* (*f x*)) *abs-summable-on A*›
apply (*rule abs-summable-on-comparison-test*)
using *bound* **by** (*auto simp*: *assms*(*1*) *onorm-pos-le*)

then show *?thesis*
by *auto*
qed

lemma *norm-plus-leq-norm-prod*: ‹*norm* (*a* + *b*) ≤ *sqrt 2* ∗ *norm* (*a*, *b*)›
proof −
have ‹(*norm* (*a* + *b*))^2 ≤ (*norm a* + *norm b*)^2›
using *norm-triangle-ineq* **by** *auto*
also have ‹. . . ≤ *2* ∗ ((*norm a*)^2 + (*norm b*)^2)›
by (*smt* (*verit*, *best*) *power2-sum sum-squares-bound*)
also have ‹. . . ≤ (*sqrt 2* ∗ *norm* (*a*, *b*))^2›
by (*auto simp*: *power-mult-distrib norm-prod-def simp del*: *power-mono-iff*)
finally show *?thesis*
by *auto*
qed

lemma *ex-norm1*:
assumes ‹(*UNIV*::*'a*::*real-normed-vector set*) ≠ {*0*}›
shows ‹∃ *x*::*'a*. *norm x* = *1*›
proof−
have ‹∃ *x*::*'a*. *x* ≠ *0*›
using *assms* **by** *fastforce*
then obtain *x*::*'a* **where** ‹*x* ≠ *0*›
by *blast*
hence ‹*norm x* ≠ *0*›
by *simp*
hence ‹(*norm x*) / (*norm x*) = *1*›
by *simp*
moreover have ‹(*norm x*) / (*norm x*) = *norm* (*x* /$_R$ (*norm x*))›
by *simp*
ultimately have ‹*norm* (*x* /$_R$ (*norm x*)) = *1*›
by *simp*
thus *?thesis*
by *blast*
qed

lemma *bdd-above-norm-f*:
assumes *bounded-linear f*
shows ‹*bdd-above* {*norm* (*f x*) |*x*. *norm x* = *1*}›
proof−
have ‹∃ *M*. ∀ *x*. *norm x* = *1* ⟶ *norm* (*f x*) ≤ *M*›
using *assms*
by (*metis bounded-linear.axioms*(*2*) *bounded-linear-axioms-def*)
thus *?thesis* **by** *auto*
qed

lemma *any-norm-exists*:
assumes ‹*n* ≥ *0*›
shows ‹∃ *ψ*::*'a*::{*real-normed-vector*,*not-singleton*}. *norm ψ* = *n*›

proof −
obtain $\psi :: 'a$ **where** ‹$\psi \neq 0$›
using *not-singleton-card*
by *force*
then have ‹$norm\ (n *_R sgn\ \psi) = n$›
using *assms* **by** (*auto simp*: *sgn-div-norm abs-mult*)
then show *?thesis*
by *blast*
qed

lemma *abs-summable-on-scaleR-left* [*intro*]:
fixes $c :: ‹'a :: real\text{-}normed\text{-}vector›$
assumes $c \neq 0 \Longrightarrow f\ abs\text{-}summable\text{-}on\ A$
shows $(\lambda x.\ f\ x *_R c)\ abs\text{-}summable\text{-}on\ A$
apply (*cases* ‹$c = 0$›)
apply *simp*
by (*auto intro*!: *summable-on-cmult-left assms simp flip*: *real-norm-def*)

lemma *abs-summable-on-scaleR-right* [*intro*]:
fixes $f :: ‹'a \Rightarrow 'b :: real\text{-}normed\text{-}vector›$
assumes $c \neq 0 \Longrightarrow f\ abs\text{-}summable\text{-}on\ A$
shows $(\lambda x.\ c *_R f\ x)\ abs\text{-}summable\text{-}on\ A$
apply (*cases* ‹$c = 0$›)
apply *simp*
by (*auto intro*!: *summable-on-cmult-right assms*)

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

end

4 *Extra-Ordered-Fields* − Additional facts about ordered fields

theory *Extra-Ordered-Fields*
imports *Complex-Main HOL−Library.Complex-Order*
begin

4.1 Ordered Fields

In this section we introduce some type classes for ordered rings/fields/etc. that are weakenings of existing classes. Most theorems in this section are copies of the eponymous theorems from Isabelle/HOL, except that they are now proven requiring weaker type classes (usually the need for a total order is removed).

Since the lemmas are identical to the originals except for weaker type constraints, we use the same names as for the original lemmas. (In fact, the new lemmas could replace the original ones in Isabelle/HOL with at most minor incompatibilities.

4.2 Missing from Rings.thy

The existing class *abs-if* requires $|a| = (\mathit{if}\ a < 0\ \mathit{then} - a\ \mathit{else}\ a)$. However, if $(<)$ is not a total order, this condition is too strong when a is incomparable with 0. (Namely, it requires the absolute value to be the identity on such elements. E.g., the absolute value for complex numbers does not satisfy this.) The following class *partial-abs-if* is analogous to *abs-if* but does not require anything if a is incomparable with 0.

class *partial-abs-if* = *minus* + *uminus* + *ord* + *zero* + *abs* +
  assumes *abs-neg*: $a \leq 0 \Longrightarrow \mathit{abs}\ a = -a$
  assumes *abs-pos*: $a \geq 0 \Longrightarrow \mathit{abs}\ a = a$

class *ordered-ring-strict* = *ring* + *ordered-semiring-strict*
  + *ordered-ab-group-add* + *partial-abs-if*
  — missing class analogous to *linordered-ring-strict* without requiring a total order
begin

subclass *ordered-ring* **..**

lemma *mult-strict-left-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow c * a < c * b$
  using *mult-strict-left-mono* [*of* $b\ a - c$] **by** *simp*

lemma *mult-strict-right-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow a * c < b * c$
  using *mult-strict-right-mono* [*of* $b\ a - c$] **by** *simp*

lemma *mult-neg-neg*: $a < 0 \Longrightarrow b < 0 \Longrightarrow 0 < a * b$
  using *mult-strict-right-mono-neg* [*of* $a\ 0\ b$] **by** *simp*

end

lemmas *mult-sign-intros* =
  mult-nonneg-nonneg mult-nonneg-nonpos
  mult-nonpos-nonneg mult-nonpos-nonpos
  mult-pos-pos mult-pos-neg
  mult-neg-pos mult-neg-neg

4.3 Ordered fields

class *ordered-field* = *field* + *order* + *ordered-comm-semiring-strict* + *ordered-ab-group-add*
  + *partial-abs-if*
  — missing class analogous to *linordered-field* without requiring a total order
begin

lemma *frac-less-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \,/\, y < w \,/\, z \longleftrightarrow (x * z - w * y) \,/\, (y * z) < 0$
by (*subst less-iff-diff-less-0*) (*simp add*: *diff-frac-eq*)

lemma *frac-le-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \,/\, y \leq w \,/\, z \longleftrightarrow (x * z - w * y) \,/\, (y * z) \leq 0$
by (*subst le-iff-diff-le-0*) (*simp add*: *diff-frac-eq*)

lemmas *sign-simps* = *algebra-simps zero-less-mult-iff mult-less-0-iff*

lemmas (**in** −) *sign-simps* = *algebra-simps zero-less-mult-iff mult-less-0-iff*

Simplify expressions equated with 1

lemma *zero-eq-1-divide-iff* [*simp*]: $0 = 1 \,/\, a \longleftrightarrow a = 0$
by (*cases* $a = 0$) (*auto simp*: *field-simps*)

lemma *one-divide-eq-0-iff* [*simp*]: $1 \,/\, a = 0 \longleftrightarrow a = 0$
using *zero-eq-1-divide-iff*[*of a*] **by** *simp*

Simplify expressions such as $0 < 1/x$ to $0 < x$

Simplify quotients that are compared with the value 1.

Conditional Simplification Rules: No Case Splits

lemma *eq-divide-eq-1* [*simp*]:
$(1 = b/a) = ((a \neq 0 \;\&\; a = b))$
by (*auto simp add*: *eq-divide-eq*)

lemma *divide-eq-eq-1* [*simp*]:
$(b/a = 1) = ((a \neq 0 \;\&\; a = b))$
by (*auto simp add*: *divide-eq-eq*)

end

The following type class intends to capture some important properties that are common both to the real and the complex numbers. The purpose is to be able to state and prove lemmas that apply both to the real and the complex numbers without needing to state the lemma twice.

class *nice-ordered-field* = *ordered-field* + *zero-less-one* + *idom-abs-sgn* +
assumes *positive-imp-inverse-positive*: $0 < a \Longrightarrow 0 < \mathit{inverse}\ a$
and *inverse-le-imp-le*: $\mathit{inverse}\ a \leq \mathit{inverse}\ b \Longrightarrow 0 < a \Longrightarrow b \leq a$
and *dense-le*: $(\bigwedge x.\ x < y \Longrightarrow x \leq z) \Longrightarrow y \leq z$
and *nn-comparable*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a \leq b \vee b \leq a$
and *abs-nn*: $|x| \geq 0$
begin

subclass (**in** *linordered-field*) *nice-ordered-field*
proof

```
  show |a| = - a
    if a ≤ 0
    for a :: 'a
    using that
    by simp
  show |a| = a
    if 0 ≤ a
    for a :: 'a
    using that
    by simp
  show 0 < inverse a
    if 0 < a
    for a :: 'a
    using that
    by simp
  show b ≤ a
    if inverse a ≤ inverse b
      and 0 < a
    for a :: 'a
      and b
    using that
    using local.inverse-le-imp-le by blast
  show y ≤ z
    if ⋀x::'a. x < y ⟹ x ≤ z
    for y
      and z
    using that
    using local.dense-le by blast
  show a ≤ b ∨ b ≤ a
    if 0 ≤ a
      and 0 ≤ b
    for a :: 'a
      and b
    using that
    by auto
  show 0 ≤ |x|
    for x :: 'a
    by simp
qed

lemma comparable:
  assumes h1: a ≤ c ∨ a ≥ c
    and h2: b ≤ c ∨ b ≥ c
  shows a ≤ b ∨ b ≤ a
proof-
  have a ≤ b
    if t1: ¬ b ≤ a and t2: a ≤ c and t3: b ≤ c
  proof-
    have 0 ≤ c-a
```

```
    by (simp add: t2)
  moreover have 0 ≤ c−b
    by (simp add: t3)
  ultimately have c−a ≤ c−b ∨ c−a ≥ c−b by (rule nn-comparable)
  hence −a ≤ −b ∨ −a ≥ −b
    using local.add-le-imp-le-right local.uminus-add-conv-diff by presburger
  thus ?thesis
    by (simp add: t1)
 qed
 moreover have a ≤ b
  if t1: ¬ b ≤ a and t2: c ≤ a and t3: b ≤ c
 proof−
  have b ≤ a
    using local.dual-order.trans t2 t3 by blast
  thus ?thesis
    using t1 by auto
 qed
 moreover have a ≤ b
  if t1: ¬ b ≤ a and t2: c ≤ a and t3: c ≤ b
 proof−
  have 0 ≤ a−c
    by (simp add: t2)
  moreover have 0 ≤ b−c
    by (simp add: t3)
  ultimately have a−c ≤ b−c ∨ a−c ≥ b−c by (rule nn-comparable)
  hence a ≤ b ∨ a ≥ b
    by (simp add: local.le-diff-eq)
  thus ?thesis
    by (simp add: t1)
 qed
 ultimately show ?thesis using assms by auto
qed

lemma negative-imp-inverse-negative:
 a < 0 ⟹ inverse a < 0
 by (insert positive-imp-inverse-positive [of −a],
     simp add: nonzero-inverse-minus-eq less-imp-not-eq)

lemma inverse-positive-imp-positive:
 assumes inv-gt-0: 0 < inverse a and nz: a ≠ 0
 shows 0 < a
proof −
 have 0 < inverse (inverse a)
  using inv-gt-0 by (rule positive-imp-inverse-positive)
 thus 0 < a
  using nz by (simp add: nonzero-inverse-inverse-eq)
qed

lemma inverse-negative-imp-negative:
```

assumes *inv-less-0*: *inverse a* < *0* **and** *nz*: *a* ≠ *0*
shows *a* < *0*
proof−
have *inverse* (*inverse a*) < *0*
using *inv-less-0* **by** (*rule negative-imp-inverse-negative*)
thus *a* < *0* **using** *nz* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *linordered-field-no-lb*:
∀ *x*. ∃ *y*. *y* < *x*
proof
fix *x*::′*a*
have *m1*: − (*1*::′*a*) < *0* **by** *simp*
from *add-strict-right-mono*[*OF m1*, **where** *c*=*x*]
have (− *1*) + *x* < *x* **by** *simp*
thus ∃ *y*. *y* < *x* **by** *blast*
qed

lemma *linordered-field-no-ub*:
∀ *x*. ∃ *y*. *y* > *x*
proof
fix *x*::′*a*
have *m1*: (*1*::′*a*) > *0* **by** *simp*
from *add-strict-right-mono*[*OF m1*, **where** *c*=*x*]
have *1* + *x* > *x* **by** *simp*
thus ∃ *y*. *y* > *x* **by** *blast*
qed

lemma *less-imp-inverse-less*:
assumes *less*: *a* < *b* **and** *apos*: *0* < *a*
shows *inverse b* < *inverse a*
using *assms* **by** (*metis local.dual-order.strict-iff-order*
local.inverse-inverse-eq local.inverse-le-imp-le local.positive-imp-inverse-positive)

lemma *inverse-less-imp-less*:
inverse a < *inverse b* ⟹ *0* < *a* ⟹ *b* < *a*
using *local.inverse-le-imp-le local.order.strict-iff-order* **by** *blast*

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp*]:
0 < *a* ⟹ *0* < *b* ⟹ *inverse a* < *inverse b* ⟷ *b* < *a*
by (*blast intro*: *less-imp-inverse-less dest*: *inverse-less-imp-less*)

lemma *le-imp-inverse-le*:
a ≤ *b* ⟹ *0* < *a* ⟹ *inverse b* ≤ *inverse a*
by (*force simp add*: *le-less less-imp-inverse-less*)

lemma *inverse-le-iff-le* [*simp*]:
0 < *a* ⟹ *0* < *b* ⟹ *inverse a* ≤ *inverse b* ⟷ *b* ≤ *a*

by (*blast intro*: *le-imp-inverse-le dest*: *inverse-le-imp-le*)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:
inverse a ≤ *inverse b* ⟹ *b* < *0* ⟹ *b* ≤ *a*
by (*metis local.inverse-le-imp-le local.inverse-minus-eq local.neg-0-less-iff-less local.neg-le-iff-le*)

lemma *inverse-less-imp-less-neg*:
inverse a < *inverse b* ⟹ *b* < *0* ⟹ *b* < *a*
using *local.dual-order.strict-iff-order local.inverse-le-imp-le-neg* **by** *blast*

lemma *inverse-less-iff-less-neg* [*simp*]:
a < *0* ⟹ *b* < *0* ⟹ *inverse a* < *inverse b* ⟷ *b* < *a*
by (*metis local.antisym-conv2 local.inverse-less-imp-less-neg local.negative-imp-inverse-negative local.nonzero-inverse-inverse-eq local.order.strict-implies-order*)

lemma *le-imp-inverse-le-neg*:
a ≤ *b* ⟹ *b* < *0* ⟹ *inverse b* ≤ *inverse a*
by (*force simp add*: *le-less less-imp-inverse-less-neg*)

lemma *inverse-le-iff-le-neg* [*simp*]:
a < *0* ⟹ *b* < *0* ⟹ *inverse a* ≤ *inverse b* ⟷ *b* ≤ *a*
by (*blast intro*: *le-imp-inverse-le-neg dest*: *inverse-le-imp-le-neg*)

lemma *one-less-inverse*:
0 < *a* ⟹ *a* < *1* ⟹ *1* < *inverse a*
using *less-imp-inverse-less* [*of a 1*, *unfolded inverse-1*] **.**

lemma *one-le-inverse*:
0 < *a* ⟹ *a* ≤ *1* ⟹ *1* ≤ *inverse a*
using *le-imp-inverse-le* [*of a 1*, *unfolded inverse-1*] **.**

lemma *pos-le-divide-eq* [*field-simps*]:
assumes *0* < *c*
shows *a* ≤ *b* / *c* ⟷ *a* ∗ *c* ≤ *b*
using *assms* **by** (*metis local.divide-eq-imp local.divide-inverse-commute local.dual-order.order-iff-strict local.dual-order.strict-iff-order local.mult-right-mono local.mult-strict-left-mono local.nonzero-divide-eq-eq local.order.strict-implies-order local.positive-imp-inverse-positive*)

lemma *pos-less-divide-eq* [*field-simps*]:
assumes *0* < *c*
shows *a* < *b* / *c* ⟷ *a* ∗ *c* < *b*
using *assms local.dual-order.strict-iff-order local.nonzero-divide-eq-eq local.pos-le-divide-eq*
by *auto*

lemma *neg-less-divide-eq* [*field-simps*]:

assumes $c < 0$
shows $a < b \mathbin{/} c \longleftrightarrow b < a * c$
by (*metis assms local.minus-divide-divide local.mult-minus-right local.neg-0-less-iff-less local.neg-less-iff-less local.pos-less-divide-eq*)

lemma *neg-le-divide-eq* [*field-simps*]:
assumes $c < 0$
shows $a \leq b \mathbin{/} c \longleftrightarrow b \leq a * c$
by (*metis assms local.dual-order.order-iff-strict local.dual-order.strict-iff-order local.neg-less-divide-eq local.nonzero-divide-eq-eq*)

lemma *pos-divide-le-eq* [*field-simps*]:
assumes $0 < c$
shows $b \mathbin{/} c \leq a \longleftrightarrow b \leq a * c$
by (*metis assms local.dual-order.strict-iff-order local.nonzero-eq-divide-eq local.pos-le-divide-eq*)

lemma *pos-divide-less-eq* [*field-simps*]:
assumes $0 < c$
shows $b \mathbin{/} c < a \longleftrightarrow b < a * c$
by (*metis assms local.minus-divide-left local.mult-minus-left local.neg-less-iff-less local.pos-less-divide-eq*)

lemma *neg-divide-le-eq* [*field-simps*]:
assumes $c < 0$
shows $b \mathbin{/} c \leq a \longleftrightarrow a * c \leq b$
by (*metis assms local.minus-divide-left local.mult-minus-left local.neg-le-divide-eq local.neg-le-iff-le*)

lemma *neg-divide-less-eq* [*field-simps*]:
assumes $c < 0$
shows $b \mathbin{/} c < a \longleftrightarrow a * c < b$
using *assms local.dual-order.strict-iff-order local.neg-divide-le-eq* **by** *auto*

The following *field-simps* rules are necessary, as minus is always moved atop of division but we want to get rid of division.

lemma *pos-le-minus-divide-eq* [*field-simps*]: $0 < c \Longrightarrow a \leq - (b \mathbin{/} c) \longleftrightarrow a * c \leq - b$
unfolding *minus-divide-left* **by** (*rule pos-le-divide-eq*)

lemma *neg-le-minus-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow a \leq - (b \mathbin{/} c) \longleftrightarrow - b \leq a * c$
unfolding *minus-divide-left* **by** (*rule neg-le-divide-eq*)

lemma *pos-less-minus-divide-eq* [*field-simps*]: $0 < c \Longrightarrow a < - (b \mathbin{/} c) \longleftrightarrow a * c < - b$
unfolding *minus-divide-left* **by** (*rule pos-less-divide-eq*)

lemma *neg-less-minus-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow a < - (b \mathbin{/} c) \longleftrightarrow - b$

$< a * c$
unfolding *minus-divide-left* **by** (*rule neg-less-divide-eq*)

lemma *pos-minus-divide-less-eq* [*field-simps*]: $0 < c \Longrightarrow - (b \mathbin{/} c) < a \longleftrightarrow - b < a * c$
unfolding *minus-divide-left* **by** (*rule pos-divide-less-eq*)

lemma *neg-minus-divide-less-eq* [*field-simps*]: $c < 0 \Longrightarrow - (b \mathbin{/} c) < a \longleftrightarrow a * c < - b$
unfolding *minus-divide-left* **by** (*rule neg-divide-less-eq*)

lemma *pos-minus-divide-le-eq* [*field-simps*]: $0 < c \Longrightarrow - (b \mathbin{/} c) \leq a \longleftrightarrow - b \leq a * c$
unfolding *minus-divide-left* **by** (*rule pos-divide-le-eq*)

lemma *neg-minus-divide-le-eq* [*field-simps*]: $c < 0 \Longrightarrow - (b \mathbin{/} c) \leq a \longleftrightarrow a * c \leq - b$
unfolding *minus-divide-left* **by** (*rule neg-divide-le-eq*)

lemma *frac-less-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \mathbin{/} y < w \mathbin{/} z \longleftrightarrow (x * z - w * y) \mathbin{/} (y * z) < 0$
by (*subst less-iff-diff-less-0*) (*simp add*: *diff-frac-eq*)

lemma *frac-le-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \mathbin{/} y \leq w \mathbin{/} z \longleftrightarrow (x * z - w * y) \mathbin{/} (y * z) \leq 0$
by (*subst le-iff-diff-le-0*) (*simp add*: *diff-frac-eq*)

Lemmas *sign-simps* is a first attempt to automate proofs of positivity/negativity needed for *field-simps*. Have not added *sign-simps* to *field-simps* because the former can lead to case explosions.

lemma *divide-pos-pos*[*simp*]:
$0 < x \Longrightarrow 0 < y \Longrightarrow 0 < x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-pos*:
$0 \leq x \Longrightarrow 0 < y \Longrightarrow 0 \leq x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *divide-neg-pos*:
$x < 0 \Longrightarrow 0 < y \Longrightarrow x \mathbin{/} y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonpos-pos*:
$x \leq 0 \Longrightarrow 0 < y \Longrightarrow x \mathbin{/} y \leq 0$
by(*simp add*:*field-simps*)

lemma *divide-pos-neg*:
$0 < x \Longrightarrow y < 0 \Longrightarrow x \mathbin{/} y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-neg*:
$0 \leq x \Longrightarrow y < 0 \Longrightarrow x \mathbin{/} y \leq 0$
by(*simp add*:*field-simps*)

lemma *divide-neg-neg*:
$x < 0 \Longrightarrow y < 0 \Longrightarrow 0 < x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *divide-nonpos-neg*:
$x \leq 0 \Longrightarrow y < 0 \Longrightarrow 0 \leq x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *divide-strict-right-mono*:
$a < b \Longrightarrow 0 < c \Longrightarrow a \mathbin{/} c < b \mathbin{/} c$
by (*simp add*: *less-imp-not-eq2 divide-inverse mult-strict-right-mono positive-imp-inverse-positive*)

lemma *divide-strict-right-mono-neg*:
$b < a \Longrightarrow c < 0 \Longrightarrow a \mathbin{/} c < b \mathbin{/} c$
by (*simp add*: *local.neg-less-divide-eq*)

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono*:
$b < a \Longrightarrow 0 < c \Longrightarrow 0 < a*b \Longrightarrow c \mathbin{/} a < c \mathbin{/} b$
by (*metis local.divide-neg-pos local.dual-order.strict-iff-order local.frac-less-eq local.less-iff-diff-less-0 local.mult-not-zero local.mult-strict-left-mono*)

lemma *divide-left-mono*:
$b \leq a \Longrightarrow 0 \leq c \Longrightarrow 0 < a*b \Longrightarrow c \mathbin{/} a \leq c \mathbin{/} b$
using *local.divide-cancel-left local.divide-strict-left-mono local.dual-order.order-iff-strict*
by *auto*

lemma *divide-strict-left-mono-neg*:
$a < b \Longrightarrow c < 0 \Longrightarrow 0 < a*b \Longrightarrow c \mathbin{/} a < c \mathbin{/} b$
by (*metis local.divide-strict-left-mono local.minus-divide-left local.neg-0-less-iff-less local.neg-less-iff-less mult-commute*)

lemma *mult-imp-div-pos-le*: $0 < y \Longrightarrow x \leq z * y \Longrightarrow x \mathbin{/} y \leq z$
by (*subst pos-divide-le-eq*, *assumption*+)

lemma *mult-imp-le-div-pos*: $0 < y \Longrightarrow z * y \leq x \Longrightarrow z \leq x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *mult-imp-div-pos-less*: $0 < y \Longrightarrow x < z * y \Longrightarrow x \mathbin{/} y < z$
by(*simp add*:*field-simps*)

lemma *mult-imp-less-div-pos*: $0 < y \Longrightarrow z * y < x \Longrightarrow z < x \mathbin{/} y$

by(*simp add*:*field-simps*)

lemma *frac-le*: $0 \leq x \Longrightarrow x \leq y \Longrightarrow 0 < w \Longrightarrow w \leq z \Longrightarrow x \,/\, z \leq y \,/\, w$
using *local.mult-imp-div-pos-le local.mult-imp-le-div-pos local.mult-mono* **by** *auto*

lemma *frac-less*: $0 \leq x \Longrightarrow x < y \Longrightarrow 0 < w \Longrightarrow w \leq z \Longrightarrow x \,/\, z < y \,/\, w$
proof−
assume *a1*: $w \leq z$
assume *a2*: $0 < w$
assume *a3*: $0 \leq x$
assume *a4*: $x < y$
have *f5*: $a = 0 \vee (b = c \,/\, a) = (b * a = c)$
for $a\ b\ c$::$'a$
by (*meson local.nonzero-eq-divide-eq*)
have *f6*: $0 < z$
using *a2 a1 less-le-trans* **by** *blast*
have $z \neq 0$
using *a2 a1* **by** (*meson local.leD*)
moreover have $x \,/\, z \neq y \,/\, w$
using *a1 a2 a3 a4 local.frac-eq-eq local.mult-less-le-imp-less* **by** *fastforce*
ultimately have $x \,/\, z \neq y \,/\, w$
using *f5* **by** (*metis* (*no-types*))
thus *?thesis*
using *a4 a3 a2 a1* **by** (*meson local.frac-le local.order.not-eq-order-implies-strict*
local.order.strict-implies-order)
qed

lemma *frac-less2*: $0 < x \Longrightarrow x \leq y \Longrightarrow 0 < w \Longrightarrow w < z \Longrightarrow x \,/\, z < y \,/\, w$
by (*metis local.antisym-conv2 local.divide-cancel-left local.dual-order.strict-implies-order*
local.frac-le local.frac-less)

lemma *less-half-sum*: $a < b \Longrightarrow a < (a+b) \,/\, (1+1)$
by (*metis local.add-pos-pos local.add-strict-left-mono local.mult-imp-less-div-pos*
local.semiring-normalization-rules(*4*) *local.zero-less-one mult-commute*)

lemma *gt-half-sum*: $a < b \Longrightarrow (a+b)/(1+1) < b$
by (*metis local.add-pos-pos local.add-strict-left-mono local.mult-imp-div-pos-less*
local.semiring-normalization-rules(*24*) *local.semiring-normalization-rules*(*4*) *local.zero-less-one*
mult-commute)

subclass *unbounded-dense-order*
proof
fix $x\ y :: 'a$
have *less-add-one*: $a < a + 1$ **for** a::$'a$ **by** *auto*
from *less-add-one* **show** $\exists\, y.\ x < y$
by *blast*

from *less-add-one* **have** $x + (-\ 1) < (x + 1) + (-\ 1)$

```
    by (rule add-strict-right-mono)
  hence x − 1 < x + 1 − 1 by simp
  hence x − 1 < x by (simp add: algebra-simps)
  thus ∃ y. y < x ..
  show x < y ⟹ ∃ z>x. z < y by (blast intro!: less-half-sum gt-half-sum)
qed


lemma dense-le-bounded:
  fixes x y z :: 'a
  assumes x < y
    and *: ⋀w. ⟦ x < w ; w < y ⟧ ⟹ w ≤ z
  shows y ≤ z
proof (rule dense-le)
  fix w assume w < y
  from dense[OF ‹x < y›] obtain u where x < u u < y by safe
  have u ≤ w ∨ w ≤ u
    using ‹u < y› ‹w < y› comparable local.order.strict-implies-order by blast
  thus w ≤ z
    using * ‹u < y› ‹w < y› ‹x < u› local.dual-order.trans local.order.strict-trans2
by blast
qed

subclass field-abs-sgn ..


lemma nonzero-abs-inverse:
  a ≠ 0 ⟹ |inverse a| = inverse |a|
  by (rule abs-inverse)

lemma nonzero-abs-divide:
  b ≠ 0 ⟹ |a / b| = |a| / |b|
  by (rule abs-divide)

lemma field-le-epsilon:
  assumes e: ⋀e. 0 < e ⟹ x ≤ y + e
  shows x ≤ y
proof (rule dense-le)
  fix t assume t < x
  hence 0 < x − t by (simp add: less-diff-eq)
  from e [OF this] have x + 0 ≤ x + (y − t) by (simp add: algebra-simps)
  hence 0 ≤ y − t by (simp only: add-le-cancel-left)
  thus t ≤ y by (simp add: algebra-simps)
qed

lemma inverse-positive-iff-positive [simp]:
  (0 < inverse a) = (0 < a)
  using local.positive-imp-inverse-positive by fastforce
```

lemma *inverse-negative-iff-negative* [*simp*]:
(*inverse a* < *0*) = (*a* < *0*)
using *local.negative-imp-inverse-negative* **by** *fastforce*

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
0 ≤ *inverse a* ⟷ *0* ≤ *a*
by (*simp add*: *local.dual-order.order-iff-strict*)

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
inverse a ≤ *0* ⟷ *a* ≤ *0*
using *local.inverse-nonnegative-iff-nonnegative local.neg-0-le-iff-le* **by** *fastforce*

lemma *one-less-inverse-iff*: *1* < *inverse x* ⟷ *0* < *x* ∧ *x* < *1*
using *less-trans*[*of 1 x 0* **for** *x*]
by (*metis local.dual-order.strict-trans local.inverse-1 local.inverse-less-imp-less local.inverse-positive-iff-positive local.one-less-inverse local.zero-less-one*)

lemma *one-le-inverse-iff*: *1* ≤ *inverse x* ⟷ *0* < *x* ∧ *x* ≤ *1*
by (*metis local.dual-order.strict-trans1 local.inverse-1 local.inverse-le-imp-le local.inverse-positive-iff-positive local.one-le-inverse local.zero-less-one*)

lemma *inverse-less-1-iff*: *inverse x* < *1* ⟷ *x* ≤ *0* ∨ *1* < *x*
proof (*rule*)
assume *invx1*: *inverse x* < *1*
have *inverse x* ≤ *0* ∨ *inverse x* ≥ *0*
using *comparable invx1 local.order.strict-implies-order local.zero-less-one* **by** *blast*
then consider (*leq0*) *inverse x* ≤ *0* | (*pos*) *inverse x* > *0* | (*zero*) *inverse x* = *0*
using *local.antisym-conv1* **by** *blast*
thus *x* ≤ *0* ∨ *1* < *x*
by (*metis invx1 local.eq-refl local.inverse-1 inverse-less-imp-less inverse-nonpositive-iff-nonpositive inverse-positive-iff-positive*)
next
assume *x* ≤ *0* ∨ *1* < *x*
then consider (*neg*) *x* ≤ *0* | (*g1*) *1* < *x* **by** *auto*
thus *inverse x* < *1*
by (*metis local.dual-order.not-eq-order-implies-strict local.dual-order.strict-trans local.inverse-1 local.inverse-negative-iff-negative local.inverse-zero local.less-imp-inverse-less local.zero-less-one*)
qed

lemma *inverse-le-1-iff*: *inverse x* ≤ *1* ⟷ *x* ≤ *0* ∨ *1* ≤ *x*
by (*metis local.dual-order.order-iff-strict local.inverse-1 local.inverse-le-iff-le local.inverse-less-1-iff local.one-le-inverse-iff*)

Simplify expressions such as *0* < *1*/*x* to *0* < *x*

lemma *zero-le-divide-1-iff* [*simp*]:
0 ≤ *1* / *a* ⟷ *0* ≤ *a*
using *local.dual-order.order-iff-strict local.inverse-eq-divide*

local.inverse-positive-iff-positive **by** *auto*

lemma *zero-less-divide-1-iff* [*simp*]:
0 < 1 / a ⟷ 0 < a
by (*simp add*: *local.dual-order.strict-iff-order*)

lemma *divide-le-0-1-iff* [*simp*]:
1 / a ≤ 0 ⟷ a ≤ 0
by (*smt local.abs-0 local.abs-1 local.abs-divide local.abs-neg local.abs-nn local.divide-cancel-left local.le-minus-iff local.minus-divide-right local.zero-neq-one*)

lemma *divide-less-0-1-iff* [*simp*]:
1 / a < 0 ⟷ a < 0
using *local.dual-order.strict-iff-order* **by** *auto*

lemma *divide-right-mono*:
a ≤ b ⟹ 0 ≤ c ⟹ a/c ≤ b/c
using *local.divide-cancel-right local.divide-strict-right-mono local.dual-order.order-iff-strict*
by *blast*

lemma *divide-right-mono-neg*: *a ≤ b*
⟹ c ≤ 0 ⟹ b / c ≤ a / c
by (*metis local.divide-cancel-right local.divide-strict-right-mono-neg local.dual-order.strict-implies-order local.eq-refl local.le-imp-less-or-eq*)

lemma *divide-left-mono-neg*: *a ≤ b*
*⟹ c ≤ 0 ⟹ 0 < a * b ⟹ c / a ≤ c / b*
by (*metis local.divide-left-mono local.minus-divide-left local.neg-0-le-iff-le local.neg-le-iff-le mult-commute*)

lemma *divide-nonneg-nonneg* [*simp*]:
0 ≤ x ⟹ 0 ≤ y ⟹ 0 ≤ x / y
using *local.divide-eq-0-iff local.divide-nonneg-pos local.dual-order.order-iff-strict*
by *blast*

lemma *divide-nonpos-nonpos*:
x ≤ 0 ⟹ y ≤ 0 ⟹ 0 ≤ x / y
using *local.divide-nonpos-neg local.dual-order.order-iff-strict* **by** *auto*

lemma *divide-nonneg-nonpos*:
0 ≤ x ⟹ y ≤ 0 ⟹ x / y ≤ 0
by (*metis local.divide-eq-0-iff local.divide-nonneg-neg local.dual-order.order-iff-strict*)

lemma *divide-nonpos-nonneg*:
x ≤ 0 ⟹ 0 ≤ y ⟹ x / y ≤ 0
using *local.divide-nonpos-pos local.dual-order.order-iff-strict* **by** *auto*

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*]:

$0 < a \Longrightarrow (1 \leq b/a) = (a \leq b)$
by (*simp add*: *local.pos-le-divide-eq*)

lemma *le-divide-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (1 \leq b/a) = (b \leq a)$
by (*metis local.le-divide-eq-1-pos local.minus-divide-divide local.neg-0-less-iff-less local.neg-le-iff-le*)

lemma *divide-le-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (b/a \leq 1) = (b \leq a)$
using *local.pos-divide-le-eq* **by** *auto*

lemma *divide-le-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (b/a \leq 1) = (a \leq b)$
by (*metis local.divide-le-eq-1-pos local.minus-divide-divide local.neg-0-less-iff-less local.neg-le-iff-le*)

lemma *less-divide-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (1 < b/a) = (a < b)$
by (*simp add*: *local.dual-order.strict-iff-order*)

lemma *less-divide-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (1 < b/a) = (b < a)$
using *local.dual-order.strict-iff-order* **by** *auto*

lemma *divide-less-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (b/a < 1) = (b < a)$
using *local.divide-le-eq-1-pos local.dual-order.strict-iff-order* **by** *auto*

lemma *divide-less-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow b/a < 1 \longleftrightarrow a < b$
using *local.dual-order.strict-iff-order* **by** *auto*

lemma *abs-div-pos*: $0 < y \Longrightarrow$
$|x| \ / \ y = |x \ / \ y|$
by (*simp add*: *local.abs-pos*)

lemma *zero-le-divide-abs-iff* [*simp*]: $(0 \leq a \ / \ |b|) = (0 \leq a \mid b = 0)$
proof
assume *assm*: $0 \leq a \ / \ |b|$
have *absb*: $abs\ b \geq 0$ **by** (*fact abs-nn*)
thus $0 \leq a \vee b = 0$
using *absb assm local.abs-eq-0-iff local.mult-nonneg-nonneg* **by** *fastforce*
next
assume $0 \leq a \vee b = 0$
then consider (*a*) $0 \leq a \mid$ (*b*) $b = 0$ **by** *atomize-elim auto*
thus $0 \leq a \ / \ |b|$
by (*metis local.abs-eq-0-iff local.abs-nn local.divide-eq-0-iff local.divide-nonneg-nonneg*)
qed

lemma *divide-le-0-abs-iff* [*simp*]: $(a \mathbin{/} |b| \leq 0) = (a \leq 0 \mid b = 0)$
by (*metis local.minus-divide-left local.neg-0-le-iff-le local.zero-le-divide-abs-iff*)

For creating values between u and v.

lemma *scaling-mono*:
assumes $u \leq v$ **and** $0 \leq r$ **and** $r \leq s$
shows $u + r * (v - u) \mathbin{/} s \leq v$
proof −
have $r/s \leq 1$ **using** *assms*
by (*metis local.divide-le-eq-1-pos local.division-ring-divide-zero*
local.dual-order.order-iff-strict local.dual-order.trans local.zero-less-one)
hence $(r/s) * (v - u) \leq 1 * (v - u)$
using *assms*(*1*) *local.diff-ge-0-iff-ge local.mult-right-mono* **by** *blast*
thus *?thesis*
by (*simp add*: *field-simps*)
qed

end

code-identifier
code-module *Ordered-Fields* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

4.4 Ordering on complex numbers

instantiation *complex* :: *nice-ordered-field* **begin**
instance
proof *intro-classes*
note *defs* = *less-eq-complex-def less-complex-def abs-complex-def*
fix *x y z a b c* :: *complex*
show $a \leq 0 \Longrightarrow |a| = - a$ **unfolding** *defs*
by (*simp add*: *cmod-eq-Re complex-is-Real-iff*)
show $0 \leq a \Longrightarrow |a| = a$
unfolding *defs*
by (*metis abs-of-nonneg cmod-eq-Re comp-apply complex.exhaust-sel complex-of-real-def zero-complex.simps*(*1*) *zero-complex.simps*(*2*))
show $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$ **unfolding** *defs* **by** *auto*
show $0 < (1{::}complex)$ **unfolding** *defs* **by** *simp*
show $0 < a \Longrightarrow 0 < inverse\ a$ **unfolding** *defs* **by** *auto*
define *ra ia rb ib rc ic* **where** *ra* = *Re a ia* = *Im a rb* = *Re b ib* = *Im b rc* = *Re c ic* = *Im c*
note *ri* = *this*[*symmetric*]
hence *a* = *Complex ra ia b* = *Complex rb ib c* = *Complex rc ic* **by** *auto*
note *ri* = *this ri*
have *rb* ≤ *ra*
if *1 / ra* ≤ (*if rb* = *0 then 0 else 1 / rb*)

and $ia = 0$ **and** $0 < ra$ **and** $ib = 0$
proof(*cases* $rb = 0$)
case *True*
thus *?thesis*
using *that*(*3*) **by** *auto*
next
case *False*
thus *?thesis*
by (*smt nice-ordered-field-class.frac-less2 that*(*1*) *that*(*3*))
qed
thus *inverse a* $\leq$ *inverse b* $\Longrightarrow$ $0 < a \Longrightarrow b \leq a$ **unfolding** *defs ri*
by (*auto simp*: *power2-eq-square*)
show $(\bigwedge a.\ a < b \Longrightarrow a \leq c) \Longrightarrow b \leq c$ **unfolding** *defs ri*
by (*metis complex.sel*(*1*) *complex.sel*(*2*) *dense less-le-not-le*
nice-ordered-field-class.linordered-field-no-lb not-le-imp-less)
show $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a \leq b \vee b \leq a$ **unfolding** *defs* **by** *auto*
show $0 \leq |x|$ **unfolding** *defs* **by** *auto*
qed
end

lemma *complex-of-real-mono*:
x $\leq$ *y* $\Longrightarrow$ *complex-of-real x* $\leq$ *complex-of-real y*
unfolding *less-eq-complex-def* **by** *auto*

lemma *complex-of-real-mono-iff*[*simp*]:
complex-of-real x $\leq$ *complex-of-real y* $\longleftrightarrow$ *x* $\leq$ *y*
unfolding *less-eq-complex-def* **by** *auto*

lemma *complex-of-real-strict-mono-iff*[*simp*]:
complex-of-real x $<$ *complex-of-real y* $\longleftrightarrow$ *x* $<$ *y*
unfolding *less-complex-def* **by** *auto*

lemma *complex-of-real-nn-iff*[*simp*]:
$0 \leq$ *complex-of-real y* $\longleftrightarrow$ $0 \leq$ *y*
unfolding *less-eq-complex-def* **by** *auto*

lemma *complex-of-real-pos-iff*[*simp*]:
$0 <$ *complex-of-real y* $\longleftrightarrow$ $0 <$ *y*
unfolding *less-complex-def* **by** *auto*

end

5 *Extra-Operator-Norm* – Additional facts bout the operator norm

theory *Extra-Operator-Norm*
imports *HOL−Analysis.Operator-Norm*
Extra-General

HOL−Analysis.Bounded-Linear-Function
Extra-Vector-Spaces
begin

This theorem complements *HOL−Analysis.Operator-Norm* additional useful facts about operator norms.

lemma *onorm-sphere*:
fixes $f :: 'a::\{real\text{-}normed\text{-}vector, not\text{-}singleton\} \Rightarrow 'b::real\text{-}normed\text{-}vector$
assumes *a1*: *bounded-linear f*
shows ‹$onorm\ f = Sup\ \{norm\ (f\ x) \mid x.\ norm\ x = 1\}$›
proof(*cases* ‹$f = (\lambda\ \text{-}.\ 0)$›)
case *True*
have ‹$(UNIV::'a\ set) \neq \{0\}$›
by *simp*
hence ‹$\exists\, x::'a.\ norm\ x = 1$›
using *ex-norm1*
by *blast*
have ‹$norm\ (f\ x) = 0$›
for *x*
by (*simp add*: *True*)
hence ‹$\{norm\ (f\ x) \mid x.\ norm\ x = 1\} = \{0\}$›
using ‹$\exists\, x.\ norm\ x = 1$› **by** *auto*
hence *v1*: ‹$Sup\ \{norm\ (f\ x) \mid x.\ norm\ x = 1\} = 0$›
by *simp*
have ‹$onorm\ f = 0$›
by (*simp add*: *True onorm-eq-0*)
thus *?thesis* **using** *v1* **by** *simp*
next
case *False*
have ‹$y \in \{norm\ (f\ x)\ |x.\ norm\ x = 1\} \cup \{0\}$›
if $y \in \{norm\ (f\ x)\ /\ norm\ x\ |x.\ True\}$
for *y*
proof(*cases* ‹$y = 0$›)
case *True*
thus *?thesis*
by *simp*
next
case *False*
have ‹$\exists\ x.\ y = norm\ (f\ x)\ /\ norm\ x$›
using ‹$y \in \{norm\ (f\ x)\ /\ norm\ x\ |x.\ True\}$› **by** *auto*
then obtain *x* **where** ‹$y = norm\ (f\ x)\ /\ norm\ x$›
by *blast*
hence ‹$y = |(1/norm\ x)| * norm\ (\ f\ x\ )$›
by *simp*
hence ‹$y = norm\ (\ (1/norm\ x) *_R f\ x\ )$›
by *simp*
hence ‹$y = norm\ (\ f\ ((1/norm\ x) *_R x)\ )$›
apply (*subst linear-cmul*[*of f*])
by (*simp-all add*: *assms bounded-linear.linear*)

```
  moreover have ‹norm ((1/norm x) *R x) = 1›
    using False ‹y = norm (f x) / norm x› by auto
  ultimately have ‹y ∈ {norm (f x) |x. norm x = 1}›
    by blast
  thus ?thesis by blast
qed
moreover have y ∈ {norm (f x) / norm x |x. True}
  if ‹y ∈ {norm (f x) |x. norm x = 1} ∪ {0}›
  for y
proof(cases ‹y = 0›)
  case True
  thus ?thesis
    by auto
next
  case False
  hence ‹y ∉ {0}›
    by simp
  hence ‹y ∈ {norm (f x) |x. norm x = 1}›
    using that by auto
  hence ‹∃ x. norm x = 1 ∧ y = norm (f x)›
    by auto
  then obtain x where ‹norm x = 1› and ‹y = norm (f x)›
    by auto
  have ‹y = norm (f x) / norm x› using ‹norm x = 1› ‹y = norm (f x)›
    by simp
  thus ?thesis
    by auto
qed
ultimately have ‹{norm (f x) / norm x |x. True} = {norm (f x) |x. norm x =
1} ∪ {0}›
  by blast
hence ‹Sup {norm (f x) / norm x |x. True} = Sup ({norm (f x) |x. norm x =
1} ∪ {0})›
  by simp
moreover have ‹Sup {norm (f x) |x. norm x = 1} ≥ 0›
proof−
  have ‹∃ x::'a. norm x = 1›
    by (metis (full-types) False assms linear-simps(3) norm-sgn)
  then obtain x::'a where ‹norm x = 1›
    by blast
  have ‹norm (f x) ≥ 0›
    by simp
  hence ‹∃ x::'a. norm x = 1 ∧ norm (f x) ≥ 0›
    using ‹norm x = 1› by blast
  hence ‹∃ y ∈ {norm (f x) |x. norm x = 1}. y ≥ 0›
    by blast
  then obtain y::real where ‹y ∈ {norm (f x) |x. norm x = 1}›
    and ‹y ≥ 0›
    by auto
```

```
    have ‹{norm (f x) |x. norm x = 1} ≠ {}›
      using ‹y ∈ {norm (f x) |x. norm x = 1}› by blast
    moreover have ‹bdd-above {norm (f x) |x. norm x = 1}›
      using bdd-above-norm-f
      by (metis (mono-tags, lifting) a1)
    ultimately have ‹y ≤ Sup {norm (f x) |x. norm x = 1}›
      using ‹y ∈ {norm (f x) |x. norm x = 1}›
      by (simp add: cSup-upper)
    thus ?thesis using ‹y ≥ 0› by simp
  qed
  moreover have ‹Sup ({norm (f x) |x. norm x = 1} ∪ {0}) = Sup {norm (f x)
|x. norm x = 1}›
  proof−
    have ‹{norm (f x) |x. norm x = 1} ≠ {}›
      by (simp add: assms(1) ex-norm1)
    moreover have ‹bdd-above {norm (f x) |x. norm x = 1}›
      using a1 bdd-above-norm-f by force
    have ‹{0::real} ≠ {}›
      by simp
    moreover have ‹bdd-above {0::real}›
      by simp
    ultimately have ‹Sup ({norm (f x) |x. norm x = 1} ∪ {(0::real)})
          = max (Sup {norm (f x) |x. norm x = 1}) (Sup {0::real})›
      by (metis (lifting) ‹0 ≤ Sup {norm (f x) |x. norm x = 1}› ‹bdd-above {0}›
‹bdd-above {norm (f x) |x. norm x = 1}› ‹{0} ≠ {}› ‹{norm (f x) |x. norm x =
1} ≠ {}› cSup-singleton cSup-union-distrib max.absorb-iff1 sup.absorb-iff1)
    moreover have ‹Sup {(0::real)} = (0::real)›
      by simp
    moreover have ‹Sup {norm (f x) |x. norm x = 1} ≥ 0›
      by (simp add: ‹0 ≤ Sup {norm (f x) |x. norm x = 1}›)
    ultimately show ?thesis
      by simp
  qed
  moreover have ‹Sup ( {norm (f x) |x. norm x = 1} ∪ {0})
        = max (Sup {norm (f x) |x. norm x = 1}) (Sup {0}) ›
    using calculation(2) calculation(3) by auto
  ultimately have w1: Sup {norm (f x) / norm x | x. True} = Sup {norm (f x)
| x. norm x = 1}
    by simp

  have ‹(SUP x. norm (f x) / (norm x)) = Sup {norm (f x) / norm x | x. True}›
    by (simp add: full-SetCompr-eq)
  also have ‹... = Sup {norm (f x) | x. norm x = 1}›
    using w1 by auto
  ultimately  have ‹(SUP x. norm (f x) / (norm x)) = Sup {norm (f x) | x. norm
x = 1}›
    by linarith
  thus ?thesis unfolding onorm-def by blast
qed
```

lemma *onormI*:
assumes $\bigwedge x.\ norm\ (f\ x) \leq b * norm\ x$
and $x \neq 0$ **and** $norm\ (f\ x) = b * norm\ x$
shows $onorm\ f = b$
apply (*unfold onorm-def*, *rule cSup-eq-maximum*)
apply (*smt* (*verit*) *UNIV-I assms*(2) *assms*(3) *image-iff nonzero-mult-div-cancel-right norm-eq-zero*)
by (*smt* (*verit*, *del-insts*) *assms*(1) *assms*(2) *divide-nonneg-nonpos norm-ge-zero norm-le-zero-iff pos-divide-le-eq rangeE zero-le-mult-iff*)

end

6 *Complex-Vector-Spaces0* – Vector Spaces and Algebras over the Complex Numbers

theory *Complex-Vector-Spaces0*
imports *HOL.Real-Vector-Spaces HOL.Topological-Spaces HOL.Vector-Spaces*
Complex-Main
HOL−Library.Complex-Order
HOL−Analysis.Product-Vector
begin

6.1 Complex vector spaces

class *scaleC* = *scaleR* +
fixes *scaleC* :: *complex* ⇒ 'a ⇒ 'a (**infixr** ‹*$_C$› 75)
assumes *scaleR-scaleC*: *scaleR r* = *scaleC* (*complex-of-real r*)
begin

abbreviation *divideC* :: 'a ⇒ *complex* ⇒ 'a (**infixl** ‹'/$_C$› 70)
where $x\ /_C\ c \equiv inverse\ c *_C x$

end

class *complex-vector* = *scaleC* + *ab-group-add* +
assumes *scaleC-add-right*: $a *_C (x + y) = (a *_C x) + (a *_C y)$
and *scaleC-add-left*: $(a + b) *_C x = (a *_C x) + (b *_C x)$
and *scaleC-scaleC*[*simp*]: $a *_C (b *_C x) = (a * b) *_C x$
and *scaleC-one*[*simp*]: $1 *_C x = x$

subclass (**in** *complex-vector*) *real-vector*
by (*standard*, *simp-all add*: *scaleR-scaleC scaleC-add-right scaleC-add-left*)

class *complex-algebra* = *complex-vector* + *ring* +
assumes *mult-scaleC-left* [*simp*]: $a *_C x * y = a *_C (x * y)$
and *mult-scaleC-right* [*simp*]: $x * a *_C y = a *_C (x * y)$

subclass (**in** *complex-algebra*) *real-algebra*
  by (*standard*, *simp-all add*: *scaleR-scaleC*)

class *complex-algebra-1* = *complex-algebra* + *ring-1*

subclass (**in** *complex-algebra-1*) *real-algebra-1* **..**

class *complex-div-algebra* = *complex-algebra-1* + *division-ring*

subclass (**in** *complex-div-algebra*) *real-div-algebra* **..**

class *complex-field* = *complex-div-algebra* + *field*

subclass (**in** *complex-field*) *real-field* **..**

instantiation *complex* :: *complex-field*
begin

definition *complex-scaleC-def* [*simp*]: *scaleC a x* = *a* ∗ *x*

instance
proof *intro-classes*
  fix *r* :: *real* **and** *a b x y* :: *complex*
  show ((∗$_R$) *r*::*complex* ⇒ -) = (∗$_C$) (*complex-of-real r*)
    by (*auto simp add*: *scaleR-conv-of-real*)
  show *a* ∗$_C$ (*x* + *y*) = *a* ∗$_C$ *x* + *a* ∗$_C$ *y*
    by (*simp add*: *ring-class.ring-distribs*(*1*))
  show (*a* + *b*) ∗$_C$ *x* = *a* ∗$_C$ *x* + *b* ∗$_C$ *x*
    by (*simp add*: *algebra-simps*)
  show *a* ∗$_C$ *b* ∗$_C$ *x* = (*a* ∗ *b*) ∗$_C$ *x*
    by *simp*
  show *1* ∗$_C$ *x* = *x*
    by *simp*
  show *a* ∗$_C$ (*x*::*complex*) ∗ *y* = *a* ∗$_C$ (*x* ∗ *y*)
    by *simp*
  show (*x*::*complex*) ∗ *a* ∗$_C$ *y* = *a* ∗$_C$ (*x* ∗ *y*)
    by *simp*
qed

end

locale *clinear* = *Vector-Spaces.linear scaleC*::-⇒-⇒-⇒′*a*::*complex-vector scaleC*::-⇒-⇒-⇒′*b*::*complex-vector*
begin

sublocale *real*: *linear*
— Gives access to all lemmas from *Real-Vector-Spaces.linear* using prefix *real*.
apply *standard*
by (*auto simp add*: *add scale scaleR-scaleC*)

lemmas *scaleC* = *scale*

end

global-interpretation *complex-vector*: *vector-space scaleC* :: *complex* ⇒ *'a* ⇒ *'a*
:: *complex-vector*
rewrites *Vector-Spaces.linear* (∗$_C$) (∗$_C$) = *clinear*
and *Vector-Spaces.linear* (∗) (∗$_C$) = *clinear*
defines *cdependent-raw-def*: *cdependent* = *complex-vector.dependent*
and *crepresentation-raw-def*: *crepresentation* = *complex-vector.representation*
and *csubspace-raw-def*: *csubspace* = *complex-vector.subspace*
and *cspan-raw-def*: *cspan* = *complex-vector.span*
and *cextend-basis-raw-def*: *cextend-basis* = *complex-vector.extend-basis*
and *cdim-raw-def*: *cdim* = *complex-vector.dim*
proof *unfold-locales*
show *Vector-Spaces.linear* (∗$_C$) (∗$_C$) = *clinear Vector-Spaces.linear* (∗) (∗$_C$) =
clinear
by (*force simp*: *clinear-def complex-scaleC-def*[*abs-def*])+
qed (*use scaleC-add-right scaleC-add-left* **in** *auto*)

abbreviation *cindependent x* ≡ ¬ *cdependent x*

global-interpretation *complex-vector*: *vector-space-pair scaleC*::-⇒-⇒*'a*::*complex-vector*
scaleC::-⇒-⇒*'b*::*complex-vector*
rewrites *Vector-Spaces.linear* (∗$_C$) (∗$_C$) = *clinear*
and *Vector-Spaces.linear* (∗) (∗$_C$) = *clinear*
defines *cconstruct-raw-def*: *cconstruct* = *complex-vector.construct*
proof *unfold-locales*
show *Vector-Spaces.linear* (∗) (∗$_C$) = *clinear*
unfolding *clinear-def complex-scaleC-def* **by** *auto*
qed (*auto simp*: *clinear-def*)

lemma *clinear-compose*: *clinear f* ⟹ *clinear g* ⟹ *clinear* (*g* ∘ *f*)
unfolding *clinear-def* **by** (*rule Vector-Spaces.linear-compose*)

Recover original theorem names

lemmas *scaleC-left-commute* = *complex-vector.scale-left-commute*
lemmas *scaleC-zero-left* = *complex-vector.scale-zero-left*

lemmas *scaleC-minus-left* = *complex-vector.scale-minus-left*
lemmas *scaleC-diff-left* = *complex-vector.scale-left-diff-distrib*
lemmas *scaleC-sum-left* = *complex-vector.scale-sum-left*
lemmas *scaleC-zero-right* = *complex-vector.scale-zero-right*
lemmas *scaleC-minus-right* = *complex-vector.scale-minus-right*
lemmas *scaleC-diff-right* = *complex-vector.scale-right-diff-distrib*
lemmas *scaleC-sum-right* = *complex-vector.scale-sum-right*
lemmas *scaleC-eq-0-iff* = *complex-vector.scale-eq-0-iff*
lemmas *scaleC-left-imp-eq* = *complex-vector.scale-left-imp-eq*
lemmas *scaleC-right-imp-eq* = *complex-vector.scale-right-imp-eq*
lemmas *scaleC-cancel-left* = *complex-vector.scale-cancel-left*
lemmas *scaleC-cancel-right* = *complex-vector.scale-cancel-right*

lemma *divideC-field-simps*[*field-simps*]:
$c \neq 0 \Longrightarrow a = b \mathbin{/_C} c \longleftrightarrow c *_C a = b$
$c \neq 0 \Longrightarrow b \mathbin{/_C} c = a \longleftrightarrow b = c *_C a$
$c \neq 0 \Longrightarrow a + b \mathbin{/_C} c = (c *_C a + b) \mathbin{/_C} c$
$c \neq 0 \Longrightarrow a \mathbin{/_C} c + b = (a + c *_C b) \mathbin{/_C} c$
$c \neq 0 \Longrightarrow a - b \mathbin{/_C} c = (c *_C a - b) \mathbin{/_C} c$
$c \neq 0 \Longrightarrow a \mathbin{/_C} c - b = (a - c *_C b) \mathbin{/_C} c$
$c \neq 0 \Longrightarrow - (a \mathbin{/_C} c) + b = (- a + c *_C b) \mathbin{/_C} c$
$c \neq 0 \Longrightarrow - (a \mathbin{/_C} c) - b = (- a - c *_C b) \mathbin{/_C} c$
for $a\ b :: 'a :: complex\text{-}vector$
by (*auto simp add*: *scaleC-add-right scaleC-add-left scaleC-diff-right scaleC-diff-left*)

Legacy names – omitted

lemmas *clinear-injective-0* = *linear-inj-iff-eq-0*
and *clinear-injective-on-subspace-0* = *linear-inj-on-iff-eq-0*
and *clinear-cmul* = *linear-scale*
and *clinear-scaleC* = *linear-scale-self*
and *csubspace-mul* = *subspace-scale*
and *cspan-linear-image* = *linear-span-image*
and *cspan-0* = *span-zero*
and *cspan-mul* = *span-scale*
and *injective-scaleC* = *injective-scale*

lemma *scaleC-minus1-left* [*simp*]: *scaleC* $(-1)\ x = - x$
for $x :: 'a::complex\text{-}vector$
using *scaleC-minus-left* [*of 1 x*] **by** *simp*

lemma *scaleC-2*:
fixes $x :: 'a::complex\text{-}vector$
shows *scaleC 2 x* $= x + x$
unfolding *one-add-one* [*symmetric*] *scaleC-add-left* **by** *simp*

lemma *scaleC-half-double* [*simp*]:
fixes $a :: 'a::complex\text{-}vector$
shows $(1 \mathbin{/} 2) *_C (a + a) = a$
proof −

have $\bigwedge r.\ r *_C (a + a) = (r * 2) *_C a$
by (*metis scaleC-2 scaleC-scaleC*)
thus *?thesis*
by *simp*
qed

lemma *clinear-scale-complex*:
fixes *c*::*complex* **shows** *clinear f* $\Longrightarrow$ *f* (*c* * *b*) = *c* * *f b*
using *complex-vector.linear-scale* **by** *fastforce*

interpretation *scaleC-left*: *additive* ($\lambda a.$ *scaleC a x* :: *'a*::*complex-vector*)
by *standard* (*rule scaleC-add-left*)

interpretation *scaleC-right*: *additive* ($\lambda x.$ *scaleC a x* :: *'a*::*complex-vector*)
by *standard* (*rule scaleC-add-right*)

lemma *nonzero-inverse-scaleC-distrib*:
a $\neq$ *0* $\Longrightarrow$ *x* $\neq$ *0* $\Longrightarrow$ *inverse* (*scaleC a x*) = *scaleC* (*inverse a*) (*inverse x*)
for *x* :: *'a*::*complex-div-algebra*
by (*rule inverse-unique*) *simp*

lemma *inverse-scaleC-distrib*: *inverse* (*scaleC a x*) = *scaleC* (*inverse a*) (*inverse x*)
for *x* :: *'a*::{*complex-div-algebra*,*division-ring*}
by (*metis inverse-zero nonzero-inverse-scaleC-distrib complex-vector.scale-eq-0-iff*)

lemma *complex-add-divide-simps*[*vector-add-divide-simps*]:
v + (*b* / *z*) *$*_C$ *w* = (*if z* = *0 then v else* (*z* *$*_C$ *v* + *b* *$*_C$ *w*) /$_C$ *z*)
a *$*_C$ *v* + (*b* / *z*) *$*_C$ *w* = (*if z* = *0 then a* *$*_C$ *v else* ((*a* * *z*) *$*_C$ *v* + *b* *$*_C$ *w*) /$_C$ *z*)
(*a* / *z*) *$*_C$ *v* + *w* = (*if z* = *0 then w else* (*a* *$*_C$ *v* + *z* *$*_C$ *w*) /$_C$ *z*)
(*a* / *z*) *$*_C$ *v* + *b* *$*_C$ *w* = (*if z* = *0 then b* *$*_C$ *w else* (*a* *$*_C$ *v* + (*b* * *z*) *$*_C$ *w*) /$_C$ *z*)
v − (*b* / *z*) *$*_C$ *w* = (*if z* = *0 then v else* (*z* *$*_C$ *v* − *b* *$*_C$ *w*) /$_C$ *z*)
a *$*_C$ *v* − (*b* / *z*) *$*_C$ *w* = (*if z* = *0 then a* *$*_C$ *v else* ((*a* * *z*) *$*_C$ *v* − *b* *$*_C$ *w*) /$_C$ *z*)
(*a* / *z*) *$*_C$ *v* − *w* = (*if z* = *0 then* −*w else* (*a* *$*_C$ *v* − *z* *$*_C$ *w*) /$_C$ *z*)
(*a* / *z*) *$*_C$ *v* − *b* *$*_C$ *w* = (*if z* = *0 then* −*b* *$*_C$ *w else* (*a* *$*_C$ *v* − (*b* * *z*) *$*_C$ *w*) /$_C$ *z*)
for *v* :: *'a* :: *complex-vector*
by (*simp-all add*: *divide-inverse-commute scaleC-add-right scaleC-diff-right*)

lemma *ceq-vector-fraction-iff* [*vector-add-divide-simps*]:
fixes *x* :: *'a* :: *complex-vector*

shows $(x = (u\ /\ v) *_C a) \longleftrightarrow (\textit{if } v{=}0 \textit{ then } x = 0 \textit{ else } v *_C x = u *_C a)$
by *auto* (*metis* (*no-types*) *divide-eq-1-iff divide-inverse-commute scaleC-one scaleC-scaleC*)

lemma *cvector-fraction-eq-iff* [*vector-add-divide-simps*]:
fixes $x :: 'a ::$ *complex-vector*
shows $((u\ /\ v) *_C a = x) \longleftrightarrow (\textit{if } v{=}0 \textit{ then } x = 0 \textit{ else } u *_C a = v *_C x)$
by (*metis ceq-vector-fraction-iff*)

lemma *complex-vector-affinity-eq*:
fixes $x :: 'a ::$ *complex-vector*
assumes *m0*: $m \neq 0$
shows $m *_C x + c = y \longleftrightarrow x = \textit{inverse } m *_C y - (\textit{inverse } m *_C c)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
hence $m *_C x = y - c$ **by** (*simp add*: *field-simps*)
hence $\textit{inverse } m *_C (m *_C x) = \textit{inverse } m *_C (y - c)$ **by** *simp*
thus $x = \textit{inverse } m *_C y - (\textit{inverse } m *_C c)$
using *m0*
by (*simp add*: *complex-vector.scale-right-diff-distrib*)
next
assume *?rhs*
with *m0* **show** $m *_C x + c = y$
by (*simp add*: *complex-vector.scale-right-diff-distrib*)
qed

lemma *complex-vector-eq-affinity*: $m \neq 0 \Longrightarrow y = m *_C x + c \longleftrightarrow \textit{inverse } m *_C y - (\textit{inverse } m *_C c) = x$
for $x :: 'a$::*complex-vector*
using *complex-vector-affinity-eq*[**where** $m{=}m$ **and** $x{=}x$ **and** $y{=}y$ **and** $c{=}c$]
by *metis*

lemma *scaleC-eq-iff* [*simp*]: $b + u *_C a = a + u *_C b \longleftrightarrow a = b \vee u = 1$
for $a :: 'a$::*complex-vector*
proof (*cases* $u = 1$)
case *True*
thus *?thesis* **by** *auto*
next
case *False*
have $a = b$ **if** $b + u *_C a = a + u *_C b$
proof −
from *that* **have** $(u - 1) *_C a = (u - 1) *_C b$
by (*simp add*: *algebra-simps*)
with *False* **show** *?thesis*
by *auto*
qed
thus *?thesis* **by** *auto*
qed

lemma *scaleC-collapse* [*simp*]: $(1 - u) *_C a + u *_C a = a$
for a :: *'a*::*complex-vector*
by (*simp add*: *algebra-simps*)

6.2 Embedding of the Complex Numbers into any *complex-algebra-1*: *of-complex*

definition *of-complex* :: *complex* ⇒ *'a*::*complex-algebra-1*
where *of-complex c* = *scaleC c 1*

lemma *scaleC-conv-of-complex*: *scaleC r x* = *of-complex r* ∗ *x*
by (*simp add*: *of-complex-def*)

lemma *of-complex-0* [*simp*]: *of-complex 0* = *0*
by (*simp add*: *of-complex-def*)

lemma *of-complex-1* [*simp*]: *of-complex 1* = *1*
by (*simp add*: *of-complex-def*)

lemma *of-complex-add* [*simp*]: *of-complex* (*x* + *y*) = *of-complex x* + *of-complex y*
by (*simp add*: *of-complex-def scaleC-add-left*)

lemma *of-complex-minus* [*simp*]: *of-complex* (− *x*) = − *of-complex x*
by (*simp add*: *of-complex-def*)

lemma *of-complex-diff* [*simp*]: *of-complex* (*x* − *y*) = *of-complex x* − *of-complex y*
by (*simp add*: *of-complex-def scaleC-diff-left*)

lemma *of-complex-mult* [*simp*]: *of-complex* (*x* ∗ *y*) = *of-complex x* ∗ *of-complex y*
by (*simp add*: *of-complex-def mult.commute*)

lemma *of-complex-sum*[*simp*]: *of-complex* (*sum f s*) = $(\sum x{\in}s.$ *of-complex* (*f x*))
by (*induct s rule*: *infinite-finite-induct*) *auto*

lemma *of-complex-prod*[*simp*]: *of-complex* (*prod f s*) = $(\prod x{\in}s.$ *of-complex* (*f x*))
by (*induct s rule*: *infinite-finite-induct*) *auto*

lemma *nonzero-of-complex-inverse*:
x ≠ *0* ⟹ *of-complex* (*inverse x*) = *inverse* (*of-complex x* :: *'a*::*complex-div-algebra*)
by (*simp add*: *of-complex-def nonzero-inverse-scaleC-distrib*)

lemma *of-complex-inverse* [*simp*]:
of-complex (*inverse x*) = *inverse* (*of-complex x* :: *'a*::{*complex-div-algebra*,*division-ring*})
by (*simp add*: *of-complex-def inverse-scaleC-distrib*)

lemma *nonzero-of-complex-divide*:
y ≠ *0* ⟹ *of-complex* (*x* / *y*) = (*of-complex x* / *of-complex y* :: *'a*::*complex-field*)
by (*simp add*: *divide-inverse nonzero-of-complex-inverse*)

lemma *of-complex-divide* [*simp*]:
of-complex (*x* / *y*) = (*of-complex x* / *of-complex y* :: *'a*::*complex-div-algebra*)
by (*simp add*: *divide-inverse*)

lemma *of-complex-power* [*simp*]:
of-complex (*x* ^ *n*) = (*of-complex x* :: *'a*::{*complex-algebra-1*}) ^ *n*
by (*induct n*) *simp-all*

lemma *of-complex-power-int* [*simp*]:
of-complex (*power-int x n*) = *power-int* (*of-complex x* :: *'a* :: {*complex-div-algebra*,*division-ring*})
n
by (*auto simp*: *power-int-def*)

lemma *of-complex-eq-iff* [*simp*]: *of-complex x* = *of-complex y* ⟷ *x* = *y*
by (*simp add*: *of-complex-def*)

lemma *inj-of-complex*: *inj of-complex*
by (*auto intro*: *injI*)

lemmas *of-complex-eq-0-iff* [*simp*] = *of-complex-eq-iff* [*of - 0*, *simplified*]
lemmas *of-complex-eq-1-iff* [*simp*] = *of-complex-eq-iff* [*of - 1*, *simplified*]

lemma *minus-of-complex-eq-of-complex-iff* [*simp*]: −*of-complex x* = *of-complex y*
⟷ −*x* = *y*
using *of-complex-eq-iff*[*of* −*x y*] **by** (*simp only*: *of-complex-minus*)

lemma *of-complex-eq-minus-of-complex-iff* [*simp*]: *of-complex x* = −*of-complex y*
⟷ *x* = −*y*
using *of-complex-eq-iff*[*of x* −*y*] **by** (*simp only*: *of-complex-minus*)

lemma *of-complex-eq-id* [*simp*]: *of-complex* = (*id* :: *complex* ⇒ *complex*)
by (*rule ext*) (*simp add*: *of-complex-def*)

Collapse nested embeddings.

lemma *of-complex-of-nat-eq* [*simp*]: *of-complex* (*of-nat n*) = *of-nat n*
by (*induct n*) *auto*

lemma *of-complex-of-int-eq* [*simp*]: *of-complex* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*) *simp*

lemma *of-complex-numeral* [*simp*]: *of-complex* (*numeral w*) = *numeral w*
using *of-complex-of-int-eq* [*of numeral w*] **by** *simp*

lemma *of-complex-neg-numeral* [*simp*]: *of-complex* (− *numeral w*) = − *numeral w*
using *of-complex-of-int-eq* [*of* − *numeral w*] **by** *simp*

lemma *numeral-power-int-eq-of-complex-cancel-iff* [*simp*]:
power-int (*numeral x*) *n* = (*of-complex y* :: *'a* :: {*complex-div-algebra*, *divi-*

sion-ring}) ⟷
    power-int (*numeral x*) *n* = *y*
proof −
  have *power-int* (*numeral x*) *n* = (*of-complex* (*power-int* (*numeral x*) *n*) :: *'a*)
    by *simp*
  also have ... = *of-complex y* ⟷ *power-int* (*numeral x*) *n* = *y*
    by (*subst of-complex-eq-iff*) *auto*
  finally show *?thesis* **.**
qed

lemma *of-complex-eq-numeral-power-int-cancel-iff* [*simp*]:
  (*of-complex y* :: *'a* :: {*complex-div-algebra*, *division-ring*}) = *power-int* (*numeral x*) *n* ⟷
    y = *power-int* (*numeral x*) *n*
  by (*subst* (*1 2*) *eq-commute*) *simp*

lemma *of-complex-eq-of-complex-power-int-cancel-iff* [*simp*]:
  power-int (*of-complex b* :: *'a* :: {*complex-div-algebra*, *division-ring*}) *w* = *of-complex x* ⟷
    power-int b w = *x*
  by (*metis of-complex-power-int of-complex-eq-iff*)

lemma *of-complex-in-Ints-iff* [*simp*]: *of-complex x* ∈ ℤ ⟷ *x* ∈ ℤ
proof *safe*
  fix *x* **assume** (*of-complex x* :: *'a*) ∈ ℤ
  then obtain *n* **where** (*of-complex x* :: *'a*) = *of-int n*
    by (*auto simp*: *Ints-def*)
  also have *of-int n* = *of-complex* (*of-int n*)
    by *simp*
  finally have *x* = *of-int n*
    by (*subst* (*asm*) *of-complex-eq-iff*)
  thus *x* ∈ ℤ
    by *auto*
qed (*auto simp*: *Ints-def*)

lemma *Ints-of-complex* [*intro*]: *x* ∈ ℤ ⟹ *of-complex x* ∈ ℤ
  by *simp*

Every complex algebra has characteristic zero.

lemma *fraction-scaleC-times* [*simp*]:
  fixes *a* :: *'a*::*complex-algebra-1*
  shows (*numeral u* / *numeral v*) *∗C* (*numeral w* ∗ *a*) = (*numeral u* ∗ *numeral w* / *numeral v*) *∗C a*
  by (*metis* (*no-types*, *lifting*) *of-complex-numeral scaleC-conv-of-complex scaleC-scaleC times-divide-eq-left*)

lemma *inverse-scaleC-times* [*simp*]:
  fixes *a* :: *'a*::*complex-algebra-1*
  shows (*1* / *numeral v*) *∗C* (*numeral w* ∗ *a*) = (*numeral w* / *numeral v*) *∗C a*

by (*metis divide-inverse-commute inverse-eq-divide of-complex-numeral scaleC-conv-of-complex scaleC-scaleC*)

lemma *scaleC-times* [*simp*]:
fixes $a :: 'a$::*complex-algebra-1*
shows (*numeral u*) $*_C$ (*numeral w* $*$ *a*) = (*numeral u* $*$ *numeral w*) $*_C$ *a*
by (*simp add*: *scaleC-conv-of-complex*)

6.3 The Set of Real Numbers

definition *Complexs* :: $'a$::*complex-algebra-1 set* (‹$\mathbb{C}$›)
where $\mathbb{C}$ = *range of-complex*

lemma *Complexs-of-complex* [*simp*]: *of-complex r* $\in \mathbb{C}$
by (*simp add*: *Complexs-def*)

lemma *Complexs-of-int* [*simp*]: *of-int z* $\in \mathbb{C}$
by (*subst of-complex-of-int-eq* [*symmetric*], *rule Complexs-of-complex*)

lemma *Complexs-of-nat* [*simp*]: *of-nat n* $\in \mathbb{C}$
by (*subst of-complex-of-nat-eq* [*symmetric*], *rule Complexs-of-complex*)

lemma *Complexs-numeral* [*simp*]: *numeral w* $\in \mathbb{C}$
by (*subst of-complex-numeral* [*symmetric*], *rule Complexs-of-complex*)

lemma *Complexs-0* [*simp*]: *0* $\in \mathbb{C}$ **and** *Complexs-1* [*simp*]: *1* $\in \mathbb{C}$
by (*simp-all add*: *Complexs-def*)

lemma *Complexs-add* [*simp*]: $a \in \mathbb{C} \Longrightarrow b \in \mathbb{C} \Longrightarrow a + b \in \mathbb{C}$
apply (*auto simp add*: *Complexs-def*)
by (*metis of-complex-add range-eqI*)

lemma *Complexs-minus* [*simp*]: $a \in \mathbb{C} \Longrightarrow - a \in \mathbb{C}$
by (*auto simp*: *Complexs-def*)

lemma *Complexs-minus-iff* [*simp*]: $- a \in \mathbb{C} \longleftrightarrow a \in \mathbb{C}$
using *Complexs-minus* **by** *fastforce*

lemma *Complexs-diff* [*simp*]: $a \in \mathbb{C} \Longrightarrow b \in \mathbb{C} \Longrightarrow a - b \in \mathbb{C}$
by (*metis Complexs-add Complexs-minus-iff add-uminus-conv-diff*)

lemma *Complexs-mult* [*simp*]: $a \in \mathbb{C} \Longrightarrow b \in \mathbb{C} \Longrightarrow a * b \in \mathbb{C}$
apply (*auto simp add*: *Complexs-def*)
by (*metis of-complex-mult rangeI*)

lemma *nonzero-Complexs-inverse*: $a \in \mathbb{C} \Longrightarrow a \neq 0 \Longrightarrow$ *inverse a* $\in \mathbb{C}$
for $a :: 'a$::*complex-div-algebra*
apply (*auto simp add*: *Complexs-def*)
by (*metis of-complex-inverse range-eqI*)

lemma *Complexs-inverse*: $a \in \mathbb{C} \Longrightarrow \mathit{inverse}\ a \in \mathbb{C}$
for $a :: 'a$::{*complex-div-algebra*,*division-ring*}
using *nonzero-Complexs-inverse* **by** *fastforce*

lemma *Complexs-inverse-iff* [*simp*]: $\mathit{inverse}\ x \in \mathbb{C} \longleftrightarrow x \in \mathbb{C}$
for $x :: 'a$::{*complex-div-algebra*,*division-ring*}
by (*metis Complexs-inverse inverse-inverse-eq*)

lemma *nonzero-Complexs-divide*: $a \in \mathbb{C} \Longrightarrow b \in \mathbb{C} \Longrightarrow b \neq 0 \Longrightarrow a\ /\ b \in \mathbb{C}$
for $a\ b :: 'a$::*complex-field*
by (*simp add*: *divide-inverse*)

lemma *Complexs-divide* [*simp*]: $a \in \mathbb{C} \Longrightarrow b \in \mathbb{C} \Longrightarrow a\ /\ b \in \mathbb{C}$
for $a\ b :: 'a$::{*complex-field*,*field*}
using *nonzero-Complexs-divide* **by** *fastforce*

lemma *Complexs-power* [*simp*]: $a \in \mathbb{C} \Longrightarrow a\ \hat{}\ n \in \mathbb{C}$
for $a :: 'a$::*complex-algebra-1*
apply (*auto simp add*: *Complexs-def*)
by (*metis range-eqI of-complex-power*[*symmetric*])

lemma *Complexs-cases* [*cases set*: *Complexs*]:
assumes $q \in \mathbb{C}$
obtains (*of-complex*) c **where** $q = \mathit{of\text{-}complex}\ c$
unfolding *Complexs-def*
proof −
from ‹$q \in \mathbb{C}$› **have** $q \in \mathit{range\ of\text{-}complex}$ **unfolding** *Complexs-def* **.**
then obtain c **where** $q = \mathit{of\text{-}complex}\ c$ **..**
then show *thesis* **..**
qed

lemma *sum-in-Complexs* [*intro*,*simp*]: $(\bigwedge i.\ i \in s \Longrightarrow f\ i \in \mathbb{C}) \Longrightarrow \mathit{sum}\ f\ s \in \mathbb{C}$
proof (*induct s rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** (*metis Complexs-0 sum.infinite*)
qed *simp-all*

lemma *prod-in-Complexs* [*intro*,*simp*]: $(\bigwedge i.\ i \in s \Longrightarrow f\ i \in \mathbb{C}) \Longrightarrow \mathit{prod}\ f\ s \in \mathbb{C}$
proof (*induct s rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** (*metis Complexs-1 prod.infinite*)
qed *simp-all*

lemma *Complexs-induct* [*case-names of-complex*, *induct set*: *Complexs*]:
$q \in \mathbb{C} \Longrightarrow (\bigwedge r.\ P\ (\mathit{of\text{-}complex}\ r)) \Longrightarrow P\ q$
by (*rule Complexs-cases*) *auto*

6.4 Ordered complex vector spaces

class *ordered-complex-vector* = *complex-vector* + *ordered-ab-group-add* +
  assumes *scaleC-left-mono*: $x \leq y \Longrightarrow 0 \leq a \Longrightarrow a *_C x \leq a *_C y$
    and *scaleC-right-mono*: $a \leq b \Longrightarrow 0 \leq x \Longrightarrow a *_C x \leq b *_C x$
begin

subclass (**in** *ordered-complex-vector*) *ordered-real-vector*
  apply *standard*
 by (*auto simp add*: *less-eq-complex-def scaleC-left-mono scaleC-right-mono scaleR-scaleC*)

lemma *scaleC-mono*:
  $a \leq b \Longrightarrow x \leq y \Longrightarrow 0 \leq b \Longrightarrow 0 \leq x \Longrightarrow a *_C x \leq b *_C y$
  by (*meson order-trans scaleC-left-mono scaleC-right-mono*)

lemma *scaleC-mono'*:
  $a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq a \Longrightarrow 0 \leq c \Longrightarrow a *_C c \leq b *_C d$
  by (*rule scaleC-mono*) (*auto intro*: *order.trans*)

lemma *pos-le-divideC-eq* [*field-simps*]:
  $a \leq b /_C c \longleftrightarrow c *_C a \leq b$ (**is** *?P* $\longleftrightarrow$ *?Q*) **if** $0 < c$
proof
  assume *?P*
  with *scaleC-left-mono that* **have** $c *_C a \leq c *_C (b /_C c)$
    using *preorder-class.less-imp-le* **by** *blast*
  with *that* **show** *?Q*
    by *auto*
next
  assume *?Q*
  with *scaleC-left-mono that* **have** $c *_C a /_C c \leq b /_C c$
    using *less-complex-def less-eq-complex-def* **by** *fastforce*
  with *that* **show** *?P*
    by *auto*
qed

lemma *pos-less-divideC-eq* [*field-simps*]:
  $a < b /_C c \longleftrightarrow c *_C a < b$ **if** $c > 0$
  using *that pos-le-divideC-eq* [*of c a b*]
  by (*auto simp add*: *le-less*)

lemma *pos-divideC-le-eq* [*field-simps*]:
  $b /_C c \leq a \longleftrightarrow b \leq c *_C a$ **if** $c > 0$
  using *that pos-le-divideC-eq* [*of inverse c b a*]
    less-complex-def **by** *auto*

lemma *pos-divideC-less-eq* [*field-simps*]:
  $b /_C c < a \longleftrightarrow b < c *_C a$ **if** $c > 0$
  using *that pos-less-divideC-eq* [*of inverse c b a*]
  by (*simp add*: *local.less-le-not-le local.pos-divideC-le-eq local.pos-le-divideC-eq*)

lemma *pos-le-minus-divideC-eq* [*field-simps*]:
$a \leq - (b /_C c) \longleftrightarrow c *_C a \leq - b$ **if** $c > 0$
using *that*
by (*metis local.ab-left-minus local.add.inverse-unique local.add.right-inverse local.add-minus-cancel local.le-minus-iff local.pos-divideC-le-eq local.scaleC-add-right local.scaleC-one local.scaleC-scaleC*)

lemma *pos-less-minus-divideC-eq* [*field-simps*]:
$a < - (b /_C c) \longleftrightarrow c *_C a < - b$ **if** $c > 0$
using *that*
by (*metis le-less less-le-not-le pos-divideC-le-eq pos-divideC-less-eq pos-le-minus-divideC-eq*)

lemma *pos-minus-divideC-le-eq* [*field-simps*]:
$- (b /_C c) \leq a \longleftrightarrow - b \leq c *_C a$ **if** $c > 0$
using *that*
by (*metis local.add-minus-cancel local.left-minus local.pos-divideC-le-eq local.scaleC-add-right*)

lemma *pos-minus-divideC-less-eq* [*field-simps*]:
$- (b /_C c) < a \longleftrightarrow - b < c *_C a$ **if** $c > 0$
using *that* **by** (*simp add*: *less-le-not-le pos-le-minus-divideC-eq pos-minus-divideC-le-eq*)

lemma *scaleC-image-atLeastAtMost*: $c > 0 \Longrightarrow$ *scaleC c* ‘ $\{x..y\} = \{c *_C x..c *_C y\}$
apply (*auto intro*!: *scaleC-left-mono simp*: *image-iff Bex-def*)
by (*meson order.eq-iff local.order.refl pos-divideC-le-eq pos-le-divideC-eq*)

end

lemma *neg-le-divideC-eq* [*field-simps*]:
$a \leq b /_C c \longleftrightarrow b \leq c *_C a$ (**is** *?P* $\longleftrightarrow$ *?Q*) **if** $c < 0$
for $a\ b :: 'a ::$ *ordered-complex-vector*
using *that pos-le-divideC-eq* [*of* $- c\ a - b$]
by (*simp add*: *less-complex-def*)

lemma *neg-less-divideC-eq* [*field-simps*]:
$a < b /_C c \longleftrightarrow b < c *_C a$ **if** $c < 0$
for $a\ b :: 'a ::$ *ordered-complex-vector*
using *that neg-le-divideC-eq* [*of c a b*]
by (*smt* (*verit*, *ccfv-SIG*) *neg-le-divideC-eq antisym-conv2 complex-vector.scale-minus-right dual-order.strict-implies-order le-less-trans neg-le-iff-le scaleC-scaleC*)

lemma *neg-divideC-le-eq* [*field-simps*]:
$b /_C c \leq a \longleftrightarrow c *_C a \leq b$ **if** $c < 0$
for $a\ b :: 'a ::$ *ordered-complex-vector*
using *that pos-divideC-le-eq* [*of* $- c - b\ a$]
by (*simp add*: *less-complex-def*)

lemma *neg-divideC-less-eq* [*field-simps*]:
$b /_C c < a \longleftrightarrow c *_C a < b$ **if** $c < 0$

for $a\ b :: 'a ::$ *ordered-complex-vector*
using *that neg-divideC-le-eq* [*of c b a*]
by (*meson neg-le-divideC-eq less-le-not-le*)

lemma *neg-le-minus-divideC-eq* [*field-simps*]:
$a \leq - (b /_C c) \longleftrightarrow - b \leq c *_C a$ **if** $c < 0$
for $a\ b :: 'a ::$ *ordered-complex-vector*
using *that pos-le-minus-divideC-eq* [*of* $-\ c\ a\ -\ b$]
by (*metis neg-le-divideC-eq complex-vector.scale-minus-right*)

lemma *neg-less-minus-divideC-eq* [*field-simps*]:
$a < - (b /_C c) \longleftrightarrow - b < c *_C a$ **if** $c < 0$
for $a\ b :: 'a ::$ *ordered-complex-vector*
proof −
have $*$: $- b = c *_C a \longleftrightarrow b = - (c *_C a)$
by (*metis add.inverse-inverse*)
from *that neg-le-minus-divideC-eq* [*of c a b*]
show *?thesis* **by** (*auto simp add*: *le-less* $*$)
qed

lemma *neg-minus-divideC-le-eq* [*field-simps*]:
$- (b /_C c) \leq a \longleftrightarrow c *_C a \leq - b$ **if** $c < 0$
for $a\ b :: 'a ::$ *ordered-complex-vector*
using *that pos-minus-divideC-le-eq* [*of* $-\ c - b\ a$]
by (*metis Complex-Vector-Spaces0.neg-divideC-le-eq complex-vector.scale-minus-right*)

lemma *neg-minus-divideC-less-eq* [*field-simps*]:
$- (b /_C c) < a \longleftrightarrow c *_C a < - b$ **if** $c < 0$
for $a\ b :: 'a ::$ *ordered-complex-vector*
using *that* **by** (*simp add*: *less-le-not-le neg-le-minus-divideC-eq neg-minus-divideC-le-eq*)

lemma *divideC-field-splits-simps-1* [*field-split-simps*]:
$a = b /_C c \longleftrightarrow$ (*if* $c = 0$ *then* $a = 0$ *else* $c *_C a = b$)
$b /_C c = a \longleftrightarrow$ (*if* $c = 0$ *then* $a = 0$ *else* $b = c *_C a$)
$a + b /_C c =$ (*if* $c = 0$ *then* a *else* $(c *_C a + b) /_C c$)
$a /_C c + b =$ (*if* $c = 0$ *then* b *else* $(a + c *_C b) /_C c$)
$a - b /_C c =$ (*if* $c = 0$ *then* a *else* $(c *_C a - b) /_C c$)
$a /_C c - b =$ (*if* $c = 0$ *then* $- b$ *else* $(a - c *_C b) /_C c$)
$- (a /_C c) + b =$ (*if* $c = 0$ *then* b *else* $(- a + c *_C b) /_C c$)
$- (a /_C c) - b =$ (*if* $c = 0$ *then* $- b$ *else* $(- a - c *_C b) /_C c$)
for $a\ b :: 'a ::$ *complex-vector*
by (*auto simp add*: *field-simps*)

lemma *divideC-field-splits-simps-2* [*field-split-simps*]:
$0 < c \Longrightarrow a \leq b /_C c \longleftrightarrow$ (*if* $c > 0$ *then* $c *_C a \leq b$ *else if* $c < 0$ *then* $b \leq c *_C a$ *else* $a \leq 0$)
$0 < c \Longrightarrow a < b /_C c \longleftrightarrow$ (*if* $c > 0$ *then* $c *_C a < b$ *else if* $c < 0$ *then* $b < c *_C a$ *else* $a < 0$)
$0 < c \Longrightarrow b /_C c \leq a \longleftrightarrow$ (*if* $c > 0$ *then* $b \leq c *_C a$ *else if* $c < 0$ *then* $c *_C a$

$\leq b$ *else* $a \geq 0$)
$0 < c \Longrightarrow b /_C c < a \longleftrightarrow$ (*if* $c > 0$ *then* $b < c *_C a$ *else if* $c < 0$ *then* $c *_C a < b$ *else* $a > 0$)
$0 < c \Longrightarrow a \leq - (b /_C c) \longleftrightarrow$ (*if* $c > 0$ *then* $c *_C a \leq - b$ *else if* $c < 0$ *then* $- b \leq c *_C a$ *else* $a \leq 0$)
$0 < c \Longrightarrow a < - (b /_C c) \longleftrightarrow$ (*if* $c > 0$ *then* $c *_C a < - b$ *else if* $c < 0$ *then* $- b < c *_C a$ *else* $a < 0$)
$0 < c \Longrightarrow - (b /_C c) \leq a \longleftrightarrow$ (*if* $c > 0$ *then* $- b \leq c *_C a$ *else if* $c < 0$ *then* $c *_C a \leq - b$ *else* $a \geq 0$)
$0 < c \Longrightarrow - (b /_C c) < a \longleftrightarrow$ (*if* $c > 0$ *then* $- b < c *_C a$ *else if* $c < 0$ *then* $c *_C a < - b$ *else* $a > 0$)
for $a\ b :: 'a ::$ *ordered-complex-vector*
by (*clarsimp intro*!: *field-simps*)+

lemma *scaleC-nonneg-nonneg*: $0 \leq a \Longrightarrow 0 \leq x \Longrightarrow 0 \leq a *_C x$
for $x :: 'a$::*ordered-complex-vector*
using *scaleC-left-mono* [*of 0 x a*] **by** *simp*

lemma *scaleC-nonneg-nonpos*: $0 \leq a \Longrightarrow x \leq 0 \Longrightarrow a *_C x \leq 0$
for $x :: 'a$::*ordered-complex-vector*
using *scaleC-left-mono* [*of x 0 a*] **by** *simp*

lemma *scaleC-nonpos-nonneg*: $a \leq 0 \Longrightarrow 0 \leq x \Longrightarrow a *_C x \leq 0$
for $x :: 'a$::*ordered-complex-vector*
using *scaleC-right-mono* [*of a 0 x*] **by** *simp*

lemma *split-scaleC-neg-le*: $(0 \leq a \wedge x \leq 0) \vee (a \leq 0 \wedge 0 \leq x) \Longrightarrow a *_C x \leq 0$
for $x :: 'a$::*ordered-complex-vector*
by (*auto simp*: *scaleC-nonneg-nonpos scaleC-nonpos-nonneg*)

lemma *cle-add-iff1*: $a *_C e + c \leq b *_C e + d \longleftrightarrow (a - b) *_C e + c \leq d$
for $c\ d\ e :: 'a$::*ordered-complex-vector*
by (*simp add*: *algebra-simps*)

lemma *cle-add-iff2*: $a *_C e + c \leq b *_C e + d \longleftrightarrow c \leq (b - a) *_C e + d$
for $c\ d\ e :: 'a$::*ordered-complex-vector*
by (*simp add*: *algebra-simps*)

lemma *scaleC-left-mono-neg*: $b \leq a \Longrightarrow c \leq 0 \Longrightarrow c *_C a \leq c *_C b$
for $a\ b :: 'a$::*ordered-complex-vector*
by (*drule scaleC-left-mono* [*of - - - c*], *simp-all add*: *less-eq-complex-def*)

lemma *scaleC-right-mono-neg*: $b \leq a \Longrightarrow c \leq 0 \Longrightarrow a *_C c \leq b *_C c$
for $c :: 'a$::*ordered-complex-vector*
by (*drule scaleC-right-mono* [*of - - - c*], *simp-all*)

lemma *scaleC-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a *_C b$
for $b :: 'a$::*ordered-complex-vector*
using *scaleC-right-mono-neg* [*of a 0 b*] **by** *simp*

lemma *split-scaleC-pos-le*: $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \Longrightarrow 0 \leq a *_C b$
for $b :: 'a$::*ordered-complex-vector*
by (*auto simp*: *scaleC-nonneg-nonneg scaleC-nonpos-nonpos*)

lemma *zero-le-scaleC-iff*:
fixes $b :: 'a$::*ordered-complex-vector*
assumes $a \in \mathbb{R}$
shows $0 \leq a *_C b \longleftrightarrow 0 < a \wedge 0 \leq b \vee a < 0 \wedge b \leq 0 \vee a = 0$
(**is** *?lhs* = *?rhs*)
proof (*cases* $a = 0$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof
assume *?lhs*
from ‹$a \neq 0$› **consider** $a > 0 \mid a < 0$
by (*metis assms complex-is-Real-iff less-complex-def less-eq-complex-def not-le order.not-eq-order-implies-strict that*(1) *zero-complex.sel*(2))
then show *?rhs*
proof *cases*
case *1*
with ‹*?lhs*› **have** *inverse a* $*_C$ $0 \leq$ *inverse a* $*_C$ $(a *_C b)$
by (*metis complex-vector.scale-zero-right ordered-complex-vector-class.pos-le-divideC-eq*)
with *1* **show** *?thesis*
by *simp*
next
case *2*
with ‹*?lhs*› **have** $-$ *inverse a* $*_C$ $0 \leq -$ *inverse a* $*_C$ $(a *_C b)$
by (*metis Complex-Vector-Spaces0.neg-le-minus-divideC-eq complex-vector.scale-zero-right neg-le-0-iff-le scaleC-left.minus*)
with *2* **show** *?thesis*
by *simp*
qed
next
assume *?rhs*
then show *?lhs*
using *less-imp-le split-scaleC-pos-le* **by** *auto*
qed
qed

lemma *scaleC-le-0-iff*:
$a *_C b \leq 0 \longleftrightarrow 0 < a \wedge b \leq 0 \vee a < 0 \wedge 0 \leq b \vee a = 0$
if $a \in \mathbb{R}$
for b::$'a$::*ordered-complex-vector*
apply (*insert zero-le-scaleC-iff* [*of* $-a$ b])
using *less-complex-def that* **by** *force*

lemma *scaleC-le-cancel-left*: $c *_C a \leq c *_C b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
if $c \in \mathbb{R}$
for $b :: 'a$::*ordered-complex-vector*
by (*smt* (*verit*, *ccfv-threshold*) *Complex-Vector-Spaces0.neg-divideC-le-eq complex-vector.scale-cancel-left complex-vector.scale-zero-right dual-order.eq-iff dual-order.trans ordered-complex-vector-class.pos-le-divideC-eq that zero-le-scaleC-iff*)

lemma *scaleC-le-cancel-left-pos*: $0 < c \Longrightarrow c *_C a \leq c *_C b \longleftrightarrow a \leq b$
for $b :: 'a$::*ordered-complex-vector*
by (*simp add*: *complex-is-Real-iff less-complex-def scaleC-le-cancel-left*)

lemma *scaleC-le-cancel-left-neg*: $c < 0 \Longrightarrow c *_C a \leq c *_C b \longleftrightarrow b \leq a$
for $b :: 'a$::*ordered-complex-vector*
by (*simp add*: *complex-is-Real-iff less-complex-def scaleC-le-cancel-left*)

lemma *scaleC-left-le-one-le*: $0 \leq x \Longrightarrow a \leq 1 \Longrightarrow a *_C x \leq x$
for $x :: 'a$::*ordered-complex-vector* **and** $a :: complex$
using *scaleC-right-mono*[*of a 1 x*] **by** *simp*

6.5 Complex normed vector spaces

class *complex-normed-vector* = *complex-vector* + *sgn-div-norm* + *dist-norm* + *uniformity-dist* + *open-uniformity* +
real-normed-vector +
assumes *norm-scaleC* [*simp*]: *norm* (*scaleC a x*) = *cmod a* * *norm x*
begin

end

class *complex-normed-algebra* = *complex-algebra* + *complex-normed-vector* +
real-normed-algebra

class *complex-normed-algebra-1* = *complex-algebra-1* + *complex-normed-algebra* +
real-normed-algebra-1

lemma (**in** *complex-normed-algebra-1*) *scaleC-power* [*simp*]: (*scaleC x y*) ^ *n* = *scaleC* (*x*^*n*) (*y*^*n*)
by (*induct n*) (*simp-all add*: *mult-ac*)

class *complex-normed-div-algebra* = *complex-div-algebra* + *complex-normed-vector* +
real-normed-div-algebra

class *complex-normed-field* = *complex-field* + *complex-normed-div-algebra*

subclass (**in** *complex-normed-field*) *real-normed-field* **..**

instance *complex-normed-div-algebra* < *complex-normed-algebra-1* **..**

context *complex-normed-vector* **begin**

end

lemma *dist-scaleC* [*simp*]: *dist* (*x* $*_C$ *a*) (*y* $*_C$ *a*) = |*x* − *y*| ∗ *norm a*
for *a* :: 'a::*complex-normed-vector*
by (*metis dist-scaleR scaleR-scaleC*)

lemma *norm-of-complex* [*simp*]: *norm* (*of-complex c* :: 'a::*complex-normed-algebra-1*)
= *cmod c*
by (*simp add*: *of-complex-def*)

lemma *norm-of-complex-add1* [*simp*]: *norm* (*of-complex x* + *1* :: 'a :: *complex-normed-div-algebra*)
= *cmod* (*x* + *1*)
by (*metis norm-of-complex of-complex-1 of-complex-add*)

lemma *norm-of-complex-addn* [*simp*]:
norm (*of-complex x* + *numeral b* :: 'a :: *complex-normed-div-algebra*) = *cmod* (*x*
+ *numeral b*)
by (*metis norm-of-complex of-complex-add of-complex-numeral*)

lemma *norm-of-complex-diff* [*simp*]:
norm (*of-complex b* − *of-complex a* :: 'a::*complex-normed-algebra-1*) ≤ *cmod* (*b*
− *a*)
by (*metis norm-of-complex of-complex-diff order-refl*)

6.6 Metric spaces

Every normed vector space is a metric space.

6.7 Class instances for complex numbers

instantiation *complex* :: *complex-normed-field*
begin

instance
apply *intro-classes*
by (*simp add*: *norm-mult*)

end

declare *uniformity-Abort*[**where** *'a=complex*, *code*]

lemma *dist-of-complex* [*simp*]: *dist* (*of-complex x* :: *'a*) (*of-complex y*) = *dist x y*
for *a* :: *'a*::*complex-normed-div-algebra*
by (*metis dist-norm norm-of-complex of-complex-diff*)

declare [[*code abort*: *open* :: *complex set* ⇒ *bool*]]

lemma *closed-complex-atMost*: ‹*closed* {..*a*::*complex*}›
proof −
have ‹{..*a*} = *Im* −‘ {*Im a*} ∩ *Re* −‘ {..*Re a*}›
by (*auto simp*: *less-eq-complex-def*)
also have ‹*closed* . . . ›
by (*auto intro*!: *closed-Int closed-vimage continuous-on-Im continuous-on-Re*)
finally show *?thesis*
by −
qed

lemma *closed-complex-atLeast*: ‹*closed* {*a*::*complex*..}›
proof −
have ‹{*a*..} = *Im* −‘ {*Im a*} ∩ *Re* −‘ {*Re a*..}›
by (*auto simp*: *less-eq-complex-def*)
also have ‹*closed* . . . ›
by (*auto intro*!: *closed-Int closed-vimage continuous-on-Im continuous-on-Re*)
finally show *?thesis*
by −
qed

lemma *closed-complex-atLeastAtMost*: ‹*closed* {*a*::*complex* .. *b*}›
proof (*cases* ‹*Im a* = *Im b*›)
case *True*
have ‹{*a*..*b*} = *Im* −‘ {*Im a*} ∩ *Re* −‘ {*Re a*..*Re b*}›
by (*auto simp add*: *less-eq-complex-def intro*!: *True*)
also have ‹*closed* . . . ›
by (*auto intro*!: *closed-Int closed-vimage continuous-on-Im continuous-on-Re*)
finally show *?thesis*
by −
next
case *False*
then have ∗: ‹{*a*..*b*} = {}›
using *less-eq-complex-def* **by** *auto*
show *?thesis*
by (*simp add*: ∗)
qed

6.8 Sign function

lemma *sgn-scaleC*: *sgn* (*scaleC r x*) = *scaleC* (*sgn r*) (*sgn x*)
for *x* :: *'a*::*complex-normed-vector*
by (*simp add*: *scaleR-scaleC sgn-div-norm ac-simps*)

lemma *sgn-of-complex*: *sgn* (*of-complex r* :: *'a*::*complex-normed-algebra-1*) = *of-complex* (*sgn r*)
unfolding *of-complex-def* **by** (*simp only*: *sgn-scaleC sgn-one*)

lemma *complex-sgn-eq*: *sgn x* = *x* / |*x*|
for *x* :: *complex*
by (*simp add*: *abs-complex-def scaleR-scaleC sgn-div-norm divide-inverse*)

lemma *czero-le-sgn-iff* [*simp*]: *0* ≤ *sgn x* ⟷ *0* ≤ *x*
for *x* :: *complex*
using *cmod-eq-Re divide-eq-0-iff less-eq-complex-def* **by** *auto*

lemma *csgn-le-0-iff* [*simp*]: *sgn x* ≤ *0* ⟷ *x* ≤ *0*
for *x* :: *complex*
by (*smt* (*verit*, *best*) *czero-le-sgn-iff Im-sgn Re-sgn divide-eq-0-iff dual-order.eq-iff less-eq-complex-def sgn-zero-iff zero-complex.sel*(*1*) *zero-complex.sel*(*2*))

6.9 Bounded Linear and Bilinear Operators

lemma *clinearI*: *clinear f*
if ⋀*b1 b2*. *f* (*b1* + *b2*) = *f b1* + *f b2*
⋀*r b*. *f* (*r* *$_C$ *b*) = *r* *$_C$ *f b*
using *that*
by *unfold-locales* (*auto simp*: *algebra-simps*)

lemma *clinear-iff*:
clinear f ⟷ (∀ *x y*. *f* (*x* + *y*) = *f x* + *f y*) ∧ (∀ *c x*. *f* (*c* *$_C$ *x*) = *c* *$_C$ *f x*)
(**is** *clinear f* ⟷ *?rhs*)
proof
assume *clinear f*
then interpret *f*: *clinear f* **.**
show *?rhs*
by (*simp add*: *f.add f.scale complex-vector.linear-scale f.clinear-axioms*)
next
assume *?rhs*
then show *clinear f* **by** (*intro clinearI*) *auto*
qed

lemmas *clinear-scaleC-left* = *complex-vector.linear-scale-left*
lemmas *clinear-imp-scaleC* = *complex-vector.linear-imp-scale*

corollary *complex-clinearD*:
fixes *f* :: *complex* ⇒ *complex*
assumes *clinear f* **obtains** *c* **where** *f* = (∗) *c*

by (*rule clinear-imp-scaleC* [*OF assms*]) (*force simp*: *scaleC-conv-of-complex*)

lemma *clinear-times-of-complex*: *clinear* (λ*x*. *a* ∗ *of-complex x*)
by (*auto intro*!: *clinearI simp*: *distrib-left*)
(*metis mult-scaleC-right scaleC-conv-of-complex*)

locale *bounded-clinear* = *clinear f* **for** *f* :: ′*a*::*complex-normed-vector* ⇒ ′*b*::*complex-normed-vector*
+
assumes *bounded*: ∃ *K*. ∀ *x*. *norm* (*f x*) ≤ *norm x* ∗ *K*
begin

sublocale *real*: *bounded-linear*
— Gives access to all lemmas from *bounded-linear* using prefix *real*.
apply *standard*
by (*auto simp add*: *add scaleR-scaleC scale bounded*)

lemmas *pos-bounded* = *real.pos-bounded*

lemmas *nonneg-bounded* = *real.nonneg-bounded*

lemma *clinear*: *clinear f*
by (*fact local.clinear-axioms*)

end

lemma *bounded-clinear-intro*:
assumes ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y*
and ⋀*r x*. *f* (*scaleC r x*) = *scaleC r* (*f x*)
and ⋀*x*. *norm* (*f x*) ≤ *norm x* ∗ *K*
shows *bounded-clinear f*
by *standard* (*blast intro*: *assms*)+

locale *bounded-cbilinear* =
fixes *prod* :: ′*a*::*complex-normed-vector* ⇒ ′*b*::*complex-normed-vector* ⇒ ′*c*::*complex-normed-vector*
(**infixl** ‹∗∗› *70*)
assumes *add-left*: *prod* (*a* + *a*′) *b* = *prod a b* + *prod a*′ *b*
and *add-right*: *prod a* (*b* + *b*′) = *prod a b* + *prod a b*′
and *scaleC-left*: *prod* (*scaleC r a*) *b* = *scaleC r* (*prod a b*)
and *scaleC-right*: *prod a* (*scaleC r b*) = *scaleC r* (*prod a b*)
and *bounded*: ∃ *K*. ∀ *a b*. *norm* (*prod a b*) ≤ *norm a* ∗ *norm b* ∗ *K*
begin

sublocale *real*: *bounded-bilinear*
— Gives access to all lemmas from *bounded-bilinear* using prefix *real*.
apply *standard*
by (*auto simp add*: *add-left add-right scaleR-scaleC scaleC-left scaleC-right bounded*)

lemmas *pos-bounded* = *real.pos-bounded*
lemmas *nonneg-bounded* = *real.nonneg-bounded*
lemmas *additive-right* = *real.additive-right*
lemmas *additive-left* = *real.additive-left*
lemmas *zero-left* = *real.zero-left*
lemmas *zero-right* = *real.zero-right*
lemmas *minus-left* = *real.minus-left*
lemmas *minus-right* = *real.minus-right*
lemmas *diff-left* = *real.diff-left*
lemmas *diff-right* = *real.diff-right*
lemmas *sum-left* = *real.sum-left*
lemmas *sum-right* = *real.sum-right*
lemmas *prod-diff-prod* = *real.prod-diff-prod*

lemma *bounded-clinear-left*: *bounded-clinear* $(\lambda a.\ a ** b)$
proof −
  obtain K **where** $\bigwedge a\ b.\ norm\ (a ** b) \leq norm\ a * norm\ b * K$
    using *pos-bounded* **by** *blast*
  then show *?thesis*
    by (*rule-tac* K=*norm* $b * K$ **in** *bounded-clinear-intro*) (*auto simp*: *algebra-simps scaleC-left add-left*)
qed

lemma *bounded-clinear-right*: *bounded-clinear* $(\lambda b.\ a ** b)$
proof −
  obtain K **where** $\bigwedge a\ b.\ norm\ (a ** b) \leq norm\ a * norm\ b * K$
    using *pos-bounded* **by** *blast*
  then show *?thesis*
    by (*rule-tac* K=*norm* $a * K$ **in** *bounded-clinear-intro*) (*auto simp*: *algebra-simps scaleC-right add-right*)
qed

lemma *flip*: *bounded-cbilinear* $(\lambda x\ y.\ y ** x)$
proof
  show $\exists K.\ \forall a\ b.\ norm\ (b ** a) \leq norm\ a * norm\ b * K$
    by (*metis bounded mult.commute*)
qed (*simp-all add*: *add-right add-left scaleC-right scaleC-left*)

lemma *comp1*:
  assumes *bounded-clinear g*
  shows *bounded-cbilinear* $(\lambda x.\ (**)\ (g\ x))$
proof
  interpret *g*: *bounded-clinear g* **by** *fact*
  show $\bigwedge a\ a'\ b.\ g\ (a + a') ** b = g\ a ** b + g\ a' ** b$
    $\bigwedge a\ b\ b'.\ g\ a ** (b + b') = g\ a ** b + g\ a ** b'$
    $\bigwedge r\ a\ b.\ g\ (r *_C a) ** b = r *_C (g\ a ** b)$
    $\bigwedge a\ r\ b.\ g\ a ** (r *_C b) = r *_C (g\ a ** b)$
    by (*auto simp*: *g.add add-left add-right g.scaleC scaleC-left scaleC-right*)

have *bounded-bilinear* ($\lambda a\ b.\ g\ a\ **\ b$)
using *g.real.bounded-linear* **by** (*rule real.comp1*)
then show $\exists K.\ \forall a\ b.\ norm\ (g\ a\ **\ b) \leq norm\ a * norm\ b * K$
by (*rule bounded-bilinear.bounded*)
qed

lemma *comp*: *bounded-clinear f* $\Longrightarrow$ *bounded-clinear g* $\Longrightarrow$ *bounded-cbilinear* (λx *y. f x ** g y*)
by (*rule bounded-cbilinear.flip*[*OF bounded-cbilinear.comp1*[*OF bounded-cbilinear.flip*[*OF comp1*]]])

end

lemma *bounded-clinear-ident*[*simp*]: *bounded-clinear* ($\lambda x.\ x$)
by *standard* (*auto intro*!: *exI*[*of - 1*])

lemma *bounded-clinear-zero*[*simp*]: *bounded-clinear* ($\lambda x.\ 0$)
by *standard* (*auto intro*!: *exI*[*of - 1*])

lemma *bounded-clinear-add*:
assumes *bounded-clinear f*
and *bounded-clinear g*
shows *bounded-clinear* ($\lambda x.\ f\ x + g\ x$)
proof −
interpret *f*: *bounded-clinear f* **by** *fact*
interpret *g*: *bounded-clinear g* **by** *fact*
show *?thesis*
proof
from *f.bounded* **obtain** *Kf* **where** *Kf*: $norm\ (f\ x) \leq norm\ x * Kf$ **for** *x*
by *blast*
from *g.bounded* **obtain** *Kg* **where** *Kg*: $norm\ (g\ x) \leq norm\ x * Kg$ **for** *x*
by *blast*
show $\exists K.\ \forall x.\ norm\ (f\ x + g\ x) \leq norm\ x * K$
using *add-mono*[*OF Kf Kg*]
by (*intro exI*[*of - Kf + Kg*]) (*auto simp*: *field-simps intro*: *norm-triangle-ineq order-trans*)
qed (*simp-all add*: *f.add g.add f.scaleC g.scaleC scaleC-add-right*)
qed

lemma *bounded-clinear-minus*:
assumes *bounded-clinear f*
shows *bounded-clinear* ($\lambda x.\ -\ f\ x$)
proof −
interpret *f*: *bounded-clinear f* **by** *fact*
show *?thesis*
by *unfold-locales* (*simp-all add*: *f.add f.scaleC f.bounded*)
qed

lemma *bounded-clinear-sub*: *bounded-clinear f* $\Longrightarrow$ *bounded-clinear g* $\Longrightarrow$ *bounded-clinear*

$(\lambda x.\ f\ x - g\ x)$
using *bounded-clinear-add*[*of f* $\lambda x.\ -\ g\ x$] *bounded-clinear-minus*[*of g*]
by (*auto simp*: *algebra-simps*)

lemma *bounded-clinear-sum*:
fixes $f :: 'i \Rightarrow 'a$::*complex-normed-vector* $\Rightarrow 'b$::*complex-normed-vector*
shows $(\bigwedge i.\ i \in I \Longrightarrow$ *bounded-clinear* $(f\ i)) \Longrightarrow$ *bounded-clinear* $(\lambda x.\ \sum i{\in}I.\ f\ i\ x)$
by (*induct I rule*: *infinite-finite-induct*) (*auto intro*!: *bounded-clinear-add*)

lemma *bounded-clinear-compose*:
assumes *bounded-clinear f*
and *bounded-clinear g*
shows *bounded-clinear* $(\lambda x.\ f\ (g\ x))$
proof
interpret *f*: *bounded-clinear f* **by** *fact*
interpret *g*: *bounded-clinear g* **by** *fact*
show $f\ (g\ (x + y)) = f\ (g\ x) + f\ (g\ y)$ **for** *x y*
by (*simp only*: *f.add g.add*)
show $f\ (g\ (scaleC\ r\ x)) = scaleC\ r\ (f\ (g\ x))$ **for** *r x*
by (*simp only*: *f.scaleC g.scaleC*)
from *f.pos-bounded* **obtain** *Kf* **where** *f*: $\bigwedge x.\ norm\ (f\ x) \leq norm\ x * Kf$ **and** *Kf*: $0 < Kf$
by *blast*
from *g.pos-bounded* **obtain** *Kg* **where** *g*: $\bigwedge x.\ norm\ (g\ x) \leq norm\ x * Kg$
by *blast*
show $\exists K.\ \forall x.\ norm\ (f\ (g\ x)) \leq norm\ x * K$
proof (*intro exI allI*)
fix *x*
have $norm\ (f\ (g\ x)) \leq norm\ (g\ x) * Kf$
using *f* **.**
also have $\ldots \leq (norm\ x * Kg) * Kf$
using *g Kf* [*THEN order-less-imp-le*] **by** (*rule mult-right-mono*)
also have $(norm\ x * Kg) * Kf = norm\ x * (Kg * Kf)$
by (*rule mult.assoc*)
finally show $norm\ (f\ (g\ x)) \leq norm\ x * (Kg * Kf)$ **.**
qed
qed

lemma *bounded-cbilinear-mult*: *bounded-cbilinear* $((*) :: 'a \Rightarrow 'a \Rightarrow 'a$::*complex-normed-algebra*)
proof (*rule bounded-cbilinear.intro*)
show $\exists K.\ \forall a\ b$::$'a.\ norm\ (a * b) \leq norm\ a * norm\ b * K$
by (*rule-tac x=1* **in** *exI*) (*simp add*: *norm-mult-ineq*)
qed (*auto simp*: *algebra-simps*)

lemma *bounded-clinear-mult-left*: *bounded-clinear* $(\lambda x$::$'a$::*complex-normed-algebra*.\ x * y)$
using *bounded-cbilinear-mult*
by (*rule bounded-cbilinear.bounded-clinear-left*)

lemma *bounded-clinear-mult-right*: *bounded-clinear* (λy::'*a*::*complex-normed-algebra*. *x* * *y*)
using *bounded-cbilinear-mult*
by (*rule bounded-cbilinear.bounded-clinear-right*)

lemmas *bounded-clinear-mult-const* =
bounded-clinear-mult-left [*THEN bounded-clinear-compose*]

lemmas *bounded-clinear-const-mult* =
bounded-clinear-mult-right [*THEN bounded-clinear-compose*]

lemma *bounded-clinear-divide*: *bounded-clinear* (λx. *x* / *y*)
for *y* :: '*a*::*complex-normed-field*
unfolding *divide-inverse* **by** (*rule bounded-clinear-mult-left*)

lemma *bounded-cbilinear-scaleC*: *bounded-cbilinear scaleC*
proof (*rule bounded-cbilinear.intro*)
obtain *K* **where** *K*: ‹$\forall$ *a* (*b*::'*a*). *norm b* ≤ *norm b* * *K*›
using *less-eq-real-def* **by** *auto*
show $\exists$ *K*. $\forall$ *a* (*b*::'*a*). *norm* (*a* *$_C$ *b*) ≤ *norm a* * *norm b* * *K*
apply (*rule exI*[**where** *x*=*K*]) **using** *K*
by (*metis norm-scaleC*)
qed (*auto simp*: *algebra-simps*)

lemma *bounded-clinear-scaleC-left*: *bounded-clinear* (λc. *scaleC c x*)
using *bounded-cbilinear-scaleC*
by (*rule bounded-cbilinear.bounded-clinear-left*)

lemma *bounded-clinear-scaleC-right*: *bounded-clinear* (λx. *scaleC c x*)
using *bounded-cbilinear-scaleC*
by (*rule bounded-cbilinear.bounded-clinear-right*)

lemmas *bounded-clinear-scaleC-const* =
bounded-clinear-scaleC-left[*THEN bounded-clinear-compose*]

lemmas *bounded-clinear-const-scaleC* =
bounded-clinear-scaleC-right[*THEN bounded-clinear-compose*]

lemma *bounded-clinear-of-complex*: *bounded-clinear* (λr. *of-complex r*)
unfolding *of-complex-def* **by** (*rule bounded-clinear-scaleC-left*)

lemma *complex-bounded-clinear*: *bounded-clinear f* ⟷ ($\exists$ *c*::*complex*. *f* = (λx. *x* * *c*))
for *f* :: *complex* ⇒ *complex*
proof −
{
fix *x*
assume *bounded-clinear f*

then interpret *bounded-clinear f* **.**
from *scaleC*[*of x 1*] **have** $f\ x = x * f\ 1$
by *simp*
}
then show *?thesis*
by (*auto intro*: *exI*[*of - f 1*] *bounded-clinear-mult-left*)
qed

6.9.1 Limits of Sequences

6.10 Cauchy sequences

lemma *cCauchy-iff2*: *Cauchy X* $\longleftrightarrow$ $(\forall j.\ (\exists M.\ \forall m \geq M.\ \forall n \geq M.\ cmod\ (X\ m - X\ n) < inverse\ (real\ (Suc\ j))))$
by (*simp only*: *metric-Cauchy-iff2 dist-complex-def*)

6.11 The set of complex numbers is a complete metric space

Proof that Cauchy sequences converge based on the one from http://pirate.shu.edu/~wachsmut/ira/numseq/proofs/cauconv.html

If sequence X is Cauchy, then its limit is the lub of $\{r.\ \exists N.\ \forall n{\geq}N.\ r < X\ n\}$

lemma *complex-increasing-LIMSEQ*:
fixes $f :: nat \Rightarrow complex$
assumes *inc*: $\bigwedge n.\ f\ n \leq f\ (Suc\ n)$
and *bdd*: $\bigwedge n.\ f\ n \leq l$
and *en*: $\bigwedge e.\ 0 < e \Longrightarrow \exists n.\ l \leq f\ n + e$
shows $f \longrightarrow l$
proof −
have ‹$(\lambda n.\ Re\ (f\ n)) \longrightarrow Re\ l$›
apply (*rule increasing-LIMSEQ*)
using *assms* **apply** (*auto simp*: *less-eq-complex-def less-complex-def*)
by (*metis Im-complex-of-real Re-complex-of-real*)
moreover have ‹$Im\ (f\ n) = Im\ l$› **for** *n*
using *bdd* **by** (*auto simp*: *less-eq-complex-def*)
then have ‹$(\lambda n.\ Im\ (f\ n)) \longrightarrow Im\ l$›
by *auto*
ultimately show ‹$f \longrightarrow l$›
by (*simp add*: *tendsto-complex-iff*)
qed

lemma *complex-Cauchy-convergent*:
fixes $X :: nat \Rightarrow complex$
assumes *X*: *Cauchy X*
shows *convergent X*
using *assms* **by** (*rule Cauchy-convergent*)

instance *complex* :: *complete-space*

by *intro-classes* (*rule complex-Cauchy-convergent*)

class *cbanach* = *complex-normed-vector* + *complete-space*

subclass (**in** *cbanach*) *banach* **..**

instance *complex* :: *banach* **..**

end

7 *Complex-Vector-Spaces* − Complex Vector Spaces

theory *Complex-Vector-Spaces*
imports
HOL−Analysis.Elementary-Topology
HOL−Analysis.Operator-Norm
HOL−Analysis.Elementary-Normed-Spaces
HOL−Library.Set-Algebras
HOL−Analysis.Starlike
HOL−Types-To-Sets.Types-To-Sets
HOL−Library.Complemented-Lattices
HOL−Library.Function-Algebras

Extra-Vector-Spaces
Extra-Ordered-Fields
Extra-Operator-Norm
Extra-General

Complex-Vector-Spaces0
begin

bundle *norm-syntax* **begin**
notation *norm* (‹‖-‖›)
end

unbundle *lattice-syntax*

7.1 Misc

lemma (**in** *vector-space*) *span-image-scale'*:
— Strengthening of *vector-space.span-image-scale* without the condition *finite S*
assumes *nz*: $\bigwedge x.\ x \in S \Longrightarrow c\ x \neq 0$
shows *span* $((\lambda x.\ c\ x\ *s\ x)\ `\ S) = span\ S$
proof

have ‹((λx. c x ∗s x) ‘ S) ⊆ span S›
by (*metis* (*mono-tags*, *lifting*) *image-subsetI in-mono local.span-superset local.subspace-scale local.subspace-span*)
then show ‹*span* ((λx. c x ∗s x) ‘ S) ⊆ span S›
by (*simp add*: *local.span-minimal*)
next
have ‹x ∈ *span* ((λx. c x ∗s x) ‘ S)› **if** ‹x ∈ S› **for** x
proof −
have ‹x = *inverse* (c x) ∗s c x ∗s x›
by (*simp add*: *nz that*)
moreover have ‹c x ∗s x ∈ (λx. c x ∗s x) ‘ S›
using *that* **by** *blast*
ultimately show *?thesis*
by (*metis local.span-base local.span-scale*)
qed
then show ‹*span* S ⊆ *span* ((λx. c x ∗s x) ‘ S)›
by (*simp add*: *local.span-minimal subsetI*)
qed

lemma (**in** *scaleC*) *scaleC-real*: **assumes** r∈ℝ **shows** r ∗$_C$ x = *Re* r ∗$_R$ x
unfolding *scaleR-scaleC* **using** *assms* **by** *simp*

lemma *of-complex-of-real-eq* [*simp*]: *of-complex* (*of-real* n) = *of-real* n
unfolding *of-complex-def of-real-def* **unfolding** *scaleR-scaleC* **by** *simp*

lemma *Complexs-of-real* [*simp*]: *of-real* r ∈ ℂ
unfolding *Complexs-def of-real-def of-complex-def*
apply (*subst scaleR-scaleC*) **by** *simp*

lemma *Reals-in-Complexs*: ℝ ⊆ ℂ
unfolding *Reals-def* **by** *auto*

lemma (**in** *bounded-clinear*) *bounded-linear*: *bounded-linear* f
by *standard*

lemma *clinear-times*: *clinear* (λx. c ∗ x)
for c :: ′a::*complex-algebra*
by (*auto simp*: *clinearI distrib-left*)

lemma (**in** *clinear*) *linear*: ‹*linear* f›
by *standard*

lemma *bounded-clinearI*:
assumes ‹⋀b1 b2. f (b1 + b2) = f b1 + f b2›
assumes ‹⋀r b. f (r ∗$_C$ b) = r ∗$_C$ f b›
assumes ‹⋀x. norm (f x) ≤ norm x ∗ K›
shows *bounded-clinear* f
using *assms* **by** (*auto intro*!: *exI bounded-clinear.intro clinearI simp*: *bounded-clinear-axioms-def*)

lemma *bounded-clinear-id*[*simp*]: ‹*bounded-clinear id*›
by (*simp add*: *id-def*)

lemma *bounded-clinear-0*[*simp*]: ‹*bounded-clinear 0*›
by (*auto intro*!: *bounded-clinearI*[**where** *K*=*0*])

definition *cbilinear* :: ‹('*a*::*complex-vector* ⇒ '*b*::*complex-vector* ⇒ '*c*::*complex-vector*) ⇒ *bool*›
where ‹*cbilinear* = (λ *f*. (∀ *y*. *clinear* (λ *x*. *f x y*)) ∧ (∀ *x*. *clinear* (λ *y*. *f x y*)))›

lemma *cbilinear-add-left*:
assumes ‹*cbilinear f*›
shows ‹*f* (*a* + *b*) *c* = *f a c* + *f b c*›
by (*smt* (*verit*, *del-insts*) *assms cbilinear-def complex-vector.linear-add*)

lemma *cbilinear-add-right*:
assumes ‹*cbilinear f*›
shows ‹*f a* (*b* + *c*) = *f a b* + *f a c*›
by (*smt* (*verit*, *del-insts*) *assms cbilinear-def complex-vector.linear-add*)

lemma *cbilinear-times*:
fixes *g'* :: ‹'*a*::*complex-vector* ⇒ *complex*› **and** *g* :: ‹'*b*::*complex-vector* ⇒ *complex*›
assumes ‹⋀ *x y*. *h x y* = (*g' x*)∗(*g y*)› **and** ‹*clinear g*› **and** ‹*clinear g'*›
shows ‹*cbilinear h*›
proof −
have *w1*: *h* (*b1* + *b2*) *y* = *h b1 y* + *h b2 y*
for *b1* :: '*a*
and *b2* :: '*a*
and *y*
proof−
have ‹*h* (*b1* + *b2*) *y* = *g'* (*b1* + *b2*) ∗ *g y*›
using ‹⋀ *x y*. *h x y* = (*g' x*)∗(*g y*)›
by *auto*
also have ‹... = (*g' b1* + *g' b2*) ∗ *g y*›
using ‹*clinear g'*›
unfolding *clinear-def*
by (*simp add*: *assms*(*3*) *complex-vector.linear-add*)
also have ‹... = *g' b1* ∗ *g y* + *g' b2* ∗ *g y*›
by (*simp add*: *ring-class.ring-distribs*(*2*))
also have ‹... = *h b1 y* + *h b2 y*›
using *assms*(*1*) **by** *auto*
finally show *?thesis* **by** *blast*
qed
have *w2*: *h* (*r* ∗*C* *b*) *y* = *r* ∗*C* *h b y*
for *r* :: *complex*
and *b* :: '*a*
and *y*

```
proof−
  have ‹h (r *C b) y = g' (r *C b) * g y›
    by (simp add: assms(1))
  also have ‹... = r *C (g' b * g y)›
    by (simp add: assms(3) complex-vector.linear-scale)
  also have ‹... = r *C (h b y)›
    by (simp add: assms(1))
  finally show ?thesis by blast
qed
have clinear (λx. h x y)
  for y :: 'b
  unfolding clinear-def
  by (meson clinearI clinear-def w1 w2)
hence t2: ∀ y. clinear (λx. h x y)
  by simp
have v1: h x (b1 + b2) = h x b1 + h x b2
  for b1 :: 'b
    and b2 :: 'b
    and x
proof−
  have ‹h x (b1 + b2)  = g' x * g (b1 + b2)›
    using ‹⋀ x y. h x y = (g' x)*(g y)›
    by auto
  also have ‹... = g' x * (g b1 + g b2)›
    using ‹clinear g'›
    unfolding clinear-def
    by (simp add: assms(2) complex-vector.linear-add)
  also have ‹... = g' x * g b1 + g' x * g b2›
    by (simp add: ring-class.ring-distribs(1))
  also have ‹... = h x b1 + h x b2›
    using assms(1) by auto
  finally show ?thesis by blast
qed

have v2:  h x (r *C b) = r *C h x b
  for r :: complex
    and b :: 'b
    and x
proof−
  have ‹h x (r *C b) =  g' x * g (r *C b)›
    by (simp add: assms(1))
  also have ‹... = r *C (g' x * g b)›
    by (simp add: assms(2) complex-vector.linear-scale)
  also have ‹... = r *C (h x b)›
    by (simp add: assms(1))
  finally show ?thesis by blast
qed
have Vector-Spaces.linear (*C) (*C) (h x)
  for x :: 'a
```

using *v1 v2*
by (*meson clinearI clinear-def*)
hence *t1*: $\forall x.$ *clinear* (*h x*)
unfolding *clinear-def*
by *simp*
show *?thesis*
unfolding *cbilinear-def*
by (*simp add*: *t1 t2*)
qed

lemma *csubspace-is-subspace*: *csubspace A* $\Longrightarrow$ *subspace A*
apply (*rule subspaceI*)
by (*auto simp*: *complex-vector.subspace-def scaleR-scaleC*)

lemma *span-subset-cspan*: *span A* $\subseteq$ *cspan A*
unfolding *span-def complex-vector.span-def*
by (*simp add*: *csubspace-is-subspace hull-antimono*)

lemma *cindependent-implies-independent*:
assumes *cindependent* (*S*::*'a*::*complex-vector set*)
shows *independent S*
using *assms* **unfolding** *dependent-def complex-vector.dependent-def*
using *span-subset-cspan* **by** *blast*

lemma *cspan-singleton*: *cspan* $\{x\} = \{\alpha *_C x | \alpha.\ True\}$
proof −
have ‹*cspan* $\{x\} = \{y.\ y \in$ *cspan* $\{x\}\}$›
by *auto*
also have ‹$\ldots = \{\alpha *_C x | \alpha.\ True\}$›
apply (*subst complex-vector.span-breakdown-eq*)
by *auto*
finally show *?thesis*
by −
qed

lemma *cspan-as-span*:
cspan (*B*::*'a*::*complex-vector set*) = *span* (*B* $\cup$ *scaleC* i ‘ *B*)
proof (*intro set-eqI iffI*)
let *?cspan* = *complex-vector.span*
let *?rspan* = *real-vector.span*
fix ψ
assume *cspan*: $\psi \in$ *?cspan B*
have $\exists B'\ r.$ *finite* $B' \wedge B' \subseteq B \wedge \psi = (\sum b \in B'.\ r\ b *_C b)$
using *complex-vector.span-explicit*[*of B*] *cspan*
by *auto*
then obtain $B'\ r$ **where** *finite* B' **and** $B' \subseteq B$ **and** ψ-*explicit*: $\psi = (\sum b \in B'.$
$r\ b *_C b)$

by *atomize-elim*
define R **where** $R = B \cup scaleC\ \mathrm{i}\ `\ B$

have *x2*: (*case x of* $(b, i) \Rightarrow$ *if i*
then Im $(r\ b) *_R \mathrm{i} *_C b$
else Re $(r\ b) *_R b) \in span\ (B \cup (*_C)\ \mathrm{i}\ `\ B)$
if $x \in B' \times (UNIV::bool\ set)$
for $x :: 'a \times bool$
using *that* ‹$B' \subseteq B$› **by** (*auto simp add*: *real-vector.span-base real-vector.span-scale subset-iff*)
have *x1*: $\psi = (\sum x{\in}B'.\ \sum i{\in}UNIV.$ *if i then Im* $(r\ x) *_R \mathrm{i} *_C x$ *else Re* $(r\ x) *_R x)$
if $\bigwedge b.\ r\ b *_C b = Re\ (r\ b) *_R b + Im\ (r\ b) *_R \mathrm{i} *_C b$
using *that* **by** (*simp add*: *UNIV-bool* ψ*-explicit*)
moreover have $r\ b *_C b = Re\ (r\ b) *_R b + Im\ (r\ b) *_R \mathrm{i} *_C b$ **for** b
using *complex-eq scaleC-add-left scaleC-scaleC scaleR-scaleC*
by (*metis* (*no-types*, *lifting*) *complex-of-real-i i-complex-of-real*)
ultimately have $\psi = (\sum (b,i){\in}(B'{\times}UNIV).$ *if i then Im* $(r\ b) *_R (\mathrm{i} *_C b)$ *else Re* $(r\ b) *_R b)$
by (*simp add*: *sum.cartesian-product*)
also have $\ldots \in$ *?rspan R*
unfolding *R-def*
using *x2*
by (*rule real-vector.span-sum*)
finally show $\psi \in$ *?rspan R* **by** −
next
let *?cspan* = *complex-vector.span*
let *?rspan* = *real-vector.span*
define R **where** $R = B \cup scaleC\ \mathrm{i}\ `\ B$
fix ψ
assume *rspan*: $\psi \in$ *?rspan R*
have *subspace* $\{a.\ a \in cspan\ B\}$
by (*rule real-vector.subspaceI*, *auto simp add*: *complex-vector.span-zero complex-vector.span-add-eq2 complex-vector.span-scale scaleR-scaleC*)
moreover have $x \in cspan\ B$
if $x \in R$
for $x :: 'a$
using *that R-def complex-vector.span-base complex-vector.span-scale* **by** *fastforce*
ultimately show $\psi \in$ *?cspan B*
using *real-vector.span-induct rspan* **by** *blast*
qed

lemma *isomorphic-equal-cdim*:
assumes *lin-f*: ‹*clinear f*›
assumes *inj-f*: ‹*inj-on f* (*cspan S*)›
assumes *im-S*: ‹$f\ `\ S = T$›
shows ‹*cdim S* = *cdim T*›

proof −
obtain *SB* **where** *SB-span*: *cspan SB* = *cspan S* **and** *indep-SB*: ‹*cindependent SB*›
by (*metis complex-vector.basis-exists complex-vector.span-mono complex-vector.span-span subset-antisym*)
with *lin-f inj-f* **have** *indep-fSB*: ‹*cindependent* (*f* ‘ *SB*)›
apply (*rule-tac complex-vector.linear-independent-injective-image*)
by *auto*
from *lin-f* **have** ‹*cspan* (*f* ‘ *SB*) = *f* ‘ *cspan SB*›
by (*meson complex-vector.linear-span-image*)
also from *SB-span lin-f* **have** ‹. . . = *cspan T*›
by (*metis complex-vector.linear-span-image im-S*)
finally have ‹*cdim T* = *card* (*f* ‘ *SB*)›
using *indep-fSB complex-vector.dim-eq-card* **by** *blast*
also have ‹. . . = *card SB*›
apply (*rule card-image*) **using** *inj-f*
by (*metis SB-span complex-vector.linear-inj-on-span-iff-independent-image indep-fSB lin-f*)
also have ‹. . . = *cdim S*›
using *indep-SB SB-span*
by (*metis complex-vector.dim-eq-card*)
finally show *?thesis* **by** *simp*
qed

lemma *cindependent-inter-scaleC-cindependent*:
assumes *a1*: *cindependent* (*B*::′*a*::*complex-vector set*) **and** *a3*: *c* ≠ *1*
shows *B* ∩ (∗*C*) *c* ‘ *B* = {}
proof (*rule classical*, *cases* ‹*c* = *0*›)
case *True*
then show *?thesis*
using *a1* **by** (*auto simp add*: *complex-vector.dependent-zero*)
next
case *False*
assume ¬(*B* ∩ (∗*C*) *c* ‘ *B* = {})
hence *B* ∩ (∗*C*) *c* ‘ *B* ≠ {}
by *blast*
then obtain *x* **where** *u1*: *x* ∈ *B* ∩ (∗*C*) *c* ‘ *B*
by *blast*
then obtain *b* **where** *u2*: *x* = *b* **and** *u3*: *b*∈*B*
by *blast*
then obtain *b*′ **where** *u2*′: *x* = *c* ∗*C* *b*′ **and** *u3*′: *b*′∈*B*
using *u1*
by *blast*
have *g1*: *b* = *c* ∗*C* *b*′
using *u2* **and** *u2*′ **by** *simp*
hence *b* ∈ *complex-vector.span* {*b*′}
using *False*
by (*simp add*: *complex-vector.span-base complex-vector.span-scale*)

hence $b = b'$
by (*metis u3' a1 complex-vector.dependent-def complex-vector.span-base complex-vector.span-scale insertE insert-Diff u2 u2' u3*)
hence $b' = c *_C b'$
using *g1* **by** *blast*
thus *?thesis*
by (*metis a1 a3 complex-vector.dependent-zero complex-vector.scale-right-imp-eq mult-cancel-right2 scaleC-scaleC u3'*)
qed

lemma *real-independent-from-complex-independent*:
assumes *cindependent* (*B*::'*a*::*complex-vector set*)
defines $B' == ((*_C)\ \mathrm{i}\ `\ B)$
shows *independent* ($B \cup B'$)
proof (*rule notI*)
assume ‹*dependent* ($B \cup B'$)›
then obtain *T f0 x* **where** [*simp*]: ‹*finite T*› **and** ‹$T \subseteq B \cup B'$› **and** *f0-sum*: ‹$(\sum v{\in}T.\ f0\ v *_R v) = 0$›
and *x*: ‹$x \in T$› **and** *f0-x*: ‹$f0\ x \neq 0$›
by (*auto simp*: *real-vector.dependent-explicit*)
define *f T1 T2 T' f' x'* **where** ‹$f\ v = (\mathit{if}\ v \in T\ \mathit{then}\ f0\ v\ \mathit{else}\ 0)$›
and ‹$T1 = T \cap B$› **and** ‹$T2 = \mathit{scaleC}\ (-\mathrm{i})\ `\ (T \cap B')$›
and ‹$T' = T1 \cup T2$› **and** ‹$f'\ v = f\ v + \mathrm{i} * f\ (\mathrm{i} *_C v)$›
and ‹$x' = (\mathit{if}\ x \in T1\ \mathit{then}\ x\ \mathit{else}\ -\mathrm{i} *_C x)$› **for** *v*
have ‹$B \cap B' = \{\}$›
by (*simp add*: *assms cindependent-inter-scaleC-cindependent*)
have ‹$T' \subseteq B$›
by (*auto simp*: *T'-def T1-def T2-def B'-def*)
have [*simp*]: ‹*finite T'*› ‹*finite T1*› ‹*finite T2*›
by (*auto simp add*: *T'-def T1-def T2-def*)
have *f-sum*: ‹$(\sum v{\in}T.\ f\ v *_R v) = 0$›
unfolding *f-def* **using** *f0-sum* **by** *auto*
have *f-x*: ‹$f\ x \neq 0$›
using *f0-x x* **by** (*auto simp*: *f-def*)
have *f'-sum*: ‹$(\sum v{\in}T'.\ f'\ v *_C v) = 0$›
proof −
have ‹$(\sum v{\in}T'.\ f'\ v *_C v) = (\sum v{\in}T'.\ \mathit{complex\text{-}of\text{-}real}\ (f\ v) *_C v) + (\sum v{\in}T'.\ (\mathrm{i} * \mathit{complex\text{-}of\text{-}real}\ (f\ (\mathrm{i} *_C v))) *_C v)$›
by (*auto simp*: *f'-def sum.distrib scaleC-add-left*)
also have ‹$(\sum v{\in}T'.\ \mathit{complex\text{-}of\text{-}real}\ (f\ v) *_C v) = (\sum v{\in}T1.\ f\ v *_R v)$› (**is** ‹- = *?left*›)
apply (*auto simp*: *T'-def scaleR-scaleC intro*!: *sum.mono-neutral-cong-right*)
using *T'-def T1-def* ‹$T' \subseteq B$› *f-def* **by** *auto*
also have ‹$(\sum v{\in}T'.\ (\mathrm{i} * \mathit{complex\text{-}of\text{-}real}\ (f\ (\mathrm{i} *_C v))) *_C v) = (\sum v{\in}T2.\ (\mathrm{i} * \mathit{complex\text{-}of\text{-}real}\ (f\ (\mathrm{i} *_C v))) *_C v)$› (**is** ‹- = *?right*›)
apply (*auto simp*: *T'-def intro*!: *sum.mono-neutral-cong-right*)
by (*smt* (*z3*) *B'-def IntE IntI T1-def T2-def* ‹$f \equiv \lambda v.\ \mathit{if}\ v \in T\ \mathit{then}\ f0\ v\ \mathit{else}\ 0$› *add.inverse-inverse complex-vector.vector-space-axioms i-squared imageI mult-minus-left vector-space.vector-space-assms*(*3*) *vector-space.vector-space-assms*(*4*))

```
    also have ‹?right = (∑ v∈T∩B'. f v *R v)› (is ‹- = ?right›)
      apply (rule sum.reindex-cong[symmetric, where l=‹scaleC i›])
        apply (auto simp: T2-def image-image scaleR-scaleC)
      using inj-on-def by fastforce
    also have ‹?left + ?right = (∑ v∈T. f v *R v)›
      apply (subst sum.union-disjoint[symmetric])
      using ‹B ∩ B' = {}› ‹T ⊆ B ∪ B'› apply (auto simp: T1-def)
      by (metis Int-Un-distrib Un-Int-eq(4) sup.absorb-iff1)
    also have ‹... = 0›
      by (rule f-sum)
    finally show ?thesis
      by −
  qed

  have x': ‹x' ∈ T'›
    using ‹T ⊆ B ∪ B'› x by (auto simp: x'-def T'-def T1-def T2-def)

  have f'-x': ‹f' x' ≠ 0›
    using Complex-eq Complex-eq-0 f'-def f-x x'-def by auto

  from ‹finite T'› ‹T' ⊆ B› f'-sum x' f'-x'
  have ‹cdependent B›
    using complex-vector.independent-explicit-module by blast
  with assms show False
    by auto
qed

lemma crepresentation-from-representation:
  assumes a1: cindependent B and a2: b ∈ B and a3: finite B
  shows crepresentation B ψ b = (representation (B ∪ (*C) i ‘ B) ψ b)
                        + i *C (representation (B ∪ (*C) i ‘ B) ψ (i *C b))
proof (cases ψ ∈ cspan B)
  define B' where B' = B ∪ (*C) i ‘ B
  case True
  define r  where r v = real-vector.representation B' ψ v for v
  define r' where r' v = real-vector.representation B' ψ (i *C v) for v
  define f  where f v = r v + i *C r' v for v
  define g  where g v = crepresentation B ψ v for v
  have (∑ v | g v ≠ 0. g v *C v) = ψ
    unfolding g-def
    using Collect-cong Collect-mono-iff DiffD1 DiffD2 True a1
      complex-vector.finite-representation
      complex-vector.sum-nonzero-representation-eq sum.mono-neutral-cong-left
    by fastforce
  moreover have finite {v. g v ≠ 0}
    unfolding g-def
    by (simp add: complex-vector.finite-representation)
  moreover have v ∈ B
    if g v ≠ 0 for v
```

using *that* **unfolding** *g-def*
by (*simp add*: *complex-vector.representation-ne-zero*)
ultimately have *rep1*: $(\sum v{\in}B.\ g\ v *_C v) = \psi$
unfolding *g-def*
using *a3 True a1 complex-vector.sum-representation-eq* **by** *blast*
have *l0'*: *inj* $((*_C)\ \mathrm{i}::'a \Rightarrow 'a)$
unfolding *inj-def*
by *simp*
have *l0*: *inj* $((*_C)\ (-\ \mathrm{i})::'a \Rightarrow 'a)$
unfolding *inj-def*
by *simp*
have *l1*: $(*_C)\ (-\ \mathrm{i})\ `\ B \cap B = \{\}$
using *cindependent-inter-scaleC-cindependent*[**where** $B{=}B$ **and** $c = -\ \mathrm{i}$]
by (*metis Int-commute a1 add.inverse-inverse complex-i-not-one i-squared mult-cancel-left1*
neg-equal-0-iff-equal)
have *l2*: $B \cap (*_C)\ \mathrm{i}\ `\ B = \{\}$
by (*simp add*: *a1 cindependent-inter-scaleC-cindependent*)
have *rr1*: $r\ (\mathrm{i} *_C v) = r'\ v$ **for** v
unfolding *r-def r'-def*
by *simp*
have *k1*: *independent* B'
unfolding B'*-def* **using** *a1 real-independent-from-complex-independent* **by** *simp*
have $\psi \in$ *span* B'
using B'*-def True cspan-as-span* **by** *blast*
have $v \in B'$
if $r\ v \neq 0$
for v
unfolding *r-def*
using *r-def real-vector.representation-ne-zero that* **by** *auto*
have *finite* B'
unfolding B'*-def* **using** *a3*
by *simp*
have $(\sum v{\in}B'.\ r\ v *_R v) = \psi$
unfolding *r-def*
using *True Real-Vector-Spaces.real-vector.sum-representation-eq*[**where** $B = B'$ **and** *basis* $= B'$
and $v = \psi$]
by (*smt Real-Vector-Spaces.dependent-raw-def* ‹$\psi \in$ *Real-Vector-Spaces.span* B'› ‹*finite* B'›
equalityD2 k1)
have *d1*: $(\sum v{\in}B.\ r\ (\mathrm{i} *_C v) *_R (\mathrm{i} *_C v)) = (\sum v{\in}(*_C)\ \mathrm{i}\ `\ B.\ r\ v *_R v)$
using *l0'*
by (*metis* (*mono-tags*, *lifting*) *inj-eq inj-on-def sum.reindex-cong*)
have $(\sum v{\in}B.\ (r\ v + \mathrm{i} * (r'\ v)) *_C v) = (\sum v{\in}B.\ r\ v *_C v + (\mathrm{i} * r'\ v) *_C v)$
by (*meson scaleC-left.add*)
also have $\ldots = (\sum v{\in}B.\ r\ v *_C v) + (\sum v{\in}B.\ (\mathrm{i} * r'\ v) *_C v)$
using *sum.distrib* **by** *fastforce*
also have $\ldots = (\sum v{\in}B.\ r\ v *_C v) + (\sum v{\in}B.\ \mathrm{i} *_C (r'\ v *_C v))$

by *auto*
also have $\ldots = (\sum v{\in}B.\ r\ v *_R v) + (\sum v{\in}B.\ \mathrm{i} *_C (r\ (\mathrm{i} *_C v) *_R v))$
unfolding *r'-def r-def*
by (*metis* (*mono-tags*, *lifting*) *scaleR-scaleC sum.cong*)
also have $\ldots = (\sum v{\in}B.\ r\ v *_R v) + (\sum v{\in}B.\ r\ (\mathrm{i} *_C v) *_R (\mathrm{i} *_C v))$
by (*metis* (*no-types*, *lifting*) *complex-vector.scale-left-commute scaleR-scaleC*)
also have $\ldots = (\sum v{\in}B.\ r\ v *_R v) + (\sum v{\in}(*_C)\ \mathrm{i}\ `\ B.\ r\ v *_R v)$
using *d1*
by *simp*
also have $\ldots = \psi$
using *l2* ‹$(\sum v{\in}B'.\ r\ v *_R v) = \psi$›
unfolding *B'-def*
by (*simp add*: *a3 sum.union-disjoint*)
finally have $(\sum v{\in}B.\ f\ v *_C v) = \psi$ **unfolding** *r'-def r-def f-def* **by** *simp*
hence $0 = (\sum v{\in}B.\ f\ v *_C v) - (\sum v{\in}B.\ \mathit{crepresentation}\ B\ \psi\ v *_C v)$
using *rep1*
unfolding *g-def*
by *simp*
also have $\ldots = (\sum v{\in}B.\ f\ v *_C v - \mathit{crepresentation}\ B\ \psi\ v *_C v)$
by (*simp add*: *sum-subtractf*)
also have $\ldots = (\sum v{\in}B.\ (f\ v - \mathit{crepresentation}\ B\ \psi\ v) *_C v)$
by (*metis scaleC-left.diff*)
finally have $0 = (\sum v{\in}B.\ (f\ v - \mathit{crepresentation}\ B\ \psi\ v) *_C v)$**.**
hence $(\sum v{\in}B.\ (f\ v - \mathit{crepresentation}\ B\ \psi\ v) *_C v) = 0$
by *simp*
hence $f\ b - \mathit{crepresentation}\ B\ \psi\ b = 0$
using *a1 a2 a3 complex-vector.independentD*[**where** $s = B$ **and** $t = B$
and $u = \lambda v.\ f\ v - \mathit{crepresentation}\ B\ \psi\ v$ **and** $v = b$]
order-refl **by** *smt*
hence $\mathit{crepresentation}\ B\ \psi\ b = f\ b$
by *simp*
thus *?thesis* **unfolding** *f-def r-def r'-def B'-def* **by** *auto*
next
define B' **where** $B' = B \cup (*_C)\ \mathrm{i}\ `\ B$
case *False*
have *b2*: $\psi \notin \mathit{real\text{-}vector.span}\ B'$
unfolding *B'-def*
using *False cspan-as-span* **by** *auto*
have $\psi \notin \mathit{complex\text{-}vector.span}\ B$
using *False* **by** *blast*
have $\mathit{crepresentation}\ B\ \psi\ b = 0$
unfolding *complex-vector.representation-def*
by (*simp add*: *False*)
moreover have $\mathit{real\text{-}vector.representation}\ B'\ \psi\ b = 0$
unfolding *real-vector.representation-def*
by (*simp add*: *b2*)
moreover have $\mathit{real\text{-}vector.representation}\ B'\ \psi\ ((*_C)\ \mathrm{i}\ b) = 0$
unfolding *real-vector.representation-def*
by (*simp add*: *b2*)

ultimately show *?thesis* **unfolding** *B′-def* **by** *simp*
qed

lemma *CARD-1-vec-0*[*simp*]: ‹(ψ :: - ::{*complex-vector*,*CARD-1*}) = *0*›
by *auto*

lemma *scaleC-cindependent*:
assumes *a1*: *cindependent* (*B*::′*a*::*complex-vector set*) **and** *a3*: *c* ≠ *0*
shows *cindependent* ((∗$_C$) *c* ‘ *B*)
proof−
have *u y* = *0*
if *g1*: *y*∈*S* **and** *g2*: ($\sum$ *x*∈*S*. *u x* ∗$_C$ *x*) = *0* **and** *g3*: *finite S* **and** *g4*: *S*⊆(∗$_C$) *c* ‘ *B*
for *u y S*
proof−
define *v* **where** *v x* = *u* (*c* ∗$_C$ *x*) **for** *x*
obtain *S′* **where** *S′*⊆*B* **and** *S-S′*: *S* = (∗$_C$) *c* ‘ *S′*
by (*meson g4 subset-imageE*)
have *inj* ((∗$_C$) *c*::′*a*⇒-)
unfolding *inj-def*
using *a3* **by** *auto*
hence *finite S′*
using *S-S′ finite-imageD g3 inj-on-subset* **by** *blast*
have *t* ∈ (∗$_C$) (*inverse c*) ‘ *S*
if *t* ∈ *S′* **for** *t*
proof−
have *c* ∗$_C$ *t* ∈ *S*
using ‹*S* = (∗$_C$) *c* ‘ *S′*› *that* **by** *blast*
hence (*inverse c*) ∗$_C$ (*c* ∗$_C$ *t*) ∈ (∗$_C$) (*inverse c*) ‘ *S*
by *blast*
moreover have (*inverse c*) ∗$_C$ (*c* ∗$_C$ *t*) = *t*
by (*simp add*: *a3*)
ultimately show *?thesis* **by** *simp*
qed
moreover have *t* ∈ *S′*
if *t* ∈ (∗$_C$) (*inverse c*) ‘ *S* **for** *t*
proof−
obtain *t′* **where** *t* = (*inverse c*) ∗$_C$ *t′* **and** *t′* ∈ *S*
using ‹*t* ∈ (∗$_C$) (*inverse c*) ‘ *S*› **by** *auto*
have *c* ∗$_C$ *t* = *c* ∗$_C$ ((*inverse c*) ∗$_C$ *t′*)
using ‹*t* = (*inverse c*) ∗$_C$ *t′*› **by** *simp*
also have ... = (*c* ∗ (*inverse c*)) ∗$_C$ *t′*
by *simp*
also have ... = *t′*
by (*simp add*: *a3*)
finally have *c* ∗$_C$ *t* = *t′*.
thus *?thesis* **using** ‹*t′* ∈ *S*›

```
      using ‹S = (*C) c ‘ S'› a3 complex-vector.scale-left-imp-eq by blast
    qed
    ultimately have S' = (*C) (inverse c) ‘ S
      by blast
    hence inverse c *C y ∈ S'
      using that(1) by blast
    have t: inj (((*C) c)::'a ⇒ -)
      using a3 complex-vector.injective-scale[where c = c]
      by blast
    have 0 = (∑ x∈(*C) c ‘ S'. u x *C x)
      using ‹S = (*C) c ‘ S'› that(2) by auto
    also have ... = (∑ x∈S'. v x *C (c *C x))
      unfolding v-def
     using t Groups-Big.comm-monoid-add-class.sum.reindex[where h = ((*C) c)
and A = S'
        and g = λx. u x *C x] inj-on-subset by auto
    also have ... = c *C (∑ x∈S'. v x *C x)
    by (metis (mono-tags, lifting) complex-vector.scale-left-commute scaleC-right.sum
sum.cong)
    finally have 0 = c *C (∑ x∈S'. v x *C x).
    hence (∑ x∈S'. v x *C x) = 0
      using a3 by auto
    hence v (inverse c *C y) = 0
      using ‹inverse c *C y ∈ S'› ‹finite S'› ‹S' ⊆ B› a1
        complex-vector.independentD
      by blast
    thus u y = 0
      unfolding v-def
      by (simp add: a3)
  qed
  thus ?thesis
    using complex-vector.dependent-explicit
    by (simp add: complex-vector.dependent-explicit )
qed

lemma cspan-eqI:
  assumes ‹⋀a. a∈A ⟹ a∈cspan B›
  assumes ‹⋀b. b∈B ⟹ b∈cspan A›
  shows ‹cspan A = cspan B›
  apply (rule complex-vector.span-subspace[rotated])
    apply (rule complex-vector.span-minimal)
  using assms by auto

lemma (in bounded-cbilinear) bounded-bilinear[simp]: bounded-bilinear prod
  by standard

lemma norm-scaleC-sgn[simp]: ‹complex-of-real (norm ψ) *C sgn ψ = ψ› for ψ
:: 'a::complex-normed-vector
  apply (cases ‹ψ = 0›)
```

by (*auto simp*: *sgn-div-norm scaleR-scaleC*)

lemma *scaleC-of-complex*[*simp*]: ‹*scaleC x* (*of-complex y*) = *of-complex* (*x* ∗ *y*)›
unfolding *of-complex-def* **using** *scaleC-scaleC* **by** *blast*

lemma *bounded-clinear-inv*:
assumes [*simp*]: ‹*bounded-clinear f*›
assumes *b*: ‹*b* > *0*›
assumes *bound*: ‹⋀*x*. *norm* (*f x*) ≥ *b* ∗ *norm x*›
assumes ‹*surj f*›
shows ‹*bounded-clinear* (*inv f*)›
proof (*rule bounded-clinear-intro*)
fix *x y* :: *'b* **and** *r* :: *complex*
define *x' y'* **where** ‹*x'* = *inv f x*› **and** ‹*y'* = *inv f y*›
have [*simp*]: ‹*clinear f*›
by (*simp add*: *bounded-clinear.clinear*)
have [*simp*]: ‹*inj f*›
proof (*rule injI*)
fix *x y* **assume** ‹*f x* = *f y*›
then have ‹*norm* (*f* (*x* − *y*)) = *0*›
by (*simp add*: *complex-vector.linear-diff*)
with *bound b* **have** ‹*norm* (*x* − *y*) = *0*›
by (*metis linorder-not-le mult-le-0-iff nle-le norm-ge-zero*)
then show ‹*x* = *y*›
by *simp*
qed

from ‹*surj f*›
have [*simp*]: ‹*x* = *f x'*› ‹*y* = *f y'*›
by (*simp-all add*: *surj-f-inv-f x'-def y'-def*)
show *inv f* (*x* + *y*) = *inv f x* + *inv f y*
by (*simp flip*: *complex-vector.linear-add*)
show *inv f* (*r* ∗*C x*) = *r* ∗*C inv f x*
by (*simp flip*: *clinear.scaleC*)
from *bound* **have** *b* ∗ *norm* (*inv f x*) ≤ *norm x*
by (*simp flip*: *clinear.scaleC*)
with *b* **show** *norm* (*inv f x*) ≤ *norm x* ∗ *inverse b*
by (*smt* (*verit*, *ccfv-threshold*) *left-inverse mult.commute mult-cancel-right1 mult-le-cancel-left-pos vector-space-over-itself.scale-scale*)
qed

lemma *range-is-csubspace*[*simp*]:
assumes *a1*: *clinear f*
shows *csubspace* (*range f*)
using *assms complex-vector.linear-subspace-image complex-vector.subspace-UNIV*
by *blast*

lemma *csubspace-is-convex*[*simp*]:
assumes *a1*: *csubspace M*

shows *convex M*
proof−
have ‹∀ *x*∈*M*. ∀ *y*∈ *M*. ∀ *u*. ∀ *v*. *u* ∗*C* *x* + *v* ∗*C* *y* ∈ *M*›
using *a1*
by (*simp add*: *complex-vector.subspace-def*)
hence ‹∀ *x*∈*M*. ∀ *y*∈*M*. ∀ *u*::*real*. ∀ *v*::*real*. *u* ∗*R* *x* + *v* ∗*R* *y* ∈ *M*›
by (*simp add*: *scaleR-scaleC*)
hence ‹∀ *x*∈*M*. ∀ *y*∈*M*. ∀ *u*≥*0*. ∀ *v*≥*0*. *u* + *v* = *1* ⟶ *u* ∗*R* *x* + *v* ∗*R* *y* ∈*M*›
by *blast*
thus *?thesis* **using** *convex-def* **by** *blast*
qed

lemma *kernel-is-csubspace*[*simp*]:
assumes *a1*: *clinear f*
shows *csubspace* (*f* −‘ {*0*})
by (*simp add*: *assms complex-vector.linear-subspace-vimage*)

lemma *bounded-cbilinear-0*[*simp*]: ‹*bounded-cbilinear* (λ- -. *0*)›
by (*auto intro*!: *bounded-cbilinear.intro exI*[**where** *x*=*0*])
lemma *bounded-cbilinear-0'*[*simp*]: ‹*bounded-cbilinear 0*›
by (*auto intro*!: *bounded-cbilinear.intro exI*[**where** *x*=*0*])

lemma *bounded-cbilinear-apply-bounded-clinear*: ‹*bounded-clinear* (*f x*)› **if** ‹*bounded-cbilinear f*›
proof −
interpret *f*: *bounded-cbilinear f*
by (*fact that*)
from *f.bounded* **obtain** *K* **where** ‹*norm* (*f a b*) ≤ *norm a* ∗ *norm b* ∗ *K*› **for** *a b*
by *auto*
then show *?thesis*
by (*auto intro*!: *bounded-clinearI*[**where** *K*=‹*norm x* ∗ *K*›]
simp add: *f.add-right f.scaleC-right mult.commute mult.left-commute*)
qed

lemma *clinear-scaleR*[*simp*]: ‹*clinear* (*scaleR x*)›
by (*simp add*: *complex-vector.linear-scale-self scaleR-scaleC*)

lemma *abs-summable-on-scaleC-left* [*intro*]:
fixes *c* :: ‹'*a* :: *complex-normed-vector*›
assumes *c* ≠ *0* ⟹ *f abs-summable-on A*
shows (λ*x*. *f x* ∗*C* *c*) *abs-summable-on A*
apply (*cases* ‹*c* = *0*›)
apply *simp*
by (*auto intro*!: *summable-on-cmult-left assms simp*: *norm-scaleC*)

lemma *abs-summable-on-scaleC-right* [*intro*]:
fixes *f* :: ‹'*a* ⇒ '*b* :: *complex-normed-vector*›

assumes $c \neq 0 \Longrightarrow f$ *abs-summable-on* A
shows $(\lambda x.\ c *_C f\ x)$ *abs-summable-on* A
apply (*cases* ‹$c = 0$›)
apply *simp*
by (*auto intro*!: *summable-on-cmult-right assms simp*: *norm-scaleC*)

lemma *clinear-of-complex*[*iff*]: ‹*clinear of-complex*›
by (*simp add*: *clinearI*)

lemma *infsum-of-bool-scaleC*: ‹$(\sum_\infty x{\in}X.$ *of-bool* $(x{=}y) *_C f\ x) =$ *of-bool* $(y{\in}X) *_C f\ y$› **for** f :: ‹- ⇒ -::*complex-vector*›
apply (*cases* ‹$y{\in}X$›)
apply (*subst infsum-cong-neutral*[**where** $T{=}\{y\}$› **and** $g{=}f$])
apply *auto*[4]
apply (*subst infsum-cong-neutral*[**where** $T{=}\{\}$› **and** $g{=}f$])
by *auto*

lemma *has-sum-bounded-clinear*:
assumes *bounded-clinear* h **and** (f *has-sum* S) A
shows (($\lambda x.\ h\ (f\ x)$) *has-sum* $h\ S$) A
apply (*rule has-sum-bounded-linear*[**where** $h{=}h$])
by (*auto intro*!: *bounded-clinear.bounded-linear assms*)

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 *csubspace-has-basis*:
assumes ‹*csubspace* S›
shows ‹$\exists B.$ *cindependent* $B \wedge$ *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 ‹$c \neq 0$›
shows ‹*inj-on* (*scaleC* c) A›
by (*meson assms inj-onI scaleC-left-imp-eq*)

7.2 Antilinear maps and friends

locale *antilinear* = *additive* f **for** f :: 'a::*complex-vector* ⇒ 'b::*complex-vector* +
assumes *scaleC*: $f\ (scaleC\ r\ x) = cnj\ r *_C f\ x$

sublocale *antilinear* ⊆ *linear*

proof (*rule linearI*)
  show *f* (*b1* + *b2*) = *f b1* + *f b2*
    for *b1* :: *'a*
      and *b2* :: *'a*
    by (*simp add*: *add*)
  show *f* (*r* $*_R$ *b*) = *r* $*_R$ *f b*
    for *r* :: *real*
      and *b* :: *'a*
    unfolding *scaleR-scaleC* **by** (*subst scaleC*, *simp*)
qed

lemma *antilinear-imp-scaleC*:
  fixes *D* :: *complex* ⇒ *'a*::*complex-vector*
  assumes *antilinear D*
  obtains *d* **where** *D* = (λ*x*. *cnj x* $*_C$ *d*)
proof −
  interpret *clinear D o cnj*
    apply *standard* **apply** *auto*
     apply (*simp add*: *additive.add assms antilinear.axioms*(*1*))
    using *assms antilinear.scaleC* **by** *fastforce*
  obtain *d* **where** *D o cnj* = (λ*x*. *x* $*_C$ *d*)
    using *clinear-axioms complex-vector.linear-imp-scale* **by** *blast*
  then have ‹*D* = (λ*x*. *cnj x* $*_C$ *d*)›
    by (*metis comp-apply complex-cnj-cnj*)
  then show *?thesis*
    by (*rule that*)
qed

corollary *complex-antilinearD*:
  fixes *f* :: *complex* ⇒ *complex*
  assumes *antilinear f* **obtains** *c* **where** *f* = (λ*x*. *c* ∗ *cnj x*)
  by (*rule antilinear-imp-scaleC* [*OF assms*]) (*force simp*: *scaleC-conv-of-complex*)

lemma *antilinearI*:
  assumes ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y*
    and ⋀*c x*. *f* (*c* $*_C$ *x*) = *cnj c* $*_C$ *f x*
  shows *antilinear f*
  by *standard* (*rule assms*)+

lemma *antilinear-o-antilinear*: *antilinear f* ⟹ *antilinear g* ⟹ *clinear* (*g o f*)
  apply (*rule clinearI*)
   apply (*simp add*: *additive.add antilinear-def*)
  by (*simp add*: *antilinear.scaleC*)

lemma *clinear-o-antilinear*: *antilinear f* ⟹ *clinear g* ⟹ *antilinear* (*g o f*)
  apply (*rule antilinearI*)
   apply (*simp add*: *additive.add complex-vector.linear-add antilinear-def*)
  by (*simp add*: *complex-vector.linear-scale antilinear.scaleC*)

lemma *antilinear-o-clinear*: *clinear f* $\Longrightarrow$ *antilinear g* $\Longrightarrow$ *antilinear* (*g o f*)
apply (*rule antilinearI*)
apply (*simp add*: *additive.add complex-vector.linear-add antilinear-def*)
by (*simp add*: *complex-vector.linear-scale antilinear.scaleC*)

locale *bounded-antilinear* = *antilinear f* **for** *f* :: *'a*::*complex-normed-vector* $\Rightarrow$ *'b*::*complex-normed-vector* +
assumes *bounded*: $\exists K.\ \forall x.\ norm\ (f\ x) \leq norm\ x * K$

lemma *bounded-antilinearI*:
assumes ‹$\bigwedge b1\ b2.\ f\ (b1 + b2) = f\ b1 + f\ b2$›
assumes ‹$\bigwedge r\ b.\ f\ (r *_C b) = cnj\ r *_C f\ b$›
assumes ‹$\forall x.\ norm\ (f\ x) \leq norm\ x * K$›
shows *bounded-antilinear f*
using *assms* **by** (*auto intro*!: *exI bounded-antilinear.intro antilinearI simp*: *bounded-antilinear-axioms-def*)

sublocale *bounded-antilinear* $\subseteq$ *real*: *bounded-linear*
— Gives access to all lemmas from *Real-Vector-Spaces.linear* using prefix *real.*
apply *standard* **by** (*fact bounded*)

lemma (**in** *bounded-antilinear*) *bounded-linear*: *bounded-linear f*
by (*fact real.bounded-linear*)

lemma (**in** *bounded-antilinear*) *antilinear*: *antilinear f*
by (*fact antilinear-axioms*)

lemma *bounded-antilinear-intro*:
assumes $\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y$
and $\bigwedge r\ x.\ f\ (scaleC\ r\ x) = scaleC\ (cnj\ r)\ (f\ x)$
and $\bigwedge x.\ norm\ (f\ x) \leq norm\ x * K$
shows *bounded-antilinear f*
by *standard* (*blast intro*: *assms*)+

lemma *bounded-antilinear-0*[*simp*]: ‹*bounded-antilinear* (λ-. *0*)›
by (*auto intro*!: *bounded-antilinearI*[**where** *K=0*])

lemma *bounded-antilinear-0'*[*simp*]: ‹*bounded-antilinear 0*›
by (*auto intro*!: *bounded-antilinearI*[**where** *K=0*])

lemma *cnj-bounded-antilinear*[*simp*]: *bounded-antilinear cnj*
apply (*rule bounded-antilinear-intro* [**where** *K = 1*])
by *auto*

lemma *bounded-antilinear-o-bounded-antilinear*:
assumes *bounded-antilinear f*
and *bounded-antilinear g*
shows *bounded-clinear* ($\lambda x.\ f\ (g\ x)$)
proof
interpret *f*: *bounded-antilinear f* **by** *fact*

interpret *g*: *bounded-antilinear g* **by** *fact*
fix *b1 b2 b r*
show *f* (*g* (*b1* + *b2*)) = *f* (*g b1*) + *f* (*g b2*)
by (*simp add*: *f.add g.add*)
show *f* (*g* (*r* *$_C$* *b*)) = *r* *$_C$* *f* (*g b*)
by (*simp add*: *f.scaleC g.scaleC*)
have *bounded-linear* (λ*x*. *f* (*g x*))
using *f.bounded-linear g.bounded-linear* **by** (*rule bounded-linear-compose*)
then show ∃ *K*. ∀ *x*. *norm* (*f* (*g x*)) ≤ *norm x* ∗ *K*
by (*rule bounded-linear.bounded*)
qed

lemma *bounded-antilinear-o-bounded-antilinear′*:
assumes *bounded-antilinear f*
and *bounded-antilinear g*
shows *bounded-clinear* (*g o f*)
using *assms* **by** (*simp add*: *o-def bounded-antilinear-o-bounded-antilinear*)

lemma *bounded-antilinear-o-bounded-clinear*:
assumes *bounded-antilinear f*
and *bounded-clinear g*
shows *bounded-antilinear* (λ*x*. *f* (*g x*))
proof
interpret *f*: *bounded-antilinear f* **by** *fact*
interpret *g*: *bounded-clinear g* **by** *fact*
show *f* (*g* (*x* + *y*)) = *f* (*g x*) + *f* (*g y*) **for** *x y*
by (*simp only*: *f.add g.add*)
show *f* (*g* (*scaleC r x*)) = *scaleC* (*cnj r*) (*f* (*g x*)) **for** *r x*
by (*simp add*: *f.scaleC g.scaleC*)
have *bounded-linear* (λ*x*. *f* (*g x*))
using *f.bounded-linear g.bounded-linear* **by** (*rule bounded-linear-compose*)
then show ∃ *K*. ∀ *x*. *norm* (*f* (*g x*)) ≤ *norm x* ∗ *K*
by (*rule bounded-linear.bounded*)
qed

lemma *bounded-antilinear-o-bounded-clinear′*:
assumes *bounded-clinear f*
and *bounded-antilinear g*
shows *bounded-antilinear* (*g o f*)
using *assms* **by** (*simp add*: *o-def bounded-antilinear-o-bounded-clinear*)

lemma *bounded-clinear-o-bounded-antilinear*:
assumes *bounded-clinear f*
and *bounded-antilinear g*
shows *bounded-antilinear* (λ*x*. *f* (*g x*))
proof
interpret *f*: *bounded-clinear f* **by** *fact*
interpret *g*: *bounded-antilinear g* **by** *fact*
show *f* (*g* (*x* + *y*)) = *f* (*g x*) + *f* (*g y*) **for** *x y*

by (*simp only*: *f.add g.add*)
show *f* (*g* (*scaleC r x*)) = *scaleC* (*cnj r*) (*f* (*g x*)) **for** *r x*
using *f.scaleC g.scaleC* **by** *fastforce*
have *bounded-linear* (λ*x*. *f* (*g x*))
using *f.bounded-linear g.bounded-linear* **by** (*rule bounded-linear-compose*)
then show ∃ *K*. ∀ *x*. *norm* (*f* (*g x*)) ≤ *norm x* ∗ *K*
by (*rule bounded-linear.bounded*)
qed

lemma *bounded-clinear-o-bounded-antilinear'*:
assumes *bounded-antilinear f*
and *bounded-clinear g*
shows *bounded-antilinear* (*g o f*)
using *assms* **by** (*simp add*: *o-def bounded-clinear-o-bounded-antilinear*)

lemma *bij-clinear-imp-inv-clinear*: *clinear* (*inv f*)
if *a1*: *clinear f* **and** *a2*: *bij f*
proof
fix *b1 b2 r b*
show *inv f* (*b1* + *b2*) = *inv f b1* + *inv f b2*
by (*simp add*: *a1 a2 bij-is-inj bij-is-surj complex-vector.linear-add inv-f-eq surj-f-inv-f*)
show *inv f* (*r* ∗*C b*) = *r* ∗*C inv f b*
using *that*
by (*smt bij-inv-eq-iff clinear-def complex-vector.linear-scale*)
qed

locale *bounded-sesquilinear* =
fixes
prod :: *'a*::*complex-normed-vector* ⇒ *'b*::*complex-normed-vector* ⇒ *'c*::*complex-normed-vector*
(**infixl** ‹∗∗› *70*)
assumes *add-left*: *prod* (*a* + *a'*) *b* = *prod a b* + *prod a' b*
and *add-right*: *prod a* (*b* + *b'*) = *prod a b* + *prod a b'*
and *scaleC-left*: *prod* (*r* ∗*C a*) *b* = (*cnj r*) ∗*C* (*prod a b*)
and *scaleC-right*: *prod a* (*r* ∗*C b*) = *r* ∗*C* (*prod a b*)
and *bounded*: ∃ *K*. ∀ *a b*. *norm* (*prod a b*) ≤ *norm a* ∗ *norm b* ∗ *K*

sublocale *bounded-sesquilinear* ⊆ *real*: *bounded-bilinear*
— Gives access to all lemmas from *Real-Vector-Spaces.linear* using prefix *real.*
apply *standard*
by (*auto simp*: *add-left add-right scaleC-left scaleC-right bounded scaleR-scaleC*)

lemma (**in** *bounded-sesquilinear*) *bounded-bilinear*[*simp*]: *bounded-bilinear prod*
by *intro-locales*

lemma (**in** *bounded-sesquilinear*) *bounded-antilinear-left*: *bounded-antilinear* (λ*a*. *prod a b*)
apply *standard*

apply (*auto simp add*: *scaleC-left add-left*)
by (*metis ab-semigroup-mult-class.mult-ac*(*1*) *bounded*)

lemma (**in** *bounded-sesquilinear*) *bounded-clinear-right*: *bounded-clinear* (λ*b*. *prod a b*)
apply *standard*
apply (*auto simp add*: *scaleC-right add-right*)
by (*metis bounded mult.assoc mult.left-commute*)

lemma (**in** *bounded-sesquilinear*) *comp1*:
assumes ‹*bounded-clinear g*›
shows ‹*bounded-sesquilinear* (λ*x*. *prod* (*g x*))›
proof
interpret *bounded-clinear g* **by** *fact*
fix *a a' b b' r*
show *prod* (*g* (*a* + *a'*)) *b* = *prod* (*g a*) *b* + *prod* (*g a'*) *b*
by (*simp add*: *add add-left*)
show *prod* (*g a*) (*b* + *b'*) = *prod* (*g a*) *b* + *prod* (*g a*) *b'*
by (*simp add*: *add add-right*)
show *prod* (*g* (*r* *$_C$ *a*)) *b* = *cnj r* *$_C$ *prod* (*g a*) *b*
by (*simp add*: *scaleC scaleC-left*)
show *prod* (*g a*) (*r* *$_C$ *b*) = *r* *$_C$ *prod* (*g a*) *b*
by (*simp add*: *scaleC-right*)
interpret *bi*: *bounded-bilinear* ‹(λ*x*. *prod* (*g x*))›
by (*simp add*: *bounded-linear real.comp1*)
show ∃ *K*. ∀ *a b*. *norm* (*prod* (*g a*) *b*) ≤ *norm a* * *norm b* * *K*
using *bi.bounded* **by** *blast*
qed

lemma (**in** *bounded-sesquilinear*) *comp2*:
assumes ‹*bounded-clinear g*›
shows ‹*bounded-sesquilinear* (λ*x y*. *prod x* (*g y*))›
proof
interpret *bounded-clinear g* **by** *fact*
fix *a a' b b' r*
show *prod* (*a* + *a'*) (*g b*) = *prod a* (*g b*) + *prod a'* (*g b*)
by (*simp add*: *add add-left*)
show *prod a* (*g* (*b* + *b'*)) = *prod a* (*g b*) + *prod a* (*g b'*)
by (*simp add*: *add add-right*)
show *prod* (*r* *$_C$ *a*) (*g b*) = *cnj r* *$_C$ *prod a* (*g b*)
by (*simp add*: *scaleC scaleC-left*)
show *prod a* (*g* (*r* *$_C$ *b*)) = *r* *$_C$ *prod a* (*g b*)
by (*simp add*: *scaleC scaleC-right*)
interpret *bi*: *bounded-bilinear* ‹(λ*x y*. *prod x* (*g y*))›
apply (*rule bounded-bilinear.flip*)
using - *bounded-linear* **apply** (*rule bounded-bilinear.comp1*)
using *bounded-bilinear* **by** (*rule bounded-bilinear.flip*)
show ∃ *K*. ∀ *a b*. *norm* (*prod a* (*g b*)) ≤ *norm a* * *norm b* * *K*
using *bi.bounded* **by** *blast*

qed

lemma (**in** *bounded-sesquilinear*) *comp*: *bounded-clinear f* $\Longrightarrow$ *bounded-clinear g*
$\Longrightarrow$ *bounded-sesquilinear* ($\lambda x\ y.$ *prod* (*f x*) (*g y*))
using *comp1 bounded-sesquilinear.comp2* **by** *auto*

lemma *bounded-clinear-const-scaleR*:
fixes *c* :: *real*
assumes ‹*bounded-clinear f*›
shows ‹*bounded-clinear* (λ *x*. *c* $*_R$ *f x*)›
proof−
have ‹*bounded-clinear* (λ *x*. (*complex-of-real c*) $*_C$ *f x*)›
by (*simp add*: *assms bounded-clinear-const-scaleC*)
thus *?thesis*
by (*simp add*: *scaleR-scaleC*)
qed

lemma *bounded-linear-bounded-clinear*:
‹*bounded-linear A* $\Longrightarrow$ $\forall$ *c x*. *A* (*c* $*_C$ *x*) = *c* $*_C$ *A x* $\Longrightarrow$ *bounded-clinear A*›
apply *standard*
by (*simp-all add*: *linear-simps bounded-linear.bounded*)

lemma *comp-bounded-clinear*:
fixes *A* :: ‹'*b*::*complex-normed-vector* $\Rightarrow$ '*c*::*complex-normed-vector*›
and *B* :: ‹'*a*::*complex-normed-vector* $\Rightarrow$ '*b*›
assumes ‹*bounded-clinear A*› **and** ‹*bounded-clinear B*›
shows ‹*bounded-clinear* (*A* $\circ$ *B*)›
by (*metis clinear-compose assms*(*1*) *assms*(*2*) *bounded-clinear-axioms-def bounded-clinear-compose bounded-clinear-def o-def*)

lemma *bounded-sesquilinear-add*:
‹*bounded-sesquilinear* (λ *x y*. *A x y* + *B x y*)› **if** ‹*bounded-sesquilinear A*› **and** ‹*bounded-sesquilinear B*›
proof
fix *a a'* :: '*a* **and** *b b'* :: '*b* **and** *r* :: *complex*
show *A* (*a* + *a'*) *b* + *B* (*a* + *a'*) *b* = (*A a b* + *B a b*) + (*A a' b* + *B a' b*)
by (*simp add*: *bounded-sesquilinear.add-left that*(*1*) *that*(*2*))
show ‹*A a* (*b* + *b'*) + *B a* (*b* + *b'*) = (*A a b* + *B a b*) + (*A a b'* + *B a b'*)›
by (*simp add*: *bounded-sesquilinear.add-right that*(*1*) *that*(*2*))
show ‹*A* (*r* $*_C$ *a*) *b* + *B* (*r* $*_C$ *a*) *b* = *cnj r* $*_C$ (*A a b* + *B a b*)›
by (*simp add*: *bounded-sesquilinear.scaleC-left scaleC-add-right that*(*1*) *that*(*2*))
show ‹*A a* (*r* $*_C$ *b*) + *B a* (*r* $*_C$ *b*) = *r* $*_C$ (*A a b* + *B a b*)›
by (*simp add*: *bounded-sesquilinear.scaleC-right scaleC-add-right that*(*1*) *that*(*2*))
show ‹$\exists$ *K*. $\forall$ *a b*. *norm* (*A a b* + *B a b*) $\leq$ *norm a* $*$ *norm b* $*$ *K*›
proof−
have ‹$\exists$ *KA*. $\forall$ *a b*. *norm* (*A a b*) $\leq$ *norm a* $*$ *norm b* $*$ *KA*›
by (*simp add*: *bounded-sesquilinear.bounded that*(*1*))
then obtain *KA* **where** ‹$\forall$ *a b*. *norm* (*A a b*) $\leq$ *norm a* $*$ *norm b* $*$ *KA*›

```
    by blast
  have ‹∃ KB. ∀ a b. norm (B a b) ≤ norm a * norm b * KB›
    by (simp add: bounded-sesquilinear.bounded that(2))
  then obtain KB where ‹∀ a b. norm (B a b) ≤ norm a * norm b * KB›
    by blast
  have ‹norm (A a b + B a b) ≤ norm a * norm b * (KA + KB)›
    for a b
  proof−
    have ‹norm (A a b + B a b) ≤ norm (A a b) +  norm (B a b)›
      using norm-triangle-ineq by blast
    also have ‹... ≤ norm a * norm b * KA + norm a * norm b * KB›
      using  ‹∀ a b. norm (A a b) ≤ norm a * norm b * KA›
        ‹∀ a b. norm (B a b) ≤ norm a * norm b * KB›
      using add-mono by blast
    also have ‹... =  norm a * norm b * (KA + KB)›
      by (simp add: mult.commute ring-class.ring-distribs(2))
    finally show ?thesis
      by blast
  qed
  thus ?thesis by blast
 qed
qed

lemma bounded-sesquilinear-uminus:
 ‹bounded-sesquilinear (λ x y. − A x y)› if ‹bounded-sesquilinear A›
proof
 fix a a′ :: 'a and b b′ :: 'b and r :: complex
 show − A (a + a′) b = (− A a b) + (− A a′ b)
   by (simp add: bounded-sesquilinear.add-left that)
 show ‹− A a (b + b′) = (− A a b) + (− A a b′)›
   by (simp add: bounded-sesquilinear.add-right that)
 show ‹− A (r *C a) b = cnj r *C (− A a b)›
   by (simp add: bounded-sesquilinear.scaleC-left that)
 show ‹− A a (r *C b) = r *C (− A a b)›
   by (simp add: bounded-sesquilinear.scaleC-right that)
 show ‹∃K. ∀a b. norm (− A a b) ≤ norm a * norm b * K›
 proof−
   have ‹∃ KA. ∀ a b. norm (A a b) ≤ norm a * norm b * KA›
     by (simp add: bounded-sesquilinear.bounded that(1))
   then obtain KA where ‹∀ a b. norm (A a b) ≤ norm a * norm b * KA›
     by blast
   have ‹norm (− A a b) ≤ norm a * norm b * KA›
     for a b
     by (simp add: ‹∀a b. norm (A a b) ≤ norm a * norm b * KA›)
   thus ?thesis by blast
 qed
qed

lemma bounded-sesquilinear-diff:
```

‹*bounded-sesquilinear* (λ *x y*. *A x y* − *B x y*)› **if** ‹*bounded-sesquilinear A*› **and** ‹*bounded-sesquilinear B*›
proof −
have ‹*bounded-sesquilinear* (λ *x y*. − *B x y*)›
using *that*(*2*) **by** (*rule bounded-sesquilinear-uminus*)
then have ‹*bounded-sesquilinear* (λ *x y*. *A x y* + (− *B x y*))›
using *that*(*1*) **by** (*rule bounded-sesquilinear-add*[*rotated*])
then show *?thesis*
by *auto*
qed

lemmas *isCont-scaleC* [*simp*] =
bounded-bilinear.isCont [*OF bounded-cbilinear-scaleC*[*THEN bounded-cbilinear.bounded-bilinear*]]

lemma *bounded-sesquilinear-0*[*simp*]: ‹*bounded-sesquilinear* (λ- -.*0*)›
by (*auto intro*!: *bounded-sesquilinear.intro exI*[**where** *x*=*0*])

lemma *bounded-sesquilinear-0'*[*simp*]: ‹*bounded-sesquilinear 0*›
by (*auto intro*!: *bounded-sesquilinear.intro exI*[**where** *x*=*0*])

lemma *bounded-sesquilinear-apply-bounded-clinear*: ‹*bounded-clinear* (*f x*)› **if** ‹*bounded-sesquilinear f*›
proof −
interpret *f*: *bounded-sesquilinear f*
by (*fact that*)
from *f.bounded* **obtain** *K* **where** ‹*norm* (*f a b*) ≤ *norm a* ∗ *norm b* ∗ *K*› **for** *a b*
by *auto*
then show *?thesis*
by (*auto intro*!: *bounded-clinearI*[**where** *K*=‹*norm x* ∗ *K*›]
simp add: *f.add-right f.scaleC-right mult.commute mult.left-commute*)
qed

lemma *antilinear-diff*:
assumes ‹*antilinear f*› **and** ‹*antilinear g*›
shows ‹*antilinear* (λ*x*. *f x* − *g x*)›
apply (*rule antilinearI*)
apply (*metis add-diff-add additive.add antilinear-def assms*(*1*,*2*))
by (*simp add*: *antilinear.scaleC assms*(*1*,*2*) *scaleC-right.diff*)

7.3 Misc 2

lemma *summable-on-scaleC-left* [*intro*]:
fixes *c* :: ‹'*a* :: *complex-normed-vector*›
assumes *c* ≠ *0* ⟹ *f summable-on A*
shows (λ*x*. *f x* ∗$_C$ *c*) *summable-on A*
apply (*cases* ‹*c* ≠ *0*›)
apply (*subst asm-rl*[*of* ‹(λ*x*. *f x* ∗$_C$ *c*) = (λ*y*. *y* ∗$_C$ *c*) *o f*›], *simp add*: *o-def*)
apply (*rule summable-on-comm-additive*)

using *assms* **by** (*auto simp add*: *scaleC-left.additive-axioms*)

lemma *summable-on-scaleC-right* [*intro*]:
fixes $f :: \langle 'a \Rightarrow 'b :: complex\text{-}normed\text{-}vector \rangle$
assumes $c \neq 0 \Longrightarrow f$ *summable-on* A
shows $(\lambda x.\ c *_C f\ x)$ *summable-on* A
apply (*cases* $\langle c \neq 0 \rangle$)
apply (*subst asm-rl*[*of* $\langle (\lambda x.\ c *_C f\ x) = (\lambda y.\ c *_C y)\ o\ f \rangle$], *simp add*: *o-def*)
apply (*rule summable-on-comm-additive*)
using *assms* **by** (*auto simp add*: *scaleC-right.additive-axioms*)

lemma *infsum-scaleC-left*:
fixes $c :: \langle 'a :: complex\text{-}normed\text{-}vector \rangle$
assumes $c \neq 0 \Longrightarrow f$ *summable-on* A
shows *infsum* $(\lambda x.\ f\ x *_C c)\ A =$ *infsum* $f\ A *_C c$
apply (*cases* $\langle c \neq 0 \rangle$)
apply (*subst asm-rl*[*of* $\langle (\lambda x.\ f\ x *_C c) = (\lambda y.\ y *_C c)\ o\ f \rangle$], *simp add*: *o-def*)
apply (*rule infsum-comm-additive*)
using *assms* **by** (*auto simp add*: *scaleC-left.additive-axioms*)

lemma *infsum-scaleC-right*:
fixes $f :: \langle 'a \Rightarrow 'b :: complex\text{-}normed\text{-}vector \rangle$
shows *infsum* $(\lambda x.\ c *_C f\ x)\ A = c *_C$ *infsum* $f\ A$
proof −
consider (*summable*) $\langle f$ *summable-on* $A \rangle$ | (*c0*) $\langle c = 0 \rangle$ | (*not-summable*) $\langle \neg f$ *summable-on* $A \rangle\ \langle c \neq 0 \rangle$
by *auto*
then show *?thesis*
proof *cases*
case *summable*
then show *?thesis*
apply (*subst asm-rl*[*of* $\langle (\lambda x.\ c *_C f\ x) = (\lambda y.\ c *_C y)\ o\ f \rangle$], *simp add*: *o-def*)
apply (*rule infsum-comm-additive*)
using *summable* **by** (*auto simp add*: *scaleC-right.additive-axioms*)
next
case *c0*
then show *?thesis* **by** *auto*
next
case *not-summable*
have $\langle \neg\ (\lambda x.\ c *_C f\ x)$ *summable-on* $A \rangle$
proof (*rule notI*)
assume $\langle (\lambda x.\ c *_C f\ x)$ *summable-on* $A \rangle$
then have $\langle (\lambda x.$ *inverse* $c *_C c *_C f\ x)$ *summable-on* $A \rangle$
using *summable-on-scaleC-right* **by** *blast*
then have $\langle f$ *summable-on* $A \rangle$
using *not-summable* **by** *auto*
with *not-summable* **show** *False*
by *simp*
qed

then show *?thesis*
by (*simp add*: *infsum-not-exists not-summable*(*1*))
qed
qed

lemma *has-sum-scaleC-right*:
fixes *f* :: ‹*'a* ⇒ *'b* :: *complex-normed-vector*›
assumes ‹(*f has-sum s*) *A*›
shows ‹((λ*x*. *c* ∗$_C$ *f x*) *has-sum c* ∗$_C$ *s*) *A*›
apply (*rule has-sum-bounded-clinear*[**where** *h*=‹(∗$_C$) *c*›])
using *bounded-clinear-scaleC-right assms* **by** *auto*

lemmas *sums-of-complex* = *bounded-linear.sums* [*OF bounded-clinear-of-complex*[*THEN bounded-clinear.bounded-linear*]]
lemmas *summable-of-complex* = *bounded-linear.summable* [*OF bounded-clinear-of-complex*[*THEN bounded-clinear.bounded-linear*]]
lemmas *suminf-of-complex* = *bounded-linear.suminf* [*OF bounded-clinear-of-complex*[*THEN bounded-clinear.bounded-linear*]]

lemmas *sums-scaleC-left* = *bounded-linear.sums*[*OF bounded-clinear-scaleC-left*[*THEN bounded-clinear.bounded-linear*]]
lemmas *summable-scaleC-left* = *bounded-linear.summable*[*OF bounded-clinear-scaleC-left*[*THEN bounded-clinear.bounded-linear*]]
lemmas *suminf-scaleC-left* = *bounded-linear.suminf*[*OF bounded-clinear-scaleC-left*[*THEN bounded-clinear.bounded-linear*]]

lemmas *sums-scaleC-right* = *bounded-linear.sums*[*OF bounded-clinear-scaleC-right*[*THEN bounded-clinear.bounded-linear*]]
lemmas *summable-scaleC-right* = *bounded-linear.summable*[*OF bounded-clinear-scaleC-right*[*THEN bounded-clinear.bounded-linear*]]
lemmas *suminf-scaleC-right* = *bounded-linear.suminf*[*OF bounded-clinear-scaleC-right*[*THEN bounded-clinear.bounded-linear*]]

lemma *closed-scaleC*:
fixes *S*::‹*'a*::*complex-normed-vector set*› **and** *a* :: *complex*
assumes ‹*closed S*›
shows ‹*closed* ((∗$_C$) *a* ‘ *S*)›
proof (*cases* ‹*a* = *0*›)
case *True*
then show *?thesis*
apply (*cases* ‹*S* = {}›)
by (*auto simp*: *image-constant*)
next
case *False*
then have ‹(∗$_C$) *a* ‘ *S* = (∗$_C$) (*inverse a*) −‘ *S*›
by (*auto simp add*: *rev-image-eqI*)
moreover have ‹*closed* ((∗$_C$) (*inverse a*) −‘ *S*)›
by (*simp add*: *assms continuous-closed-vimage*)

```
  ultimately show ?thesis
    by simp
qed

lemma closure-scaleC:
  fixes S::‹'a::complex-normed-vector set›
  shows ‹closure ((*C) a ‘ S) = (*C) a ‘ closure S›
proof
  have ‹closed (closure S)›
    by simp
  show closure ((*C) a ‘ S) ⊆ (*C) a ‘ closure S
    by (simp add: closed-scaleC closure-minimal closure-subset image-mono)

  have x ∈ closure ((*C) a ‘ S)
    if x ∈ (*C) a ‘ closure S
    for x :: 'a
  proof−
    obtain t where ‹x = ((*C) a) t› and ‹t ∈ closure S›
      using ‹x ∈ (*C) a ‘ closure S› by auto
    have ‹∃ s. (∀ n. s n ∈ S) ∧ s ⟶ t›
      using ‹t ∈ closure S› Elementary-Topology.closure-sequential
      by blast
    then obtain s where ‹∀ n. s n ∈ S› and ‹s ⟶ t›
      by blast
    have ‹(∀ n. scaleC a (s n) ∈ ((*C) a ‘ S))›
      using ‹∀ n. s n ∈ S› by blast
    moreover have ‹(λ n. scaleC a (s n)) ⟶ x›
    proof−
      have ‹isCont (scaleC a) t›
        by simp
      thus ?thesis
        using ‹s ⟶ t› ‹x = ((*C) a) t›
        by (simp add: isCont-tendsto-compose)
    qed
    ultimately show ?thesis using Elementary-Topology.closure-sequential
      by metis
  qed
  thus (*C) a ‘ closure S ⊆ closure ((*C) a ‘ S) by blast
qed

lemma onorm-scalarC:
  fixes f :: ‹'a::complex-normed-vector ⇒ 'b::complex-normed-vector›
  assumes a1: ‹bounded-clinear f›
  shows ‹onorm (λ x. r *C (f x)) = (cmod r) * onorm f›
proof−
  have ‹(norm (f x)) / norm x ≤ onorm f›
    for x
    using a1
    by (simp add: bounded-clinear.bounded-linear le-onorm)
```

hence *t2*: ‹*bdd-above* {(*norm* (*f x*)) / *norm x* | *x*. *True*}›
by *fastforce*
have ‹*continuous-on UNIV* ((∗) *w*) ›
for *w*::*real*
by *simp*
hence ‹*isCont* (((∗) (*cmod r*))) *x*›
for *x*
by *simp*
hence *t3*: ‹*continuous* (*at-left* (*Sup* {(*norm* (*f x*)) / *norm x* | *x*. *True*})) ((∗) (*cmod r*))›
using *Elementary-Topology.continuous-at-imp-continuous-within*
by *blast*
have ‹{(*norm* (*f x*)) / *norm x* | *x*. *True*} ≠ {}›
by *blast*
moreover have ‹*mono* ((∗) (*cmod r*))›
by (*simp add*: *monoI ordered-comm-semiring-class.comm-mult-left-mono*)
ultimately have ‹*Sup* {((∗) (*cmod r*)) ((*norm* (*f x*)) / *norm x*) | *x*. *True*}
= ((∗) (*cmod r*)) (*Sup* {(*norm* (*f x*)) / *norm x* | *x*. *True*})›
using *t2 t3*
by (*simp add*: *continuous-at-Sup-mono full-SetCompr-eq image-image*)
hence ‹*Sup* {(*cmod r*) ∗ ((*norm* (*f x*)) / *norm x*) | *x*. *True*}
= (*cmod r*) ∗ (*Sup* {(*norm* (*f x*)) / *norm x* | *x*. *True*})›
by *blast*
moreover have ‹*Sup* {(*cmod r*) ∗ ((*norm* (*f x*)) / *norm x*) | *x*. *True*}
= (*SUP x*. *cmod r* ∗ *norm* (*f x*) / *norm x*)›
by (*simp add*: *full-SetCompr-eq*)
moreover have ‹(*Sup* {(*norm* (*f x*)) / *norm x* | *x*. *True*})
= (*SUP x*. *norm* (*f x*) / *norm x*)›
by (*simp add*: *full-SetCompr-eq*)
ultimately have *t1*: (*SUP x*. *cmod r* ∗ *norm* (*f x*) / *norm x*)
= *cmod r* ∗ (*SUP x*. *norm* (*f x*) / *norm x*)
by *simp*
have ‹*onorm* (λ *x*. *r* ∗$_C$ (*f x*)) = (*SUP x*. *norm* ((λ *t*. *r* ∗$_C$ (*f t*)) *x*) / *norm x*)›
by (*simp add*: *onorm-def*)
hence ‹*onorm* (λ *x*. *r* ∗$_C$ (*f x*)) = (*SUP x*. (*cmod r*) ∗ (*norm* (*f x*)) / *norm x*)›
by *simp*
also have ‹... = (*cmod r*) ∗ (*SUP x*. (*norm* (*f x*)) / *norm x*)›
using *t1*.
finally show *?thesis*
by (*simp add*: *onorm-def*)
qed

lemma *onorm-scaleC-left-lemma*:
fixes *f* :: *'a*::*complex-normed-vector*
assumes *r*: *bounded-clinear r*
shows *onorm* (λ*x*. *r x* ∗$_C$ *f*) ≤ *onorm r* ∗ *norm f*
proof (*rule onorm-bound*)
fix *x*
have *norm* (*r x* ∗$_C$ *f*) = *norm* (*r x*) ∗ *norm f*

```
    by simp
  also have ... ≤ onorm r * norm x * norm f
     by (simp add: bounded-clinear.bounded-linear mult.commute mult-left-mono
onorm r)
  finally show norm (r x *C f) ≤ onorm r * norm f * norm x
    by (simp add: ac-simps)
  show 0 ≤ onorm r * norm f
    by (simp add: bounded-clinear.bounded-linear onorm-pos-le r)
qed

lemma onorm-scaleC-left:
  fixes f :: 'a::complex-normed-vector
  assumes f: bounded-clinear r
  shows onorm (λx. r x *C f) = onorm r * norm f
proof (cases f = 0)
  assume f ≠ 0
  show ?thesis
  proof (rule order-antisym)
    show onorm (λx. r x *C f) ≤ onorm r * norm f
      using f by (rule onorm-scaleC-left-lemma)
  next
    have bl1: bounded-clinear (λx. r x *C f)
      by (metis bounded-clinear-scaleC-const f)
    have x1:bounded-clinear (λx. r x * norm f)
      by (metis bounded-clinear-mult-const f)

    have onorm r ≤ onorm (λx. r x * complex-of-real (norm f)) / norm f
       if onorm r ≤ onorm (λx. r x * complex-of-real (norm f)) * cmod (1 /
complex-of-real (norm f))
        and f ≠ 0
      using that
      by (smt (verit) divide-inverse mult-1 norm-divide norm-ge-zero norm-of-real
of-real-1 of-real-eq-iff of-real-mult)
    hence onorm r ≤ onorm (λx. r x * norm f) * inverse (norm f)
      using ‹f ≠ 0› onorm-scaleC-left-lemma[OF x1, of inverse (norm f)]
      by (simp add: inverse-eq-divide)
    also have onorm (λx. r x * norm f) ≤ onorm (λx. r x *C f)
    proof (rule onorm-bound)
      have bounded-linear (λx. r x *C f)
        using bl1 bounded-clinear.bounded-linear by auto
      thus 0 ≤ onorm (λx. r x *C f)
        by (rule Operator-Norm.onorm-pos-le)
     show cmod (r x * complex-of-real (norm f)) ≤ onorm (λx. r x *C f) * norm
x
       for x :: 'b
        by (smt (verit) ‹bounded-linear (λx. r x *C f)› norm-ge-zero norm-mult
norm-of-real norm-scaleC onorm)
    qed
    finally show onorm r * norm f ≤ onorm (λx. r x *C f)
```

using ‹$f \neq 0$›
by (*simp add*: *inverse-eq-divide pos-le-divide-eq mult.commute*)
qed
qed (*simp add*: *onorm-zero*)

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*)

7.4 Finite dimension and canonical basis

lemma *vector-finitely-spanned*:
assumes ‹$z \in$ *cspan T*›
shows ‹$\exists$ *S*. *finite S* $\wedge$ $S \subseteq T \wedge z \in$ *cspan S*›
proof−
have ‹$\exists$ *S r*. *finite S* $\wedge$ $S \subseteq T \wedge z = (\sum a{\in}S.\ r\ a *_C\ a)$›
using *complex-vector.span-explicit*[**where** $b = T$]
assms **by** *auto*
then obtain *S r* **where** ‹*finite S*› **and** ‹$S \subseteq T$› **and** ‹$z = (\sum a{\in}S.\ r\ a *_C\ a)$›
by *blast*
thus *?thesis*
by (*meson complex-vector.span-scale complex-vector.span-sum complex-vector.span-superset subset-iff*)
qed

setup ‹*Sign.add-const-constraint* (*Complex-Vector-Spaces0.cindependent*, *SOME* ***typ*** ‹*'a set* ⇒ *bool*›)›
setup ‹*Sign.add-const-constraint* (***const-name*** ‹*cdependent*›, *SOME* ***typ*** ‹*'a set* ⇒ *bool*›)›
setup ‹*Sign.add-const-constraint* (***const-name*** ‹*cspan*›, *SOME* ***typ*** ‹*'a set* ⇒ *'a set*›)›

class *cfinite-dim* = *complex-vector* +
assumes *cfinitely-spanned*: $\exists S{::}'a$ *set*. *finite S* $\wedge$ *cspan S* = *UNIV*

class *basis-enum* = *complex-vector* +
fixes *canonical-basis* :: ‹*'a list*›
and *canonical-basis-length* :: ‹*'a itself* ⇒ *nat*›
assumes *distinct-canonical-basis*[*simp*]:
distinct canonical-basis
and *is-cindependent-set*[*simp*]:
cindependent (*set canonical-basis*)
and *is-generator-set*[*simp*]:
cspan (*set canonical-basis*) = *UNIV*
and *canonical-basis-length*:
‹*canonical-basis-length TYPE*(*'a*) = *length canonical-basis*›

setup ‹*Sign.add-const-constraint* (*Complex-Vector-Spaces0.cindependent*, *SOME* ***typ*** ‹'*a*::*complex-vector set* ⇒ *bool*›)›
setup ‹*Sign.add-const-constraint* (***const-name*** ‹*cdependent*›, *SOME* ***typ*** ‹'*a*::*complex-vector set* ⇒ *bool*›)›
setup ‹*Sign.add-const-constraint* (***const-name*** ‹*cspan*›, *SOME* ***typ*** ‹'*a*::*complex-vector set* ⇒ '*a set*›)›

instantiation *complex* :: *basis-enum* **begin**
definition *canonical-basis* = [*1*::*complex*]
definition ‹*canonical-basis-length* (-::*complex itself*) = *1*›
instance
proof
show *distinct* (*canonical-basis*::*complex list*)
by (*simp add*: *canonical-basis-complex-def*)
show *cindependent* (*set* (*canonical-basis*::*complex list*))
unfolding *canonical-basis-complex-def*
by *auto*
show *cspan* (*set* (*canonical-basis*::*complex list*)) = *UNIV*
unfolding *canonical-basis-complex-def*
apply (*auto simp add*: *cspan-raw-def vector-space-over-itself.span-Basis*)
by (*metis complex-scaleC-def complex-vector.span-base complex-vector.span-scale cspan-raw-def insertI1 mult.right-neutral*)
show ‹*canonical-basis-length TYPE*(*complex*) = *length* (*canonical-basis* :: *complex list*)›
by (*simp add*: *canonical-basis-length-complex-def canonical-basis-complex-def*)
qed
end

lemma *cdim-UNIV-basis-enum*[*simp*]: ‹*cdim* (*UNIV*::'*a*::*basis-enum set*) = *length* (*canonical-basis*::'*a list*)›
apply (*subst is-generator-set*[*symmetric*])
apply (*subst complex-vector.dim-span-eq-card-independent*)
apply (*rule is-cindependent-set*)
using *distinct-canonical-basis distinct-card* **by** *blast*

lemma *finite-basis*: ∃ *basis*::'*a*::*cfinite-dim set*. *finite basis* ∧ *cindependent basis* ∧ *cspan basis* = *UNIV*
proof −
from *cfinitely-spanned*
obtain *S* :: ‹'*a set*› **where** ‹*finite S*› **and** ‹*cspan S* = *UNIV*›
by *auto*
from *complex-vector.maximal-independent-subset*
obtain *B* :: ‹'*a set*› **where** ‹*B* ⊆ *S*› **and** ‹*cindependent B*› **and** ‹*S* ⊆ *cspan B*›
by *metis*
moreover have ‹*finite B*›
using ‹*B* ⊆ *S*› ‹*finite S*›
by (*meson finite-subset*)
moreover have ‹*cspan B* = *UNIV*›

using ‹*cspan S = UNIV*› ‹*S ⊆ cspan B*›
by (*metis complex-vector.span-eq top-greatest*)
ultimately show *?thesis*
by *auto*
qed

instance *basis-enum* ⊆ *cfinite-dim*
apply *intro-classes*
apply (*rule exI*[*of* - ‹*set canonical-basis*›])
using *is-cindependent-set is-generator-set* **by** *auto*

lemma *cindependent-cfinite-dim-finite*:
assumes ‹*cindependent* (*S*::*'a*::*cfinite-dim set*)›
shows ‹*finite S*›
by (*metis assms cfinitely-spanned complex-vector.independent-span-bound top-greatest*)

lemma *cfinite-dim-finite-subspace-basis*:
assumes ‹*csubspace X*›
shows ∃ *basis*::*'a*::*cfinite-dim set*. *finite basis* ∧ *cindependent basis* ∧ *cspan basis* = *X*
by (*meson assms cindependent-cfinite-dim-finite complex-vector.basis-exists complex-vector.span-subspace*)

The following auxiliary lemma (*finite-span-complete-aux*) shows more or less the same as *finite-span-representation-bounded*, *finite-span-complete* below (see there for an intuition about the mathematical content of the lemmas). However, there is one difference: Here we additionally assume here that there is a bijection rep/abs between a finite type *'basis* and the set B. This is needed to be able to use results about euclidean spaces that are formulated w.r.t. the type class *finite*

Since we anyway assume that B is finite, this added assumption does not make the lemma weaker. However, we cannot derive the existence of *'basis* inside the proof (HOL does not support such reasoning). Therefore we have the type *'basis* as an explicit assumption and remove it using *internalize-sort* after the proof.

lemma *finite-span-complete-aux*:
fixes *b* :: *'b*::*real-normed-vector* **and** *B* :: *'b set*
and *rep* :: *'basis*::*finite* ⇒ *'b* **and** *abs* :: *'b* ⇒ *'basis*
assumes *t*: *type-definition rep abs B*
and *t1*: *finite B* **and** *t2*: *b*∈*B* **and** *t3*: *independent B*
shows ∃ *D*>*0*. ∀ *ψ*. *norm* (*representation B ψ b*) ≤ *norm ψ* ∗ *D*
and *complete* (*span B*)
proof −
define *repr* **where** *repr* = *real-vector.representation B*
define *repr'* **where** *repr' ψ* = *Abs-euclidean-space* (*repr ψ o rep*) **for** *ψ*
define *comb* **where** *comb l* = (∑ *b*∈*B*. *l b* ∗$_R$ *b*) **for** *l*
define *comb'* **where** *comb' l* = *comb* (*Rep-euclidean-space l o abs*) **for** *l*

have *comb-cong*: *comb x = comb y* **if** $\bigwedge z.\ z{\in}B \Longrightarrow x\ z = y\ z$ **for** *x y*
unfolding *comb-def* **using** *that* **by** *auto*
have *comb-repr*[*simp*]: *comb* (*repr* ψ) = ψ **if** $\psi \in$ *real-vector.span B* **for** ψ
using ‹*comb* ≡ $\lambda l.\ \sum b{\in}B.\ l\ b *_R b$› *local.repr-def real-vector.sum-representation-eq t1 t3 that*
by *fastforce*

have *w5*:($\sum b \mid (b \in B \longrightarrow x\ b \neq 0) \wedge b \in B.\ x\ b *_R b$) =
($\sum b{\in}B.\ x\ b *_R b$) **for** *x*
using ‹*finite B*›
by (*smt DiffD1 DiffD2 mem-Collect-eq real-vector.scale-eq-0-iff subset-eq sum.mono-neutral-left*)
have *representation B* ($\sum b{\in}B.\ x\ b *_R b$) = ($\lambda b.$ *if* $b \in B$ *then x b else 0*)
for *x*
proof (*rule real-vector.representation-eqI*)
show *independent B*
by (*simp add*: *t3*)
show ($\sum b{\in}B.\ x\ b *_R b$) ∈ *span B*
by (*meson real-vector.span-scale real-vector.span-sum real-vector.span-superset subset-iff*)
show $b \in B$
if (*if* $b \in B$ *then x b else 0*) ≠ *0*
for $b :: 'b$
using *that*
by *meson*
show *finite* {$b.$ (*if* $b \in B$ *then x b else 0*) ≠ *0*}
using *t1* **by** *auto*
show ($\sum b \mid$ (*if* $b \in B$ *then x b else 0*) ≠ *0*. (*if* $b \in B$ *then x b else 0*) $*_R$ *b*) = ($\sum b{\in}B.\ x\ b *_R b$)
using *w5*
by *simp*
qed
hence *repr-comb*[*simp*]: *repr* (*comb x*) = ($\lambda b.$ *if* $b{\in}B$ *then x b else 0*) **for** *x*
unfolding *repr-def comb-def*.
have *repr-bad*[*simp*]: *repr* ψ = (λ-. *0*) **if** $\psi \notin$ *real-vector.span B* **for** ψ
unfolding *repr-def* **using** *that*
by (*simp add*: *real-vector.representation-def*)
have [*simp*]: *repr′* ψ = *0* **if** $\psi \notin$ *real-vector.span B* **for** ψ
unfolding *repr′-def repr-bad*[*OF that*]
apply *transfer*
by *auto*
have *comb′-repr′*[*simp*]: *comb′* (*repr′* ψ) = ψ
if $\psi \in$ *real-vector.span B* **for** ψ
proof −
have *x1*: (*repr* ψ ∘ *rep* ∘ *abs*) *z* = *repr* ψ *z*
if $z \in B$
for *z*
unfolding *o-def*
using *t that type-definition.Abs-inverse* **by** *fastforce*
have *comb′* (*repr′* ψ) = *comb* ((*repr* ψ ∘ *rep*) ∘ *abs*)

```
    unfolding comb'-def repr'-def
    by (subst Abs-euclidean-space-inverse; simp)
  also have ... = comb (repr ψ)
    using x1 comb-cong by blast
  also have ... = ψ
    using that by simp
  finally show ?thesis by −
qed

have t1: Abs-euclidean-space (Rep-euclidean-space t) = t
  if ⋀x. rep x ∈ B
  for t::'a euclidean-space
  apply (subst Rep-euclidean-space-inverse)
  by simp
have Abs-euclidean-space
   (λy. if rep y ∈ B
        then Rep-euclidean-space x y
        else 0) = x
  for x
  using type-definition.Rep[OF t] apply simp
  using t1 by blast
hence Abs-euclidean-space
   (λy. if rep y ∈ B
        then Rep-euclidean-space x (abs (rep y))
        else 0) = x
  for x
  apply (subst type-definition.Rep-inverse[OF t])
  by simp
hence repr'-comb'[simp]: repr' (comb' x) = x for x
  unfolding comb'-def repr'-def o-def
  by simp
have sphere: compact (sphere 0 d :: 'basis euclidean-space set) for d
  using compact-sphere by blast
have complete (UNIV :: 'basis euclidean-space set)
  by (simp add: complete-UNIV)


have (∑ b∈B. (Rep-euclidean-space (x + y) ∘ abs) b *R b) = (∑ b∈B. (Rep-euclidean-space
x ∘ abs) b *R b) + (∑ b∈B. (Rep-euclidean-space y ∘ abs) b *R b)
  for x :: 'basis euclidean-space
    and y :: 'basis euclidean-space
  apply (transfer fixing: abs)
  by (simp add: scaleR-add-left sum.distrib)
moreover have (∑ b∈B. (Rep-euclidean-space (c *R x) ∘ abs) b *R b) = c *R
(∑ b∈B. (Rep-euclidean-space x ∘ abs) b *R b)
  for c :: real
    and x :: 'basis euclidean-space
  apply (transfer fixing: abs)
  by (simp add: real-vector.scale-sum-right)
```

ultimately have *blin-comb'*: *bounded-linear comb'*
unfolding *comb-def comb'-def*
by (*rule bounded-linearI'*)
hence *continuous-on X comb'* **for** *X*
by (*simp add*: *linear-continuous-on*)
hence *compact* (*comb'* ‘ *sphere 0 d*) **for** *d*
using *sphere*
by (*rule compact-continuous-image*)
hence *compact-norm-comb'*: *compact* (*norm* ‘ *comb'* ‘ *sphere 0 1*)
using *compact-continuous-image continuous-on-norm-id* **by** *blast*
have *not0*: *0* ∉ *norm* ‘ *comb'* ‘ *sphere 0 1*
proof (*rule ccontr*, *simp*)
assume *0* ∈ *norm* ‘ *comb'* ‘ *sphere 0 1*
then obtain *x* **where** *nc0*: *norm* (*comb' x*) = *0* **and** *x*: *x* ∈ *sphere 0 1*
by *auto*
hence *comb' x* = *0*
by *simp*
hence *repr'* (*comb' x*) = *0*
unfolding *repr'-def o-def repr-def* **apply** *simp*
by (*smt repr'-comb' blin-comb' dist-0-norm linear-simps*(*3*) *mem-sphere norm-zero x*)
hence *x* = *0*
by *auto*
with *x* **show** *False*
by *simp*
qed

have *closed* (*norm* ‘ *comb'* ‘ *sphere 0 1*)
using *compact-imp-closed compact-norm-comb'* **by** *blast*
moreover have *0* ∉ *norm* ‘ *comb'* ‘ *sphere 0 1*
by (*simp add*: *not0*)
ultimately have ∃ *d*>*0*. ∀ *x*∈*norm* ‘ *comb'* ‘ *sphere 0 1*. *d* ≤ *dist 0 x*
by (*meson separate-point-closed*)

then obtain *d* **where** *d*: *x*∈*norm* ‘ *comb'* ‘ *sphere 0 1* ⟹ *d* ≤ *dist 0 x*
and *d* > *0* **for** *x*
by *metis*
define *D* **where** *D* = *1*/*d*
hence *D* > *0*
using ‹*d*>*0*› **unfolding** *D-def* **by** *auto*
have *x* ≥ *d*
if *x*∈*norm* ‘ *comb'* ‘ *sphere 0 1*
for *x*
using *d that*
apply *auto*
by *fastforce*
hence ∗: *norm* (*comb' x*) ≥ *d* **if** *norm x* = *1* **for** *x*
using *that* **by** *auto*
have *norm-comb'*: *norm* (*comb' x*) ≥ *d* ∗ *norm x* **for** *x*

```
  proof (cases x=0)
    show d * norm x ≤ norm (comb' x)
      if x = 0
      using that
      by simp
    show d * norm x ≤ norm (comb' x)
      if x ≠ 0
      using that
      using *[of (1/norm x) *R x]
      unfolding linear-simps(5)[OF blin-comb']
      apply auto
      by (simp add: le-divide-eq)
  qed

  have *: norm (repr' ψ) ≤ norm ψ * D for ψ
  proof (cases ψ ∈ real-vector.span B)
    show norm (repr' ψ) ≤ norm ψ * D
      if ψ ∈ span B
      using that     unfolding D-def
      using norm-comb'[of repr' ψ] ‹d>0›
      by (simp-all add: linordered-field-class.mult-imp-le-div-pos mult.commute)

    show norm (repr' ψ) ≤ norm ψ * D
      if ψ ∉ span B
      using that ‹0 < D› by auto
  qed

  hence norm (Rep-euclidean-space (repr' ψ) (abs b)) ≤ norm ψ * D for ψ
  proof −
   have (Rep-euclidean-space (repr' ψ) (abs b)) = repr' ψ • euclidean-space-basis-vector
(abs b)
      apply (transfer fixing: abs b)
      by auto
    also have |...| ≤ norm (repr' ψ)
      apply (rule Basis-le-norm)
      unfolding Basis-euclidean-space-def by simp
    also have ... ≤ norm ψ * D
      using * by auto
    finally show ?thesis by simp
  qed
  hence norm (repr ψ b) ≤ norm ψ * D for ψ
    unfolding repr'-def
    by (smt ‹comb' ≡ λl. comb (Rep-euclidean-space l ∘ abs)›
        ‹repr' ≡ λψ. Abs-euclidean-space (repr ψ ∘ rep)› comb'-repr' comp-apply
norm-le-zero-iff
       repr-bad repr-comb)
  thus ∃ D>0. ∀ ψ. norm (repr ψ b) ≤ norm ψ * D
    using ‹D>0› by auto
  from ‹d>0›
```

have *complete-comb'*: *complete* (*comb'* ‘ *UNIV*)
proof (*rule complete-isometric-image*)
show *subspace* (*UNIV*::*'basis euclidean-space set*)
by *simp*
show *bounded-linear comb'*
by (*simp add*: *blin-comb'*)
show ∀ *x*∈*UNIV*. *d* ∗ *norm x* ≤ *norm* (*comb' x*)
by (*simp add*: *norm-comb'*)
show *complete* (*UNIV*::*'basis euclidean-space set*)
by (*simp add*: ‹*complete UNIV*›)
qed

have *range-comb'*: *comb'* ‘ *UNIV* = *real-vector.span B*
proof (*auto simp*: *image-def*)
show *comb' x* ∈ *real-vector.span B* **for** *x*
by (*metis comb'-def comb-cong comb-repr local.repr-def repr-bad repr-comb real-vector.representation-zero real-vector.span-zero*)
next
fix *ψ* **assume** *ψ* ∈ *real-vector.span B*
then obtain *f* **where** *f*: *comb f* = *ψ*
apply *atomize-elim*
unfolding *span-finite*[*OF* ‹*finite B*›] *comb-def*
by *auto*
define *f'* **where** *f' b* = (*if b*∈*B then f b else 0*) **for** *b* :: *'b*
have *f'*: *comb f'* = *ψ*
unfolding *f*[*symmetric*]
apply (*rule comb-cong*)
unfolding *f'-def* **by** *simp*
define *x* :: *'basis euclidean-space* **where** *x* = *Abs-euclidean-space* (*f' o rep*)
have *ψ* = *comb' x*
by (*metis* (*no-types*, *lifting*) ‹*ψ* ∈ *span B*› ‹*repr'* ≡ *λψ*. *Abs-euclidean-space* (*repr ψ* ∘ *rep*)›
comb'-repr' f' fun.map-cong repr-comb t type-definition.Rep-range x-def)
thus ∃ *x*. *ψ* = *comb' x*
by *auto*
qed

from *range-comb' complete-comb'*
show *complete* (*real-vector.span B*)
by *simp*
qed

lemma *finite-span-complete*[*simp*]:
fixes *A* :: *'a*::*real-normed-vector set*
assumes *finite A*
shows *complete* (*span A*)

The span of a finite set is complete.

proof (*cases A* ≠ {} ∧ *A* ≠ {*0*})

```
  case True
  obtain B where
    BT: real-vector.span B = real-vector.span A
    and independent B
    and finite B
      by (meson True assms finite-subset real-vector.maximal-independent-subset
real-vector.span-eq
        real-vector.span-superset subset-trans)

  have B≠{}
    apply (rule ccontr, simp)
    using BT True
    by (metis real-vector.span-superset real-vector.span-empty subset-singletonD)


  {

    assume ∃(Rep :: 'basisT⇒'a) Abs. type-definition Rep Abs B
   then obtain rep :: 'basisT ⇒ 'a and abs :: 'a ⇒ 'basisT where t: type-definition
rep abs B
      by auto
    have basisT-finite: class.finite TYPE('basisT)
      apply intro-classes
      using ‹finite B› t
      by (metis (mono-tags, opaque-lifting) ex-new-if-finite finite-imageI image-eqI
type-definition-def)
    note finite-span-complete-aux(2)[internalize-sort 'basis::finite]
    note this[OF basisT-finite t]
  }
  note this[cancel-type-definition, OF ‹B≠{}› ‹finite B› - ‹independent B›]
  hence complete (real-vector.span B)
    using ‹B≠{}› by auto
  thus complete (real-vector.span A)
    unfolding BT by simp
next
  case False
  thus ?thesis
    using complete-singleton by auto
qed

lemma finite-span-representation-bounded:
  fixes B :: 'a::real-normed-vector set
  assumes finite B and independent B
  shows ∃D>0. ∀ψ b. abs (representation B ψ b) ≤ norm ψ * D
```

Assume B is a finite linear independent set of vectors (in a real normed vector space). Let α_b^ψ be the coefficients of ψ expressed as a linear combination over B. Then α is is uniformly cblinfun (i.e., $|\alpha_b^\psi \leq D\|\psi\|\psi$ for some D independent of ψ, b).

(This also holds when b is not in the span of B because of the way *real-vector.representation* is defined in this corner case.)

proof (*cases* $B \neq \{\}$)
case *True*

define *repr* **where** *repr* = *real-vector.representation B*
{

assume $\exists$ (*Rep* :: $'basisT \Rightarrow 'a$) *Abs*. *type-definition Rep Abs B*
then obtain *rep* :: $'basisT \Rightarrow 'a$ **and** *abs* :: $'a \Rightarrow 'basisT$ **where** *t*: *type-definition rep abs B*
by *auto*

have *basisT-finite*: *class.finite TYPE*($'basisT$)
apply *intro-classes*
using ‹*finite B*› *t*
by (*metis* (*mono-tags*, *opaque-lifting*) *ex-new-if-finite finite-imageI image-eqI type-definition-def*)

note *finite-span-complete-aux*(*1*)[*internalize-sort* $'basis$::*finite*]

note *this*[*OF basisT-finite t*]
}

note *this*[*cancel-type-definition*, *OF True* ‹*finite B*› - ‹*independent B*›]

hence *d2*:$\exists\, D.\ \forall \psi.\ D>0 \wedge norm\ (repr\ \psi\ b) \leq norm\ \psi * D$ **if** ‹$b \in B$› **for** *b*
by (*simp add*: *repr-def that True*)
have *d1*: $(\bigwedge b.\ b \in B \Longrightarrow \exists\, D.\ \forall \psi.\ 0 < D \wedge norm\ (repr\ \psi\ b) \leq norm\ \psi * D) \Longrightarrow \exists\, D.\ \forall\, b\ \psi.\ b \in B \longrightarrow 0 < D\ b \wedge norm\ (repr\ \psi\ b) \leq norm\ \psi * D\ b$
apply (*rule choice*) **by** *auto*
then obtain *D* **where** *D*: $D\ b > 0 \wedge norm\ (repr\ \psi\ b) \leq norm\ \psi * D\ b$ **if** $b \in B$ **for** *b* ψ
apply *atomize-elim*
using *d2* **by** *blast*

hence *Dpos*: $D\ b > 0$ **and** *Dbound*: $norm\ (repr\ \psi\ b) \leq norm\ \psi * D\ b$
if $b \in B$ **for** *b* ψ
using *that* **by** *auto*
define *Dall* **where** *Dall* = *Max* (*D*‘*B*)
have *Dall* > *0*
unfolding *Dall-def* **using** ‹*finite B*› ‹$B \neq \{\}$› *Dpos*
by (*metis* (*mono-tags*, *lifting*) *Max-in finite-imageI image-iff image-is-empty*)
have *Dall* $\geq$ *D b* **if** $b \in B$ **for** *b*
unfolding *Dall-def* **using** ‹*finite B*› *that* **by** *auto*

with *Dbound*
have *norm* (*repr* ψ *b*) $\leq$ *norm* ψ $*$ *Dall* **if** $b \in B$ **for** *b* ψ
using *that*
by (*smt mult-left-mono norm-not-less-zero*)
moreover have *norm* (*repr* ψ *b*) $\leq$ *norm* ψ $*$ *Dall* **if** $b \notin B$ **for** *b* ψ
unfolding *repr-def* **using** *real-vector.representation-ne-zero True*
by (*metis calculation empty-subsetI less-le-trans local.repr-def norm-ge-zero norm-zero not-less subsetI subset-antisym*)
ultimately show $\exists D{>}0.\ \forall \psi\ b.$ *abs* (*repr* ψ *b*) $\leq$ *norm* ψ $*$ *D*
using ‹*Dall* $>$ *0*› *real-norm-def* **by** *metis*
next
case *False*
thus *?thesis*
unfolding *repr-def* **using** *real-vector.representation-ne-zero*[*of B*]
using *nice-ordered-field-class.linordered-field-no-ub* **by** *fastforce*
qed

hide-fact *finite-span-complete-aux*

lemma *finite-cspan-complete*[*simp*]:
fixes *B* :: *'a*::*complex-normed-vector set*
assumes *finite B*
shows *complete* (*cspan B*)
by (*simp add*: *assms cspan-as-span*)

lemma *finite-span-closed*[*simp*]:
fixes *B* :: *'a*::*real-normed-vector set*
assumes *finite B*
shows *closed* (*real-vector.span B*)
by (*simp add*: *assms complete-imp-closed*)

lemma *finite-cspan-closed*[*simp*]:
fixes *S*::‹*'a*::*complex-normed-vector set*›
assumes *a1*: ‹*finite S*›
shows ‹*closed* (*cspan S*)›
by (*simp add*: *assms complete-imp-closed*)

lemma *closure-finite-cspan*:
fixes *T*::‹*'a*::*complex-normed-vector set*›
assumes ‹*finite T*›
shows ‹*closure* (*cspan T*) = *cspan T*›
by (*simp add*: *assms*)

lemma *finite-cspan-crepresentation-bounded*:
fixes *B* :: *'a*::*complex-normed-vector set*

assumes *a1*: *finite B* **and** *a2*: *cindependent B*
shows $\exists D{>}0.\ \forall \psi\ b.\ cmod\ (crepresentation\ B\ \psi\ b) \leq norm\ \psi * D$
proof −
define B' **where** $B' = (B \cup scaleC\ \mathrm{i}\ `\ B)$
have *independent-B'*: *independent* B'
using B'*-def* ‹*cindependent B*›
by (*simp add*: *real-independent-from-complex-independent a1*)
have *finite* B'
unfolding B'*-def* **using** ‹*finite B*› **by** *simp*
obtain D' **where** $D' > 0$ **and** D': $norm\ (real\text{-}vector.representation\ B'\ \psi\ b) \leq norm\ \psi * D'$
for $\psi\ b$
apply *atomize-elim*
using *independent-B'* ‹*finite B'*›
by (*simp add*: *finite-span-representation-bounded*)

define D **where** $D = 2{*}D'$
from ‹$D' > 0$› **have** ‹$D > 0$›
unfolding *D-def* **by** *simp*
have $norm\ (crepresentation\ B\ \psi\ b) \leq norm\ \psi * D$ **for** $\psi\ b$
proof (*cases* $b{\in}B$)
case *True*
have *d3*: $norm\ \mathrm{i} = 1$
by *simp*

have $norm\ (\mathrm{i} *_C complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ (\mathrm{i} *_C b)))$
$= norm\ \mathrm{i} * norm\ (complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ (\mathrm{i} *_C b)))$
using *norm-scaleC* **by** *blast*
also have $\ldots = norm\ (complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ (\mathrm{i} *_C b)))$
using *d3* **by** *simp*
finally have *d2*:$norm\ (\mathrm{i} *_C complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ (\mathrm{i} *_C b)))$
$= norm\ (complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ (\mathrm{i} *_C b)))$.
have $norm\ (crepresentation\ B\ \psi\ b)$
$= norm\ (complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ b)$
$+ \mathrm{i} *_C complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ (\mathrm{i} *_C b)))$
by (*simp add*: B'*-def True a1 a2 crepresentation-from-representation*)
also have $\ldots \leq norm\ (complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ b))$
$+ norm\ (\mathrm{i} *_C complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ (\mathrm{i} *_C b)))$
using *norm-triangle-ineq* **by** *blast*
also have $\ldots = norm\ (complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ b))$
$+ norm\ (complex\text{-}of\text{-}real\ (real\text{-}vector.representation\ B'\ \psi\ (\mathrm{i} *_C b)))$
using *d2* **by** *simp*
also have $\ldots = norm\ (real\text{-}vector.representation\ B'\ \psi\ b)$
$+ norm\ (real\text{-}vector.representation\ B'\ \psi\ (\mathrm{i} *_C b))$
by *simp*
also have $\ldots \leq norm\ \psi * D' + norm\ \psi * D'$

```
    by (rule add-mono; rule D')
  also have ... ≤ norm ψ * D
    unfolding D-def by linarith
  finally show ?thesis
    by auto
 next
  case False
  hence crepresentation B ψ b = 0
    using complex-vector.representation-ne-zero by blast
  thus ?thesis
    by (smt ‹0 < D› norm-ge-zero norm-zero split-mult-pos-le)
 qed
 with ‹D > 0›
 show ?thesis
  by auto
qed

lemma bounded-clinear-finite-dim[simp]:
 fixes f :: ‹'a::{cfinite-dim,complex-normed-vector} ⇒ 'b::complex-normed-vector›
 assumes ‹clinear f›
 shows ‹bounded-clinear f›
proof −
 include norm-syntax
 obtain basis :: ‹'a set› where b1: complex-vector.span basis = UNIV
  and b2: cindependent basis
  and b3:finite basis
  using finite-basis by auto
 have ∃ C>0. ∀ψ b. cmod (crepresentation basis ψ b) ≤ ‖ψ‖ * C
  using finite-cspan-crepresentation-bounded[where B = basis] b2 b3 by blast
 then obtain C where s1: cmod (crepresentation basis ψ b) ≤ ‖ψ‖ * C
  and s2: C > 0
 for ψ b by blast
 define M where M = C * (∑ a∈basis. ‖f a‖)
 have ‖f x‖ ≤ ‖x‖ * M
  for x
 proof−
  define r where r b = crepresentation basis x b for b
  have x-span: x ∈ complex-vector.span basis
   by (simp add: b1)
  have f0: v ∈ basis
   if r v ≠ 0 for v
   using complex-vector.representation-ne-zero r-def that by auto
  have w:{a|a. r a ≠ 0} ⊆ basis
   using f0 by blast
  hence f1: finite {a|a. r a ≠ 0}
   using b3 rev-finite-subset by auto
  have f2: (∑ a| r a ≠ 0. r a *C a) = x
   unfolding r-def
   using b2 complex-vector.sum-nonzero-representation-eq x-span
```

Collect-cong **by** *fastforce*
have *g1*: $(\sum a{\in}basis.\ crepresentation\ basis\ x\ a *_C a) = x$
by (*simp add*: *b2 b3 complex-vector.sum-representation-eq x-span*)
have *f3*: $(\sum a{\in}basis.\ r\ a *_C a) = x$
unfolding *r-def*
by (*simp add*: *g1*)
hence $f\ x = f\ (\sum a{\in}basis.\ r\ a *_C a)$
by *simp*
also have $\ldots = (\sum a{\in}basis.\ r\ a *_C f\ a)$
by (*smt* (*verit*, *ccfv-SIG*) *assms complex-vector.linear-scale complex-vector.linear-sum sum.cong*)
finally have $f\ x = (\sum a{\in}basis.\ r\ a *_C f\ a)$**.**
hence $\|f\ x\| = \|(\sum a{\in}basis.\ r\ a *_C f\ a)\|$
by *simp*
also have $\ldots \leq (\sum a{\in}basis.\ \|r\ a *_C f\ a\|)$
by (*simp add*: *sum-norm-le*)
also have $\ldots \leq (\sum a{\in}basis.\ \|r\ a\| * \|f\ a\|)$
by *simp*
also have $\ldots \leq (\sum a{\in}basis.\ \|x\| * C * \|f\ a\|)$
using *sum-mono s1* **unfolding** *r-def*
by (*simp add*: *sum-mono mult-right-mono*)
also have $\ldots \leq \|x\| * C * (\sum a{\in}basis.\ \|f\ a\|)$
using *sum-distrib-left*
by (*smt sum.cong*)
also have $\ldots = \|x\| * M$
unfolding *M-def*
by *linarith*
finally show *?thesis* **.**
qed
thus *?thesis*
using *assms bounded-clinear-def bounded-clinear-axioms-def* **by** *blast*
qed

lemma *summable-on-scaleR-left-converse*:
— This result has nothing to do with the bounded operator library but it uses *finite-span-closed* so it is proven here.
fixes $f :: \langle 'b \Rightarrow real\rangle$
and $c :: \langle 'a :: real\text{-}normed\text{-}vector\rangle$
assumes $\langle c \neq 0\rangle$
assumes $\langle(\lambda x.\ f\ x *_R c)\ summable\text{-}on\ A\rangle$
shows $\langle f\ summable\text{-}on\ A\rangle$
proof −
define *fromR toR T* **where** $\langle fromR\ x = x *_R c\rangle$ **and** $\langle toR = inv\ fromR\rangle$ **and** $\langle T = range\ fromR\rangle$ **for** $x :: real$
have $\langle additive\ fromR\rangle$
by (*simp add*: *fromR-def additive.intro scaleR-left-distrib*)
have $\langle inj\ fromR\rangle$
by (*simp add*: *fromR-def* $\langle c \neq 0\rangle$ *inj-def*)
have *toR-fromR*: $\langle toR\ (fromR\ x) = x\rangle$ **for** x

```
    by (simp add: ‹inj fromR› toR-def)
  have fromR-toR: ‹fromR (toR x) = x› if ‹x ∈ T› for x
    by (metis T-def f-inv-into-f that toR-def)

  have 1: ‹sum (toR ∘ (fromR ∘ f)) F = toR (sum (fromR ∘ f) F)› if ‹finite F›
for F
    by (simp add: o-def additive.sum[OF ‹additive fromR›, symmetric] toR-fromR)
  have 2: ‹sum (fromR ∘ f) F ∈ T› if ‹finite F› for F
    by (simp add: o-def additive.sum[OF ‹additive fromR›, symmetric] T-def)
  have 3: ‹(toR ⟶ toR x) (at x within T)› for x
  proof (cases ‹x ∈ T›)
    case True
    have ‹dist (toR y) (toR x) < e› if ‹y∈T› ‹e>0› ‹dist y x < e * norm c› for
e y
    proof −
      obtain x' y' where x: ‹x = fromR x'› and y: ‹y = fromR y'›
        using T-def True ‹y ∈ T› by blast
      have ‹dist (toR y) (toR x) = dist (fromR (toR y)) (fromR (toR x)) / norm
c›
        by (auto simp: dist-real-def fromR-def ‹c ≠ 0›)
      also have ‹... = dist y x / norm c›
        using ‹x∈T› ‹y∈T› by (simp add: fromR-toR)
      also have ‹... < e›
        using ‹dist y x < e * norm c›
        by (simp add: divide-less-eq that(2))
      finally show ?thesis
        by (simp add: x y toR-fromR)
    qed
    then show ?thesis
      apply (auto simp: tendsto-iff at-within-def eventually-inf-principal eventu-
ally-nhds-metric)
      by (metis assms(1) div-0 divide-less-eq zero-less-norm-iff)
  next
    case False
    have ‹T = span {c}›
      by (simp add: T-def fromR-def span-singleton)
    then have ‹closed T›
      by simp
    with False have ‹x ∉ closure T›
      by simp
    then have ‹(at x within T) = bot›
      by (rule not-in-closure-trivial-limitI)
    then show ?thesis
      by simp
  qed
  have 4: ‹(fromR ∘ f) summable-on A›
    by (simp add: assms(2) fromR-def summable-on-cong)

  have ‹(toR o (fromR o f)) summable-on A›
```

using *1 2 3 4*
by (*rule summable-on-comm-additive-general*[**where** *T*=*T*])
with *toR-fromR*
show *?thesis*
by (*auto simp*: *o-def*)
qed

lemma *infsum-scaleR-left*:
— This result has nothing to do with the bounded operator library but it uses *finite-span-closed* so it is proven here.
It is a strengthening of *infsum-scaleR-left*.
fixes *c* :: ‹*'a* :: *real-normed-vector*›
shows *infsum* (λ*x*. *f x* ∗*R* *c*) *A* = *infsum f A* ∗*R* *c*
proof (*cases* ‹*f summable-on A*›)
case *True*
then show *?thesis*
apply (*subst asm-rl*[*of* ‹(λ*x*. *f x* ∗*R* *c*) = (λ*y*. *y* ∗*R* *c*) *o f*›], *simp add*: *o-def*)
apply (*rule infsum-comm-additive*)
using *True* **by** (*auto simp add*: *scaleR-left.additive-axioms*)
next
case *False*
then have ‹¬ (λ*x*. *f x* ∗*R* *c*) *summable-on A*› **if** ‹*c* ≠ *0*›
using *summable-on-scaleR-left-converse*[**where** *A*=*A* **and** *f*=*f* **and** *c*=*c*]
using *that* **by** *auto*
with *False* **show** *?thesis*
apply (*cases* ‹*c* = *0*›)
by (*auto simp add*: *infsum-not-exists*)
qed

lemma *infsum-of-real*:
shows ‹(∑∞*x*∈*A*. *of-real* (*f x*) :: *'b*::{*real-normed-vector*, *real-algebra-1*}) = *of-real* (∑∞*x*∈*A*. *f x*)›
— This result has nothing to do with the bounded operator library but it uses *finite-span-closed* so it is proven here.
unfolding *of-real-def*
by (*rule infsum-scaleR-left*)

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

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

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

7.5 Closed subspaces

lemma *csubspace-INF*[*simp*]: (⋀*x*. *x* ∈ *A* ⟹ *csubspace x*) ⟹ *csubspace* (⋂ *A*)
by (*simp add*: *complex-vector.subspace-Inter*)

locale *closed-csubspace* =
fixes *A*::('*a*::{*complex-vector*,*topological-space*}) *set*
assumes *subspace*: *csubspace A*
assumes *closed*: *closed A*

declare *closed-csubspace.subspace*[*simp*]

lemma *closure-is-csubspace*[*simp*]:
fixes *A*::('*a*::*complex-normed-vector*) *set*
assumes ‹*csubspace A*›
shows ‹*csubspace* (*closure A*)›
proof−
have *x* ∈ *closure A* ⟹ *y* ∈ *closure A* ⟹ *x*+*y* ∈ *closure A* **for** *x y*
proof−
assume ‹*x*∈(*closure A*)›
then obtain *xx* **where** ‹∀ *n*::*nat*. *xx n* ∈ *A*› **and** ‹*xx* ⟶ *x*›
using *closure-sequential* **by** *blast*
assume ‹*y*∈(*closure A*)›
then obtain *yy* **where** ‹∀ *n*::*nat*. *yy n* ∈ *A*› **and** ‹*yy* ⟶ *y*›
using *closure-sequential* **by** *blast*
have ‹∀ *n*::*nat*. (*xx n*) + (*yy n*) ∈ *A*›
using ‹∀*n*. *xx n* ∈ *A*› ‹∀*n*. *yy n* ∈ *A*› *assms complex-vector.subspace-def*
by (*simp add*: *complex-vector.subspace-def*)
hence ‹(λ *n*. (*xx n*) + (*yy n*)) ⟶ *x* + *y*› **using** ‹*xx* ⟶ *x*› ‹*yy* ⟶ *y*›
by (*simp add*: *tendsto-add*)
thus *?thesis* **using** ‹∀ *n*::*nat*. (*xx n*) + (*yy n*) ∈ *A*›
by (*meson closure-sequential*)
qed

moreover have $x{\in}(closure\ A) \Longrightarrow c *_C x \in (closure\ A)$ **for** $x\ c$
proof−
assume ‹$x{\in}(closure\ A)$›
then obtain xx **where** ‹$\forall\ n{::}nat.\ xx\ n \in A$› **and** ‹$xx \longrightarrow x$›
using *closure-sequential* **by** *blast*
have ‹$\forall\ n{::}nat.\ c *_C (xx\ n) \in A$›
using ‹$\forall n.\ xx\ n \in A$› *assms complex-vector.subspace-def*
by (*simp add*: *complex-vector.subspace-def*)
have ‹*isCont* ($\lambda\ t.\ c *_C t$) x›
using *bounded-clinear.bounded-linear bounded-clinear-scaleC-right linear-continuous-at*
by *auto*
hence ‹$(\lambda\ n.\ c *_C (xx\ n)) \longrightarrow c *_C x$› **using** ‹$xx \longrightarrow x$›
by (*simp add*: *isCont-tendsto-compose*)
thus *?thesis* **using** ‹$\forall\ n{::}nat.\ c *_C (xx\ n) \in A$›
by (*meson closure-sequential*)
qed
moreover have $0 \in (closure\ A)$
using *assms closure-subset complex-vector.subspace-def*
by (*metis in-mono*)
ultimately show *?thesis*
by (*simp add*: *complex-vector.subspaceI*)
qed

lemma *csubspace-set-plus*:
assumes ‹*csubspace* A› **and** ‹*csubspace* B›
shows ‹*csubspace* $(A + B)$›
proof −
define C **where** ‹$C = \{\psi{+}\varphi|\ \psi\ \varphi.\ \psi{\in}A \wedge \varphi{\in}B\}$›
have $x{\in}C \Longrightarrow y{\in}C \Longrightarrow x{+}y{\in}C$ **for** $x\ y$
using *C-def assms*(*1*) *assms*(*2*) *complex-vector.subspace-add complex-vector.subspace-sums*
by *blast*
moreover have $c *_C x \in C$ **if** ‹$x{\in}C$› **for** $x\ c$
proof −
have *csubspace C*
by (*simp add*: *C-def assms*(*1*) *assms*(*2*) *complex-vector.subspace-sums*)
then show *?thesis*
using *that* **by** (*simp add*: *complex-vector.subspace-def*)
qed
moreover have $0 \in C$
using ‹$C = \{\psi + \varphi\ |\psi\ \varphi.\ \psi \in A \wedge \varphi \in B\}$› *add.inverse-neutral add-uminus-conv-diff*
assms(*1*) *assms*(*2*) *diff-0 mem-Collect-eq*
add.right-inverse
by (*metis* (*mono-tags*, *lifting*) *complex-vector.subspace-0*)
ultimately show *?thesis*
unfolding *C-def complex-vector.subspace-def*
by (*smt mem-Collect-eq set-plus-elim set-plus-intro*)
qed

lemma *closed-csubspace-0*[*simp*]:

closed-csubspace ({*0*} :: ('*a*::{*complex-vector*,*t1-space*}) *set*)
proof−
have ‹*csubspace* {*0*}›
using *add.right-neutral complex-vector.subspace-def scaleC-right.zero*
by *blast*
moreover have *closed* ({*0*} :: '*a set*)
by *simp*
ultimately show *?thesis*
by (*simp add*: *closed-csubspace-def*)
qed

lemma *closed-csubspace-UNIV*[*simp*]: *closed-csubspace* (*UNIV*::('*a*::{*complex-vector*,*topological-space*}) *set*)
proof−
have ‹*csubspace UNIV*›
by *simp*
moreover have ‹*closed UNIV*›
by *simp*
ultimately show *?thesis*
unfolding *closed-csubspace-def* **by** *auto*
qed

lemma *closed-csubspace-inter*[*simp*]:
assumes *closed-csubspace A* **and** *closed-csubspace B*
shows *closed-csubspace* (*A*∩*B*)
proof−
obtain *C* **where** ‹*C* = *A* ∩ *B*› **by** *blast*
have ‹*csubspace C*›
proof−
have *x*∈*C* ⟹ *y*∈*C* ⟹ *x*+*y*∈*C* **for** *x y*
by (*metis IntD1 IntD2 IntI* ‹*C* = *A* ∩ *B*› *assms*(*1*) *assms*(*2*) *complex-vector.subspace-def closed-csubspace-def*)
moreover have *x*∈*C* ⟹ *c* *$_C$ *x* ∈ *C* **for** *x c*
by (*metis IntD1 IntD2 IntI* ‹*C* = *A* ∩ *B*› *assms*(*1*) *assms*(*2*) *complex-vector.subspace-def closed-csubspace-def*)
moreover have *0* ∈ *C*
using ‹*C* = *A* ∩ *B*› *assms*(*1*) *assms*(*2*) *complex-vector.subspace-def closed-csubspace-def* **by** *fastforce*
ultimately show *?thesis*
by (*simp add*: *complex-vector.subspace-def*)
qed
moreover have ‹*closed C*›
using ‹*C* = *A* ∩ *B*›
by (*simp add*: *assms*(*1*) *assms*(*2*) *closed-Int closed-csubspace.closed*)
ultimately show *?thesis*
using ‹*C* = *A* ∩ *B*›
by (*simp add*: *closed-csubspace-def*)
qed

lemma *closed-csubspace-INF*[*simp*]:
assumes *a1*: $\forall A{\in}\mathcal{A}$. *closed-csubspace A*
shows *closed-csubspace* $(\bigcap \mathcal{A})$
proof−
have ‹*csubspace* $(\bigcap \mathcal{A})$›
by (*simp add*: *assms closed-csubspace.subspace complex-vector.subspace-Inter*)
moreover have ‹*closed* $(\bigcap \mathcal{A})$›
by (*simp add*: *assms closed-Inter closed-csubspace.closed*)
ultimately show *?thesis*
by (*simp add*: *closed-csubspace.intro*)
qed

typedef (**overloaded**) ('*a*::{*complex-vector*,*topological-space*})
ccsubspace = ‹{*S*::'*a set*. *closed-csubspace S*}›
morphisms *space-as-set Abs-ccsubspace*
using *Complex-Vector-Spaces.closed-csubspace-UNIV* **by** *blast*

setup-lifting *type-definition-ccsubspace*

lemma *csubspace-space-as-set*[*simp*]: ‹*csubspace* (*space-as-set S*)›
by (*metis closed-csubspace-def mem-Collect-eq space-as-set*)

lemma *closed-space-as-set*[*simp*]: ‹*closed* (*space-as-set S*)›
apply *transfer* **by** (*simp add*: *closed-csubspace.closed*)

lemma *zero-space-as-set*[*simp*]: ‹*0* ∈ *space-as-set A*›
by (*simp add*: *complex-vector.subspace-0*)

instantiation *ccsubspace* :: (*complex-normed-vector*) *scaleC* **begin**
lift-definition *scaleC-ccsubspace* :: *complex* ⇒ '*a ccsubspace* ⇒ '*a ccsubspace* **is**
$\lambda c\ S.\ (*_C)\ c\ `\ S$
proof
show *csubspace* $((*_C)\ c\ `\ S)$ **if** *closed-csubspace S* **for** *c* :: *complex* **and** *S* :: '*a set*
using *that*
by (*simp add*: *complex-vector.linear-subspace-image*)
show *closed* $((*_C)\ c\ `\ S)$ **if** *closed-csubspace S* **for** *c* :: *complex* **and** *S* :: '*a set*
using *that*
by (*simp add*: *closed-scaleC closed-csubspace.closed*)
qed

lift-definition *scaleR-ccsubspace* :: *real* ⇒ '*a ccsubspace* ⇒ '*a ccsubspace* **is**
$\lambda c\ S.\ (*_R)\ c\ `\ S$
proof
show *csubspace* $((*_R)\ r\ `\ S)$
if *closed-csubspace S*
for *r* :: *real*
and *S* :: '*a set*

```
  using that  using bounded-clinear-def bounded-clinear-scaleC-right scaleR-scaleC
   by (simp add: scaleR-scaleC complex-vector.linear-subspace-image)
 show closed ((*R) r ‘ S)
   if closed-csubspace S
   for r :: real
     and S :: 'a set
   using that
   by (simp add: closed-scaling closed-csubspace.closed)
qed

instance
proof
 show ((*R) r::'a ccsubspace ⇒ -) = (*C) (complex-of-real r) for r :: real
   by (simp add: scaleR-scaleC scaleC-ccsubspace-def scaleR-ccsubspace-def)
qed
end

instantiation ccsubspace :: ({complex-vector,t1-space}) bot begin
lift-definition bot-ccsubspace :: ‹'a ccsubspace› is ‹{0}›
 by simp
instance..
end

lemma zero-cblinfun-image[simp]: 0 *C S = bot for S :: - ccsubspace
proof transfer
 have (0::'b) ∈ (λx. 0) ‘ S
   if closed-csubspace S
   for S::'b set
   using that unfolding closed-csubspace-def
  by (simp add: complex-vector.linear-subspace-image complex-vector.module-hom-zero
      complex-vector.subspace-0)
 thus (*C) 0 ‘ S = {0::'b}
   if closed-csubspace (S::'b set)
   for S :: 'b set
   using that
   by (auto intro !: exI [of - 0])
qed

lemma csubspace-scaleC-invariant:
 fixes a S
 assumes ‹a ≠ 0› and ‹csubspace S›
 shows ‹(*C) a ‘ S = S›
proof−
 have  ‹x ∈ (*C) a ‘ S ⟹ x ∈ S›
   for x
   using assms(2) complex-vector.subspace-scale by blast
 moreover have  ‹x ∈ S ⟹ x ∈ (*C) a ‘ S›
   for x
 proof −
```

assume $x \in S$
hence $\exists\, c\ aa.\ (c\ /\ a) *_C aa \in S \wedge c *_C aa = x$
using *assms*(2) *complex-vector.subspace-def scaleC-one* **by** *metis*
hence $\exists\, aa.\ aa \in S \wedge a *_C aa = x$
using *assms*(1) **by** *auto*
thus *?thesis*
by (*meson image-iff*)
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *ccsubspace-scaleC-invariant*[*simp*]: $a \neq 0 \Longrightarrow a *_C S = S$ **for** S :: - *cc-subspace*
apply *transfer*
by (*simp add*: *closed-csubspace.subspace csubspace-scaleC-invariant*)

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *top*
begin
lift-definition *top-ccsubspace* :: ‹'*a ccsubspace*› **is** ‹*UNIV*›
by *simp*

instance ..
end

lemma *space-as-set-bot*[*simp*]: ‹*space-as-set bot* = {*0*}›
by (*rule bot-ccsubspace.rep-eq*)

lemma *ccsubspace-top-not-bot*[*simp*]:
(*top*::'*a*::{*complex-vector*,*t1-space*,*not-singleton*} *ccsubspace*) ≠ *bot*

by (*metis UNIV-not-singleton bot-ccsubspace.rep-eq top-ccsubspace.rep-eq*)

lemma *ccsubspace-bot-not-top*[*simp*]:
(*bot*::'*a*::{*complex-vector*,*t1-space*,*not-singleton*} *ccsubspace*) ≠ *top*
using *ccsubspace-top-not-bot* **by** *metis*

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *Inf*
begin
lift-definition *Inf-ccsubspace*::‹'*a ccsubspace set* ⇒ '*a ccsubspace*›
is ‹λ *S*. ⋂ *S*›
proof
fix *S* :: '*a set set*
assume *closed*: *closed-csubspace x* **if** ‹*x* ∈ *S*› **for** *x*
show *csubspace* (⋂ *S*::'*a set*)
by (*simp add*: *closed closed-csubspace.subspace*)
show *closed* (⋂ *S*::'*a set*)
by (*simp add*: *closed closed-csubspace.closed*)

qed

instance ..
end

lift-definition *ccspan* :: *'a::complex-normed-vector set* ⇒ *'a ccsubspace*
is *λG. closure* (*cspan G*)
proof (*rule closed-csubspace.intro*)
fix *S* :: *'a set*
show *csubspace* (*closure* (*cspan S*))
by (*simp add*: *closure-is-csubspace*)
show *closed* (*closure* (*cspan S*))
by *simp*
qed

lemma *ccspan-superset*:
‹*A* ⊆ *space-as-set* (*ccspan A*)›
for *A* :: ‹*'a::complex-normed-vector set*›
apply *transfer*
by (*meson closure-subset complex-vector.span-superset subset-trans*)

lemma *ccspan-superset'*: ‹*x* ∈ *X* ⟹ *x* ∈ *space-as-set* (*ccspan X*)›
using *ccspan-superset* **by** *auto*

lemma *ccspan-canonical-basis*[*simp*]: *ccspan* (*set canonical-basis*) = *top*
using *ccspan.rep-eq space-as-set-inject top-ccsubspace.rep-eq*
closure-UNIV is-generator-set
by *metis*

lemma *ccspan-Inf-def*: ‹*ccspan A* = *Inf* {*S*. *A* ⊆ *space-as-set S*}›
for *A*::‹(*'a::cbanach*) *set*›
proof −
have ‹*x* ∈ *space-as-set* (*ccspan A*)
⟹ *x* ∈ *space-as-set* (*Inf* {*S*. *A* ⊆ *space-as-set S*})›
for *x*::*'a*
proof−
assume ‹*x* ∈ *space-as-set* (*ccspan A*)›
hence *x* ∈ *closure* (*cspan A*)
by (*simp add*: *ccspan.rep-eq*)
hence ‹*x* ∈ *closure* (*complex-vector.span A*)›
unfolding *ccspan-def*
by *simp*
hence ‹∃ *y*::*nat* ⇒ *'a*. (∀ *n*. *y n* ∈ (*complex-vector.span A*)) ∧ *y* ⟶ *x*›
by (*simp add*: *closure-sequential*)
then obtain *y* **where** ‹∀ *n*. *y n* ∈ (*complex-vector.span A*)› **and** ‹*y* ⟶ *x*›
by *blast*
have ‹*y n* ∈ ⋂ {*S*. (*complex-vector.span A*) ⊆ *S* ∧ *closed-csubspace S*}›
for *n*
using ‹∀ *n*. *y n* ∈ (*complex-vector.span A*)›

by *auto*

have ‹*closed-csubspace S* ⟹ *closed S*›
for *S*::‹*'a set*›
by (*simp add*: *closed-csubspace.closed*)
hence ‹*closed* (⋂ {*S*. (*complex-vector.span A*) ⊆ *S* ∧ *closed-csubspace S*})›
by *simp*
hence ‹*x* ∈ ⋂ {*S*. (*complex-vector.span A*) ⊆ *S* ∧ *closed-csubspace S*}› **using** ‹*y* ⟶ *x*›
using ‹⋀*n*. *y n* ∈ ⋂ {*S*. *complex-vector.span A* ⊆ *S* ∧ *closed-csubspace S*}› *closed-sequentially*
by *blast*
moreover have ‹{*S*. *A* ⊆ *S* ∧ *closed-csubspace S*} ⊆ {*S*. (*complex-vector.span A*) ⊆ *S* ∧ *closed-csubspace S*}›
using *Collect-mono-iff*
by (*simp add*: *Collect-mono-iff closed-csubspace.subspace complex-vector.span-minimal*)
ultimately have ‹*x* ∈ ⋂ {*S*. *A* ⊆ *S* ∧ *closed-csubspace S*}›
by *blast*
moreover have (*x*::*'a*) ∈ ⋂ {*x*. *A* ⊆ *x* ∧ *closed-csubspace x*}
if (*x*::*'a*) ∈ ⋂ {*S*. *A* ⊆ *S* ∧ *closed-csubspace S*}
for *x* :: *'a*
and *A* :: *'a set*
using *that*
by *simp*
ultimately show ‹*x* ∈ *space-as-set* (*Inf* {*S*. *A* ⊆ *space-as-set S*})›
apply *transfer*.
qed
moreover have ‹*x* ∈ *space-as-set* (*Inf* {*S*. *A* ⊆ *space-as-set S*})
⟹ *x* ∈ *space-as-set* (*ccspan A*)›
for *x*::*'a*
proof−
assume ‹*x* ∈ *space-as-set* (*Inf* {*S*. *A* ⊆ *space-as-set S*})›
hence ‹*x* ∈ ⋂ {*S*. *A* ⊆ *S* ∧ *closed-csubspace S*}›
apply *transfer*
by *blast*
moreover have ‹{*S*. (*complex-vector.span A*) ⊆ *S* ∧ *closed-csubspace S*} ⊆ {*S*. *A* ⊆ *S* ∧ *closed-csubspace S*}›
using *Collect-mono-iff complex-vector.span-superset* **by** *fastforce*
ultimately have ‹*x* ∈ ⋂ {*S*. (*complex-vector.span A*) ⊆ *S* ∧ *closed-csubspace S*}›
by *blast*
thus ‹*x* ∈ *space-as-set* (*ccspan A*)›
by (*metis* (*no-types*, *lifting*) *Inter-iff space-as-set closure-subset mem-Collect-eq ccspan.rep-eq*)
qed
ultimately have ‹*space-as-set* (*ccspan A*) = *space-as-set* (*Inf* {*S*. *A* ⊆ *space-as-set S*})›
by *blast*
thus *?thesis*

using *space-as-set-inject* **by** *auto*
qed

lemma *cspan-singleton-scaleC*[*simp*]: (*a*::*complex*)≠*0* ⟹ *cspan* { *a* ∗$_C$ *ψ* } = *cspan* {*ψ*}
for *ψ*::′*a*::*complex-vector*
by (*smt complex-vector.dependent-single complex-vector.independent-insert complex-vector.scale-eq-0-iff complex-vector.span-base complex-vector.span-redundant complex-vector.span-scale doubleton-eq-iff insert-absorb insert-absorb2 insert-commute singletonI*)

lemma *closure-is-closed-csubspace*[*simp*]:
fixes *S*::‹′*a*::*complex-normed-vector set›*
assumes ‹*csubspace S*›
shows ‹*closed-csubspace* (*closure S*)›
using *assms closed-csubspace.intro closure-is-csubspace* **by** *blast*

lemma *ccspan-singleton-scaleC*[*simp*]: (*a*::*complex*)≠*0* ⟹ *ccspan* {*a* ∗$_C$ *ψ*} = *ccspan* {*ψ*}
apply *transfer* **by** *simp*

lemma *clinear-continuous-at*:
assumes ‹*bounded-clinear f*›
shows ‹*isCont f x*›
by (*simp add*: *assms bounded-clinear.bounded-linear linear-continuous-at*)

lemma *clinear-continuous-within*:
assumes ‹*bounded-clinear f*›
shows ‹*continuous* (*at x within s*) *f*›
by (*simp add*: *assms bounded-clinear.bounded-linear linear-continuous-within*)

lemma *antilinear-continuous-at*:
assumes ‹*bounded-antilinear f*›
shows ‹*isCont f x*›
by (*simp add*: *assms bounded-antilinear.bounded-linear linear-continuous-at*)

lemma *antilinear-continuous-within*:
assumes ‹*bounded-antilinear f*›
shows ‹*continuous* (*at x within s*) *f*›
by (*simp add*: *assms bounded-antilinear.bounded-linear linear-continuous-within*)

lemma *bounded-clinear-eq-on-closure*:
fixes *A B* :: ′*a*::*complex-normed-vector* ⇒ ′*b*::*complex-normed-vector*
assumes ‹*bounded-clinear A*› **and** ‹*bounded-clinear B*› **and**
eq: ‹⋀*x*. *x* ∈ *G* ⟹ *A x* = *B x*› **and** *t*: ‹*t* ∈ *closure* (*cspan G*)›
shows ‹*A t* = *B t*›
proof −
have *eq′*: ‹*A t* = *B t*› **if** ‹*t* ∈ *cspan G*› **for** *t*

using - - *that eq* **apply** (*rule complex-vector.linear-eq-on*)
by (*auto simp*: *assms bounded-clinear.clinear*)
have ‹*A t − B t = 0*›
using - - *t* **apply** (*rule continuous-constant-on-closure*)
by (*auto simp add*: *eq′ assms*(*1*) *assms*(*2*) *clinear-continuous-at continuous-at-imp-continuous-on*)
then show *?thesis*
by *auto*
qed

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *order*
begin
lift-definition *less-eq-ccsubspace* :: ‹*′a ccsubspace ⇒ ′a ccsubspace ⇒ bool*›
is ‹(⊆)›**.**
declare *less-eq-ccsubspace-def*[*code del*]
lift-definition *less-ccsubspace* :: ‹*′a ccsubspace ⇒ ′a ccsubspace ⇒ bool*›
is ‹(⊂)›**.**
declare *less-ccsubspace-def*[*code del*]
instance
proof
fix *x y z* :: *′a ccsubspace*
show (*x < y*) = (*x ≤ y ∧ ¬ y ≤ x*)
by (*simp add*: *less-eq-ccsubspace.rep-eq less-le-not-le less-ccsubspace.rep-eq*)
show *x ≤ x*
by (*simp add*: *less-eq-ccsubspace.rep-eq*)
show *x ≤ z* **if** *x ≤ y* **and** *y ≤ z*
using *that less-eq-ccsubspace.rep-eq* **by** *auto*
show *x = y* **if** *x ≤ y* **and** *y ≤ x*
using *that* **by** (*simp add*: *space-as-set-inject less-eq-ccsubspace.rep-eq*)
qed
end

lemma *ccspan-leqI*:
assumes ‹*M ⊆ space-as-set S*›
shows ‹*ccspan M ≤ S*›
using *assms* **apply** *transfer*
by (*simp add*: *closed-csubspace.closed closure-minimal complex-vector.span-minimal*)

lemma *ccspan-mono*:
assumes ‹*A ⊆ B*›
shows ‹*ccspan A ≤ ccspan B*›
apply (*transfer fixing*: *A B*)
by (*simp add*: *assms closure-mono complex-vector.span-mono*)

lemma *ccsubspace-leI*:
assumes *t1*: *space-as-set A ⊆ space-as-set B*
shows *A ≤ B*
using *t1* **apply** *transfer* **by** −

lemma *ccspan-of-empty*[*simp*]: *ccspan* {} = *bot*
proof *transfer*
show *closure* (*cspan* {}) = {*0*::*'a*}
by *simp*
qed

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *inf* **begin**
lift-definition *inf-ccsubspace* :: *'a ccsubspace* ⇒ *'a ccsubspace* ⇒ *'a ccsubspace*
is (∩) **by** *simp*
instance .. end

lemma *space-as-set-inf*[*simp*]: *space-as-set* (*A* ⊓ *B*) = *space-as-set A* ∩ *space-as-set B*
by (*rule inf-ccsubspace.rep-eq*)

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *order-top* **begin**
instance
proof
show *a* ≤ ⊤
for *a* :: *'a ccsubspace*
apply *transfer*
by *simp*
qed
end

instantiation *ccsubspace* :: ({*complex-vector*,*t1-space*}) *order-bot* **begin**
instance
proof
show (⊥::*'a ccsubspace*) ≤ *a*
for *a* :: *'a ccsubspace*
apply *transfer*
apply *auto*
using *closed-csubspace.subspace complex-vector.subspace-0* **by** *blast*
qed
end

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *semilattice-inf* **begin**
instance
proof
fix *x y z* :: ‹*'a ccsubspace*›
show *x* ⊓ *y* ≤ *x*
apply *transfer* **by** *simp*
show *x* ⊓ *y* ≤ *y*
apply *transfer* **by** *simp*
show *x* ≤ *y* ⊓ *z* **if** *x* ≤ *y* **and** *x* ≤ *z*

using *that* **apply** *transfer* **by** *simp*
qed
end

instantiation *ccsubspace* :: ({*complex-vector*,*t1-space*}) *zero* **begin**
definition *zero-ccsubspace* :: *'a ccsubspace* **where** [*simp*]: *zero-ccsubspace* = *bot*
lemma *zero-ccsubspace-transfer*[*transfer-rule*]: ‹*pcr-ccsubspace* (=) {*0*} *0*›
unfolding *zero-ccsubspace-def* **by** *transfer-prover*
instance ..
end

lemma *ccspan-0*[*simp*]: ‹*ccspan* {*0*} = *0*›
apply *transfer*
by *simp*

definition ‹*rel-ccsubspace R x y* = *rel-set R* (*space-as-set x*) (*space-as-set y*)›

lemma *left-unique-rel-ccsubspace*[*transfer-rule*]: ‹*left-unique* (*rel-ccsubspace R*)› **if** ‹*left-unique R*›
proof (*rule left-uniqueI*)
fix *S T* :: ‹*'a ccsubspace*› **and** *U*
assume *assms*: ‹*rel-ccsubspace R S U*› ‹*rel-ccsubspace R T U*›
have ‹*space-as-set S* = *space-as-set T*›
apply (*rule left-uniqueD*)
using *that* **apply** (*rule left-unique-rel-set*)
using *assms* **unfolding** *rel-ccsubspace-def* **by** *auto*
then show ‹*S* = *T*›
by (*simp add*: *space-as-set-inject*)
qed

lemma *right-unique-rel-ccsubspace*[*transfer-rule*]: ‹*right-unique* (*rel-ccsubspace R*)›
if ‹*right-unique R*›
by (*metis rel-ccsubspace-def right-unique-def right-unique-rel-set space-as-set-inject that*)

lemma *bi-unique-rel-ccsubspace*[*transfer-rule*]: ‹*bi-unique* (*rel-ccsubspace R*)› **if** ‹*bi-unique R*›
by (*metis* (*no-types*, *lifting*) *bi-unique-def bi-unique-rel-set rel-ccsubspace-def space-as-set-inject that*)

lemma *converse-rel-ccsubspace*: ‹*conversep* (*rel-ccsubspace R*) = *rel-ccsubspace* (*conversep R*)›
by (*auto simp*: *rel-ccsubspace-def*[*abs-def*])

lemma *space-as-set-top*[*simp*]: ‹*space-as-set top* = *UNIV*›
by (*rule top-ccsubspace.rep-eq*)

lemma *ccsubspace-eqI*:
assumes ‹$\bigwedge x.\ x \in$ *space-as-set* $S \longleftrightarrow x \in$ *space-as-set* T›
shows ‹$S = T$›
by (*metis Abs-ccsubspace-cases Abs-ccsubspace-inverse antisym assms subsetI*)

lemma *ccspan-remove-0*: ‹*ccspan* $(A - \{0\}) =$ *ccspan* A›
apply *transfer*
by *auto*

lemma *sgn-in-spaceD*: ‹$\psi \in$ *space-as-set* A› **if** ‹*sgn* $\psi \in$ *space-as-set* A› **and** ‹$\psi \neq 0$›
for ψ :: ‹- :: *complex-normed-vector*›
using *that*
apply (*transfer fixing*: ψ)
by (*metis closed-csubspace.subspace complex-vector.subspace-scale divideC-field-simps*(*1*) *scaleR-eq-0-iff scaleR-scaleC sgn-div-norm sgn-zero-iff*)

lemma *sgn-in-spaceI*: ‹*sgn* $\psi \in$ *space-as-set* A› **if** ‹$\psi \in$ *space-as-set* A›
for ψ :: ‹- :: *complex-normed-vector*›
using *that* **by** (*auto simp*: *sgn-div-norm scaleR-scaleC complex-vector.subspace-scale*)

lemma *ccsubspace-leI-unit*:
fixes A B :: ‹- :: *complex-normed-vector ccsubspace*›
assumes $\bigwedge \psi.$ *norm* $\psi = 1 \Longrightarrow \psi \in$ *space-as-set* $A \Longrightarrow \psi \in$ *space-as-set* B
shows $A \leq B$
proof (*rule ccsubspace-leI*, *rule subsetI*)
fix ψ **assume** ψA: ‹$\psi \in$ *space-as-set* A›
show ‹$\psi \in$ *space-as-set* B›
apply (*cases* ‹$\psi = 0$›)
apply *simp*
using *assms*[*of* ‹*sgn* ψ›] ψA *sgn-in-spaceD sgn-in-spaceI*
by (*auto simp*: *norm-sgn*)
qed

lemma *kernel-is-closed-csubspace*[*simp*]:
assumes *a1*: *bounded-clinear* f
shows *closed-csubspace* $(f -` \{0\})$
proof−
have ‹*csubspace* $(f -` \{0\})$›
using *assms bounded-clinear.clinear complex-vector.linear-subspace-vimage complex-vector.subspace-single-0* **by** *blast*
have $L \in \{x.\ f\ x = 0\}$
if $r \longrightarrow L$ **and** $\forall\ n.\ r\ n \in \{x.\ f\ x = 0\}$
for r **and** L
proof−
have *d1*: ‹$\forall\ n.\ f\ (r\ n) = 0$›
using *that*(*2*) **by** *auto*
have ‹$(\lambda\ n.\ f\ (r\ n)) \longrightarrow f\ L$›
using *assms clinear-continuous-at continuous-within-tendsto-compose′ that*(*1*)

```
    by fastforce
  hence ‹(λ n. 0) ⟶ f L›
    using d1 by simp
  hence ‹f L = 0›
    using limI by fastforce
  thus ?thesis by blast
 qed
 then have s3: ‹closed (f -` {0})›
  using closed-sequential-limits by force
 with ‹csubspace (f -` {0})›
 show ?thesis
  using closed-csubspace.intro by blast
qed

lemma ccspan-closure[simp]: ‹ccspan (closure X) = ccspan X›
 by (simp add: basic-trans-rules(24) ccspan.rep-eq ccspan-leqI ccspan-mono clo-
sure-mono closure-subset complex-vector.span-superset)

lemma ccspan-finite: ‹space-as-set (ccspan X) = cspan X› if ‹finite X›
 by (simp add: ccspan.rep-eq that)

lemma ccspan-UNIV[simp]: ‹ccspan UNIV = ⊤›
 by (simp add: ccspan.abs-eq top-ccsubspace-def)

lemma infsum-in-closed-csubspaceI:
 assumes ‹⋀x. x∈X ⟹ f x ∈ A›
 assumes ‹closed-csubspace A›
 shows ‹infsum f X ∈ A›
proof (cases ‹f summable-on X›)
 case True
 then have lim: ‹(sum f ⟶ infsum f X) (finite-subsets-at-top X)›
  by (simp add: infsum-tendsto)
 have sumA: ‹sum f F ∈ A› if ‹finite F› and ‹F ⊆ X› for F
  apply (rule complex-vector.subspace-sum)
  using that assms by auto
 from lim show ‹infsum f X ∈ A›
  apply (rule Lim-in-closed-set[rotated -1])
  using assms sumA by (auto intro!: closed-csubspace.closed eventually-finite-subsets-at-top-weakI)
next
 case False
 then show ?thesis
  using assms by (auto intro!: closed-csubspace.closed complex-vector.subspace-0
simp add: infsum-not-exists)
qed

lemma closed-csubspace-space-as-set[simp]: ‹closed-csubspace (space-as-set X)›
 using space-as-set by simp

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 *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

7.6 Closed sums

definition *closed-sum*:: ‹*'a*::{*semigroup-add*,*topological-space*} *set* ⇒ *'a set* ⇒ *'a set*› **where**
‹*closed-sum A B* = *closure* (*A* + *B*)›

notation *closed-sum* (**infixl** ‹+$_M$› *65*)

lemma *closed-sum-comm*: ‹*A* +$_M$ *B* = *B* +$_M$ *A*› **for** *A B* :: -::*ab-semigroup-add*
by (*simp add*: *add.commute closed-sum-def*)

lemma *closed-sum-left-subset*: ‹*0* ∈ *B* ⟹ *A* ⊆ *A* +$_M$ *B*› **for** *A B* :: -::*monoid-add*
by (*metis add.right-neutral closed-sum-def closure-subset in-mono set-plus-intro*

subsetI)

lemma *closed-sum-right-subset*: ‹$0 \in A \Longrightarrow B \subseteq A +_M B$› **for** A B :: -::*monoid-add*
by (*metis add.left-neutral closed-sum-def closure-subset set-plus-intro subset-iff*)

lemma *finite-cspan-closed-csubspace*:
assumes *finite* (S::'*a*::*complex-normed-vector set*)
shows *closed-csubspace* (*cspan S*)
by (*simp add*: *assms closed-csubspace.intro*)

lemma *closed-sum-is-sup*:
fixes A B C:: ‹('*a*::{*complex-vector*,*topological-space*}) *set*›
assumes ‹*closed-csubspace C*›
assumes ‹$A \subseteq C$› **and** ‹$B \subseteq C$›
shows ‹$(A +_M B) \subseteq C$›
proof −
have ‹$A + B \subseteq C$›
using *assms* **unfolding** *set-plus-def*
using *closed-csubspace.subspace complex-vector.subspace-add* **by** *blast*
then show ‹$(A +_M B) \subseteq C$›
unfolding *closed-sum-def*
using ‹*closed-csubspace C*›
by (*simp add*: *closed-csubspace.closed closure-minimal*)
qed

lemma *closed-subspace-closed-sum*:
fixes A B::('*a*::*complex-normed-vector*) *set*
assumes *a1*: ‹*csubspace A*› **and** *a2*: ‹*csubspace B*›
shows ‹*closed-csubspace* ($A +_M B$)›
using *a1 a2 closed-sum-def*
by (*metis closure-is-closed-csubspace csubspace-set-plus*)

lemma *closed-sum-assoc*:
fixes A B C::'*a*::*real-normed-vector set*
shows ‹$A +_M (B +_M C) = (A +_M B) +_M C$›
proof −
have ‹$A +$ *closure* $B \subseteq$ *closure* $(A + B)$› **for** A B :: '*a set*
by (*meson closure-subset closure-sum dual-order.trans order-refl set-plus-mono2*)
then have ‹$A +_M (B +_M C) =$ *closure* $(A + (B + C))$›
unfolding *closed-sum-def*
by (*meson antisym-conv closed-closure closure-minimal closure-mono closure-subset equalityD1 set-plus-mono2*)
moreover
have ‹*closure* $A + B \subseteq$ *closure* $(A + B)$› **for** A B :: '*a set*
by (*meson closure-subset closure-sum dual-order.trans order-refl set-plus-mono2*)
then have ‹$(A +_M B) +_M C =$ *closure* $((A + B) + C)$›
unfolding *closed-sum-def*
by (*meson closed-closure closure-minimal closure-mono closure-subset eq-iff*

set-plus-mono2)
ultimately show *?thesis*
by (*simp add*: *ab-semigroup-add-class.add-ac*(*1*))
qed

lemma *closed-sum-zero-left*[*simp*]:
fixes *A* :: ‹(*'a*::{*monoid-add*, *topological-space*}) *set›*
shows ‹{*0*} +$_M$ *A* = *closure A›*
unfolding *closed-sum-def*
by (*metis add.left-neutral set-zero*)

lemma *closed-sum-zero-right*[*simp*]:
fixes *A* :: ‹(*'a*::{*monoid-add*, *topological-space*}) *set›*
shows ‹*A* +$_M$ {*0*} = *closure A›*
unfolding *closed-sum-def*
by (*metis add.right-neutral set-zero*)

lemma *closed-sum-closure-right*[*simp*]:
fixes *A B* :: ‹*'a*::*real-normed-vector set›*
shows ‹*A* +$_M$ *closure B* = *A* +$_M$ *B›*
by (*metis closed-sum-assoc closed-sum-def closed-sum-zero-right closure-closure*)

lemma *closed-sum-closure-left*[*simp*]:
fixes *A B* :: ‹*'a*::*real-normed-vector set›*
shows ‹*closure A* +$_M$ *B* = *A* +$_M$ *B›*
by (*simp add*: *closed-sum-comm*)

lemma *closed-sum-mono-left*:
assumes ‹*A* ⊆ *B›*
shows ‹*A* +$_M$ *C* ⊆ *B* +$_M$ *C›*
by (*simp add*: *assms closed-sum-def closure-mono set-plus-mono2*)

lemma *closed-sum-mono-right*:
assumes ‹*A* ⊆ *B›*
shows ‹*C* +$_M$ *A* ⊆ *C* +$_M$ *B›*
by (*simp add*: *assms closed-sum-def closure-mono set-plus-mono2*)

instantiation *ccsubspace* :: (*complex-normed-vector*) *sup* **begin**
lift-definition *sup-ccsubspace* :: *'a ccsubspace* ⇒ *'a ccsubspace* ⇒ *'a ccsubspace*
— Note that *A* + *B* would not be a closed subspace, we need the closure. See, e.g., https://math.stackexchange.com/a/1786792/403528.
is λ*A B*::*'a set*. *A* +$_M$ *B*
by (*simp add*: *closed-subspace-closed-sum*)
instance ..
end

lemma *closed-sum-cspan*[*simp*]:
shows ‹*cspan X* +$_M$ *cspan Y* = *closure* (*cspan* (*X* ∪ *Y*))›

by (*smt* (*verit*, *best*) *Collect-cong closed-sum-def complex-vector.span-Un set-plus-def*)

lemma *closure-image-closed-sum*:
assumes ‹*bounded-linear U*›
shows ‹*closure* (*U* ‘ (*A* +$_M$ *B*)) = *closure* (*U* ‘ *A*) +$_M$ *closure* (*U* ‘ *B*)›
proof −
have ‹*closure* (*U* ‘ (*A* +$_M$ *B*)) = *closure* (*U* ‘ *closure* (*closure A* + *closure B*))›
unfolding *closed-sum-def*
by (*smt* (*verit*, *best*) *closed-closure closure-minimal closure-mono closure-subset closure-sum set-plus-mono2 subset-antisym*)
also have ‹... = *closure* (*U* ‘ (*closure A* + *closure B*))›
using *assms closure-bounded-linear-image-subset-eq* **by** *blast*
also have ‹... = *closure* (*U* ‘ *closure A* + *U* ‘ *closure B*)›
apply (*subst image-set-plus*)
by (*simp-all add*: *assms bounded-linear.linear*)
also have ‹... = *closure* (*closure* (*U* ‘ *A*) + *closure* (*U* ‘ *B*))›
by (*smt* (*verit*, *ccfv-SIG*) *assms closed-closure closure-bounded-linear-image-subset closure-bounded-linear-image-subset-eq closure-minimal closure-mono closure-sum dual-order.eq-iff set-plus-mono2*)
also have ‹... = *closure* (*U* ‘ *A*) +$_M$ *closure* (*U* ‘ *B*)›
using *closed-sum-def* **by** *blast*
finally show *?thesis*
by −
qed

lemma *ccspan-union*: *ccspan A* ⊔ *ccspan B* = *ccspan* (*A* ∪ *B*)
apply *transfer* **by** *simp*

instantiation *ccsubspace* :: (*complex-normed-vector*) *Sup*
begin
lift-definition *Sup-ccsubspace*::‹'*a ccsubspace set* ⇒ '*a ccsubspace*›
is ‹λ*S*. *closure* (*complex-vector.span* (*Union S*))›
proof
show *csubspace* (*closure* (*complex-vector.span* (⋃ *S*::'*a set*)))
if ⋀*x*::'*a set*. *x* ∈ *S* ⟹ *closed-csubspace x*
for *S* :: '*a set set*
using *that*
by (*simp add*: *closure-is-closed-csubspace*)
show *closed* (*closure* (*complex-vector.span* (⋃ *S*::'*a set*)))
if ⋀*x*. (*x*::'*a set*) ∈ *S* ⟹ *closed-csubspace x*
for *S* :: '*a set set*
using *that*
by *simp*
qed

instance..
end

instance *ccsubspace* :: ({*complex-normed-vector*}) *semilattice-sup*
proof
  fix *x y z* :: ‹*'a ccsubspace*›
  show ‹*x* ≤ *sup x y*›
    apply *transfer*
   by (*simp add*: *closed-csubspace-def closed-sum-left-subset complex-vector.subspace-0*)

  show *y* ≤ *sup x y*
    apply *transfer*
   by (*simp add*: *closed-csubspace-def closed-sum-right-subset complex-vector.subspace-0*)

  show *sup x y* ≤ *z* **if** *x* ≤ *z* **and** *y* ≤ *z*
    using *that* **apply** *transfer*
    apply (*rule closed-sum-is-sup*) **by** *auto*
qed

instance *ccsubspace* :: (*complex-normed-vector*) *complete-lattice*
proof
  show *Inf A* ≤ *x* **if** *x* ∈ *A*
    for *x* :: *'a ccsubspace* **and** *A* :: *'a ccsubspace set*
    using *that*
    apply *transfer*
    by *auto*

  have *b1*: *z* ⊆ ⋂ *A*
    if *Ball A closed-csubspace* **and**
      closed-csubspace z **and**
      (⋀*x*. *closed-csubspace x* ⟹ *x* ∈ *A* ⟹ *z* ⊆ *x*)
    for *z*::*'a set* **and** *A*
    using *that*
    by *auto*
  show *z* ≤ *Inf A*
    if ⋀*x*::*'a ccsubspace*. *x* ∈ *A* ⟹ *z* ≤ *x*
    for *A* :: *'a ccsubspace set*
      and *z* :: *'a ccsubspace*
    using *that*
    apply *transfer*
    using *b1* **by** *blast*

  show *x* ≤ *Sup A*
    if *x* ∈ *A*
    for *x* :: *'a ccsubspace*
      and *A* :: *'a ccsubspace set*
    using *that*
    apply *transfer*
   by (*meson Union-upper closure-subset complex-vector.span-superset dual-order.trans*)

show *Sup A* ≤ *z*
if ⋀*x*::*'a ccsubspace*. *x* ∈ *A* ⟹ *x* ≤ *z*
for *A* :: *'a ccsubspace set*
and *z* :: *'a ccsubspace*
using *that* **apply** *transfer*
proof −
fix *A* :: *'a set set* **and** *z* :: *'a set*
assume *A-closed*: *Ball A closed-csubspace*
assume *closed-csubspace z*
assume *in-z*: ⋀*x*. *closed-csubspace x* ⟹ *x* ∈ *A* ⟹ *x* ⊆ *z*
from *A-closed in-z*
have ‹*V* ⊆ *z*› **if** ‹*V* ∈ *A*› **for** *V*
by (*simp add*: *that*)
then have ‹⋃ *A* ⊆ *z*›
by (*simp add*: *Sup-le-iff*)
with ‹*closed-csubspace z*›
show *closure* (*cspan* (⋃ *A*)) ⊆ *z*
by (*simp add*: *closed-csubspace-def closure-minimal complex-vector.span-def subset-hull*)
qed

show *Inf* {} = (*top*::*'a ccsubspace*)
using ‹⋀*z A*. (⋀*x*. *x* ∈ *A* ⟹ *z* ≤ *x*) ⟹ *z* ≤ *Inf A*› *top.extremum-uniqueI* **by** *auto*

show *Sup* {} = (*bot*::*'a ccsubspace*)
using ‹⋀*z A*. (⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *z*) ⟹ *Sup A* ≤ *z*› *bot.extremum-uniqueI* **by** *auto*
qed

instantiation *ccsubspace* :: (*complex-normed-vector*) *comm-monoid-add* **begin**
definition *plus-ccsubspace* :: *'a ccsubspace* ⇒ - ⇒ -
where [*simp*]: *plus-ccsubspace* = *sup*
instance
proof
fix *a b c* :: ‹*'a ccsubspace*›
show *a* + *b* + *c* = *a* + (*b* + *c*)
using *sup.assoc* **by** *auto*
show *a* + *b* = *b* + *a*
by (*simp add*: *sup.commute*)
show *0* + *a* = *a*
by (*simp add*: *zero-ccsubspace-def*)
qed
end

lemma *SUP-ccspan*: ‹(*SUP x*∈*X*. *ccspan* (*S x*)) = *ccspan* (⋃*x*∈*X*. *S x*)›
proof (*rule SUP-eqI*)
show ‹*ccspan* (*S x*) ≤ *ccspan* (⋃*x*∈*X*. *S x*)› **if** ‹*x* ∈ *X*› **for** *x*
apply (*rule ccspan-mono*)

using *that* **by** *auto*
show ‹*ccspan* $(\bigcup x{\in}X.\ S\ x) \leq y$› **if** ‹$\bigwedge x.\ x \in X \Longrightarrow$ *ccspan* $(S\ x) \leq y$› **for** *y*
apply (*intro ccspan-leqI UN-least*)
using *that ccspan-superset* **by** (*auto simp*: *less-eq-ccsubspace.rep-eq*)
qed

lemma *ccsubspace-plus-sup*: $y \leq x \Longrightarrow z \leq x \Longrightarrow y + z \leq x$
for *x y z* :: *'a*::*complex-normed-vector ccsubspace*
unfolding *plus-ccsubspace-def* **by** *auto*

lemma *ccsubspace-Sup-empty*: *Sup* $\{\} = (0$::- *ccsubspace*$)$
unfolding *zero-ccsubspace-def* **by** *auto*

lemma *ccsubspace-add-right-incr*[*simp*]: $a \leq a + c$ **for** *a*::- *ccsubspace*
by (*simp add*: *add-increasing2*)

lemma *ccsubspace-add-left-incr*[*simp*]: $a \leq c + a$ **for** *a*::- *ccsubspace*
by (*simp add*: *add-increasing*)

lemma *sum-bot-ccsubspace*[*simp*]: ‹$(\sum x{\in}X.\ \bot) = (\bot$ :: - *ccsubspace*$)$›
by (*simp flip*: *zero-ccsubspace-def*)

7.7 Conjugate space

typedef *'a conjugate-space* = *UNIV* :: *'a set*
morphisms *from-conjugate-space to-conjugate-space* ..
setup-lifting *type-definition-conjugate-space*

instantiation *conjugate-space* :: (*complex-vector*) *complex-vector* **begin**
lift-definition *scaleC-conjugate-space* :: ‹*complex* ⇒ *'a conjugate-space* ⇒ *'a conjugate-space*› **is** ‹$\lambda c\ x.$ *cnj c* $*_C$ *x*›.
lift-definition *scaleR-conjugate-space* :: ‹*real* ⇒ *'a conjugate-space* ⇒ *'a conjugate-space*› **is** ‹$\lambda r\ x.$ *r* $*_R$ *x*›.
lift-definition *plus-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* ⇒ *'a conjugate-space* **is** $(+)$.
lift-definition *uminus-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* **is** ‹$\lambda x.\ -x$›.
lift-definition *zero-conjugate-space* :: *'a conjugate-space* **is** *0*.
lift-definition *minus-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* ⇒ *'a conjugate-space* **is** $(-)$.
instance
apply (*intro-classes*; *transfer*)
by (*simp-all add*: *scaleR-scaleC scaleC-add-right scaleC-left.add*)
end

instantiation *conjugate-space* :: (*complex-normed-vector*) *complex-normed-vector*
begin
lift-definition *sgn-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* **is** *sgn*.

lift-definition *norm-conjugate-space* :: *'a conjugate-space* ⇒ *real* **is** *norm*.
lift-definition *dist-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* ⇒
real **is** *dist*.
lift-definition *uniformity-conjugate-space* :: (*'a conjugate-space* × *'a conjugate-space*)
filter **is** *uniformity*.
lift-definition *open-conjugate-space* :: *'a conjugate-space set* ⇒ *bool* **is** *open*.
instance
apply (*intro-classes*; *transfer*)
by (*simp-all add*: *dist-norm sgn-div-norm open-uniformity uniformity-dist norm-triangle-ineq*)
end

instantiation *conjugate-space* :: (*cbanach*) *cbanach* **begin**
instance
apply *intro-classes*
unfolding *Cauchy-def convergent-def LIMSEQ-def* **apply** *transfer*
using *Cauchy-convergent* **unfolding** *Cauchy-def convergent-def LIMSEQ-def* **by**
metis
end

lemma *bounded-antilinear-to-conjugate-space*[*simp*]: ‹*bounded-antilinear to-conjugate-space*›
by (*rule bounded-antilinear-intro*[**where** *K=1*]; *transfer*; *auto*)

lemma *bounded-antilinear-from-conjugate-space*[*simp*]: ‹*bounded-antilinear from-conjugate-space*›
by (*rule bounded-antilinear-intro*[**where** *K=1*]; *transfer*; *auto*)

lemma *antilinear-to-conjugate-space*[*simp*]: ‹*antilinear to-conjugate-space*›
by (*rule antilinearI*; *transfer*, *auto*)

lemma *antilinear-from-conjugate-space*[*simp*]: ‹*antilinear from-conjugate-space*›
by (*rule antilinearI*; *transfer*, *auto*)

lemma *cspan-to-conjugate-space*[*simp*]: *cspan* (*to-conjugate-space* ‘ *X*) = *to-conjugate-space*
‘ *cspan X*
unfolding *complex-vector.span-def complex-vector.subspace-def hull-def*
apply *transfer*
apply *simp*
by (*metis* (*no-types*, *opaque-lifting*) *complex-cnj-cnj*)

lemma *surj-to-conjugate-space*[*simp*]: *surj to-conjugate-space*
by (*meson surj-def to-conjugate-space-cases*)

lemmas *has-derivative-scaleC*[*simp*, *derivative-intros*] =
bounded-bilinear.FDERIV[*OF bounded-cbilinear-scaleC*[*THEN bounded-cbilinear.bounded-bilinear*]]

lemma *norm-to-conjugate-space*[*simp*]: ‹*norm* (*to-conjugate-space x*) = *norm x*›
by (*fact norm-conjugate-space.abs-eq*)

lemma *norm-from-conjugate-space*[*simp*]: ‹*norm* (*from-conjugate-space x*) = *norm*
x›

by (*simp add*: *norm-conjugate-space.rep-eq*)

lemma *closure-to-conjugate-space*: ‹*closure* (*to-conjugate-space* ‘ *X*) = *to-conjugate-space* ‘ *closure X*›
proof −
have *1*: ‹*to-conjugate-space* ‘ *closure X* ⊆ *closure* (*to-conjugate-space* ‘ *X*)›
apply (*rule closure-bounded-linear-image-subset*)
by (*simp add*: *bounded-antilinear.bounded-linear*)
have ‹... = *to-conjugate-space* ‘ *from-conjugate-space* ‘ *closure* (*to-conjugate-space* ‘ *X*)›
by (*simp add*: *from-conjugate-space-inverse image-image*)
also have ‹... ⊆ *to-conjugate-space* ‘ *closure* (*from-conjugate-space* ‘ *to-conjugate-space* ‘ *X*)›
apply (*rule image-mono*)
apply (*rule closure-bounded-linear-image-subset*)
by (*simp add*: *bounded-antilinear.bounded-linear*)
also have ‹... = *to-conjugate-space* ‘ *closure X*›
by (*simp add*: *to-conjugate-space-inverse image-image*)
finally show *?thesis*
using *1* **by** *simp*
qed

lemma *closure-from-conjugate-space*: ‹*closure* (*from-conjugate-space* ‘ *X*) = *from-conjugate-space* ‘ *closure X*›
proof −
have *1*: ‹*from-conjugate-space* ‘ *closure X* ⊆ *closure* (*from-conjugate-space* ‘ *X*)›
apply (*rule closure-bounded-linear-image-subset*)
by (*simp add*: *bounded-antilinear.bounded-linear*)
have ‹... = *from-conjugate-space* ‘ *to-conjugate-space* ‘ *closure* (*from-conjugate-space* ‘ *X*)›
by (*simp add*: *to-conjugate-space-inverse image-image*)
also have ‹... ⊆ *from-conjugate-space* ‘ *closure* (*to-conjugate-space* ‘ *from-conjugate-space* ‘ *X*)›
apply (*rule image-mono*)
apply (*rule closure-bounded-linear-image-subset*)
by (*simp add*: *bounded-antilinear.bounded-linear*)
also have ‹... = *from-conjugate-space* ‘ *closure X*›
by (*simp add*: *from-conjugate-space-inverse image-image*)
finally show *?thesis*
using *1* **by** *simp*
qed

lemma *bounded-antilinear-eq-on*:
fixes *A B* :: *'a*::*complex-normed-vector* ⇒ *'b*::*complex-normed-vector*
assumes ‹*bounded-antilinear A*› **and** ‹*bounded-antilinear B*› **and**
eq: ‹⋀*x*. *x* ∈ *G* ⟹ *A x* = *B x*› **and** *t*: ‹*t* ∈ *closure* (*cspan G*)›
shows ‹*A t* = *B t*›
proof −
let *?A* = ‹λ*x*. *A* (*from-conjugate-space x*)› **and** *?B* = ‹λ*x*. *B* (*from-conjugate-space*

```
x)›
    and ?G = ‹to-conjugate-space ` G› and ?t = ‹to-conjugate-space t›
  have ‹bounded-clinear ?A› and ‹bounded-clinear ?B›
   by (auto intro!: bounded-antilinear-o-bounded-antilinear[OF ‹bounded-antilinear
A›]
        bounded-antilinear-o-bounded-antilinear[OF ‹bounded-antilinear B›])
  moreover from eq have ‹⋀x. x ∈ ?G ⟹ ?A x = ?B x›
    by (metis image-iff iso-tuple-UNIV-I to-conjugate-space-inverse)
  moreover from t have ‹?t ∈ closure (cspan ?G)›
   by (metis bounded-antilinear.bounded-linear bounded-antilinear-to-conjugate-space
closure-bounded-linear-image-subset cspan-to-conjugate-space imageI subsetD)
  ultimately have ‹?A ?t = ?B ?t›
    by (rule bounded-clinear-eq-on-closure)
  then show ‹A t = B t›
    by (simp add: to-conjugate-space-inverse)
qed

lemma to-conjugate-space-0[simp]: ‹to-conjugate-space 0 = 0›
  by (simp add: zero-conjugate-space.abs-eq)

lemma from-conjugate-space-0[simp]: ‹from-conjugate-space 0 = 0›
  using zero-conjugate-space.rep-eq by blast

lemma antilinear-eq-0-on-span:
  assumes ‹antilinear f›
    and ‹⋀x. x ∈ b ⟹ f x = 0›
    and ‹x ∈ cspan b›
  shows ‹f x = 0›
proof −
  from assms(1)
  have ‹clinear (λx. to-conjugate-space (f x))›
    apply (rule antilinear-o-antilinear[unfolded o-def])
    by simp
  then have ‹to-conjugate-space (f x) = 0›
    apply (rule complex-vector.linear-eq-0-on-span)
    using assms by auto
  then have ‹from-conjugate-space (to-conjugate-space (f x)) = 0›
    by simp
  then show ?thesis
    by (simp add: to-conjugate-space-inverse)
qed
```

7.8 Separating sets

```
lemma separating-set-bounded-clinear-dense:
  assumes ‹ccspan S = ⊤›
  shows ‹separating-set bounded-clinear S›
  unfolding separating-set-def
  apply (intro allI impI ext, rule bounded-clinear-eq-on-closure[where G=S])
```

by (*use assms ccspan.rep-eq top-ccsubspace.rep-eq* **in** *force*)+

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 con-*

jugate-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* (λ*x*. *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* (λ*x*. *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* (λ*y*. *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* (λ*y*. *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 ‹$z \in (\lambda(x,\ y).\ h\ x\ y)$ ` $(A \times B) \Longrightarrow f\ z = g\ z$› **for** z
then have ‹$f\ (h\ x\ y) = g\ (h\ x\ y)$› **if** ‹$x \in A$› **and** ‹$y \in B$› **for** $x\ y$
using *that* **by** *auto*
then have ‹$(\lambda x.\ f\ (h\ x\ y)) = (\lambda x.\ g\ (h\ x\ y))$› **if** ‹$y \in B$› **for** y
by (*intro eq-from-separatingI*[*OF sep-A'*]) (*use that* **in** *auto*)
then have ‹$(\lambda y.\ f\ (h\ x\ y)) = (\lambda y.\ g\ (h\ x\ y))$› **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 *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*)

7.9 Product is a Complex Vector Space

instantiation *prod* :: (*complex-vector*, *complex-vector*) *complex-vector*
begin

definition *scaleC-prod-def*:
scaleC r A = (*scaleC r* (*fst A*), *scaleC r* (*snd A*))

lemma *fst-scaleC* [*simp*]: *fst* (*scaleC r A*) = *scaleC r* (*fst A*)
unfolding *scaleC-prod-def* **by** *simp*

lemma *snd-scaleC* [*simp*]: *snd* (*scaleC r A*) = *scaleC r* (*snd A*)
unfolding *scaleC-prod-def* **by** *simp*

proposition *scaleC-Pair* [*simp*]: *scaleC r* (*a*, *b*) = (*scaleC r a*, *scaleC r b*)
unfolding *scaleC-prod-def* **by** *simp*

instance
proof
fix *a b* :: *complex* **and** *x y* :: *'a* × *'b*
show *scaleC a* (*x* + *y*) = *scaleC a x* + *scaleC a y*
by (*simp add*: *scaleC-add-right scaleC-prod-def*)

show *scaleC* (*a* + *b*) *x* = *scaleC a x* + *scaleC b x*
by (*simp add*: *Complex-Vector-Spaces.scaleC-prod-def scaleC-left.add*)
show *scaleC a* (*scaleC b x*) = *scaleC* (*a* ∗ *b*) *x*
by (*simp add*: *prod-eq-iff*)
show *scaleC 1 x* = *x*
by (*simp add*: *prod-eq-iff*)
show ‹(*scaleR* :: - ⇒ - ⇒ *'a*∗*'b*) *r* = (∗*C*) (*complex-of-real r*)› **for** *r*
by (*auto intro*!: *ext simp*: *scaleR-scaleC scaleC-prod-def scaleR-prod-def*)
qed

end

lemma *module-prod-scale-eq-scaleC*: *module-prod.scale* (∗*C*) (∗*C*) = *scaleC*
apply (*rule ext*) **apply** (*rule ext*)
apply (*subst module-prod.scale-def*)
subgoal by *unfold-locales*
by (*simp add*: *scaleC-prod-def*)

interpretation *complex-vector?*: *vector-space-prod scaleC*::-⇒-⇒*'a*::*complex-vector*
scaleC::-⇒-⇒*'b*::*complex-vector*
rewrites *scale* = ((∗*C*)::-⇒-⇒(*'a* × *'b*))
and *module.dependent* (∗*C*) = *cdependent*
and *module.representation* (∗*C*) = *crepresentation*
and *module.subspace* (∗*C*) = *csubspace*
and *module.span* (∗*C*) = *cspan*
and *vector-space.extend-basis* (∗*C*) = *cextend-basis*
and *vector-space.dim* (∗*C*) = *cdim*
and *Vector-Spaces.linear* (∗*C*) (∗*C*) = *clinear*
subgoal by *unfold-locales*
subgoal by (*fact module-prod-scale-eq-scaleC*)
unfolding *cdependent-raw-def crepresentation-raw-def csubspace-raw-def cspan-raw-def*
cextend-basis-raw-def cdim-raw-def clinear-def
by (*rule refl*)+

instance *prod* :: (*complex-normed-vector*, *complex-normed-vector*) *complex-normed-vector*

proof
fix *c* :: *complex* **and** *x y* :: *'a* × *'b*
show *norm* (*c* ∗*C x*) = *cmod c* ∗ *norm x*
unfolding *norm-prod-def*
apply (*simp add*: *power-mult-distrib*)
apply (*simp add*: *distrib-left* [*symmetric*])
by (*simp add*: *real-sqrt-mult*)
qed

lemma *cspan-Times*: ‹*cspan* (*S* × *T*) = *cspan S* × *cspan T*› **if** ‹*0* ∈ *S*› **and** ‹*0* ∈ *T*›
proof

```
  have ‹fst ‘ cspan (S × T) ⊆ cspan S›
    apply (subst complex-vector.linear-span-image[symmetric])
    using that complex-vector.module-hom-fst by auto
  moreover have ‹snd ‘ cspan (S × T) ⊆ cspan T›
    apply (subst complex-vector.linear-span-image[symmetric])
    using that complex-vector.module-hom-snd by auto
  ultimately show ‹cspan (S × T) ⊆ cspan S × cspan T›
    by auto

  show ‹cspan S × cspan T ⊆ cspan (S × T)›
  proof
    fix x assume assm: ‹x ∈ cspan S × cspan T›
    then have ‹fst x ∈ cspan S›
      by auto
     then obtain t1 r1 where fst-x: ‹fst x = (∑ a∈t1. r1 a *C a)› and [simp]:
‹finite t1› and ‹t1 ⊆ S›
      by (auto simp add: complex-vector.span-explicit)
    from assm
    have ‹snd x ∈ cspan T›
      by auto
    then obtain t2 r2 where snd-x: ‹snd x = (∑ a∈t2. r2 a *C a)› and [simp]:
‹finite t2› and ‹t2 ⊆ T›
      by (auto simp add: complex-vector.span-explicit)
     define t :: ‹('a+'b) set› and r :: ‹('a+'b) ⇒ complex› and f :: ‹('a+'b) ⇒
('a×'b)›
      where ‹t = t1 <+> t2›
      and ‹r a = (case a of Inl a1 ⇒ r1 a1 | Inr a2 ⇒ r2 a2)›
      and ‹f a = (case a of Inl a1 ⇒ (a1,0) | Inr a2 ⇒ (0,a2))›
    for a
    have ‹finite t›
      by (simp add: t-def)
    moreover have ‹f ‘ t ⊆ S × T›
      using ‹t1 ⊆ S› ‹t2 ⊆ T› that
      by (auto simp: f-def t-def)
    moreover have ‹(fst x, snd x) = (∑ a∈t. r a *C f a)›
      apply (simp only: fst-x snd-x)
      by (auto simp: t-def sum.Plus r-def f-def sum-prod)
    ultimately show ‹x ∈ cspan (S × T)›
      apply auto
     by (smt (verit, best) complex-vector.span-scale complex-vector.span-sum com-
plex-vector.span-superset image-subset-iff subset-iff)
  qed
qed

lemma onorm-case-prod-plus: ‹onorm (case-prod plus :: - ⇒ 'a::{real-normed-vector,
not-singleton}) = sqrt 2›
proof −
  obtain x :: 'a where ‹x ≠ 0›
    apply atomize-elim by auto
```

show *?thesis*
apply (*rule onormI*[**where** *x*=‹(*x*,*x*)›])
using *norm-plus-leq-norm-prod* **apply** *force*
using ‹*x* ≠ *0*›
by (*auto simp add*: *zero-prod-def norm-prod-def real-sqrt-mult*
simp flip: *scaleR-2*)
qed

7.10 Copying existing theorems into sublocales

context *bounded-clinear* **begin**
interpretation *bounded-linear f* **by** (*rule bounded-linear*)
lemmas *continuous* = *real.continuous*
lemmas *uniform-limit* = *real.uniform-limit*
lemmas *Cauchy* = *real.Cauchy*
end

context *bounded-antilinear* **begin**
interpretation *bounded-linear f* **by** (*rule bounded-linear*)
lemmas *continuous* = *real.continuous*
lemmas *uniform-limit* = *real.uniform-limit*
end

context *bounded-cbilinear* **begin**
interpretation *bounded-bilinear prod* **by** *simp*
lemmas *tendsto* = *real.tendsto*
lemmas *isCont* = *real.isCont*
lemmas *scaleR-right* = *real.scaleR-right*
lemmas *scaleR-left* = *real.scaleR-left*
end

context *bounded-sesquilinear* **begin**
interpretation *bounded-bilinear prod* **by** *simp*
lemmas *tendsto* = *real.tendsto*
lemmas *isCont* = *real.isCont*
lemmas *scaleR-right* = *real.scaleR-right*
lemmas *scaleR-left* = *real.scaleR-left*
end

lemmas *tendsto-scaleC* [*tendsto-intros*] =
bounded-cbilinear.tendsto [*OF bounded-cbilinear-scaleC*]

unbundle *no lattice-syntax*

end

8 *Complex-Inner-Product0* – Inner Product Spaces and Gradient Derivative

theory *Complex-Inner-Product0*
imports
Complex-Main Complex-Vector-Spaces
HOL−Analysis.Inner-Product
Complex-Bounded-Operators.Extra-Ordered-Fields
begin

8.1 Complex inner product spaces

Temporarily relax type constraints for *open*, *uniformity*, *dist*, and *norm*.

setup ‹*Sign.add-const-constraint*
(***const-name*** ‹*open*›, *SOME* ***typ*** ‹'*a*::*open set* ⇒ *bool*›)›

setup ‹*Sign.add-const-constraint*
(***const-name*** ‹*dist*›, *SOME* ***typ*** ‹'*a*::*dist* ⇒ '*a* ⇒ *real*›)›

setup ‹*Sign.add-const-constraint*
(***const-name*** ‹*uniformity*›, *SOME* ***typ*** ‹('*a*::*uniformity* × '*a*) *filter*›)›

setup ‹*Sign.add-const-constraint*
(***const-name*** ‹*norm*›, *SOME* ***typ*** ‹'*a*::*norm* ⇒ *real*›)›

class *complex-inner* = *complex-vector* + *sgn-div-norm* + *dist-norm* + *uniformity-dist* + *open-uniformity* +
fixes *cinner* :: '*a* ⇒ '*a* ⇒ *complex*
assumes *cinner-commute*: *cinner x y* = *cnj* (*cinner y x*)
and *cinner-add-left*: *cinner* (*x* + *y*) *z* = *cinner x z* + *cinner y z*
and *cinner-scaleC-left* [*simp*]: *cinner* (*scaleC r x*) *y* = (*cnj r*) ∗ (*cinner x y*)
and *cinner-ge-zero* [*simp*]: *0* ≤ *cinner x x*
and *cinner-eq-zero-iff* [*simp*]: *cinner x x* = *0* ⟷ *x* = *0*
and *norm-eq-sqrt-cinner*: *norm x* = *sqrt* (*cmod* (*cinner x x*))
begin

lemma *cinner-zero-left* [*simp*]: *cinner 0 x* = *0*
using *cinner-add-left* [*of 0 0 x*] **by** *simp*

lemma *cinner-minus-left* [*simp*]: *cinner* (− *x*) *y* = − *cinner x y*
using *cinner-add-left* [*of x* − *x y*]
by (*simp add*: *group-add-class.add-eq-0-iff*)

lemma *cinner-diff-left*: *cinner* (*x* − *y*) *z* = *cinner x z* − *cinner y z*
using *cinner-add-left* [*of x* − *y z*] **by** *simp*

lemma *cinner-sum-left*: *cinner* (∑ *x*∈*A*. *f x*) *y* = (∑ *x*∈*A*. *cinner* (*f x*) *y*)
by (*cases finite A*, *induct set*: *finite*, *simp-all add*: *cinner-add-left*)

lemma *call-zero-iff* [*simp*]: $(\forall\, u.\ cinner\ x\ u = 0) \longleftrightarrow (x = 0)$
by *auto* (*use cinner-eq-zero-iff* **in** *blast*)

Transfer distributivity rules to right argument.

lemma *cinner-add-right*: $cinner\ x\ (y + z) = cinner\ x\ y + cinner\ x\ z$
using *cinner-add-left* [*of y z x*]
by (*metis complex-cnj-add local.cinner-commute*)

lemma *cinner-scaleC-right* [*simp*]: $cinner\ x\ (scaleC\ r\ y) = r * (cinner\ x\ y)$
using *cinner-scaleC-left* [*of r y x*]
by (*metis complex-cnj-cnj complex-cnj-mult local.cinner-commute*)

lemma *cinner-zero-right* [*simp*]: $cinner\ x\ 0 = 0$
using *cinner-zero-left* [*of x*]
by (*metis* (*mono-tags*, *opaque-lifting*) *complex-cnj-zero local.cinner-commute*)

lemma *cinner-minus-right* [*simp*]: $cinner\ x\ (-\ y) = -\ cinner\ x\ y$
using *cinner-minus-left* [*of y x*]
by (*metis complex-cnj-minus local.cinner-commute*)

lemma *cinner-diff-right*: $cinner\ x\ (y - z) = cinner\ x\ y - cinner\ x\ z$
using *cinner-diff-left* [*of y z x*]
by (*metis complex-cnj-diff local.cinner-commute*)

lemma *cinner-sum-right*: $cinner\ x\ (\sum y{\in}A.\ f\ y) = (\sum y{\in}A.\ cinner\ x\ (f\ y))$
proof (*subst cinner-commute*)
have $(\sum y{\in}A.\ cinner\ (f\ y)\ x) = (\sum y{\in}A.\ cinner\ (f\ y)\ x)$
by *blast*
hence $cnj\ (\sum y{\in}A.\ cinner\ (f\ y)\ x) = cnj\ (\sum y{\in}A.\ (cinner\ (f\ y)\ x))$
by *simp*
hence $cnj\ (cinner\ (sum\ f\ A)\ x) = (\sum y{\in}A.\ cnj\ (cinner\ (f\ y)\ x))$
by (*simp add*: *cinner-sum-left*)
thus $cnj\ (cinner\ (sum\ f\ A)\ x) = (\sum y{\in}A.\ (cinner\ x\ (f\ y)))$
by (*subst* (*2*) *cinner-commute*)
qed

lemmas *cinner-add* [*algebra-simps*] = *cinner-add-left cinner-add-right*
lemmas *cinner-diff* [*algebra-simps*] = *cinner-diff-left cinner-diff-right*
lemmas *cinner-scaleC* = *cinner-scaleC-left cinner-scaleC-right*

lemma *cinner-gt-zero-iff* [*simp*]: $0 < cinner\ x\ x \longleftrightarrow x \neq 0$
by (*smt* (*verit*) *less-irrefl local.cinner-eq-zero-iff local.cinner-ge-zero order.not-eq-order-implies-strict*)

lemma *power2-norm-eq-cinner*:

shows (*complex-of-real* (*norm x*))^2 = (*cinner x x*)
by (*smt* (*verit*, *del-insts*) *Im-complex-of-real Re-complex-of-real cinner-gt-zero-iff cinner-zero-right cmod-def complex-eq-0 complex-eq-iff less-complex-def local.norm-eq-sqrt-cinner of-real-power real-sqrt-abs real-sqrt-pow2-iff zero-complex.sel*(*1*))

lemma *power2-norm-eq-cinner'*:
shows (*norm x*)^2 = *Re* (*cinner x x*)
by (*metis Re-complex-of-real of-real-power power2-norm-eq-cinner*)

Identities involving real multiplication and division.

lemma *cinner-mult-left*: *cinner* (*of-complex m* ∗ *a*) *b* = *cnj m* ∗ (*cinner a b*)
by (*simp add*: *of-complex-def*)

lemma *cinner-mult-right*: *cinner a* (*of-complex m* ∗ *b*) = *m* ∗ (*cinner a b*)
by (*metis complex-inner-class.cinner-scaleC-right scaleC-conv-of-complex*)

lemma *cinner-mult-left'*: *cinner* (*a* ∗ *of-complex m*) *b* = *cnj m* ∗ (*cinner a b*)
by (*metis cinner-mult-left mult.right-neutral mult-scaleC-right scaleC-conv-of-complex*)

lemma *cinner-mult-right'*: *cinner a* (*b* ∗ *of-complex m*) = (*cinner a b*) ∗ *m*
by (*simp add*: *complex-inner-class.cinner-scaleC-right of-complex-def*)

lemma *Cauchy-Schwarz-ineq*:
(*cinner x y*) ∗ (*cinner y x*) ≤ *cinner x x* ∗ *cinner y y*
proof (*cases*)
assume *y* = *0*
thus *?thesis* **by** *simp*
next
assume *y*: *y* ≠ *0*
have [*simp*]: *cnj* (*cinner y y*) = *cinner y y* **for** *y*
by (*metis cinner-commute*)
define *r* **where** *r* = *cnj* (*cinner x y*) / *cinner y y*
have *0* ≤ *cinner* (*x* − *scaleC r y*) (*x* − *scaleC r y*)
by (*rule cinner-ge-zero*)
also have ... = *cinner x x* − *r* ∗ *cinner x y* − *cnj r* ∗ *cinner y x* + *r* ∗ *cnj r* ∗ *cinner y y*
unfolding *cinner-diff-left cinner-diff-right cinner-scaleC-left cinner-scaleC-right*
by (*smt* (*z3*) *cancel-comm-monoid-add-class.diff-cancel cancel-comm-monoid-add-class.diff-zero complex-cnj-divide group-add-class.diff-add-cancel local.cinner-commute local.cinner-eq-zero-iff local.cinner-scaleC-left mult.assoc mult.commute mult-eq-0-iff nonzero-eq-divide-eq r-def y*)
also have ... = *cinner x x* − *cinner y x* ∗ *cnj r*
unfolding *r-def* **by** *auto*
also have ... = *cinner x x* − *cinner x y* ∗ *cnj* (*cinner x y*) / *cinner y y*
unfolding *r-def*
by (*metis complex-cnj-divide local.cinner-commute mult.commute times-divide-eq-left*)
finally have *0* ≤ *cinner x x* − *cinner x y* ∗ *cnj* (*cinner x y*) / *cinner y y* **.**

hence *cinner x y * cnj (cinner x y) / cinner y y ≤ cinner x x*
by (*simp add*: *le-diff-eq*)
thus *cinner x y * cinner y x ≤ cinner x x * cinner y y*
by (*metis cinner-gt-zero-iff local.cinner-commute nice-ordered-field-class.pos-divide-le-eq y*)
qed

lemma *Cauchy-Schwarz-ineq2*:
shows *norm (cinner x y) ≤ norm x * norm y*
proof (*rule power2-le-imp-le*)
have *(norm (cinner x y))^2 = Re (cinner x y * cinner y x)*
by (*metis* (*full-types*) *Re-complex-of-real complex-norm-square local.cinner-commute*)
also have *... ≤ Re (cinner x x * cinner y y)*
using *Cauchy-Schwarz-ineq* **by** (*rule Re-mono*)
also have *... = Re (complex-of-real ((norm x)^2) * complex-of-real ((norm y)^2))*
by (*simp add*: *power2-norm-eq-cinner*)
also have *... = (norm x * norm y)^2*
by (*simp add*: *power-mult-distrib*)
finally show *(cmod (cinner x y))^2 ≤ (norm x * norm y)^2* **.**
show *0 ≤ norm x * norm y*
by (*simp add*: *local.norm-eq-sqrt-cinner*)
qed

subclass *complex-normed-vector*
proof
fix *a* :: *complex* **and** *r* :: *real* **and** *x y* :: *'a*
show *norm x = 0 ⟷ x = 0*
unfolding *norm-eq-sqrt-cinner* **by** *simp*
show *norm (x + y) ≤ norm x + norm y*
proof (*rule power2-le-imp-le*)
have *Re (cinner x y) ≤ cmod (cinner x y)*
if *⋀x. Re x ≤ cmod x* **and**
⋀x y. x ≤ y ⟹ complex-of-real x ≤ complex-of-real y
using *that* **by** *simp*
hence *a1*: *2 * Re (cinner x y) ≤ 2 * cmod (cinner x y)*
if *⋀x. Re x ≤ cmod x* **and**
⋀x y. x ≤ y ⟹ complex-of-real x ≤ complex-of-real y
using *that* **by** *simp*
have *cinner x y + cinner y x = complex-of-real (2 * Re (cinner x y))*
by (*metis complex-add-cnj local.cinner-commute*)
also have *... ≤ complex-of-real (2 * cmod (cinner x y))*
using *complex-Re-le-cmod complex-of-real-mono a1*
by *blast*
also have *... = 2 * abs (cinner x y)*

unfolding *abs-complex-def* **by** *simp*
also have ... $\leq$ *2* $*$ *complex-of-real* (*norm x*) $*$ *complex-of-real* (*norm y*)
using *Cauchy-Schwarz-ineq2* **unfolding** *abs-complex-def less-eq-complex-def*
by *auto*
finally have *xyyx*: *cinner x y* + *cinner y x* $\leq$ *complex-of-real* (*2* $*$ *norm x* $*$ *norm y*)
by *auto*
have *complex-of-real* $((norm\ (x + y))^2)$ = *cinner* (*x*+*y*) (*x*+*y*)
by (*simp add*: *power2-norm-eq-cinner*)
also have ... = *cinner x x* + *cinner x y* + *cinner y x* + *cinner y y*
by (*simp add*: *cinner-add*)
also have ... = *complex-of-real* $((norm\ x)^2)$ + *complex-of-real* $((norm\ y)^2)$ + *cinner x y* + *cinner y x*
by (*simp add*: *power2-norm-eq-cinner*)
also have ... $\leq$ *complex-of-real* $((norm\ x)^2)$ + *complex-of-real* $((norm\ y)^2)$ + *complex-of-real* (*2* $*$ *norm x* $*$ *norm y*)
using *xyyx* **by** *auto*
also have ... = *complex-of-real* $((norm\ x + norm\ y)^2)$
unfolding *power2-sum* **by** *auto*
finally show $(norm\ (x + y))^2 \leq (norm\ x + norm\ y)^2$
using *complex-of-real-mono-iff* **by** *blast*
show *0* $\leq$ *norm x* + *norm y*
unfolding *norm-eq-sqrt-cinner* **by** *simp*
qed
show *norm-scaleC*: *norm* (*a* $*_C$ *x*) = *cmod a* $*$ *norm x* **for** *a*
proof (*rule power2-eq-imp-eq*)
show $(norm\ (a *_C\ x))^2 = (cmod\ a * norm\ x)^2$
by (*simp-all add*: *norm-eq-sqrt-cinner norm-mult power2-eq-square*)
show *0* $\leq$ *norm* (*a* $*_C$ *x*)
by (*simp-all add*: *norm-eq-sqrt-cinner*)
show *0* $\leq$ *cmod a* $*$ *norm x*
by (*simp-all add*: *norm-eq-sqrt-cinner*)
qed
show *norm* (*r* $*_R$ *x*) = $|r|$ $*$ *norm x*
unfolding *scaleR-scaleC norm-scaleC* **by** *auto*
qed

end

lemma *csquare-continuous*:
fixes *e* :: *real*
shows *e* > *0* $\Longrightarrow$ $\exists$ *d*. *0* < *d* $\wedge$ ($\forall$ *y*. *cmod* (*y* − *x*) < *d* $\longrightarrow$ *cmod* (*y* $*$ *y* − *x* $*$ *x*) < *e*)
using *isCont-power*[*OF continuous-ident*, *of x*, *unfolded isCont-def LIM-eq*, *rule-format*, *of e 2*]
by (*force simp add*: *power2-eq-square*)

lemma *cnorm-le*: *norm x* ≤ *norm y* ⟷ *cinner x x* ≤ *cinner y y*
by (*smt* (*verit*) *complex-of-real-mono-iff norm-eq-sqrt-cinner norm-ge-zero of-real-power power2-norm-eq-cinner real-sqrt-le-mono real-sqrt-pow2*)

lemma *cnorm-lt*: *norm x* < *norm y* ⟷ *cinner x x* < *cinner y y*
by (*meson cnorm-le less-le-not-le*)

lemma *cnorm-eq*: *norm x* = *norm y* ⟷ *cinner x x* = *cinner y y*
by (*metis norm-eq-sqrt-cinner power2-norm-eq-cinner*)

lemma *cnorm-eq-1*: *norm x* = 1 ⟷ *cinner x x* = 1
by (*metis cinner-ge-zero cmod-Re norm-eq-sqrt-cinner norm-one of-real-1 of-real-power power2-norm-eq-cinner power2-norm-eq-cinner' real-sqrt-eq-iff real-sqrt-one*)

lemma *cinner-divide-left*:
fixes *a* :: *'a* :: {*complex-inner*,*complex-div-algebra*}
shows *cinner* (*a* / *of-complex m*) *b* = (*cinner a b*) / *cnj m*
by (*metis cinner-mult-left' complex-cnj-inverse divide-inverse mult.commute of-complex-inverse*)

lemma *cinner-divide-right*:
fixes *a* :: *'a* :: {*complex-inner*,*complex-div-algebra*}
shows *cinner a* (*b* / *of-complex m*) = (*cinner a b*) / *m*
by (*metis cinner-mult-right' divide-inverse of-complex-inverse*)

Re-enable constraints for *open*, *uniformity*, *dist*, and *norm*.

setup ‹*Sign.add-const-constraint*
(***const-name*** ‹*open*›, *SOME* ***typ*** ‹*'a*::*topological-space set* ⇒ *bool*›)›

setup ‹*Sign.add-const-constraint*
(***const-name*** ‹*uniformity*›, *SOME* ***typ*** ‹(*'a*::*uniform-space* × *'a*) *filter*›)›

setup ‹*Sign.add-const-constraint*
(***const-name*** ‹*dist*›, *SOME* ***typ*** ‹*'a*::*metric-space* ⇒ *'a* ⇒ *real*›)›

setup ‹*Sign.add-const-constraint*
(***const-name*** ‹*norm*›, *SOME* ***typ*** ‹*'a*::*real-normed-vector* ⇒ *real*›)›

lemma *bounded-sesquilinear-cinner*:
bounded-sesquilinear (*cinner*::*'a*::*complex-inner* ⇒ *'a* ⇒ *complex*)
proof
fix *x y z* :: *'a* **and** *r* :: *complex*
show *cinner* (*x* + *y*) *z* = *cinner x z* + *cinner y z*
by (*rule cinner-add-left*)
show *cinner x* (*y* + *z*) = *cinner x y* + *cinner x z*
by (*rule cinner-add-right*)
show *cinner* (*scaleC r x*) *y* = *scaleC* (*cnj r*) (*cinner x y*)
unfolding *complex-scaleC-def* **by** (*rule cinner-scaleC-left*)

show *cinner x* (*scaleC r y*) = *scaleC r* (*cinner x y*)
unfolding *complex-scaleC-def* **by** (*rule cinner-scaleC-right*)
have ∀ *x y*::′*a*. *norm* (*cinner x y*) ≤ *norm x* ∗ *norm y* ∗ *1*
by (*simp add*: *complex-inner-class.Cauchy-Schwarz-ineq2*)
thus ∃ *K*. ∀ *x y*::′*a*. *norm* (*cinner x y*) ≤ *norm x* ∗ *norm y* ∗ *K*
by *metis*
qed

lemmas *tendsto-cinner* [*tendsto-intros*] =
bounded-bilinear.tendsto [*OF bounded-sesquilinear-cinner*[*THEN bounded-sesquilinear.bounded-bilinear*]]

lemmas *isCont-cinner* [*simp*] =
bounded-bilinear.isCont [*OF bounded-sesquilinear-cinner*[*THEN bounded-sesquilinear.bounded-bilinear*]]

lemmas *has-derivative-cinner* [*derivative-intros*] =
bounded-bilinear.FDERIV [*OF bounded-sesquilinear-cinner*[*THEN bounded-sesquilinear.bounded-bilinear*]]

lemmas *bounded-antilinear-cinner-left* =
bounded-sesquilinear.bounded-antilinear-left [*OF bounded-sesquilinear-cinner*]

lemmas *bounded-clinear-cinner-right* =
bounded-sesquilinear.bounded-clinear-right [*OF bounded-sesquilinear-cinner*]

lemmas *bounded-antilinear-cinner-left-comp* = *bounded-antilinear-cinner-left*[*THEN*
bounded-antilinear-o-bounded-clinear]

lemmas *bounded-clinear-cinner-right-comp* = *bounded-clinear-cinner-right*[*THEN*
bounded-clinear-compose]

lemmas *has-derivative-cinner-right* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-clinear-cinner-right*[*THEN bounded-clinear.bounded-linear*]]

lemmas *has-derivative-cinner-left* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-antilinear-cinner-left*[*THEN bounded-antilinear.bounded-linear*]]

lemma *differentiable-cinner* [*simp*]:
f differentiable (*at x within s*) ⟹ *g differentiable at x within s* ⟹ (λ*x*. *cinner*
(*f x*) (*g x*)) *differentiable at x within s*
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-cinner*)

8.2 Class instances

instantiation *complex* :: *complex-inner*
begin

definition *cinner-complex-def* [*simp*]: *cinner x y* = *cnj x* ∗ *y*

instance
proof

fix *x y z r* :: *complex*
show *cinner x y* = *cnj* (*cinner y x*)
unfolding *cinner-complex-def* **by** *auto*
show *cinner* (*x* + *y*) *z* = *cinner x z* + *cinner y z*
unfolding *cinner-complex-def*
by (*simp add*: *ring-class.ring-distribs*(*2*))
show *cinner* (*scaleC r x*) *y* = *cnj r* ∗ *cinner x y*
unfolding *cinner-complex-def complex-scaleC-def* **by** *simp*
show *0* ≤ *cinner x x*
by *simp*
show *cinner x x* = *0* ⟷ *x* = *0*
unfolding *cinner-complex-def* **by** *simp*
have *cmod* (*Complex x1 x2*) = *sqrt* (*cmod* (*cinner* (*Complex x1 x2*) (*Complex x1 x2*)))
for *x1 x2*
unfolding *cinner-complex-def complex-cnj complex-mult complex-norm*
by (*simp add*: *power2-eq-square*)
thus *norm x* = *sqrt* (*cmod* (*cinner x x*))
by (*cases x*, *hypsubst-thin*)
qed

end

lemma
shows *complex-inner-1-left*[*simp*]: *cinner 1 x* = *x*
and *complex-inner-1-right*[*simp*]: *cinner x 1* = *cnj x*
by *simp-all*

lemma *cdot-square-norm*: *cinner x x* = *complex-of-real* ((*norm x*)^2)
by (*metis Im-complex-of-real Re-complex-of-real cinner-ge-zero complex-eq-iff less-eq-complex-def power2-norm-eq-cinner′ zero-complex.simps*(*2*))

lemma *cnorm-eq-square*: *norm x* = *a* ⟷ *0* ≤ *a* ∧ *cinner x x* = *complex-of-real* (a^2)
by (*metis cdot-square-norm norm-ge-zero of-real-eq-iff power2-eq-iff-nonneg*)

lemma *cnorm-le-square*: *norm x* ≤ *a* ⟷ *0* ≤ *a* ∧ *cinner x x* ≤ *complex-of-real* (a^2)
by (*smt* (*verit*) *cdot-square-norm complex-of-real-mono-iff norm-ge-zero power2-le-imp-le*)

lemma *cnorm-ge-square*: *norm x* ≥ *a* ⟷ *a* ≤ *0* ∨ *cinner x x* ≥ *complex-of-real* (a^2)
by (*smt* (*verit*, *best*) *antisym-conv cnorm-eq-square cnorm-le-square complex-of-real-nn-iff nn-comparable zero-le-power2*)

lemma *norm-lt-square*: *norm x* < *a* ⟷ *0* < *a* ∧ *cinner x x* < *complex-of-real* (a^2)

by (*meson cnorm-ge-square cnorm-le-square less-le-not-le*)

lemma *norm-gt-square*: *norm x > a* $\longleftrightarrow$ *a < 0* $\vee$ *cinner x x > complex-of-real* (a^2)
by (*smt* (*verit*, *ccfv-SIG*) *cdot-square-norm complex-of-real-strict-mono-iff norm-ge-zero power2-eq-imp-eq power-mono*)

Dot product in terms of the norm rather than conversely.

lemmas *cinner-simps* = *cinner-add-left cinner-add-right cinner-diff-right cinner-diff-left cinner-scaleC-left cinner-scaleC-right*

lemma *cdot-norm*: *cinner x y* = $((norm\ (x+y))^2 - (norm\ (x-y))^2 - \mathrm{i} * (norm\ (x + \mathrm{i} *_C y))^2 + \mathrm{i} * (norm\ (x - \mathrm{i} *_C y))^2)$ / *4*
unfolding *power2-norm-eq-cinner*
by (*simp add*: *power2-norm-eq-cinner cinner-add-left cinner-add-right cinner-diff-left cinner-diff-right ring-distribs*)

lemma *of-complex-inner-1* [*simp*]:
cinner (*of-complex x*) (*1* :: *'a* :: {*complex-inner*, *complex-normed-algebra-1*}) = *cnj x*
by (*metis Complex-Inner-Product0.complex-inner-1-right cinner-complex-def cinner-mult-left complex-cnj-one norm-one of-complex-def power2-norm-eq-cinner scaleC-conv-of-complex*)

lemma *summable-of-complex-iff*:
summable (*λx. of-complex* (*f x*) :: *'a* :: {*complex-normed-algebra-1*,*complex-inner*}) $\longleftrightarrow$ *summable f*
proof
assume ∗: *summable* (*λx. of-complex* (*f x*) :: *'a*)
have *bounded-clinear* (*cinner* (*1*::*'a*))
by (*rule bounded-clinear-cinner-right*)
then interpret *bounded-linear λx*::*'a. cinner 1 x*
by (*rule bounded-clinear.bounded-linear*)
from *summable* [*OF* ∗] **show** *summable f*
apply (*subst* (*asm*) *cinner-commute*) **by** *simp*
next
assume *sum*: *summable f*
thus *summable* (*λx. of-complex* (*f x*) :: *'a*)
by (*rule summable-of-complex*)
qed

8.3 Gradient derivative

definition
cgderiv :: [*'a*::*complex-inner* ⇒ *complex*, *'a*, *'a*] ⇒ *bool*
(‹(*cGDERIV* (-)/ (-)/ :> (-))› [*1000*, *1000*, *60*] *60*)
where

cGDERIV f x :> D $\longleftrightarrow$ *FDERIV f x :> cinner D*

lemma *cgderiv-deriv* [*simp*]: *cGDERIV f x :> D* $\longleftrightarrow$ *DERIV f x :> cnj D*
by (*simp only*: *cgderiv-def has-field-derivative-def cinner-complex-def*[*THEN ext*])

lemma *cGDERIV-DERIV-compose*:
assumes *cGDERIV f x :> df* **and** *DERIV g* (*f x*) *:> cnj dg*
shows *cGDERIV* (*λx. g* (*f x*)) *x :> scaleC dg df*
proof (*insert assms*)
show *cGDERIV* (*λx. g* (*f x*)) *x :> dg* $*_C$ *df*
if *cGDERIV f x :> df*
and (*g has-field-derivative cnj dg*) (*at* (*f x*))
unfolding *cgderiv-def has-field-derivative-def cinner-scaleC-left complex-cnj-cnj*
using *that*
by (*simp add*: *cgderiv-def has-derivative-compose has-field-derivative-imp-has-derivative*)

qed

lemma *cGDERIV-subst*: ⟦*cGDERIV f x :> df*; *df* = *d*⟧ ⟹ *cGDERIV f x :> d*
by *simp*

lemma *cGDERIV-const*: *cGDERIV* (*λx. k*) *x :> 0*
unfolding *cgderiv-def cinner-zero-left*[*THEN ext*] **by** (*rule has-derivative-const*)

lemma *cGDERIV-add*:
⟦*cGDERIV f x :> df*; *cGDERIV g x :> dg*⟧
⟹ *cGDERIV* (*λx. f x* + *g x*) *x :> df* + *dg*
unfolding *cgderiv-def cinner-add-left*[*THEN ext*] **by** (*rule has-derivative-add*)

lemma *cGDERIV-minus*:
cGDERIV f x :> df ⟹ *cGDERIV* (*λx.* − *f x*) *x :>* − *df*
unfolding *cgderiv-def cinner-minus-left*[*THEN ext*] **by** (*rule has-derivative-minus*)

lemma *cGDERIV-diff*:
⟦*cGDERIV f x :> df*; *cGDERIV g x :> dg*⟧
⟹ *cGDERIV* (*λx. f x* − *g x*) *x :> df* − *dg*
unfolding *cgderiv-def cinner-diff-left* **by** (*rule has-derivative-diff*)

lemma *cGDERIV-scaleC*:
⟦*DERIV f x :> df*; *cGDERIV g x :> dg*⟧
⟹ *cGDERIV* (*λx. scaleC* (*f x*) (*g x*)) *x*
:> (*scaleC* (*cnj* (*f x*)) *dg* + *scaleC* (*cnj df*) (*cnj* (*g x*)))
unfolding *cgderiv-def has-field-derivative-def cinner-add-left cinner-scaleC-left*
apply (*rule has-derivative-subst*)
apply (*erule* (*1*) *has-derivative-scaleC*)
by (*simp add*: *ac-simps*)

lemma *GDERIV-mult*:
 ⟦*cGDERIV f x :> df*; *cGDERIV g x :> dg*⟧
  ⟹ *cGDERIV* (λ*x*. *f x* ∗ *g x*) *x :> cnj* (*f x*) ∗$_C$ *dg* + *cnj* (*g x*) ∗$_C$ *df*
 unfolding *cgderiv-def*
 apply (*rule has-derivative-subst*)
 apply (*erule* (*1*) *has-derivative-mult*)
 apply (*rule ext*)
 by (*simp add*: *cinner-add ac-simps*)

lemma *cGDERIV-inverse*:
 ⟦*cGDERIV f x :> df*; *f x* ≠ *0*⟧
  ⟹ *cGDERIV* (λ*x*. *inverse* (*f x*)) *x :>* − *cnj* ((*inverse* (*f x*))^2) ∗$_C$ *df*
 by (*metis DERIV-inverse cGDERIV-DERIV-compose complex-cnj-cnj complex-cnj-minus numerals*(*2*))

lemma *has-derivative-norm*[*derivative-intros*]:
 fixes *x* :: *'a*::*complex-inner*
 assumes *x* ≠ *0*
 shows (*norm has-derivative* (λ*h*. *Re* (*cinner* (*sgn x*) *h*))) (*at x*)
 thm *has-derivative-norm*
proof −
 have *Re-pos*: *0* < *Re* (*cinner x x*)
  using *assms*
  by (*metis Re-strict-mono cinner-gt-zero-iff zero-complex.simps*(*1*))
 have *Re-plus-Re*: *Re* (*cinner x y*) + *Re* (*cinner y x*) = *2* ∗ *Re* (*cinner x y*)
  for *x y* :: *'a*
  by (*metis cinner-commute cnj.simps*(*1*) *mult-2-right semiring-normalization-rules*(*7*))
 have *norm*: *norm x* = *sqrt* (*Re* (*cinner x x*)) **for** *x* :: *'a*
  apply (*subst norm-eq-sqrt-cinner*, *subst cmod-Re*)
  using *cinner-ge-zero* **by** *auto*
 have *v2*:((λ*x*. *sqrt* (*Re* (*cinner x x*))) *has-derivative*
   (λ*xa*. (*Re* (*cinner x xa*) + *Re* (*cinner xa x*)) ∗ (*inverse* (*sqrt* (*Re* (*cinner x x*))) / *2*))) (*at x*)
  by (*rule derivative-eq-intros* | *simp add*: *Re-pos*)+
 have *v1*: ((λ*x*. *sqrt* (*Re* (*cinner x x*))) *has-derivative* (λ*y*. *Re* (*cinner x y*) / *sqrt* (*Re* (*cinner x x*)))) (*at x*)
  if ((λ*x*. *sqrt* (*Re* (*cinner x x*))) *has-derivative* (λ*xa*. *Re* (*cinner x xa*) ∗ *inverse* (*sqrt* (*Re* (*cinner x x*))))) (*at x*)
  using *that* **apply** (*subst divide-real-def*)
  by *simp*
 have ‹(*norm has-derivative* (λ*y*. *Re* (*cinner x y*) / *norm x*)) (*at x*)›
  using *v2*
  apply (*auto simp*: *Re-plus-Re norm* [*abs-def*])
  using *v1* **by** *blast*
 then show *?thesis*

by (*auto simp*: *power2-eq-square sgn-div-norm scaleR-scaleC*)
qed

bundle *cinner-syntax*
begin
notation *cinner* (**infix** $\langle \cdot_C \rangle$ *70*)
end

end

9 *Complex-Inner-Product* – Complex Inner Product Spaces

theory *Complex-Inner-Product*
imports
Complex-Inner-Product0
begin

9.1 Complex inner product spaces

unbundle *cinner-syntax*

lemma *cinner-real*: *cinner x x* $\in \mathbb{R}$
by (*simp add*: *cdot-square-norm*)

lemmas *cinner-commute'* [*simp*] = *cinner-commute*[*symmetric*]

lemma (**in** *complex-inner*) *cinner-eq-flip*: $\langle$(*cinner x y* = *cinner z w*) $\longleftrightarrow$ (*cinner y x* = *cinner w z*)$\rangle$
by (*metis cinner-commute*)

lemma *Im-cinner-x-x*[*simp*]: *Im* (*x* $\cdot_C$ *x*) = *0*
using *comp-Im-same*[*OF cinner-ge-zero*] **by** *simp*

lemma *of-complex-inner-1'* [*simp*]:
cinner (*1* :: *'a* :: {*complex-inner*, *complex-normed-algebra-1*}) (*of-complex x*) = *x*
by (*metis cinner-commute complex-cnj-cnj of-complex-inner-1*)

class *chilbert-space* = *complex-inner* + *complete-space*
begin
subclass *cbanach* **by** *standard*
end

instantiation *complex* :: *chilbert-space* **begin**

instance ..
end

9.2 Misc facts

lemma *cinner-scaleR-left* [*simp*]: *cinner* (*scaleR r x*) *y* = *of-real r* ∗ (*cinner x y*)
by (*simp add*: *scaleR-scaleC*)

lemma *cinner-scaleR-right* [*simp*]: *cinner x* (*scaleR r y*) = *of-real r* ∗ (*cinner x y*)
by (*simp add*: *scaleR-scaleC*)

This is a useful rule for establishing the equality of vectors

lemma *cinner-extensionality*:
assumes ‹⋀γ. γ ·$_C$ ψ = γ ·$_C$ φ›
shows ‹ψ = φ›
by (*metis assms cinner-eq-zero-iff cinner-simps*(3) *right-minus-eq*)

lemma *polar-identity*:
includes *norm-syntax*
shows ‹‖*x* + *y*‖^2 = ‖*x*‖^2 + ‖*y*‖^2 + 2 ∗ *Re* (*x* ·$_C$ *y*)›
— Shown in the proof of Corollary 1.5 in [1]
proof −
have ‹(*x* ·$_C$ *y*) + (*y* ·$_C$ *x*) = (*x* ·$_C$ *y*) + *cnj* (*x* ·$_C$ *y*)›
by *simp*
hence ‹(*x* ·$_C$ *y*) + (*y* ·$_C$ *x*) = 2 ∗ *Re* (*x* ·$_C$ *y*) ›
using *complex-add-cnj* **by** *presburger*
have ‹‖*x* + *y*‖^2 = (*x*+*y*) ·$_C$ (*x*+*y*)›
by (*simp add*: *cdot-square-norm*)
hence ‹‖*x* + *y*‖^2 = (*x* ·$_C$ *x*) + (*x* ·$_C$ *y*) + (*y* ·$_C$ *x*) + (*y* ·$_C$ *y*)›
by (*simp add*: *cinner-add-left cinner-add-right*)
thus *?thesis* **using** ‹(*x* ·$_C$ *y*) + (*y* ·$_C$ *x*) = 2 ∗ *Re* (*x* ·$_C$ *y*)›
by (*smt* (*verit*, *ccfv-SIG*) *Re-complex-of-real plus-complex.simps*(1) *power2-norm-eq-cinner′*)
qed

lemma *polar-identity-minus*:
includes *norm-syntax*
shows ‹‖*x* − *y*‖^2 = ‖*x*‖^2 + ‖*y*‖^2 − 2 ∗ *Re* (*x* ·$_C$ *y*)›
proof−
have ‹‖*x* + (−*y*)‖^2 = ‖*x*‖^2 + ‖−*y*‖^2 + 2 ∗ *Re* (*x* ·$_C$ −*y*)›
using *polar-identity* **by** *blast*
hence ‹‖*x* − *y*‖^2 = ‖*x*‖^2 + ‖*y*‖^2 − 2∗*Re* (*x* ·$_C$ *y*)›
by *simp*
thus *?thesis*
by *blast*
qed

proposition *parallelogram-law*:
includes *norm-syntax*
fixes *x y* :: *′a*::*complex-inner*

shows ‹$\|x+y\|\hat{}2 + \|x-y\|\hat{}2 = 2*(\ \|x\|\hat{}2 + \|y\|\hat{}2\ )$›
— Shown in the proof of Theorem 2.3 in [1]
by (*simp add*: *polar-identity-minus polar-identity*)

theorem *pythagorean-theorem*:
includes *norm-syntax*
shows ‹$(x \cdot_C y) = 0 \Longrightarrow \|\ x + y\ \|\hat{}2 = \|\ x\ \|\hat{}2 + \|\ y\ \|\hat{}2$›
— Shown in the proof of Theorem 2.2 in [1]
by (*simp add*: *polar-identity*)

lemma *pythagorean-theorem-sum*:
assumes *q1*: $\bigwedge a\ a'.\ a \in t \Longrightarrow a' \in t \Longrightarrow a \neq a' \Longrightarrow f\ a \cdot_C f\ a' = 0$
and *q2*: *finite t*
shows $(norm\ \ (\sum a{\in}t.\ f\ a))\hat{}2 = (\sum a{\in}t.(norm\ (f\ a))\hat{}2)$
proof (*insert q1*, *use q2* **in** *induction*)
case *empty*
show *?case*
by *auto*
next
case (*insert x F*)
have *r1*: $f\ x \cdot_C f\ a = 0$
if $a \in F$
for *a*
using *that insert.hyps*(*2*) *insert.prems* **by** *auto*
have $sum\ f\ F = (\sum a{\in}F.\ f\ a)$
by *simp*
hence *s4*: $f\ x \cdot_C sum\ f\ F = f\ x \cdot_C (\sum a{\in}F.\ f\ a)$
by *simp*
also have *s3*: $\ldots = (\sum a{\in}F.\ f\ x \cdot_C f\ a)$
using *cinner-sum-right* **by** *auto*
also have *s2*: $\ldots = (\sum a{\in}F.\ 0)$
using *r1*
by *simp*
also have *s1*: $\ldots = 0$
by *simp*
finally have *xF-ortho*: $f\ x \cdot_C sum\ f\ F = 0$
using *s2 s3* **by** *auto*
have $(norm\ (sum\ f\ (insert\ x\ F)))^2 = (norm\ (f\ x + sum\ f\ F))^2$
by (*simp add*: *insert.hyps*(*1*) *insert.hyps*(*2*))
also have $\ldots = (norm\ (f\ x))^2 + (norm\ (sum\ f\ F))^2$
using *xF-ortho* **by** (*rule pythagorean-theorem*)
also have $\ldots = (norm\ (f\ x))^2 + (\sum a{\in}F.(norm\ (f\ a))\hat{}2)$
apply (*subst insert.IH*) **using** *insert.prems* **by** *auto*
also have $\ldots = (\sum a{\in}insert\ x\ F.(norm\ (f\ a))\hat{}2)$
by (*simp add*: *insert.hyps*(*1*) *insert.hyps*(*2*))
finally show *?case*
by *simp*
qed

lemma *Cauchy-cinner-Cauchy*:
fixes *x y* :: ‹*nat* ⇒ *'a*::*complex-inner*›
assumes *a1*: ‹*Cauchy x*› **and** *a2*: ‹*Cauchy y*›
shows ‹*Cauchy* (λ *n*. *x n* ·$_C$ *y n*)›
proof−
have ‹*bounded* (*range x*)›
using *a1*
by (*simp add*: *Elementary-Metric-Spaces.cauchy-imp-bounded*)
hence *b1*: ‹∃ *M*. ∀ *n*. *norm* (*x n*) < *M*›
by (*meson bounded-pos-less rangeI*)
have ‹*bounded* (*range y*)›
using *a2*
by (*simp add*: *Elementary-Metric-Spaces.cauchy-imp-bounded*)
hence *b2*: ‹∃ *M*. ∀ *n*. *norm* (*y n*) < *M*›
by (*meson bounded-pos-less rangeI*)
have ‹∃ *M*. ∀ *n*. *norm* (*x n*) < *M* ∧ *norm* (*y n*) < *M*›
using *b1 b2*
by (*metis dual-order.strict-trans linorder-neqE-linordered-idom*)
then obtain *M* **where** *M1*: ‹⋀*n*. *norm* (*x n*) < *M*› **and** *M2*: ‹⋀*n*. *norm* (*y n*) < *M*›
by *blast*
have *M3*: ‹*M* > *0*›
by (*smt M2 norm-not-less-zero*)
have ‹∃ *N*. ∀ *n* ≥ *N*. ∀ *m* ≥ *N*. *norm* ((λ *i*. *x i* ·$_C$ *y i*) *n* − (λ *i*. *x i* ·$_C$ *y i*) *m*) < *e*›
if *e* > *0* **for** *e*
proof−
have ‹*e* / (*2*∗*M*) > *0*›
using *M3*
by (*simp add*: *that*)
hence ‹∃ *N*. ∀ *n*≥*N*. ∀ *m*≥*N*. *norm* (*x n* − *x m*) < *e* / (*2*∗*M*)›
using *a1*
by (*simp add*: *Cauchy-iff*)
then obtain *N1* **where** *N1-def*: ‹⋀*n m*. *n*≥*N1* ⟹ *m*≥*N1* ⟹ *norm* (*x n* − *x m*) < *e* / (*2*∗*M*)›
by *blast*
have *x1*: ‹∃ *N*. ∀ *n*≥*N*. ∀ *m*≥*N*. *norm* (*y n* − *y m*) < *e* / (*2*∗*M*)›
using *a2* ‹*e* / (*2*∗*M*) > *0*›
by (*simp add*: *Cauchy-iff*)
obtain *N2* **where** *N2-def*: ‹⋀*n m*. *n*≥*N2* ⟹ *m*≥*N2* ⟹ *norm* (*y n* − *y m*) < *e* / (*2*∗*M*)›
using *x1*
by *blast*
define *N* **where** *N-def*: ‹*N* = *N1* + *N2*›
hence ‹*N* ≥ *N1*›
by *auto*
have ‹*N* ≥ *N2*›

```
    using N-def
    by auto
  have ‹norm (x n ·C y n − x m ·C y m) < e›
    if ‹n ≥ N› and ‹m ≥ N›
    for n m
  proof −
    have ‹x n ·C y n − x m ·C y m = (x n ·C y n − x m ·C y n) + (x m ·C y n
− x m ·C y m)›
      by simp
    hence y1: ‹norm (x n ·C y n − x m ·C y m) ≤ norm (x n ·C y n − x m ·C
y n)
        + norm (x m ·C y n − x m ·C y m)›
      by (metis norm-triangle-ineq)

    have ‹x n ·C y n − x m ·C y n = (x n − x m) ·C y n›
      by (simp add: cinner-diff-left)
    hence ‹norm (x n ·C y n − x m ·C y n) = norm ((x n − x m) ·C y n)›
      by simp
    moreover have ‹norm ((x n − x m) ·C y n) ≤ norm (x n − x m) ∗ norm
(y n)›
      using complex-inner-class.Cauchy-Schwarz-ineq2 by blast
    moreover have ‹norm (y n) < M›
      by (simp add: M2)
    moreover have ‹norm (x n − x m) < e/(2∗M)›
      using ‹N ≤ m› ‹N ≤ n› ‹N1 ≤ N› N1-def by auto
    ultimately have ‹norm ((x n ·C y n) − (x m ·C y n)) < (e/(2∗M)) ∗ M›
      by (smt mult-strict-mono norm-ge-zero)
    moreover have ‹ (e/(2∗M)) ∗ M = e/2›
      using ‹M > 0› by simp
    ultimately have  ‹norm ((x n ·C y n) − (x m ·C y n)) < e/2›
      by simp
    hence y2: ‹norm (x n ·C y n − x m ·C y n) < e/2›
      by blast
    have ‹x m ·C y n − x m ·C y m = x m ·C (y n − y m)›
      by (simp add: cinner-diff-right)
    hence ‹norm ((x m ·C y n) − (x m ·C y m)) = norm (x m ·C (y n − y m))›
      by simp
    moreover have ‹norm (x m ·C (y n − y m)) ≤ norm (x m) ∗ norm (y n −
y m)›
      by (meson complex-inner-class.Cauchy-Schwarz-ineq2)
    moreover have ‹norm (x m) < M›
      by (simp add: M1)
    moreover have ‹norm (y n − y m) < e/(2∗M)›
      using ‹N ≤ m› ‹N ≤ n› ‹N2 ≤ N› N2-def by auto
    ultimately have ‹norm ((x m ·C y n) − (x m ·C y m)) < M ∗ (e/(2∗M))›
      by (smt mult-strict-mono norm-ge-zero)
    moreover have ‹M ∗ (e/(2∗M)) = e/2›
      using ‹M > 0› by simp
    ultimately have  ‹norm ((x m ·C y n) − (x m ·C y m)) < e/2›
```

```
      by simp
    hence y3: ‹norm ((x m ·C y n) − (x m ·C y m)) < e/2›
      by blast
    show ‹norm ( (x n ·C y n) − (x m ·C y m) ) < e›
      using y1 y2 y3 by simp
  qed
  thus ?thesis by blast
 qed
 thus ?thesis
  by (simp add: CauchyI)
qed


lemma cinner-sup-norm: ‹norm ψ = (SUP φ. cmod (cinner φ ψ) / norm φ)›
proof (rule sym, rule cSup-eq-maximum)
 have ‹norm ψ = cmod (cinner ψ ψ) / norm ψ›
  by (metis norm-eq-sqrt-cinner norm-ge-zero real-div-sqrt)
 then show ‹norm ψ ∈ range (λφ. cmod (cinner φ ψ) / norm φ)›
  by blast
next
 fix n assume ‹n ∈ range (λφ. cmod (cinner φ ψ) / norm φ)›
 then obtain φ where nφ: ‹n = cmod (cinner φ ψ) / norm φ›
  by auto
 show ‹n ≤ norm ψ›
  unfolding nφ
  by (simp add: Cauchy-Schwarz-ineq2 mult.commute divide-le-eq)
qed

lemma cinner-sup-onorm:
 fixes A :: ‹'a::{real-normed-vector,not-singleton} ⇒ 'b::complex-inner›
 assumes ‹bounded-linear A›
 shows ‹onorm A = (SUP (ψ,φ). cmod (cinner ψ (A φ)) / (norm ψ * norm φ))›
proof (unfold onorm-def, rule cSup-eq-cSup)
 show ‹bdd-above (range (λx. norm (A x) / norm x))›
  by (meson assms bdd-aboveI2 le-onorm)
next
 fix a
 assume ‹a ∈ range (λφ. norm (A φ) / norm φ)›
 then obtain φ where ‹a = norm (A φ) / norm φ›
  by auto
 then have ‹a ≤ cmod (cinner (A φ) (A φ)) / (norm (A φ) * norm φ)›
  apply auto
  by (smt (verit) divide-divide-eq-left norm-eq-sqrt-cinner norm-imp-pos-and-ge
real-div-sqrt)
 then show ‹∃ b∈range (λ(ψ, φ). cmod (cinner ψ (A φ)) / (norm ψ * norm φ)).
a ≤ b›
  by force
next
 fix b
```

assume ‹$b \in$ *range* ($\lambda(\psi, \varphi)$. *cmod* (ψ ·$_C$ A φ) / (*norm* ψ * *norm* φ))›
then obtain ψ φ **where** *b*: ‹b = *cmod* (ψ ·$_C$ A φ) / (*norm* ψ * *norm* φ)›
by *auto*
then have ‹$b \leq$ *norm* (A φ) / *norm* φ›
using *Cauchy-Schwarz-ineq2*[*of* ψ A φ]
apply (*simp add*: *divide-simps split*: *if-split-asm*)
by (*metis mult-le-cancel-left-pos mult.left-commute zero-less-norm-iff*)
then show ‹$\exists$ a∈*range* (λx. *norm* (A x) / *norm* x). $b \leq a$›
by *auto*
qed

lemma *sum-cinner*:
fixes f :: $'a \Rightarrow$ $'b$::*complex-inner*
shows *cinner* (*sum f A*) (*sum g B*) = ($\sum i{\in}A$. $\sum j{\in}B$. *cinner* (*f i*) (*g j*))
by (*simp add*: *cinner-sum-right cinner-sum-left*) (*rule sum.swap*)

lemma *Cauchy-cinner-product-summable'*:
fixes *a b* :: *nat* ⇒ $'a$::*complex-inner*
shows ‹($\lambda(x, y)$. *cinner* (*a x*) (*b y*)) *summable-on UNIV* ⟷ ($\lambda(x, y)$. *cinner* (*a y*) (*b* (*x* − *y*))) *summable-on* {(*k*, *i*). *i* ≤ *k*}›
proof −
have *img*: ‹(λ(*k*::*nat*, *i*). (*i*, *k* − *i*)) ‘ {(*k*, *i*). *i* ≤ *k*} = *UNIV*›
apply (*auto simp*: *image-def*)
by (*metis add.commute add-diff-cancel-right' diff-le-self*)
have *inj*: ‹*inj-on* (λ(*k*::*nat*, *i*). (*i*, *k* − *i*)) {(*k*, *i*). *i* ≤ *k*}›
by (*smt* (*verit*, *del-insts*) *Pair-inject case-prodE case-prod-conv eq-diff-iff inj-onI mem-Collect-eq*)

have ‹($\lambda(x, y)$. *cinner* (*a x*) (*b y*)) *summable-on UNIV* ⟷ ($\lambda(k, l)$. *cinner* (*a k*) (*b l*)) *summable-on* ($\lambda(k, i)$. (*i*, *k* − *i*)) ‘ {(*k*, *i*). *i* ≤ *k*}›
by (*simp only*: *img*)
also have ‹... ⟷ (($\lambda(k, l)$. *cinner* (*a k*) (*b l*)) ∘ ($\lambda(k, i)$. (*i*, *k* − *i*))) *summable-on* {(*k*, *i*). *i* ≤ *k*}›
using *inj* **by** (*rule summable-on-reindex*)
also have ‹... ⟷ ($\lambda(x, y)$. *cinner* (*a y*) (*b* (*x* − *y*))) *summable-on* {(*k*, *i*). *i* ≤ *k*}›
by (*simp add*: *o-def case-prod-unfold*)
finally show *?thesis*
by −
qed

instantiation *prod* :: (*complex-inner*, *complex-inner*) *complex-inner*
begin

definition *cinner-prod-def*:
cinner x y = *cinner* (*fst x*) (*fst y*) + *cinner* (*snd x*) (*snd y*)

instance

proof
fix r :: *complex*
fix $x\ y\ z$ :: *'a::complex-inner* × *'b::complex-inner*
show *cinner x y = cnj (cinner y x)*
unfolding *cinner-prod-def*
by *simp*
show *cinner (x + y) z = cinner x z + cinner y z*
unfolding *cinner-prod-def*
by (*simp add*: *cinner-add-left*)
show *cinner (scaleC r x) y = cnj r * cinner x y*
unfolding *cinner-prod-def*
by (*simp add*: *distrib-left*)
show *0 ≤ cinner x x*
unfolding *cinner-prod-def*
by (*intro add-nonneg-nonneg cinner-ge-zero*)
show *cinner x x = 0 ⟷ x = 0*
unfolding *cinner-prod-def prod-eq-iff*
by (*metis antisym cinner-eq-zero-iff cinner-ge-zero fst-zero le-add-same-cancel2 snd-zero verit-sum-simplify*)
show *norm x = sqrt (cmod (cinner x x))*
unfolding *norm-prod-def cinner-prod-def*
apply (*simp add*: *norm-prod-def cinner-prod-def*)
by (*metis* (*no-types*, *lifting*) *Complex-Inner-Product.cinner-prod-def Re-complex-of-real ‹0 ≤ x ·C x› cmod-Re of-real-add of-real-power power2-norm-eq-cinner*)
qed

end

lemma *sgn-cinner*[*simp*]: *‹sgn ψ ·C ψ = norm ψ›*
apply (*cases ‹ψ = 0›*)
apply (*auto simp*: *sgn-div-norm*)
by (*metis* (*no-types*, *lifting*) *cdot-square-norm cinner-ge-zero cmod-Re divide-inverse mult.commute norm-eq-sqrt-cinner norm-ge-zero of-real-inverse of-real-mult power2-norm-eq-cinner′ real-div-sqrt*)

instance *prod* :: (*chilbert-space*, *chilbert-space*) *chilbert-space*..

9.3 Orthogonality

definition *orthogonal-complement S* = {*x*| *x*. ∀ *y*∈*S*. *cinner x y* = *0*}

lemma *orthogonal-complement-orthoI*:
‹x ∈ orthogonal-complement M ⟹ y ∈ M ⟹ x ·C y = 0›
unfolding *orthogonal-complement-def* **by** *auto*

lemma *orthogonal-complement-orthoI′*:
‹x ∈ M ⟹ y ∈ orthogonal-complement M ⟹ x ·C y = 0›
by (*metis cinner-commute′ complex-cnj-zero orthogonal-complement-orthoI*)

lemma *orthogonal-complementI*:
‹$(\bigwedge x.\ x \in M \Longrightarrow y \cdot_C x = 0) \Longrightarrow y \in$ *orthogonal-complement* M›
unfolding *orthogonal-complement-def*
by *simp*

abbreviation *is-orthogonal*::‹'*a*::*complex-inner* ⇒ '*a* ⇒ *bool*› **where**
‹*is-orthogonal* $x\ y \equiv x \cdot_C y = 0$›

bundle *orthogonal-syntax*
begin
notation *is-orthogonal* (**infixl** ‹⊥› *69*)
end

lemma *is-orthogonal-sym*: *is-orthogonal* $\psi\ \varphi =$ *is-orthogonal* $\varphi\ \psi$
by (*metis cinner-commute' complex-cnj-zero*)

lemma *is-orthogonal-sgn-right*[*simp*]: ‹*is-orthogonal* e (*sgn* f) ⟷ *is-orthogonal* e f›
proof (*cases* ‹$f = 0$›)
case *True*
then show *?thesis*
by *simp*
next
case *False*
have ‹*cinner* e (*sgn* f) = *cinner* $e\ f$ / *norm* f›
by (*simp add*: *sgn-div-norm divide-inverse scaleR-scaleC*)
moreover have ‹*norm* $f \neq 0$›
by (*simp add*: *False*)
ultimately show *?thesis*
by *force*
qed

lemma *is-orthogonal-sgn-left*[*simp*]: ‹*is-orthogonal* (*sgn* e) f ⟷ *is-orthogonal* e f›
by (*simp add*: *is-orthogonal-sym*)

lemma *orthogonal-complement-closed-subspace*[*simp*]:
closed-csubspace (*orthogonal-complement* A)
for A :: ‹('*a*::*complex-inner*) *set*›
proof (*intro closed-csubspace.intro complex-vector.subspaceI*)
fix $x\ y$ **and** c
show ‹$0 \in$ *orthogonal-complement* A›
by (*rule orthogonal-complementI*, *simp*)
show ‹$x + y \in$ *orthogonal-complement* A›
if ‹$x \in$ *orthogonal-complement* A› **and** ‹$y \in$ *orthogonal-complement* A›
using *that* **by** (*auto intro*!: *orthogonal-complementI dest*!: *orthogonal-complement-orthoI*
simp add: *cinner-add-left*)
show ‹$c *_C x \in$ *orthogonal-complement* A› **if** ‹$x \in$ *orthogonal-complement* A›
using *that* **by** (*auto intro*!: *orthogonal-complementI dest*!: *orthogonal-complement-orthoI*)

```
  show closed (orthogonal-complement A)
  proof (auto simp add: closed-sequential-limits, rename-tac an a)
    fix an a
    assume ortho: ‹∀ n::nat. an n ∈ orthogonal-complement A›
    assume lim: ‹an ⟶ a›

    have ‹∀ y ∈ A. ∀ n. is-orthogonal y (an n)›
      using orthogonal-complement-orthoI'
      by (simp add: orthogonal-complement-orthoI' ortho)
    moreover have ‹isCont (λ x. y ⋅C x) a› for y
      using bounded-clinear-cinner-right clinear-continuous-at
      by (simp add: clinear-continuous-at bounded-clinear-cinner-right)
    ultimately have ‹(λ n. (λ v. y ⋅C v) (an n)) ⟶ (λ v. y ⋅C v) a› for y
      using isCont-tendsto-compose
      by (simp add: isCont-tendsto-compose lim)
    hence ‹∀ y∈A. (λ n. y ⋅C an n) ⟶ y ⋅C a›
      by simp
    hence ‹∀ y∈A. (λ n. 0) ⟶ y ⋅C a›
      using ‹∀ y ∈ A. ∀ n. is-orthogonal y (an n)›
      by fastforce
    hence ‹∀ y ∈ A. is-orthogonal y a›
      using limI by fastforce
    then show ‹a ∈ orthogonal-complement A›
      by (simp add: orthogonal-complementI is-orthogonal-sym)
  qed
qed

lemma orthogonal-complement-zero-intersection:
  assumes 0∈M
  shows ‹M ∩ (orthogonal-complement M) = {0}›
proof −
  have x=0 if x∈M and x∈orthogonal-complement M for x
  proof −
    from that have is-orthogonal x x
      unfolding orthogonal-complement-def by auto
    thus x=0
      by auto
  qed
  with assms show ?thesis
    unfolding orthogonal-complement-def by auto
qed

lemma is-orthogonal-closure-cspan:
  assumes ⋀x y. x ∈ X ⟹ y ∈ Y ⟹ is-orthogonal x y
  assumes ‹x ∈ closure (cspan X)› ‹y ∈ closure (cspan Y)›
  shows is-orthogonal x y
proof −
  have ∗: ‹cinner x y = 0› if ‹y ∈ Y› for y
```

using *bounded-antilinear-cinner-left* **apply** (*rule bounded-antilinear-eq-on*[**where** *G*=*X*])
using *assms that* **by** *auto*
show ‹*cinner x y = 0*›
using *bounded-clinear-cinner-right* **apply** (*rule bounded-clinear-eq-on-closure*[**where** *G*=*Y*])
using ∗ *assms* **by** *auto*
qed

instantiation *ccsubspace* :: (*complex-inner*) *uminus*
begin
lift-definition *uminus-ccsubspace*::‹*'a ccsubspace ⇒ 'a ccsubspace*›
is ‹*orthogonal-complement*›
by *simp*

instance ..
end

lemma *orthocomplement-top*[*simp*]: ‹*− top* = (*bot* :: *'a*::*complex-inner ccsubspace*)›
— For *'a* of sort *chilbert-space*, this is covered by *orthocomplemented-lattice-class.compl-top-eq* already. But here we give it a wider sort.
apply *transfer*
by (*metis Int-UNIV-left UNIV-I orthogonal-complement-zero-intersection*)

instantiation *ccsubspace* :: (*complex-inner*) *minus* **begin**
lift-definition *minus-ccsubspace* :: *'a ccsubspace ⇒ 'a ccsubspace ⇒ 'a ccsubspace*
is *λA B. A ∩* (*orthogonal-complement B*)
by *simp*
instance..
end

definition *is-ortho-set* :: *'a*::*complex-inner set ⇒ bool* **where**
— Orthogonal set
‹*is-ortho-set S ⟷* (*∀ x*∈*S. ∀ y*∈*S. x ≠ y ⟶* (*x ·C y*) = *0*) *∧ 0 ∉ S*›

definition *is-onb* :: *'a*::*complex-inner set ⇒ bool* **where**
— Orthonormal basis
‹*is-onb E ⟷ is-ortho-set E ∧* (*∀ b*∈*E. norm b = 1*) *∧ ccspan E = top*›

lemma *is-ortho-set-empty*[*simp*]: *is-ortho-set* {}
unfolding *is-ortho-set-def* **by** *auto*

lemma *is-ortho-set-antimono*: ‹*A* ⊆ *B ⟹ is-ortho-set B ⟹ is-ortho-set A*›
unfolding *is-ortho-set-def* **by** *auto*

lemma *orthogonal-complement-of-closure*:
fixes *A* ::(*'a*::*complex-inner*) *set*
shows *orthogonal-complement A = orthogonal-complement* (*closure A*)

proof−
have *s1*: ‹*is-orthogonal y x*›
if *a1*: $x \in$ (*orthogonal-complement A*)
and *a2*: ‹$y \in$ *closure A*›
for *x y*
proof−
have ‹$\forall\ y \in A$. *is-orthogonal y x*›
by (*simp add*: *a1 orthogonal-complement-orthoI'*)
then obtain *yy* **where** ‹$\forall\ n$. *yy n* $\in A$› **and** ‹*yy* $\longrightarrow$ *y*›
using *a2 closure-sequential* **by** *blast*
have ‹*isCont* (λ *t*. *t* $\cdot_C$ *x*) *y*›
by *simp*
hence ‹(λ *n*. *yy n* $\cdot_C$ *x*) $\longrightarrow$ *y* $\cdot_C$ *x*›
using ‹*yy* $\longrightarrow$ *y*› *isCont-tendsto-compose*
by *fastforce*
hence ‹(λ *n*. *0*) $\longrightarrow$ *y* $\cdot_C$ *x*›
using ‹$\forall\ y \in A$. *is-orthogonal y x*› ‹$\forall\ n$. *yy n* $\in A$› **by** *simp*
thus *?thesis*
using *limI* **by** *force*
qed
hence *x* $\in$ *orthogonal-complement* (*closure A*)
if *a1*: $x \in$ (*orthogonal-complement A*)
for *x*
using *that*
by (*meson orthogonal-complementI is-orthogonal-sym*)
moreover have ‹$x \in$ (*orthogonal-complement A*)›
if $x \in$ (*orthogonal-complement* (*closure A*))
for *x*
using *that*
by (*meson closure-subset orthogonal-complement-orthoI orthogonal-complementI subset-eq*)
ultimately show *?thesis* **by** *blast*
qed

lemma *is-orthogonal-closure*:
assumes ‹$\bigwedge s$. $s \in S \Longrightarrow$ *is-orthogonal a s*›
assumes ‹$x \in$ *closure S*›
shows ‹*is-orthogonal a x*›
by (*metis assms*(*1*) *assms*(*2*) *orthogonal-complementI orthogonal-complement-of-closure orthogonal-complement-orthoI*)

lemma *is-orthogonal-cspan*:
assumes *a1*: $\bigwedge s$. $s \in S \Longrightarrow$ *is-orthogonal a s* **and** *a3*: $x \in$ *cspan S*
shows *is-orthogonal a x*
proof−
have $\exists\, t\ r$. *finite* $t \wedge t \subseteq S \wedge (\sum a{\in}t.\ r\ a *_C a) = x$
using *complex-vector.span-explicit*

by (*smt a3 mem-Collect-eq*)
then obtain *t r* **where** *b1*: *finite t* **and** *b2*: $t \subseteq S$ **and** *b3*: $(\sum a{\in}t.\ r\ a *_C a) = x$
by *blast*
have *x1*: *is-orthogonal a i*
if $i{\in}t$ **for** *i*
using *b2 a1 that* **by** *blast*
have $a \cdot_C x = a \cdot_C (\sum i{\in}t.\ r\ i *_C i)$
by (*simp add*: *b3*)
also have $\ldots = (\sum i{\in}t.\ r\ i *_C (a \cdot_C i))$
by (*simp add*: *cinner-sum-right*)
also have $\ldots = 0$
using *x1* **by** *simp*
finally show *?thesis*.
qed

lemma *ccspan-leq-ortho-ccspan*:
assumes $\bigwedge s\ t.\ s{\in}S \Longrightarrow t{\in}T \Longrightarrow$ *is-orthogonal s t*
shows *ccspan S* $\leq -$ (*ccspan T*)
using *assms* **apply** *transfer*
by (*smt* (*verit*, *ccfv-threshold*) *is-orthogonal-closure is-orthogonal-cspan is-orthogonal-sym orthogonal-complementI subsetI*)

lemma *double-orthogonal-complement-increasing*[*simp*]:
shows $M \subseteq$ *orthogonal-complement* (*orthogonal-complement M*)
proof (*rule subsetI*)
fix *x* **assume** *s1*: $x \in M$
have ‹$\forall\ y \in$ (*orthogonal-complement M*). *is-orthogonal x y*›
using *s1 orthogonal-complement-orthoI'* **by** *auto*
hence ‹$x \in$ *orthogonal-complement* (*orthogonal-complement M*)›
by (*simp add*: *orthogonal-complement-def*)
then show $x \in$ *orthogonal-complement* (*orthogonal-complement M*)
by *blast*
qed

lemma *orthonormal-basis-of-cspan*:
fixes *S*::'*a*::*complex-inner set*
assumes *finite S*
shows $\exists A.$ *is-ortho-set A* $\wedge$ $(\forall x{\in}A.$ *norm x* $= 1)$ $\wedge$ *cspan A* $=$ *cspan S* $\wedge$ *finite A*
proof (*use assms* **in** *induction*)
case *empty*
show *?case*
apply (*rule exI*[*of* - {}])
by *auto*
next
case (*insert s S*)
from *insert.IH*

```
  obtain A where orthoA: is-ortho-set A and normA: ⋀x. x∈A ⟹ norm x = 1
and spanA: cspan A = cspan S and finiteA: finite A
    by auto
  show ?case
  proof (cases ‹s ∈ cspan S›)
    case True
    then have ‹cspan (insert s S) = cspan S›
      by (simp add: complex-vector.span-redundant)
    with orthoA normA spanA finiteA
    show ?thesis
      by auto
  next
    case False
    obtain a where a-ortho: ‹⋀x. x∈A ⟹ is-orthogonal x a› and sa-span: ‹s −
a ∈ cspan A›
    proof (atomize-elim, use ‹finite A› ‹is-ortho-set A› in induction)
      case empty
      then show ?case
        by auto
    next
      case (insert x A)
      then obtain a where orthoA: ‹⋀x. x ∈ A ⟹ is-orthogonal x a› and sa: ‹s
− a ∈ cspan A›
        by (meson is-ortho-set-antimono subset-insertI)
      define a' where ‹a' = a − cinner x a *C inverse (cinner x x) *C x›
      have ‹is-orthogonal x a'›
        unfolding a'-def cinner-diff-right cinner-scaleC-right
        apply (cases ‹cinner x x = 0›)
        by auto
      have orthoA: ‹is-orthogonal y a'› if ‹y ∈ A› for y
        unfolding a'-def cinner-diff-right cinner-scaleC-right
         apply auto by (metis insert.prems insertCI is-ortho-set-def mult-not-zero
orthoA that)
      have ‹s − a' ∈ cspan (insert x A)›
        unfolding a'-def apply auto
       by (metis (no-types, lifting) complex-vector.span-breakdown-eq diff-add-cancel
diff-diff-add sa)
      with ‹is-orthogonal x a'› orthoA
      show ?case
        apply (rule-tac exI[of - a'])
        by auto
    qed

    from False sa-span
    have ‹a ≠ 0›
      unfolding spanA by auto
    define a' where ‹a' = inverse (norm a) *C a›
    with ‹a ≠ 0› have ‹norm a' = 1›
      by (simp add: norm-inverse)
```

```
    have a: ‹a = norm a *C a'›
      by (simp add: ‹a ≠ 0› a'-def)

    from sa-span spanA
    have a'-span: ‹a' ∈ cspan (insert s S)›
      unfolding a'-def
        by (metis complex-vector.eq-span-insert-eq complex-vector.span-scale com-
plex-vector.span-superset in-mono insertI1)
    from sa-span
    have s-span: ‹s ∈ cspan (insert a' A)›
      apply (subst (asm) a)
      using complex-vector.span-breakdown-eq by blast

    from ‹a ≠ 0› a-ortho orthoA
    have ortho: is-ortho-set (insert a' A)
      unfolding is-ortho-set-def a'-def
      apply auto
      by (meson is-orthogonal-sym)

    have span: ‹cspan (insert a' A) = cspan (insert s S)›
      using a'-span s-span spanA apply auto
     apply (metis (full-types) complex-vector.span-breakdown-eq complex-vector.span-redundant
insert-commute s-span)
    by (metis (full-types) complex-vector.span-breakdown-eq complex-vector.span-redundant
insert-commute s-span)

    show ?thesis
      apply (rule exI[of - ‹insert a' A›])
      by (simp add: ortho ‹norm a' = 1› normA finiteA span)
  qed
qed

lemma is-ortho-set-cindependent:
  assumes is-ortho-set A
  shows cindependent A
proof −
  have u v = 0
    if b1: finite t and b2: t ⊆ A and b3: (∑ v∈t. u v *C v) = 0 and b4: v ∈ t
    for t u v
  proof −
    have is-orthogonal v v' if c1: v'∈t−{v} for v'
      by (metis DiffE assms b2 b4 insertI1 is-ortho-set-antimono is-ortho-set-def
that)
    hence sum0: (∑ v'∈t−{v}. u v' * (v ·C v')) = 0
      by simp
    have v ·C (∑ v'∈t. u v' *C v') = (∑ v'∈t. u v' * (v ·C v'))
      using b1
     by (metis (mono-tags, lifting) cinner-scaleC-right cinner-sum-right sum.cong)
    also have ... = u v * (v ·C v) + (∑ v'∈t−{v}. u v' * (v ·C v'))
```

```
    by (meson b1 b4 sum.remove)
   also have ... = u v * (v ·C v)
    using sum0 by simp
   finally have v ·C (∑ v'∈t. u v' *C v') =  u v * (v ·C v)
    by blast
   hence u v * (v ·C v) = 0 using b3 by simp
   moreover have (v ·C v) ≠ 0
    using assms is-ortho-set-def b2 b4 by auto
   ultimately show u v = 0 by simp
  qed
  thus ?thesis using complex-vector.independent-explicit-module
   by (smt cdependent-raw-def)
qed


lemma onb-expansion-finite:
  includes norm-syntax
  fixes T::‹'a::{complex-inner,cfinite-dim} set›
  assumes a1: ‹cspan T = UNIV› and a3: ‹is-ortho-set T›
   and a4: ‹⋀t. t∈T ⟹ ‖t‖ = 1›
  shows ‹x = (∑ t∈T. (t ·C x) *C t)›
proof −
  have ‹finite T›
   apply (rule cindependent-cfinite-dim-finite)
   by (simp add: a3 is-ortho-set-cindependent)
  have ‹closure (complex-vector.span T) = complex-vector.span T›
   by (simp add: a1)
  have ‹{∑ a∈t. r a *C a |t r. finite t ∧ t ⊆ T} = {∑ a∈T. r a *C a |r. True}›
   apply auto
    apply (rule-tac x=‹λa. if a ∈ t then r a else 0› in exI)
    apply (simp add: ‹finite T› sum.mono-neutral-cong-right)
   using ‹finite T› by blast

  have f1: ∀A. {a. ∃Aa f. (a::'a) = (∑ a∈Aa. f a *C a) ∧ finite Aa ∧ Aa ⊆ A}
= cspan A
   by (simp add: complex-vector.span-explicit)
  have f2: ∀a. (∃f. a = (∑ a∈T. f a *C a)) ∨ (∀A. (∀f. a ≠ (∑ a∈A. f a *C a))
∨ infinite A ∨ ¬ A ⊆ T)
   using ‹{∑ a∈t. r a *C a |t r. finite t ∧ t ⊆ T} = {∑ a∈T. r a *C a |r. True}›
by auto
  have f3: ∀A a. (∃Aa f. (a::'a) = (∑ a∈Aa. f a *C a) ∧ finite Aa ∧ Aa ⊆ A) ∨
a ∉ cspan A
   using f1 by blast
  have cspan T = UNIV
   by (metis (full-types, lifting) ‹complex-vector.span T = UNIV›)
  hence ‹∃ r. x = (∑ a∈T. r a *C a)›
   using f3 f2 by blast
  then obtain r where ‹x = (∑ a∈T. r a *C a)›
   by blast
```

```
  have ‹r a = a ·C x› if ‹a ∈ T› for a
  proof−
    have ‹norm a = 1›
      using a4
      by (simp add: ‹a ∈ T›)
    moreover have ‹norm a = sqrt (norm (a ·C a))›
      using norm-eq-sqrt-cinner by auto
    ultimately have ‹sqrt (norm (a ·C a)) = 1›
      by simp
    hence ‹norm (a ·C a) = 1›
      using real-sqrt-eq-1-iff by blast
    moreover have ‹(a ·C a) ∈ ℝ›
      by (simp add: cinner-real)
    moreover have ‹(a ·C a) ≥ 0›
      using cinner-ge-zero by blast
    ultimately have w1: ‹(a ·C a) = 1›
      using ‹‖a‖ = 1› cnorm-eq-1 by blast
    have ‹r t * (a ·C t) = 0› if ‹t ∈ T−{a}› for t
      by (metis DiffD1 DiffD2 ‹a ∈ T› a3 is-ortho-set-def mult-eq-0-iff singletonI
that)
    hence s1: ‹(∑ t∈T−{a}. r t * (a ·C t)) = 0›
      by (simp add: ‹⋀t. t ∈ T − {a} ⟹ r t * (a ·C t) = 0›)
    have ‹(a ·C x) = a ·C (∑ t∈T. r t *C t)›
      using ‹x = (∑ a∈T. r a *C a)›
      by simp
    also have ‹... = (∑ t∈T. a ·C (r t *C t))›
      using cinner-sum-right by blast
    also have ‹... = (∑ t∈T. r t * (a ·C t))›
      by simp
    also have ‹... = r a * (a ·C a) + (∑ t∈T−{a}. r t * (a ·C t))›
      using ‹a ∈ T›
      by (meson ‹finite T› sum.remove)
    also have ‹... = r a * (a ·C a)›
      using s1
      by simp
    also have ‹... = r a›
      by (simp add: w1)
    finally show ?thesis by auto
  qed
  thus ?thesis
    using ‹x = (∑ a∈T. r a *C a)›
    by fastforce
qed

lemma is-ortho-set-singleton[simp]: ‹is-ortho-set {x} ⟷ x ≠ 0›
  by (simp add: is-ortho-set-def)

lemma orthogonal-complement-antimono[simp]:
```

fixes *A B* :: ‹('*a*::*complex-inner*) *set*›
assumes $A \supseteq B$
shows ‹*orthogonal-complement A* ⊆ *orthogonal-complement B*›
by (*meson assms orthogonal-complementI orthogonal-complement-orthoI' subsetD subsetI*)

lemma *orthogonal-complement-UNIV*[*simp*]:
orthogonal-complement UNIV = {*0*}
by (*metis Int-UNIV-left complex-vector.subspace-UNIV complex-vector.subspace-def orthogonal-complement-zero-intersection*)

lemma *orthogonal-complement-zero*[*simp*]:
orthogonal-complement {*0*} = *UNIV*
unfolding *orthogonal-complement-def* **by** *auto*

lemma *mem-ortho-ccspanI*:
assumes ‹⋀*y*. *y* ∈ *S* ⟹ *is-orthogonal x y*›
shows ‹*x* ∈ *space-as-set* (− *ccspan S*)›
proof −
have ‹*x* ∈ *space-as-set* (*ccspan* {*x*})›
using *ccspan-superset* **by** *blast*
also have ‹... ⊆ *space-as-set* (− *ccspan S*)›
apply (*simp add*: *flip*: *less-eq-ccsubspace.rep-eq*)
apply (*rule ccspan-leq-ortho-ccspan*)
using *assms* **by** *auto*
finally show *?thesis*
by −
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

9.4 Projections

lemma *smallest-norm-exists*:
— Theorem 2.5 in [1] (inside the proof)
includes *norm-syntax*
fixes M :: ‹*'a*::*chilbert-space set*›
assumes *q1*: ‹*convex M*› **and** *q2*: ‹*closed M*› **and** *q3*: ‹$M \neq \{\}$›
shows ‹$\exists k.$ *is-arg-min* $(\lambda\ x.\ \|x\|)$ $(\lambda\ t.\ t \in M)$ *k*›
proof −
define *d* **where** ‹d = *Inf* $\{\ \|x\|\hat{}2 \mid x.\ x \in M\ \}$›
have *w4*: ‹$\{\ \|x\|\hat{}2 \mid x.\ x \in M\ \} \neq \{\}$›
by (*simp add*: *assms*(*3*))
have ‹$\forall\ x.\ \|x\|\hat{}2 \geq 0$›
by *simp*
hence *bdd-below1*: ‹*bdd-below* $\{\ \|x\|\hat{}2 \mid x.\ x \in M\ \}$›
by *fastforce*
have ‹$d \leq \|x\|\hat{}2$› **if** *a1*: $x \in M$ **for** x
proof−
have $\forall\, v.\ (\exists\, w.\ Re\ (v \cdot_C v) = \|w\|^2 \wedge w \in M) \vee v \notin M$
by (*metis* (*no-types*) *power2-norm-eq-cinner'*)
hence *Re* $(x \cdot_C x) \in \{\|v\|^2 \mid v.\ v \in M\}$
using *a1* **by** *blast*
thus *?thesis*
unfolding *d-def*
by (*metis* (*lifting*) *bdd-below1 cInf-lower power2-norm-eq-cinner'*)
qed

have ‹$\forall\ \varepsilon > 0.\ \exists\ t \in \{\ \|x\|\hat{}2 \mid x.\ x \in M\ \}.\ t < d + \varepsilon$›
unfolding *d-def*
using *w4 bdd-below1*
by (*meson cInf-lessD less-add-same-cancel1*)
hence ‹$\forall\ \varepsilon > 0.\ \exists\ x \in M.\ \|x\|\hat{}2 < d + \varepsilon$›
by *auto*
hence ‹$\forall\ \varepsilon > 0.\ \exists\ x \in M.\ \|x\|\hat{}2 < d + \varepsilon$›
by (*simp add*: ‹$\bigwedge x.\ x \in M \Longrightarrow d \leq \|x\|^2$›)
hence *w1*: ‹$\forall\ n$::*nat*. $\exists\ x \in M.\ \|x\|\hat{}2 < d + 1/(n+1)$› **by** *auto*

then obtain *r*::‹*nat* ⇒ *'a*› **where** *w2*: ‹$\forall\ n.\ r\ n \in M \wedge\ \|\ r\ n\ \|\hat{}2 < d + 1/(n+1)$›
by *metis*
have *w3*: ‹$\forall\ n.\ r\ n \in M$›
by (*simp add*: *w2*)
have ‹$\forall\ n.\ \|\ r\ n\ \|\hat{}2 < d + 1/(n+1)$›
by (*simp add*: *w2*)
have *w5*: ‹$\|\ (r\ n) - (r\ m)\ \|\hat{}2 < 2*(1/(n+1) + 1/(m+1))$›
for *m n*
proof−
have *w6*: ‹$\|\ r\ n\ \|\hat{}2 < d + 1/(n+1)$›
by (*metis w2 of-nat-1 of-nat-add*)
have ‹ $\|\ r\ m\ \|\hat{}2 < d + 1/(m+1)$›

by (*metis w2 of-nat-1 of-nat-add*)
have ‹(*r n*) ∈ *M*›
by (*simp add*: ‹∀ *n*. *r n* ∈ *M*›)
moreover have ‹(*r m*) ∈ *M*›
by (*simp add*: ‹∀ *n*. *r n* ∈ *M*›)
ultimately have ‹(*1/2*) ∗*R* (*r n*) + (*1/2*) ∗*R* (*r m*) ∈ *M*›
using ‹*convex M*›
by (*simp add*: *convexD*)
hence ‹‖ (*1/2*) ∗*R* (*r n*) + (*1/2*) ∗*R* (*r m*) ‖^2 ≥ *d*›
by (*simp add*: ‹⋀*x*. *x* ∈ *M* ⟹ *d* ≤ ‖*x*‖^2›)
have ‹‖ (*1/2*) ∗*R* (*r n*) − (*1/2*) ∗*R* (*r m*) ‖^2
= (*1/2*)∗(‖ *r n* ‖^2 + ‖ *r m* ‖^2) − ‖ (*1/2*) ∗*R* (*r n*) + (*1/2*) ∗*R* (*r m*) ‖^2›
by (*smt* (*z3*) *div-by-1 field-sum-of-halves nonzero-mult-div-cancel-left parallelogram-law polar-identity power2-norm-eq-cinner′ scaleR-collapse times-divide-eq-left*)
also have ‹...
< (*1/2*)∗(*d* + *1*/(*n*+*1*) + ‖ *r m* ‖^2) − ‖ (*1/2*) ∗*R* (*r n*) + (*1/2*) ∗*R* (*r m*) ‖^2›
using ‹‖*r n*‖^2 < *d* + *1* / *real* (*n* + *1*)› **by** *auto*
also have ‹...
< (*1/2*)∗(*d* + *1*/(*n*+*1*) + *d* + *1*/(*m*+*1*)) − ‖ (*1/2*) ∗*R* (*r n*) + (*1/2*) ∗*R* (*r m*) ‖^2›
using ‹‖*r m*‖^2 < *d* + *1* / *real* (*m* + *1*)› **by** *auto*
also have ‹...
≤ (*1/2*)∗(*d* + *1*/(*n*+*1*) + *d* + *1*/(*m*+*1*)) − *d*›
by (*simp add*: ‹*d* ≤ ‖(*1* / *2*) ∗*R* *r n* + (*1* / *2*) ∗*R* *r m*‖^2›)
also have ‹...
≤ (*1/2*)∗(*1*/(*n*+*1*) + *1*/(*m*+*1*) + *2*∗*d*) − *d*›
by *simp*
also have ‹...
≤ (*1/2*)∗(*1*/(*n*+*1*) + *1*/(*m*+*1*)) + (*1/2*)∗(*2*∗*d*) − *d*›
by (*simp add*: *distrib-left*)
also have ‹...
≤ (*1/2*)∗(*1*/(*n*+*1*) + *1*/(*m*+*1*)) + *d* − *d*›
by *simp*
also have ‹...
≤ (*1/2*)∗(*1*/(*n*+*1*) + *1*/(*m*+*1*))›
by *simp*
finally have ‹ ‖(*1* / *2*) ∗*R* *r n* − (*1* / *2*) ∗*R* *r m*‖^2 < *1* / *2* ∗ (*1* / *real* (*n* + *1*) + *1* / *real* (*m* + *1*)) ›
by *blast*
hence ‹ ‖(*1* / *2*) ∗*R* (*r n* − *r m*) ‖^2 < (*1* / *2*) ∗ (*1* / *real* (*n* + *1*) + *1* / *real* (*m* + *1*)) ›
by (*simp add*: *real-vector.scale-right-diff-distrib*)
hence ‹ ((*1* / *2*)∗‖ (*r n* − *r m*) ‖)^2 < (*1* / *2*) ∗ (*1* / *real* (*n* + *1*) + *1* / *real* (*m* + *1*)) ›
by *simp*
hence ‹ (*1* / *2*)^2∗(‖ (*r n* − *r m*) ‖)^2 < (*1* / *2*) ∗ (*1* / *real* (*n* + *1*) + *1* / *real* (*m* + *1*)) ›

```
    by (metis power-mult-distrib)
   hence ‹ (1 / 4) *(‖ (r n − r m) ‖)^2 < (1 / 2) * (1 / real (n + 1) + 1 / real
(m + 1)) ›
    by (simp add: power-divide)
   hence ‹ ‖ (r n − r m) ‖^2 < 2 * (1 / real (n + 1) + 1 / real (m + 1)) ›
    by simp
   thus ?thesis
    by (metis of-nat-1 of-nat-add)
  qed
  hence ∃ N. ∀ n m. n ≥ N ∧ m ≥ N ⟶ ‖ (r n) − (r m) ‖^2 < ε^2
   if ε > 0
   for ε
  proof−
   obtain N::nat where ‹1/(N + 1) < ε^2/4›
    using LIMSEQ-ignore-initial-segment[OF lim-inverse-n', where k=1]
    by (metis Suc-eq-plus1 ‹0 < ε› nat-approx-posE zero-less-divide-iff zero-less-numeral
        zero-less-power )
   hence ‹4/(N + 1) < ε^2›
    by simp
   have 2*(1/(n+1) + 1/(m+1)) < ε^2
    if f1: n ≥ N and f2: m ≥ N
    for m n::nat
   proof−
    have ‹1/(n+1) ≤ 1/(N+1)›
     by (simp add: f1 linordered-field-class.frac-le)
    moreover have ‹1/(m+1) ≤ 1/(N+1)›
     by (simp add: f2 linordered-field-class.frac-le)
    ultimately have ‹2*(1/(n+1) + 1/(m+1)) ≤ 4/(N+1)›
     by simp
    thus ?thesis using ‹4/(N + 1) < ε^2›
     by linarith
   qed
   hence ‖ (r n) − (r m) ‖^2 < ε^2
    if y1: n ≥ N and y2: m ≥ N
    for m n::nat
    using that
    by (smt ‹⋀n m. ‖r n − r m‖^2 < 2 * (1 / (real n + 1) + 1 / (real m + 1))›
of-nat-1 of-nat-add)
   thus ?thesis
    by blast
  qed
  hence ‹∀ ε > 0. ∃ N::nat. ∀ n m::nat. n ≥ N ∧ m ≥ N ⟶ ‖ (r n) − (r m)
‖^2 < ε^2›
   by blast
  hence ‹∀ ε > 0. ∃ N::nat. ∀ n m::nat. n ≥ N ∧ m ≥ N ⟶ ‖ (r n) − (r m)
‖ < ε›
   by (meson less-eq-real-def power-less-imp-less-base)
  hence ‹Cauchy r›
   using CauchyI by fastforce
```

then obtain k **where** ‹$r \longrightarrow k$›
using *convergent-eq-Cauchy* **by** *auto*
have ‹$k \in M$› **using** ‹*closed M*›
using ‹$\forall n.\ r\ n \in M$› ‹$r \longrightarrow k$› *closed-sequentially* **by** *auto*
have ‹$(\lambda\ n.\ \|\ r\ n\ \|\hat{}2) \longrightarrow \|\ k\ \|\hat{}2$›
by (*simp add*: ‹$r \longrightarrow k$› *tendsto-norm tendsto-power*)
moreover have ‹$(\lambda\ n.\ \|\ r\ n\ \|\hat{}2) \longrightarrow d$›
proof−
have ‹$|\|\ r\ n\ \|\hat{}2 - d| < 1/(n+1)$› **for** n :: *nat*
using ‹$\bigwedge x.\ x \in M \Longrightarrow d \leq \|x\|^2$› ‹$\forall n.\ r\ n \in M \wedge \|r\ n\|^2 < d + 1\ /\ (real\ n + 1)$› *of-nat-1 of-nat-add*
by *smt*
moreover have ‹$(\lambda n.\ 1\ /\ real\ (n + 1)) \longrightarrow 0$›
using *LIMSEQ-ignore-initial-segment*[*OF lim-inverse-n'*, **where** *k=1*] **by** *blast*
ultimately have ‹$(\lambda\ n.\ |\|\ r\ n\ \|\hat{}2 - d|\ ) \longrightarrow 0$›
by (*simp add*: *LIMSEQ-norm-0*)
hence ‹$(\lambda\ n.\ \|\ r\ n\ \|\hat{}2 - d\ ) \longrightarrow 0$›
by (*simp add*: *tendsto-rabs-zero-iff*)
moreover have ‹$(\lambda\ n.\ d\ ) \longrightarrow d$›
by *simp*
ultimately have ‹$(\lambda\ n.\ (\|\ r\ n\ \|\hat{}2 - d)+d\ ) \longrightarrow 0+d$›
using *tendsto-add* **by** *fastforce*
thus *?thesis* **by** *simp*
qed
ultimately have ‹$d = \|\ k\ \|\hat{}2$›
using *LIMSEQ-unique* **by** *auto*
hence ‹$t \in M \Longrightarrow \|\ k\ \|\hat{}2 \leq \|\ t\ \|\hat{}2$› **for** t
using ‹$\bigwedge x.\ x \in M \Longrightarrow d \leq \|x\|^2$› **by** *auto*
hence *q1*: ‹$\exists\ k.\ is\text{-}arg\text{-}min\ (\lambda\ x.\ \|x\|\hat{}2)\ (\lambda\ t.\ t \in M)\ k$›
using ‹$k \in M$›
is-arg-min-def ‹$d = \|k\|^2$›
by *smt*
thus ‹$\exists\ k.\ is\text{-}arg\text{-}min\ (\lambda\ x.\ \|x\|)\ (\lambda\ t.\ t \in M)\ k$›
by (*smt is-arg-min-def norm-ge-zero power2-eq-square power2-le-imp-le*)
qed

lemma *smallest-norm-unique*:
— Theorem 2.5 in [1] (inside the proof)
includes *norm-syntax*
fixes M :: ‹*'a*::*complex-inner set*›
assumes *q1*: ‹*convex M*›
assumes *r*: ‹*is-arg-min* $(\lambda\ x.\ \|x\|)\ (\lambda\ t.\ t \in M)\ r$›
assumes *s*: ‹*is-arg-min* $(\lambda\ x.\ \|x\|)\ (\lambda\ t.\ t \in M)\ s$›
shows ‹$r = s$›
proof −
have ‹$r \in M$›
using ‹*is-arg-min* $(\lambda x.\ \|x\|)\ (\lambda\ t.\ t \in M)\ r$›

by (*simp add*: *is-arg-min-def*)
moreover have ‹*s* ∈ *M*›
using ‹*is-arg-min* (λ*x*. ‖*x*‖) (λ *t*. *t* ∈ *M*) *s*›
by (*simp add*: *is-arg-min-def*)
ultimately have ‹((*1*/*2*) ∗*R* *r* + (*1*/*2*) ∗*R* *s*) ∈ *M*› **using** ‹*convex M*›
by (*simp add*: *convexD*)
hence ‹‖*r*‖ ≤ ‖ (*1*/*2*) ∗*R* *r* + (*1*/*2*) ∗*R* *s* ‖›
by (*metis is-arg-min-linorder r*)
hence *u2*: ‹‖*r*‖^2 ≤ ‖ (*1*/*2*) ∗*R* *r* + (*1*/*2*) ∗*R* *s* ‖^2›
using *norm-ge-zero power-mono* **by** *blast*

have ‹‖*r*‖ ≤ ‖*s*‖›
using *r s is-arg-min-def*
by (*metis is-arg-min-linorder*)
moreover have ‹‖*s*‖ ≤ ‖*r*‖›
using *r s is-arg-min-def*
by (*metis is-arg-min-linorder*)
ultimately have *u3*: ‹‖*r*‖ = ‖*s*‖› **by** *simp*

have ‹‖ (*1*/*2*) ∗*R* *r* − (*1*/*2*) ∗*R* *s* ‖^2 ≤ *0*›
using *u2 u3 parallelogram-law*
by (*smt* (*verit*, *ccfv-SIG*) *polar-identity-minus power2-norm-eq-cinner' scaleR-add-right scaleR-half-double*)
hence ‹‖ (*1*/*2*) ∗*R* *r* − (*1*/*2*) ∗*R* *s* ‖^2 = *0*›
by *simp*
hence ‹‖ (*1*/*2*) ∗*R* *r* − (*1*/*2*) ∗*R* *s* ‖ = *0*›
by *auto*
hence ‹(*1*/*2*) ∗*R* *r* − (*1*/*2*) ∗*R* *s* = *0*›
using *norm-eq-zero* **by** *blast*
thus *?thesis* **by** *simp*
qed

theorem *smallest-dist-exists*:
— Theorem 2.5 in [1]
fixes *M*::‹'*a*::*chilbert-space set*› **and** *h*
assumes *a1*: ‹*convex M*› **and** *a2*: ‹*closed M*› **and** *a3*: ‹*M* ≠ {}›
shows ‹∃ *k*. *is-arg-min* (λ *x*. *dist x h*) (λ *x*. *x* ∈ *M*) *k*›
proof −
have ∗: *is-arg-min* (λ*x*. *dist x h*) (λ*x*. *x*∈*M*) (*k*+*h*) ⟷ *is-arg-min* (λ*x*. *norm x*) (λ*x*. *x*∈(λ*x*. *x*−*h*) ‘ *M*) *k* **for** *k*
unfolding *dist-norm is-arg-min-def* **apply** *auto* **using** *add-implies-diff* **by** *blast*
have ‹∃ *k*. *is-arg-min* (λ*x*. *dist x h*) (λ*x*. *x*∈*M*) (*k*+*h*)›
apply (*subst* ∗)
apply (*rule smallest-norm-exists*)
using *assms* **by** (*auto simp*: *closed-translation-subtract*)
then show ‹∃ *k*. *is-arg-min* (λ *x*. *dist x h*) (λ *x*. *x* ∈ *M*) *k*›
by *metis*
qed

theorem *smallest-dist-unique*:
— Theorem 2.5 in [1]
fixes M::‹'a::*complex-inner set*› **and** h
assumes *a1*: ‹*convex M*›
assumes ‹*is-arg-min* (λ *x*. *dist x h*) (λ *x*. *x* ∈ *M*) *r*›
assumes ‹*is-arg-min* (λ *x*. *dist x h*) (λ *x*. *x* ∈ *M*) *s*›
shows ‹*r* = *s*›
proof−
have ∗: *is-arg-min* (λ*x*. *dist x h*) (λ*x*. *x*∈*M*) *k* ⟷ *is-arg-min* (λ*x*. *norm x*) (λ*x*. *x*∈(λ*x*. *x*−*h*) ‘ *M*) (*k*−*h*) **for** *k*
unfolding *dist-norm is-arg-min-def* **by** *auto*
have ‹*r* − *h* = *s* − *h*›
using - *assms*(*2*,*3*)[*unfolded* ∗] **apply** (*rule smallest-norm-unique*)
by (*simp add*: *a1*)
thus ‹*r* = *s*›
by *auto*
qed

— Theorem 2.6 in [1]
theorem *smallest-dist-is-ortho*:
fixes M::‹'a::*complex-inner set*› **and** h k::'a
assumes *b1*: ‹*closed-csubspace M*›
shows ‹(*is-arg-min* (λ *x*. *dist x h*) (λ *x*. *x* ∈ *M*) *k*) ⟷
h − *k* ∈ *orthogonal-complement M* ∧ *k* ∈ *M*›
proof −
include *norm-syntax*
have ‹*csubspace M*›
using ‹*closed-csubspace M*› **unfolding** *closed-csubspace-def* **by** *blast*
have *r1*: ‹*2* ∗ *Re* ((*h* − *k*) ·$_C$ *f*) ≤ ‖ *f* ‖^*2*›
if *f* ∈ *M* **and** ‹*k* ∈ *M*› **and** ‹*is-arg-min* (λ*x*. *dist x h*) (λ *x*. *x* ∈ *M*) *k*›
for *f*
proof−
have ‹*k* + *f* ∈ *M*›
using ‹*csubspace M*›
by (*simp add*:*complex-vector.subspace-add that*)
have ∀ *f A a b*. ¬ *is-arg-min f* (λ *x*. *x* ∈ *A*) (*a*::'*a*) ∨ (*f a*::*real*) ≤ *f b* ∨ *b* ∉ *A*
by (*metis* (*no-types*) *is-arg-min-linorder*)
hence *dist k h* ≤ *dist* (*f* + *k*) *h*
by (*metis* ‹*is-arg-min* (λ*x*. *dist x h*) (λ *x*. *x* ∈ *M*) *k*› ‹*k* + *f* ∈ *M*› *add.commute*)
hence ‹*dist h k* ≤ *dist h* (*k* + *f*)›
by (*simp add*: *add.commute dist-commute*)
hence ‹‖ *h* − *k* ‖ ≤ ‖ *h* − (*k* + *f*) ‖›
by (*simp add*: *dist-norm*)
hence ‹‖ *h* − *k* ‖^*2* ≤ ‖ *h* − (*k* + *f*) ‖^*2*›
by (*simp add*: *power-mono*)
also have ‹... ≤ ‖ (*h* − *k*) − *f* ‖^*2*›
by (*simp add*: *diff-diff-add*)
also have ‹... ≤ ‖ (*h* − *k*) ‖^*2* + ‖ *f* ‖^*2* − *2* ∗ *Re* ((*h* − *k*) ·$_C$ *f*)›

```
    by (simp add: polar-identity-minus)
   finally have ‹‖ (h − k) ‖^2 ≤ ‖ (h − k) ‖^2 + ‖ f ‖^2 −  2 * Re ((h − k)
·C f)›
    by simp
   thus ?thesis by simp
  qed

  have q4: ‹∀ c > 0.  2 * Re ((h − k) ·C f) ≤ c›
   if ‹∀ c>0. 2 * Re ((h − k) ·C f) ≤ c * ‖f‖^2›
   for f
  proof (cases ‹‖ f ‖^2 > 0›)
   case True
   hence ‹∀ c > 0.  2 * Re (((h − k) ·C f)) ≤ (c/‖ f ‖^2)*‖ f ‖^2›
    using that linordered-field-class.divide-pos-pos by blast
   thus ?thesis
    using True by auto
  next
   case False
   hence ‹‖ f ‖^2 = 0›
    by simp
   thus ?thesis
    by auto
  qed
  have q3: ‹∀ c::real. c > 0 ⟶ 2 * Re (((h − k) ·C f)) ≤ 0›
   if a3: ‹∀f. f ∈ M ⟶ (∀ c>0. 2 * Re ((h − k) ·C f) ≤ c * ‖f‖^2)›
    and a2: f ∈  M
    and a1: is-arg-min (λ x. dist x h) (λ x. x ∈ M) k
   for f
  proof−
   have ‹∀ c > 0.  2 * Re (((h − k) ·C f)) ≤ c*‖ f ‖^2›
    by (simp add: that )
   thus ?thesis
    using q4 by smt
  qed
  have w2: h − k ∈ orthogonal-complement M ∧ k ∈ M
   if a1: is-arg-min (λ x. dist x h) (λ x. x ∈ M) k
  proof−
   have ‹k ∈ M›
    using is-arg-min-def that by fastforce
   hence ‹∀ f. f ∈  M ⟶ 2 * Re (((h − k) ·C f)) ≤ ‖ f ‖^2›
    using r1
    by (simp add: that)
   have ‹∀ f. f ∈  M ⟶
            (∀ c::real.  2 * Re ((h − k) ·C (c *R f)) ≤ ‖ c *R f ‖^2)›
    using  assms scaleR-scaleC complex-vector.subspace-def ‹csubspace M›
    by (metis ‹∀f. f ∈ M ⟶ 2 * Re ((h − k) ·C f) ≤ ‖f‖^2›)
   hence  ‹∀ f. f ∈  M ⟶
            (∀ c::real. c * (2 * Re (((h − k) ·C f))) ≤ ‖ c *R f ‖^2)›
   by (metis Re-complex-of-real cinner-scaleC-right complex-add-cnj complex-cnj-complex-of-real
```

complex-cnj-mult of-real-mult scaleR-scaleC semiring-normalization-rules(*34*))
hence ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* ∗ (2 ∗ *Re* (((*h* − *k*) ·*C f*))) ≤ |*c*|^2∗‖ *f* ‖^2)›
by (*simp add*: *power-mult-distrib*)
hence ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* ∗ (2 ∗ *Re* (((*h* − *k*) ·*C f*))) ≤ *c*^2∗‖ *f* ‖^2)›
by *auto*
hence ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* > 0 ⟶ *c* ∗ (2 ∗ *Re* (((*h* − *k*) ·*C f*))) ≤ *c*^2∗‖ *f* ‖^2)›
by *simp*
hence ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* > 0 ⟶ *c*∗(2 ∗ *Re* (((*h* − *k*) ·*C f*))) ≤ *c*∗(*c*∗‖ *f* ‖^2))›
by (*simp add*: *power2-eq-square*)
hence *q4*: ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* > 0 ⟶ 2 ∗ *Re* (((*h* − *k*) ·*C f*)) ≤ *c*∗‖ *f* ‖^2)›
by *simp*
have ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* > 0 ⟶ 2 ∗ *Re* (((*h* − *k*) ·*C f*)) ≤ 0)›
using *q3*
by (*simp add*: *q4 that*)
hence ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* > 0 ⟶ (2 ∗ *Re* ((*h* − *k*) ·*C* (−1 ∗*R f*))) ≤ 0)›
using *assms scaleR-scaleC complex-vector.subspace-def*
by (*metis* ‹*csubspace M*›)
hence ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* > 0 ⟶ −(2 ∗ *Re* (((*h* − *k*) ·*C f*))) ≤ 0)›
by *simp*
hence ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* > 0 ⟶ 2 ∗ *Re* (((*h* − *k*) ·*C f*)) ≥ 0)›
by *simp*
hence ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* > 0 ⟶ 2 ∗ *Re* (((*h* − *k*) ·*C f*)) = 0)›
using ‹∀ *f*. *f* ∈ *M* ⟶
(∀ *c*::*real*. *c* > 0 ⟶ (2 ∗ *Re* (((*h* − *k*) ·*C f*))) ≤ 0)›
by *fastforce*

have ‹∀ *f*. *f* ∈ *M* ⟶
((1::*real*) > 0 ⟶ 2 ∗ *Re* (((*h* − *k*) ·*C f*)) = 0)›
using ‹∀*f*. *f* ∈ *M* ⟶ (∀ *c*>0. 2 ∗ *Re* (((*h* − *k*) ·*C f*)) = 0)› **by** *blast*
hence ‹∀ *f*. *f* ∈ *M* ⟶ 2 ∗ *Re* (((*h* − *k*) ·*C f*)) = 0›
by *simp*
hence ‹∀ *f*. *f* ∈ *M* ⟶ *Re* (((*h* − *k*) ·*C f*)) = 0›
by *simp*
have ‹∀ *f*. *f* ∈ *M* ⟶ *Re* ((*h* − *k*) ·*C* ((*Complex* 0 (−1)) ∗*C f*)) = 0›
using *assms complex-vector.subspace-def* ‹*csubspace M*›
by (*metis* ‹∀*f*. *f* ∈ *M* ⟶ *Re* ((*h* − *k*) ·*C f*) = 0›)
hence ‹∀ *f*. *f* ∈ *M* ⟶ *Re* ((*Complex* 0 (−1))∗(((*h* − *k*) ·*C f*))) = 0›
by *simp*
hence ‹∀ *f*. *f* ∈ *M* ⟶ *Im* (((*h* − *k*) ·*C f*)) = 0›

using *Complex-eq-neg-1 Re-i-times cinner-scaleC-right complex-of-real-def* **by** *auto*

have ‹$\forall\ f.\ f \in\ M \longrightarrow (((h - k) \cdot_C f)) = 0$›
using *complex-eq-iff*
by (*simp add*: ‹$\forall f.\ f \in M \longrightarrow Im\ ((h - k) \cdot_C f) = 0$› ‹$\forall f.\ f \in M \longrightarrow Re\ ((h - k) \cdot_C f) = 0$›)
hence ‹$h - k \in$ *orthogonal-complement* $M \wedge k \in M$›
by (*simp add*: ‹$k \in M$› *orthogonal-complementI*)
have ‹$\forall\ c.\ c *_R f \in M$›
if $f \in M$
for f
using *that scaleR-scaleC* ‹*csubspace* M› *complex-vector.subspace-def*
by (*simp add*: *complex-vector.subspace-def scaleR-scaleC*)
have ‹$((h - k) \cdot_C f) = 0$›
if $f \in M$
for f
using ‹$h - k \in$ *orthogonal-complement* $M \wedge k \in M$› *orthogonal-complement-orthoI that* **by** *auto*
hence ‹$h - k \in$ *orthogonal-complement* M›
by (*simp add*: *orthogonal-complement-def*)
thus *?thesis*
using ‹$k \in M$› **by** *auto*
qed

have *q1*: ‹*dist* $h\ k \leq$ *dist* $h\ f$ ›
if $f \in M$ **and** ‹$h - k \in$ *orthogonal-complement* $M \wedge k \in M$›
for f
proof−
have ‹$(h - k) \cdot_C (k - f) = 0$›
by (*metis* (*no-types*, *lifting*) *that*
cinner-diff-right diff-0-right orthogonal-complement-orthoI that)
have ‹$\| h - f \|\widehat{}2 = \| (h - k) + (k - f) \|\widehat{}2$›
by *simp*
also have ‹$... = \| h - k \|\widehat{}2 + \| k - f \|\widehat{}2$›
using ‹$((h - k) \cdot_C (k - f)) = 0$› *pythagorean-theorem* **by** *blast*
also have ‹$... \geq \| h - k \|\widehat{}2$›
by *simp*
finally have ‹$\|h - k\|^2 \leq \|h - f\|^2$ ›
by *blast*
hence ‹$\|h - k\| \leq \|h - f\|$›
using *norm-ge-zero power2-le-imp-le* **by** *blast*
thus *?thesis*
by (*simp add*: *dist-norm*)
qed

have *w1*: *is-arg-min* $(\lambda\ x.\ dist\ x\ h)\ (\lambda\ x.\ x \in M)\ k$
if $h - k \in$ *orthogonal-complement* $M \wedge k \in\ M$
proof−

have ‹$h - k \in$ *orthogonal-complement* M›
using *that* **by** *blast*
have ‹$k \in M$› **using** ‹$h - k \in$ *orthogonal-complement* $M \wedge k \in M$›
by *blast*
thus *?thesis*
by (*metis* (*no-types*, *lifting*) *dist-commute is-arg-min-linorder q1 that*)
qed
show *?thesis*
using *w1 w2* **by** *blast*
qed

corollary *orthog-proj-exists*:
fixes M :: ‹'*a*::*chilbert-space set*›
assumes ‹*closed-csubspace* M›
shows ‹$\exists k.\ h - k \in$ *orthogonal-complement* $M \wedge k \in M$›
proof −
from ‹*closed-csubspace* M›
have ‹$M \neq \{\}$›
using *closed-csubspace.subspace complex-vector.subspace-0* **by** *blast*
have ‹*closed* M›
using ‹*closed-csubspace* M›
by (*simp add*: *closed-csubspace.closed*)
have ‹*convex* M›
using ‹*closed-csubspace* M›
by (*simp*)
have ‹$\exists k.$ *is-arg-min* (λ x. *dist* x h) (λ x. $x \in M$) k›
by (*simp add*: *smallest-dist-exists* ‹*closed* M› ‹*convex* M› ‹$M \neq \{\}$›)
thus *?thesis*
by (*simp add*: *assms smallest-dist-is-ortho*)
qed

corollary *orthog-proj-unique*:
fixes M :: ‹'*a*::*complex-inner set*›
assumes ‹*closed-csubspace* M›
assumes ‹$h - r \in$ *orthogonal-complement* $M \wedge r \in M$›
assumes ‹$h - s \in$ *orthogonal-complement* $M \wedge s \in M$›
shows ‹$r = s$›
using - *assms*(*2*,*3*) **unfolding** *smallest-dist-is-ortho*[*OF assms*(*1*), *symmetric*]
apply (*rule smallest-dist-unique*)
using *assms*(*1*) **by** (*simp*)

definition *is-projection-on*::‹('*a* ⇒ '*a*) ⇒ ('*a*::*metric-space*) *set* ⇒ *bool*› **where**
‹*is-projection-on* π $M \longleftrightarrow$ ($\forall h.$ *is-arg-min* (λ x. *dist* x h) (λ x. $x \in M$) (π h))›

lemma *is-projection-on-iff-orthog*:
‹*closed-csubspace* $M \Longrightarrow$ *is-projection-on* π $M \longleftrightarrow$ ($\forall h.$ $h - \pi$ $h \in$ *orthogonal-complement* $M \wedge \pi$ $h \in M$)›
by (*simp add*: *is-projection-on-def smallest-dist-is-ortho*)

lemma *is-projection-on-exists*:
fixes M :: ‹'a::*chilbert-space set*›
assumes ‹*convex* M› **and** ‹*closed* M› **and** ‹$M \neq \{\}$›
shows $\exists \pi$. *is-projection-on* π M
unfolding *is-projection-on-def* **apply** (*rule choice*)
using *smallest-dist-exists*[*OF assms*] **by** *auto*

lemma *is-projection-on-unique*:
fixes M :: ‹'a::*complex-inner set*›
assumes ‹*convex* M›
assumes *is-projection-on* π_1 M
assumes *is-projection-on* π_2 M
shows $\pi_1 = \pi_2$
using *smallest-dist-unique*[*OF assms*(*1*)] **using** *assms*(*2*,*3*)
unfolding *is-projection-on-def* **by** *blast*

definition *projection* :: ‹'a::*metric-space set* ⇒ ('a ⇒ 'a)› **where**
‹*projection* M = (*SOME* π. *is-projection-on* π M)›

lemma *projection-is-projection-on*:
fixes M :: ‹'a::*chilbert-space set*›
assumes ‹*convex* M› **and** ‹*closed* M› **and** ‹$M \neq \{\}$›
shows *is-projection-on* (*projection* M) M
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *is-projection-on-exists projection-def someI*)

lemma *projection-is-projection-on'*[*simp*]:
— Common special case of ⟦*convex ?M*; *closed ?M*; *?M* ≠ {}⟧ ⟹ *is-projection-on* (*projection ?M*) *?M*
fixes M :: ‹'a::*chilbert-space set*›
assumes ‹*closed-csubspace* M›
shows *is-projection-on* (*projection* M) M
apply (*rule projection-is-projection-on*)
apply (*auto simp add*: *assms closed-csubspace.closed*)
using *assms closed-csubspace.subspace complex-vector.subspace-0* **by** *blast*

lemma *projection-orthogonal*:
fixes M :: ‹'a::*chilbert-space set*›
assumes *closed-csubspace* M **and** ‹$m \in M$›
shows ‹*is-orthogonal* (h − *projection* M h) m›
by (*metis assms*(*1*) *assms*(*2*) *closed-csubspace.closed closed-csubspace.subspace csubspace-is-convex empty-iff is-projection-on-iff-orthog orthogonal-complement-orthoI projection-is-projection-on*)

lemma *is-projection-on-in-image*:
assumes *is-projection-on* π M
shows π $h \in M$
using *assms*
by (*simp add*: *is-arg-min-def is-projection-on-def*)

lemma *is-projection-on-image*:
assumes *is-projection-on* π M
shows *range* $\pi = M$
using *assms*
apply (*auto simp*: *is-projection-on-in-image*)
by (*smt* (*verit*, *ccfv-threshold*) *dist-pos-lt dist-self is-arg-min-def is-projection-on-def rangeI*)

lemma *projection-in-image*[*simp*]:
fixes M :: ‹'*a*::*chilbert-space set*›
assumes ‹*convex M*› **and** ‹*closed M*› **and** ‹$M \neq \{\}$›
shows ‹*projection M h* $\in M$›
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *is-projection-on-in-image projection-is-projection-on*)

lemma *projection-image*[*simp*]:
fixes M :: ‹'*a*::*chilbert-space set*›
assumes ‹*convex M*› **and** ‹*closed M*› **and** ‹$M \neq \{\}$›
shows ‹*range* (*projection M*) = *M*›
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *is-projection-on-image projection-is-projection-on*)

lemma *projection-eqI'*:
fixes M :: ‹'*a*::*complex-inner set*›
assumes ‹*convex M*›
assumes ‹*is-projection-on f M*›
shows ‹*projection M* = *f*›
by (*metis assms*(*1*) *assms*(*2*) *is-projection-on-unique projection-def someI-ex*)

lemma *is-projection-on-eqI*:
fixes M :: ‹'*a*::*complex-inner set*›
assumes *a1*: ‹*closed-csubspace M*› **and** *a2*: ‹$h - x \in$ *orthogonal-complement M*› **and** *a3*: ‹$x \in M$›
and *a4*: ‹*is-projection-on* π *M*›
shows ‹π *h* = *x*›
by (*meson a1 a2 a3 a4 closed-csubspace.subspace csubspace-is-convex is-projection-on-def smallest-dist-is-ortho smallest-dist-unique*)

lemma *projection-eqI*:
fixes M :: ‹('*a*::*chilbert-space*) *set*›
assumes ‹*closed-csubspace M*› **and** ‹$h - x \in$ *orthogonal-complement M*› **and** ‹$x \in M$›
shows ‹*projection M h* = *x*›
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *is-projection-on-iff-orthog orthog-proj-exists projection-def is-projection-on-eqI tfl-some*)

lemma *is-projection-on-fixes-image*:
fixes M :: ‹'*a*::*metric-space set*›
assumes *a1*: *is-projection-on* π M **and** *a3*: $x \in M$
shows $\pi\ x = x$

by (*metis a1 a3 dist-pos-lt dist-self is-arg-min-def is-projection-on-def*)

lemma *projection-fixes-image*:
fixes M :: ‹('a::*chilbert-space*) *set*›
assumes *closed-csubspace* M **and** $x \in M$
shows *projection* $M\ x = x$
using *is-projection-on-fixes-image*
— Theorem 2.7 in [1]
by (*simp add*: *assms complex-vector.subspace-0 projection-eqI*)

lemma *is-projection-on-closed*:
assumes *cont-f*: ‹$\bigwedge x.\ x \in$ *closure* $M \Longrightarrow$ *isCont* $f\ x$›
assumes ‹*is-projection-on* $f\ M$›
shows ‹*closed* M›
proof −
have ‹$x \in M$› **if** ‹$s \longrightarrow x$› **and** ‹*range* $s \subseteq M$› **for** $s\ x$
proof −
from ‹*is-projection-on* $f\ M$› ‹*range* $s \subseteq M$›
have ‹$s = (f\ o\ s)$›
by (*simp add*: *comp-def is-projection-on-fixes-image range-subsetD*)
also from *cont-f* ‹$s \longrightarrow x$›
have ‹$(f\ o\ s) \longrightarrow f\ x$›
apply (*rule continuous-imp-tendsto*)
using ‹$s \longrightarrow x$› ‹*range* $s \subseteq M$›
by (*meson closure-sequential range-subsetD*)
finally have ‹$x = f\ x$›
using ‹$s \longrightarrow x$›
by (*simp add*: *LIMSEQ-unique*)
then have ‹$x \in$ *range* f›
by *simp*
with ‹*is-projection-on* $f\ M$› **show** ‹$x \in M$›
by (*simp add*: *is-projection-on-image*)
qed
then show *?thesis*
by (*metis closed-sequential-limits image-subset-iff*)
qed

proposition *is-projection-on-reduces-norm*:
includes *norm-syntax*
fixes M :: ‹('a::*complex-inner*) *set*›
assumes ‹*is-projection-on* $\pi\ M$› **and** ‹*closed-csubspace* M›
shows ‹$\| \pi\ \ h \| \leq \| h \|$›
proof−
have ‹$h - \pi\ h \in$ *orthogonal-complement* M›
using *assms is-projection-on-iff-orthog* **by** *blast*
hence ‹$\forall\ k \in M.$ *is-orthogonal* $(h - \pi\ h)\ k$›
using *orthogonal-complement-orthoI* **by** *blast*
also have ‹$\pi\ h \in\ M$›
using ‹*is-projection-on* $\pi\ M$›

by (*simp add*: *is-projection-on-in-image*)
ultimately have ‹*is-orthogonal* (*h* − *π h*) (*π h*)›
by *auto*
hence ‹‖ *π h* ‖^2 + ‖ *h* − *π h* ‖^2 = ‖ *h* ‖^2›
using *pythagorean-theorem* **by** *fastforce*
hence ‹‖*π h* ‖^2 ≤ ‖ *h* ‖^2›
by (*smt zero-le-power2*)
thus *?thesis*
using *norm-ge-zero power2-le-imp-le* **by** *blast*
qed

proposition *projection-reduces-norm*:
includes *norm-syntax*
fixes *M* :: ‹'*a*::*chilbert-space set*›
assumes *a1*: *closed-csubspace M*
shows ‹‖ *projection M h* ‖ ≤ ‖ *h* ‖›
using *assms is-projection-on-iff-orthog orthog-proj-exists is-projection-on-reduces-norm projection-eqI* **by** *blast*

— Theorem 2.7 (version) in [1]
theorem *is-projection-on-bounded-clinear*:
fixes *M* :: ‹'*a*::*complex-inner set*›
assumes *a1*: *is-projection-on π M* **and** *a2*: *closed-csubspace M*
shows *bounded-clinear π*
proof
have *b1*: ‹*csubspace* (*orthogonal-complement M*)›
by (*simp add*: *a2*)
have *f1*: ∀ *a*. *a* − *π a* ∈ *orthogonal-complement M* ∧ *π a* ∈ *M*
using *a1 a2 is-projection-on-iff-orthog* **by** *blast*
hence *c* *_*C x* − *c* *_*C π x* ∈ *orthogonal-complement M*
for *c x*
by (*metis* (*no-types*) *b1*
add-diff-cancel-right' complex-vector.subspace-def diff-add-cancel scaleC-add-right)
thus *r1*: ‹*π* (*c* *_*C x*) = *c* *_*C* (*π x*)› **for** *x c*
using *f1* **by** (*meson a2 a1 closed-csubspace.subspace*
complex-vector.subspace-def is-projection-on-eqI)
show *r2*: ‹*π* (*x* + *y*) = (*π x*) + (*π y*)›
for *x y*
proof−
have ∀ *A*. ¬ *closed-csubspace* (*A*::'*a set*) ∨ *csubspace A*
by (*metis closed-csubspace.subspace*)
hence *csubspace M*
using *a2* **by** *auto*
hence ‹*π* (*x* + *y*) − ((*π x*) + (*π y*)) ∈ *M*›
by (*simp add*: *complex-vector.subspace-add complex-vector.subspace-diff f1*)
have ‹*closed-csubspace* (*orthogonal-complement M*)›
using *a2*
by *simp*
have *f1*: ∀ *a b*. (*b*::'*a*) + (*a* − *b*) = *a*

by (*metis add.commute diff-add-cancel*)
have *f2*: $\forall\, a\ b.\ (b::'a) - b = a - a$
by *auto*
hence *f3*: $\forall\, a.\ a - a \in$ *orthogonal-complement M*
by (*simp add*: *complex-vector.subspace-0*)
have $\forall\, a\ b.\ (a \in$ *orthogonal-complement M* $\vee\ a + b \notin$ *orthogonal-complement M*)
$\vee\ b \notin$ *orthogonal-complement M*
using *add-diff-cancel-right' b1 complex-vector.subspace-diff*
by *metis*
hence $\forall\, a\ b\ c.\ (a \in$ *orthogonal-complement M* $\vee\ c - (b + a) \notin$ *orthogonal-complement M*)
$\vee\ c - b \notin$ *orthogonal-complement M*
using *f1* **by** (*metis diff-diff-add*)
hence *f4*: $\forall\, a\ b\ f.\ (f\ a - b \in$ *orthogonal-complement M* $\vee\ a - b \notin$ *orthogonal-complement M*)
$\vee\ \neg$ *is-projection-on f M*
using *f1*
by (*metis a2 is-projection-on-iff-orthog*)
have *f5*: $\forall\, a\ b\ c\ d.\ (d::'a) - (c + (b - a)) = d + (a - (b + c))$
by *auto*
have $x - \pi\ x \in$ *orthogonal-complement M*
using *a1 a2 is-projection-on-iff-orthog* **by** *blast*
hence *q1*: ‹$\pi\ (x + y) - (\ (\pi\ x) + (\pi\ y)\ ) \in$ *orthogonal-complement M*›
using *f5 f4 f3* **by** (*metis* ‹*csubspace* (*orthogonal-complement M*)›
‹*is-projection-on* π *M*› *add-diff-eq complex-vector.subspace-diff diff-diff-add diff-diff-eq2*)
hence ‹$\pi\ (x + y) - (\ (\pi\ x) + (\pi\ y)\ ) \in M \cap$ (*orthogonal-complement M*)›
by (*simp add*: ‹$\pi\ (x + y) - (\pi\ x + \pi\ y) \in M$›)
moreover have ‹$M \cap$ (*orthogonal-complement M*) $= \{0\}$›
by (*simp add*: ‹*closed-csubspace M*› *complex-vector.subspace-0 orthogonal-complement-zero-intersection*)
ultimately have ‹$\pi\ (x + y) - (\ (\pi\ x) + (\pi\ y)\ ) = 0$›
by *auto*
thus *?thesis* **by** *simp*
qed
from *is-projection-on-reduces-norm*
show *t1*: ‹$\exists\ K.\ \forall\ x.$ *norm* $(\pi\ x) \leq$ *norm* $x * K$›
by (*metis a1 a2 mult.left-neutral ordered-field-class.sign-simps*(5))
qed

theorem *projection-bounded-clinear*:
fixes *M* :: ‹('*a*::*chilbert-space*) *set*›
assumes *a1*: *closed-csubspace M*
shows ‹*bounded-clinear* (*projection M*)›
— Theorem 2.7 in [1]
using *assms is-projection-on-iff-orthog orthog-proj-exists is-projection-on-bounded-clinear projection-eqI* **by** *blast*

proposition *is-projection-on-idem*:

fixes M :: ‹('a::*complex-inner*) *set*›
assumes *is-projection-on* π M
shows π (π x) = π x
using *is-projection-on-fixes-image is-projection-on-in-image assms* **by** *blast*

proposition *projection-idem*:
fixes M :: 'a::*chilbert-space set*
assumes *a1*: *closed-csubspace M*
shows *projection M* (*projection M x*) = *projection M x*
by (*metis assms closed-csubspace.closed closed-csubspace.subspace complex-vector.subspace-0 csubspace-is-convex equals0D projection-fixes-image projection-in-image*)

proposition *is-projection-on-kernel-is-orthogonal-complement*:
fixes M :: ‹'a::*complex-inner set*›
assumes *a1*: *is-projection-on* π M **and** *a2*: *closed-csubspace M*
shows π −‘ {*0*} = *orthogonal-complement M*
proof−
have $x \in (\pi$ −‘ {*0*}$)$
if $x \in$ *orthogonal-complement M*
for x
by (*smt* (*verit*, *ccfv-SIG*) *a1 a2 closed-csubspace-def complex-vector.subspace-def complex-vector.subspace-diff is-projection-on-eqI orthogonal-complement-closed-subspace that vimage-singleton-eq*)
moreover have $x \in$ *orthogonal-complement M*
if *s1*: $x \in \pi$ −‘ {*0*} **for** x
by (*metis a1 a2 diff-zero is-projection-on-iff-orthog that vimage-singleton-eq*)
ultimately show *?thesis*
by *blast*
qed

— Theorem 2.7 in [1]
proposition *projection-kernel-is-orthogonal-complement*:
fixes M :: ‹'a::*chilbert-space set*›
assumes *closed-csubspace M*
shows (*projection M*) −‘ {*0*} = (*orthogonal-complement M*)
by (*metis assms closed-csubspace-def complex-vector.subspace-def csubspace-is-convex insert-absorb insert-not-empty is-projection-on-kernel-is-orthogonal-complement projection-is-projection-on*)

lemma *is-projection-on-id-minus*:
fixes M :: ‹'a::*complex-inner set*›
assumes *is-proj*: *is-projection-on* π M
and *cc*: *closed-csubspace M*
shows *is-projection-on* (*id* − π) (*orthogonal-complement M*)
using *is-proj* **apply** (*simp add*: *cc is-projection-on-iff-orthog*)
using *double-orthogonal-complement-increasing* **by** *blast*

Exercise 2 (section 2, chapter I) in [1]

lemma *projection-on-orthogonal-complement*[*simp*]:
fixes M :: *'a*::*chilbert-space set*
assumes *a1*: *closed-csubspace M*
shows *projection* (*orthogonal-complement M*) = *id* − *projection M*
apply (*auto intro*!: *ext*)
by (*smt* (*verit*, *ccfv-SIG*) *add-diff-cancel-left' assms closed-csubspace.closed closed-csubspace.subspace complex-vector.subspace-0 csubspace-is-convex diff-add-cancel double-orthogonal-complement-increasing insert-absorb insert-not-empty is-projection-on-iff-orthog orthogonal-complement-closed-subspace projection-eqI projection-is-projection-on subset-eq*)

lemma *is-projection-on-zero*:
is-projection-on (λ-. *0*) {*0*}
by (*simp add*: *is-projection-on-def is-arg-min-def*)

lemma *projection-zero*[*simp*]:
projection {*0*} = (λ-. *0*)
using *is-projection-on-zero*
by (*metis* (*full-types*) *is-projection-on-in-image projection-def singletonD someI-ex*)

lemma *is-projection-on-rank1*:
fixes t :: ‹*'a*::*complex-inner*›
shows ‹*is-projection-on* (λ*x*. ((*t* ·$_C$ *x*) / (*t* ·$_C$ *t*)) *$_C$ *t*) (*cspan* {*t*})›
proof (*cases* ‹*t* = *0*›)
case *True*
then show *?thesis*
by (*simp add*: *is-projection-on-zero*)
next
case *False*
define *P* **where** ‹*P x* = ((*t* ·$_C$ *x*) / (*t* ·$_C$ *t*)) *$_C$ *t*› **for** *x*
define *t'* **where** ‹*t'* = *t* /$_C$ *norm t*›
with *False* **have** ‹*norm t'* = *1*›
by (*simp add*: *norm-inverse*)
have *P-def'*: ‹*P x* = *cinner t' x* *$_C$ *t'*› **for** *x*
unfolding *P-def t'-def* **apply** *auto*
by (*metis divide-divide-eq-left divide-inverse mult.commute power2-eq-square power2-norm-eq-cinner*)
have *spant'*: ‹*cspan* {*t*} = *cspan* {*t'*}›
by (*simp add*: *False t'-def*)
have *cc*: ‹*closed-csubspace* (*cspan* {*t*})›
by (*auto intro*!: *finite-cspan-closed closed-csubspace.intro*)
have *ortho*: ‹*h* − *P h* ∈ *orthogonal-complement* (*cspan* {*t*})› **for** *h*
unfolding *orthogonal-complement-def P-def' spant'* **apply** *auto*
by (*smt* (*verit*, *ccfv-threshold*) ‹*norm t'* = *1*› *add-cancel-right-left cinner-add-right cinner-commute' cinner-scaleC-right cnorm-eq-1 complex-vector.span-breakdown-eq complex-vector.span-empty diff-add-cancel mult-cancel-left1 singletonD*)
have *inspan*: ‹*P h* ∈ *cspan* {*t*}› **for** *h*
unfolding *P-def' spant'*
by (*simp add*: *complex-vector.span-base complex-vector.span-scale*)
show ‹*is-projection-on P* (*cspan* {*t*})›

apply (*subst is-projection-on-iff-orthog*)
using *cc ortho inspan* **by** *auto*
qed

lemma *projection-rank1*:
fixes *t x* :: ‹'*a*::*complex-inner*›
shows ‹*projection* (*cspan* {*t*}) *x* = ((*t* ·$_C$ *x*) / (*t* ·$_C$ *t*)) *$*$_C$ *t*›
apply (*rule fun-cong*, *rule projection-eqI'*, *simp*)
by (*rule is-projection-on-rank1*)

9.5 More orthogonal complement

The following lemmas logically fit into the "orthogonality" section but depend on projections for their proofs.

Corollary 2.8 in [1]

theorem *double-orthogonal-complement-id*[*simp*]:
fixes *M* :: ‹'*a*::*chilbert-space set*›
assumes *a1*: *closed-csubspace M*
shows *orthogonal-complement* (*orthogonal-complement M*) = *M*
proof−
have *b2*: *x* ∈ (*id* − *projection M*) −‘ {*0*}
if *c1*: *x* ∈ *M* **for** *x*
by (*simp add*: *assms projection-fixes-image that*)

have *b3*: ‹*x* ∈ *M*›
if *c1*: ‹*x* ∈ (*id* − *projection M*) −‘ {*0*}› **for** *x*
by (*metis assms closed-csubspace.closed closed-csubspace.subspace complex-vector.subspace-0 csubspace-is-convex eq-id-iff equals0D fun-diff-def projection-in-image right-minus-eq that vimage-singleton-eq*)
have ‹*x* ∈ *M* ⟷ *x* ∈ (*id* − *projection M*) −‘ {*0*}› **for** *x*
using *b2 b3* **by** *blast*
hence *b4*: ‹(*id* − (*projection M*)) −‘ {*0*} = *M*›
by *blast*
have *b1*: *orthogonal-complement* (*orthogonal-complement M*)
= (*projection* (*orthogonal-complement M*)) −‘ {*0*}
by (*simp add*: *a1 projection-kernel-is-orthogonal-complement del*: *projection-on-orthogonal-complement*)
also have ‹... = (*id* − (*projection M*)) −‘ {*0*}›
by (*simp add*: *a1*)
also have ‹... = *M*›
by (*simp add*: *b4*)
finally show *?thesis* **by** *blast*
qed

lemma *orthogonal-complement-antimono-iff*[*simp*]:
fixes *A B* :: ‹('*a*::*chilbert-space*) *set*›
assumes ‹*closed-csubspace A*› **and** ‹*closed-csubspace B*›
shows ‹*orthogonal-complement A* ⊆ *orthogonal-complement B* ⟷ *A* ⊇ *B*›
proof (*rule iffI*)

show ‹*orthogonal-complement A* ⊆ *orthogonal-complement B*› **if** ‹*A* ⊇ *B*›
using *that* **by** *auto*

assume ‹*orthogonal-complement A* ⊆ *orthogonal-complement B*›
then have ‹*orthogonal-complement* (*orthogonal-complement A*) ⊇ *orthogonal-complement* (*orthogonal-complement B*)›
by *simp*
then show ‹*A* ⊇ *B*›
using *assms* **by** *auto*
qed

lemma *de-morgan-orthogonal-complement-plus*:
fixes *A B*::('*a*::*complex-inner*) *set*
assumes ‹*0* ∈ *A*› **and** ‹*0* ∈ *B*›
shows ‹*orthogonal-complement* (*A* +$_M$ *B*) = *orthogonal-complement A* ∩ *orthogonal-complement B*›
proof −
have *x* ∈ (*orthogonal-complement A*) ∩ (*orthogonal-complement B*)
if *x* ∈ *orthogonal-complement* (*A* +$_M$ *B*) **for** *x*
proof −
have ‹*orthogonal-complement* (*A* +$_M$ *B*) = *orthogonal-complement* (*A* + *B*)›
unfolding *closed-sum-def* **by** (*subst orthogonal-complement-of-closure*[*symmetric*], *simp*)
hence ‹*x* ∈ *orthogonal-complement* (*A* + *B*)›
using *that* **by** *blast*
hence *t1*: ‹∀ *z* ∈ (*A* + *B*). (*z* ·$_C$ *x*) = *0*›
by (*simp add*: *orthogonal-complement-orthoI'*)
have ‹*A* ⊆ *A* + *B*›
using *subset-iff add.commute set-zero-plus2* ‹*0* ∈ *B*›
by *fastforce*
hence ‹∀ *z* ∈ *A*. (*z* ·$_C$ *x*) = *0*›
using *t1* **by** *auto*
hence *w1*: ‹*x* ∈ (*orthogonal-complement A*)›
by (*smt mem-Collect-eq is-orthogonal-sym orthogonal-complement-def*)
have ‹*B* ⊆ *A* + *B*›
using ‹*0* ∈ *A*› *subset-iff set-zero-plus2* **by** *blast*
hence ‹∀ *z* ∈ *B*. (*z* ·$_C$ *x*) = *0*›
using *t1* **by** *auto*
hence ‹*x* ∈ (*orthogonal-complement B*)›
by (*smt mem-Collect-eq is-orthogonal-sym orthogonal-complement-def*)
thus *?thesis*
using *w1* **by** *auto*
qed
moreover have *x* ∈ (*orthogonal-complement* (*A* +$_M$ *B*))
if *v1*: *x* ∈ (*orthogonal-complement A*) ∩ (*orthogonal-complement B*)
for *x*
proof−
have ‹*x* ∈ (*orthogonal-complement A*)›
using *v1*

by *blast*
hence ‹$\forall\, y{\in}\ A.\ (y \cdot_C x) = 0$›
by (*simp add*: *orthogonal-complement-orthoI'*)
have ‹$x \in (\mathit{orthogonal\text{-}complement}\ B)$›
using *v1*
by *blast*
hence ‹$\forall\ y{\in}\ B.\ (y \cdot_C x) = 0$›
by (*simp add*: *orthogonal-complement-orthoI'*)
have ‹$\forall\ a{\in}A.\ \forall\ b{\in}B.\ (a{+}b) \cdot_C x = 0$›
by (*simp add*: ‹$\forall\, y{\in}A.\ y \cdot_C x = 0$› ‹$\forall\, y{\in}B.\ (y \cdot_C x) = 0$› *cinner-add-left*)
hence ‹$\forall\ y \in (A + B).\ y \cdot_C x = 0$›
using *set-plus-elim* **by** *force*
hence ‹$x \in (\mathit{orthogonal\text{-}complement}\ (A + B))$›
by (*smt mem-Collect-eq is-orthogonal-sym orthogonal-complement-def*)
moreover have ‹$(\mathit{orthogonal\text{-}complement}\ (A + B)) = (\mathit{orthogonal\text{-}complement}$
$(A +_M B))$›
unfolding *closed-sum-def* **by** (*subst orthogonal-complement-of-closure*[*symmetric*],
simp)
ultimately have ‹$x \in (\mathit{orthogonal\text{-}complement}\ (A +_M B))$›
by *blast*
thus *?thesis*
by *blast*
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *de-morgan-orthogonal-complement-inter*:
fixes *A B*::'*a*::*chilbert-space set*
assumes *a1*: ‹*closed-csubspace A*› **and** *a2*: ‹*closed-csubspace B*›
shows ‹$\mathit{orthogonal\text{-}complement}\ (A \cap B) = \mathit{orthogonal\text{-}complement}\ A +_M \mathit{orthogonal\text{-}complement}\ B$›
proof−
have ‹$\mathit{orthogonal\text{-}complement}\ A +_M \mathit{orthogonal\text{-}complement}\ B$
$= \mathit{orthogonal\text{-}complement}\ (\mathit{orthogonal\text{-}complement}\ (\mathit{orthogonal\text{-}complement}\ A$
$+_M\ \mathit{orthogonal\text{-}complement}\ B))$›
by (*simp add*: *closed-subspace-closed-sum*)
also have ‹$\ldots = \mathit{orthogonal\text{-}complement}\ (\mathit{orthogonal\text{-}complement}\ (\mathit{orthogonal\text{-}complement}$
$A) \cap \mathit{orthogonal\text{-}complement}\ (\mathit{orthogonal\text{-}complement}\ B))$›
by (*simp add*: *de-morgan-orthogonal-complement-plus orthogonal-complementI*)
also have ‹$\ldots = \mathit{orthogonal\text{-}complement}\ (A \cap B)$›
by (*simp add*: *a1 a2*)
finally show *?thesis*
by *simp*
qed

lemma *orthogonal-complement-of-cspan*: ‹$\mathit{orthogonal\text{-}complement}\ A = \mathit{orthogonal\text{-}complement}$
$(\mathit{cspan}\ A)$›
by (*metis* (*no-types*, *opaque-lifting*) *closed-csubspace.subspace complex-vector.span-minimal*
complex-vector.span-superset double-orthogonal-complement-increasing orthogonal-complement-antimono

orthogonal-complement-closed-subspace subset-antisym)

lemma *orthogonal-complement-orthogonal-complement-closure-cspan*:
‹*orthogonal-complement* (*orthogonal-complement S*) = *closure* (*cspan S*)› **for** *S*
:: ‹'*a*::*chilbert-space set*›
proof −
have ‹*orthogonal-complement* (*orthogonal-complement S*) = *orthogonal-complement*
(*orthogonal-complement* (*closure* (*cspan S*)))›
by (*simp flip*: *orthogonal-complement-of-closure orthogonal-complement-of-cspan*)
also have ‹... = *closure* (*cspan S*)›
by *simp*
finally show ‹*orthogonal-complement* (*orthogonal-complement S*) = *closure* (*cspan*
S)›
by −
qed

instance *ccsubspace* :: (*chilbert-space*) *complete-orthomodular-lattice*
proof
fix *X Y* :: ‹'*a ccsubspace*›

show *inf X* (− *X*) = *bot*
apply *transfer*
by (*simp add*: *closed-csubspace-def complex-vector.subspace-0 orthogonal-complement-zero-intersection*)

have ‹*t* ∈ *M* +*M orthogonal-complement M*›
if ‹*closed-csubspace M*› **for** *t*::'*a* **and** *M*
by (*metis* (*no-types*, *lifting*) *UNIV-I closed-csubspace.subspace complex-vector.subspace-def*
de-morgan-orthogonal-complement-inter double-orthogonal-complement-id orthogo-
nal-complement-closed-subspace orthogonal-complement-zero orthogonal-complement-zero-intersection
that)
hence *b1*: ‹*M* +*M orthogonal-complement M* = *UNIV*›
if ‹*closed-csubspace M*› **for** *M* :: ‹'*a set*›
using *that* **by** *blast*
show *sup X* (− *X*) = *top*
apply *transfer*
using *b1* **by** *auto*
show − (− *X*) = *X*
apply *transfer* **by** *simp*

show − *Y* ≤ − *X*
if *X* ≤ *Y*
using *that* **apply** *transfer* **by** *simp*

have *c1*: *M* +*M orthogonal-complement M* ∩ *N* ⊆ *N*
if *closed-csubspace M* **and** *closed-csubspace N* **and** *M* ⊆ *N*
for *M N* :: '*a set*
using *that*
by (*simp add*: *closed-sum-is-sup*)

```
  have c2: ‹u ∈ M +_M (orthogonal-complement M ∩ N)›
    if a1: closed-csubspace M and a2: closed-csubspace N and a3: M ⊆ N and
x1: ‹u ∈ N›
    for M :: 'a set and N :: 'a set  and u
  proof −
    have d4: ‹(projection M) u ∈ M›
    by (metis a1 closed-csubspace-def csubspace-is-convex equals0D orthog-proj-exists
projection-in-image)
    hence d2: ‹(projection M) u ∈ N›
      using a3 by auto
    have d1: ‹csubspace N›
      by (simp add: a2)
    have ‹u − (projection M) u ∈ orthogonal-complement M›
      by (simp add: a1 orthogonal-complementI projection-orthogonal)
    moreover have  ‹u − (projection M) u ∈ N›
      by (simp add: d1 d2 complex-vector.subspace-diff x1)
   ultimately have d3: ‹u − (projection M) u ∈ ((orthogonal-complement M) ∩
N)›
      by simp
    hence ‹∃ v ∈ ((orthogonal-complement M) ∩ N). u = (projection M) u + v›
      by (metis d3 diff-add-cancel ordered-field-class.sign-simps(2))
     then obtain v where ‹v ∈ ((orthogonal-complement M) ∩ N)› and ‹u =
(projection M) u + v›
      by blast
    hence ‹u ∈ M + ((orthogonal-complement M) ∩ N)›
      by (metis d4 set-plus-intro)
    thus ?thesis
      unfolding closed-sum-def
      using closure-subset by blast
  qed

  have c3: N ⊆ M +_M ((orthogonal-complement M) ∩ N)
    if closed-csubspace M and closed-csubspace N and M ⊆ N
    for M N :: 'a set
    using c2 that by auto

  show sup X (inf (− X) Y) = Y
    if X ≤ Y
    using that apply transfer
    using c1 c3
    by (simp add: subset-antisym)

  show X − Y = inf X (− Y)
    apply transfer by simp
qed
```

9.6 Orthogonal spaces

definition ‹*orthogonal-spaces S T* ⟷ (∀ *x*∈*space-as-set S*. ∀ *y*∈*space-as-set T*. *is-orthogonal x y*)›

lemma *orthogonal-spaces-leq-compl*: ‹*orthogonal-spaces S T* ⟷ *S* ≤ − *T*›
unfolding *orthogonal-spaces-def* **apply** *transfer*
by (*auto simp*: *orthogonal-complement-def*)

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

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

lemma *orthogonal-spaces-sym*: ‹*orthogonal-spaces S T* ⟹ *orthogonal-spaces T S*›
unfolding *orthogonal-spaces-def*
using *is-orthogonal-sym* **by** *blast*

lemma *orthogonal-sup*: ‹*orthogonal-spaces S T1* ⟹ *orthogonal-spaces S T2* ⟹ *orthogonal-spaces S* (*sup T1 T2*)›
apply (*rule orthogonal-spaces-sym*)
apply (*simp add*: *orthogonal-spaces-leq-compl*)
using *orthogonal-spaces-leq-compl orthogonal-spaces-sym* **by** *blast*

lemma *orthogonal-sum*:
assumes ‹*finite F*› **and** ‹⋀*x*. *x*∈*F* ⟹ *orthogonal-spaces S* (*T x*)›
shows ‹*orthogonal-spaces S* (*sum T F*)›
using *assms*
apply *induction*
by (*auto intro*!: *orthogonal-sup*)

lemma *orthogonal-spaces-ccspan*: ‹(∀ *x*∈*S*. ∀ *y*∈*T*. *is-orthogonal x y*) ⟷ *orthogonal-spaces* (*ccspan S*) (*ccspan T*)›
by (*meson ccspan-leq-ortho-ccspan ccspan-superset orthogonal-spaces-def orthogonal-spaces-leq-compl subset-iff*)

lemma *orthogonal-spaces-SUP-left*:
assumes ‹⋀*x*. *x* ∈ *X* ⟹ *orthogonal-spaces* (*A x*) *B*›
shows ‹*orthogonal-spaces* (*SUP x*∈*X*. *A x*) *B*›
by (*meson SUP-least assms orthogonal-spaces-leq-compl*)

lemma *orthogonal-spaces-SUP-right*:
assumes ‹⋀*x*. *x* ∈ *X* ⟹ *orthogonal-spaces A* (*B x*)›
shows ‹*orthogonal-spaces A* (*SUP x*∈*X*. *B x*)›
by (*meson assms orthogonal-spaces-SUP-left orthogonal-spaces-sym*)

9.7 Orthonormal bases

lemma *ortho-basis-exists*:

fixes $S :: \langle 'a::chilbert\text{-}space\ set\rangle$
assumes $\langle is\text{-}ortho\text{-}set\ S\rangle$
shows $\langle \exists B.\ B \supseteq S \wedge is\text{-}ortho\text{-}set\ B \wedge closure\ (cspan\ B) = UNIV\rangle$
proof −
define *on* **where** $\langle on\ B \longleftrightarrow B \supseteq S \wedge is\text{-}ortho\text{-}set\ B\rangle$ **for** $B :: \langle 'a\ set\rangle$
have $\langle \exists B{\in}Collect\ on.\ \forall B'{\in}Collect\ on.\ B \subseteq B' \longrightarrow B' = B\rangle$
proof (*rule subset-Zorn-nonempty*; *simp*)
show $\langle \exists S.\ on\ S\rangle$
apply (*rule exI*[*of* - *S*])
using *assms on-def* **by** *fastforce*
next
fix $C :: \langle 'a\ set\ set\rangle$
assume $\langle C \neq \{\}\rangle$
assume $\langle subset.chain\ (Collect\ on)\ C\rangle$
then have *C-on*: $\langle B \in C \Longrightarrow on\ B\rangle$ **and** *C-order*: $\langle B \in C \Longrightarrow B' \in C \Longrightarrow B \subseteq B' \vee B' \subseteq B\rangle$ **for** *B B'*
by (*auto simp*: *subset.chain-def*)
have $\langle is\text{-}orthogonal\ x\ y\rangle$ **if** $\langle x{\in}\bigcup C\rangle$ $\langle y{\in}\bigcup C\rangle$ $\langle x \neq y\rangle$ **for** *x y*
by (*smt* (*verit*) *UnionE C-order C-on on-def is-ortho-set-def subsetD that*(*1*) *that*(*2*) *that*(*3*))
moreover have $\langle 0 \notin \bigcup\ C\rangle$
by (*meson UnionE C-on is-ortho-set-def on-def*)
moreover have $\langle \bigcup C \supseteq S\rangle$
using *C-on* $\langle C \neq \{\}\rangle$ *on-def* **by** *blast*
ultimately show $\langle on\ (\bigcup\ C)\rangle$
unfolding *on-def is-ortho-set-def* **by** *simp*
qed
then obtain *B* **where** $\langle on\ B\rangle$ **and** *B-max*: $\langle B' \supseteq B \Longrightarrow on\ B' \Longrightarrow B{=}B'\rangle$ **for** *B'*
by *auto*
have $\langle \psi = 0\rangle$ **if** $\psi ortho$: $\langle \forall b{\in}B.\ is\text{-}orthogonal\ \psi\ b\rangle$ **for** $\psi :: 'a$
proof (*rule ccontr*)
assume $\langle \psi \neq 0\rangle$
define $\varphi\ B'$ **where** $\langle \varphi = \psi\ /_R\ norm\ \psi\rangle$ **and** $\langle B' = B \cup \{\varphi\}\rangle$
have [*simp*]: $\langle norm\ \varphi = 1\rangle$
using $\langle \psi \neq 0\rangle$ **by** (*auto simp*: φ-*def*)
have $\varphi ortho$: $\langle is\text{-}orthogonal\ \varphi\ b\rangle$ **if** $\langle b \in B\rangle$ **for** *b*
using $\psi ortho$ *that* φ-*def* **by** *auto*
have *orthoB'*: $\langle is\text{-}orthogonal\ x\ y\rangle$ **if** $\langle x{\in}B'\rangle$ $\langle y{\in}B'\rangle$ $\langle x \neq y\rangle$ **for** *x y*
using *that* $\langle on\ B\rangle$ $\varphi ortho$ $\varphi ortho$[*THEN is-orthogonal-sym*[*THEN iffD1*]]
by (*auto simp*: *B'-def on-def is-ortho-set-def*)
have *B'0*: $\langle 0 \notin B'\rangle$
using *B'-def* $\langle norm\ \varphi = 1\rangle$ $\langle on\ B\rangle$ *is-ortho-set-def on-def* **by** *fastforce*
have $\langle S \subseteq B'\rangle$
using *B'-def* $\langle on\ B\rangle$ *on-def* **by** *auto*
from *orthoB' B'0* $\langle S \subseteq B'\rangle$ **have** $\langle on\ B'\rangle$
by (*simp add*: *on-def is-ortho-set-def*)
with *B-max* **have** $\langle B = B'\rangle$
by (*metis B'-def Un-upper1*)

```
    then have ‹φ ∈ B›
      using B'-def by blast
    then have ‹is-orthogonal φ φ›
      using φortho by blast
    then show False
      using B'0 ‹B = B'› ‹φ ∈ B› by fastforce
  qed
  then have ‹orthogonal-complement B = {0}›
    by (auto simp: orthogonal-complement-def)
  then have ‹UNIV = orthogonal-complement (orthogonal-complement B)›
    by simp
  also have ‹... = orthogonal-complement (orthogonal-complement (closure (cspan
B)))›
     by (metis (mono-tags, opaque-lifting) ‹orthogonal-complement B = {0}› cin-
ner-zero-left complex-vector.span-superset empty-iff insert-iff orthogonal-complementI
orthogonal-complement-antimono orthogonal-complement-of-closure subsetI subset-antisym)
  also have ‹... = closure (cspan B)›
    apply (rule double-orthogonal-complement-id)
    by simp
  finally have ‹closure (cspan B) = UNIV›
    by simp
  with ‹on B› show ?thesis
    by (auto simp: on-def)
qed

lemma orthonormal-basis-exists:
  fixes S :: ‹'a::chilbert-space set›
  assumes ‹is-ortho-set S› and ‹⋀x. x∈S ⟹ norm x = 1›
  shows ‹∃B. B ⊇ S ∧ is-onb B›
proof −
  from ‹is-ortho-set S›
  obtain B where ‹is-ortho-set B› and ‹B ⊇ S› and ‹closure (cspan B) = UNIV›
    using ortho-basis-exists by blast
  define B' where ‹B' = (λx. x /R norm x) ‘ B›
  have ‹S = (λx. x /R norm x) ‘ S›
    by (simp add: assms(2))
  then have ‹B' ⊇ S›
    using B'-def ‹S ⊆ B› by blast
  moreover
  have ‹ccspan B' = top›
    apply (transfer fixing: B')
    apply (simp add: B'-def scaleR-scaleC)
    apply (subst complex-vector.span-image-scale')
    using ‹is-ortho-set B› ‹closure (cspan B) = UNIV› is-ortho-set-def
    by auto
  moreover have ‹is-ortho-set B'›
    using ‹is-ortho-set B› by (auto simp: B'-def is-ortho-set-def)
  moreover have ‹∀b∈B'. norm b = 1›
    using ‹is-ortho-set B› apply (auto simp: B'-def is-ortho-set-def)
```

by (*metis field-class.field-inverse norm-eq-zero*)
ultimately show *?thesis*
by (*auto simp*: *is-onb-def*)
qed

definition *some-chilbert-basis* :: ‹'*a*::*chilbert-space set*› **where**
‹*some-chilbert-basis* = (*SOME B*::'*a set*. *is-onb B*)›

lemma *is-onb-some-chilbert-basis*[*simp*]: ‹*is-onb* (*some-chilbert-basis* :: '*a*::*chilbert-space set*)›
using *orthonormal-basis-exists*[*OF is-ortho-set-empty*]
by (*auto simp add*: *some-chilbert-basis-def intro*: *someI2*)

lemma *is-ortho-set-some-chilbert-basis*[*simp*]: ‹*is-ortho-set some-chilbert-basis*›
using *is-onb-def is-onb-some-chilbert-basis* **by** *blast*

lemma *is-normal-some-chilbert-basis*: ‹⋀*x*. *x* ∈ *some-chilbert-basis* ⟹ *norm x* = *1*›
using *is-onb-def is-onb-some-chilbert-basis* **by** *blast*

lemma *ccspan-some-chilbert-basis*[*simp*]: ‹*ccspan some-chilbert-basis* = *top*›
using *is-onb-def is-onb-some-chilbert-basis* **by** *blast*

lemma *span-some-chilbert-basis*[*simp*]: ‹*closure* (*cspan some-chilbert-basis*) = *UNIV*›
by (*metis ccspan.rep-eq ccspan-some-chilbert-basis top-ccsubspace.rep-eq*)

lemma *cindependent-some-chilbert-basis*[*simp*]: ‹*cindependent some-chilbert-basis*›
using *is-ortho-set-cindependent is-ortho-set-some-chilbert-basis* **by** *blast*

lemma *finite-some-chilbert-basis*[*simp*]: ‹*finite* (*some-chilbert-basis* :: '*a* :: {*chilbert-space*, *cfinite-dim*} *set*)›
apply (*rule cindependent-cfinite-dim-finite*)
by *simp*

lemma *some-chilbert-basis-nonempty*: ‹(*some-chilbert-basis* :: '*a*::{*chilbert-space*, *not-singleton*} *set*) ≠ {}›
proof (*rule ccontr*, *simp*)
define *B* :: ‹'*a set*› **where** ‹*B* = *some-chilbert-basis*›
assume [*simp*]: ‹*B* = {}›
have ‹*UNIV* = *closure* (*cspan B*)›
using *B-def span-some-chilbert-basis* **by** *blast*
also have ‹... = {*0*}›
by *simp*
also have ‹... ≠ *UNIV*›
using *Extra-General.UNIV-not-singleton* **by** *blast*
finally show *False*
by *simp*
qed

lemma *basis-projections-reconstruct-has-sum*:
assumes ‹*is-ortho-set B*› **and** *normB*: ‹$\bigwedge b.\ b{\in}B \Longrightarrow$ *norm b* = *1*› **and** ψB: ‹ψ $\in$ *space-as-set* (*ccspan B*)›
shows ‹((λ*b*. (*b* ·$_C$ ψ) *$_C$ *b*) *has-sum* ψ) *B*›
proof (*rule has-sumI-metric*)
fix *e* :: *real* **assume** ‹*e* > *0*›
define *e2* **where** ‹*e2* = *e*/*2*›
have [*simp*]: ‹*e2* > *0*›
by (*simp add*: ‹*0* < *e*› *e2-def*)
define *bb* **where** ‹*bb* φ *b* = (*b* ·$_C$ φ) *$_C$ *b*› **for** φ **and** *b* :: *'a*
have *linear-bb*: ‹*clinear* (λφ. *bb* φ *b*)› **for** *b*
by (*simp add*: *bb-def cinner-add-right clinear-iff scaleC-left.add*)
from ψB **obtain** φ **where** *dist*$\varphi\psi$: ‹*dist* φ ψ < *e2*› **and** φB: ‹φ $\in$ *cspan B*›
apply *atomize-elim* **apply** (*simp add*: *ccspan.rep-eq closure-approachable*)
using ‹*0* < *e2*› **by** *blast*
from φB **obtain** *F* **where** ‹*finite F*› **and** ‹*F* ⊆ *B*› **and** φF: ‹φ $\in$ *cspan F*›
by (*meson vector-finitely-spanned*)
have ‹*dist* (*sum* (*bb* ψ) *G*) ψ < *e*›
if ‹*finite G*› **and** ‹*F* ⊆ *G*› **and** ‹*G* ⊆ *B*› **for** *G*
proof −
have *sum*φ: ‹*sum* (*bb* φ) *G* = φ›
proof −
from φF ‹*F* ⊆ *G*› **have** φG: ‹φ $\in$ *cspan G*›
using *complex-vector.span-mono* **by** *blast*
then obtain *f* **where** φ*sum*: ‹φ = ($\sum$ *b*∈*G*. *f b* *$_C$ *b*)›
using *complex-vector.span-finite*[*OF* ‹*finite G*›]
by *auto*
have ‹*sum* (*bb* φ) *G* = ($\sum$ *c*∈*G*. $\sum$ *b*∈*G*. *bb* (*f b* *$_C$ *b*) *c*)›
apply (*simp add*: φ*sum*)
apply (*rule sum.cong*, *simp*)
apply (*rule complex-vector.linear-sum*[**where** *f*=‹λ*x*. *bb x* ·›])
by (*rule linear-bb*)
also have ‹... = ($\sum$ (*c*,*b*)∈*G*×*G*. *bb* (*f b* *$_C$ *b*) *c*)›
by (*simp add*: *sum.cartesian-product*)
also have ‹... = ($\sum$ *b*∈*G*. *f b* *$_C$ *b*)›
apply (*rule sym*)
apply (*rule sum.reindex-bij-witness-not-neutral*
[**where** *j*=‹λ*b*. (*b*,*b*)› **and** *i*=*fst* **and** *T'*=‹*G*×*G* − (λ*b*. (*b*,*b*)) ‘ *G*› **and** *S'*=‹{}›])
using ‹*finite G*› **apply** (*auto simp*: *bb-def*)
apply (*metis* (*no-types*, *lifting*) *assms*(*1*) *imageI is-ortho-set-antimono is-ortho-set-def that*(*3*))
using *normB* ‹*G* ⊆ *B*› *cnorm-eq-1* **by** *blast*
also have ‹... = φ›
by (*simp flip*: φ*sum*)
finally show *?thesis*
by −
qed

have ‹*dist* (*sum* (*bb* φ) *G*) (*sum* (*bb* ψ) *G*) < *e2*›
proof −
define γ **where** ‹$\gamma = \varphi - \psi$›
have ‹(*dist* (*sum* (*bb* φ) *G*) (*sum* (*bb* ψ) *G*))^2 = (*norm* (*sum* (*bb* γ) *G*))^2›
by (*simp add*: *dist-norm* γ*-def complex-vector.linear-diff*[*OF linear-bb*] *sum-subtractf*)
also have ‹... = (*norm* (*sum* (*bb* γ) *G*))^2 + (*norm* (γ − *sum* (*bb* γ) *G*))^2 − (*norm* (γ − *sum* (*bb* γ) *G*))^2›
by *simp*
also have ‹... = (*norm* (*sum* (*bb* γ) *G* + (γ − *sum* (*bb* γ) *G*)))^2 − (*norm* (γ − *sum* (*bb* γ) *G*))^2›
proof −
have ‹($\sum$ *b*∈*G*. *bb* γ *b* ·$_C$ *bb* γ *c*) = *bb* γ *c* ·$_C$ γ› **if** ‹*c* ∈ *G*› **for** *c*
apply (*subst sum-single*[**where** *i*=*c*])
using *that* **apply** (*auto intro*!: ‹*finite G*› *simp*: *bb-def*)
apply (*metis* ‹*G* ⊆ *B*› ‹*is-ortho-set B*› *is-ortho-set-antimono is-ortho-set-def*)
using ‹*G* ⊆ *B*› *normB cnorm-eq-1* **by** *blast*
then have ‹*is-orthogonal* (*sum* (*bb* γ) *G*) (γ − *sum* (*bb* γ) *G*)›
by (*simp add*: *cinner-sum-left cinner-diff-right cinner-sum-right*)
then show *?thesis*
apply (*rule-tac arg-cong*[**where** *f*=‹$\lambda x.\ x$ - ›])
by (*rule pythagorean-theorem*[*symmetric*])
qed
also have ‹... = (*norm* γ)^2 − (*norm* (γ − *sum* (*bb* γ) *G*))^2›
by *simp*
also have ‹... ≤ (*norm* γ)^2›
by *simp*
also have ‹... = (*dist* φ ψ)^2›
by (*simp add*: γ*-def dist-norm*)
also have ‹... < *e2*^{2}›
using *dist*$\varphi\psi$ **apply** (*rule power-strict-mono*)
by *auto*
finally show *?thesis*
by (*smt* (*verit*) ‹*0* < *e2*› *power-mono*)
qed
with *sum*φ *dist*$\varphi\psi$ **show** ‹*dist* (*sum* (*bb* ψ) *G*) ψ < *e*›
apply (*rule-tac dist-triangle-lt*[**where** *z*=φ])
by (*simp add*: *e2-def dist-commute*)
qed
with ‹*finite F*› **and** ‹*F* ⊆ *B*›
show ‹∃ *F*. *finite F* ∧
F ⊆ *B* ∧ (∀ *G*. *finite G* ∧ *F* ⊆ *G* ∧ *G* ⊆ *B* ⟶ *dist* (*sum* (*bb* ψ) *G*) ψ < *e*)›
by (*auto intro*!: *exI*[*of* - *F*])
qed

lemma *basis-projections-reconstruct*:
assumes ‹*is-ortho-set B*› **and** ‹⋀*b*. *b*∈*B* ⟹ *norm b* = *1*› **and** ‹ψ ∈ *space-as-set* (*ccspan B*)›

shows ‹$(\sum_\infty b{\in}B.\ (b \cdot_C \psi) *_C b) = \psi$›
using *assms basis-projections-reconstruct-has-sum infsumI* **by** *blast*

lemma *basis-projections-reconstruct-summable*:
assumes ‹*is-ortho-set B*› **and** ‹$\bigwedge b.\ b{\in}B \Longrightarrow$ *norm b* $= 1$› **and** ‹$\psi \in$ *space-as-set* (*ccspan B*)›
shows ‹$(\lambda b.\ (b \cdot_C \psi) *_C b)$ *summable-on B*›
by (*simp add*: *assms basis-projections-reconstruct basis-projections-reconstruct-has-sum summable-iff-has-sum-infsum*)

lemma *parseval-identity-has-sum*:
assumes ‹*is-ortho-set B*› **and** *normB*: ‹$\bigwedge b.\ b{\in}B \Longrightarrow$ *norm b* $= 1$› **and** ‹$\psi \in$ *space-as-set* (*ccspan B*)›
shows ‹$((\lambda b.\ (\mathit{norm}\ (b \cdot_C \psi))^2)$ *has-sum* $(\mathit{norm}\ \psi)^2)$ *B*›
proof −
have $*$: ‹$(\lambda v.\ (\mathit{norm}\ v)^2)\ (\sum b{\in}F.\ (b \cdot_C \psi) *_C b) = (\sum b{\in}F.\ (\mathit{norm}\ (b \cdot_C \psi))^2)$›
if ‹*finite F*› **and** ‹$F \subseteq B$› **for** *F*
apply (*subst pythagorean-theorem-sum*)
using ‹*is-ortho-set B*› *normB that*
apply (*auto intro*!: *sum.cong*[*OF refl*] *simp*: *is-ortho-set-def*)
by *blast*

from *assms* **have** ‹$((\lambda b.\ (b \cdot_C \psi) *_C b)$ *has-sum* $\psi)$ *B*›
by (*simp add*: *basis-projections-reconstruct-has-sum*)
then have ‹$((\lambda F.\ \sum b{\in}F.\ (b \cdot_C \psi) *_C b) \longrightarrow \psi)$ (*finite-subsets-at-top B*)›
by (*simp add*: *has-sum-def*)
then have ‹$((\lambda F.\ (\lambda v.\ (\mathit{norm}\ v)^2)\ (\sum b{\in}F.\ (b \cdot_C \psi) *_C b)) \longrightarrow (\mathit{norm}\ \psi)^2)$ (*finite-subsets-at-top B*)›
apply (*rule isCont-tendsto-compose*[*rotated*])
by *simp*
then have ‹$((\lambda F.\ (\sum b{\in}F.\ (\mathit{norm}\ (b \cdot_C \psi))^2)) \longrightarrow (\mathit{norm}\ \psi)^2)$ (*finite-subsets-at-top B*)›
apply (*rule tendsto-cong*[*THEN iffD2*, *rotated*])
apply (*rule eventually-finite-subsets-at-top-weakI*)
by (*simp add*: $*$)
then show ‹$((\lambda b.\ (\mathit{norm}\ (b \cdot_C \psi))^2)$ *has-sum* $(\mathit{norm}\ \psi)^2)$ *B*›
by (*simp add*: *has-sum-def*)
qed

lemma *parseval-identity-summable*:
assumes ‹*is-ortho-set B*› **and** ‹$\bigwedge b.\ b{\in}B \Longrightarrow$ *norm b* $= 1$› **and** ‹$\psi \in$ *space-as-set* (*ccspan B*)›
shows ‹$(\lambda b.\ (\mathit{norm}\ (b \cdot_C \psi))^2)$ *summable-on B*›
using *parseval-identity-has-sum*[*OF assms*] *has-sum-imp-summable* **by** *blast*

lemma *parseval-identity*:
assumes ‹*is-ortho-set B*› **and** ‹$\bigwedge b.\ b{\in}B \Longrightarrow$ *norm b* $= 1$› **and** ‹$\psi \in$ *space-as-set* (*ccspan B*)›
shows ‹$(\sum_\infty b{\in}B.\ (\mathit{norm}\ (b \cdot_C \psi))^2) = (\mathit{norm}\ \psi)^2$›

using *parseval-identity-has-sum*[*OF assms*]
using *infsumI* **by** *blast*

9.8 Riesz-representation theorem

lemma *orthogonal-complement-kernel-functional*:
fixes f :: ‹'*a*::*complex-inner* ⇒ *complex*›
assumes ‹*bounded-clinear f*›
shows ‹∃ *x*. *orthogonal-complement* (*f* −‘ {*0*}) = *cspan* {*x*}›
proof (*cases* ‹*orthogonal-complement* (*f* −‘ {*0*}) = {*0*}›)
case *True*
then show *?thesis*
apply (*rule-tac x=0* **in** *exI*) **by** *auto*
next
case *False*
then obtain *x* **where** *xortho*: ‹*x* ∈ *orthogonal-complement* (*f* −‘ {*0*})› **and** *xnon0*: ‹*x* ≠ *0*›
using *complex-vector.subspace-def* **by** *fastforce*

from *xnon0 xortho*
have *r1*: ‹*f x* ≠ *0*›
by (*metis cinner-eq-zero-iff orthogonal-complement-orthoI vimage-singleton-eq*)

have ‹∃ *k*. *y* = *k* *$_C$ *x*› **if** ‹*y* ∈ *orthogonal-complement* (*f* −‘ {*0*})› **for** *y*
proof (*cases* ‹*y* = *0*›)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
with *that*
have ‹*f y* ≠ *0*›
by (*metis cinner-eq-zero-iff orthogonal-complement-orthoI vimage-singleton-eq*)
then obtain *k* **where** *k-def*: ‹*f x* = *k* ∗ *f y*›
by (*metis add.inverse-inverse minus-divide-eq-eq*)
with *assms* **have** ‹*f x* = *f* (*k* *$_C$ *y*)›
by (*simp add*: *bounded-clinear.axioms*(*1*) *clinear.scaleC*)
hence ‹*f x* − *f* (*k* *$_C$ *y*) = *0*›
by *simp*
with *assms* **have** *s1*: ‹*f* (*x* − *k* *$_C$ *y*) = *0*›
by (*simp add*: *bounded-clinear.axioms*(*1*) *complex-vector.linear-diff*)
from *that* **have** ‹*k* *$_C$ *y* ∈ *orthogonal-complement* (*f* −‘ {*0*})›
by (*simp add*: *complex-vector.subspace-scale*)
with *xortho* **have** *s2*: ‹*x* − (*k* *$_C$ *y*) ∈ *orthogonal-complement* (*f* −‘ {*0*})›
by (*simp add*: *complex-vector.subspace-diff*)
have *s3*: ‹(*x* − (*k* *$_C$ *y*)) ∈ *f* −‘ {*0*}›
using *s1* **by** *simp*
moreover have ‹(*f* −‘ {*0*}) ∩ (*orthogonal-complement* (*f* −‘ {*0*})) = {*0*}›
by (*meson assms closed-csubspace-def complex-vector.subspace-def kernel-is-closed-csubspace orthogonal-complement-zero-intersection*)

```
    ultimately have ‹x − (k *C y) = 0›
      using s2 by blast
    thus ?thesis
      by (metis ceq-vector-fraction-iff eq-iff-diff-eq-0 k-def r1 scaleC-scaleC)
  qed
  then have ‹orthogonal-complement (f −‘ {0}) ⊆ cspan {x}›
    using complex-vector.span-superset complex-vector.subspace-scale by blast

  moreover from xortho have ‹orthogonal-complement (f −‘ {0}) ⊇ cspan {x}›
    by (simp add: complex-vector.span-minimal)

  ultimately show ?thesis
    by auto
qed

lemma riesz-representation-existence:
  — Theorem 3.4 in [1]
  fixes f :: ‹'a::chilbert-space ⇒ complex›
  assumes a1: ‹bounded-clinear f›
  shows ‹∃ t. ∀ x.  f x = t ·C x›
proof(cases ‹∀  x. f x = 0›)
  case True
  thus ?thesis
    by (metis cinner-zero-left)
next
  case False
  obtain t where spant: ‹orthogonal-complement (f −‘ {0}) = cspan {t}›
    using orthogonal-complement-kernel-functional
    using assms by blast
  have ‹projection (orthogonal-complement (f −‘ {0})) x = ((t ·C x)/(t ·C t)) *C
t› for x
    apply (subst spant) by (rule projection-rank1)
  hence ‹f (projection (orthogonal-complement (f −‘ {0})) x) = (((t ·C x))/(t ·C
t)) * (f t)› for x
    using a1 unfolding bounded-clinear-def
    by (simp add: complex-vector.linear-scale)
  hence l2: ‹f (projection (orthogonal-complement (f −‘ {0})) x) = ((cnj (f t)/(t
·C t)) *C t) ·C x› for x
    using complex-cnj-divide by force
  have ‹f (projection (f −‘ {0}) x) = 0› for x
    by (metis (no-types, lifting) assms bounded-clinear-def closed-csubspace.closed
      complex-vector.linear-subspace-vimage complex-vector.subspace-0 complex-vector.subspace-single-0
        csubspace-is-convex insert-absorb insert-not-empty kernel-is-closed-csubspace
projection-in-image vimage-singleton-eq)
  hence ⋀a b. f (projection (f −‘ {0}) a + b) = 0 + f b
    using additive.add assms
    by (simp add: bounded-clinear-def complex-vector.linear-add)
  hence ⋀a. 0 + f (projection (orthogonal-complement (f −‘ {0})) a) = f a
    apply (simp add: assms)
```

by (*metis add.commute diff-add-cancel*)
hence ‹*f x* = ((*cnj* (*f t*)/(*t* ·$_C$ *t*)) *$_C$ *t*) ·$_C$ *x*› **for** *x*
by (*simp add*: *l2*)
thus *?thesis*
by *blast*
qed

lemma *riesz-representation-unique*:
— Theorem 3.4 in [1]
fixes *f*::‹'*a*::*complex-inner* ⇒ *complex*›
assumes ‹⋀*x*. *f x* = (*t* ·$_C$ *x*)›
assumes ‹⋀*x*. *f x* = (*u* ·$_C$ *x*)›
shows ‹*t* = *u*›
by (*metis add-diff-cancel-left' assms*(*1*) *assms*(*2*) *cinner-diff-left cinner-gt-zero-iff diff-add-cancel diff-zero*)

9.9 Adjoints

definition ‹*is-cadjoint F G* ⟷ (∀ *x y*. (*F x* ·$_C$ *y*) = (*x* ·$_C$ *G y*))›

lemma *is-adjoint-sym*:
‹*is-cadjoint F G* ⟹ *is-cadjoint G F*›
unfolding *is-cadjoint-def* **apply** *auto*
by (*metis cinner-commute'*)

definition ‹*cadjoint G* = (*SOME F*. *is-cadjoint F G*)›
for *G* :: '*b*::*complex-inner* ⇒ '*a*::*complex-inner*

lemma *cadjoint-exists*:
fixes *G* :: '*b*::*chilbert-space* ⇒ '*a*::*complex-inner*
assumes [*simp*]: ‹*bounded-clinear G*›
shows ‹∃ *F*. *is-cadjoint F G*›
proof −
include *norm-syntax*
have [*simp*]: ‹*clinear G*›
using *assms* **unfolding** *bounded-clinear-def* **by** *blast*
define *g* :: ‹'*a* ⇒ '*b* ⇒ *complex*›
where ‹*g x y* = (*x* ·$_C$ *G y*)› **for** *x y*
have ‹*bounded-clinear* (*g x*)› **for** *x*
proof −
have ‹*g x* (*a* + *b*) = *g x a* + *g x b*› **for** *a b*
unfolding *g-def*
using *additive.add cinner-add-right clinear-def*
by (*simp add*: *cinner-add-right complex-vector.linear-add*)
moreover have ‹*g x* (*k* *$_C$ *a*) = *k* *$_C$ (*g x a*)›
for *a k*
unfolding *g-def*
by (*simp add*: *complex-vector.linear-scale*)
ultimately have ‹*clinear* (*g x*)›

by (*simp add*: *clinearI*)
moreover
have ‹∃ *M*. ∀ *y*. ‖ *G y* ‖ ≤ ‖ *y* ‖ ∗ *M*›
using ‹*bounded-clinear G*›
unfolding *bounded-clinear-def bounded-clinear-axioms-def* **by** *blast*
then have ‹∃ *M*. ∀ *y*. ‖ *g x y* ‖ ≤ ‖ *y* ‖ ∗ *M*›
using *g-def*
by (*simp add*: *bounded-clinear.bounded bounded-clinear-cinner-right-comp*)
ultimately show *?thesis* **unfolding** *bounded-linear-def*
using *bounded-clinear.intro*
using *bounded-clinear-axioms-def* **by** *blast*
qed
hence ‹∀ *x*. ∃ *t*. ∀ *y*. *g x y* = (*t* ·$_C$ *y*)›
using *riesz-representation-existence* **by** *blast*
then obtain *F* **where** ‹∀ *x*. ∀ *y*. *g x y* = (*F x* ·$_C$ *y*)›
by *metis*
then have ‹*is-cadjoint F G*›
unfolding *is-cadjoint-def g-def* **by** *simp*
thus *?thesis*
by *auto*
qed

lemma *cadjoint-is-cadjoint*[*simp*]:
fixes *G* :: *'b*::*chilbert-space* ⇒ *'a*::*complex-inner*
assumes [*simp*]: ‹*bounded-clinear G*›
shows ‹*is-cadjoint* (*cadjoint G*) *G*›
by (*metis assms cadjoint-def cadjoint-exists someI-ex*)

lemma *is-cadjoint-unique*:
assumes ‹*is-cadjoint F1 G*›
assumes ‹*is-cadjoint F2 G*›
shows ‹*F1* = *F2*›
by (*metis* (*full-types*) *assms*(*1*) *assms*(*2*) *ext is-cadjoint-def riesz-representation-unique*)

lemma *cadjoint-univ-prop*:
fixes *G* :: *'b*::*chilbert-space* ⇒ *'a*::*complex-inner*
assumes *a1*: ‹*bounded-clinear G*›
shows ‹*cadjoint G x* ·$_C$ *y* = *x* ·$_C$ *G y*›
using *assms cadjoint-is-cadjoint is-cadjoint-def* **by** *blast*

lemma *cadjoint-univ-prop'*:
fixes *G* :: *'b*::*chilbert-space* ⇒ *'a*::*complex-inner*
assumes *a1*: ‹*bounded-clinear G*›
shows ‹*x* ·$_C$ *cadjoint G y* = *G x* ·$_C$ *y*›
by (*metis cadjoint-univ-prop assms cinner-commute'*)

notation *cadjoint* (‹-†› [*99*] *100*)

lemma *cadjoint-eqI*:

fixes G:: ‹$'b$::*complex-inner* ⇒ $'a$::*complex-inner*›
and F:: ‹$'a$ ⇒ $'b$›
assumes ‹$\bigwedge x\ y.\ (F\ x \cdot_C\ y) = (x \cdot_C\ G\ y)$›
shows ‹$G^\dagger = F$›
by (*metis assms cadjoint-def is-cadjoint-def is-cadjoint-unique someI-ex*)

lemma *cadjoint-bounded-clinear*:
fixes A :: $'a$::*chilbert-space* ⇒ $'b$::*complex-inner*
assumes *a1*: *bounded-clinear A*
shows ‹*bounded-clinear* $(A^\dagger)$›
proof
include *norm-syntax*
have *b1*: ‹$((A^\dagger)\ x \cdot_C\ y) = (x \cdot_C\ A\ y)$› **for** *x y*
using *cadjoint-univ-prop a1* **by** *auto*
have ‹*is-orthogonal* $((A^\dagger)\ (x1 + x2) - ((A^\dagger)\ x1 + (A^\dagger)\ x2))\ y$› **for** *x1 x2 y*
by (*simp add*: *b1 cinner-diff-left cinner-add-left*)
hence *b2*: ‹$(A^\dagger)\ (x1 + x2) - ((A^\dagger)\ x1 + (A^\dagger)\ x2) = 0$› **for** *x1 x2*
using *cinner-eq-zero-iff* **by** *blast*
thus *z1*: ‹$(A^\dagger)\ (x1 + x2) = (A^\dagger)\ x1 + (A^\dagger)\ x2$› **for** *x1 x2*
by (*simp add*: *b2 eq-iff-diff-eq-0*)

have *f1*: ‹*is-orthogonal* $((A^\dagger)\ (r *_C\ x) - (r *_C\ (A^\dagger)\ x))\ y$› **for** *r x y*
by (*simp add*: *b1 cinner-diff-left*)
thus *z2*: ‹$(A^\dagger)\ (r *_C\ x) = r *_C\ (A^\dagger)\ x$› **for** *r x*
using *cinner-eq-zero-iff eq-iff-diff-eq-0* **by** *blast*
have ‹$\| (A^\dagger)\ x \|\hat{}2 = ((A^\dagger)\ x \cdot_C\ (A^\dagger)\ x)$› **for** *x*
by (*metis cnorm-eq-square*)
moreover have ‹$\| (A^\dagger)\ x \|\hat{}2 \geq 0$› **for** *x*
by *simp*
ultimately have ‹$\| (A^\dagger)\ x \|\hat{}2 = |\ ((A^\dagger)\ x \cdot_C\ (A^\dagger)\ x)\ |$› **for** *x*
by (*metis abs-pos cinner-ge-zero*)
hence ‹$\| (A^\dagger)\ x \|\hat{}2 = |\ (x \cdot_C\ A\ ((A^\dagger)\ x))\ |$› **for** *x*
by (*simp add*: *b1*)
moreover have ‹$|(x \cdot_C\ A\ ((A^\dagger)\ x))| \leq \|x\| *\ \|A\ ((A^\dagger)\ x)\|$› **for** *x*
by (*simp add*: *abs-complex-def complex-inner-class.Cauchy-Schwarz-ineq2 less-eq-complex-def*)
ultimately have *b5*: ‹$\| (A^\dagger)\ x \|\hat{}2 \leq \|x\| * \|A\ ((A^\dagger)\ x)\|$› **for** *x*
by (*metis complex-of-real-mono-iff*)
have ‹$\exists M.\ M \geq 0 \wedge (\forall\ x.\ \|A\ ((A^\dagger)\ x)\| \leq M *\ \|(A^\dagger)\ x\|)$›
using *a1*
by (*metis* (*mono-tags*, *opaque-lifting*) *bounded-clinear.bounded linear mult-nonneg-nonpos mult-zero-right norm-ge-zero order.trans semiring-normalization-rules*(*7*))
then obtain *M* **where** *q1*: ‹$M \geq 0$› **and** *q2*: ‹$\forall\ x.\ \|A\ ((A^\dagger)\ x)\| \leq M * \|(A^\dagger)\ x\|$›
by *blast*
have ‹$\forall\ x$::$'b.\ \|x\| \geq 0$›
by *simp*
hence *b6*: ‹$\|x\| * \|A\ ((A^\dagger)\ x)\| \leq\ \|x\| * M * \|(A^\dagger)\ x\|$› **for** *x*
using *q2*
by (*smt ordered-comm-semiring-class.comm-mult-left-mono vector-space-over-itself.scale-scale*)

have *z3*: ‹‖ $(A^\dagger)$ x ‖ ≤ ‖x‖ * M› **for** x
proof(*cases* ‹‖$(A^\dagger)$ x‖ = *0*›)
case *True*
thus *?thesis*
by (*simp add*: ‹*0* ≤ M›)
next
case *False*
have ‹‖ $(A^\dagger)$ x ‖^2 ≤ ‖x‖ * M * ‖$(A^\dagger)$ x‖›
by (*smt b5 b6*)
thus *?thesis*
by (*smt False mult-right-cancel mult-right-mono norm-ge-zero semiring-normalization-rules*(*29*))
qed
thus ‹∃ K. ∀ x. ‖$(A^\dagger)$ x‖ ≤ ‖x‖ * K›
by *auto*
qed

proposition *double-cadjoint*:
fixes U :: ‹'*a*::*chilbert-space* ⇒ '*b*::*complex-inner*›
assumes *a1*: *bounded-clinear* U
shows $U^{\dagger\dagger} = U$
by (*metis assms cadjoint-def cadjoint-is-cadjoint is-adjoint-sym is-cadjoint-unique someI-ex*)

lemma *cadjoint-id*[*simp*]: ‹$id^\dagger = id$›
by (*simp add*: *cadjoint-eqI id-def*)

lemma *scaleC-cadjoint*:
fixes A::'*a*::*chilbert-space* ⇒ '*b*::*complex-inner*
assumes *bounded-clinear* A
shows ‹$(\lambda t.\ a *_C A\ t)^\dagger = (\lambda s.\ cnj\ a *_C (A^\dagger)\ s)$›
proof −
have *b3*: ‹$((\lambda\ s.\ (cnj\ a) *_C ((A^\dagger)\ s))\ x \cdot_C y) = (x \cdot_C (\lambda\ t.\ a *_C (A\ t))\ y)$›
for x y
by (*simp add*: *assms cadjoint-univ-prop*)

have $((\lambda t.\ a *_C A\ t)^\dagger)\ b = cnj\ a *_C (A^\dagger)\ b$
for b::'*b*
proof−
have *bounded-clinear* $(\lambda t.\ a *_C A\ t)$
by (*simp add*: *assms bounded-clinear-const-scaleC*)
thus *?thesis*
by (*metis* (*no-types*) *cadjoint-eqI b3*)
qed
thus *?thesis*
by *blast*
qed

lemma *is-projection-on-is-cadjoint*:

fixes M :: ‹$'a$::*complex-inner set*›
assumes *a1*: ‹*is-projection-on* π M› **and** *a2*: ‹*closed-csubspace* M›
shows ‹*is-cadjoint* π π›
by (*smt* (*verit*, *ccfv-threshold*) *a1 a2 cinner-diff-left cinner-eq-flip is-cadjoint-def is-projection-on-iff-orthog orthogonal-complement-orthoI right-minus-eq*)

lemma *is-projection-on-cadjoint*:
fixes M :: ‹$'a$::*complex-inner set*›
assumes ‹*is-projection-on* π M› **and** ‹*closed-csubspace* M›
shows ‹$\pi^\dagger = \pi$›
using *assms is-projection-on-is-cadjoint cadjoint-eqI is-cadjoint-def* **by** *blast*

lemma *projection-cadjoint*:
fixes M :: ‹$'a$::*chilbert-space set*›
assumes ‹*closed-csubspace* M›
shows ‹(*projection* M)† = *projection* M›
using *is-projection-on-cadjoint assms*
by (*metis closed-csubspace.closed closed-csubspace.subspace csubspace-is-convex empty-iff orthog-proj-exists projection-is-projection-on*)

9.10 More projections

These lemmas logically belong in the "projections" section above but depend on lemmas developed later.

lemma *is-projection-on-plus*:
assumes $\bigwedge x\ y.\ x \in A \Longrightarrow y \in B \Longrightarrow$ *is-orthogonal* x y
assumes ‹*closed-csubspace* A›
assumes ‹*closed-csubspace* B›
assumes ‹*is-projection-on* πA A›
assumes ‹*is-projection-on* πB B›
shows ‹*is-projection-on* $(\lambda x.\ \pi A\ x + \pi B\ x)$ $(A +_M B)$›
proof (*rule is-projection-on-iff-orthog*[*THEN iffD2*, *rule-format*])
show *clAB*: ‹*closed-csubspace* $(A +_M B)$›
by (*simp add*: *assms*(*2*) *assms*(*3*) *closed-subspace-closed-sum*)
fix h
have *1*: ‹$\pi A\ h + \pi B\ h \in A +_M B$›
by (*meson clAB assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *closed-csubspace-def closed-sum-left-subset closed-sum-right-subset complex-vector.subspace-def in-mono is-projection-on-in-image*)

have ‹$\pi A\ (\pi B\ h) = 0$›
by (*smt* (*verit*, *del-insts*) *assms*(*1*) *assms*(*2*) *assms*(*4*) *assms*(*5*) *cinner-eq-zero-iff is-cadjoint-def is-projection-on-in-image is-projection-on-is-cadjoint*)
then have ‹$h - (\pi A\ h + \pi B\ h) = (h - \pi B\ h) - \pi A\ (h - \pi B\ h)$›
by (*smt* (*verit*) *add.right-neutral add-diff-cancel-left' assms*(*2*) *assms*(*4*) *closed-csubspace.subspace complex-vector.subspace-diff diff-add-eq-diff-diff-swap diff-diff-add is-projection-on-iff-orthog orthog-proj-unique orthogonal-complement-closed-subspace*)
also have ‹$\ldots \in$ *orthogonal-complement* A›
using *assms*(*2*) *assms*(*4*) *is-projection-on-iff-orthog* **by** *blast*

finally have *orthoA*: ‹$h - (\pi A\ h + \pi B\ h) \in$ *orthogonal-complement* A›
by −

have ‹$\pi B\ (\pi A\ h) = 0$›
by (*smt* (*verit*, *del-insts*) *assms*(*1*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *cinner-eq-zero-iff* *is-cadjoint-def is-projection-on-in-image is-projection-on-is-cadjoint*)
then have ‹$h - (\pi A\ h + \pi B\ h) = (h - \pi A\ h) - \pi B\ (h - \pi A\ h)$›
by (*smt* (*verit*) *add.right-neutral add-diff-cancel assms*(*3*) *assms*(*5*) *closed-csubspace.subspace* *complex-vector.subspace-diff diff-add-eq-diff-diff-swap diff-diff-add is-projection-on-iff-orthog* *orthog-proj-unique orthogonal-complement-closed-subspace*)
also have ‹$\ldots \in$ *orthogonal-complement* B›
using *assms*(*3*) *assms*(*5*) *is-projection-on-iff-orthog* **by** *blast*
finally have *orthoB*: ‹$h - (\pi A\ h + \pi B\ h) \in$ *orthogonal-complement* B›
by −

from *orthoA orthoB*
have *2*: ‹$h - (\pi A\ h + \pi B\ h) \in$ *orthogonal-complement* $(A +_M B)$›
by (*metis IntI assms*(*2*) *assms*(*3*) *closed-csubspace-def complex-vector.subspace-def* *de-morgan-orthogonal-complement-plus*)

from *1 2* **show** ‹$h - (\pi A\ h + \pi B\ h) \in$ *orthogonal-complement* $(A +_M B) \wedge \pi A$ $h + \pi B\ h \in A +_M B$›
by *simp*
qed

lemma *projection-plus*:
fixes $A\ B ::$ *'a*::*chilbert-space set*
assumes $\bigwedge x\ y.\ x{:}A \Longrightarrow y{:}B \Longrightarrow$ *is-orthogonal* $x\ y$
assumes ‹*closed-csubspace* A›
assumes ‹*closed-csubspace* B›
shows ‹*projection* $(A +_M B) = (\lambda x.$ *projection* $A\ x +$ *projection* $B\ x)$›
proof −
have ‹*is-projection-on* $(\lambda x.$ *projection* $A\ x +$ *projection* $B\ x)\ (A +_M B)$›
apply (*rule is-projection-on-plus*)
using *assms* **by** *auto*
then show *?thesis*
by (*meson assms*(*2*) *assms*(*3*) *closed-csubspace.subspace closed-subspace-closed-sum* *csubspace-is-convex projection-eqI'*)
qed

lemma *is-projection-on-insert*:
assumes *ortho*: $\bigwedge s.\ s \in S \Longrightarrow$ *is-orthogonal* $a\ s$
assumes ‹*is-projection-on* π (*closure* (*cspan* S))›
assumes ‹*is-projection-on* πa (*cspan* $\{a\}$)›
shows *is-projection-on* $(\lambda x.\ \pi a\ x + \pi\ x)$ (*closure* (*cspan* (*insert* $a\ S$)))
proof −
from *ortho*
have ‹$x \in$ *cspan* $\{a\} \Longrightarrow y \in$ *closure* (*cspan* $S) \Longrightarrow$ *is-orthogonal* $x\ y$› **for** $x\ y$
using *is-orthogonal-cspan is-orthogonal-closure is-orthogonal-sym*

by (*smt* (*verit*, *ccfv-threshold*) *empty-iff insert-iff*)
then have ‹*is-projection-on* (λ*x*. π*a x* + π *x*) (*cspan* {*a*} +$_M$ *closure* (*cspan S*))›
apply (*rule is-projection-on-plus*)
using *assms* **by** (*auto simp add*: *closed-csubspace.intro*)
also have ‹... = *closure* (*cspan* (*insert a S*))›
using *closed-sum-cspan*[**where** *X*=‹{*a*}›] **by** *simp*
finally show *?thesis*
by −
qed

lemma *projection-insert*:
fixes *a* :: ‹'*a*::*chilbert-space*›
assumes *a1*: ⋀*s*. *s* ∈ *S* ⟹ *is-orthogonal a s*
shows *projection* (*closure* (*cspan* (*insert a S*))) *u*
= *projection* (*cspan* {*a*}) *u* + *projection* (*closure* (*cspan S*)) *u*
using *is-projection-on-insert*[**where** *S*=*S*, *OF a1*]
by (*metis* (*no-types*, *lifting*) *closed-closure closed-csubspace.intro closure-is-csubspace complex-vector.subspace-span csubspace-is-convex finite.intros*(*1*) *finite.intros*(*2*) *finite-cspan-closed-csubspace projection-eqI' projection-is-projection-on'*)

lemma *projection-insert-finite*:
fixes *S* :: ‹'*a*::*chilbert-space set*›
assumes *a1*: ⋀*s*. *s* ∈ *S* ⟹ *is-orthogonal a s* **and** *a2*: *finite S*
shows *projection* (*cspan* (*insert a S*)) *u*
= *projection* (*cspan* {*a*}) *u* + *projection* (*cspan S*) *u*
using *projection-insert*
by (*metis a1 a2 closure-finite-cspan finite.insertI*)

9.11 Canonical basis (*onb-enum*)

setup ‹*Sign.add-const-constraint* (***const-name*** ‹*is-ortho-set*›, *SOME* ***typ*** ‹'*a set* ⇒ *bool*›)›

class *onb-enum* = *basis-enum* + *complex-inner* +
assumes *is-orthonormal*: *is-ortho-set* (*set canonical-basis*)
and *is-normal*: ⋀*x*. *x* ∈ (*set canonical-basis*) ⟹ *norm x* = *1*

setup ‹*Sign.add-const-constraint* (***const-name*** ‹*is-ortho-set*›, *SOME* ***typ*** ‹'*a*::*complex-inner set* ⇒ *bool*›)›

lemma *cinner-canonical-basis*:
assumes ‹*i* < *length* (*canonical-basis* :: '*a*::*onb-enum list*)›
assumes ‹*j* < *length* (*canonical-basis* :: '*a*::*onb-enum list*)›
shows ‹*cinner* (*canonical-basis*!*i* :: '*a*) (*canonical-basis*!*j*) = (*if i*=*j then 1 else 0*)›
by (*metis assms*(*1*) *assms*(*2*) *distinct-canonical-basis is-normal is-ortho-set-def is-orthonormal nth-eq-iff-index-eq nth-mem of-real-1 power2-norm-eq-cinner power-one*)

lemma *canonical-basis-is-onb*[*simp*]: ‹*is-onb* (*set canonical-basis* :: *'a*::*onb-enum set*)›
by (*simp add*: *is-normal is-onb-def is-orthonormal*)

instance *onb-enum* ⊆ *chilbert-space*
proof
have ‹*complete* (*UNIV* :: *'a set*)›
using *finite-cspan-complete*[**where** *B*=‹*set canonical-basis*›]
by *simp*
then show *convergent X* **if** *Cauchy X* **for** *X* :: *nat* ⇒ *'a*
by (*simp add*: *complete-def convergent-def that*)
qed

9.12 Conjugate space

instantiation *conjugate-space* :: (*complex-inner*) *complex-inner* **begin**
lift-definition *cinner-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* ⇒ *complex* **is**
‹λ*x y*. *cinner y x*›**.**
instance
apply (*intro-classes*; *transfer*)
apply (*simp-all add*:)
apply (*simp add*: *cinner-add-right*)
using *cinner-ge-zero norm-eq-sqrt-cinner* **by** *auto*
end

instance *conjugate-space* :: (*chilbert-space*) *chilbert-space*..

9.13 Misc (ctd.)

lemma *separating-dense-span*:
assumes ‹⋀*F G* :: *'a*::*chilbert-space* ⇒ *'b*::{*complex-normed-vector*,*not-singleton*}.

bounded-clinear F ⟹ *bounded-clinear G* ⟹ (∀ *x*∈*S*. *F x* = *G x*) ⟹ *F* = *G*›
shows ‹*closure* (*cspan S*) = *UNIV*›
proof −
have ‹*ψ* = *0*› **if** ‹*ψ* ∈ *orthogonal-complement S*› **for** *ψ*
proof −
obtain *φ* :: *'b* **where** ‹*φ* ≠ *0*›
by *fastforce*
have ‹(λ*x*. *cinner ψ x* ∗*C φ*) = (λ-. *0*)›
apply (*rule assms*[*rule-format*])
using *orthogonal-complement-orthoI that*
by (*auto simp add*: *bounded-clinear-cinner-right bounded-clinear-scaleC-const*)
then have ‹*cinner ψ ψ* = *0*›
by (*meson* ‹*φ* ≠ *0*› *scaleC-eq-0-iff*)
then show ‹*ψ* = *0*›
by *auto*
qed

then have ‹*orthogonal-complement* (*orthogonal-complement S*) = *UNIV*›
by (*metis UNIV-eq-I cinner-zero-right orthogonal-complementI*)
then show ‹*closure* (*cspan S*) = *UNIV*›
by (*simp add*: *orthogonal-complement-orthogonal-complement-closure-cspan*)
qed

lemma *antilinear-cinner*:
shows ‹*antilinear* (λ*x*. *x* ·$_C$ *y*)›
by (*simp add*: *antilinearI cinner-add-left*)

lemma *cinner-extensionality-basis*:
fixes *g h* :: ‹'*a*::*complex-inner*›
assumes ‹*ccspan B* = *top*›
assumes ‹⋀*x*. *x* ∈ *B* ⟹ *x* ·$_C$ *g* = *x* ·$_C$ *h*›
shows ‹*g* = *h*›
proof (*rule cinner-extensionality*)
fix *y* :: '*a*
have ‹*y* ∈ *closure* (*cspan B*)›
using *assms*(*1*) *ccspan.rep-eq* **by** *fastforce*
then obtain *x* **where** ‹*x* ⟶ *y*› **and** *xB*: ‹*x i* ∈ *cspan B*› **for** *i*
using *closure-sequential* **by** *blast*
have *lin*: ‹*antilinear* (λ*a*. *a* ·$_C$ *g* − *a* ·$_C$ *h*)›
by (*intro antilinear-diff antilinear-cinner*)
from *lin* **have** ‹*x i* ·$_C$ *g* − *x i* ·$_C$ *h* = *0*› **for** *i*
apply (*rule antilinear-eq-0-on-span*[*of* - *B*])
using *xB assms* **by** *auto*
then have ‹(λ*i*. *x i* ·$_C$ *g* − *x i* ·$_C$ *h*) ⟶ *0*› **for** *i*
by *simp*
moreover have ‹(λ*i*. *x i* ·$_C$ *g* − *x i* ·$_C$ *h*) ⟶ *y* ·$_C$ *g* − *y* ·$_C$ *h*›
by (*simp add*: ‹*x* ⟶ *y*› *tendsto-cinner tendsto-diff*)
ultimately have ‹*y* ·$_C$ *g* − *y* ·$_C$ *h* = *0*›
using *LIMSEQ-unique* **by** *blast*
then show ‹*y* ·$_C$ *g* = *y* ·$_C$ *h*›
by *simp*
qed

end

10 *One-Dimensional-Spaces* – One dimensional complex vector spaces

theory *One-Dimensional-Spaces*
imports
Complex-Inner-Product
Complex-Bounded-Operators.Extra-Operator-Norm
begin

The class *one-dim* applies to one-dimensional vector spaces. Those are ad-

ditionally interpreted as *complex-algebra-1*s via the canonical isomorphism between a one-dimensional vector space and *complex*.

class *one-dim* = *onb-enum* + *one* + *times* + *inverse* +
  assumes *one-dim-canonical-basis*[*simp*]: *canonical-basis* = [*1*]
  assumes *one-dim-prod-scale1*: (*a* $*_C$ *1*) * (*b* $*_C$ *1*) = (*a* * *b*) $*_C$ *1*
  assumes *divide-inverse*: *x* / *y* = *x* * *inverse y*
  assumes *one-dim-inverse*: *inverse* (*a* $*_C$ *1*) = *inverse a* $*_C$ *1*

hide-fact (**open**) *divide-inverse*
  — *divide-inverse* from class *field*, instantiated below, subsumes this fact.

instance *complex* :: *one-dim*
  apply *intro-classes*
  unfolding *canonical-basis-complex-def is-ortho-set-def*
  by (*auto simp*: *divide-complex-def*)

lemma *one-cinner-one*[*simp*]: ‹(*1*::('*a*::*one-dim*)) $\cdot_C$ *1* = *1*›
proof−
  include *norm-syntax*
  have ‹(*canonical-basis*::'*a list*) = [*1*::('*a*::*one-dim*)]›
    by *simp*
  hence ‹‖*1*::'*a*::*one-dim*‖ = *1*›
    by (*metis is-normal list.set-intros*(*1*))
  hence ‹‖*1*::'*a*::*one-dim*‖^2 = *1*›
    by *simp*
  moreover have ‹‖(*1*::('*a*::*one-dim*))‖^2 = (*1*::('*a*::*one-dim*)) $\cdot_C$ *1*›
    by (*metis cnorm-eq-square*)
  ultimately show *?thesis* **by** *simp*
qed

lemma *one-cinner-a-scaleC-one*[*simp*]: ‹((*1*::'*a*::*one-dim*) $\cdot_C$ *a*) $*_C$ *1* = *a*›
proof−
  have ‹(*canonical-basis*::'*a list*) = [*1*]›
    by *simp*
  hence *r2*: ‹*a* ∈ *complex-vector.span* ({*1*::'*a*})›
    using *iso-tuple-UNIV-I empty-set is-generator-set list.simps*(*15*)
    by *metis*
  have (*1*::'*a*) ∉ {}
    by (*metis equals0D*)
  hence *r1*: ‹∃ *s*. *a* = *s* $*_C$ *1*›
    by (*metis Diff-insert-absorb r2 complex-vector.span-breakdown*
      complex-vector.span-empty eq-iff-diff-eq-0 singleton-iff)
  then obtain *s* **where** *s-def*: ‹*a* = *s* $*_C$ *1*›
    by *blast*
  have ‹(*1*::'*a*) $\cdot_C$ *a* = (*1*::'*a*) $\cdot_C$ (*s* $*_C$ *1*)›
    using ‹*a* = *s* $*_C$ *1*›
    by *simp*
  also have ‹... = *s* * ((*1*::'*a*) $\cdot_C$ *1*)›
    by *simp*

also have ‹... = s›
using *one-cinner-one* **by** *auto*
finally show *?thesis*
by (*simp add*: *s-def*)
qed

lemma *one-dim-apply-is-times-def*:
$\psi * \varphi = ((1 \cdot_C \psi) * (1 \cdot_C \varphi)) *_C 1$ **for** ψ :: ‹'a::*one-dim*›
by (*metis one-cinner-a-scaleC-one one-dim-prod-scale1*)

instance *one-dim* ⊆ *complex-algebra-1*
proof
fix x y z :: ‹'a::*one-dim*› **and** c :: *complex*
show $(x * y) * z = x * (y * z)$
by (*simp add*: *one-dim-apply-is-times-def*[**where** *?'a*='a])
show $(x + y) * z = x * z + y * z$
by (*metis* (*no-types*, *lifting*) *cinner-simps*(*2*) *complex-vector.vector-space-assms*(*2*)
complex-vector.vector-space-assms(*3*) *one-dim-apply-is-times-def*)
show $x * (y + z) = x * y + x * z$
by (*metis* (*mono-tags*, *lifting*) *cinner-simps*(*2*) *complex-vector.vector-space-assms*(*2*)
distrib-left one-dim-apply-is-times-def)
show $(c *_C x) * y = c *_C (x * y)$
by (*simp add*: *one-dim-apply-is-times-def*[**where** *?'a*='a])
show $x * (c *_C y) = c *_C (x * y)$
by (*simp add*: *one-dim-apply-is-times-def*[**where** *?'a*='a])
show $1 * x = x$
by (*metis mult.left-neutral one-cinner-a-scaleC-one one-cinner-one one-dim-apply-is-times-def*)
show $x * 1 = x$
by (*simp add*: *one-dim-apply-is-times-def* [**where** *?'a* = 'a])
show $(0::'a) \neq 1$
by (*metis cinner-eq-zero-iff one-cinner-one zero-neq-one*)
qed

instance *one-dim* ⊆ *complex-normed-algebra*
proof
fix x y :: ‹'a::*one-dim*›
show *norm* $(x * y) \leq$ *norm* $x *$ *norm* y
proof−
have *r1*: *cmod* $((1::'a) \cdot_C x) \leq$ *norm* $(1::'a) *$ *norm* x
by (*simp add*: *complex-inner-class.Cauchy-Schwarz-ineq2*)
have *r2*: *cmod* $((1::'a) \cdot_C y) \leq$ *norm* $(1::'a) *$ *norm* y
by (*simp add*: *complex-inner-class.Cauchy-Schwarz-ineq2*)

have *q1*: $(1::'a) \cdot_C 1 = 1$
by *simp*
hence (*norm* $(1::'a))\hat{}2 = 1$
by (*simp add*: *cnorm-eq-1 power2-eq-1-iff*)
hence *norm* $(1::'a) = 1$
by (*smt abs-norm-cancel power2-eq-1-iff*)

hence *cmod* (((*1*::*'a*) ·$_C$ *x*) * ((*1*::*'a*) ·$_C$ *y*)) * *norm* (*1*::*'a*) = *cmod* (((*1*::*'a*) ·$_C$ *x*) * ((*1*::*'a*) ·$_C$ *y*))
by *simp*
also have ... = *cmod* (((*1*::*'a*) ·$_C$ *x*)) * *cmod* (((*1*::*'a*) ·$_C$ *y*))
by (*simp add*: *norm-mult*)
also have ... ≤ *norm* (*1*::*'a*) * *norm x* * *norm* (*1*::*'a*) * *norm y*
by (*smt* ‹*norm 1 = 1*› *mult.commute mult-cancel-right1 norm-scaleC one-cinner-a-scaleC-one*)
also have ... = *norm x* * *norm y*
by (*simp add*: ‹*norm 1 = 1*›)
finally show *?thesis*
by (*simp add*: *one-dim-apply-is-times-def*[**where** *?'a*=*'a*])
qed
qed

instance *one-dim* ⊆ *complex-normed-algebra-1*
proof *intro-classes*
show *norm* (*1*::*'a*) = *1*
by (*metis complex-inner-1-left norm-eq-sqrt-cinner norm-one one-cinner-one*)
qed

This is the canonical isomorphism between any two one dimensional spaces. Specifically, if 1 denotes the element of the canonical basis (which is specified by type class *basis-enum*), then *one-dim-iso* is the unique isomorphism that maps 1 to 1.

definition *one-dim-iso* :: *'a*::*one-dim* ⇒ *'b*::*one-dim*
where *one-dim-iso a* = *of-complex* (*1* ·$_C$ *a*)

lemma *one-dim-iso-idem*[*simp*]: *one-dim-iso* (*one-dim-iso x*) = *one-dim-iso x*
by (*simp add*: *one-dim-iso-def*)

lemma *one-dim-iso-id*[*simp*]: *one-dim-iso* = *id*
unfolding *one-dim-iso-def*
by (*auto simp add*: *of-complex-def*)

lemma *one-dim-iso-adjoint*[*simp*]: ‹*cadjoint one-dim-iso* = *one-dim-iso*›
apply (*rule cadjoint-eqI*)
by (*simp add*: *one-dim-iso-def of-complex-def*)

lemma *one-dim-iso-is-of-complex*[*simp*]: *one-dim-iso* = *of-complex*
unfolding *one-dim-iso-def* **by** *auto*

lemma *of-complex-one-dim-iso*[*simp*]: *of-complex* (*one-dim-iso* ψ) = *one-dim-iso* ψ
by (*metis one-dim-iso-is-of-complex one-dim-iso-idem*)

lemma *one-dim-iso-of-complex*[*simp*]: *one-dim-iso* (*of-complex c*) = *of-complex c*
by (*metis one-dim-iso-is-of-complex one-dim-iso-idem*)

lemma *one-dim-iso-add*[*simp*]:

‹*one-dim-iso* (*a* + *b*) = *one-dim-iso a* + *one-dim-iso b*›
by (*simp add*: *cinner-simps*(*2*) *one-dim-iso-def*)

lemma *one-dim-iso-minus*[*simp*]:
‹*one-dim-iso* (*a* − *b*) = *one-dim-iso a* − *one-dim-iso b*›
by (*simp add*: *cinner-simps*(*3*) *one-dim-iso-def*)

lemma *one-dim-iso-scaleC*[*simp*]: *one-dim-iso* (*c* ∗$_C$ *ψ*) = *c* ∗$_C$ *one-dim-iso ψ*
by (*metis cinner-scaleC-right of-complex-mult one-dim-iso-def scaleC-conv-of-complex*)

lemma *clinear-one-dim-iso*[*simp*]: *clinear one-dim-iso*
by (*rule clinearI*, *auto*)

lemma *bounded-clinear-one-dim-iso*[*simp*]: *bounded-clinear one-dim-iso*
proof (*rule bounded-clinear-intro* [**where** *K* = *1*] , *auto*)
fix *x* :: ‹*'a*::*one-dim*›
show *norm* (*one-dim-iso x*) ≤ *norm x*
by (*metis* (*full-types*) *norm-of-complex of-complex-def one-cinner-a-scaleC-one one-dim-iso-def*
order-refl)
qed

lemma *one-dim-iso-of-one*[*simp*]: *one-dim-iso 1* = *1*
by (*simp add*: *one-dim-iso-def*)

lemma *onorm-one-dim-iso*[*simp*]: *onorm one-dim-iso* = *1*
proof (*rule onormI* [**where** *b* = *1* **and** *x* = *1*])
fix *x* :: ‹*'a*::*one-dim*›
have *norm* (*one-dim-iso x* ::*'b*) ≤ *norm x*
by (*metis eq-iff norm-of-complex of-complex-def one-cinner-a-scaleC-one one-dim-iso-def*)
thus *norm* (*one-dim-iso* (*x*::*'a*)::*'b*) ≤ *1* ∗ *norm x*
by *auto*
show (*1*::*'a*) ≠ *0*
by *simp*
show *norm* (*one-dim-iso* (*1*::*'a*)::*'b*) = *1* ∗ *norm* (*1*::*'a*)
by *auto*
qed

lemma *one-dim-iso-times*[*simp*]: *one-dim-iso* (*ψ* ∗ *φ*) = *one-dim-iso ψ* ∗ *one-dim-iso φ*
by (*metis mult.left-neutral mult-scaleC-left of-complex-def one-cinner-a-scaleC-one one-dim-iso-def one-dim-iso-scaleC*)

lemma *one-dim-iso-of-zero*[*simp*]: *one-dim-iso 0* = *0*
by (*simp add*: *complex-vector.linear-0*)

lemma *one-dim-iso-of-zero'*: *one-dim-iso x* = *0* ⟹ *x* = *0*
by (*metis of-complex-def of-complex-eq-0-iff one-cinner-a-scaleC-one one-dim-iso-def*)

lemma *one-dim-scaleC-1*[*simp*]: *one-dim-iso x* $*_C$ *1* = *x*
by (*simp add*: *one-dim-iso-def*)

lemma *one-dim-clinear-eqI*:
assumes (*x*::′*a*::*one-dim*) ≠ *0* **and** *clinear f* **and** *clinear g* **and** *f x* = *g x*
shows *f* = *g*
proof (*rule ext*)
fix *y* :: ′*a*
from ‹*f x* = *g x*›
have ‹*one-dim-iso x* $*_C$ *f 1* = *one-dim-iso x* $*_C$ *g 1*›
by (*metis assms*(*2*) *assms*(*3*) *complex-vector.linear-scale one-dim-scaleC-1*)
hence *f 1* = *g 1*
using *assms*(*1*) *one-dim-iso-of-zero*′ **by** *auto*
then show *f y* = *g y*
by (*metis assms*(*2*) *assms*(*3*) *complex-vector.linear-scale one-dim-scaleC-1*)
qed

lemma *one-dim-norm*: *norm x* = *cmod* (*one-dim-iso x*)
proof (*subst one-dim-scaleC-1* [*symmetric*])
show *norm* (*one-dim-iso x* $*_C$ (*1*::′*a*)) = *cmod* (*one-dim-iso x*)
by (*metis norm-of-complex of-complex-def*)
qed

lemma *norm-one-dim-iso*[*simp*]: ‹*norm* (*one-dim-iso x*) = *norm x*›
by (*metis norm-of-complex of-complex-one-dim-iso one-dim-norm*)

lemma *one-dim-onorm*:
fixes *f* :: ′*a*::*one-dim* ⇒ ′*b*::*complex-normed-vector*
assumes *clinear f*
shows *onorm f* = *norm* (*f 1*)
proof (*rule onormI*[**where** *x*=*1*])
fix *x* :: ′*a*
have *norm x* ∗ *norm* (*f 1*) ≤ *norm* (*f 1*) ∗ *norm x*
by *simp*
hence *norm* (*f* (*one-dim-iso x* $*_C$ *1*)) ≤ *norm* (*f 1*) ∗ *norm x*
by (*metis* (*mono-tags*, *lifting*) *assms complex-vector.linear-scale norm-scaleC one-dim-norm*)
thus *norm* (*f x*) ≤ *norm* (*f 1*) ∗ *norm x*
by (*subst one-dim-scaleC-1* [*symmetric*])
qed *auto*

lemma *one-dim-onorm*′:
fixes *f* :: ′*a*::*one-dim* ⇒ ′*b*::*one-dim*
assumes *clinear f*
shows *onorm f* = *cmod* (*one-dim-iso* (*f 1*))
using *assms one-dim-norm one-dim-onorm* **by** *fastforce*

instance *one-dim* ⊆ *zero-neq-one* **..**

lemma *one-dim-iso-inj*: *one-dim-iso x = one-dim-iso y ⟹ x = y*
by (*metis one-dim-iso-idem one-dim-scaleC-1*)

instance *one-dim ⊆ comm-ring*
proof *intro-classes*
fix *x y z :: 'a*
show *x * y = y * x*
by (*metis one-dim-apply-is-times-def ordered-field-class.sign-simps(5)*)
show *(x + y) * z = x * z + y * z*
by (*simp add*: *ring-class.ring-distribs(2)*)
qed

instance *one-dim ⊆ field*
proof *intro-classes*
fix *x y z :: ‹'a::one-dim›*
show *1 * x = x*
by *simp*

have *(inverse ((1::'a) ·C x) * ((1::'a) ·C x)) *C (1::'a) = 1* **if** *x ≠ 0*
by (*metis left-inverse of-complex-def one-cinner-a-scaleC-one one-dim-iso-of-zero*

one-dim-iso-is-of-complex one-dim-iso-of-one that)
hence *inverse (((1::'a) ·C x) *C 1) * ((1::'a) ·C x) *C 1 = (1::'a)* **if** *x ≠ 0*
by (*metis one-dim-inverse one-dim-prod-scale1 that*)
hence *inverse (((1::'a) ·C x) *C 1) * x = 1* **if** *x ≠ 0*
using *one-cinner-a-scaleC-one*[*of x*, *symmetric*] *that* **by** *auto*
thus *inverse x * x = 1* **if** *x ≠ 0*
by (*simp add*: *that*)
show *x / y = x * inverse y*
by (*simp add*: *one-dim-class.divide-inverse*)
show *inverse (0::'a) = 0*
by (*subst complex-vector.scale-zero-left*[*symmetric*], *subst one-dim-inverse*, *simp*)
qed

instance *one-dim ⊆ complex-normed-field*
proof *intro-classes*
fix *x y :: 'a*
show *norm (x * y) = norm x * norm y*
by (*metis norm-mult one-dim-iso-times one-dim-norm*)
qed

instance *one-dim ⊆ chilbert-space*..

lemma *ccspan-one-dim*[*simp*]: ‹*ccspan {x} = top*› **if** ‹*x ≠ 0*› **for** *x :: ‹- :: one-dim›*
proof −
have ‹*y ∈ cspan {x}*› **for** *y*
using *that* **by** (*metis complex-vector.span-base complex-vector.span-zero cspan-singleton-scaleC insertI1 one-dim-scaleC-1 scaleC-zero-left*)

then show *?thesis*
by (*auto intro*!: *order.antisym ccsubspace-leI*
simp: *top-ccsubspace.rep-eq ccspan.rep-eq*)
qed

lemma *one-dim-ccsubspace-all-or-nothing*: ‹$A = bot \vee A = top$› **for** A :: ‹-::*one-dim ccsubspace*›
proof (*rule Meson.disj-comm*, *rule disjCI*)
assume ‹$A \neq bot$›
then obtain ψ **where** ‹$\psi \in$ *space-as-set A*› **and** ‹$\psi \neq 0$›
by (*metis ccsubspace-eqI singleton-iff space-as-set-bot zero-space-as-set*)
then have ‹$A \geq$ *ccspan* $\{\psi\}$› (**is** ‹- ≥ ...›)
by (*metis bot.extremum ccspan-leqI insert-absorb insert-mono*)
also have ‹... = *ccspan* {*one-dim-iso* $\psi *_C$ *1*}›
by *auto*
also have ‹... = *ccspan* {*1*}›
apply (*rule ccspan-singleton-scaleC*)
using ‹$\psi \neq 0$› *one-dim-iso-of-zero'* **by** *auto*
also have ‹... = *top*›
by *auto*
finally show ‹A = *top*›
by (*simp add*: *top.extremum-uniqueI*)
qed

lemma *scaleC-1-right*[*simp*]: ‹*scaleC x* (*1*::'*a*::*one-dim*) = *of-complex x*›
unfolding *of-complex-def* **by** *simp*

lemma *canonical-basis-length-one-dim*[*simp*]: ‹*canonical-basis-length TYPE*('*a*::*one-dim*) = *1*›
by (*simp add*: *canonical-basis-length*)

end

11 *Complex-Euclidean-Space0* – Finite-Dimensional Inner Product Spaces

theory *Complex-Euclidean-Space0*
imports
HOL−Analysis.L2-Norm
Complex-Inner-Product
HOL−Analysis.Product-Vector
HOL−Library.Rewrite
begin

11.1 Type class of Euclidean spaces

class *ceuclidean-space* = *complex-inner* +
fixes *CBasis* :: '*a set*

assumes *nonempty-CBasis* [*simp*]: *CBasis* ≠ {}
assumes *finite-CBasis* [*simp*]: *finite CBasis*
assumes *cinner-CBasis*:
⟦*u* ∈ *CBasis*; *v* ∈ *CBasis*⟧ ⟹ *cinner u v* = (*if u* = *v then 1 else 0*)
assumes *ceuclidean-all-zero-iff*:
(∀ *u*∈*CBasis*. *cinner x u* = *0*) ⟷ (*x* = *0*)

syntax *-type-cdimension* :: *type* ⇒ *nat* (‹(*1CDIM*/(*1*'(-')))›)
syntax-consts *-type-cdimension* ⇌ *card*
translations *CDIM*('*a*) ⇀ *CONST card* (*CONST CBasis* :: '*a set*)
typed-print-translation ‹
[(***const-syntax*** ‹*card*›,
fn ctxt => *fn* - => *fn* [*Const* (***const-syntax*** ‹*CBasis*›, *Type* (***type-name*** ‹*set*›,
[*T*]))] =>
Syntax.const ***syntax-const*** ‹*-type-cdimension*› $ *Syntax-Phases.term-of-typ*
ctxt T)]
›

lemma (**in** *ceuclidean-space*) *norm-CBasis*[*simp*]: *u* ∈ *CBasis* ⟹ *norm u* = *1*
unfolding *norm-eq-sqrt-cinner* **by** (*simp add*: *cinner-CBasis*)

lemma (**in** *ceuclidean-space*) *cinner-same-CBasis*[*simp*]: *u* ∈ *CBasis* ⟹ *cinner u*
u = *1*
by (*simp add*: *cinner-CBasis*)

lemma (**in** *ceuclidean-space*) *cinner-not-same-CBasis*: *u* ∈ *CBasis* ⟹ *v* ∈ *CBasis*
⟹ *u* ≠ *v* ⟹ *cinner u v* = *0*
by (*simp add*: *cinner-CBasis*)

lemma (**in** *ceuclidean-space*) *sgn-CBasis*: *u* ∈ *CBasis* ⟹ *sgn u* = *u*
unfolding *sgn-div-norm* **by** (*simp add*: *scaleR-one*)

lemma (**in** *ceuclidean-space*) *CBasis-zero* [*simp*]: *0* ∉ *CBasis*
proof
assume *0* ∈ *CBasis* **thus** *False*
using *cinner-CBasis* [*of 0 0*] **by** *simp*
qed

lemma (**in** *ceuclidean-space*) *nonzero-CBasis*: *u* ∈ *CBasis* ⟹ *u* ≠ *0*
by *clarsimp*

lemma (**in** *ceuclidean-space*) *SOME-CBasis*: (*SOME i*. *i* ∈ *CBasis*) ∈ *CBasis*
by (*metis ex-in-conv nonempty-CBasis someI-ex*)

lemma *norm-some-CBasis* [*simp*]: *norm* (*SOME i*. *i* ∈ *CBasis*) = *1*
by (*simp add*: *SOME-CBasis*)

lemma (**in** *ceuclidean-space*) *cinner-sum-left-CBasis*[*simp*]:
b ∈ *CBasis* ⟹ *cinner* (∑ *i*∈*CBasis*. *f i* ∗$_C$ *i*) *b* = *cnj* (*f b*)

by (*simp add*: *cinner-sum-left cinner-CBasis if-distrib comm-monoid-add-class.sum.If-cases*)

lemma (**in** *ceuclidean-space*) *ceuclidean-eqI*:
assumes *b*: $\bigwedge b.\ b \in CBasis \Longrightarrow cinner\ x\ b = cinner\ y\ b$ **shows** $x = y$
proof −
from *b* **have** $\forall\, b{\in}CBasis.\ cinner\ (x - y)\ b = 0$
by (*simp add*: *cinner-diff-left*)
then show $x = y$
by (*simp add*: *ceuclidean-all-zero-iff*)
qed

lemma (**in** *ceuclidean-space*) *ceuclidean-eq-iff*:
$x = y \longleftrightarrow (\forall\, b{\in}CBasis.\ cinner\ x\ b = cinner\ y\ b)$
by (*auto intro*: *ceuclidean-eqI*)

lemma (**in** *ceuclidean-space*) *ceuclidean-representation-sum*:
$(\sum i{\in}CBasis.\ f\ i *_C i) = b \longleftrightarrow (\forall\, i{\in}CBasis.\ f\ i = cnj\ (cinner\ b\ i))$
apply (*subst ceuclidean-eq-iff*)
apply *simp* **by** (*metis complex-cnj-cnj cinner-commute*)

lemma (**in** *ceuclidean-space*) *ceuclidean-representation-sum′*:
$b = (\sum i{\in}CBasis.\ f\ i *_C i) \longleftrightarrow (\forall\, i{\in}CBasis.\ f\ i = cinner\ i\ b)$
apply (*auto simp add*: *ceuclidean-representation-sum*[*symmetric*])
apply (*metis ceuclidean-representation-sum cinner-commute*)
by (*metis local.ceuclidean-representation-sum local.cinner-commute*)

lemma (**in** *ceuclidean-space*) *ceuclidean-representation*: $(\sum b{\in}CBasis.\ cinner\ b\ x *_C b) = x$
unfolding *ceuclidean-representation-sum*
using *local.cinner-commute* **by** *blast*

lemma (**in** *ceuclidean-space*) *ceuclidean-cinner*: $cinner\ x\ y = (\sum b{\in}CBasis.\ cinner\ x\ b * cnj\ (cinner\ y\ b))$
apply (*subst* (*1 2*) *ceuclidean-representation* [*symmetric*])
apply (*simp add*: *cinner-sum-right cinner-CBasis ac-simps*)
by (*metis local.cinner-commute mult.commute*)

lemma (**in** *ceuclidean-space*) *choice-CBasis-iff*:
fixes $P :: 'a \Rightarrow complex \Rightarrow bool$
shows $(\forall\, i{\in}CBasis.\ \exists\, x.\ P\ i\ x) \longleftrightarrow (\exists\, x.\ \forall\, i{\in}CBasis.\ P\ i\ (cinner\ x\ i))$
unfolding *bchoice-iff*
proof *safe*
fix *f* **assume** $\forall\, i{\in}CBasis.\ P\ i\ (f\ i)$
then show $\exists\, x.\ \forall\, i{\in}CBasis.\ P\ i\ (cinner\ x\ i)$
by (*auto intro*!: *exI*[*of* - $\sum i{\in}CBasis.\ cnj\ (f\ i) *_C i$])
qed *auto*

lemma (**in** *ceuclidean-space*) *bchoice-CBasis-iff*:
fixes $P :: 'a \Rightarrow complex \Rightarrow bool$
shows $(\forall i{\in}CBasis.\ \exists x{\in}A.\ P\ i\ x) \longleftrightarrow (\exists x.\ \forall i{\in}CBasis.\ cinner\ x\ i \in A \wedge P\ i\ (cinner\ x\ i))$
by (*simp add*: *choice-CBasis-iff Bex-def*)

lemma (**in** *ceuclidean-space*) *ceuclidean-representation-sum-fun*:
$(\lambda x.\ \sum b{\in}CBasis.\ cinner\ b\ (f\ x)\ *_C\ b) = f$
apply (*rule ext*)
apply (*simp add*: *ceuclidean-representation-sum*)
by (*meson local.cinner-commute*)

lemma *euclidean-isCont*:
assumes $\bigwedge b.\ b \in CBasis \Longrightarrow isCont\ (\lambda x.\ (cinner\ b\ (f\ x))\ *_C\ b)\ x$
shows *isCont f x*
apply (*subst ceuclidean-representation-sum-fun* [*symmetric*])
apply (*rule isCont-sum*)
by (*blast intro*: *assms*)

lemma *CDIM-positive* [*simp*]: $0 < CDIM('a{::}ceuclidean\text{-}space)$
by (*simp add*: *card-gt-0-iff*)

lemma *CDIM-ge-Suc0* [*simp*]: $Suc\ 0 \leq card\ CBasis$
by (*meson CDIM-positive Suc-leI*)

lemma *sum-cinner-CBasis-scaleC* [*simp*]:
fixes $f :: 'a{::}ceuclidean\text{-}space \Rightarrow 'b{::}complex\text{-}vector$
assumes $b \in CBasis$ **shows** $(\sum i{\in}CBasis.\ (cinner\ i\ b)\ *_C\ f\ i) = f\ b$
by (*simp add*: *comm-monoid-add-class.sum.remove* [*OF finite-CBasis assms*] *assms cinner-not-same-CBasis comm-monoid-add-class.sum.neutral*)

lemma *sum-cinner-CBasis-eq* [*simp*]:
assumes $b \in CBasis$ **shows** $(\sum i{\in}CBasis.\ (cinner\ i\ b) * f\ i) = f\ b$
by (*simp add*: *comm-monoid-add-class.sum.remove* [*OF finite-CBasis assms*] *assms cinner-not-same-CBasis comm-monoid-add-class.sum.neutral*)

lemma *sum-if-cinner* [*simp*]:
assumes $i \in CBasis\ j \in CBasis$
shows $cinner\ (\sum k{\in}CBasis.\ if\ k = i\ then\ f\ i\ *_C\ i\ else\ g\ k\ *_C\ k)\ j = (if\ j{=}i\ then\ cnj\ (f\ j)\ else\ cnj\ (g\ j))$
proof (*cases i=j*)
case *True*
with *assms* **show** *?thesis*
by (*auto simp*: *cinner-sum-left if-distrib* [*of* $\lambda x.\ cinner\ x\ j$] *cinner-CBasis cong*: *if-cong*)
next
case *False*

have $(\sum k{\in}CBasis.\ cinner\ (if\ k = i\ then\ f\ i *_C i\ else\ g\ k *_C k)\ j) =$
$(\sum k{\in}CBasis.\ if\ k = j\ then\ cnj\ (g\ k)\ else\ 0)$
apply (*rule sum.cong*)
using *False assms* **by** (*auto simp*: *cinner-CBasis*)
also have ... = *cnj* (*g j*)
using *assms* **by** *auto*
finally show *?thesis*
using *False* **by** (*auto simp*: *cinner-sum-left*)
qed

lemma *norm-le-componentwise*:
$(\bigwedge b.\ b \in CBasis \Longrightarrow cmod(cinner\ x\ b) \leq cmod(cinner\ y\ b)) \Longrightarrow norm\ x \leq norm\ y$
apply (*auto simp*: *cnorm-le ceuclidean-cinner* [*of x x*] *ceuclidean-cinner* [*of y y*] *power2-eq-square intro*!: *sum-mono*)
by (*smt* (*verit*, *best*) *mult.commute sum.cong*)

lemma *CBasis-le-norm*: $b \in CBasis \Longrightarrow cmod\ (cinner\ x\ b) \leq norm\ x$
by (*rule order-trans* [*OF Cauchy-Schwarz-ineq2*]) *simp*

lemma *norm-bound-CBasis-le*: $b \in CBasis \Longrightarrow norm\ x \leq e \Longrightarrow cmod\ (inner\ x\ b) \leq e$
by (*metis inner-commute mult.left-neutral norm-CBasis norm-of-real order-trans real-inner-class.Cauchy-Schwarz-ineq2*)

lemma *norm-bound-CBasis-lt*: $b \in CBasis \Longrightarrow norm\ x < e \Longrightarrow cmod\ (inner\ x\ b) < e$
by (*metis inner-commute le-less-trans mult.left-neutral norm-CBasis norm-of-real real-inner-class.Cauchy-Schwarz-ineq2*)

lemma *cnorm-le-l1*: $norm\ x \leq (\sum b{\in}CBasis.\ cmod\ (cinner\ x\ b))$
apply (*subst ceuclidean-representation*[*of x*, *symmetric*])
apply (*rule order-trans*[*OF norm-sum*])
apply (*auto intro*!: *sum-mono*)
by (*metis cinner-commute complex-inner-1-left complex-inner-class.Cauchy-Schwarz-ineq2 mult.commute mult.left-neutral norm-one*)

11.2 Class instances

11.2.1 Type *complex*

instantiation *complex* :: *ceuclidean-space*
begin

definition
[*simp*]: $CBasis = \{1{::}complex\}$

instance
by *standard auto*

end

lemma *CDIM-complex*[*simp*]: *CDIM*(*complex*) = *1*
by *simp*

11.2.2 Type $'a \times 'b$

lemma *cinner-Pair* [*simp*]: *cinner* (*a*, *b*) (*c*, *d*) = *cinner a c* + *cinner b d*
unfolding *cinner-prod-def* **by** *simp*

lemma *cinner-Pair-0*: *cinner x* (*0*, *b*) = *cinner* (*snd x*) *b cinner x* (*a*, *0*) = *cinner* (*fst x*) *a*
by (*cases x*, *simp*)+

instantiation *prod* :: (*ceuclidean-space*, *ceuclidean-space*) *ceuclidean-space*
begin

definition
CBasis = (λ*u*. (*u*, *0*)) ' *CBasis* ∪ (λ*v*. (*0*, *v*)) ' *CBasis*

lemma *sum-CBasis-prod-eq*:
fixes *f*::('*a*∗'*b*)⇒('*a*∗'*b*)
shows *sum f CBasis* = *sum* (λ*i*. *f* (*i*, *0*)) *CBasis* + *sum* (λ*i*. *f* (*0*, *i*)) *CBasis*
proof −
have *inj-on* (λ*u*. (*u*::'*a*, *0*::'*b*)) *CBasis inj-on* (λ*u*. (*0*::'*a*, *u*::'*b*)) *CBasis*
by (*auto intro*!: *inj-onI Pair-inject*)
thus *?thesis*
unfolding *CBasis-prod-def*
by (*subst sum.union-disjoint*) (*auto simp*: *CBasis-prod-def sum.reindex*)
qed

instance proof
show (*CBasis* :: ('*a* × '*b*) *set*) ≠ {}
unfolding *CBasis-prod-def* **by** *simp*
next
show *finite* (*CBasis* :: ('*a* × '*b*) *set*)
unfolding *CBasis-prod-def* **by** *simp*
next
fix *u v* :: '*a* × '*b*
assume *u* ∈ *CBasis* **and** *v* ∈ *CBasis*
thus *cinner u v* = (*if u* = *v then 1 else 0*)
unfolding *CBasis-prod-def cinner-prod-def*
by (*auto simp add*: *cinner-CBasis split*: *if-split-asm*)
next
fix *x* :: '*a* × '*b*
show (∀ *u*∈*CBasis*. *cinner x u* = *0*) ⟷ *x* = *0*
unfolding *CBasis-prod-def ball-Un ball-simps*
by (*simp add*: *cinner-prod-def prod-eq-iff ceuclidean-all-zero-iff*)
qed

lemma *CDIM-prod*[*simp*]: *CDIM*('*a* × '*b*) = *CDIM*('*a*) + *CDIM*('*b*)
unfolding *CBasis-prod-def*
by (*subst card-Un-disjoint*) (*auto intro*!: *card-image arg-cong2*[**where** *f*=(+)] *inj-onI*)

end

11.3 Locale instances

lemma *finite-dimensional-vector-space-euclidean*:
finite-dimensional-vector-space (∗$_C$) *CBasis*
proof *unfold-locales*
show *finite* (*CBasis*::'*a set*) **by** (*metis finite-CBasis*)
show *complex-vector.independent* (*CBasis*::'*a set*)
unfolding *complex-vector.dependent-def cdependent-raw-def*[*symmetric*]
apply (*subst complex-vector.span-finite*)
apply *simp*
apply *clarify*
apply (*drule-tac f*=*cinner a* **in** *arg-cong*)
by (*simp add*: *cinner-CBasis cinner-sum-right eq-commute*)
show *module.span* (∗$_C$) *CBasis* = *UNIV*
unfolding *complex-vector.span-finite* [*OF finite-CBasis*] *cspan-raw-def*[*symmetric*]
by (*auto intro*!: *ceuclidean-representation*[*symmetric*])
qed

interpretation *ceucl*: *finite-dimensional-vector-space scaleC* :: *complex* => '*a* => '*a*::*ceuclidean-space CBasis*
rewrites *module.dependent* (∗$_C$) = *cdependent*
and *module.representation* (∗$_C$) = *crepresentation*
and *module.subspace* (∗$_C$) = *csubspace*
and *module.span* (∗$_C$) = *cspan*
and *vector-space.extend-basis* (∗$_C$) = *cextend-basis*
and *vector-space.dim* (∗$_C$) = *cdim*
and *Vector-Spaces.linear* (∗$_C$) (∗$_C$) = *clinear*
and *Vector-Spaces.linear* (∗) (∗$_C$) = *clinear*
and *finite-dimensional-vector-space.dimension CBasis* = *CDIM*('*a*)

by (*auto simp add*: *cdependent-raw-def crepresentation-raw-def*
csubspace-raw-def cspan-raw-def cextend-basis-raw-def cdim-raw-def clinear-def
complex-scaleC-def[*abs-def*]
finite-dimensional-vector-space.dimension-def
intro!: *finite-dimensional-vector-space.dimension-def*
finite-dimensional-vector-space-euclidean)

interpretation *ceucl*: *finite-dimensional-vector-space-pair-1*
scaleC::*complex*⇒'*a*::*ceuclidean-space*⇒'*a CBasis*
scaleC::*complex*⇒'*b*::*complex-vector* ⇒ '*b*
by *unfold-locales*

interpretation *ceucl?*: *finite-dimensional-vector-space-prod scaleC scaleC CBasis CBasis*
rewrites *Basis-pair = CBasis*
and *module-prod.scale* $(*_C)$ $(*_C) = (scaleC::-\Rightarrow-\Rightarrow('a \times 'b))$
proof −
show *finite-dimensional-vector-space-prod* $(*_C)$ $(*_C)$ *CBasis CBasis*
by *unfold-locales*
interpret *finite-dimensional-vector-space-prod* $(*_C)$ $(*_C)$ *CBasis*::$'a$ *set CBasis*::$'b$ *set*
by *fact*
show *Basis-pair = CBasis*
unfolding *Basis-pair-def CBasis-prod-def* **by** *auto*
show *module-prod.scale* $(*_C)$ $(*_C) =$ *scaleC*
by (*fact module-prod-scale-eq-scaleC*)
qed

end

12 *Complex-Bounded-Linear-Function0* − Bounded Linear Function

theory *Complex-Bounded-Linear-Function0*
imports
HOL−Analysis.Bounded-Linear-Function
Complex-Inner-Product
Complex-Euclidean-Space0
begin

unbundle *cinner-syntax*

lemma *conorm-componentwise*:
assumes *bounded-clinear f*
shows *onorm f* $\leq (\sum i{\in}CBasis.\ norm\ (f\ i))$
proof −
{
fix *i*::$'a$
assume $i \in CBasis$
hence *onorm* $(\lambda x.\ (i \cdot_C x) *_C f\ i) \leq$ *onorm* $(\lambda x.\ (i \cdot_C x)) * norm\ (f\ i)$
by (*auto intro*!: *onorm-scaleC-left-lemma bounded-clinear-cinner-right*)
also have $\ldots \leq$ *norm i* $*$ *norm* $(f\ i)$
apply (*rule mult-right-mono*)
apply (*simp add*: *complex-inner-class.Cauchy-Schwarz-ineq2 onorm-bound*)
by *simp*
finally have *onorm* $(\lambda x.\ (i \cdot_C x) *_C f\ i) \leq norm\ (f\ i)$ **using** ‹$i \in CBasis$›
by *simp*
} **hence** *onorm* $(\lambda x.\ \sum i{\in}CBasis.\ (i \cdot_C x) *_C f\ i) \leq (\sum i{\in}CBasis.\ norm\ (f\ i))$
by (*auto intro*!: *order-trans*[*OF onorm-sum-le*] *bounded-clinear-scaleC-const*

sum-mono bounded-clinear-cinner-right bounded-clinear.bounded-linear)
also have $(\lambda x.\ \sum i{\in}CBasis.\ (i \cdot_C x) *_C f\ i) = (\lambda x.\ f\ (\sum i{\in}CBasis.\ (i \cdot_C x) *_C i))$
by (*simp add*: *clinear.scaleC linear-sum bounded-clinear.clinear clinear.linear assms*)
also have $\ldots = f$
by (*simp add*: *ceuclidean-representation*)
finally show *?thesis* **.**
qed

lemmas *conorm-componentwise-le* = *order-trans*[*OF conorm-componentwise*]

12.1 Intro rules for *bounded-linear*

lemma *onorm-cinner-left*:
assumes *bounded-linear r*
shows $onorm\ (\lambda x.\ r\ x \cdot_C f) \leq onorm\ r * norm\ f$
proof (*rule onorm-bound*)
fix *x*
have $norm\ (r\ x \cdot_C f) \leq norm\ (r\ x) * norm\ f$
by (*simp add*: *Cauchy-Schwarz-ineq2*)
also have $\ldots \leq onorm\ r * norm\ x * norm\ f$
by (*simp add*: *assms mult.commute mult-left-mono onorm*)
finally show $norm\ (r\ x \cdot_C f) \leq onorm\ r * norm\ f * norm\ x$
by (*simp add*: *ac-simps*)
qed (*intro mult-nonneg-nonneg norm-ge-zero onorm-pos-le assms*)

lemma *onorm-cinner-right*:
assumes *bounded-linear r*
shows $onorm\ (\lambda x.\ f \cdot_C r\ x) \leq norm\ f * onorm\ r$
proof (*rule onorm-bound*)
fix *x*
have $norm\ (f \cdot_C r\ x) \leq norm\ f * norm\ (r\ x)$
by (*simp add*: *Cauchy-Schwarz-ineq2*)
also have $\ldots \leq onorm\ r * norm\ x * norm\ f$
by (*simp add*: *assms mult.commute mult-left-mono onorm*)
finally show $norm\ (f \cdot_C r\ x) \leq norm\ f * onorm\ r * norm\ x$
by (*simp add*: *ac-simps*)
qed (*intro mult-nonneg-nonneg norm-ge-zero onorm-pos-le assms*)

lemmas [*bounded-linear-intros*] =
bounded-clinear-zero
bounded-clinear-add
bounded-clinear-const-mult
bounded-clinear-mult-const
bounded-clinear-scaleC-const
bounded-clinear-const-scaleC
bounded-clinear-const-scaleR
bounded-clinear-ident

bounded-clinear-sum

bounded-clinear-sub

bounded-antilinear-cinner-left-comp
bounded-clinear-cinner-right-comp

12.2 declaration of derivative/continuous/tendsto introduction rules for bounded linear functions

attribute-setup *bounded-clinear* =
‹*let val bounded-linear* = *Attrib.attribute* ***context*** (*the-single* @{*attributes* [*bounded-linear*]})
in
Scan.succeed (*Thm.declaration-attribute* (*fn thm* =>
Thm.attribute-declaration bounded-linear (*thm RS* @{*thm bounded-clinear.bounded-linear*})
o
fold (*fn* (*r*, *s*) => *Named-Theorems.add-thm s* (*thm RS r*))
[
(∗ *Not present in Bounded-Linear-Function* ∗)
(@{*thm bounded-clinear-compose*}, ***named-theorems*** ‹*bounded-linear-intros*›),
(@{*thm bounded-clinear-o-bounded-antilinear*[*unfolded o-def*]}, ***named-theorems*** ‹*bounded-linear-intros*›)
]))
end›

attribute-setup *bounded-antilinear* =
‹*let val bounded-linear* = *Attrib.attribute* ***context*** (*the-single* @{*attributes* [*bounded-linear*]})
in
Scan.succeed (*Thm.declaration-attribute* (*fn thm* =>
Thm.attribute-declaration bounded-linear (*thm RS* @{*thm bounded-antilinear.bounded-linear*})
o
fold (*fn* (*r*, *s*) => *Named-Theorems.add-thm s* (*thm RS r*))
[
(∗ *Not present in Bounded-Linear-Function* ∗)
(@{*thm bounded-antilinear-o-bounded-clinear*[*unfolded o-def*]}, ***named-theorems*** ‹*bounded-linear-intros*›),
(@{*thm bounded-antilinear-o-bounded-antilinear*[*unfolded o-def*]}, ***named-theorems*** ‹*bounded-linear-intros*›)
]))
end›

attribute-setup *bounded-cbilinear* =
‹*let val bounded-bilinear* = *Attrib.attribute* ***context*** (*the-single* @{*attributes* [*bounded-bilinear*]})
in
Scan.succeed (*Thm.declaration-attribute* (*fn thm* =>
Thm.attribute-declaration bounded-bilinear (*thm RS* @{*thm bounded-cbilinear.bounded-bilinear*})
o

fold (*fn* (*r*, *s*) => *Named-Theorems.add-thm s* (*thm RS r*))
[
(@{*thm bounded-clinear-compose*[*OF bounded-cbilinear.bounded-clinear-left*]},
named-theorems ‹*bounded-linear-intros*›),
(@{*thm bounded-clinear-compose*[*OF bounded-cbilinear.bounded-clinear-right*]},
named-theorems ‹*bounded-linear-intros*›),
(@{*thm bounded-clinear-o-bounded-antilinear*[*unfolded o-def*, *OF bounded-cbilinear.bounded-clinear-left*]},
named-theorems ‹*bounded-linear-intros*›),
(@{*thm bounded-clinear-o-bounded-antilinear*[*unfolded o-def*, *OF bounded-cbilinear.bounded-clinear-right*]}
named-theorems ‹*bounded-linear-intros*›)
]))
end›

attribute-setup *bounded-sesquilinear* =
‹*let val bounded-bilinear* = *Attrib.attribute* ***context*** (*the-single* @{*attributes* [*bounded-bilinear*]})
in
Scan.succeed (*Thm.declaration-attribute* (*fn thm* =>
Thm.attribute-declaration bounded-bilinear (*thm RS* @{*thm bounded-sesquilinear.bounded-bilinear*})
o
fold (*fn* (*r*, *s*) => *Named-Theorems.add-thm s* (*thm RS r*))
[
(@{*thm bounded-antilinear-o-bounded-clinear*[*unfolded o-def*, *OF bounded-sesquilinear.bounded-antilinear-*
named-theorems ‹*bounded-linear-intros*›),
(@{*thm bounded-clinear-compose*[*OF bounded-sesquilinear.bounded-clinear-right*]},
named-theorems ‹*bounded-linear-intros*›),
(@{*thm bounded-antilinear-o-bounded-antilinear*[*unfolded o-def*, *OF bounded-sesquilinear.bounded-antiline*
named-theorems ‹*bounded-linear-intros*›),
(@{*thm bounded-clinear-o-bounded-antilinear*[*unfolded o-def*, *OF bounded-sesquilinear.bounded-clinear-rig*
named-theorems ‹*bounded-linear-intros*›)
]))
end›

12.3 Type of complex bounded linear functions

typedef (**overloaded**) (*'a*, *'b*) *cblinfun* (‹(- $\Rightarrow_{CL}$ /-)› [*22*, *21*] *21*) =
{*f*::*'a*::*complex-normed-vector*⇒*'b*::*complex-normed-vector*. *bounded-clinear f*}
morphisms *cblinfun-apply CBlinfun*
by (*blast intro*: *bounded-linear-intros*)

declare [[*coercion*
cblinfun-apply :: (*'a*::*complex-normed-vector* $\Rightarrow_{CL}$ *'b*::*complex-normed-vector*)
⇒ *'a* ⇒ *'b*]]

lemma *bounded-clinear-cblinfun-apply*[*bounded-linear-intros*]:
bounded-clinear g ⟹ *bounded-clinear* (λ*x*. *cblinfun-apply f* (*g x*))
by (*metis cblinfun-apply mem-Collect-eq bounded-clinear-compose*)

setup-lifting *type-definition-cblinfun*

lemma *cblinfun-eqI*: $(\bigwedge i.\ \textit{cblinfun-apply}\ x\ i = \textit{cblinfun-apply}\ y\ i) \Longrightarrow x = y$
by *transfer auto*

lemma *bounded-clinear-CBlinfun-apply*: *bounded-clinear f* $\Longrightarrow$ *cblinfun-apply* (*CBlinfun f*) = *f*
by (*auto simp*: *CBlinfun-inverse*)

12.4 Type class instantiations

instantiation *cblinfun* :: (*complex-normed-vector*, *complex-normed-vector*) *complex-normed-vector*
begin

lift-definition *norm-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow$ *real* **is** *onorm* **.**

lift-definition *minus-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$
is $\lambda f\ g\ x.\ f\ x - g\ x$
by (*rule bounded-clinear-sub*)

definition *dist-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b \Rightarrow$ *real*
where *dist-cblinfun a b* = *norm* $(a - b)$

definition [*code del*]:
(*uniformity* :: $(('a \Rightarrow_{CL} 'b) \times ('a \Rightarrow_{CL} 'b))$ *filter*) = (*INF* $e{\in}\{0 <..\}$. *principal* $\{(x, y).\ \textit{dist}\ x\ y < e\}$)

definition *open-cblinfun* :: $('a \Rightarrow_{CL} 'b)$ *set* $\Rightarrow$ *bool*
where [*code del*]: *open-cblinfun S* = $(\forall x{\in}S.\ \forall_F\ (x', y)$ *in uniformity*. $x' = x \longrightarrow y \in S)$

lift-definition *uminus-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$ **is** $\lambda f\ x.\ -\ f\ x$
by (*rule bounded-clinear-minus*)

lift-definition *zero-cblinfun* :: $'a \Rightarrow_{CL} 'b$ **is** $\lambda x.\ 0$
by (*rule bounded-clinear-zero*)

lift-definition *plus-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$
is $\lambda f\ g\ x.\ f\ x + g\ x$
by (*metis bounded-clinear-add*)

lift-definition *scaleC-cblinfun*::*complex* $\Rightarrow$ $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$ **is** $\lambda r\ f\ x.\ r\ *_C\ f\ x$
by (*metis bounded-clinear-compose bounded-clinear-scaleC-right*)
lift-definition *scaleR-cblinfun*::*real* $\Rightarrow$ $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$ **is** $\lambda r\ f\ x.\ r\ *_R\ f\ x$
by (*rule bounded-clinear-const-scaleR*)

definition *sgn-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$
where *sgn-cblinfun x* = *scaleC* (*inverse* (*norm x*)) *x*

instance
proof
fix $a\ b\ c :: 'a \Rightarrow_{CL} 'b$ **and** $r\ q :: real$ **and** $s\ t :: complex$

show ‹$a + b + c = a + (b + c)$›
apply *transfer* **by** *auto*
show ‹$0 + a = a$›
apply *transfer* **by** *auto*
show ‹$a + b = b + a$›
apply *transfer* **by** *auto*
show ‹$- a + a = 0$›
apply *transfer* **by** *auto*
show ‹$a - b = a + - b$›
apply *transfer* **by** *auto*
show *scaleR-scaleC*: ‹$((*_R)\ r::('a \Rightarrow_{CL} 'b) \Rightarrow -) = (*_C)\ (complex\text{-}of\text{-}real\ r)$› **for** *r*
apply (*rule ext*, *transfer fixing*: *r*) **by** (*simp add*: *scaleR-scaleC*)
show ‹$s *_C (b + c) = s *_C b + s *_C c$›
apply *transfer* **by** (*simp add*: *scaleC-add-right*)
show ‹$(s + t) *_C a = s *_C a + t *_C a$›
apply *transfer* **by** (*simp add*: *scaleC-left.add*)
show ‹$s *_C t *_C a = (s * t) *_C a$›
apply *transfer* **by** *auto*
show ‹$1 *_C a = a$›
apply *transfer* **by** *auto*
show ‹$dist\ a\ b = norm\ (a - b)$›
unfolding *dist-cblinfun-def* **by** *simp*
show ‹$sgn\ a = (inverse\ (norm\ a)) *_R a$›
unfolding *sgn-cblinfun-def* **unfolding** *scaleR-scaleC* **by** *auto*
show ‹$uniformity = (INF\ e{\in}\{0<..\}.\ principal\ \{(x,\ y).\ dist\ (x::('a \Rightarrow_{CL} 'b))\ y < e\})$›
by (*simp add*: *uniformity-cblinfun-def*)
show ‹$open\ U = (\forall x{\in}U.\ \forall_F\ (x',\ y)\ in\ uniformity.\ (x'::('a \Rightarrow_{CL} 'b)) = x \longrightarrow y \in U)$› **for** *U*
by (*simp add*: *open-cblinfun-def*)
show ‹$(norm\ a = 0) = (a = 0)$›
apply *transfer* **using** *bounded-clinear.bounded-linear onorm-eq-0* **by** *blast*
show ‹$norm\ (a + b) \leq norm\ a + norm\ b$›
apply *transfer* **by** (*simp add*: *bounded-clinear.bounded-linear onorm-triangle*)
show ‹$norm\ (s *_C a) = cmod\ s * norm\ a$›
apply *transfer* **using** *onorm-scalarC* **by** *blast*
show ‹$norm\ (r *_R a) = |r| * norm\ a$›
apply *transfer* **using** *bounded-clinear.bounded-linear onorm-scaleR* **by** *blast*
show ‹$r *_R (a + b) = r *_R a + r *_R b$›
apply *transfer* **by** (*simp add*: *scaleR-add-right*)
show ‹$(r + q) *_R a = r *_R a + q *_R a$›
apply *transfer* **by** (*simp add*: *scaleR-add-left*)
show ‹$r *_R q *_R a = (r * q) *_R a$›

apply *transfer* **by** *auto*
show ‹*1* $*_R$ *a* = *a*›
apply *transfer* **by** *auto*
qed

end

declare *uniformity-Abort*[**where** *'a*=(*'a* :: *complex-normed-vector*) $\Rightarrow_{CL}$ (*'b* :: *complex-normed-vector*), *code*]

lemma *norm-cblinfun-eqI*:
assumes *n* ≤ *norm* (*cblinfun-apply f x*) / *norm x*
assumes ⋀*x*. *norm* (*cblinfun-apply f x*) ≤ *n* * *norm x*
assumes *0* ≤ *n*
shows *norm f* = *n*
by (*auto simp*: *norm-cblinfun-def*
intro!: *antisym onorm-bound assms order-trans*[*OF - le-onorm*] *bounded-clinear.bounded-linear*
bounded-linear-intros)

lemma *norm-cblinfun*: *norm* (*cblinfun-apply f x*) ≤ *norm f* * *norm x*
apply *transfer* **by** (*simp add*: *bounded-clinear.bounded-linear onorm*)

lemma *norm-cblinfun-bound*: *0* ≤ *b* ⟹ (⋀*x*. *norm* (*cblinfun-apply f x*) ≤ *b* * *norm x*) ⟹ *norm f* ≤ *b*
by *transfer* (*rule onorm-bound*)

lemma *bounded-cbilinear-cblinfun-apply*[*bounded-cbilinear*]: *bounded-cbilinear cblinfun-apply*
proof
fix *f g*::*'a* $\Rightarrow_{CL}$ *'b* **and** *a b*::*'a* **and** *r*::*complex*
show (*f* + *g*) *a* = *f a* + *g a* (*r* $*_C$ *f*) *a* = *r* $*_C$ *f a*
by (*transfer*, *simp*)+
interpret *bounded-clinear f* **for** *f*::*'a* $\Rightarrow_{CL}$ *'b*
by (*auto intro*!: *bounded-linear-intros*)
show *f* (*a* + *b*) = *f a* + *f b f* (*r* $*_C$ *a*) = *r* $*_C$ *f a*
by (*simp-all add*: *add scaleC*)
show ∃ *K*. ∀ *a b*. *norm* (*cblinfun-apply a b*) ≤ *norm a* * *norm b* * *K*
by (*auto intro*!: *exI*[**where** *x=1*] *norm-cblinfun*)
qed

interpretation *cblinfun*: *bounded-cbilinear cblinfun-apply*
by (*rule bounded-cbilinear-cblinfun-apply*)

lemmas *bounded-clinear-apply-cblinfun*[*intro*, *simp*] = *cblinfun.bounded-clinear-left*

declare *cblinfun.zero-left* [*simp*] *cblinfun.zero-right* [*simp*]

context *bounded-cbilinear*

begin

named-theorems *cbilinear-simps*

lemmas [*cbilinear-simps*] =
add-left
add-right
diff-left
diff-right
minus-left
minus-right
scaleC-left
scaleC-right
zero-left
zero-right
sum-left
sum-right

end

instance *cblinfun* :: (*complex-normed-vector*, *cbanach*) *cbanach*

proof
fix X::$nat \Rightarrow 'a \Rightarrow_{CL} 'b$
assume *Cauchy X*
{
fix x::$'a$
{
fix x::$'a$
assume *norm x* $\leq$ *1*
have *Cauchy* ($\lambda n.$ *X n x*)
proof (*rule CauchyI*)
fix *e*::*real*
assume *0* $<$ *e*
from *CauchyD*[*OF* ‹*Cauchy X*› ‹*0* $<$ *e*›] **obtain** *M*
where *M*: $\bigwedge m\ n.\ m \geq M \Longrightarrow n \geq M \Longrightarrow$ *norm* (*X m* $-$ *X n*) $<$ *e*
by *auto*
show $\exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.$ *norm* (*X m x* $-$ *X n x*) $<$ *e*
proof (*safe intro*!: *exI*[**where** *x*=*M*])
fix *m n*
assume *le*: *M* $\leq$ *m M* $\leq$ *n*
have *norm* (*X m x* $-$ *X n x*) = *norm* ((*X m* $-$ *X n*) *x*)
by (*simp add*: *cblinfun.cbilinear-simps*)
also have ... $\leq$ *norm* (*X m* $-$ *X n*) $*$ *norm x*
by (*rule norm-cblinfun*)
also have ... $\leq$ *norm* (*X m* $-$ *X n*) $*$ *1*
using ‹*norm x* $\leq$ *1*› *norm-ge-zero* **by** (*rule mult-left-mono*)
also have ... = *norm* (*X m* $-$ *X n*) **by** *simp*

```
      also have ... < e using le by fact
      finally show norm (X m x − X n x) < e .
    qed
  qed
  hence convergent (λn. X n x)
    by (metis Cauchy-convergent-iff)
 } note convergent-norm1 = this
 define y where y = x /R norm x
 have y: norm y ≤ 1 and xy: x = norm x *R y
  by (simp-all add: y-def inverse-eq-divide)
 have convergent (λn. norm x *R X n y)
    by (intro bounded-bilinear.convergent[OF bounded-bilinear-scaleR] convergent-const
      convergent-norm1 y)
 also have (λn. norm x *R X n y) = (λn. X n x)
  by (metis cblinfun.scaleC-right scaleR-scaleC xy)
 finally have convergent (λn. X n x) .
}
then obtain v where v: ⋀x. (λn. X n x) ⟶ v x
 unfolding convergent-def
 by metis

have Cauchy (λn. norm (X n))
proof (rule CauchyI)
 fix e::real
 assume e > 0
 from CauchyD[OF ‹Cauchy X› ‹0 < e›] obtain M
  where M: ⋀m n. m ≥ M ⟹ n ≥ M ⟹ norm (X m − X n) < e
  by auto
 show ∃ M. ∀ m≥M. ∀ n≥M. norm (norm (X m) − norm (X n)) < e
 proof (safe intro!: exI[where x=M])
  fix m n assume mn: m ≥ M n ≥ M
  have norm (norm (X m) − norm (X n)) ≤ norm (X m − X n)
   by (metis norm-triangle-ineq3 real-norm-def)
  also have ... < e using mn by fact
  finally show norm (norm (X m) − norm (X n)) < e .
 qed
qed
then obtain K where K: (λn. norm (X n)) ⟶ K
 unfolding Cauchy-convergent-iff convergent-def
 by metis

have bounded-clinear v
proof
 fix x y and r::complex
 from tendsto-add[OF v[of x] v [of y]] v[of x + y, unfolded cblinfun.cbilinear-simps]
    tendsto-scaleC[OF tendsto-const[of r] v[of x]] v[of r *C x, unfolded cblinfun.cbilinear-simps]
 show v (x + y) = v x + v y v (r *C x) = r *C v x
```

```
    by (metis (poly-guards-query) LIMSEQ-unique)+
   show ∃K. ∀x. norm (v x) ≤ norm x * K
   proof (safe intro!: exI[where x=K])
    fix x
    have norm (v x) ≤ K * norm x
    apply (rule tendsto-le[OF - tendsto-mult[OF K tendsto-const] tendsto-norm[OF
v]])
     by (auto simp: norm-cblinfun)
    thus norm (v x) ≤ norm x * K
     by (simp add: ac-simps)
   qed
  qed
  hence Bv: ⋀x. (λn. X n x) ⟶ CBlinfun v x
   by (auto simp: bounded-clinear-CBlinfun-apply v)

  have X ⟶ CBlinfun v
  proof (rule LIMSEQ-I)
   fix r::real assume r > 0
   define r' where r' = r / 2
   have 0 < r' r' < r using ‹r > 0› by (simp-all add: r'-def)
   from CauchyD[OF ‹Cauchy X› ‹r' > 0›]
   obtain M where M: ⋀m n. m ≥ M ⟹ n ≥ M ⟹ norm (X m - X n) < r'
    by metis
   show ∃no. ∀n≥no. norm (X n - CBlinfun v) < r
   proof (safe intro!: exI[where x=M])
    fix n assume n: M ≤ n
    have norm (X n - CBlinfun v) ≤ r'
    proof (rule norm-cblinfun-bound)
     fix x
     have eventually (λm. m ≥ M) sequentially
      by (metis eventually-ge-at-top)
      hence ev-le: eventually (λm. norm (X n x - X m x) ≤ r' * norm x)
sequentially
      proof eventually-elim
       case (elim m)
       have norm (X n x - X m x) = norm ((X n - X m) x)
        by (simp add: cblinfun.cbilinear-simps)
       also have ... ≤ norm ((X n - X m)) * norm x
        by (rule norm-cblinfun)
       also have ... ≤ r' * norm x
        using M[OF n elim] by (simp add: mult-right-mono)
       finally show ?case .
      qed
      have tendsto-v: (λm. norm (X n x - X m x)) ⟶ norm (X n x -
CBlinfun v x)
       by (auto intro!: tendsto-intros Bv)
      show norm ((X n - CBlinfun v) x) ≤ r' * norm x
     by (auto intro!: tendsto-upperbound tendsto-v ev-le simp: cblinfun.cbilinear-simps)
    qed (simp add: ‹0 < r'› less-imp-le)
```

thus *norm* $(X\ n - CBlinfun\ v) < r$
by (*metis* ‹$r' < r$› *le-less-trans*)
qed
qed
thus *convergent X*
by (*rule convergentI*)
qed

12.5 On Euclidean Space

lemma *norm-cblinfun-ceuclidean-le*:
fixes *a*::'*a*::*ceuclidean-space* $\Rightarrow_{CL}$ 'b::*complex-normed-vector*
shows *norm a* $\leq$ *sum* $(\lambda x.\ norm\ (a\ x))$ *CBasis*
apply (*rule norm-cblinfun-bound*)
apply (*simp add*: *sum-nonneg*)
apply (*subst ceuclidean-representation*[*symmetric*, **where** 'a='a])
apply (*simp only*: *cblinfun.cbilinear-simps sum-distrib-right*)
apply (*rule order.trans*[*OF norm-sum sum-mono*])
apply (*simp add*: *abs-mult mult-right-mono ac-simps CBasis-le-norm*)
by (*metis complex-inner-class.Cauchy-Schwarz-ineq2 mult.commute mult.left-neutral mult-right-mono norm-CBasis norm-ge-zero*)

lemma *ctendsto-componentwise1*:
fixes *a*::'*a*::*ceuclidean-space* $\Rightarrow_{CL}$ 'b::*complex-normed-vector*
and *b*::'*c* $\Rightarrow$ 'a $\Rightarrow_{CL}$ 'b
assumes $(\bigwedge j.\ j \in CBasis \Longrightarrow ((\lambda n.\ b\ n\ j) \longrightarrow a\ j)\ F)$
shows $(b \longrightarrow a)\ F$
proof −
have $\bigwedge j.\ j \in CBasis \Longrightarrow Zfun\ (\lambda x.\ norm\ (b\ x\ j - a\ j))\ F$
using *assms* **unfolding** *tendsto-Zfun-iff Zfun-norm-iff* **.**
hence *Zfun* $(\lambda x.\ \sum j{\in}CBasis.\ norm\ (b\ x\ j - a\ j))\ F$
by (*auto intro*!: *Zfun-sum*)
thus *?thesis*
unfolding *tendsto-Zfun-iff*
by (*rule Zfun-le*)
(*auto intro*!: *order-trans*[*OF norm-cblinfun-ceuclidean-le*] *simp*: *cblinfun.cbilinear-simps*)
qed

lift-definition
cblinfun-of-matrix::('b::*ceuclidean-space* $\Rightarrow$ 'a::*ceuclidean-space* $\Rightarrow$ *complex*) $\Rightarrow$ 'a $\Rightarrow_{CL}$ 'b
is $\lambda a\ x.\ \sum i{\in}CBasis.\ \sum j{\in}CBasis.\ ((j \cdot_C x) * a\ i\ j) *_C i$
by (*intro bounded-linear-intros*)

lemma *cblinfun-of-matrix-works*:
fixes *f*::'*a*::*ceuclidean-space* $\Rightarrow_{CL}$ 'b::*ceuclidean-space*
shows *cblinfun-of-matrix* $(\lambda i\ j.\ i \cdot_C (f\ j)) = f$
proof (*transfer*, *rule*, *rule ceuclidean-eqI*)
fix *f*::'*a* $\Rightarrow$ 'b **and** *x*::'*a* **and** *b*::'*b* **assume** *bounded-clinear f* **and** *b*: $b \in CBasis$

then interpret *bounded-clinear f* **by** *simp*
have $(\sum j{\in}CBasis.\ \sum i{\in}CBasis.\ (i \cdot_C x * (j \cdot_C f\ i)) *_C j) \cdot_C b$
$= (\sum j{\in}CBasis.\ if\ j = b\ then\ (\sum i{\in}CBasis.\ (x \cdot_C i * (f\ i \cdot_C j)))\ else\ 0)$
using *b*
apply (*simp add*: *cinner-sum-left cinner-CBasis if-distrib cong*: *if-cong*)
by (*simp add*: *sum.swap*)
also have $\ldots = (\sum i{\in}CBasis.\ ((x \cdot_C i) * (f\ i \cdot_C b)))$
using *b* **by** (*simp*)
also have $\ldots = f\ x \cdot_C b$
proof −
have ‹$(\sum i{\in}CBasis.\ (x \cdot_C i) * (f\ i \cdot_C b)) = (\sum i{\in}CBasis.\ (i \cdot_C x) *_C f\ i) \cdot_C b$›
by (*auto simp*: *cinner-sum-left*)
also have ‹$\ldots = f\ x \cdot_C b$›
by (*simp add*: *ceuclidean-representation sum*[*symmetric*] *scale*[*symmetric*])
finally show *?thesis* **by** −
qed
finally show $(\sum j{\in}CBasis.\ \sum i{\in}CBasis.\ (i \cdot_C x * (j \cdot_C f\ i)) *_C j) \cdot_C b = f\ x \cdot_C b$ **.**
qed

lemma *cblinfun-of-matrix-apply*:
cblinfun-of-matrix a x $= (\sum i{\in}CBasis.\ \sum j{\in}CBasis.\ ((j \cdot_C x) * a\ i\ j) *_C i)$
apply *transfer* **by** *simp*

lemma *cblinfun-of-matrix-minus*: *cblinfun-of-matrix x* − *cblinfun-of-matrix y* = *cblinfun-of-matrix* $(x - y)$
by *transfer* (*auto simp*: *algebra-simps sum-subtractf*)

lemma *norm-cblinfun-of-matrix*:
norm (*cblinfun-of-matrix a*) $\leq (\sum i{\in}CBasis.\ \sum j{\in}CBasis.\ cmod\ (a\ i\ j))$
apply (*rule norm-cblinfun-bound*)
apply (*simp add*: *sum-nonneg*)
apply (*simp only*: *cblinfun-of-matrix-apply sum-distrib-right*)
apply (*rule order-trans*[*OF norm-sum sum-mono*])
apply (*rule order-trans*[*OF norm-sum sum-mono*])
apply (*simp add*: *abs-mult mult-right-mono ac-simps Basis-le-norm*)
by (*metis complex-inner-class.Cauchy-Schwarz-ineq2 complex-scaleC-def mult.left-neutral mult-right-mono norm-CBasis norm-ge-zero norm-scaleC*)

lemma *tendsto-cblinfun-of-matrix*:
assumes $\bigwedge i\ j.\ i \in CBasis \Longrightarrow j \in CBasis \Longrightarrow ((\lambda n.\ b\ n\ i\ j) \longrightarrow a\ i\ j)\ F$
shows $((\lambda n.\ cblinfun\text{-}of\text{-}matrix\ (b\ n)) \longrightarrow cblinfun\text{-}of\text{-}matrix\ a)\ F$
proof −
have $\bigwedge i\ j.\ i \in CBasis \Longrightarrow j \in CBasis \Longrightarrow Zfun\ (\lambda x.\ norm\ (b\ x\ i\ j - a\ i\ j))\ F$
using *assms* **unfolding** *tendsto-Zfun-iff Zfun-norm-iff* **.**
hence $Zfun\ (\lambda x.\ (\sum i{\in}CBasis.\ \sum j{\in}CBasis.\ cmod\ (b\ x\ i\ j - a\ i\ j)))\ F$
by (*auto intro*!: *Zfun-sum*)

thus *?thesis*
unfolding *tendsto-Zfun-iff cblinfun-of-matrix-minus*
by (*rule Zfun-le*) (*auto intro*!: *order-trans*[*OF norm-cblinfun-of-matrix*])
qed

lemma *ctendsto-componentwise*:
fixes *a*::*'a*::*ceuclidean-space* $\Rightarrow_{CL}$ *'b*::*ceuclidean-space*
and *b*::*'c* $\Rightarrow$ *'a* $\Rightarrow_{CL}$ *'b*
shows ($\bigwedge i\ j.\ i \in CBasis \Longrightarrow j \in CBasis \Longrightarrow ((\lambda n.\ b\ n\ j \cdot_C i) \longrightarrow a\ j \cdot_C i)$ *F*) $\Longrightarrow$ $(b \longrightarrow a)$ *F*
apply (*subst cblinfun-of-matrix-works*[*of a*, *symmetric*])
apply (*subst cblinfun-of-matrix-works*[*of b x* **for** *x*, *symmetric*, *abs-def*])
apply (*rule tendsto-cblinfun-of-matrix*)
apply (*subst* (*1*) *cinner-commute*, *subst* (*2*) *cinner-commute*)
by (*metis lim-cnj*)

lemma
continuous-cblinfun-componentwiseI:
fixes *f*:: *'b*::*t2-space* $\Rightarrow$ *'a*::*ceuclidean-space* $\Rightarrow_{CL}$ *'c*::*ceuclidean-space*
assumes $\bigwedge i\ j.\ i \in CBasis \Longrightarrow j \in CBasis \Longrightarrow$ *continuous F* $(\lambda x.\ (f\ x)\ j \cdot_C i)$
shows *continuous F f*
using *assms* **by** (*auto simp*: *continuous-def intro*!: *ctendsto-componentwise*)

lemma
continuous-cblinfun-componentwiseI1:
fixes *f*:: *'b*::*t2-space* $\Rightarrow$ *'a*::*ceuclidean-space* $\Rightarrow_{CL}$ *'c*::*complex-normed-vector*
assumes $\bigwedge i.\ i \in CBasis \Longrightarrow$ *continuous F* $(\lambda x.\ f\ x\ i)$
shows *continuous F f*
using *assms* **by** (*auto simp*: *continuous-def intro*!: *ctendsto-componentwise1*)

lemma
continuous-on-cblinfun-componentwise:
fixes *f*:: *'d*::*t2-space* $\Rightarrow$ *'e*::*ceuclidean-space* $\Rightarrow_{CL}$ *'f*::*complex-normed-vector*
assumes $\bigwedge i.\ i \in CBasis \Longrightarrow$ *continuous-on s* $(\lambda x.\ f\ x\ i)$
shows *continuous-on s f*
using *assms*
by (*auto intro*!: *continuous-at-imp-continuous-on intro*!: *ctendsto-componentwise1*
simp: *continuous-on-eq-continuous-within continuous-def*)

lemma *bounded-antilinear-cblinfun-matrix*: *bounded-antilinear* $(\lambda x.\ (x$::-$\Rightarrow_{CL}$ -$)\ j \cdot_C i)$
by (*auto intro*!: *bounded-linear-intros*)

lemma *continuous-cblinfun-matrix*:
fixes *f*:: *'b*::*t2-space* $\Rightarrow$ *'a*::*complex-normed-vector* $\Rightarrow_{CL}$ *'c*::*complex-inner*
assumes *continuous F f*
shows *continuous F* $(\lambda x.\ (f\ x)\ j \cdot_C i)$
by (*rule bounded-antilinear.continuous*[*OF bounded-antilinear-cblinfun-matrix assms*])

lemma *continuous-on-cblinfun-matrix*:
fixes f::$'a$::*t2-space* $\Rightarrow$ $'b$::*complex-normed-vector* $\Rightarrow_{CL}$ $'c$::*complex-inner*
assumes *continuous-on S f*
shows *continuous-on S* ($\lambda x.$ (f x) j $\cdot_C$ i)
using *assms*
by (*auto simp*: *continuous-on-eq-continuous-within continuous-cblinfun-matrix*)

lemma *continuous-on-cblinfun-of-matrix*[*continuous-intros*]:
assumes $\bigwedge i\ j.\ i \in CBasis \Longrightarrow j \in CBasis \Longrightarrow$ *continuous-on S* ($\lambda s.$ g s i j)
shows *continuous-on S* ($\lambda s.$ *cblinfun-of-matrix* (g s))
using *assms*
by (*auto simp*: *continuous-on intro*!: *tendsto-cblinfun-of-matrix*)

lemma *cblinfun-euclidean-eqI*: ($\bigwedge i.\ i \in CBasis \Longrightarrow$ *cblinfun-apply* x i = *cblinfun-apply* y i) $\Longrightarrow$ $x = y$
apply (*auto intro*!: *cblinfun-eqI*)
apply (*subst* (*2*) *ceuclidean-representation*[*symmetric*, **where** $'a$=$'a$])
apply (*subst* (*1*) *ceuclidean-representation*[*symmetric*, **where** $'a$=$'a$])
by (*simp add*: *cblinfun.cbilinear-simps*)

lemma *CBlinfun-eq-matrix*: *bounded-clinear f* $\Longrightarrow$ *CBlinfun f* = *cblinfun-of-matrix* ($\lambda i\ j.\ i \cdot_C f\ j$)
apply (*intro cblinfun-euclidean-eqI*)
by (*auto simp*: *cblinfun-of-matrix-apply bounded-clinear-CBlinfun-apply cinner-CBasis if-distrib*
if-distribR sum.delta$'$ *ceuclidean-representation*
cong: *if-cong*)

12.6 concrete bounded linear functions

lemma *transfer-bounded-cbilinear-bounded-clinearI*:
assumes g = ($\lambda i\ x.$ (*cblinfun-apply* (f i) x))
shows *bounded-cbilinear g* = *bounded-clinear f*
proof
assume *bounded-cbilinear g*
then interpret *bounded-cbilinear f* **by** (*simp add*: *assms*)
show *bounded-clinear f*
proof (*unfold-locales*, *safe intro*!: *cblinfun-eqI*)
fix i
show f (x + y) i = (f x + f y) i f (r $*_C$ x) i = (r $*_C$ f x) i **for** r x y
by (*auto intro*!: *cblinfun-eqI simp*: *cblinfun.cbilinear-simps*)

from - *nonneg-bounded* **show** $\exists K.\ \forall x.\ norm\ (f\ x) \leq norm\ x * K$
by (*rule ex-reg*) (*auto intro*!: *onorm-bound simp*: *norm-cblinfun.rep-eq ac-simps*)
qed
qed (*auto simp*: *assms intro*!: *cblinfun.comp*)

lemma *transfer-bounded-cbilinear-bounded-clinear*[*transfer-rule*]:
(*rel-fun* (*rel-fun* (=) (*pcr-cblinfun* (=) (=))) (=)) *bounded-cbilinear bounded-clinear*
by (*auto simp*: *pcr-cblinfun-def cr-cblinfun-def rel-fun-def OO-def*
intro!: *transfer-bounded-cbilinear-bounded-clinearI*)

lemma *transfer-bounded-sesquilinear-bounded-antilinearI*:
assumes $g = (\lambda i\ x.\ (cblinfun\text{-}apply\ (f\ i)\ x))$
shows *bounded-sesquilinear g* = *bounded-antilinear f*
proof
assume *bounded-sesquilinear g*
then interpret *bounded-sesquilinear f* **by** (*simp add*: *assms*)
show *bounded-antilinear f*
proof (*unfold-locales*, *safe intro*!: *cblinfun-eqI*)
fix *i*
show $f\ (x + y)\ i = (f\ x + f\ y)\ i\ f\ (r *_C x)\ i = (cnj\ r *_C f\ x)\ i$ **for** *r x y*
by (*auto intro*!: *cblinfun-eqI simp*: *cblinfun.scaleC-left scaleC-left add-left*
cblinfun.add-left)
from - *real.nonneg-bounded* **show** $\exists K.\ \forall x.\ norm\ (f\ x) \leq norm\ x * K$
by (*rule ex-reg*) (*auto intro*!: *onorm-bound simp*: *norm-cblinfun.rep-eq ac-simps*)
qed
next
assume *bounded-antilinear f*
then obtain *K* **where** *K*: $\langle norm\ (f\ x) \leq norm\ x * K\rangle$ **for** *x*
using *bounded-antilinear.bounded* **by** *blast*
have $\langle norm\ (cblinfun\text{-}apply\ (f\ a)\ b) \leq norm\ (f\ a) * norm\ b\rangle$ **for** *a b*
by (*simp add*: *norm-cblinfun*)
also have $\langle \ldots\ a\ b \leq norm\ a * norm\ b * K\rangle$ **for** *a b*
by (*smt* (*verit*, *best*) *K mult.assoc mult.commute mult-mono' norm-ge-zero*)
finally have *: $\langle norm\ (cblinfun\text{-}apply\ (f\ a)\ b) \leq norm\ a * norm\ b * K\rangle$ **for** *a b*
by *simp*
show *bounded-sesquilinear g*
using $\langle bounded\text{-}antilinear\ f\rangle$
apply (*auto intro*!: *bounded-sesquilinear.intro simp*: *assms cblinfun.add-left*
cblinfun.add-right
linear-simps bounded-antilinear.bounded-linear antilinear.scaleC bounded-antilinear.antilinear
cblinfun.scaleC-left cblinfun.scaleC-right)
using * **by** *blast*
qed

lemma *transfer-bounded-sesquilinear-bounded-antilinear*[*transfer-rule*]:
(*rel-fun* (*rel-fun* (=) (*pcr-cblinfun* (=) (=))) (=)) *bounded-sesquilinear bounded-antilinear*
by (*auto simp*: *pcr-cblinfun-def cr-cblinfun-def rel-fun-def OO-def*
intro!: *transfer-bounded-sesquilinear-bounded-antilinearI*)

context *bounded-cbilinear*
begin

lift-definition *prod-left*::*'b* ⇒ *'a* ⇒$_{CL}$ *'c* **is** (λ*b a*. *prod a b*)
by (*rule bounded-clinear-left*)
declare *prod-left.rep-eq*[*simp*]

lemma *bounded-clinear-prod-left*[*bounded-clinear*]: *bounded-clinear prod-left*
by *transfer* (*rule flip*)

lift-definition *prod-right*::*'a* ⇒ *'b* ⇒$_{CL}$ *'c* **is** (λ*a b*. *prod a b*)
by (*rule bounded-clinear-right*)
declare *prod-right.rep-eq*[*simp*]

lemma *bounded-clinear-prod-right*[*bounded-clinear*]: *bounded-clinear prod-right*
by *transfer* (*rule bounded-cbilinear-axioms*)

end

lift-definition *id-cblinfun*::*'a*::*complex-normed-vector* ⇒$_{CL}$ *'a* **is** λ*x*. *x*
by (*rule bounded-clinear-ident*)

lemmas *cblinfun-id-cblinfun-apply*[*simp*] = *id-cblinfun.rep-eq*

lemma *norm-cblinfun-id*[*simp*]:
norm (*id-cblinfun*::*'a*::{*complex-normed-vector*, *not-singleton*} ⇒$_{CL}$ *'a*) = *1*
apply *transfer*
apply (*rule onorm-id*[*internalize-sort' 'a*])
apply *standard*[*1*]
by *simp*

lemma *norm-cblinfun-id-le*:
norm (*id-cblinfun*::*'a*::*complex-normed-vector* ⇒$_{CL}$ *'a*) ≤ *1*
by *transfer* (*auto simp*: *onorm-id-le*)

lift-definition *cblinfun-compose*::
'a::*complex-normed-vector* ⇒$_{CL}$ *'b*::*complex-normed-vector* ⇒
'c::*complex-normed-vector* ⇒$_{CL}$ *'a* ⇒

$'c \Rightarrow_{CL} 'b$ (**infixl** ‹o_{CL}› *67*) **is** (*o*)

parametric *comp-transfer*
unfolding *o-def*
by (*rule bounded-clinear-compose*)

lemma *cblinfun-apply-cblinfun-compose*[*simp*]: (*a* o_{CL} *b*) *c* = *a* (*b c*)
by (*simp add*: *cblinfun-compose.rep-eq*)

lemma *norm-cblinfun-compose*:
norm (*f* o_{CL} *g*) ≤ *norm f* ∗ *norm g*
apply *transfer*
by (*auto intro*!: *onorm-compose simp*: *bounded-clinear.bounded-linear*)

lemma *bounded-cbilinear-cblinfun-compose*[*bounded-cbilinear*]: *bounded-cbilinear* (o_{CL})
by *unfold-locales*
(*auto intro*!: *cblinfun-eqI exI*[**where** *x*=*1*] *simp*: *cblinfun.cbilinear-simps norm-cblinfun-compose*)

lemma *cblinfun-compose-zero*[*simp*]:
blinfun-compose 0 = (λ-. *0*)
blinfun-compose x 0 = *0*
by (*auto simp*: *blinfun.bilinear-simps intro*!: *blinfun-eqI*)

lemma *cblinfun-bij2*:
fixes *f*::*'a* ⇒$_{CL}$ *'a*::*ceuclidean-space*
assumes *f* o_{CL} *g* = *id-cblinfun*
shows *bij* (*cblinfun-apply g*)
proof (*rule bijI*)
show *inj g*
using *assms*
by (*metis cblinfun-id-cblinfun-apply cblinfun-compose.rep-eq injI inj-on-imageI2*)
then show *surj g*
using *bounded-clinear-def cblinfun.bounded-clinear-right ceucl.linear-inj-imp-surj*
by *blast*
qed

lemma *cblinfun-bij1*:
fixes *f*::*'a* ⇒$_{CL}$ *'a*::*ceuclidean-space*
assumes *f* o_{CL} *g* = *id-cblinfun*
shows *bij* (*cblinfun-apply f*)
proof (*rule bijI*)
show *surj* (*cblinfun-apply f*)
by (*metis assms cblinfun-apply-cblinfun-compose cblinfun-id-cblinfun-apply surjI*)
then show *inj* (*cblinfun-apply f*)
using *bounded-clinear-def cblinfun.bounded-clinear-right ceucl.linear-surjective-imp-injective*
by *blast*
qed

lift-definition *cblinfun-cinner-right*::*'a*::*complex-inner* ⇒ *'a* ⇒$_{CL}$ *complex* **is** ($\cdot_C$)

by (*rule bounded-clinear-cinner-right*)
declare *cblinfun-cinner-right.rep-eq*[*simp*]

lemma *bounded-antilinear-cblinfun-cinner-right*[*bounded-antilinear*]: *bounded-antilinear cblinfun-cinner-right*
apply *transfer* **by** (*simp add*: *bounded-sesquilinear-cinner*)

lift-definition *cblinfun-scaleC-right*::*complex* ⇒ *'a* ⇒$_{CL}$ *'a*::*complex-normed-vector* **is** (∗$_C$)
by (*rule bounded-clinear-scaleC-right*)
declare *cblinfun-scaleC-right.rep-eq*[*simp*]

lemma *bounded-clinear-cblinfun-scaleC-right*[*bounded-clinear*]: *bounded-clinear cblinfun-scaleC-right*
by *transfer* (*rule bounded-cbilinear-scaleC*)

lift-definition *cblinfun-scaleC-left*::*'a*::*complex-normed-vector* ⇒ *complex* ⇒$_{CL}$ *'a* **is** λ*x y*. *y* ∗$_C$ *x*
by (*rule bounded-clinear-scaleC-left*)
lemmas [*simp*] = *cblinfun-scaleC-left.rep-eq*

lemma *bounded-clinear-cblinfun-scaleC-left*[*bounded-clinear*]: *bounded-clinear cblinfun-scaleC-left*
by *transfer* (*rule bounded-cbilinear.flip*[*OF bounded-cbilinear-scaleC*])

lift-definition *cblinfun-mult-right*::*'a* ⇒ *'a* ⇒$_{CL}$ *'a*::*complex-normed-algebra* **is** (∗)
by (*rule bounded-clinear-mult-right*)
declare *cblinfun-mult-right.rep-eq*[*simp*]

lemma *bounded-clinear-cblinfun-mult-right*[*bounded-clinear*]: *bounded-clinear cblinfun-mult-right*
by *transfer* (*rule bounded-cbilinear-mult*)

lift-definition *cblinfun-mult-left*::*'a*::*complex-normed-algebra* ⇒ *'a* ⇒$_{CL}$ *'a* **is** λ*x y*. *y* ∗ *x*
by (*rule bounded-clinear-mult-left*)
lemmas [*simp*] = *cblinfun-mult-left.rep-eq*

lemma *bounded-clinear-cblinfun-mult-left*[*bounded-clinear*]: *bounded-clinear cblinfun-mult-left*
by *transfer* (*rule bounded-cbilinear.flip*[*OF bounded-cbilinear-mult*])

lemmas *bounded-clinear-function-uniform-limit-intros*[*uniform-limit-intros*] =

bounded-clinear.uniform-limit[*OF bounded-clinear-apply-cblinfun*]
bounded-clinear.uniform-limit[*OF bounded-clinear-cblinfun-apply*]
bounded-antilinear.uniform-limit[*OF bounded-antilinear-cblinfun-matrix*]

12.7 The strong operator topology on continuous linear operators

Let $'a$ and $'b$ be two normed real vector spaces. Then the space of linear continuous operators from $'a$ to $'b$ has a canonical norm, and therefore a canonical corresponding topology (the type classes instantiation are given in `Complex_Bounded_Linear_Function0.thy`).

However, there is another topology on this space, the strong operator topology, where T_n tends to T iff, for all x in $'a$, then $T_n\ x$ tends to $T\ x$. This is precisely the product topology where the target space is endowed with the norm topology. It is especially useful when $'b$ is the set of real numbers, since then this topology is compact.

We can not implement it using type classes as there is already a topology, but at least we can define it as a topology.

Note that there is yet another (common and useful) topology on operator spaces, the weak operator topology, defined analogously using the product topology, but where the target space is given the weak-* topology, i.e., the pullback of the weak topology on the bidual of the space under the canonical embedding of a space into its bidual. We do not define it there, although it could also be defined analogously.

definition *cstrong-operator-topology*::(*'a*::*complex-normed-vector* $\Rightarrow_{CL}$ *'b*::*complex-normed-vector*) *topology*
where *cstrong-operator-topology* = *pullback-topology UNIV cblinfun-apply euclidean*

lemma *cstrong-operator-topology-topspace*:
topspace cstrong-operator-topology = *UNIV*
unfolding *cstrong-operator-topology-def topspace-pullback-topology topspace-euclidean*
by *auto*

lemma *cstrong-operator-topology-basis*:
fixes *f*::(*'a*::*complex-normed-vector* $\Rightarrow_{CL}$ *'b*::*complex-normed-vector*) **and** *U*::*'i* $\Rightarrow$ *'b set* **and** *x*::*'i* $\Rightarrow$ *'a*
assumes *finite I* $\bigwedge i.\ i \in I \Longrightarrow$ *open* (*U i*)
shows *openin cstrong-operator-topology* {*f*. $\forall i{\in}I.$ *cblinfun-apply f* (*x i*) $\in$ *U i*}
proof −
have *open* {*g*::(*'a*$\Rightarrow$*'b*). $\forall i{\in}I.$ *g* (*x i*) $\in$ *U i*}
by (*rule product-topology-basis'*[*OF assms*])
moreover have {*f*. $\forall i{\in}I.$ *cblinfun-apply f* (*x i*) $\in$ *U i*}
= *cblinfun-apply*−‘{*g*::(*'a*$\Rightarrow$*'b*). $\forall i{\in}I.$ *g* (*x i*) $\in$ *U i*} $\cap$ *UNIV*
by *auto*
ultimately show *?thesis*

unfolding *cstrong-operator-topology-def* **by** (*subst openin-pullback-topology*)
auto
qed

lemma *cstrong-operator-topology-continuous-evaluation*:
continuous-map cstrong-operator-topology euclidean (λ*f*. *cblinfun-apply f x*)
proof −
have *continuous-map cstrong-operator-topology euclidean* ((λ*f*. *f x*) *o cblinfun-apply*)
unfolding *cstrong-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-cstrong-operator-topo-iff-coordinatewise*:
continuous-map T cstrong-operator-topology f
⟷ (∀ *x*. *continuous-map T euclidean* (λ*y*. *cblinfun-apply* (*f y*) *x*))
proof (*auto*)
fix *x*::′*b*
assume *continuous-map T cstrong-operator-topology f*
with *continuous-map-compose*[*OF this cstrong-operator-topology-continuous-evaluation*]
have *continuous-map T euclidean* ((λ*z*. *cblinfun-apply z x*) *o f*)
by *simp*
then show *continuous-map T euclidean* (λ*y*. *cblinfun-apply* (*f y*) *x*)
unfolding *comp-def* **by** *auto*
next
assume ∗: ∀ *x*. *continuous-map T euclidean* (λ*y*. *cblinfun-apply* (*f y*) *x*)
have ⋀*i*. *continuous-map T euclidean* (λ*x*. *cblinfun-apply* (*f x*) *i*)
using ∗ **unfolding** *comp-def* **by** *auto*
then have *continuous-map T euclidean* (*cblinfun-apply o f*)
unfolding *o-def*
by (*metis* (*no-types*) *continuous-map-componentwise-UNIV euclidean-product-topology*)
show *continuous-map T cstrong-operator-topology f*
unfolding *cstrong-operator-topology-def*
apply (*rule continuous-map-pullback′*)
by (*auto simp add*: ‹*continuous-map T euclidean* (*cblinfun-apply o f*)›)
qed

lemma *cstrong-operator-topology-weaker-than-euclidean*:
continuous-map euclidean cstrong-operator-topology (λ*f*. *f*)
apply (*subst continuous-on-cstrong-operator-topo-iff-coordinatewise*)
by (*auto simp add*: *linear-continuous-on continuous-at-imp-continuous-on linear-continuous-at*
bounded-clinear.bounded-linear)
end

13 *Complex-Bounded-Linear-Function* – Complex bounded linear functions (bounded operators)

theory *Complex-Bounded-Linear-Function*
imports
HOL−Types-To-Sets.Types-To-Sets
HOL−Library.Function-Algebras
Banach-Steinhaus.Banach-Steinhaus
Wlog.Wlog

Complex-Inner-Product
One-Dimensional-Spaces
Complex-Bounded-Linear-Function0
begin

unbundle *lattice-syntax*

13.1 Misc basic facts and declarations

notation *cblinfun-apply* (**infixr** ‹$*_V$› *70*)

lemma *id-cblinfun-apply*[*simp*]: *id-cblinfun* $*_V$ $\psi = \psi$
by *simp*

lemma *apply-id-cblinfun*[*simp*]: ‹(*$*_V$*) *id-cblinfun* = *id*›
by *auto*

lemma *isCont-cblinfun-apply*[*simp*]: *isCont* ((*$*_V$*) *A*) ψ
by *transfer* (*simp add*: *clinear-continuous-at*)

declare *cblinfun.scaleC-left*[*simp*]

lemma *cblinfun-apply-clinear*[*simp*]: ‹*clinear* (*cblinfun-apply A*)›
using *bounded-clinear.axioms*(*1*) *cblinfun-apply* **by** *blast*

lemma *cblinfun-cinner-eqI*:
fixes *A B* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*a*›
assumes ‹$\bigwedge\psi$. *norm* ψ = *1* ⟹ *cinner* ψ (*A* $*_V$ ψ) = *cinner* ψ (*B* $*_V$ ψ)›
shows ‹*A* = *B*›
proof −
define *C* **where** ‹*C* = *A* − *B*›
have *C0*[*simp*]: ‹*cinner* ψ (*C* ψ) = *0*› **for** ψ
apply (*cases* ‹ψ = *0*›)
using *assms*[*of* ‹*sgn* ψ›]
by (*simp-all add*: *C-def norm-sgn sgn-div-norm cblinfun.scaleR-right assms cblinfun.diff-left cinner-diff-right*)
{ fix *f g* α
have ‹*0* = *cinner* (*f* + α $*_C$ *g*) (*C* $*_V$ (*f* + α $*_C$ *g*))›
by (*simp add*: *cinner-diff-right minus-cblinfun.rep-eq*)

also have ‹... = α *$_C$ cinner f (C g) + cnj α *$_C$ cinner g (C f)›
by (*smt* (*z3*) *C0 add.commute add.right-neutral cblinfun.add-right cblinfun.scaleC-right cblinfun-cinner-right.rep-eq cinner-add-left cinner-scaleC-left complex-scaleC-def*)
finally have ‹α *$_C$ cinner f (C g) = − cnj α *$_C$ cinner g (C f)›
by (*simp add*: *eq-neg-iff-add-eq-0*)
}
then have ‹*cinner f (C g) = 0*› **for** *f g*
by (*metis complex-cnj-i complex-cnj-one complex-vector.scale-cancel-right complex-vector.scale-left-imp-eq equation-minus-iff i-squared mult-eq-0-iff one-neq-neg-one*)
then have ‹*C g = 0*› **for** *g*
using *cinner-eq-zero-iff* **by** *blast*
then have ‹*C = 0*›
by (*simp add*: *cblinfun-eqI*)
then show ‹*A = B*›
using *C-def* **by** *auto*
qed

lemma *id-cblinfun-not-0*[*simp*]: ‹(*id-cblinfun* :: *'a*::{*complex-normed-vector*, *not-singleton*} $\Rightarrow_{CL}$ -) $\neq$ *0*›
by (*metis* (*full-types*) *Extra-General.UNIV-not-singleton cblinfun.zero-left cblinfun-id-cblinfun-apply ex-norm1 norm-zero one-neq-zero*)

lemma *cblinfun-norm-geqI*:
assumes ‹*norm* (*f *$_V$ *x*) / *norm x* $\geq$ *K*›
shows ‹*norm f* $\geq$ *K*›
using *assms* **by** *transfer* (*smt* (*z3*) *bounded-clinear.bounded-linear le-onorm*)

declare *scaleC-conv-of-complex*[*simp*]

lemma *cblinfun-eq-0-on-span*:
fixes *S*::‹*'a*::*complex-normed-vector set*›
assumes *x* $\in$ *cspan S*
and $\bigwedge$*s*. *s*$\in$*S* $\Longrightarrow$ *F *$_V$ *s = 0*
shows ‹*F *$_V$ *x = 0*›
using *bounded-clinear.axioms*(*1*) *cblinfun-apply assms complex-vector.linear-eq-0-on-span*
by *blast*

lemma *cblinfun-eq-on-span*:
fixes *S*::‹*'a*::*complex-normed-vector set*›
assumes *x* $\in$ *cspan S*
and $\bigwedge$*s*. *s*$\in$*S* $\Longrightarrow$ *F *$_V$ *s = G *$_V$ *s*
shows ‹*F *$_V$ *x = G *$_V$ *x*›
using *bounded-clinear.axioms*(*1*) *cblinfun-apply assms complex-vector.linear-eq-0-on-span*
by *blast*

lemma *cblinfun-eq-0-on-UNIV-span*:
fixes *basis*::‹*'a*::*complex-normed-vector set*›

assumes *cspan basis* = *UNIV*
and $\bigwedge s.\ s{\in}basis \Longrightarrow F *_V s = 0$
shows ‹*F* = *0*›
by (*metis cblinfun-eq-0-on-span UNIV-I assms cblinfun.zero-left cblinfun-eqI*)

lemma *cblinfun-eq-on-UNIV-span*:
fixes *basis*::'*a*::*complex-normed-vector set* **and** φ::'*a* ⇒ '*b*::*complex-normed-vector*
assumes *cspan basis* = *UNIV*
and $\bigwedge s.\ s{\in}basis \Longrightarrow F *_V s = G *_V s$
shows ‹*F* = *G*›
by (*metis* (*no-types*) *assms cblinfun-eqI cblinfun-eq-on-span iso-tuple-UNIV-I*)

lemma *cblinfun-eq-on-canonical-basis*:
fixes *f g*::'*a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ '*b*::*complex-normed-vector*
defines *basis* == *set* (*canonical-basis*::'*a list*)
assumes $\bigwedge u.\ u \in basis \Longrightarrow f *_V u = g *_V u$
shows *f* = *g*
using *assms cblinfun-eq-on-UNIV-span is-generator-set* **by** *blast*

lemma *cblinfun-eq-0-on-canonical-basis*:
fixes *f* ::'*a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ '*b*::*complex-normed-vector*
defines *basis* == *set* (*canonical-basis*::'*a list*)
assumes $\bigwedge u.\ u \in basis \Longrightarrow f *_V u = 0$
shows *f* = *0*
by (*simp add*: *assms cblinfun-eq-on-canonical-basis*)

lemma *cinner-canonical-basis-eq-0*:
defines *basisA* == *set* (*canonical-basis*::'*a*::*onb-enum list*)
and *basisB* == *set* (*canonical-basis*::'*b*::*onb-enum list*)
assumes $\bigwedge u\ v.\ u{\in}basisA \Longrightarrow v{\in}basisB \Longrightarrow$ *is-orthogonal v* (*F* $*_V$ *u*)
shows *F* = *0*
proof−
have *F* $*_V$ *u* = *0*
if *u*∈*basisA* **for** *u*
proof−
have $\bigwedge v.\ v{\in}basisB \Longrightarrow$ *is-orthogonal v* (*F* $*_V$ *u*)
by (*simp add*: *assms*(*3*) *that*)
moreover have ($\bigwedge v.\ v{\in}basisB \Longrightarrow$ *is-orthogonal v x*) ⟹ *x* = *0*
for *x*
proof−
assume *r1*: $\bigwedge v.\ v{\in}basisB \Longrightarrow$ *is-orthogonal v x*
have *is-orthogonal v x* **for** *v*
proof−
have *cspan basisB* = *UNIV*
using *basisB-def is-generator-set* **by** *auto*
hence *v* ∈ *cspan basisB*
by (*smt iso-tuple-UNIV-I*)
hence ∃ *t s*. *v* = ($\sum$ *a*∈*t*. *s a* $*_C$ *a*) ∧ *finite t* ∧ *t* ⊆ *basisB*
using *complex-vector.span-explicit*

by (*smt mem-Collect-eq*)
then obtain *t s* **where** *b1*: $v = (\sum a{\in}t.\ s\ a *_C a)$ **and** *b2*: *finite t* **and** *b3*: $t \subseteq$ *basisB*
by *blast*
have $v \cdot_C x = (\sum a{\in}t.\ s\ a *_C a) \cdot_C x$
by (*simp add*: *b1*)
also have $\ldots = (\sum a{\in}t.\ (s\ a *_C a) \cdot_C x)$
using *cinner-sum-left* **by** *blast*
also have $\ldots = (\sum a{\in}t.\ cnj\ (s\ a) * (a \cdot_C x))$
by *auto*
also have $\ldots = 0$
using *b3 r1 subsetD* **by** *force*
finally show *?thesis* **by** *simp*
qed
thus *?thesis*
by (*simp add*: ‹$\bigwedge v.\ (v \cdot_C x) = 0$› *cinner-extensionality*)
qed
ultimately show *?thesis* **by** *simp*
qed
thus *?thesis*
using *basisA-def cblinfun-eq-0-on-canonical-basis* **by** *auto*
qed

lemma *cinner-canonical-basis-eq*:
defines *basisA* == *set* (*canonical-basis*::′*a*::*onb-enum list*)
and *basisB* == *set* (*canonical-basis*::′*b*::*onb-enum list*)
assumes $\bigwedge u\ v.\ u{\in}basisA \Longrightarrow v{\in}basisB \Longrightarrow v \cdot_C (F *_V u) = v \cdot_C (G *_V u)$
shows $F = G$
proof−
define *H* **where** $H = F - G$
have $\bigwedge u\ v.\ u{\in}basisA \Longrightarrow v{\in}basisB \Longrightarrow$ *is-orthogonal v* $(H *_V u)$
unfolding *H-def*
by (*simp add*: *assms*(*3*) *cinner-diff-right minus-cblinfun.rep-eq*)
hence $H = 0$
by (*simp add*: *basisA-def basisB-def cinner-canonical-basis-eq-0*)
thus *?thesis* **unfolding** *H-def* **by** *simp*
qed

lemma *cinner-canonical-basis-eq*′:
defines *basisA* == *set* (*canonical-basis*::′*a*::*onb-enum list*)
and *basisB* == *set* (*canonical-basis*::′*b*::*onb-enum list*)
assumes $\bigwedge u\ v.\ u{\in}basisA \Longrightarrow v{\in}basisB \Longrightarrow (F *_V u) \cdot_C v = (G *_V u) \cdot_C v$
shows $F = G$
using *cinner-canonical-basis-eq assms*
by (*metis cinner-commute*′)

lemma *not-not-singleton-cblinfun-zero*:
‹$x = 0$› **if** ‹¬ *class.not-singleton TYPE*(′*a*)› **for** x :: ‹′*a*::*complex-normed-vector* ⇒$_{CL}$ ′*b*::*complex-normed-vector*›

apply (*rule cblinfun-eqI*)
apply (*subst not-not-singleton-zero*[*OF that*])
by *simp*

lemma *cblinfun-norm-approx-witness*:
fixes $A :: \langle 'a{::}\{not\text{-}singleton, complex\text{-}normed\text{-}vector\} \Rightarrow_{CL} 'b{::}complex\text{-}normed\text{-}vector\rangle$
assumes $\langle \varepsilon > 0\rangle$
shows $\langle \exists \psi.\ norm\ (A *_V \psi) \geq norm\ A - \varepsilon \wedge norm\ \psi = 1\rangle$
proof (*transfer fixing*: ε)
fix $A :: \langle 'a \Rightarrow 'b\rangle$ **assume** [*simp*]: $\langle bounded\text{-}clinear\ A\rangle$
have $\langle \exists y{\in}\{norm\ (A\ x)\ |x.\ norm\ x = 1\}.\ y > \bigsqcup \{norm\ (A\ x)\ |x.\ norm\ x = 1\} - \varepsilon\rangle$
apply (*rule Sup-real-close*)
using *assms* **by** (*auto simp*: *ex-norm1 bounded-clinear.bounded-linear bdd-above-norm-f*)
also have $\langle \bigsqcup \{norm\ (A\ x)\ |x.\ norm\ x = 1\} = onorm\ A\rangle$
by (*simp add*: *bounded-clinear.bounded-linear onorm-sphere*)
finally
show $\langle \exists \psi.\ onorm\ A - \varepsilon \leq norm\ (A\ \psi) \wedge norm\ \psi = 1\rangle$
by *force*
qed

lemma *cblinfun-norm-approx-witness-mult*:
fixes $A :: \langle 'a{::}\{not\text{-}singleton, complex\text{-}normed\text{-}vector\} \Rightarrow_{CL} 'b{::}complex\text{-}normed\text{-}vector\rangle$
assumes $\langle \varepsilon < 1\rangle$
shows $\langle \exists \psi.\ \ norm\ (A *_V \psi) \geq norm\ A * \varepsilon \wedge norm\ \psi = 1\rangle$
proof (*cases* $\langle norm\ A = 0\rangle$)
case *True*
then show *?thesis*
by *auto* (*simp add*: *ex-norm1*)
next
case *False*
then have $\langle (1 - \varepsilon) * norm\ A > 0\rangle$
using *assms* **by** *fastforce*
then obtain ψ **where** *geq*: $\langle norm\ (A *_V \psi) \geq norm\ A - ((1 - \varepsilon) * norm\ A)\rangle$
and $\langle norm\ \psi = 1\rangle$
using *cblinfun-norm-approx-witness* **by** *blast*
have $\langle norm\ A * \varepsilon = norm\ A - (1 - \varepsilon) * norm\ A\rangle$
by (*simp add*: *mult.commute right-diff-distrib'*)
also have $\langle \ldots \leq norm\ (A *_V \psi)\rangle$
by (*rule geq*)
finally show *?thesis*
using $\langle norm\ \psi = 1\rangle$ **by** *auto*
qed

lemma *cblinfun-norm-approx-witness'*:
fixes $A :: \langle 'a{::}complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'b{::}complex\text{-}normed\text{-}vector\rangle$
assumes $\langle \varepsilon > 0\rangle$
shows $\langle \exists \psi.\ norm\ (A *_V \psi) / norm\ \psi \geq norm\ A - \varepsilon\rangle$

proof (*cases* ‹*class.not-singleton TYPE*('*a*)›)
case *True*
obtain ψ **where** ‹*norm* (*A* $*_V$ ψ) $\geq$ *norm A* $-$ ε› **and** ‹*norm* ψ = *1*›
apply *atomize-elim*
using *complex-normed-vector-axioms True assms*
by (*rule cblinfun-norm-approx-witness*[*internalize-sort*' '*a*])
then have ‹*norm* (*A* $*_V$ ψ) / *norm* ψ $\geq$ *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 *cblinfun-to-CARD-1-0*[*simp*]: ‹(*A* :: - $\Rightarrow_{CL}$ -::*CARD-1*) = *0*›
by (*simp add*: *cblinfun-eqI*)

lemma *cblinfun-from-CARD-1-0*[*simp*]: ‹(*A* :: -::*CARD-1* $\Rightarrow_{CL}$ -) = *0*›
using *cblinfun-eq-on-UNIV-span* **by** *force*

lemma *cblinfun-cspan-UNIV*:
fixes *basis* :: ‹('*a*::{*complex-normed-vector*,*cfinite-dim*} $\Rightarrow_{CL}$ '*b*::*complex-normed-vector*) *set*›
and *basisA* :: ‹'*a set*› **and** *basisB* :: ‹'*b set*›
assumes ‹*cspan basisA* = *UNIV*› **and** ‹*cspan basisB* = *UNIV*›
assumes *basis*: ‹$\bigwedge$*a b*. *a*$\in$*basisA* $\Longrightarrow$ *b*$\in$*basisB* $\Longrightarrow$ $\exists$ *F*$\in$*basis*. $\forall$ *a'*$\in$*basisA*. *F* $*_V$ *a'* = (*if a'*=*a then b else 0*)›
shows ‹*cspan basis* = *UNIV*›
proof $-$
obtain *basisA'* **where** ‹*basisA'* $\subseteq$ *basisA*› **and** ‹*cindependent basisA'*› **and** ‹*cspan basisA'* = *UNIV*›
by (*metis assms*(*1*) *complex-vector.maximal-independent-subset complex-vector.span-eq top-greatest*)
then have [*simp*]: ‹*finite basisA'*›
by (*simp add*: *cindependent-cfinite-dim-finite*)
have *basis'*: ‹$\bigwedge$*a b*. *a*$\in$*basisA'* $\Longrightarrow$ *b*$\in$*basisB* $\Longrightarrow$ $\exists$ *F*$\in$*basis*. $\forall$ *a'*$\in$*basisA'*. *F* $*_V$ *a'* = (*if a'*=*a then b else 0*)›
using *basis* ‹*basisA'* $\subseteq$ *basisA*› **by** *fastforce*

obtain *F* **where** *F*: ‹*F a b* $\in$ *basis* $\wedge$ *F a b* $*_V$ *a'* = (*if a'*=*a then b else 0*)›
if ‹*a*$\in$*basisA'*› ‹*b*$\in$*basisB*› ‹*a'*$\in$*basisA'*› **for** *a b a'*
apply *atomize-elim* **apply** (*intro choice allI*)
using *basis'* **by** *metis*

then have *F-apply*: ‹$F\ a\ b *_V a' = (\mathit{if}\ a'{=}a\ \mathit{then}\ b\ \mathit{else}\ 0)$›
if ‹$a{\in}\mathit{basisA}'$› ‹$b{\in}\mathit{basisB}$› ‹$a'{\in}\mathit{basisA}'$› **for** $a\ b\ a'$
using *that* **by** *auto*
have *F-basis*: ‹$F\ a\ b \in \mathit{basis}$›
if ‹$a{\in}\mathit{basisA}'$› ‹$b{\in}\mathit{basisB}$› **for** $a\ b$
using *that F* **by** *auto*
have *b-span*: ‹$\exists\, G{\in}\mathit{cspan}\ \{F\ a\ b|b.\ b{\in}\mathit{basisB}\}.\ \forall\, a'{\in}\mathit{basisA}'.\ G *_V a' = (\mathit{if}\ a'{=}a\ \mathit{then}\ b\ \mathit{else}\ 0)$› **if** ‹$a{\in}\mathit{basisA}'$› **for** $a\ b$
proof −
from ‹$\mathit{cspan}\ \mathit{basisB} = \mathit{UNIV}$›
obtain $r\ t$ **where** ‹$\mathit{finite}\ t$› **and** ‹$t \subseteq \mathit{basisB}$› **and** *b-lincom*: ‹$b = (\sum a{\in}t.\ r\ a *_C a)$›
unfolding *complex-vector.span-alt* **by** *atomize-elim blast*
define G **where** ‹$G = (\sum i{\in}t.\ r\ i *_C F\ a\ i)$›
have ‹$G \in \mathit{cspan}\ \{F\ a\ b|b.\ b{\in}\mathit{basisB}\}$›
using ‹$\mathit{finite}\ t$› ‹$t \subseteq \mathit{basisB}$› **unfolding** *G-def*
by (*smt* (*verit*) *complex-vector.span-scale complex-vector.span-sum complex-vector.span-superset mem-Collect-eq subsetD*)
moreover have ‹$G *_V a' = (\mathit{if}\ a'{=}a\ \mathit{then}\ b\ \mathit{else}\ 0)$› **if** ‹$a'{\in}\mathit{basisA}'$› **for** a'
using ‹$t \subseteq \mathit{basisB}$› ‹$a{\in}\mathit{basisA}'$› ‹$a'{\in}\mathit{basisA}'$›
by (*auto simp*: *b-lincom G-def cblinfun.sum-left F-apply intro*!: *sum.neutral sum.cong*)
ultimately show *?thesis*
by *blast*
qed

have *a-span*: ‹$\mathit{cspan}\ (\bigcup a{\in}\mathit{basisA}'.\ \mathit{cspan}\ \{F\ a\ b|b.\ b{\in}\mathit{basisB}\}) = \mathit{UNIV}$›
proof (*intro equalityI subset-UNIV subsetI*, *rename-tac H*)
fix H
obtain G **where** G: ‹$G\ a\ b \in \mathit{cspan}\ \{F\ a\ b|b.\ b{\in}\mathit{basisB}\} \wedge G\ a\ b *_V a' = (\mathit{if}\ a'{=}a\ \mathit{then}\ b\ \mathit{else}\ 0)$›
if ‹$a{\in}\mathit{basisA}'$› **and** ‹$a'{\in}\mathit{basisA}'$› **for** $a\ b\ a'$
apply *atomize-elim* **apply** (*intro choice allI*)
using *b-span* **by** *blast*
then have *G-cspan*: ‹$G\ a\ b \in \mathit{cspan}\ \{F\ a\ b|b.\ b{\in}\mathit{basisB}\}$› **if** ‹$a{\in}\mathit{basisA}'$› **for** $a\ b$
using *that* **by** *auto*
from G **have** G: ‹$G\ a\ b *_V a' = (\mathit{if}\ a'{=}a\ \mathit{then}\ b\ \mathit{else}\ 0)$› **if** ‹$a{\in}\mathit{basisA}'$› **and** ‹$a'{\in}\mathit{basisA}'$› **for** $a\ b\ a'$
using *that* **by** *auto*
define H' **where** ‹$H' = (\sum a{\in}\mathit{basisA}'.\ G\ a\ (H *_V a))$›
have ‹$H' \in \mathit{cspan}\ (\bigcup a{\in}\mathit{basisA}'.\ \mathit{cspan}\ \{F\ a\ b|b.\ b{\in}\mathit{basisB}\})$›
unfolding *H'-def* **using** *G-cspan*
by (*smt* (*verit*, *del-insts*) *UN-iff complex-vector.span-clauses*(*1*) *complex-vector.span-sum*)
moreover have ‹$H' = H$›
using ‹$\mathit{cspan}\ \mathit{basisA}' = \mathit{UNIV}$›
by (*auto simp*: *H'-def cblinfun-eq-on-UNIV-span cblinfun.sum-left G*)
ultimately show ‹$H \in \mathit{cspan}\ (\bigcup a{\in}\mathit{basisA}'.\ \mathit{cspan}\ \{F\ a\ b\ |b.\ b \in \mathit{basisB}\})$›
by *simp*

qed

moreover have ‹*cspan basis* ⊇ *cspan* (⋃ *a*∈*basisA'*. *cspan* {*F a b*|*b*. *b*∈*basisB*})›
by (*smt* (*verit*) *F-basis UN-subset-iff complex-vector.span-base complex-vector.span-minimal complex-vector.subspace-span mem-Collect-eq subsetI*)

ultimately show ‹*cspan basis* = *UNIV*›
by *auto*
qed

instance *cblinfun* :: (‹{*cfinite-dim*,*complex-normed-vector*}›, ‹{*cfinite-dim*,*complex-normed-vector*}›) *cfinite-dim*
proof *intro-classes*
obtain *basisA* :: ‹'*a set*› **where** [*simp*]: ‹*cspan basisA* = *UNIV*› ‹*cindependent basisA*› ‹*finite basisA*›
using *finite-basis* **by** *blast*
obtain *basisB* :: ‹'*b set*› **where** [*simp*]: ‹*cspan basisB* = *UNIV*› ‹*cindependent basisB*› ‹*finite basisB*›
using *finite-basis* **by** *blast*
define *f* **where** ‹*f a b* = *cconstruct basisA* (λ*x*. *if x*=*a then b else 0*)› **for** *a* :: '*a* **and** *b* :: '*b*
have *f-a*: ‹*f a b a* = *b*› **if** ‹*a* : *basisA*› **for** *a b*
by (*simp add*: *complex-vector.construct-basis f-def that*)
have *f-not-a*: ‹*f a b c* = *0*› **if** ‹*a* : *basisA*› **and** ‹*c* : *basisA*› **and** ‹*a* ≠ *c*›**for** *a b c*
using *that* **by** (*simp add*: *complex-vector.construct-basis f-def*)
define *F* **where** ‹*F a b* = *CBlinfun* (*f a b*)› **for** *a b*
have ‹*clinear* (*f a b*)› **for** *a b*
by (*auto intro*: *complex-vector.linear-construct simp*: *f-def*)
then have ‹*bounded-clinear* (*f a b*)› **for** *a b*
by *auto*
then have *F-apply*: ‹*cblinfun-apply* (*F a b*) = *f a b*› **for** *a b*
by (*simp add*: *F-def bounded-clinear-CBlinfun-apply*)
define *basis* **where** ‹*basis* = {*F a b*| *a b*. *a*∈*basisA* ∧ *b*∈*basisB*}›
have ⋀*a b*. ⟦*a* ∈ *basisA*; *b* ∈ *basisB*⟧ ⟹ ∃ *F*∈*basis*. ∀ *a'*∈*basisA*. *F* ∗$_V$ *a'* = (*if a'* = *a then b else 0*)
by (*smt* (*verit*, *del-insts*) *F-apply basis-def f-a f-not-a mem-Collect-eq*)
then have ‹*cspan basis* = *UNIV*›
by (*metis* ‹*cspan basisA* = *UNIV*› ‹*cspan basisB* = *UNIV*› *cblinfun-cspan-UNIV*)

moreover have ‹*finite basis*›
unfolding *basis-def* **by** (*auto intro*: *finite-image-set2*)
ultimately show ‹∃ *S* :: ('*a* ⇒$_{CL}$ '*b*) *set*. *finite S* ∧ *cspan S* = *UNIV*›
by *auto*
qed

lemma *norm-cblinfun-bound-dense*:
assumes ‹*0* ≤ *b*›
assumes *S*: ‹*closure S* = *UNIV*›

assumes *bound*: ‹$\bigwedge x.\ x{\in}S \Longrightarrow$ *norm* (*cblinfun-apply f x*) $\leq b *$ *norm x*›
shows ‹*norm f* $\leq b$›
proof −
have *1*: ‹*continuous-on UNIV* ($\lambda a.$ *norm* (*f* $*_V$ *a*))›
by (*simp add*: *continuous-on-eq-continuous-within*)
have *2*: ‹*continuous-on UNIV* ($\lambda a.$ *b* $*$ *norm a*)›
using *continuous-on-mult-left continuous-on-norm-id* **by** *blast*
have ‹*norm* (*cblinfun-apply f x*) $\leq b *$ *norm x*› **for** *x*
by (*metis* (*mono-tags*, *lifting*) *UNIV-I S bound 1 2 on-closure-leI*)
then show ‹*norm f* $\leq b$›
using ‹*0* $\leq b$› *norm-cblinfun-bound* **by** *blast*
qed

lemma *infsum-cblinfun-apply*:
assumes ‹*g summable-on S*›
shows ‹*infsum* ($\lambda x.$ *A* $*_V$ *g x*) *S* = *A* $*_V$ (*infsum g S*)›
using *infsum-bounded-linear*[*unfolded o-def*] *assms cblinfun.real.bounded-linear-right*
by *blast*

lemma *has-sum-cblinfun-apply*:
assumes ‹(*g has-sum x*) *S*›
shows ‹(($\lambda x.$ *A* $*_V$ *g x*) *has-sum* (*A* $*_V$ *x*)) *S*›
using *assms has-sum-bounded-linear*[*unfolded o-def*] **using** *cblinfun.real.bounded-linear-right*
by *blast*

lemma *abs-summable-on-cblinfun-apply*:
assumes ‹*g abs-summable-on S*›
shows ‹($\lambda x.$ *A* $*_V$ *g x*) *abs-summable-on S*›
using *bounded-clinear.bounded-linear*[*OF cblinfun.bounded-clinear-right*] *assms*
by (*rule abs-summable-on-bounded-linear*[*unfolded o-def*])

lemma *summable-on-cblinfun-apply*:
assumes ‹*g summable-on S*›
shows ‹($\lambda x.$ *A* $*_V$ *g x*) *summable-on S*›
using *bounded-clinear.bounded-linear*[*OF cblinfun.bounded-clinear-right*] *assms*
by (*rule summable-on-bounded-linear*[*unfolded o-def*])

lemma *summable-on-cblinfun-apply-left*:
assumes ‹*A summable-on S*›
shows ‹($\lambda x.$ *A x* $*_V$ *g*) *summable-on S*›
using *bounded-clinear.bounded-linear*[*OF cblinfun.bounded-clinear-left*] *assms*
by (*rule summable-on-bounded-linear*[*unfolded o-def*])

lemma *abs-summable-on-cblinfun-apply-left*:
assumes ‹*A abs-summable-on S*›
shows ‹($\lambda x.$ *A x* $*_V$ *g*) *abs-summable-on S*›
using *bounded-clinear.bounded-linear*[*OF cblinfun.bounded-clinear-left*] *assms*
by (*rule abs-summable-on-bounded-linear*[*unfolded o-def*])
lemma *infsum-cblinfun-apply-left*:

assumes ‹*A summable-on S*›
shows ‹*infsum* (λ*x*. *A x* ∗$_V$ *g*) *S* = (*infsum A S*) ∗$_V$ *g*›
apply (*rule infsum-bounded-linear*[*unfolded o-def*, *of* ‹λ*A*. *cblinfun-apply A g*›])
using *assms*
by (*auto simp add*: *bounded-clinear.bounded-linear bounded-cbilinear-cblinfun-apply*)
lemma *has-sum-cblinfun-apply-left*:
assumes ‹(*A has-sum x*) *S*›
shows ‹((λ*x*. *A x* ∗$_V$ *g*) *has-sum* (*x* ∗$_V$ *g*)) *S*›
apply (*rule has-sum-bounded-linear*[*unfolded o-def*, *of* ‹λ*A*. *cblinfun-apply A g*›])
using *assms* **by** (*auto simp add*: *bounded-clinear.bounded-linear cblinfun.bounded-clinear-left*)

The next eight lemmas logically belong in *Complex-Bounded-Operators.Complex-Inner-Product* but the proofs use facts from this theory.

lemma *has-sum-cinner-left*:
assumes ‹(*f has-sum x*) *I*›
shows ‹((λ*i*. *cinner a* (*f i*)) *has-sum cinner a x*) *I*›
by (*metis assms cblinfun-cinner-right.rep-eq has-sum-cblinfun-apply*)

lemma *summable-on-cinner-left*:
assumes ‹*f summable-on I*›
shows ‹(λ*i*. *cinner a* (*f i*)) *summable-on I*›
by (*metis assms has-sum-cinner-left summable-on-def*)

lemma *infsum-cinner-left*:
assumes ‹φ *summable-on I*›
shows ‹*cinner* ψ (∑$_\infty$*i*∈*I*. φ *i*) = (∑$_\infty$*i*∈*I*. *cinner* ψ (φ *i*))›
by (*metis assms has-sum-cinner-left has-sum-infsum infsumI*)

lemma *has-sum-cinner-right*:
assumes ‹(*f has-sum x*) *I*›
shows ‹((λ*i*. *f i* ·$_C$ *a*) *has-sum* (*x* ·$_C$ *a*)) *I*›
using *assms has-sum-bounded-linear*[*unfolded o-def*] *bounded-antilinear.bounded-linear*

bounded-antilinear-cinner-left **by** *blast*

lemma *summable-on-cinner-right*:
assumes ‹*f summable-on I*›
shows ‹(λ*i*. *f i* ·$_C$ *a*) *summable-on I*›
by (*metis assms has-sum-cinner-right summable-on-def*)

lemma *infsum-cinner-right*:
assumes ‹φ *summable-on I*›
shows ‹(∑$_\infty$*i*∈*I*. φ *i*) ·$_C$ ψ = (∑$_\infty$*i*∈*I*. φ *i* ·$_C$ ψ)›
by (*metis assms has-sum-cinner-right has-sum-infsum infsumI*)

lemma *Cauchy-cinner-product-summable*:
assumes *asum*: ‹*a summable-on UNIV*›
assumes *bsum*: ‹*b summable-on UNIV*›

assumes ‹*finite X*› ‹*finite Y*›
assumes *pos*: ‹$\bigwedge x\ y.\ x \notin X \Longrightarrow y \notin Y \Longrightarrow$ *cinner* (*a x*) (*b y*) ≥ 0›
shows *absum*: ‹($\lambda(x,\ y).$ *cinner* (*a x*) (*b y*)) *summable-on UNIV*›
proof −
have ‹($\sum(x,y)\in F.$ *norm* (*cinner* (*a x*) (*b y*))) $\leq$ *norm* (*cinner* (*infsum a* (−*X*)) (*infsum b* (−*Y*))) + *norm* (*infsum a* (−*X*)) + *norm* (*infsum b* (−*Y*)) + *1*›
if ‹*finite F*› **and** ‹*F* $\subseteq$ (−*X*)×(−*Y*)› **for** *F*
proof −
from *asum* ‹*finite X*›
have ‹*a summable-on* (−*X*)›
by (*simp add*: *Compl-eq-Diff-UNIV summable-on-cofin-subset*)
then obtain *MA* **where** ‹*finite MA*› **and** ‹*MA* $\subseteq$ −*X*›
and *MA*: ‹*G* $\supseteq$ *MA* $\Longrightarrow$ *G* $\subseteq$ −*X* $\Longrightarrow$ *finite G* $\Longrightarrow$ *norm* (*sum a G* − *infsum a* (−*X*)) $\leq$ *1*› **for** *G*
apply (*simp add*: *summable-iff-has-sum-infsum has-sum-metric dist-norm*)
by (*meson less-eq-real-def zero-less-one*)

from *bsum* ‹*finite Y*›
have ‹*b summable-on* (−*Y*)›
by (*simp add*: *Compl-eq-Diff-UNIV summable-on-cofin-subset*)
then obtain *MB* **where** ‹*finite MB*› **and** ‹*MB* $\subseteq$ −*Y*›
and *MB*: ‹*G* $\supseteq$ *MB* $\Longrightarrow$ *G* $\subseteq$ −*Y* $\Longrightarrow$ *finite G* $\Longrightarrow$ *norm* (*sum b G* − *infsum b* (−*Y*)) $\leq$ *1*› **for** *G*
apply (*simp add*: *summable-iff-has-sum-infsum has-sum-metric dist-norm*)
by (*meson less-eq-real-def zero-less-one*)

define *F1 F2* **where** ‹*F1* = *fst* ‘ *F* ∪ *MA*› **and** ‹*F2* = *snd* ‘ *F* ∪ *MB*›
define *t1 t2* **where** ‹*t1* = *sum a F1* − *infsum a* (−*X*)› **and** ‹*t2* = *sum b F2* − *infsum b* (−*Y*)›

have [*simp*]: ‹*finite F1*› ‹*finite F2*›
using *F1-def F2-def* ‹*finite MA*› ‹*finite MB*› *that* **by** *auto*
have [*simp*]: ‹*F1* $\subseteq$ −*X*› ‹*F2* $\subseteq$ −*Y*›
using ‹*F* $\subseteq$ (−*X*)×(−*Y*)› ‹*MA* $\subseteq$ −*X*› ‹*MB* $\subseteq$ −*Y*›
by (*auto simp*: *F1-def F2-def*)
from *MA*[*OF* - ‹*F1* $\subseteq$ −*X*› ‹*finite F1*›] **have** ‹*norm t1* $\leq$ *1*›
by (*auto simp*: *t1-def F1-def*)
from *MB*[*OF* - ‹*F2* $\subseteq$ −*Y*› ‹*finite F2*›] **have** ‹*norm t2* $\leq$ *1*›
by (*auto simp*: *t2-def F2-def*)
have [*simp*]: ‹*F* $\subseteq$ *F1* × *F2*›
by (*force simp*: *F1-def F2-def image-def*)
have ‹($\sum(x,y)\in F.$ *norm* (*cinner* (*a x*) (*b y*))) $\leq$ ($\sum(x,y)\in F1\times F2.$ *norm* (*cinner* (*a x*) (*b y*)))›
by (*intro sum-mono2*) *auto*
also have ... = ($\sum x\in F1 \times F2.$ *norm* (*a* (*fst x*) $\cdot_C$ *b* (*snd x*)))
by (*auto simp*: *case-prod-beta*)
also have ... = *norm* ($\sum x\in F1 \times F2.$ *a* (*fst x*) $\cdot_C$ *b* (*snd x*))
proof −
have ($\sum x\in F1 \times F2.$ |*a* (*fst x*) $\cdot_C$ *b* (*snd x*)|) = |$\sum x\in F1 \times F2.$ *a* (*fst x*) $\cdot_C$

b (*snd x*)|
by (*smt* (*verit*, *best*) *pos sum.cong sum-nonneg ComplD SigmaE* ‹*F1* ⊆ − *X*› ‹*F2* ⊆ − *Y*› *abs-pos prod.sel subset-iff*)
then show *?thesis*
by (*smt* (*verit*) *abs-complex-def norm-ge-zero norm-of-real o-def of-real-sum sum.cong sum-norm-le*)
qed
also from *pos* **have** ‹... = *norm* (∑(*x*,*y*)∈*F1*×*F2*. *cinner* (*a x*) (*b y*))›
by (*auto simp*: *case-prod-beta*)
also have ‹... = *norm* (*cinner* (*sum a F1*) (*sum b F2*))›
by (*simp add*: *sum.cartesian-product sum-cinner*)
also have ‹... = *norm* (*cinner* (*infsum a* (−*X*) + *t1*) (*infsum b* (−*Y*) + *t2*))›
by (*simp add*: *t1-def t2-def*)
also have ‹... ≤ *norm* (*cinner* (*infsum a* (−*X*)) (*infsum b* (−*Y*))) + *norm* (*infsum a* (−*X*)) ∗ *norm t2* + *norm t1* ∗ *norm* (*infsum b* (−*Y*)) + *norm t1* ∗ *norm t2*›
apply (*simp add*: *cinner-add-right cinner-add-left*)
by (*smt* (*verit*, *del-insts*) *complex-inner-class.Cauchy-Schwarz-ineq2 norm-triangle-ineq*)
also from ‹*norm t1* ≤ *1*› ‹*norm t2* ≤ *1*›
have ‹... ≤ *norm* (*cinner* (*infsum a* (−*X*)) (*infsum b* (−*Y*))) + *norm* (*infsum a* (−*X*)) + *norm* (*infsum b* (−*Y*)) + *1*›
by (*auto intro*!: *add-mono mult-left-le mult-left-le-one-le mult-le-one*)
finally show *?thesis*
by −
qed

then have ‹(λ(*x*, *y*). *cinner* (*a x*) (*b y*)) *abs-summable-on* (−*X*)×(−*Y*)›
apply (*rule-tac nonneg-bdd-above-summable-on*)
by (*auto intro*!: *bdd-aboveI2 simp*: *case-prod-unfold*)
then have *1*: ‹(λ(*x*, *y*). *cinner* (*a x*) (*b y*)) *summable-on* (−*X*)×(−*Y*)›
using *abs-summable-summable* **by** *blast*

from *bsum*
have ‹(λ*y*. *b y*) *summable-on* (−*Y*)›
by (*simp add*: *Compl-eq-Diff-UNIV assms*(*4*) *summable-on-cofin-subset*)
then have ‹(λ*y*. *cinner* (*a x*) (*b y*)) *summable-on* (−*Y*)› **for** *x*
using *summable-on-cinner-left* **by** *blast*
with ‹*finite X*› **have** *2*: ‹(λ(*x*, *y*). *cinner* (*a x*) (*b y*)) *summable-on X*×(−*Y*)›
by (*force intro*: *summable-on-product-finite-left*)

from *asum*
have ‹(λ*x*. *a x*) *summable-on* (−*X*)›
by (*simp add*: *Compl-eq-Diff-UNIV assms*(*3*) *summable-on-cofin-subset*)
then have ‹(λ*x*. *cinner* (*a x*) (*b y*)) *summable-on* (−*X*)› **for** *y*
using *summable-on-cinner-right* **by** *blast*
with ‹*finite Y*› **have** *3*: ‹(λ(*x*, *y*). *cinner* (*a x*) (*b y*)) *summable-on* (−*X*)×*Y*›
by (*force intro*: *summable-on-product-finite-right*)

have *4*: ‹(λ(*x*, *y*). *cinner* (*a x*) (*b y*)) *summable-on X*×*Y*›

by (*simp add*: ‹*finite X*› ‹*finite Y*›)

have §: *UNIV* = ((−*X*) × −*Y*) ∪ (*X* × −*Y*) ∪ ((−*X*) × *Y*) ∪ (*X* × *Y*)
by *auto*
show *?thesis*
using *1 2 3 4* **by** (*force simp*: § *intro*!: *summable-on-Un-disjoint*)
qed

A variant of *Series.Cauchy-product-sums* with (∗) replaced by ($\cdot_C$). Differently from *Series.Cauchy-product-sums*, we do not require absolute summability of *a* and *b* individually but only unconditional summability of *a*, *b*, and their product. While on, e.g., reals, unconditional summability is equivalent to absolute summability, in general unconditional summability is a weaker requirement.

Logically belong in *Complex-Bounded-Operators.Complex-Inner-Product* but the proof uses facts from this theory.

lemma
fixes *a b* :: *nat* ⇒ *'a*::*complex-inner*
assumes *asum*: ‹*a summable-on UNIV*›
assumes *bsum*: ‹*b summable-on UNIV*›
assumes *absum*: ‹(λ(*x*, *y*). *cinner* (*a x*) (*b y*)) *summable-on UNIV*›

shows *Cauchy-cinner-product-infsum*: ‹(∑∞*k*. ∑*i*≤*k*. *cinner* (*a i*) (*b* (*k* − *i*))) = *cinner* (∑∞*k*. *a k*) (∑∞*k*. *b k*)›
and *Cauchy-cinner-product-infsum-exists*: ‹(λ*k*. ∑*i*≤*k*. *cinner* (*a i*) (*b* (*k* − *i*))) *summable-on UNIV*›
proof −
have *img*: ‹(λ(*k*::*nat*, *i*). (*i*, *k* − *i*)) ‘ {(*k*, *i*). *i* ≤ *k*} = *UNIV*›
apply (*auto simp*: *image-def*)
by (*metis add.commute add-diff-cancel-right' diff-le-self*)
have *inj*: ‹*inj-on* (λ(*k*::*nat*, *i*). (*i*, *k* − *i*)) {(*k*, *i*). *i* ≤ *k*}›
by (*smt* (*verit*, *del-insts*) *Pair-inject case-prodE case-prod-conv eq-diff-iff inj-onI mem-Collect-eq*)
have *sigma*: ‹(*SIGMA k*:*UNIV*. {*i*. *i* ≤ *k*}) = {(*k*, *i*). *i* ≤ *k*}›
by *auto*

from *absum*
have §: ‹(λ(*x*, *y*). *cinner* (*a y*) (*b* (*x* − *y*))) *summable-on* {(*k*, *i*). *i* ≤ *k*}›
by (*rule Cauchy-cinner-product-summable'*[*THEN iffD1*])
then have ‹(λ*k*. ∑∞*i*|*i*≤*k*. *cinner* (*a i*) (*b* (*k*−*i*))) *summable-on UNIV*›
by (*metis* (*mono-tags*, *lifting*) *sigma summable-on-Sigma-banach summable-on-cong*)
then show ‹(λ*k*. ∑*i*≤*k*. *cinner* (*a i*) (*b* (*k* − *i*))) *summable-on UNIV*›
by (*metis* (*mono-tags*, *lifting*) *atMost-def finite-Collect-le-nat infsum-finite summable-on-cong*)

have ‹*cinner* (∑∞*k*. *a k*) (∑∞*k*. *b k*) = (∑∞*k*. ∑∞*l*. *cinner* (*a k*) (*b l*))›
by (*smt* (*verit*, *best*) *asum bsum infsum-cinner-left infsum-cinner-right infsum-cong*)

also have ‹... = ($\sum_\infty$(k,l). *cinner* (*a k*) (*b l*))›
by (*smt* (*verit*) *UNIV-Times-UNIV infsum-Sigma'-banach infsum-cong local.absum*)
also have ‹... = ($\sum_\infty$(k, l)∈(λ(k, i). (i, k − i)) ‘ {(k, i). i ≤ k}. *cinner* (*a k*) (*b l*))›
by (*simp only*: *img*)
also have ‹... = *infsum* ((λ(k, l). *a k* ·$_C$ *b l*) ∘ (λ(k, i). (i, k − i))) {(k, i). i ≤ k}›
using *inj* **by** (*rule infsum-reindex*)
also have ‹... = ($\sum_\infty$(k,i)|i≤k. *a i* ·$_C$ *b* (k−i))›
by (*simp add*: *o-def case-prod-unfold*)
also have ‹... = ($\sum_\infty$k. $\sum_\infty$i|i≤k. *a i* ·$_C$ *b* (k−i))›
by (*metis* (*no-types*) § *infsum-Sigma'-banach sigma*)
also have ‹... = ($\sum_\infty$k. $\sum$i≤k. *a i* ·$_C$ *b* (k−i))›
by (*simp add*: *atMost-def*)
finally show ‹($\sum_\infty$k. $\sum$i≤k. *a i* ·$_C$ *b* (k − i)) = ($\sum_\infty$k. *a k*) ·$_C$ ($\sum_\infty$k. *b k*)›
by *simp*
qed

lemma *CBlinfun-plus*:
assumes [*simp*]: ‹*bounded-clinear f*› ‹*bounded-clinear g*›
shows ‹*CBlinfun* (*f* + *g*) = *CBlinfun f* + *CBlinfun g*›
by (*auto intro*!: *cblinfun-eqI simp*: *plus-fun-def bounded-clinear-add CBlinfun-inverse cblinfun.add-left*)

lemma *CBlinfun-scaleC*:
assumes ‹*bounded-clinear f*›
shows ‹*CBlinfun* (λy. *c* *$_C$ *f y*) = *c* *$_C$ *CBlinfun f*›
by (*simp add*: *assms eq-onp-same-args scaleC-cblinfun.abs-eq*)

lemma *cinner-sup-norm-cblinfun*:
fixes *A* :: ‹'*a*::{*complex-normed-vector*,*not-singleton*} ⇒$_{CL}$ '*b*::*complex-inner*›
shows ‹*norm A* = (*SUP* (ψ,φ). *cmod* (*cinner* ψ (*A* *$_V$ φ)) / (*norm* ψ * *norm* φ))›
apply *transfer*
apply (*rule cinner-sup-onorm*)
by (*simp add*: *bounded-clinear.bounded-linear*)

lemma *norm-cblinfun-Sup*: ‹*norm A* = (*SUP* ψ. *norm* (*A* *$_V$ ψ) / *norm* ψ)›
by (*simp add*: *norm-cblinfun.rep-eq onorm-def*)

lemma *cblinfun-eq-on*:
fixes *A B* :: '*a*::*cbanach* ⇒$_{CL}$'*b*::*complex-normed-vector*
assumes ⋀x. *x* ∈ *G* ⟹ *A* *$_V$ *x* = *B* *$_V$ *x* **and** ‹*t* ∈ *closure* (*cspan G*)›
shows *A* *$_V$ *t* = *B* *$_V$ *t*
using *assms*

apply *transfer*
using *bounded-clinear-eq-on-closure* **by** *blast*

lemma *cblinfun-eq-gen-eqI*:
fixes A B :: $'a$::*cbanach* $\Rightarrow_{CL}$ $'b$::*complex-normed-vector*
assumes $\bigwedge x.\ x \in G \Longrightarrow A *_V x = B *_V x$ **and** ‹*ccspan* $G = \top$›
shows $A = B$
by (*metis assms cblinfun-eqI cblinfun-eq-on ccspan.rep-eq iso-tuple-UNIV-I top-ccsubspace.rep-eq*)

declare *cnj-bounded-antilinear*[*bounded-antilinear*]

lemma *Cblinfun-comp-bounded-cbilinear*: ‹*bounded-clinear* (*CBlinfun o p*)› **if** ‹*bounded-cbilinear p*›
by (*metis bounded-cbilinear.bounded-clinear-prod-right bounded-cbilinear.prod-right-def comp-id map-fun-def that*)

lemma *Cblinfun-comp-bounded-sesquilinear*: ‹*bounded-antilinear* (*CBlinfun o p*)›
if ‹*bounded-sesquilinear p*›
by (*metis* (*mono-tags*, *opaque-lifting*) *bounded-clinear-CBlinfun-apply bounded-sesquilinear.bounded-clinear-ri comp-apply that transfer-bounded-sesquilinear-bounded-antilinearI*)

declare (**in** *bounded-cbilinear*) *bounded-cbilinear-axioms*[*bounded-cbilinear*]
declare (**in** *bounded-clinear*) *bounded-clinear-axioms*[*bounded-clinear*]

lemma *norm-cblinfun-bound-both-sides*:
fixes a :: ‹$'a$::*complex-normed-vector* $\Rightarrow_{CL}$ $'b$::*complex-inner*›
assumes ‹$b \geq 0$›
assumes *leq*: ‹$\bigwedge \psi\ \varphi.$ *norm* $\psi = 1 \Longrightarrow$ *norm* $\varphi = 1 \Longrightarrow$ *norm* $(\psi \cdot_C a\ \varphi) \leq b$›
shows ‹*norm* $a \leq b$›
proof −
wlog *not-singleton*: ‹*class.not-singleton TYPE*($'a$)›
apply (*subst not-not-singleton-cblinfun-zero*)
by (*simp-all add*: *negation assms*)
have ‹*norm* $a = (\bigsqcup(\psi, \varphi).$ *cmod* $(\psi \cdot_C (a *_V \varphi)) / (\text{norm}\ \psi * \text{norm}\ \varphi))$›
apply (*rule cinner-sup-norm-cblinfun*[*internalize-sort'* $'a$])
apply (*rule complex-normed-vector-axioms*)
by (*fact not-singleton*)
also have ‹$\ldots \leq b$›
proof (*rule cSUP-least*)
show ‹*UNIV* $\neq \{\}$›
by *simp*
fix x :: ‹$'b \times 'a$›
obtain $\psi\ \varphi$ **where** x: ‹$x = (\psi, \varphi)$›
by *fastforce*
have ‹(*case* x *of* $(\psi, \varphi) \Rightarrow$ *cmod* $(\psi \cdot_C (a *_V \varphi)) / (\text{norm}\ \psi * \text{norm}\ \varphi)) =$ *cmod* $(\psi \cdot_C a\ \varphi) / (\text{norm}\ \psi * \text{norm}\ \varphi)$›
using x **by** *force*
also have ‹$\ldots =$ *cmod* (*sgn* $\psi \cdot_C a$ (*sgn* $\varphi)$)›
by (*simp add*: *sgn-div-norm cblinfun.scaleR-right divide-inverse-commute*

norm-inverse norm-mult)
also have ‹... ≤ *b*›
apply (*cases* ‹*ψ* = *0*›, *simp add*: *assms*)
apply (*cases* ‹*φ* = *0*›, *simp add*: *assms*)
apply (*rule leq*)
by (*simp-all add*: *norm-sgn*)
finally show ‹(*case x of* (*ψ*, *φ*) ⇒ *cmod* (*ψ* ·*C* (*a* *V *φ*)) / (*norm ψ* * *norm φ*)) ≤ *b*›
by −
qed
finally show *?thesis*
by −
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-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*:)

13.2 Relationship to real bounded operators (- ⇒$_L$ -)

instantiation *blinfun* :: (*real-normed-vector*, *complex-normed-vector*) *complex-normed-vector*
begin
lift-definition *scaleC-blinfun* :: ‹*complex* ⇒
(*'a*::*real-normed-vector*, *'b*::*complex-normed-vector*) *blinfun* ⇒
(*'a*, *'b*) *blinfun*›
is ‹λ *c*::*complex*. λ *f*::*'a*⇒*'b*. (λ *x*. *c* *$_C$ (*f x*))›
proof
fix *c*::*complex* **and** *f* :: ‹*'a*⇒*'b*› **and** *b1*::*'a* **and** *b2*::*'a*
assume ‹*bounded-linear f*›
show ‹*c* *$_C$ *f* (*b1* + *b2*) = *c* *$_C$ *f b1* + *c* *$_C$ *f b2*›
by (*simp add*: ‹*bounded-linear f*› *linear-simps scaleC-add-right*)

fix *c*::*complex* **and** *f* :: ‹*'a*⇒*'b*› **and** *b*::*'a* **and** *r*::*real*
assume ‹*bounded-linear f*›
show ‹*c* *$_C$ *f* (*r* *$_R$ *b*) = *r* *$_R$ (*c* *$_C$ *f b*)›
by (*simp add*: ‹*bounded-linear f*› *linear-simps*(*5*) *scaleR-scaleC*)

fix *c*::*complex* **and** *f* :: ‹*'a*⇒*'b*›
assume ‹*bounded-linear f*›

have ‹∃ *K*. ∀ *x*. *norm* (*f x*) ≤ *norm x* * *K*›
using ‹*bounded-linear f*›
by (*simp add*: *bounded-linear.bounded*)
then obtain *K* **where** ‹∀ *x*. *norm* (*f x*) ≤ *norm x* * *K*›
by *blast*
have ‹*cmod c* ≥ *0*›
by *simp*
hence ‹∀ *x*. (*cmod c*) * *norm* (*f x*) ≤ (*cmod c*) * *norm x* * *K*›
using ‹∀ *x*. *norm* (*f x*) ≤ *norm x* * *K*›
by (*metis ordered-comm-semiring-class.comm-mult-left-mono vector-space-over-itself.scale-scale*)
moreover have ‹*norm* (*c* *$_C$ *f x*) = (*cmod c*) * *norm* (*f x*)›
for *x*
by *simp*
ultimately show ‹∃ *K*. ∀ *x*. *norm* (*c* *$_C$ *f x*) ≤ *norm x* * *K*›
by (*metis ab-semigroup-mult-class.mult-ac*(*1*) *mult.commute*)

qed

instance
proof
have $r *_R x = complex\text{-}of\text{-}real\ r *_C x$
for $x :: ('a, 'b)\ blinfun$ **and** r
by *transfer* (*simp add*: *scaleR-scaleC*)
thus $((*_R)\ r::'a \Rightarrow_L 'b \Rightarrow \text{-}) = (*_C)\ (complex\text{-}of\text{-}real\ r)$ **for** r
by *auto*
show $a *_C (x + y) = a *_C x + a *_C y$
for $a :: complex$ **and** $x\ y :: 'a \Rightarrow_L 'b$
by *transfer* (*simp add*: *scaleC-add-right*)

show $(a + b) *_C x = a *_C x + b *_C x$
for $a\ b :: complex$ **and** $x :: 'a \Rightarrow_L 'b$
by *transfer* (*simp add*: *scaleC-add-left*)

show $a *_C b *_C x = (a * b) *_C x$
for $a\ b :: complex$ **and** $x :: 'a \Rightarrow_L 'b$
by *transfer simp*

have $‹1 *_C f\ x = f\ x›$
for $f :: ‹'a \Rightarrow 'b›$ **and** x
by *auto*
thus $1 *_C x = x$
for $x :: 'a \Rightarrow_L 'b$
by (*simp add*: *scaleC-blinfun.rep-eq blinfun-eqI*)

have $‹onorm\ (\lambda x.\ a *_C f\ x) = cmod\ a * onorm\ f›$
if $‹bounded\text{-}linear\ f›$
for $f :: ‹'a \Rightarrow 'b›$ **and** $a :: complex$
proof−
have $‹cmod\ a \geq 0›$
by *simp*
have $‹\exists\ K::real.\ \forall\ x.\ (|\ ereal\ ((norm\ (f\ x))\ /\ (norm\ x))\ |) \leq K›$
using $‹bounded\text{-}linear\ f›$ *le-onorm* **by** *fastforce*
then obtain $K::real$ **where** $‹\forall\ x.\ (|\ ereal\ ((norm\ (f\ x))\ /\ (norm\ x))\ |) \leq K›$
by *blast*
hence $‹\forall\ x.\ (cmod\ a) *(|\ ereal\ ((norm\ (f\ x))\ /\ (norm\ x))\ |) \leq (cmod\ a) * K›$
using $‹cmod\ a \geq 0›$
by (*metis abs-ereal.simps*(*1*) *abs-ereal-pos* *abs-pos ereal-mult-left-mono times-ereal.simps*(*1*))
hence $‹\forall\ x.\ (|\ ereal\ ((cmod\ a) * (norm\ (f\ x))\ /\ (norm\ x))\ |) \leq (cmod\ a) * K›$
by *simp*
hence $‹bdd\text{-}above\ \{ereal\ (cmod\ a * (norm\ (f\ x))\ /\ (norm\ x)) \mid x.\ True\}›$
by *simp*
moreover have $‹\{ereal\ (cmod\ a * (norm\ (f\ x))\ /\ (norm\ x)) \mid x.\ True\} \neq \{\}›$
by *auto*

ultimately have *p1*: ‹(*SUP x*. |*ereal* (*cmod a* * (*norm* (*f x*)) / (*norm x*))|) ≤ *cmod a* * *K*›
using ‹∀ *x*. | *ereal* (*cmod a* * (*norm* (*f x*)) / (*norm x*)) | ≤ *cmod a* * *K*›
Sup-least mem-Collect-eq
by (*simp add*: *SUP-le-iff*)
have *p2*: ‹⋀*i*. *i* ∈ *UNIV* ⟹ 0 ≤ *ereal* (*cmod a* * *norm* (*f i*) / *norm i*)›
by *simp*
hence ‹|*SUP x*. *ereal* (*cmod a* * (*norm* (*f x*)) / (*norm x*))|
≤ (*SUP x*. |*ereal* (*cmod a* * (*norm* (*f x*)) / (*norm x*))|)›
using ‹*bdd-above* {*ereal* (*cmod a* * (*norm* (*f x*)) / (*norm x*)) | *x*. *True*}›
‹{*ereal* (*cmod a* * (*norm* (*f x*)) / (*norm x*)) | *x*. *True*} ≠ {}›
by (*metis* (*mono-tags*, *lifting*) *SUP-upper2 Sup.SUP-cong UNIV-I*
p2 abs-ereal-ge0 ereal-le-real)
hence ‹|*SUP x*. *ereal* (*cmod a* * (*norm* (*f x*)) / (*norm x*))| ≤ *cmod a* * *K*›
using ‹(*SUP x*. |*ereal* (*cmod a* * (*norm* (*f x*)) / (*norm x*))|) ≤ *cmod a* * *K*›
by *simp*
hence ‹| (*SUP i*∈*UNIV*::'*a set*. *ereal* ((λ *x*. (*cmod a*) * (*norm* (*f x*)) / *norm x*) *i*)) | ≠ ∞›
by *auto*
hence *w2*: ‹(*SUP i*∈*UNIV*::'*a set*. *ereal* ((λ *x*. *cmod a* * (*norm* (*f x*)) / *norm x*) *i*))
= *ereal* (*Sup* ((λ *x*. *cmod a* * (*norm* (*f x*)) / *norm x*) ' (*UNIV*::'*a set*)))›
by (*simp add*: *ereal-SUP*)
have ‹(*UNIV*::('*a set*)) ≠ {}›
by *simp*
moreover have ‹⋀ *i*. *i* ∈ (*UNIV*::('*a set*)) ⟹ (λ *x*. (*norm* (*f x*)) / *norm x* :: *ereal*) *i* ≥ 0›
by *simp*
moreover have ‹*cmod a* ≥ 0›
by *simp*
ultimately have ‹(*SUP i*∈(*UNIV*::('*a set*)). ((*cmod a*)::*ereal*) * (λ *x*. (*norm* (*f x*)) / *norm x* :: *ereal*) *i*)
= ((*cmod a*)::*ereal*) * (*SUP i*∈(*UNIV*::('*a set*)). (λ *x*. (*norm* (*f x*)) / *norm x* :: *ereal*) *i*)›
by (*simp add*: *Sup-ereal-mult-left'*)
hence ‹(*SUP x*. ((*cmod a*)::*ereal*) * ((*norm* (*f x*)) / *norm x* :: *ereal*))
= ((*cmod a*)::*ereal*) * (*SUP x*. ((*norm* (*f x*)) / *norm x* :: *ereal*))›
by *simp*
hence *z1*: ‹*real-of-ereal* ((*SUP x*. ((*cmod a*)::*ereal*) * ((*norm* (*f x*)) / *norm x* :: *ereal*)))
= *real-of-ereal* (((*cmod a*)::*ereal*) * (*SUP x*. ((*norm* (*f x*)) / *norm x* :: *ereal*)))›
by *simp*
have *z2*: ‹*real-of-ereal* (*SUP x*. ((*cmod a*)::*ereal*) * ((*norm* (*f x*)) / *norm x* :: *ereal*))
= (*SUP x*. *cmod a* * (*norm* (*f x*) / *norm x*))›
using *w2*
by *auto*

have ‹*real-of-ereal* (((*cmod a*)::*ereal*) ∗ (*SUP x*. ((*norm* (*f x*)) / *norm x* :: *ereal*)))
= (*cmod a*) ∗ *real-of-ereal* (*SUP x*. ((*norm* (*f x*)) / *norm x* :: *ereal*))›
by *simp*
moreover have ‹*real-of-ereal* (*SUP x*. ((*norm* (*f x*)) / *norm x* :: *ereal*))
= (*SUP x*. ((*norm* (*f x*)) / *norm x*))›
proof−
have ‹| (*SUP i*∈*UNIV*::′*a set*. *ereal* ((λ *x*. (*norm* (*f x*)) / *norm x*) *i*)) | ≠ ∞›
proof−
have ‹∃ *K*::*real*. ∀ *x*. (| *ereal* ((*norm* (*f x*)) / (*norm x*)) |) ≤ *K*›
using ‹*bounded-linear f*› *le-onorm* **by** *fastforce*
then obtain *K*::*real* **where** ‹∀ *x*. (| *ereal* ((*norm* (*f x*)) / (*norm x*)) |) ≤ *K*›
by *blast*
hence ‹*bdd-above* {*ereal* ((*norm* (*f x*)) / (*norm x*)) | *x*. *True*}›
by *simp*
moreover have ‹{*ereal* ((*norm* (*f x*)) / (*norm x*)) | *x*. *True*} ≠ {}›
by *auto*
ultimately have ‹(*SUP x*. |*ereal* ((*norm* (*f x*)) / (*norm x*))|) ≤ *K*›
using ‹∀ *x*. | *ereal* ((*norm* (*f x*)) / (*norm x*)) | ≤ *K*›
Sup-least mem-Collect-eq
by (*simp add*: *SUP-le-iff*)
hence ‹|*SUP x*. *ereal* ((*norm* (*f x*)) / (*norm x*))|
≤ (*SUP x*. |*ereal* ((*norm* (*f x*)) / (*norm x*))|)›
using ‹*bdd-above* {*ereal* ((*norm* (*f x*)) / (*norm x*)) | *x*. *True*}›
‹{*ereal* ((*norm* (*f x*)) / (*norm x*)) | *x*. *True*} ≠ {}›
by (*metis* (*mono-tags*, *lifting*) *SUP-upper2 Sup.SUP-cong UNIV-I* ‹⋀*i*. *i* ∈ *UNIV* ⟹ 0 ≤ *ereal* (*norm* (*f i*) / *norm i*)› *abs-ereal-ge0 ereal-le-real*)
hence ‹|*SUP x*. *ereal* ((*norm* (*f x*)) / (*norm x*))| ≤ *K*›
using ‹(*SUP x*. |*ereal* ((*norm* (*f x*)) / (*norm x*))|) ≤ *K*›
by *simp*
thus *?thesis*
by *auto*
qed
hence ‹ (*SUP i*∈*UNIV*::′*a set*. *ereal* ((λ *x*. (*norm* (*f x*)) / *norm x*) *i*))
= *ereal* (*Sup* ((λ *x*. (*norm* (*f x*)) / *norm x*) ‘ (*UNIV*::′*a set*)))›
by (*simp add*: *ereal-SUP*)
thus *?thesis*
by *simp*
qed
have *z3*: ‹*real-of-ereal* (((*cmod a*)::*ereal*) ∗ (*SUP x*. ((*norm* (*f x*)) / *norm x* :: *ereal*)))
= *cmod a* ∗ (*SUP x*. *norm* (*f x*) / *norm x*)›
by (*simp add*: ‹*real-of-ereal* (*SUP x*. *ereal* (*norm* (*f x*) / *norm x*)) = (*SUP x*. *norm* (*f x*) / *norm x*)›)
hence *w1*: ‹(*SUP x*. *cmod a* ∗ (*norm* (*f x*) / *norm x*)) =
cmod a ∗ (*SUP x*. *norm* (*f x*) / *norm x*)›

using *z1 z2* **by** *linarith*
have *v1*: ‹*onorm* (λ*x*. *a* *$_C$ *f x*) = (*SUP x*. *norm* (*a* *$_C$ *f x*) / *norm x*)›
by (*simp add*: *onorm-def*)
have *v2*: ‹(*SUP x*. *norm* (*a* *$_C$ *f x*) / *norm x*) = (*SUP x*. ((*cmod a*) * *norm* (*f x*)) / *norm x*)›
by *simp*
have *v3*: ‹(*SUP x*. ((*cmod a*) * *norm* (*f x*)) / *norm x*) = (*SUP x*. (*cmod a*) * ((*norm* (*f x*)) / *norm x*))›
by *simp*
have *v4*: ‹(*SUP x*. (*cmod a*) * ((*norm* (*f x*)) / *norm x*)) = (*cmod a*) * (*SUP x*. ((*norm* (*f x*)) / *norm x*))›
using *w1*
by *blast*
show ‹*onorm* (λ*x*. *a* *$_C$ *f x*) = *cmod a* * *onorm f*›
using *v1 v2 v3 v4*
by (*metis* (*mono-tags*, *lifting*) *onorm-def*)
qed
thus ‹*norm* (*a* *$_C$ *x*) = *cmod a* * *norm x*›
for *a*::*complex* **and** *x*::‹('*a*, '*b*) *blinfun*›
by *transfer blast*
qed
end

lemma *clinear-blinfun-compose-left*: ‹*clinear* (λ*x*. *blinfun-compose x y*)›
by (*auto intro*!: *clinearI simp*: *blinfun-eqI scaleC-blinfun.rep-eq bounded-bilinear.add-left bounded-bilinear-blinfun-compose*)

instance *blinfun* :: (*real-normed-vector*, *cbanach*) *cbanach*..

lemma *blinfun-compose-assoc*: (*A* *$_L$ *B*) *$_L$ *C* = *A* *$_L$ (*B* *$_L$ *C*)
by (*simp add*: *blinfun-eqI*)

lift-definition *blinfun-of-cblinfun*::‹'*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector* ⇒ '*a* ⇒$_L$ '*b*› **is** *id*
by *transfer* (*simp add*: *bounded-clinear.bounded-linear*)

lift-definition *blinfun-cblinfun-eq* ::
‹'*a* ⇒$_L$ '*b* ⇒ '*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector* ⇒ *bool*›
is (=) .

lemma *blinfun-cblinfun-eq-bi-unique*[*transfer-rule*]: ‹*bi-unique blinfun-cblinfun-eq*›
unfolding *bi-unique-def* **by** *transfer auto*

lemma *blinfun-cblinfun-eq-right-total*[*transfer-rule*]: ‹*right-total blinfun-cblinfun-eq*›
unfolding *right-total-def* **by** *transfer* (*simp add*: *bounded-clinear.bounded-linear*)

named-theorems *cblinfun-blinfun-transfer*

lemma *cblinfun-blinfun-transfer-0*[*cblinfun-blinfun-transfer*]:
blinfun-cblinfun-eq (*0*::(-,-) *blinfun*) (*0*::(-,-) *cblinfun*)
by *transfer simp*

lemma *cblinfun-blinfun-transfer-plus*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (+) (+)
unfolding *rel-fun-def* **by** *transfer auto*

lemma *cblinfun-blinfun-transfer-minus*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (−) (−)
unfolding *rel-fun-def* **by** *transfer auto*

lemma *cblinfun-blinfun-transfer-uminus*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (*uminus*) (*uminus*)
unfolding *rel-fun-def* **by** *transfer auto*

definition *real-complex-eq r c* ⟷ *complex-of-real r* = *c*

lemma *bi-unique-real-complex-eq*[*transfer-rule*]: ‹*bi-unique real-complex-eq*›
unfolding *real-complex-eq-def bi-unique-def* **by** *auto*

lemma *left-total-real-complex-eq*[*transfer-rule*]: ‹*left-total real-complex-eq*›
unfolding *real-complex-eq-def left-total-def* **by** *auto*

lemma *cblinfun-blinfun-transfer-scaleC*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*real-complex-eq* ===> *blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (*scaleR*) (*scaleC*)
unfolding *rel-fun-def* **by** *transfer* (*simp add*: *real-complex-eq-def scaleR-scaleC*)

lemma *cblinfun-blinfun-transfer-CBlinfun*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*eq-onp bounded-clinear* ===> *blinfun-cblinfun-eq*) *Blinfun CBlinfun*
unfolding *rel-fun-def blinfun-cblinfun-eq.rep-eq eq-onp-def*
by (*auto simp*: *CBlinfun-inverse Blinfun-inverse bounded-clinear.bounded-linear*)

lemma *cblinfun-blinfun-transfer-norm*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> (=)) *norm norm*
unfolding *rel-fun-def* **by** *transfer auto*

lemma *cblinfun-blinfun-transfer-dist*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq* ===> (=)) *dist dist*

unfolding *rel-fun-def dist-norm* **by** *transfer auto*

lemma *cblinfun-blinfun-transfer-sgn*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) *sgn sgn*
unfolding *rel-fun-def sgn-blinfun-def sgn-cblinfun-def* **by** *transfer* (*auto simp*: *scaleR-scaleC*)

lemma *cblinfun-blinfun-transfer-Cauchy*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (((=) ===> *blinfun-cblinfun-eq*) ===> (=)) *Cauchy Cauchy*
proof −
note *cblinfun-blinfun-transfer*[*transfer-rule*]
show *?thesis*
unfolding *Cauchy-def*
by *transfer-prover*
qed

lemma *cblinfun-blinfun-transfer-tendsto*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (((=) ===> *blinfun-cblinfun-eq*) ===> *blinfun-cblinfun-eq* ===> (=) ===> (=)) *tendsto tendsto*
proof −
note *cblinfun-blinfun-transfer*[*transfer-rule*]
show *?thesis*
unfolding *tendsto-iff*
by *transfer-prover*
qed

lemma *cblinfun-blinfun-transfer-compose*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (o_L) ($o_{C\,L}$)
unfolding *rel-fun-def* **by** *transfer auto*

lemma *cblinfun-blinfun-transfer-apply*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> (=) ===> (=)) *blinfun-apply cblinfun-apply*
unfolding *rel-fun-def* **by** *transfer auto*

lemma *blinfun-of-cblinfun-inj*:
‹*blinfun-of-cblinfun f* = *blinfun-of-cblinfun g* ⟹ *f* = *g*›
by (*metis cblinfun-apply-inject blinfun-of-cblinfun.rep-eq*)

lemma *blinfun-of-cblinfun-inv*:
assumes ⋀*c*. ⋀*x*. *f* *$_v$ (*c* *$_C$ *x*) = *c* *$_C$ (*f* *$_v$ *x*)
shows ∃ *g*. *blinfun-of-cblinfun g* = *f*
using *assms*
proof *transfer*

show $\exists g{\in}Collect\ bounded\text{-}clinear.\ id\ g = f$
if *bounded-linear f*
and $\bigwedge c\ x.\ f\ (c *_C x) = c *_C f\ x$
for $f :: 'a \Rightarrow 'b$
using *that bounded-linear-bounded-clinear* **by** *auto*
qed

lemma *blinfun-of-cblinfun-zero*:
‹*blinfun-of-cblinfun 0 = 0*›
by *transfer simp*

lemma *blinfun-of-cblinfun-uminus*:
‹*blinfun-of-cblinfun* $(-f) = -$ (*blinfun-of-cblinfun f*)›
by *transfer auto*

lemma *blinfun-of-cblinfun-minus*:
‹*blinfun-of-cblinfun* $(f - g) =$ *blinfun-of-cblinfun f* $-$ *blinfun-of-cblinfun g*›
by *transfer auto*

lemma *blinfun-of-cblinfun-scaleC*:
‹*blinfun-of-cblinfun* $(c *_C f) = c *_C$ (*blinfun-of-cblinfun f*)›
by *transfer auto*

lemma *blinfun-of-cblinfun-scaleR*:
‹*blinfun-of-cblinfun* $(c *_R f) = c *_R$ (*blinfun-of-cblinfun f*)›
by *transfer auto*

lemma *blinfun-of-cblinfun-norm*:
fixes f::‹$'a$::*complex-normed-vector* $\Rightarrow_{CL}$ $'b$::*complex-normed-vector*›
shows ‹*norm f = norm* (*blinfun-of-cblinfun f*)›
by *transfer auto*

lemma *blinfun-of-cblinfun-cblinfun-compose*:
fixes f::‹$'b$::*complex-normed-vector* $\Rightarrow_{CL}$ $'c$::*complex-normed-vector*›
and g::‹$'a$::*complex-normed-vector* $\Rightarrow_{CL}$ $'b$›
shows ‹*blinfun-of-cblinfun* $(f\ o_{CL}\ g) =$ (*blinfun-of-cblinfun f*) o_L (*blinfun-of-cblinfun g*)›
by *transfer auto*

13.3 Composition

lemma *cblinfun-compose-assoc*:
shows $(A\ o_{CL}\ B)\ o_{CL}\ C = A\ o_{CL}\ (B\ o_{CL}\ C)$
by (*metis* (*no-types*, *lifting*) *cblinfun-apply-inject fun.map-comp cblinfun-compose.rep-eq*)

lemma *cblinfun-compose-zero-right*[*simp*]: $U\ o_{CL}\ 0 = 0$
using *bounded-cbilinear.zero-right bounded-cbilinear-cblinfun-compose* **by** *blast*

lemma *cblinfun-compose-zero-left*[*simp*]: $0\ o_{CL}\ U = 0$

using *bounded-cbilinear.zero-left bounded-cbilinear-cblinfun-compose* **by** *blast*

lemma *cblinfun-compose-scaleC-left*[*simp*]:
fixes *A*::*'b*::*complex-normed-vector* $\Rightarrow_{CL}$ *'c*::*complex-normed-vector*
and *B*::*'a*::*complex-normed-vector* $\Rightarrow_{CL}$ *'b*
shows ‹$(a *_C A) \; o_{CL} \; B = a *_C (A \; o_{CL} \; B)$›
by (*simp add*: *bounded-cbilinear.scaleC-left bounded-cbilinear-cblinfun-compose*)

lemma *cblinfun-compose-scaleR-left*[*simp*]:
fixes *A*::*'b*::*complex-normed-vector* $\Rightarrow_{CL}$ *'c*::*complex-normed-vector*
and *B*::*'a*::*complex-normed-vector* $\Rightarrow_{CL}$ *'b*
shows ‹$(a *_R A) \; o_{CL} \; B = a *_R (A \; o_{CL} \; B)$›
by (*simp add*: *scaleR-scaleC*)

lemma *cblinfun-compose-scaleC-right*[*simp*]:
fixes *A*::*'b*::*complex-normed-vector* $\Rightarrow_{CL}$ *'c*::*complex-normed-vector*
and *B*::*'a*::*complex-normed-vector* $\Rightarrow_{CL}$ *'b*
shows ‹$A \; o_{CL} \; (a *_C B) = a *_C (A \; o_{CL} \; B)$›
by *transfer* (*auto intro*!: *ext bounded-clinear.clinear complex-vector.linear-scale*)

lemma *cblinfun-compose-scaleR-right*[*simp*]:
fixes *A*::*'b*::*complex-normed-vector* $\Rightarrow_{CL}$ *'c*::*complex-normed-vector*
and *B*::*'a*::*complex-normed-vector* $\Rightarrow_{CL}$ *'b*
shows ‹$A \; o_{CL} \; (a *_R B) = a *_R (A \; o_{CL} \; B)$›
by (*simp add*: *scaleR-scaleC*)

lemma *cblinfun-compose-id-right*[*simp*]:
shows $U \; o_{CL} \; id\text{-}cblinfun = U$
by *transfer auto*

lemma *cblinfun-compose-id-left*[*simp*]:
shows $id\text{-}cblinfun \; o_{CL} \; U = U$
by *transfer auto*

lemma *cblinfun-compose-add-left*: ‹$(a + b) \; o_{CL} \; c = (a \; o_{CL} \; c) + (b \; o_{CL} \; c)$›
by (*simp add*: *bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose*)

lemma *cblinfun-compose-add-right*: ‹$a \; o_{CL} \; (b + c) = (a \; o_{CL} \; b) + (a \; o_{CL} \; c)$›
by (*simp add*: *bounded-cbilinear.add-right bounded-cbilinear-cblinfun-compose*)

lemma *cbilinear-cblinfun-compose*[*simp*]: *cbilinear cblinfun-compose*
by (*auto intro*!: *clinearI simp add*: *cbilinear-def bounded-cbilinear.add-left bounded-cbilinear.add-right bounded-cbilinear-cblinfun-compose*)

lemma *cblinfun-compose-sum-left*: ‹$(\sum i{\in}S.\ g\ i) \; o_{CL} \; x = (\sum i{\in}S.\ g\ i \; o_{CL} \; x)$›
by (*induction S rule*:*infinite-finite-induct*) (*auto simp*: *cblinfun-compose-add-left*)

lemma *cblinfun-compose-sum-right*: ‹$x \; o_{CL} \; (\sum i{\in}S.\ g\ i) = (\sum i{\in}S.\ x \; o_{CL} \; g\ i)$›
by (*induction S rule*:*infinite-finite-induct*) (*auto simp*: *cblinfun-compose-add-right*)

lemma *cblinfun-compose-minus-right*: ‹$a\ o_{CL}\ (b - c) = (a\ o_{CL}\ b) - (a\ o_{CL}\ c)$›
by (*simp add*: *bounded-cbilinear.diff-right bounded-cbilinear-cblinfun-compose*)
lemma *cblinfun-compose-minus-left*: ‹$(a - b)\ o_{CL}\ c = (a\ o_{CL}\ c) - (b\ o_{CL}\ c)$›
by (*simp add*: *bounded-cbilinear.diff-left bounded-cbilinear-cblinfun-compose*)

lemma *simp-a-oCL-b*: ‹$a\ o_{CL}\ b = c \Longrightarrow a\ o_{CL}\ (b\ o_{CL}\ d) = c\ o_{CL}\ d$›
— A convenience lemma to transform simplification rules of the form $a\ o_{CL}\ b = c$. E.g., *simp-a-oCL-b*[*OF isometryD*, *simp*] declares a simp-rule for simplifying $adj\ x\ o_{CL}\ (x\ o_{CL}\ y) = id\text{-}cblinfun\ o_{CL}\ y$.
by (*auto simp*: *cblinfun-compose-assoc*)

lemma *simp-a-oCL-b'*: ‹$a\ o_{CL}\ b = c \Longrightarrow a *_V (b *_V d) = c *_V d$›
— A convenience lemma to transform simplification rules of the form $a\ o_{CL}\ b = c$. E.g., *simp-a-oCL-b'*[*OF isometryD*, *simp*] declares a simp-rule for simplifying $adj\ x *_V x *_V y = id\text{-}cblinfun *_V y$.
by *auto*

lemma *cblinfun-compose-uminus-left*: ‹$(- a)\ o_{CL}\ b = - (a\ o_{CL}\ b)$›
by (*simp add*: *bounded-cbilinear.minus-left bounded-cbilinear-cblinfun-compose*)

lemma *cblinfun-compose-uminus-right*: ‹$a\ o_{CL}\ (- b) = - (a\ o_{CL}\ b)$›
by (*simp add*: *bounded-cbilinear.minus-right bounded-cbilinear-cblinfun-compose*)

lemma *bounded-clinear-cblinfun-compose-left*: ‹*bounded-clinear* $(\lambda x.\ x\ o_{CL}\ y)$›
by (*simp add*: *bounded-cbilinear.bounded-clinear-left bounded-cbilinear-cblinfun-compose*)
lemma *bounded-clinear-cblinfun-compose-right*: ‹*bounded-clinear* $(\lambda y.\ x\ o_{CL}\ y)$›
by (*simp add*: *bounded-cbilinear.bounded-clinear-right bounded-cbilinear-cblinfun-compose*)
lemma *clinear-cblinfun-compose-left*: ‹*clinear* $(\lambda x.\ x\ o_{CL}\ y)$›
by (*simp add*: *bounded-cbilinear.bounded-clinear-left bounded-cbilinear-cblinfun-compose bounded-clinear.clinear*)
lemma *clinear-cblinfun-compose-right*: ‹*clinear* $(\lambda y.\ x\ o_{CL}\ y)$›
by (*simp add*: *bounded-clinear.clinear bounded-clinear-cblinfun-compose-right*)

lemma *additive-cblinfun-compose-left*[*simp*]: ‹*Modules.additive* $(\lambda x.\ x\ o_{CL}\ a)$›
by (*simp add*: *Modules.additive-def cblinfun-compose-add-left*)
lemma *additive-cblinfun-compose-right*[*simp*]: ‹*Modules.additive* $(\lambda x.\ a\ o_{CL}\ x)$›
by (*simp add*: *Modules.additive-def cblinfun-compose-add-right*)
lemma *isCont-cblinfun-compose-left*: ‹*isCont* $(\lambda x.\ x\ o_{CL}\ a)\ y$›
apply (*rule clinear-continuous-at*)
by (*rule bounded-clinear-cblinfun-compose-left*)
lemma *isCont-cblinfun-compose-right*: ‹*isCont* $(\lambda x.\ a\ o_{CL}\ x)\ y$›
apply (*rule clinear-continuous-at*)
by (*rule bounded-clinear-cblinfun-compose-right*)

lemma *cspan-compose-closed*:
assumes ‹$\bigwedge a\ b.\ a \in A \Longrightarrow b \in A \Longrightarrow a\ o_{CL}\ b \in A$›
assumes ‹$a \in$ *cspan* A› **and** ‹$b \in$ *cspan* A›

shows ‹a o_{CL} $b \in$ *cspan* A›
proof −
from ‹$a \in$ *cspan* A›
obtain F f **where** ‹*finite* F› **and** ‹$F \subseteq A$› **and** *a-def*: ‹$a = (\sum x{\in}F.\ f\ x *_C x)$›
by (*smt* (*verit*, *del-insts*) *complex-vector.span-explicit mem-Collect-eq*)
from ‹$b \in$ *cspan* A›
obtain G g **where** ‹*finite* G› **and** ‹$G \subseteq A$› **and** *b-def*: ‹$b = (\sum x{\in}G.\ g\ x *_C x)$›
by (*smt* (*verit*, *del-insts*) *complex-vector.span-explicit mem-Collect-eq*)
have ‹a o_{CL} $b = (\sum (x,y){\in}F{\times}G.\ (f\ x *_C x)\ o_{CL}\ (g\ y *_C y))$›
apply (*simp add*: *a-def b-def cblinfun-compose-sum-left*)
by (*auto intro*!: *sum.cong simp add*: *cblinfun-compose-sum-right scaleC-sum-right sum.cartesian-product case-prod-beta*)
also have ‹$\ldots = (\sum (x,y){\in}F{\times}G.\ (f\ x * g\ y) *_C (x\ o_{CL}\ y))$›
by (*metis* (*no-types*, *opaque-lifting*) *cblinfun-compose-scaleC-left cblinfun-compose-scaleC-right scaleC-scaleC*)
also have ‹$\ldots \in$ *cspan* A›
using *assms* ‹$G \subseteq A$› ‹$F \subseteq A$›
apply (*auto intro*!: *complex-vector.span-sum complex-vector.span-scale simp*: *complex-vector.span-clauses*)
using *complex-vector.span-clauses*(*1*) **by** *blast*
finally show *?thesis*
by −
qed

13.4 Adjoint

lift-definition
adj :: *'a*::*chilbert-space* ⇒$_{CL}$ *'b*::*complex-inner* ⇒ *'b* ⇒$_{CL}$ *'a* (‹-*›) [*99*] *100*)
is *cadjoint* **by** (*fact cadjoint-bounded-clinear*)

definition *selfadjoint* :: ‹(*'a*::*chilbert-space* ⇒$_{CL}$ *'a*) ⇒ *bool*› **where**
‹*selfadjoint* $a \longleftrightarrow a* = a$›

lemma *id-cblinfun-adjoint*[*simp*]: *id-cblinfun** = *id-cblinfun*
by (*metis adj.rep-eq apply-id-cblinfun cadjoint-id cblinfun-apply-inject*)

lemma *double-adj*[*simp*]: $(A*)* = A$
apply *transfer* **using** *double-cadjoint* **by** *blast*

lemma *adj-cblinfun-compose*[*simp*]:
fixes *B*::‹*'a*::*chilbert-space* ⇒$_{CL}$ *'b*::*chilbert-space*›
and *A*::‹*'b* ⇒$_{CL}$ *'c*::*complex-inner*›
shows $(A\ o_{CL}\ B)* =\ (B*)\ o_{CL}\ (A*)$
proof *transfer*
fix *A* :: ‹*'b* ⇒ *'c*› **and** *B* :: ‹*'a* ⇒ *'b*›
assume ‹*bounded-clinear A*› **and** ‹*bounded-clinear B*›
hence ‹*bounded-clinear* $(A \circ B)$›
by (*simp add*: *comp-bounded-clinear*)

have ‹$((A \circ B)\ u \cdot_C\ v) = (u \cdot_C\ (B^\dagger \circ A^\dagger)\ v)$›
for $u\ v$
by (*metis* (*no-types*, *lifting*) *cadjoint-univ-prop* ‹*bounded-clinear* A› ‹*bounded-clinear* B› *cinner-commute′ comp-def*)
thus ‹$(A \circ B)^\dagger = B^\dagger \circ A^\dagger$›
using ‹*bounded-clinear* $(A \circ B)$›
by (*metis cadjoint-eqI cinner-commute′*)
qed

lemma *scaleC-adj*[*simp*]: $(a *_C\ A)* = (cnj\ a) *_C\ (A*)$
by *transfer* (*simp add*: *bounded-clinear.bounded-linear bounded-clinear-def complex-vector.linear-scale scaleC-cadjoint*)

lemma *scaleR-adj*[*simp*]: $(a *_R\ A)* = a *_R\ (A*)$
by (*simp add*: *scaleR-scaleC*)

lemma *adj-plus*: ‹$(A + B)* = (A*) + (B*)$›
proof *transfer*
fix $A\ B$::‹$'b \Rightarrow 'a$›
assume *a1*: ‹*bounded-clinear* A› **and** *a2*: ‹*bounded-clinear* B›
define F **where** ‹$F = (\lambda x.\ (A^\dagger)\ x + (B^\dagger)\ x)$›
define G **where** ‹$G = (\lambda x.\ A\ x + B\ x)$›
have ‹*bounded-clinear* G›
unfolding *G-def*
by (*simp add*: *a1 a2 bounded-clinear-add*)
moreover have ‹$(F\ u \cdot_C\ v) = (u \cdot_C\ G\ v)$› **for** $u\ v$
unfolding *F-def G-def*
using *cadjoint-univ-prop a1 a2 cinner-add-left*
by (*simp add*: *cadjoint-univ-prop cinner-add-left cinner-add-right*)
ultimately have ‹$F = G^\dagger$ ›
using *cadjoint-eqI* **by** *blast*
thus ‹$(\lambda x.\ A\ x + B\ x)^\dagger = (\lambda x.\ (A^\dagger)\ x + (B^\dagger)\ x)$›
unfolding *F-def G-def*
by *auto*
qed

lemma *cinner-adj-left*:
fixes G :: $'b$::*chilbert-space* $\Rightarrow_{CL}$ $'a$::*complex-inner*
shows ‹$(G* *_V\ x) \cdot_C\ y = x \cdot_C\ (G *_V\ y)$›
apply *transfer* **using** *cadjoint-univ-prop* **by** *blast*

lemma *cinner-adj-right*:
fixes G :: $'b$::*chilbert-space* $\Rightarrow_{CL}$ $'a$::*complex-inner*
shows ‹$x \cdot_C\ (G* *_V\ y) = (G *_V\ x) \cdot_C\ y$›
apply *transfer* **using** *cadjoint-univ-prop′* **by** *blast*

lemma *adj-0*[*simp*]: ‹$0* = 0$›
by (*metis add-cancel-right-left adj-plus*)

lemma *selfadjoint-0*[*simp*]: ‹*selfadjoint 0*›
by (*simp add*: *selfadjoint-def*)

lemma *norm-adj*[*simp*]: ‹*norm* (*A*∗) = *norm A*›
for *A* :: ‹'*b*::*chilbert-space* ⇒$_{CL}$ '*c*::*complex-inner*›
proof (*cases* ‹(∃ *x y* :: '*b*. *x* ≠ *y*) ∧ (∃ *x y* :: '*c*. *x* ≠ *y*)›)
case *True*
then have *c1*: ‹*class.not-singleton TYPE*('*b*)›
by *intro-classes simp*
from *True* **have** *c2*: ‹*class.not-singleton TYPE*('*c*)›
by *intro-classes simp*
have *normA*: ‹*norm A* = (*SUP* (*ψ*, *φ*). *cmod* (*ψ* ·$_C$ (*A* ∗$_V$ *φ*)) / (*norm ψ* ∗ *norm φ*))›
apply (*rule cinner-sup-norm-cblinfun*[*internalize-sort* ‹'*a*::{*complex-normed-vector*,*not-singleton*}›])
apply (*rule complex-normed-vector-axioms*)
by (*rule c1*)
have *normAadj*: ‹*norm* (*A*∗) = (*SUP* (*ψ*, *φ*). *cmod* (*ψ* ·$_C$ (*A*∗ ∗$_V$ *φ*)) / (*norm ψ* ∗ *norm φ*))›
apply (*rule cinner-sup-norm-cblinfun*[*internalize-sort* ‹'*a*::{*complex-normed-vector*,*not-singleton*}›])
apply (*rule complex-normed-vector-axioms*)
by (*rule c2*)

have ‹*norm* (*A*∗) = (*SUP* (*ψ*, *φ*). *cmod* (*φ* ·$_C$ (*A* ∗$_V$ *ψ*)) / (*norm ψ* ∗ *norm φ*))›
unfolding *normAadj*
apply (*subst cinner-adj-right*)
apply (*subst cinner-commute*)
apply (*subst complex-mod-cnj*)
by *rule*
also have ‹... = *Sup* ((λ(*ψ*, *φ*). *cmod* (*φ* ·$_C$ (*A* ∗$_V$ *ψ*)) / (*norm ψ* ∗ *norm φ*)) ‘ *prod.swap* ‘ *UNIV*)›
by *auto*
also have ‹... = (*SUP* (*φ*, *ψ*). *cmod* (*φ* ·$_C$ (*A* ∗$_V$ *ψ*)) / (*norm ψ* ∗ *norm φ*))›
apply (*subst image-image*)
by *auto*
also have ‹... = *norm A*›
unfolding *normA*
by (*simp add*: *mult.commute*)
finally show *?thesis*
by −
next
case *False*
then consider (*b*) ‹⋀*x*::'*b*. *x* = *0*› | (*c*) ‹⋀*x*::'*c*. *x* = *0*›
by *auto*
then have ‹*A* = *0*›
apply (*cases*; *transfer*)
apply (*metis* (*full-types*) *bounded-clinear-def complex-vector.linear-0*)
by *auto*
then show ‹*norm* (*A*∗) = *norm A*›
by *simp*

qed

lemma *antilinear-adj*[*simp*]: ‹*antilinear adj*›
by (*simp add*: *adj-plus antilinearI*)

lemma *bounded-antilinear-adj*[*bounded-antilinear*, *simp*]: ‹*bounded-antilinear adj*›
by (*auto intro*!: *antilinearI exI*[*of - 1*] *simp*: *bounded-antilinear-def bounded-antilinear-axioms-def adj-plus*)

lemma *adjoint-eqI*:
fixes *G*:: ‹*'b*::*chilbert-space* ⇒$_{CL}$ *'a*::*complex-inner*›
and *F*:: ‹*'a* ⇒$_{CL}$ *'b*›
assumes ‹⋀*x y*. ((*cblinfun-apply F*) *x* ∙$_C$ *y*) = (*x* ∙$_C$ (*cblinfun-apply G*) *y*)›
shows ‹*F* = *G*∗›
using *assms* **apply** *transfer* **using** *cadjoint-eqI* **by** *auto*

lemma *adj-uminus*: ‹(−*A*)∗ = − (*A*∗)›
by (*metis scaleR-adj scaleR-minus1-left scaleR-minus1-left*)

lemma *cinner-real-selfadjointI*:
— Prop. II.2.12 in [1]
assumes ‹⋀*ψ*. *ψ* ∙$_C$ (*A* ∗$_V$ *ψ*) ∈ ℝ›
shows ‹*selfadjoint A*›
proof −
{ fix *g h* :: *'a*
{
fix *α* :: *complex*
have ‹*cinner h* (*A h*) + *cnj α* ∗$_C$ *cinner g* (*A h*) + *α* ∗$_C$ *cinner h* (*A g*) + (*abs α*)^2 ∗ *cinner g* (*A g*)
= *cinner* (*h* + *α* ∗$_C$ *g*) (*A* ∗$_V$ (*h* + *α* ∗$_C$ *g*))› (**is** ‹*?sum4* = -›)
apply (*auto simp*: *cinner-add-right cinner-add-left cblinfun.add-right cblinfun.scaleC-right ring-class.ring-distribs*)
by (*metis cnj-x-x mult.commute*)
also have ‹... ∈ ℝ›
using *assms* **by** *auto*
finally have ‹*?sum4* = *cnj ?sum4*›
using *Reals-cnj-iff* **by** *fastforce*
then have ‹*cnj α* ∗$_C$ *cinner g* (*A h*) + *α* ∗$_C$ *cinner h* (*A g*)
= *α* ∗$_C$ *cinner* (*A h*) *g* + *cnj α* ∗$_C$ *cinner* (*A g*) *h*›
using *Reals-cnj-iff abs-complex-real assms* **by** *force*
also have ‹... = *α* ∗$_C$ *cinner h* (*A*∗ ∗$_V$ *g*) + *cnj α* ∗$_C$ *cinner g* (*A*∗ ∗$_V$ *h*)›
by (*simp add*: *cinner-adj-right*)
finally have ‹*cnj α* ∗$_C$ *cinner g* (*A h*) + *α* ∗$_C$ *cinner h* (*A g*) = *α* ∗$_C$ *cinner h* (*A*∗ ∗$_V$ *g*) + *cnj α* ∗$_C$ *cinner g* (*A*∗ ∗$_V$ *h*)›
by −
}
from *this*[**where** *α2=1*] *this*[**where** *α2*=i]
have *1*: ‹*cinner g* (*A h*) + *cinner h* (*A g*) = *cinner h* (*A*∗ ∗$_V$ *g*) + *cinner g*

```
(A* *_V h)›
      and i: ‹− i * cinner g (A h) + i *_C cinner h (A g) =  i *_C cinner h (A* *_V
g) − i *_C cinner g (A* *_V h)›
      by auto
    from arg-cong2[OF 1 arg-cong[OF i, where f=‹(*) (−i)›], where f=plus]
    have ‹cinner h (A g) = cinner h (A* *_V g)›
      by (auto simp: ring-class.ring-distribs)
  }
  then have ‹A* = A›
    apply (rule-tac sym)
    by (simp add: adjoint-eqI cinner-adj-right)
  then show selfadjoint A
    by (simp add: selfadjoint-def)
qed


lemma norm-AAadj[simp]: ‹norm (A o_CL A*) = (norm A)²› for A :: ‹'a::chilbert-space
⇒_CL 'b::{complex-inner}›
proof (cases ‹class.not-singleton TYPE('b)›)
  case True
  then have [simp]: ‹class.not-singleton TYPE('b)›
    by −
  have 1: ‹(norm A)² * ε ≤ norm (A o_CL A*)› if ‹ε < 1› and ‹ε ≥ 0› for ε
  proof −
    obtain ψ where ψ: ‹norm ((A*) *_V ψ) ≥ norm (A*) * sqrt ε› and [simp]:
‹norm ψ = 1›
      apply atomize-elim
      apply (rule cblinfun-norm-approx-witness-mult[internalize-sort' 'a])
    using ‹ε < 1› by (auto intro: complex-normed-vector-class.complex-normed-vector-axioms)
    have ‹complex-of-real ((norm A)² * ε) = (norm (A*) * sqrt ε)²›
      by (simp add: ordered-field-class.sign-simps(23) that(2))
    also have ‹... ≤ (norm ((A* *_V ψ)))²›
    by (meson ψ complex-of-real-mono mult-nonneg-nonneg norm-ge-zero power-mono
real-sqrt-ge-zero ‹ε ≥ 0›)
    also have ‹... ≤ cinner (A* *_V ψ) (A* *_V ψ)›
      by (auto simp flip: power2-norm-eq-cinner)
    also have §: ‹... = cinner ψ ((A o_CL A*) *_V ψ)›
      by (auto simp: cinner-adj-left)
    also have ‹... ≤ norm (A o_CL A*)›
      using ‹norm ψ = 1›
     by (smt (verit) Re-complex-of-real § cdot-square-norm cinner-ge-zero cmod-Re
complex-inner-class.Cauchy-Schwarz-ineq2 complex-of-real-mono mult-cancel-left1
mult-cancel-right1 norm-cblinfun)
    finally show ?thesis
      by (auto simp: less-eq-complex-def)
  qed
  then have 1: ‹(norm A)² ≤ norm (A o_CL A*)›
   by (metis field-le-mult-one-interval less-eq-real-def ordered-field-class.sign-simps(5))
```

```
  have 2: ‹norm (A o_CL A*) ≤ (norm A)²›
  proof (rule norm-cblinfun-bound)
    show ‹0 ≤ (norm A)²› by simp
    fix ψ
    have ‹norm ((A o_CL A*) *_V ψ) = norm (A *_V A* *_V ψ)›
      by auto
    also have ‹... ≤ norm A * norm (A* *_V ψ)›
      by (simp add: norm-cblinfun)
    also have ‹... ≤ norm A * norm (A*) * norm ψ›
     by (metis mult.assoc norm-cblinfun norm-imp-pos-and-ge ordered-comm-semiring-class.comm-mult-left-mo
    also have ‹... = (norm A)² * norm ψ›
      by (simp add: power2-eq-square)
    finally show ‹norm ((A o_CL A*) *_V ψ) ≤ (norm A)² * norm ψ›
      by −
  qed

  from 1 2 show ?thesis by simp
next
  case False
  then have [simp]: ‹class.CARD-1 TYPE('b)›
    by (rule not-singleton-vs-CARD-1)
  have ‹A = 0›
    apply (rule cblinfun-to-CARD-1-0[internalize-sort' 'b])
    by (auto intro: complex-normed-vector-class.complex-normed-vector-axioms)
  then show ?thesis
    by auto
qed

lemma sum-adj: ‹(sum a F)* = sum (λi. (a i)*) F›
  by (induction rule:infinite-finite-induct) (auto simp add: adj-plus)

lemma has-sum-adj:
  assumes ‹(f has-sum x) I›
  shows ‹((λx. adj (f x)) has-sum adj x) I›

  apply (rule has-sum-comm-additive[where f=adj, unfolded o-def])
  apply (simp add: antilinear.axioms(1))
 apply (metis (no-types, lifting) LIM-eq adj-plus adj-uminus norm-adj uminus-add-conv-diff)
  by (simp add: assms)

lemma adj-minus: ‹(A − B)* = (A*) − (B*)›
  by (metis add-implies-diff adj-plus diff-add-cancel)

lemma cinner-selfadjoint-real: ‹x •_C (A *_V x) ∈ ℝ› if ‹selfadjoint A›
  by (metis Reals-cnj-iff cinner-adj-right cinner-commute' that selfadjoint-def)

lemma adj-inject: ‹adj a = adj b ⟷ a = b›
 by (metis (no-types, opaque-lifting) adj-minus eq-iff-diff-eq-0 norm-adj norm-eq-zero)
```

lemma *norm-AadjA*[*simp*]: ‹*norm* (*A*∗ o_{CL} *A*) = (*norm A*)^2› **for** *A* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*›
by (*metis double-adj norm-AAadj norm-adj*)

lemma *cspan-adj-closed*:
assumes ‹⋀*a*. *a* ∈ *A* ⟹ *a*∗ ∈ *A*›
assumes ‹*a* ∈ *cspan A*›
shows ‹*a*∗ ∈ *cspan A*›
proof −
from ‹*a* ∈ *cspan A*›
obtain *F f* **where** ‹*finite F*› **and** ‹*F* ⊆ *A*› **and** ‹*a* = (∑ *x*∈*F*. *f x* ∗$_C$ *x*)›
by (*smt* (*verit*, *del-insts*) *complex-vector.span-explicit mem-Collect-eq*)
then have ‹*a*∗ = (∑ *x*∈*F*. *cnj* (*f x*) ∗$_C$ *x*∗)›
by (*auto simp*: *sum-adj*)
also have ‹... ∈ *cspan A*›
using *assms* ‹*F* ⊆ *A*›
by (*auto intro*!: *complex-vector.span-sum complex-vector.span-scale simp*: *complex-vector.span-clauses*)
finally show *?thesis*
by −
qed

lemma *cblinfun-norm-is-Sup-cinner*:
— [1], 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* ‹$\psi = 0$›, *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 ‹$\ldots \leq c * (norm\ (h + g))^2 - Re\ ((h - g) \cdot_C A\ (h - g))$›
using *c-upper'*[*of* ‹$h + g$›]
by (*smt* (*verit*, *best*) *complex-Re-le-cmod*)
also have ‹$\ldots \leq c * (norm\ (h + g))^2 + c * (norm\ (h - g))^2$›
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 ‹$\ldots = 2 * c * ((norm\ h)^2 + (norm\ g)^2)$›
by (*auto intro*!: *simp*: *polar-identity polar-identity-minus ring-distribs*)
also have ‹$\ldots \leq 4 * c$›
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* γ = *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 ‹$\ldots = Re\ (g \cdot_C (A\ (h /_C \gamma)))$›
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 ‹... ≤ c›
      using ‹norm γ = 1›
      by (auto intro!: ∗ simp: that norm-inverse)
    finally show ‹cmod (g ∙C (A ∗V h)) ≤ 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
```

13.5 Powers of operators

```
lift-definition cblinfun-power :: ‹'a::complex-normed-vector ⇒CL 'a ⇒ nat ⇒ 'a
⇒CL 'a› is
  ‹λ(a::'a⇒'a) n. a ^^ n›
  apply (rename-tac f n, induct-tac n, auto simp: Nat.funpow-code-def)
```

by (*simp add*: *bounded-clinear-compose*)

lemma *cblinfun-power-0*[*simp*]: ‹*cblinfun-power A 0 = id-cblinfun*›
by *transfer auto*

lemma *cblinfun-power-Suc'*: ‹*cblinfun-power A* (*Suc n*) = *A* o_{CL} *cblinfun-power A n*›
by *transfer auto*

lemma *cblinfun-power-Suc*: ‹*cblinfun-power A* (*Suc n*) = *cblinfun-power A n* o_{CL} *A*›
apply (*induction n*)
by (*auto simp*: *cblinfun-power-Suc' simp flip*: *cblinfun-compose-assoc*)

lemma *cblinfun-power-compose*[*simp*]: ‹*cblinfun-power A n* o_{CL} *cblinfun-power A m* = *cblinfun-power A* (*n*+*m*)›
apply (*induction n*)
by (*auto simp*: *cblinfun-power-Suc' cblinfun-compose-assoc*)

lemma *cblinfun-power-scaleC*: ‹*cblinfun-power* (*c* $*_C$ *a*) *n* = *c*^*n* $*_C$ *cblinfun-power a n*›
apply (*induction n*)
by (*auto simp*: *cblinfun-power-Suc*)

lemma *cblinfun-power-scaleR*: ‹*cblinfun-power* (*c* $*_R$ *a*) *n* = *c*^*n* $*_R$ *cblinfun-power a n*›
apply (*induction n*)
by (*auto simp*: *cblinfun-power-Suc*)

lemma *cblinfun-power-uminus*: ‹*cblinfun-power* (−*a*) *n* = (−*1*)^*n* $*_R$ *cblinfun-power a n*›
apply (*subst asm-rl*[*of* ‹−*a* = (−*1*) $*_R$ *a*›])
by *simp* (*rule cblinfun-power-scaleR*)

lemma *cblinfun-power-adj*: ‹(*cblinfun-power S n*)* = *cblinfun-power* (*S**) *n*›
apply (*induction n*)
apply *simp*
apply (*subst cblinfun-power-Suc*)
apply (*subst cblinfun-power-Suc'*)
by *auto*

13.6 Unitaries / isometries

definition *isometry*::‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*b*::*complex-inner* ⇒ *bool*› **where**
‹*isometry U* ⟷ *U** o_{CL} *U* = *id-cblinfun*›

definition *unitary*::‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*b*::*complex-inner* ⇒ *bool*› **where**
‹*unitary U* ⟷ (*U** o_{CL} *U* = *id-cblinfun*) ∧ (*U* o_{CL} *U** = *id-cblinfun*)›

lemma *unitaryI*: ‹*unitary a*› **if** ‹*a∗ o*$_{CL}$ *a* = *id-cblinfun*› **and** ‹*a o*$_{CL}$ *a∗* = *id-cblinfun*›
unfolding *unitary-def* **using** *that* **by** *simp*

lemma *unitary-twosided-isometry*: *unitary U* ⟷ *isometry U* ∧ *isometry* (*U∗*)
unfolding *unitary-def isometry-def* **by** *simp*

lemma *isometryD*[*simp*]: *isometry U* ⟹ *U∗ o*$_{CL}$ *U* = *id-cblinfun*
unfolding *isometry-def* **by** *simp*

lemma *unitaryD1*: *unitary U* ⟹ *U∗ o*$_{CL}$ *U* = *id-cblinfun*
unfolding *unitary-def* **by** *simp*

lemma *unitaryD2*[*simp*]: *unitary U* ⟹ *U o*$_{CL}$ *U∗* = *id-cblinfun*
unfolding *unitary-def* **by** *simp*

lemma *unitary-isometry*[*simp*]: *unitary U* ⟹ *isometry U*
unfolding *unitary-def isometry-def* **by** *simp*

lemma *unitary-adj*[*simp*]: *unitary* (*U∗*) = *unitary U*
unfolding *unitary-def* **by** *auto*

lemma *isometry-cblinfun-compose*[*simp*]:
assumes *isometry A* **and** *isometry B*
shows *isometry* (*A o*$_{CL}$ *B*)
proof−
have *B∗ o*$_{CL}$ *A∗ o*$_{CL}$ (*A o*$_{CL}$ *B*) = *id-cblinfun* **if** *A∗ o*$_{CL}$ *A* = *id-cblinfun* **and** *B∗ o*$_{CL}$ *B* = *id-cblinfun*
using *that*
by (*smt* (*verit*, *del-insts*) *adjoint-eqI cblinfun-apply-cblinfun-compose cblinfun-id-cblinfun-apply*)
thus *?thesis*
using *assms* **unfolding** *isometry-def* **by** *simp*
qed

lemma *unitary-cblinfun-compose*[*simp*]: *unitary* (*A o*$_{CL}$ *B*)
if *unitary A* **and** *unitary B*
using *that*
by (*smt* (*z3*) *adj-cblinfun-compose cblinfun-compose-assoc cblinfun-compose-id-right double-adj isometryD isometry-cblinfun-compose unitary-def unitary-isometry*)

lemma *unitary-surj*:
assumes *unitary U*
shows *surj* (*cblinfun-apply U*)
apply (*rule surjI*[**where** *f*=‹*cblinfun-apply* (*U∗*)›])
using *assms* **unfolding** *unitary-def* **apply** *transfer*
using *comp-eq-dest-lhs* **by** *force*

lemma *unitary-id*[*simp*]: *unitary id-cblinfun*
by (*simp add*: *unitary-def*)

lemma *orthogonal-on-basis-is-isometry*:
assumes *spanB*: ‹*ccspan B* = ⊤›
assumes *orthoU*: ‹⋀*b c*. *b*∈*B* ⟹ *c*∈*B* ⟹ *cinner* (*U* *$_V$ *b*) (*U* *$_V$ *c*) = *cinner b c*›
shows ‹*isometry U*›
proof −
have [*simp*]: ‹*b* ∈ *closure* (*cspan B*)› **for** *b*
using *spanB* **by** *transfer simp*
have ∗: ‹*cinner* (*U*∗ *$_V$ *U* *$_V$ *ψ*) *φ* = *cinner ψ φ*› **if** ‹*ψ*∈*B*› **and** ‹*φ*∈*B*› **for** *ψ φ*
by (*simp add*: *cinner-adj-left orthoU that*(*1*) *that*(*2*))
have ∗: ‹*cinner* (*U*∗ *$_V$ *U* *$_V$ *ψ*) *φ* = *cinner ψ φ*› **if** ‹*ψ*∈*B*› **for** *ψ φ*
apply (*rule bounded-clinear-eq-on-closure*[**where** *t*=*φ* **and** *G*=*B*])
using *bounded-clinear-cinner-right* ∗[*OF that*]
by *auto*
have ‹*U*∗ *$_V$ *U* *$_V$ *φ* = *φ*› **if** ‹*φ*∈*B*› **for** *φ*
apply (*rule cinner-extensionality*)
apply (*subst cinner-eq-flip*)
by (*simp add*: ∗ *that*)
then have ‹*U*∗ *o$_{CL}$ *U* = *id-cblinfun*›
by (*metis cblinfun-apply-cblinfun-compose cblinfun-eq-gen-eqI cblinfun-id-cblinfun-apply spanB*)
then show ‹*isometry U*›
using *isometry-def* **by** *blast*
qed

lemma *isometry-preserves-norm*: ‹*isometry U* ⟹ *norm* (*U* *$_V$ *ψ*) = *norm ψ*›
by (*metis* (*no-types*, *lifting*) *cblinfun-apply-cblinfun-compose cblinfun-id-cblinfun-apply cinner-adj-right cnorm-eq isometryD*)

lemma *norm-isometry-compose*:
assumes ‹*isometry U*›
shows ‹*norm* (*U* *o$_{CL}$ *A*) = *norm A*›
proof −
have ∗: ‹*norm* (*U* *$_V$ *A* *$_V$ *ψ*) = *norm* (*A* *$_V$ *ψ*)› **for** *ψ*
by (*smt* (*verit*, *ccfv-threshold*) *assms cblinfun-apply-cblinfun-compose cinner-adj-right cnorm-eq id-cblinfun-apply isometryD*)

have ‹*norm* (*U* *o$_{CL}$ *A*) = (*SUP ψ*. *norm* (*U* *$_V$ *A* *$_V$ *ψ*) / *norm ψ*)›
unfolding *norm-cblinfun-Sup* **by** *auto*
also have ‹... = (*SUP ψ*. *norm* (*A* *$_V$ *ψ*) / *norm ψ*)›
using ∗ **by** *auto*
also have ‹... = *norm A*›
unfolding *norm-cblinfun-Sup* **by** *auto*
finally show *?thesis*
by *simp*

qed

lemma *norm-isometry*:
fixes $U :: \langle 'a::\{chilbert\text{-}space, not\text{-}singleton\} \Rightarrow_{CL} 'b::complex\text{-}inner\rangle$
assumes ‹*isometry U*›
shows ‹*norm U = 1*›
apply (*subst asm-rl*[*of* ‹$U = U\ o_{CL}\ id\text{-}cblinfun$›], *simp*)
apply (*subst norm-isometry-compose*, *simp add*: *assms*)
by *simp*

lemma *norm-preserving-isometry*: ‹*isometry U*› **if** ‹$\bigwedge\psi.\ norm\ (U *_V \psi) = norm\ \psi$›
by (*smt* (*verit*, *ccfv-SIG*) *cblinfun-cinner-eqI cblinfun-id-cblinfun-apply cinner-adj-right cnorm-eq isometry-def simp-a-oCL-b' that*)

lemma *norm-isometry-compose'*: ‹*norm* ($A\ o_{CL}\ U$) = *norm A*› **if** ‹*isometry* ($U*$)›
by (*smt* (*verit*) *cblinfun-compose-assoc cblinfun-compose-id-right double-adj isometryD mult-cancel-left2 norm-AadjA norm-cblinfun-compose norm-isometry-compose norm-zero power2-eq-square right-diff-distrib that zero-less-norm-iff*)

lemma *unitary-nonzero*[*simp*]: ‹¬ *unitary* (*0* :: $'a::\{chilbert\text{-}space, not\text{-}singleton\} \Rightarrow_{CL}$ -)›
by (*simp add*: *unitary-def*)

lemma *isometry-inj*:
assumes ‹*isometry U*›
shows ‹*inj-on U X*›
apply (*rule inj-on-inverseI*[**where** *g*=‹$U*$›])
using *assms* **by** (*simp flip*: *cblinfun-apply-cblinfun-compose*)

lemma *unitary-inj*:
assumes ‹*unitary U*›
shows ‹*inj-on U X*›
apply (*rule isometry-inj*)
using *assms* **by** *simp*

lemma *unitary-adj-inv*: ‹*unitary U* ⟹ *cblinfun-apply* ($U*$) = *inv* (*cblinfun-apply U*)›
apply (*rule inj-imp-inv-eq*[*symmetric*])
apply (*simp add*: *unitary-inj*)
unfolding *unitary-def*
by (*simp flip*: *cblinfun-apply-cblinfun-compose*)

lemma *isometry-cinner-both-sides*:
assumes ‹*isometry U*›
shows ‹*cinner* (*U x*) (*U y*) = *cinner x y*›
using *assms* **by** (*simp add*: *flip*: *cinner-adj-right cblinfun-apply-cblinfun-compose*)

lemma *isometry-image-is-ortho-set*:

assumes ‹*is-ortho-set A*›
assumes ‹*isometry U*›
shows ‹*is-ortho-set* (*U* ‘ *A*)›
using *assms* **apply** (*auto simp add*: *is-ortho-set-def isometry-cinner-both-sides*)
by (*metis cinner-eq-zero-iff isometry-cinner-both-sides*)

13.7 Product spaces

lift-definition *cblinfun-left* :: ‹'*a*::*complex-normed-vector* ⇒$_{CL}$ ('*a*×'*b*::*complex-normed-vector*)›
is ‹(λ*x*. (*x*,*0*))›
by (*auto intro*!: *bounded-clinearI*[**where** *K*=*1*])
lift-definition *cblinfun-right* :: ‹'*b*::*complex-normed-vector* ⇒$_{CL}$ ('*a*::*complex-normed-vector*×'*b*)›
is ‹(λ*x*. (*0*,*x*))›
by (*auto intro*!: *bounded-clinearI*[**where** *K*=*1*])

lemma *isometry-cblinfun-left*[*simp*]: ‹*isometry cblinfun-left*›
apply (*rule orthogonal-on-basis-is-isometry*[*of some-chilbert-basis*])
apply *simp*
by *transfer simp*

lemma *isometry-cblinfun-right*[*simp*]: ‹*isometry cblinfun-right*›
apply (*rule orthogonal-on-basis-is-isometry*[*of some-chilbert-basis*])
apply *simp*
by *transfer simp*

lemma *cblinfun-left-right-ortho*[*simp*]: ‹*cblinfun-left*∗ *o*$_{CL}$ *cblinfun-right* = *0*›
proof −
have ‹*x* •$_C$ ((*cblinfun-left*∗ *o*$_{CL}$ *cblinfun-right*) ∗$_V$ *y*) = *0*› **for** *x* :: '*b* **and** *y* :: '*a*
apply (*simp add*: *cinner-adj-right*)
by *transfer auto*
then show *?thesis*
by (*metis cblinfun.zero-left cblinfun-eqI cinner-eq-zero-iff*)
qed

lemma *cblinfun-right-left-ortho*[*simp*]: ‹*cblinfun-right*∗ *o*$_{CL}$ *cblinfun-left* = *0*›
proof −
have ‹*x* •$_C$ ((*cblinfun-right*∗ *o*$_{CL}$ *cblinfun-left*) ∗$_V$ *y*) = *0*› **for** *x* :: '*b* **and** *y* :: '*a*
apply (*simp add*: *cinner-adj-right*)
by *transfer auto*
then show *?thesis*
by (*metis cblinfun.zero-left cblinfun-eqI cinner-eq-zero-iff*)
qed

lemma *cblinfun-left-apply*[*simp*]: ‹*cblinfun-left* ∗$_V$ ψ = (ψ,*0*)›
by *transfer simp*

lemma *cblinfun-left-adj-apply*[*simp*]: ‹*cblinfun-left*∗ ∗$_V$ ψ = *fst* ψ›
apply (*cases* ψ)
by (*auto intro*!: *cinner-extensionality*[*of* ‹- ∗$_V$ -›] *simp*: *cinner-adj-right*)

lemma *cblinfun-right-apply*[*simp*]: ‹*cblinfun-right* $*_V$ ψ = (*0*,ψ)›
by *transfer simp*

lemma *cblinfun-right-adj-apply*[*simp*]: ‹*cblinfun-right** $*_V$ ψ = *snd* ψ›
apply (*cases* ψ)
by (*auto intro*!: *cinner-extensionality*[*of* ‹- $*_V$ -›] *simp*: *cinner-adj-right*)

lift-definition *ccsubspace-Times* :: ‹'*a*::*complex-normed-vector ccsubspace* ⇒ '*b*::*complex-normed-vector ccsubspace* ⇒ ('*a*×'*b*) *ccsubspace*› **is**
‹λ*S T*. *S* × *T*›
proof −
fix *S* :: ‹'*a set*› **and** *T* :: ‹'*b set*›
assume [*simp*]: ‹*closed-csubspace S*› ‹*closed-csubspace T*›
have ‹*csubspace* (*S* × *T*)›
by (*simp add*: *complex-vector.subspace-Times*)
moreover have ‹*closed* (*S* × *T*)›
by (*simp add*: *closed-Times closed-csubspace.closed*)
ultimately show ‹*closed-csubspace* (*S* × *T*)›
by (*rule closed-csubspace.intro*)
qed

lemma *ccspan-Times*: ‹*ccspan* (*S* × *T*) = *ccsubspace-Times* (*ccspan S*) (*ccspan T*)› **if** ‹*0* ∈ *S*› **and** ‹*0* ∈ *T*›
proof (*transfer fixing*: *S T*)
from *that* **have** ‹*closure* (*cspan* (*S* × *T*)) = *closure* (*cspan S* × *cspan T*)›
by (*simp add*: *cspan-Times*)
also have ‹... = *closure* (*cspan S*) × *closure* (*cspan T*)›
using *closure-Times* **by** *blast*
finally show ‹*closure* (*cspan* (*S* × *T*)) = *closure* (*cspan S*) × *closure* (*cspan T*)›
by −
qed

lemma *ccspan-Times-sing1*: ‹*ccspan* ({*0*::'*a*::*complex-normed-vector*} × *B*) = *ccsubspace-Times 0* (*ccspan B*)›
proof (*transfer fixing*: *B*)
have ‹*closure* (*cspan* ({*0*::'*a*} × *B*)) = *closure* ({*0*} × *cspan B*)›
by (*simp add*: *complex-vector.span-Times-sing1*)
also have ‹... = *closure* {*0*} × *closure* (*cspan B*)›
using *closure-Times* **by** *blast*
also have ‹... = {*0*} × *closure* (*cspan B*)›
by *simp*
finally show ‹*closure* (*cspan* ({*0*::'*a*} × *B*)) = {*0*} × *closure* (*cspan B*)›
by −
qed

lemma *ccspan-Times-sing2*: ‹*ccspan* (*B* × {*0*::'*a*::*complex-normed-vector*}) = *cc-*

subspace-Times (*ccspan B*) *0*›
proof (*transfer fixing*: *B*)
have ‹*closure* (*cspan* (*B* × {*0*::*'a*})) = *closure* (*cspan B* × {*0*})›
by (*simp add*: *complex-vector.span-Times-sing2*)
also have ‹... = *closure* (*cspan B*) × *closure* {*0*}›
using *closure-Times* **by** *blast*
also have ‹... = *closure* (*cspan B*) × {*0*}›
by *simp*
finally show ‹*closure* (*cspan* (*B* × {*0*::*'a*})) = *closure* (*cspan B*) × {*0*}›
by −
qed

lemma *ccsubspace-Times-sup*: ‹*sup* (*ccsubspace-Times A B*) (*ccsubspace-Times C D*) = *ccsubspace-Times* (*sup A C*) (*sup B D*)›
proof *transfer*
fix *A C* :: ‹*'a set*› **and** *B D* :: ‹*'b set*›
have ‹*A* × *B* +$_M$ *C* × *D* = *closure* ((*A* × *B*) + (*C* × *D*))›
using *closed-sum-def* **by** *blast*
also have ‹... = *closure* ((*A* + *C*) × (*B* + *D*))›
by (*simp add*: *set-Times-plus-distrib*)
also have ‹... = *closure* (*A* + *C*) × *closure* (*B* + *D*)›
by (*simp add*: *closure-Times*)
also have ‹... = (*A* +$_M$ *C*) × (*B* +$_M$ *D*)›
by (*simp add*: *closed-sum-def*)
finally show ‹*A* × *B* +$_M$ *C* × *D* = (*A* +$_M$ *C*) × (*B* +$_M$ *D*)›
by −
qed

lemma *ccsubspace-Times-top-top*[*simp*]: ‹*ccsubspace-Times top top* = *top*›
by *transfer simp*

lemma *is-ortho-set-prod*:
assumes ‹*is-ortho-set B*› ‹*is-ortho-set B'*›
shows ‹*is-ortho-set* ((*B* × {*0*}) ∪ ({*0*} × *B'*))›
using *assms* **unfolding** *is-ortho-set-def*
apply (*auto simp*: *is-onb-def is-ortho-set-def zero-prod-def*)
by (*meson is-onb-def is-ortho-set-def*)+

lemma *ccsubspace-Times-ccspan*:
assumes ‹*ccspan B* = *S*› **and** ‹*ccspan B'* = *S'*›
shows ‹*ccspan* ((*B* × {*0*}) ∪ ({*0*} × *B'*)) = *ccsubspace-Times S S'*›
by (*smt* (*z3*) *Diff-eq-empty-iff Sigma-cong assms*(*1*) *assms*(*2*) *ccspan.rep-eq ccspan-0 ccspan-Times-sing1 ccspan-Times-sing2 ccspan-of-empty ccspan-remove-0 ccspan-superset ccspan-union ccsubspace-Times-sup complex-vector.span-insert-0 space-as-set-bot sup-bot-left sup-bot-right*)

lemma *is-onb-prod*:
assumes ‹*is-onb B*› ‹*is-onb B'*›
shows ‹*is-onb* ((*B* × {*0*}) ∪ ({*0*} × *B'*))›

using *assms* **by** (*auto intro*!: *is-ortho-set-prod simp add*: *is-onb-def ccsubspace-Times-ccspan*)

13.8 Images

The following definition defines the image of a closed subspace S under a bounded operator A. We do not define that image as the image of A seen as a function (A ‘ S) but as the topological closure of that image. This is because A ‘ S might in general not be closed.

For example, if e_i ($i \in \mathbb{N}$) form an orthonormal basis, and A maps e_i to e_i/i, then all e_i are in A ‘ S, so the closure of A ‘ S is the whole space. However, $\sum_i e_i/i$ is not in A ‘ S because its preimage would have to be $\sum_i e_i$ which does not converge. So A ‘ S does not contain the whole space, hence it is not closed.

lift-definition *cblinfun-image* :: ‹'*a*::*complex-normed-vector* ⇒$_{CL}$ '*b*::*complex-normed-vector* ⇒ '*a ccsubspace* ⇒ '*b ccsubspace*› (**infixr** ‹*$_S$› *70*)
is λ*A S*. *closure* (*A* ‘ *S*)
using *bounded-clinear-def closed-closure closed-csubspace.intro*
by (*simp add*: *bounded-clinear-def complex-vector.linear-subspace-image closure-is-closed-csubspace*)

lemma *cblinfun-image-mono*:
assumes *a1*: *S* ≤ *T*
shows *A* *$_S$ *S* ≤ *A* *$_S$ *T*
using *a1*
by (*simp add*: *cblinfun-image.rep-eq closure-mono image-mono less-eq-ccsubspace.rep-eq*)

lemma *cblinfun-image-0*[*simp*]:
shows *U* *$_S$ *0* = *0*
thm *zero-ccsubspace-def*
by *transfer* (*simp add*: *bounded-clinear-def complex-vector.linear-0*)

lemma *cblinfun-image-bot*[*simp*]: *U* *$_S$ *bot* = *bot*
using *cblinfun-image-0* **by** *auto*

lemma *cblinfun-image-sup*[*simp*]:
fixes *A B* :: ‹'*a*::*chilbert-space ccsubspace*› **and** *U* :: '*a* ⇒$_{CL}$ '*b*::*chilbert-space*
shows ‹*U* *$_S$ (*sup A B*) = *sup* (*U* *$_S$ *A*) (*U* *$_S$ *B*)›
apply *transfer* **using** *bounded-clinear.bounded-linear closure-image-closed-sum*
by *blast*

lemma *scaleC-cblinfun-image*[*simp*]:
fixes *A* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b* :: *chilbert-space*›
and *S* :: ‹'*a ccsubspace*› **and** α :: *complex*
shows ‹(α *$_C$ *A*) *$_S$ *S* = α *$_C$ (*A* *$_S$ *S*)›
proof−
have ‹*closure* ((((*$_C$) α) ∘ (*cblinfun-apply A*)) ‘ *space-as-set S*) =
((*$_C$) α) ‘ (*closure* (*cblinfun-apply A* ‘ *space-as-set S*))›
by (*metis closure-scaleC image-comp*)
hence ‹(*closure* (*cblinfun-apply* (α *$_C$ *A*) ‘ *space-as-set S*)) =

```
    ((*C) α) ‘ (closure (cblinfun-apply A ‘ space-as-set S))›
    by (metis (mono-tags, lifting) comp-apply image-cong scaleC-cblinfun.rep-eq)
  hence ‹Abs-ccsubspace (closure (cblinfun-apply (α *C A) ‘ space-as-set S)) =
          α *C Abs-ccsubspace (closure (cblinfun-apply A ‘ space-as-set S))›
    by (metis space-as-set-inverse cblinfun-image.rep-eq scaleC-ccsubspace.rep-eq)
  have x1: Abs-ccsubspace (closure ((*V) (α *C A) ‘ space-as-set S)) =
          α *C Abs-ccsubspace (closure ((*V) A ‘ space-as-set S))
    using ‹Abs-ccsubspace (closure (cblinfun-apply (α *C A) ‘ space-as-set S)) =
          α *C Abs-ccsubspace (closure (cblinfun-apply A ‘ space-as-set S))›
    by blast
  show ?thesis
    unfolding cblinfun-image-def using x1 by force
qed

lemma cblinfun-image-id[simp]:
  id-cblinfun *S ψ = ψ
  by transfer (simp add: closed-csubspace.closed)

lemma cblinfun-compose-image:
  ‹(A oCL B) *S S = A *S (B *S S)›
  apply transfer unfolding image-comp[symmetric]
  apply (rule closure-bounded-linear-image-subset-eq[symmetric])
  by (simp add: bounded-clinear.bounded-linear)

lemmas cblinfun-assoc-left = cblinfun-compose-assoc[symmetric] cblinfun-compose-image[symmetric]
  add.assoc[where ?'a='a::chilbert-space ⇒CL 'b::chilbert-space, symmetric]
lemmas cblinfun-assoc-right = cblinfun-compose-assoc cblinfun-compose-image
  add.assoc[where ?'a='a::chilbert-space ⇒CL 'b::chilbert-space]

lemma cblinfun-image-INF-leq[simp]:
  fixes U :: 'b::complex-normed-vector ⇒CL 'c::complex-normed-vector
    and V :: 'a ⇒ 'b ccsubspace
  shows ‹U *S (INF i∈X. V i) ≤ (INF i∈X. U *S (V i))›
  apply transfer
  by (simp add: INT-greatest Inter-lower closure-mono image-mono)

lemma isometry-cblinfun-image-inf-distrib':
  fixes U::‹'a::complex-normed-vector ⇒CL 'b::cbanach› and B C::'a ccsubspace
  shows U *S (inf B C) ≤ inf (U *S B) (U *S C)
proof −
  define V where ‹V b = (if b then B else C)› for b
  have ‹U *S (INF i. V i) ≤ (INF i. U *S (V i))›
    by auto
  then show ?thesis
    unfolding V-def
    by (metis (mono-tags, lifting) INF-UNIV-bool-expand)
qed

lemma cblinfun-image-eq:
```

fixes $S :: 'a::cbanach\ ccsubspace$
and $A\ B :: 'a::cbanach \Rightarrow_{CL} 'b::cbanach$
assumes $\bigwedge x.\ x \in G \Longrightarrow A *_V x = B *_V x$ **and** $ccspan\ G \geq S$
shows $A *_S S = B *_S S$
proof (*use assms* **in** *transfer*)
fix $G :: 'a\ set$ **and** $A :: 'a \Rightarrow 'b$ **and** $B :: 'a \Rightarrow 'b$ **and** $S :: 'a\ set$
assume *a1*: *bounded-clinear A*
assume *a2*: *bounded-clinear B*
assume *a3*: $\bigwedge x.\ x \in G \Longrightarrow A\ x = B\ x$
assume *a4*: $S \subseteq closure\ (cspan\ G)$

have $A\ `\ closure\ S = B\ `\ closure\ S$
by (*smt* (*verit*, *best*) *UnCI a1 a2 a3 a4 bounded-clinear-eq-on-closure closure-Un closure-closure image-cong sup.absorb-iff1*)
then show $closure\ (A\ `\ S) = closure\ (B\ `\ S)$
by (*metis bounded-clinear.bounded-linear a1 a2 closure-bounded-linear-image-subset-eq*)
qed

lemma *cblinfun-fixes-range*:
assumes $A\ o_{CL}\ B = B$ **and** $\psi \in space\text{-}as\text{-}set\ (B *_S top)$
shows $A *_V \psi = \psi$
proof−
define *rangeB rangeB'* **where** $rangeB = space\text{-}as\text{-}set\ (B *_S top)$
and $rangeB' = range\ (cblinfun\text{-}apply\ B)$
from *assms* **have** $\psi \in closure\ rangeB'$
by (*simp add*: *cblinfun-image.rep-eq rangeB'-def top-ccsubspace.rep-eq*)

then obtain ψi **where** $\psi i\text{-}lim$: $\psi i \longrightarrow \psi$ **and** $\psi i\text{-}B$: $\psi i\ i \in rangeB'$ **for** i
using *closure-sequential* **by** *blast*
have *A-invariant*: $A *_V \psi i\ i = \psi i\ i$
for i
proof−
from $\psi i\text{-}B$ **obtain** φ **where** φ: $\psi i\ i = B *_V \varphi$
using *rangeB'-def* **by** *blast*
hence $A *_V \psi i\ i = (A\ o_{CL}\ B) *_V \varphi$
by (*simp add*: *cblinfun-compose.rep-eq*)
also have $\ldots = B *_V \varphi$
by (*simp add*: *assms*)
also have $\ldots = \psi i\ i$
by (*simp add*: φ)
finally show *?thesis*.
qed
from $\psi i\text{-}lim$ **have** $(\lambda i.\ A *_V (\psi i\ i)) \longrightarrow A *_V \psi$
by (*rule isCont-tendsto-compose*[*rotated*], *simp*)
with *A-invariant* **have** $(\lambda i.\ \psi i\ i) \longrightarrow A *_V \psi$
by *auto*
with $\psi i\text{-}lim$ **show** $A *_V \psi = \psi$
using *LIMSEQ-unique* **by** *blast*
qed

lemma *zero-cblinfun-image*[*simp*]: $0 *_S S = (0 ::- ccsubspace)$
by *transfer* (*simp add*: *complex-vector.subspace-0 image-constant*[**where** *x*=*0*])

lemma *cblinfun-image-INF-eq-general*:
fixes $V :: 'a \Rightarrow 'b :: chilbert\text{-}space\ ccsubspace$
and $U :: 'b \Rightarrow_{CL} 'c :: chilbert\text{-}space$
and $Uinv :: 'c \Rightarrow_{CL} 'b$
assumes *UinvUUinv*: $Uinv\ o_{CL}\ U\ o_{CL}\ Uinv = Uinv$ **and** *UUinvU*: $U\ o_{CL}\ Uinv\ o_{CL}\ U = U$
— Meaning: *Uinv* is a Pseudoinverse of *U*
and *V*: $\bigwedge i.\ V\ i \leq Uinv *_S top$
and $\langle X \neq \{\} \rangle$
shows $U *_S (INF\ i{\in}X.\ V\ i) = (INF\ i{\in}X.\ U *_S V\ i)$
proof (*rule antisym*)
show $U *_S (INF\ i{\in}X.\ V\ i) \leq (INF\ i{\in}X.\ U *_S V\ i)$
by (*rule cblinfun-image-INF-leq*)
next
define *rangeU rangeUinv* **where** $rangeU = U *_S top$ **and** $rangeUinv = Uinv *_S top$
define *INFUV INFV* **where** *INFUV-def*: $INFUV = (INF\ i{\in}X.\ U *_S V\ i)$ **and** *INFV-def*: $INFV = (INF\ i{\in}X.\ V\ i)$
from *assms* **have** $V\ i \leq rangeUinv$
for *i*
unfolding *rangeUinv-def* **by** *simp*
moreover have $(Uinv\ o_{CL}\ U) *_V \psi = \psi$ **if** $\psi \in space\text{-}as\text{-}set\ rangeUinv$
for ψ
using *UinvUUinv cblinfun-fixes-range rangeUinv-def that* **by** *fastforce*
ultimately have $(Uinv\ o_{CL}\ U) *_V \psi = \psi$ **if** $\psi \in space\text{-}as\text{-}set\ (V\ i)$
for $\psi\ i$
using *less-eq-ccsubspace.rep-eq that* **by** *blast*
hence *d1*: $(Uinv\ o_{CL}\ U) *_S (V\ i) = (V\ i)$ **for** *i*
proof (*transfer fixing*: *i*)
fix $V :: 'a \Rightarrow 'b\ set$
and $Uinv :: 'c \Rightarrow 'b$
and $U :: 'b \Rightarrow 'c$
assume *pred-fun* $\top$ *closed-csubspace V*
and *bounded-clinear Uinv*
and *bounded-clinear U*
and $\bigwedge \psi\ i.\ \psi \in V\ i \Longrightarrow (Uinv \circ U)\ \psi = \psi$
then show $closure\ ((Uinv \circ U)\ `\ V\ i) = V\ i$
proof *auto*
fix *x*
from $\langle pred\text{-}fun\ \top\ closed\text{-}csubspace\ V \rangle$
show $x \in V\ i$
if $x \in closure\ (V\ i)$
using *that* **apply** *simp*
by (*metis orthogonal-complement-of-closure closed-csubspace.subspace double-orthogonal-complement-id closure-is-closed-csubspace*)

```
    with ‹pred-fun ⊤ closed-csubspace V›
    show x ∈ closure (V i)
      if x ∈ V i
      using that
      using setdist-eq-0-sing-1 setdist-sing-in-set
      by blast
  qed
 qed
 have U *S V i ≤ rangeU for i
  by (simp add: cblinfun-image-mono rangeU-def)
 hence INFUV ≤ rangeU
  unfolding INFUV-def using ‹X ≠ {}›
  by (metis INF-eq-const INF-lower2)
 moreover have (U oCL Uinv) *V ψ = ψ if ψ ∈ space-as-set rangeU for ψ
  using UUinvU cblinfun-fixes-range rangeU-def that by fastforce
 ultimately have x: (U oCL Uinv) *V ψ = ψ if ψ ∈ space-as-set INFUV for ψ
  by (simp add: in-mono less-eq-ccsubspace.rep-eq that)

 have closure ((U ∘ Uinv) ‘ INFUV) = INFUV
  if closed-csubspace INFUV
    and bounded-clinear U
    and bounded-clinear Uinv
    and ⋀ψ. ψ ∈ INFUV ⟹ (U ∘ Uinv) ψ = ψ
  for INFUV :: 'c set
  using that
 proof auto
  fix x
  show x ∈ INFUV if x ∈ closure INFUV
    using that ‹closed-csubspace INFUV›
    by (metis orthogonal-complement-of-closure closed-csubspace.subspace dou-
ble-orthogonal-complement-id closure-is-closed-csubspace)
  show x ∈ closure INFUV
    if x ∈ INFUV
    using that ‹closed-csubspace INFUV›
    using setdist-eq-0-sing-1 setdist-sing-in-set
    by (simp add: closed-csubspace.closed)
 qed
 hence (U oCL Uinv) *S INFUV = INFUV
  by (metis (mono-tags, opaque-lifting) x cblinfun-image.rep-eq cblinfun-image-id
id-cblinfun-apply image-cong
      space-as-set-inject)
 hence INFUV = U *S Uinv *S INFUV
  by (simp add: cblinfun-compose-image)
 also have ... ≤ U *S (INF i∈X. Uinv *S U *S V i)
  unfolding INFUV-def
  by (metis cblinfun-image-mono cblinfun-image-INF-leq)
 also have ... = U *S INFV
  using d1
  by (metis (no-types, lifting) INFV-def cblinfun-assoc-left(2) image-cong)
```

finally show $INFUV \leq U *_S INFV$.
qed

lemma *unitary-range*[*simp*]:
assumes *unitary U*
shows $U *_S top = top$
using *assms* **unfolding** *unitary-def* **by** *transfer* (*metis closure-UNIV comp-apply surj-def*)

lemma *range-adjoint-isometry*:
assumes *isometry U*
shows $U* *_S top = top$
proof−
from *assms* **have** $top = U* *_S U *_S top$
by (*simp add*: *cblinfun-assoc-left*(*2*))
also have $\ldots \leq U* *_S top$
by (*simp add*: *cblinfun-image-mono*)
finally show *?thesis*
using *top.extremum-unique* **by** *blast*
qed

lemma *cblinfun-image-INF-eq*[*simp*]:
fixes $V :: 'a \Rightarrow 'b$::*chilbert-space ccsubspace*
and $U :: 'b \Rightarrow_{CL} 'c$::*chilbert-space*
assumes ‹*isometry U*› ‹$X \neq \{\}$›
shows $U *_S (INF\ i{\in}X.\ V\ i) = (INF\ i{\in}X.\ U *_S V\ i)$
proof −
from ‹*isometry U*› **have** $U* \circ_{CL} U \circ_{CL} U* = U*$
unfolding *isometry-def* **by** *simp*
moreover from ‹*isometry U*› **have** $U \circ_{CL} U* \circ_{CL} U = U$
unfolding *isometry-def*
by (*simp add*: *cblinfun-compose-assoc*)
moreover have $V\ i \leq U* *_S top$ **for** *i*
by (*simp add*: *range-adjoint-isometry assms*)
ultimately show *?thesis*
using ‹$X \neq \{\}$› **by** (*rule cblinfun-image-INF-eq-general*)
qed

lemma *isometry-cblinfun-image-inf-distrib*[*simp*]:
fixes U::‹$'a$::*chilbert-space* $\Rightarrow_{CL} 'b$::*chilbert-space*›
and $X\ Y$::$'a$ *ccsubspace*
assumes *isometry U*
shows $U *_S (inf\ X\ Y) = inf\ (U *_S X)\ (U *_S Y)$
using *cblinfun-image-INF-eq*[**where** $V{=}\lambda b.$ *if b then X else Y* **and** $U{=}U$ **and** $X{=}UNIV$]
unfolding *INF-UNIV-bool-expand*
using *assms* **by** *auto*

lemma *cblinfun-image-ccspan*:

shows $A *_S$ *ccspan* $G =$ *ccspan* $((*_V)\ A\ `\ G)$
by *transfer* (*simp add*: *bounded-clinear.bounded-linear bounded-clinear-def closure-bounded-linear-image-subset-eq complex-vector.linear-span-image*)

lemma *cblinfun-apply-in-image*[*simp*]: $A *_V \psi \in$ *space-as-set* $(A *_S \top)$
by (*metis cblinfun-image.rep-eq closure-subset in-mono range-eqI top-ccsubspace.rep-eq*)

lemma *cblinfun-plus-image-distr*:
$\langle (A + B) *_S S \leq A *_S S \sqcup B *_S S \rangle$
by *transfer* (*smt* (*verit*, *ccfv-threshold*) *closed-closure closed-sum-def closure-minimal closure-subset image-subset-iff set-plus-intro subset-eq*)

lemma *cblinfun-sum-image-distr*:
$\langle (\sum i{\in}I.\ A\ i) *_S S \leq (SUP\ i{\in}I.\ A\ i *_S S) \rangle$
proof (*cases* ‹*finite I*›)
case *True*
then show *?thesis*
proof *induction*
case *empty*
then show *?case*
by *auto*
next
case (*insert x F*)
then show *?case*
by *auto* (*smt* (*z3*) *cblinfun-plus-image-distr inf-sup-aci*(*6*) *le-iff-sup*)
qed
next
case *False*
then show *?thesis*
by *auto*
qed

lemma *space-as-set-image-commute*:
assumes *UV*: ‹$U\ o_{CL}\ V =$ *id-cblinfun*› **and** *VU*: ‹$V\ o_{CL}\ U =$ *id-cblinfun*›

shows $\langle (*_V)\ U\ `$ *space-as-set* $T =$ *space-as-set* $(U *_S T) \rangle$
proof −
have ‹*space-as-set* $(U *_S T) = U\ `\ V\ `$ *space-as-set* $(U *_S T)$›
by (*simp add*: *image-image UV flip*: *cblinfun-apply-cblinfun-compose*)
also have ‹$\ldots \subseteq U\ `$ *space-as-set* $(V *_S U *_S T)$›
by (*metis cblinfun-image.rep-eq closure-subset image-mono*)
also have ‹$\ldots = U\ `$ *space-as-set* T›
by (*simp add*: *VU cblinfun-assoc-left*(*2*))
finally have *1*: ‹*space-as-set* $(U *_S T) \subseteq U\ `$ *space-as-set* T›
by −
have *2*: ‹$U\ `$ *space-as-set* $T \subseteq$ *space-as-set* $(U *_S T)$›
by (*simp add*: *cblinfun-image.rep-eq closure-subset*)
from *1 2* **show** *?thesis*

by *simp*
qed

lemma *right-total-rel-ccsubspace*:
fixes $R :: \langle 'a::complex\text{-}normed\text{-}vector \Rightarrow 'b::complex\text{-}normed\text{-}vector \Rightarrow bool\rangle$
assumes UV: $\langle U\ o_{CL}\ V = id\text{-}cblinfun\rangle$
assumes VU: $\langle V\ o_{CL}\ U = id\text{-}cblinfun\rangle$
assumes *R-def*: $\langle \bigwedge x\ y.\ R\ x\ y \longleftrightarrow x = U\ *_V\ y\rangle$
shows $\langle right\text{-}total\ (rel\text{-}ccsubspace\ R)\rangle$
proof (*rule right-totalI*)
fix $T :: \langle 'b\ ccsubspace\rangle$
show $\langle \exists S.\ rel\text{-}ccsubspace\ R\ S\ T\rangle$
apply (*rule exI*[*of* - $\langle U\ *_S\ T\rangle$])
using *UV VU* **by** (*auto simp add*: *rel-ccsubspace-def R-def rel-set-def simp flip*: *space-as-set-image-commute*)
qed

lemma *left-total-rel-ccsubspace*:
fixes $R :: \langle 'a::complex\text{-}normed\text{-}vector \Rightarrow 'b::complex\text{-}normed\text{-}vector \Rightarrow bool\rangle$
assumes UV: $\langle U\ o_{CL}\ V = id\text{-}cblinfun\rangle$
assumes VU: $\langle V\ o_{CL}\ U = id\text{-}cblinfun\rangle$
assumes *R-def*: $\langle \bigwedge x\ y.\ R\ x\ y \longleftrightarrow y = U\ *_V\ x\rangle$
shows $\langle left\text{-}total\ (rel\text{-}ccsubspace\ R)\rangle$
proof −
have $\langle right\text{-}total\ (rel\text{-}ccsubspace\ (conversep\ R))\rangle$
apply (*rule right-total-rel-ccsubspace*)
using *assms* **by** *auto*
then show *?thesis*
by (*auto intro*!: *right-total-conversep*[*THEN iffD1*] *simp*: *converse-rel-ccsubspace*)
qed

lemma *cblinfun-image-bot-zero*[*simp*]: $\langle A\ *_S\ top = bot \longleftrightarrow A = 0\rangle$
by (*metis Complex-Bounded-Linear-Function.zero-cblinfun-image bot-ccsubspace.rep-eq cblinfun-apply-in-image cblinfun-eqI empty-iff insert-iff zero-ccsubspace-def*)

lemma *surj-isometry-is-unitary*:
— This lemma is a bit stronger than its name suggests: We actually only require that the image of U is dense.
The converse is *unitary-surj*
fixes $U :: \langle 'a::chilbert\text{-}space \Rightarrow_{CL} 'b::chilbert\text{-}space\rangle$
assumes $\langle isometry\ U\rangle$
assumes $\langle U\ *_S\ \top = \top\rangle$
shows $\langle unitary\ U\rangle$
by (*metis UNIV-I assms*(*1*) *assms*(*2*) *cblinfun-assoc-left*(*1*) *cblinfun-compose-id-right cblinfun-eqI cblinfun-fixes-range id-cblinfun-apply isometry-def space-as-set-top unitary-def*)

lemma *cblinfun-apply-in-image'*: $A\ *_V\ \psi \in space\text{-}as\text{-}set\ (A\ *_S\ S)$ **if** $\langle \psi \in space\text{-}as\text{-}set\ S\rangle$

by (*metis cblinfun-image.rep-eq closure-subset image-subset-iff that*)

lemma *cblinfun-image-ccspan-leqI*:
assumes ‹$\bigwedge v.\ v \in M \Longrightarrow A *_V v \in$ *space-as-set* T›
shows ‹$A *_S$ *ccspan* $M \leq T$›
by (*simp add*: *assms cblinfun-image-ccspan ccspan-leqI image-subsetI*)

lemma *cblinfun-same-on-image*: ‹$A\ \psi = B\ \psi$› **if** *eq*: ‹$A\ o_{CL}\ C = B\ o_{CL}\ C$› **and** *mem*: ‹$\psi \in$ *space-as-set* $(C *_S \top)$›
proof −
have ‹$A\ \psi = B\ \psi$› **if** ‹$\psi \in$ *range* C› **for** ψ
by (*metis* (*no-types*, *lifting*) *eq cblinfun-apply-cblinfun-compose image-iff that*)
moreover have ‹$\psi \in$ *closure* (*range* C)›
by (*metis cblinfun-image.rep-eq mem top-ccsubspace.rep-eq*)
ultimately show *?thesis*
apply (*rule on-closure-eqI*)
by (*auto simp*: *continuous-on-eq-continuous-at*)
qed

lemma *lift-cblinfun-comp*:
— Utility lemma to lift a lemma of the form $a\ o_{CL}\ b = c$ to become a more general rewrite rule.
assumes ‹$a\ o_{CL}\ b = c$›
shows ‹$a\ o_{CL}\ b = c$›
and ‹$a\ o_{CL}\ (b\ o_{CL}\ d) = c\ o_{CL}\ d$›
and ‹$a *_S (b *_S S) = c *_S S$›
and ‹$a *_V (b *_V x) = c *_V x$›
apply (*fact assms*)
apply (*simp add*: *assms cblinfun-assoc-left*(*1*))
using *assms cblinfun-assoc-left*(*2*) **apply** *force*
using *assms* **by** *force*

lemma *cblinfun-image-def2*: ‹$A *_S S =$ *ccspan* $((*_V)\ A\ `$ *space-as-set* $S)$›
apply (*simp add*: *flip*: *cblinfun-image-ccspan*)
by (*metis ccspan-leqI ccspan-superset less-eq-ccsubspace.rep-eq order-class.order-eq-iff*)

lemma *unitary-image-onb*:
— Logically belongs in an earlier section but the proof uses results from this section.
assumes ‹*is-onb* A›
assumes ‹*unitary* U›
shows ‹*is-onb* $(U\ `\ A)$›
using *assms*
by (*auto simp add*: *is-onb-def isometry-image-is-ortho-set isometry-preserves-norm simp flip*: *cblinfun-image-ccspan*)

13.9 Sandwiches

lift-definition *sandwich* :: ‹('a::*chilbert-space* ⇒$_{CL}$ 'b::*complex-inner*) ⇒ (('a ⇒$_{CL}$ 'a) ⇒$_{CL}$ ('b ⇒$_{CL}$ 'b))› **is**
‹λ(A::'a⇒$_{CL}$'b) B. A o$_{CL}$ B o$_{CL}$ A*›
proof
fix A :: ‹'a ⇒$_{CL}$ 'b› **and** B B1 B2 :: ‹'a ⇒$_{CL}$ 'a› **and** c :: *complex*
show ‹A o$_{CL}$ (B1 + B2) o$_{CL}$ A* = (A o$_{CL}$ B1 o$_{CL}$ A*) + (A o$_{CL}$ B2 o$_{CL}$ A*)›
by (*simp add*: *cblinfun-compose-add-left cblinfun-compose-add-right*)
show ‹A o$_{CL}$ (c *$_C$ B) o$_{CL}$ A* = c *$_C$ (A o$_{CL}$ B o$_{CL}$ A*)›
by *auto*
show ‹∃K. ∀B. *norm* (A o$_{CL}$ B o$_{CL}$ A*) ≤ *norm* B * K›
proof (*rule exI*[*of* - ‹*norm* A * *norm* (A*)›], *rule allI*)
fix B
have ‹*norm* (A o$_{CL}$ B o$_{CL}$ A*) ≤ *norm* (A o$_{CL}$ B) * *norm* (A*)›
using *norm-cblinfun-compose* **by** *blast*
also have ‹... ≤ (*norm* A * *norm* B) * *norm* (A*)›
by (*simp add*: *mult-right-mono norm-cblinfun-compose*)
finally show ‹*norm* (A o$_{CL}$ B o$_{CL}$ A*) ≤ *norm* B * (*norm* A * *norm* (A*))›
by (*simp add*: *mult.assoc vector-space-over-itself.scale-left-commute*)
qed
qed

lemma *sandwich-0*[*simp*]: ‹*sandwich* 0 = 0›
by (*simp add*: *cblinfun-eqI sandwich.rep-eq*)

lemma *sandwich-apply*: ‹*sandwich* A *$_V$ B = A o$_{CL}$ B o$_{CL}$ A*›
apply (*transfer fixing*: A B) **by** *auto*

lemma *sandwich-arg-compose*:
assumes ‹*isometry* U›
shows ‹*sandwich* U x o$_{CL}$ *sandwich* U y = *sandwich* U (x o$_{CL}$ y)›
apply (*simp add*: *sandwich-apply*)
by (*metis* (*no-types*, *lifting*) *lift-cblinfun-comp*(*2*) *assms cblinfun-compose-id-right isometryD*)

lemma *norm-sandwich*: ‹*norm* (*sandwich* A) = (*norm* A)^2› **for** A :: ‹'a::{*chilbert-space*} ⇒$_{CL}$ 'b::{*complex-inner*}›
proof −
have *main*: ‹*norm* (*sandwich* A) = (*norm* A)^2› **for** A :: ‹'c::{*chilbert-space*,*not-singleton*} ⇒$_{CL}$ 'd::{*complex-inner*}›
proof (*rule norm-cblinfun-eqI*)
show ‹(*norm* A)^2 ≤ *norm* (*sandwich* A *$_V$ *id-cblinfun*) / *norm* (*id-cblinfun* :: 'c ⇒$_{CL}$ -)›
apply (*auto simp*: *sandwich-apply*)
by −
fix B
have ‹*norm* (*sandwich* A *$_V$ B) ≤ *norm* (A o$_{CL}$ B) * *norm* (A*)›
using *norm-cblinfun-compose* **by** (*auto simp*: *sandwich-apply simp del*: *norm-adj*)

```
    also have ‹... ≤ (norm A * norm B) * norm (A*)›
      by (simp add: mult-right-mono norm-cblinfun-compose)
    also have ‹... ≤ (norm A)² * norm B›
     by (simp add: power2-eq-square mult.assoc vector-space-over-itself.scale-left-commute)
    finally show ‹norm (sandwich A *V B) ≤ (norm A)² * norm B›
      by −
    show ‹0 ≤ (norm A)²›
      by auto
  qed

  show ?thesis
  proof (cases ‹class.not-singleton TYPE('a)›)
    case True
    show ?thesis
      apply (rule main[internalize-sort' 'c2])
       apply standard[1]
      using True by simp
  next
    case False
    have ‹A = 0›
      apply (rule cblinfun-from-CARD-1-0[internalize-sort' 'a])
       apply (rule not-singleton-vs-CARD-1)
       apply (rule False)
      by standard
    then show ?thesis
      by simp
  qed
qed

lemma sandwich-apply-adj: ‹sandwich A (B*) = (sandwich A B)*›
  by (simp add: cblinfun-assoc-left(1) sandwich-apply)

lemma sandwich-id[simp]: sandwich id-cblinfun = id-cblinfun
  apply (rule cblinfun-eqI)
  by (auto simp: sandwich-apply)

lemma sandwich-compose: ‹sandwich (A oCL B) = sandwich A oCL sandwich B›
  by (auto intro!: cblinfun-eqI simp: sandwich-apply)

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 inj-sandwich-isometry: ‹inj (sandwich U)› if [simp]: ‹isometry U› for U
:: ‹'a::chilbert-space ⇒CL 'b::chilbert-space›
  apply (rule inj-on-inverseI[where g=‹(*V) (sandwich (U*))›])
  by (auto simp flip: cblinfun-apply-cblinfun-compose sandwich-compose)
```

lemma *sandwich-isometry-id*: ‹*isometry* (*U**) ⟹ *sandwich U id-cblinfun* = *id-cblinfun*›
by (*simp add*: *sandwich-apply isometry-def*)

13.10 Projectors

lift-definition *Proj* :: ('*a*::*chilbert-space*) *ccsubspace* ⇒ '*a* ⇒$_{CL}$ '*a*
is ‹*projection*›
by (*rule projection-bounded-clinear*)

lemma *Proj-range*[*simp*]: *Proj S* *$_S$ *top* = *S*
proof *transfer*
fix *S* :: ‹'*a set*› **assume** ‹*closed-csubspace S*›
then have *closure* (*range* (*projection S*)) ⊆ *S*
by (*metis closed-csubspace.closed closed-csubspace.subspace closure-closed complex-vector.subspace-0 csubspace-is-convex dual-order.eq-iff insert-absorb insert-not-empty projection-image*)
moreover have *S* ⊆ *closure* (*range* (*projection S*))
using ‹*closed-csubspace S*›
by (*metis closed-csubspace-def closure-subset csubspace-is-convex equals0D projection-image subset-iff*)
ultimately show ‹*closure* (*range* (*projection S*)) = *S*›
by *auto*
qed

lemma *adj-Proj*: ‹(*Proj M*)* = *Proj M*›
by *transfer* (*simp add*: *projection-cadjoint*)

lemma *Proj-idempotent*[*simp*]: ‹*Proj M* o$_{CL}$ *Proj M* = *Proj M*›
proof −
have *u1*: ‹(*cblinfun-apply* (*Proj M*)) = *projection* (*space-as-set M*)›
by *transfer blast*
have ‹*closed-csubspace* (*space-as-set M*)›
using *space-as-set* **by** *auto*
hence *u2*: ‹(*projection* (*space-as-set M*))∘(*projection* (*space-as-set M*))
= (*projection* (*space-as-set M*))›
using *projection-idem* **by** *fastforce*
have ‹(*cblinfun-apply* (*Proj M*)) ∘ (*cblinfun-apply* (*Proj M*)) = *cblinfun-apply* (*Proj M*)›
using *u1 u2*
by *simp*
hence ‹*cblinfun-apply* ((*Proj M*) o$_{CL}$ (*Proj M*)) = *cblinfun-apply* (*Proj M*)›
by (*simp add*: *cblinfun-compose.rep-eq*)
thus *?thesis* **using** *cblinfun-apply-inject*
by *auto*
qed

lift-definition *is-Proj* :: ‹'*a*::*complex-normed-vector* ⇒$_{CL}$ '*a* ⇒ *bool*› **is**
‹λ*P*. ∃ *M*. *is-projection-on P M*› .

lemma *is-Proj-id*[*simp*]: ‹*is-Proj id-cblinfun*›
apply *transfer*
by (*auto intro*!: *exI*[*of - UNIV*] *simp*: *is-projection-on-def is-arg-min-def*)

lemma *Proj-top*[*simp*]: ‹*Proj* ⊤ = *id-cblinfun*›
by (*metis Proj-idempotent Proj-range cblinfun-eqI cblinfun-fixes-range id-cblinfun-apply iso-tuple-UNIV-I space-as-set-top*)

lemma *Proj-on-own-range'*:
fixes *P* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*a*›
assumes ‹*P* o_{CL} *P* = *P*› **and** ‹*P* = *P**›
shows ‹*Proj* (*P* $*_S$ *top*) = *P*›
proof −
define *M* **where** *M* = *P* $*_S$ *top*
have *v3*: *x* ∈ (λ*x*. *x* − *P* $*_V$ *x*) −‘ {*0*}
if *x* ∈ *range* (*cblinfun-apply P*)
for *x* :: '*a*
proof−
have *v3-1*: ‹*cblinfun-apply P* ∘ *cblinfun-apply P* = *cblinfun-apply P*›
by (*metis* ‹*P* o_{CL} *P* = *P*› *cblinfun-compose.rep-eq*)
have ‹∃ *t*. *P* $*_V$ *t* = *x*›
using *that* **by** *blast*
then obtain *t* **where** *t-def*: ‹*P* $*_V$ *t* = *x*›
by *blast*
hence ‹*x* − *P* $*_V$ *x* = *x* − *P* $*_V$ (*P* $*_V$ *t*)›
by *simp*
also have ‹... = *x* − (*P* $*_V$ *t*)›
using *v3-1*
by (*metis comp-apply*)
also have ‹... = *0*›
by (*simp add*: *t-def*)
finally have ‹*x* − *P* $*_V$ *x* = *0*›
by *blast*
thus *?thesis*
by *simp*
qed

have *v1*: *range* (*cblinfun-apply P*) ⊆ (λ*x*. *x* − *cblinfun-apply P x*) −‘ {*0*}
using *v3*
by *blast*

have *x* ∈ *range* (*cblinfun-apply P*)
if *x* ∈ (λ*x*. *x* − *P* $*_V$ *x*) −‘ {*0*}
for *x* :: '*a*
proof−
have *x1*:‹*x* − *P* $*_V$ *x* = *0*›
using *that* **by** *blast*
have ‹*x* = *P* $*_V$ *x*›

```
    by (simp add: x1 eq-iff-diff-eq-0)
  thus ?thesis
    by blast
qed
hence v2: (λx. x − cblinfun-apply P x) −‘ {0} ⊆ range (cblinfun-apply P)
  by blast
have i1: ‹range (cblinfun-apply P) = (λ x. x − cblinfun-apply P x) −‘ {0}›
  using v1 v2
  by (simp add: v1 dual-order.antisym)
have p1: ‹closed {(0::'a)}›
  by simp
have p2: ‹continuous (at x) (λ x. x − P *V x)›
  for x
proof−
  have ‹cblinfun-apply (id-cblinfun − P) = (λ x. x − P *V x)›
    by (simp add: id-cblinfun.rep-eq minus-cblinfun.rep-eq)
  hence ‹bounded-clinear (cblinfun-apply (id-cblinfun − P))›
    using cblinfun-apply
    by blast
  hence ‹continuous (at x) (cblinfun-apply (id-cblinfun − P))›
    by (simp add: clinear-continuous-at)
  thus ?thesis
    using ‹cblinfun-apply (id-cblinfun − P) = (λ x. x − P *V x)›
    by simp
qed

have i2: ‹closed ( (λ x. x − P *V x) −‘ {0} )›
  using p1 p2
  by (rule Abstract-Topology.continuous-closed-vimage)

have ‹closed (range (cblinfun-apply P))›
  using i1 i2
  by simp
have u2: ‹cblinfun-apply P x ∈ space-as-set M›
  for x
 by (simp add: M-def ‹closed (range ((*V) P))› cblinfun-image.rep-eq top-ccsubspace.rep-eq)

have xy: ‹is-orthogonal (x − P *V x) y›
  if y1: ‹y ∈ space-as-set M›
  for x y
proof−
  have ‹∃ t. y = P *V t›
    using y1
   by (simp add: M-def ‹closed (range ((*V) P))› cblinfun-image.rep-eq image-iff
       top-ccsubspace.rep-eq)
  then obtain t where t-def: ‹y = P *V t›
    by blast
  have ‹(x − P *V x) •C y = (x − P *V x) •C (P *V t)›
    by (simp add: t-def)
```

```
    also have ‹... = (P *V (x - P *V x)) ·C t›
      by (metis ‹P = P*› cinner-adj-left)
    also have ‹... = (P *V x - P *V (P *V x)) ·C t›
      by (simp add: cblinfun.diff-right)
    also have ‹... = (P *V x - P *V x) ·C t›
      by (metis assms(1) comp-apply cblinfun-compose.rep-eq)
    also have ‹... = (0 ·C t)›
      by simp
    also have ‹... = 0›
      by simp
    finally show ?thesis by blast
  qed
  hence u1: ‹x - P *V x ∈ orthogonal-complement (space-as-set M)›
    for x
    by (simp add: orthogonal-complementI)
  have closed-csubspace (space-as-set M)
    using space-as-set by auto
  hence f1: (Proj M) *V a = P *V a for a
    by (simp add: Proj.rep-eq projection-eqI u1 u2)
  have (+) ((P - Proj M) *V a) = id for a
    using f1
    by (auto intro!: ext simp add: minus-cblinfun.rep-eq)
  hence b - b = cblinfun-apply (P - Proj M) a
    for a b
    by (metis (no-types) add-diff-cancel-right' id-apply)
  hence cblinfun-apply (id-cblinfun - (P - Proj M)) a = a
    for a
    by (simp add: minus-cblinfun.rep-eq)
  thus ?thesis
   using u1 u2 cblinfun-apply-inject diff-diff-eq2 diff-eq-diff-eq eq-id-iff id-cblinfun.rep-eq
    by (metis (no-types, opaque-lifting) M-def)
qed

lemma Proj-range-closed:
  assumes is-Proj P
  shows closed (range (cblinfun-apply P))
  apply (rule is-projection-on-closed[where f=‹cblinfun-apply P›])
  using assms is-Proj.rep-eq is-projection-on-image by auto

lemma Proj-is-Proj[simp]:
  fixes M::‹'a::chilbert-space ccsubspace›
  shows ‹is-Proj (Proj M)›
proof-
  have u1: closed-csubspace (space-as-set M)
    using space-as-set by blast
  have v1: h - Proj M *V h
        ∈ orthogonal-complement (space-as-set M) for h
    by (simp add: Proj.rep-eq orthogonal-complementI projection-orthogonal u1)
  have v2: Proj M *V h ∈ space-as-set M for h
```

by (*metis Proj.rep-eq mem-Collect-eq orthog-proj-exists projection-eqI space-as-set*)
have *u2*: *is-projection-on* ((∗$_V$) (*Proj M*)) (*space-as-set M*)
unfolding *is-projection-on-def*
by (*simp add*: *smallest-dist-is-ortho u1 v1 v2*)
show *?thesis*
using *u1 u2 is-Proj.rep-eq* **by** *blast*
qed

lemma *is-Proj-algebraic*:
fixes *P*::‹*'a*::*chilbert-space* ⇒$_{CL}$ *'a*›
shows ‹*is-Proj P* ⟷ *P o*$_{CL}$ *P* = *P* ∧ *P* = *P*∗›
proof
have *P o*$_{CL}$ *P* = *P*
if *is-Proj P*
using *that* **apply** *transfer*
using *is-projection-on-idem*
by *fastforce*
moreover have *P* = *P*∗
if *is-Proj P*
using *that Proj-range-closed*[*OF that*] *is-projection-on-cadjoint*[**where** *π*=*P*
and *M*=‹*range P*›]
by *transfer* (*metis bounded-clinear.axioms*(*1*) *closed-csubspace-UNIV closed-csubspace-def complex-vector.linear-subspace-image is-projection-on-image*)
ultimately show *P o*$_{CL}$ *P* = *P* ∧ *P* = *P*∗
if *is-Proj P*
using *that*
by *blast*
show *is-Proj P*
if *P o*$_{CL}$ *P* = *P* ∧ *P* = *P*∗
using *that Proj-on-own-range' Proj-is-Proj* **by** *metis*
qed

lemma *Proj-on-own-range*:
fixes *P* :: ‹*'a*::*chilbert-space* ⇒$_{CL}$ *'a*›
assumes ‹*is-Proj P*›
shows ‹*Proj* (*P* ∗$_S$ *top*) = *P*›
using *Proj-on-own-range' assms is-Proj-algebraic* **by** *blast*

lemma *Proj-image-leq*: (*Proj S*) ∗$_S$ *A* ≤ *S*
by (*metis Proj-range inf-top-left le-inf-iff isometry-cblinfun-image-inf-distrib'*)

lemma *Proj-sandwich*:
fixes *A*::*'a*::*chilbert-space* ⇒$_{CL}$ *'b*::*chilbert-space*
assumes *isometry A*
shows *sandwich A* ∗$_V$ *Proj S* = *Proj* (*A* ∗$_S$ *S*)
proof −
define *P* **where** ‹*P* = *A o*$_{CL}$ *Proj S o*$_{CL}$ (*A*∗)›
have ‹*P o*$_{CL}$ *P* = *P*›
using *assms*

unfolding *P-def isometry-def*
by (*metis* (*no-types*, *lifting*) *Proj-idempotent cblinfun-assoc-left*(*1*) *cblinfun-compose-id-left*)
moreover have ‹$P = P*$›
unfolding *P-def*
by (*metis adj-Proj adj-cblinfun-compose cblinfun-assoc-left*(*1*) *double-adj*)
ultimately have
‹$\exists M.\ P = Proj\ M \wedge space\text{-}as\text{-}set\ M = range\ (cblinfun\text{-}apply\ (A\ o_{CL}\ (Proj\ S)\ o_{CL}\ (A*)))$›
using *P-def Proj-on-own-range'*
by (*metis Proj-is-Proj Proj-range-closed cblinfun-image.rep-eq closure-closed top-ccsubspace.rep-eq*)
then obtain *M* **where** ‹$P = Proj\ M$› **and** ‹$space\text{-}as\text{-}set\ M = range\ (cblinfun\text{-}apply\ (A\ o_{CL}\ (Proj\ S)\ o_{CL}\ (A*)))$›
by *blast*

have *f1*: $A\ o_{CL}\ Proj\ S = P\ o_{CL}\ A$
by (*simp add*: *P-def assms cblinfun-compose-assoc*)
hence $P\ o_{CL}\ A\ o_{CL}\ A* = P$
using *P-def* **by** *presburger*
hence $(P\ o_{CL}\ A) *_S (c \sqcup A* *_S d) = P *_S (A *_S c \sqcup d)$
for *c d*

by (*simp add*: *cblinfun-assoc-left*(*2*))
hence $P *_S (A *_S \top \sqcup c) = (P\ o_{CL}\ A) *_S \top$
for *c*
by (*metis sup-top-left*)
hence ‹$M = A *_S S$›
using *f1*
by (*metis* ‹$P = Proj\ M$› *cblinfun-assoc-left*(*2*) *Proj-range sup-top-right*)
thus *?thesis*
using ‹$P = Proj\ M$›
unfolding *P-def sandwich-apply* **by** *blast*
qed

lemma *Proj-orthog-ccspan-union*:
assumes $\bigwedge x\ y.\ x \in X \Longrightarrow y \in Y \Longrightarrow is\text{-}orthogonal\ x\ y$
shows ‹$Proj\ (ccspan\ (X \cup Y)) = Proj\ (ccspan\ X) + Proj\ (ccspan\ Y)$›
proof −
have ‹$x \in cspan\ X \Longrightarrow y \in cspan\ Y \Longrightarrow is\text{-}orthogonal\ x\ y$› **for** *x y*
apply (*rule is-orthogonal-closure-cspan*[**where** *X*=*X* **and** *Y*=*Y*])
using *closure-subset assms* **by** *auto*
then have ‹$x \in closure\ (cspan\ X) \Longrightarrow y \in closure\ (cspan\ Y) \Longrightarrow is\text{-}orthogonal\ x\ y$› **for** *x y*
by (*metis orthogonal-complementI orthogonal-complement-of-closure orthogonal-complement-orthoI'*)
then show *?thesis*
apply (*transfer fixing*: *X Y*)
apply (*subst projection-plus*[*symmetric*])
by *auto*

qed

abbreviation *proj* :: *'a::chilbert-space* ⇒ *'a* ⇒$_{CL}$ *'a* **where** *proj ψ* ≡ *Proj* (*ccspan* {*ψ*})

lemma *proj-0*[*simp*]: ‹*proj 0* = *0*›
by *transfer auto*

lemma *ccsubspace-supI-via-Proj*:
fixes *A B C*::*'a*::*chilbert-space ccsubspace*
assumes *a1*: ‹*Proj* (− *C*) *∗S A* ≤ *B*›
shows *A* ≤ *B* ⊔ *C*
proof−
have *x2*: ‹*x* ∈ *space-as-set B*›
if *x* ∈ *closure* ((*projection* (*orthogonal-complement* (*space-as-set C*))) ‘ *space-as-set A*) **for** *x*
using *that*
by (*metis Proj.rep-eq cblinfun-image.rep-eq assms less-eq-ccsubspace.rep-eq subsetD*
uminus-ccsubspace.rep-eq)
have *q1*: ‹*x* ∈ *closure* {*ψ* + *φ* |*ψ φ*. *ψ* ∈ *space-as-set B* ∧ *φ* ∈ *space-as-set C*}›
if ‹*x* ∈ *space-as-set A*›
for *x*
proof−
have *p1*: ‹*closed-csubspace* (*space-as-set C*)›
using *space-as-set* **by** *auto*
hence ‹*x* = (*projection* (*space-as-set C*)) *x*
+ (*projection* (*orthogonal-complement* (*space-as-set C*))) *x*›
by *simp*
hence ‹*x* = (*projection* (*orthogonal-complement* (*space-as-set C*))) *x*
+ (*projection* (*space-as-set C*)) *x*›
by (*metis ordered-field-class.sign-simps*(*2*))
moreover have ‹(*projection* (*orthogonal-complement* (*space-as-set C*))) *x* ∈ *space-as-set B*›
using *x2*
by (*meson closure-subset image-subset-iff that*)
moreover have ‹(*projection* (*space-as-set C*)) *x* ∈ *space-as-set C*›
by (*metis mem-Collect-eq orthog-proj-exists projection-eqI space-as-set*)
ultimately show *?thesis*
using *closure-subset* **by** *force*
qed
have *x1*: ‹*x* ∈ (*space-as-set B* +$_M$ *space-as-set C*)›
if *x* ∈ *space-as-set A* **for** *x*
proof −
have *f1*: *x* ∈ *closure* {*a* + *b* |*a b*. *a* ∈ *space-as-set B* ∧ *b* ∈ *space-as-set C*}
by (*simp add*: *q1 that*)
have {*a* + *b* |*a b*. *a* ∈ *space-as-set B* ∧ *b* ∈ *space-as-set C*} = {*a*. ∃ *p*. *p* ∈ *space-as-set B*
∧ (∃ *q*. *q* ∈ *space-as-set C* ∧ *a* = *p* + *q*)}

```
    by blast
  hence x ∈ closure {a. ∃ b∈space-as-set B. ∃ c∈space-as-set C. a = b + c}
    using f1 by (simp add: Bex-def-raw)
  thus ?thesis
    using that
    unfolding closed-sum-def set-plus-def
    by blast
qed

hence ‹x ∈ space-as-set (Abs-ccsubspace (space-as-set B +_M space-as-set C))›
  if x ∈ space-as-set A for x
  using that
  by (metis space-as-set-inverse sup-ccsubspace.rep-eq)
thus ?thesis
  by (simp add: x1 less-eq-ccsubspace.rep-eq subset-eq sup-ccsubspace.rep-eq)
qed

lemma is-Proj-idempotent:
  assumes is-Proj P
  shows P o_CL P = P
  using assms apply transfer
  using is-projection-on-fixes-image is-projection-on-in-image by fastforce

lemma is-proj-selfadj:
  assumes is-Proj P
  shows P* = P
  using assms
  unfolding is-Proj-def
  by (metis is-Proj-algebraic is-Proj-def)

lemma is-Proj-I:
  assumes P o_CL P = P and P* = P
  shows is-Proj P
  using assms is-Proj-algebraic by metis

lemma is-Proj-0[simp]: is-Proj 0
  apply transfer apply (rule exI[of - 0])
  by (simp add: is-projection-on-zero)

lemma is-Proj-complement[simp]:
  fixes P :: ‹'a::chilbert-space ⇒_CL 'a›
  assumes a1: is-Proj P
  shows is-Proj (id-cblinfun - P)
 by (smt (z3) add-diff-cancel-left add-diff-cancel-left' adj-cblinfun-compose adj-plus
assms bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose diff-add-cancel
id-cblinfun-adjoint is-Proj-algebraic cblinfun-compose-id-left)

lemma Proj-bot[simp]: Proj bot = 0
  by (metis zero-cblinfun-image Proj-on-own-range' is-Proj-0 is-Proj-algebraic
```

zero-ccsubspace-def)

lemma *Proj-ortho-compl*:
Proj (− *X*) = *id-cblinfun* − *Proj X*
by (*transfer*, *auto*)

lemma *Proj-inj*:
assumes *Proj X* = *Proj Y*
shows *X* = *Y*
by (*metis assms Proj-range*)

lemma *norm-Proj-leq1*: ‹*norm* (*Proj M*) ≤ *1*› **for** *M* :: ‹'*a* :: *chilbert-space ccsubspace*›
by *transfer* (*metis* (*no-types*, *opaque-lifting*) *mult.left-neutral onorm-bound projection-reduces-norm zero-less-one-class.zero-le-one*)

lemma *Proj-orthog-ccspan-insert*:
assumes ⋀*y*. *y* ∈ *Y* ⟹ *is-orthogonal x y*
shows ‹*Proj* (*ccspan* (*insert x Y*)) = *proj x* + *Proj* (*ccspan Y*)›
apply (*subst asm-rl*[*of* ‹*insert x Y* = {*x*} ∪ *Y*›], *simp*)
apply (*rule Proj-orthog-ccspan-union*)
using *assms* **by** *auto*

lemma *Proj-fixes-image*: ‹*Proj S* *$_V$ *ψ* = *ψ*› **if** ‹*ψ* ∈ *space-as-set S*›
by (*metis Proj-idempotent Proj-range that cblinfun-fixes-range*)

lemma *norm-is-Proj*: ‹*norm P* ≤ *1*› **if** ‹*is-Proj P*› **for** *P* :: ‹'*a* :: *chilbert-space* ⇒$_{CL}$ '*a*›
by (*metis Proj-on-own-range norm-Proj-leq1 that*)

lemma *Proj-sup*: ‹*orthogonal-spaces S T* ⟹ *Proj* (*sup S T*) = *Proj S* + *Proj T*›
unfolding *orthogonal-spaces-def*
by *transfer* (*simp add*: *projection-plus*)

lemma *Proj-sum-spaces*:
assumes ‹*finite X*›
assumes ‹⋀*x y*. *x*∈*X* ⟹ *y*∈*X* ⟹ *x*≠*y* ⟹ *orthogonal-spaces* (*J x*) (*J y*)›
shows ‹*Proj* (∑ *x*∈*X*. *J x*) = (∑ *x*∈*X*. *Proj* (*J x*))›
using *assms*
proof *induction*
case *empty*
show *?case*
by *auto*
next
case (*insert x F*)
have ‹*Proj* (*sum J* (*insert x F*)) = *Proj* (*J x* ⊔ *sum J F*)›
by (*simp add*: *insert*)
also have ‹. . . = *Proj* (*J x*) + *Proj* (*sum J F*)›
apply (*rule Proj-sup*)

apply (*rule orthogonal-sum*)
using *insert* **by** *auto*
also have ‹... = ($\sum x{\in}$*insert x F. Proj* (*J x*))›
by (*simp add*: *insert.IH insert.hyps*(*1*) *insert.hyps*(*2*) *insert.prems*)
finally show *?case*
by −
qed

lemma *is-Proj-reduces-norm*:
fixes *P* :: ‹'*a*::*complex-inner* ⇒$_{CL}$ '*a*›
assumes ‹*is-Proj P*›
shows ‹*norm* (*P* *$_V$ ψ) ≤ *norm* ψ›
apply (*rule is-projection-on-reduces-norm*[**where** *M*=‹*range P*›])
using *assms is-Proj.rep-eq is-projection-on-image* **by** *blast* (*simp add*: *Proj-range-closed assms closed-csubspace.intro*)

lemma *norm-Proj-apply*: ‹*norm* (*Proj T* *$_V$ ψ) = *norm* ψ ⟷ ψ ∈ *space-as-set T*›
proof (*rule iffI*)
show ‹*norm* (*Proj T* *$_V$ ψ) = *norm* ψ› **if** ‹ψ ∈ *space-as-set T*›
by (*simp add*: *Proj-fixes-image that*)
assume *assm*: ‹*norm* (*Proj T* *$_V$ ψ) = *norm* ψ›
have *ψ-decomp*: ‹ψ = *Proj T* ψ + *Proj* (−*T*) ψ›
by (*simp add*: *Proj-ortho-compl cblinfun.real.diff-left*)
have ‹(*norm* (*Proj* (−*T*) ψ))^2 = (*norm* (*Proj T* ψ))^2 + (*norm* (*Proj* (−*T*) ψ))^2 − (*norm* (*Proj T* ψ))^2›
by *auto*
also have ‹... = (*norm* (*Proj T* ψ + *Proj* (−*T*) ψ))^2 − (*norm* (*Proj T* ψ))^2›
apply (*subst pythagorean-theorem*)
apply (*metis* (*no-types*, *lifting*) *Proj-idempotent ψ-decomp add-cancel-right-left adj-Proj cblinfun.real.add-right cblinfun-apply-cblinfun-compose cinner-adj-left cinner-zero-left*)
by *simp*
also with *ψ-decomp* **have** ‹... = (*norm* ψ)^2 − (*norm* (*Proj T* ψ))^2›
by *metis*
also with *assm* **have** ‹... = *0*›
by *simp*
finally have ‹*norm* (*Proj* (−*T*) ψ) = *0*›
by *auto*
with *ψ-decomp* **have** ‹ψ = *Proj T* ψ›
by *auto*
then show ‹ψ ∈ *space-as-set T*›
by (*metis Proj-range cblinfun-apply-in-image*)
qed

lemma *norm-Proj-apply-1*: ‹*norm* ψ = *1* ⟹ *norm* (*Proj T* *$_V$ ψ) = *1* ⟷ ψ ∈ *space-as-set T*›
using *norm-Proj-apply* **by** *metis*

lemma *norm-is-Proj-nonzero*: ‹*norm P = 1*› **if** ‹*is-Proj P*› **and** ‹*P ≠ 0*› **for** *P* :: ‹*'a*::*chilbert-space* ⇒$_{CL}$ *'a*›
proof (*rule antisym*)
show ‹*norm P ≤ 1*›
by (*simp add*: *norm-is-Proj that*(*1*))
from ‹*P ≠ 0*›
have ‹*range P ≠ 0*›
by (*metis cblinfun-eq-0-on-UNIV-span complex-vector.span-UNIV rangeI set-zero singletonD*)
then obtain *ψ* **where** ‹*ψ ∈ range P*› **and** ‹*ψ ≠ 0*›
by *force*
then have ‹*P ψ = ψ*›
using *is-Proj.rep-eq is-projection-on-fixes-image is-projection-on-image that*(*1*)
by *blast*
then show ‹*norm P ≥ 1*›
apply (*rule-tac cblinfun-norm-geqI*[*of - - ψ*])
using ‹*ψ ≠ 0*› **by** *simp*
qed

lemma *Proj-compose-cancelI*:
assumes ‹*A* ∗$_S$ ⊤ ≤ *S*›
shows ‹*Proj S* o$_{CL}$ *A = A*›
apply (*rule cblinfun-eqI*)
proof −
fix *x*
have ‹(*Proj S* o$_{CL}$ *A*) ∗$_V$ *x = Proj S* ∗$_V$ (*A* ∗$_V$ *x*)›
by *simp*
also have ‹*... = A* ∗$_V$ *x*›
apply (*rule Proj-fixes-image*)
using *assms cblinfun-apply-in-image less-eq-ccsubspace.rep-eq* **by** *blast*
finally show ‹(*Proj S* o$_{CL}$ *A*) ∗$_V$ *x = A* ∗$_V$ *x*›
by −
qed

lemma *Proj-o-Proj-subspace-right*:
assumes ‹*A ≥ 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 ≤ B*›
shows ‹*Proj A* o$_{CL}$ *Proj B = Proj A*›
by (*metis Proj-o-Proj-subspace-right adj-Proj adj-cblinfun-compose assms*)

lemma *space-as-setI-via-Proj*:
assumes ‹*Proj M* ∗$_V$ *x = x*›
shows ‹*x ∈ space-as-set M*›
using *assms norm-Proj-apply* **by** *fastforce*

lemma *unitary-image-ortho-compl*:
— Logically, this lemma belongs in an earlier section but its proof uses projectors.
fixes U :: ‹'a::*chilbert-space* ⇒$_{CL}$ 'b::*chilbert-space*›
assumes [*simp*]: ‹*unitary U*›
shows ‹$U *_S (- A) = - (U *_S A)$›
proof −
have ‹*Proj* ($U *_S (- A)$) = *sandwich U* (*Proj* ($- A$))›
by (*simp add*: *Proj-sandwich*)
also have ‹... = *sandwich U* (*id-cblinfun* − *Proj A*)›
by (*simp add*: *Proj-ortho-compl*)
also have ‹... = *id-cblinfun* − *sandwich U* (*Proj A*)›
by (*metis assms cblinfun.diff-right sandwich-isometry-id unitary-twosided-isometry*)
also have ‹... = *id-cblinfun* − *Proj* ($U *_S A$)›
by (*simp add*: *Proj-sandwich*)
also have ‹... = *Proj* ($- (U *_S A)$)›
by (*simp add*: *Proj-ortho-compl*)
finally show *?thesis*
by (*simp add*: *Proj-inj*)
qed

lemma *Proj-on-image* [*simp*]: ‹*Proj S* $*_S$ *S* = *S*›
by (*metis Proj-idempotent Proj-range cblinfun-compose-image*)

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 *Proj-0-compl*: ‹*Proj S x* = *0*› **if** ‹*x* ∈ *space-as-set* (−*S*)›
by (*metis* (*no-types*, *lifting*) *ext Proj-fixes-image Proj-idempotent Proj-ortho-compl Proj-top UNIV-I*
cancel-comm-monoid-add-class.diff-cancel cblinfun.real.diff-left cblinfun.real.diff-right cblinfun-apply-cblinfun-compose space-as-set-top that)

13.11 Kernel / eigenspaces

lift-definition *kernel* :: 'a::*complex-normed-vector* ⇒$_{CL}$ 'b::*complex-normed-vector* ⇒ 'a *ccsubspace*
is λ*f*. *f* −‘ {*0*}
by (*metis kernel-is-closed-csubspace*)

definition *eigenspace* :: *complex* ⇒ *'a*::*complex-normed-vector* ⇒$_{CL}$ *'a* ⇒ *'a ccsubspace* **where**
eigenspace a A = *kernel* (*A* − *a* ∗$_C$ *id-cblinfun*)

lemma *kernel-scaleC*[*simp*]: *a*≠*0* ⟹ *kernel* (*a* ∗$_C$ *A*) = *kernel A*
for *a* :: *complex* **and** *A* :: (-,-) *cblinfun*
apply *transfer*
using *complex-vector.scale-eq-0-iff* **by** *blast*

lemma *kernel-0*[*simp*]: *kernel 0* = *top*
by *transfer auto*

lemma *kernel-id*[*simp*]: *kernel id-cblinfun* = *0*
by *transfer simp*

lemma *eigenspace-scaleC*[*simp*]:
assumes *a1*: *a* ≠ *0*
shows *eigenspace b* (*a* ∗$_C$ *A*) = *eigenspace* (*b*/*a*) *A*
proof −
have *b* ∗$_C$ (*id-cblinfun*::(*'a*, -) *cblinfun*) = *a* ∗$_C$ (*b* / *a*) ∗$_C$ *id-cblinfun*
using *a1*
by (*metis ceq-vector-fraction-iff*)
hence *kernel* (*a* ∗$_C$ *A* − *b* ∗$_C$ *id-cblinfun*) = *kernel* (*A* − (*b* / *a*) ∗$_C$ *id-cblinfun*)
using *a1* **by** (*metis* (*no-types*) *complex-vector.scale-right-diff-distrib kernel-scaleC*)
thus *?thesis*
unfolding *eigenspace-def*
by *blast*
qed

lemma *eigenspace-memberD*:
assumes *x* ∈ *space-as-set* (*eigenspace e A*)
shows *A* ∗$_V$ *x* = *e* ∗$_C$ *x*
using *assms* **unfolding** *eigenspace-def* **by** *transfer auto*

lemma *kernel-memberD*:
assumes *x* ∈ *space-as-set* (*kernel A*)
shows *A* ∗$_V$ *x* = *0*
using *assms* **by** *transfer auto*

lemma *eigenspace-memberI*:
assumes *A* ∗$_V$ *x* = *e* ∗$_C$ *x*
shows *x* ∈ *space-as-set* (*eigenspace e A*)
using *assms* **unfolding** *eigenspace-def* **by** *transfer auto*

lemma *kernel-memberI*:
assumes *A* ∗$_V$ *x* = *0*
shows *x* ∈ *space-as-set* (*kernel A*)
using *assms* **by** *transfer auto*

lemma *kernel-Proj*[*simp*]: ‹*kernel* (*Proj S*) = − *S*›
apply *transfer*
apply *auto*
apply (*metis diff-0-right is-projection-on-iff-orthog projection-is-projection-on'*)
by (*simp add*: *complex-vector.subspace-0 projection-eqI*)

lemma *orthogonal-projectors-orthogonal-spaces*:
— Logically belongs in section "Projectors".
fixes *S T* :: ‹'*a*::*chilbert-space ccsubspace*›
shows ‹*orthogonal-spaces S T* ⟷ *Proj S* o_{CL} *Proj T* = *0*›
proof (*intro ballI iffI*)
assume ‹*Proj S* o_{CL} *Proj T* = *0*›
then have ‹*is-orthogonal x y*› **if** ‹*x* ∈ *space-as-set S*› ‹*y* ∈ *space-as-set T*› **for** *x y*
by (*metis* (*no-types*, *opaque-lifting*) *Proj-fixes-image adj-Proj cblinfun.zero-left cblinfun-apply-cblinfun-compose cinner-adj-right cinner-zero-right that*(*1*) *that*(*2*))
then show ‹*orthogonal-spaces S T*›
by (*simp add*: *orthogonal-spaces-def*)
next
assume ‹*orthogonal-spaces S T*›
then have ‹*S* ≤ − *T*›
by (*simp add*: *orthogonal-spaces-leq-compl*)
then show ‹*Proj S* o_{CL} *Proj T* = *0*›
by (*metis* (*no-types*, *opaque-lifting*) *Proj-range adj-Proj adj-cblinfun-compose basic-trans-rules*(*31*) *cblinfun.zero-left cblinfun-apply-cblinfun-compose cblinfun-apply-in-image cblinfun-eqI kernel-Proj kernel-memberD less-eq-ccsubspace.rep-eq*)
qed

lemma *cblinfun-compose-Proj-kernel*[*simp*]: ‹*a* o_{CL} *Proj* (*kernel a*) = *0*›
apply (*rule cblinfun-eqI*)
by *simp* (*metis Proj-range cblinfun-apply-in-image kernel-memberD*)

lemma *kernel-compl-adj-range*:
shows ‹*kernel a* = − (*a*∗ $*_S$ *top*)›
proof (*rule ccsubspace-eqI*)
fix *x*
have ‹*x* ∈ *space-as-set* (*kernel a*) ⟷ *a x* = *0*›
by *transfer simp*
also have ‹*a x* = *0* ⟷ (∀ *y*. *is-orthogonal y* (*a x*))›
by (*metis cinner-gt-zero-iff cinner-zero-right*)
also have ‹... ⟷ (∀ *y*. *is-orthogonal* (*a*∗ $*_V$ *y*) *x*)›
by (*simp add*: *cinner-adj-left*)
also have ‹... ⟷ *x* ∈ *space-as-set* (− (*a*∗ $*_S$ *top*))›
by *transfer* (*metis* (*mono-tags*, *opaque-lifting*) *UNIV-I image-iff is-orthogonal-sym orthogonal-complementI orthogonal-complement-of-closure orthogonal-complement-orthoI'*)
finally show ‹*x* ∈ *space-as-set* (*kernel a*) ⟷ *x* ∈ *space-as-set* (− (*a*∗ $*_S$ *top*))›
by −

qed

lemma *kernel-apply-self*: ‹$A *_S$ kernel $A = 0$›
proof *transfer*
  fix $A :: ‹'b \Rightarrow 'a›$
  assume ‹*bounded-clinear* A›
  then have ‹$A\ 0 = 0$›
    by (*simp add*: *bounded-clinear-def complex-vector.linear-0*)
  then have ‹A ‘ $A -‘ \{0\} = \{0\}$›
    by *fastforce*
  then show ‹*closure* (A ‘ $A -‘ \{0\}$) $= \{0\}$›
    by *auto*
qed

lemma *leq-kernel-iff*:
  shows ‹$A \leq$ *kernel* $B \longleftrightarrow B *_S A = 0$›
proof (*rule iffI*)
  assume ‹$A \leq$ *kernel* B›
  then have ‹$B *_S A \leq B *_S$ *kernel* B›
    by (*simp add*: *cblinfun-image-mono*)
  also have ‹$\ldots = 0$›
    by (*simp add*: *kernel-apply-self*)
  finally show ‹$B *_S A = 0$›
    by (*simp add*: *bot.extremum-unique*)
next
  assume ‹$B *_S A = 0$›
  then show ‹$A \leq$ *kernel* B›
    apply *transfer*
    by (*metis closure-subset image-subset-iff-subset-vimage*)
qed

lemma *cblinfun-image-kernel*:
  assumes ‹$C *_S A *_S$ *kernel* $B \leq$ *kernel* B›
  assumes ‹$A\ o_{CL}\ C =$ *id-cblinfun*›
  shows ‹$A *_S$ *kernel* $B =$ *kernel* $(B\ o_{CL}\ C)$›
proof (*rule antisym*)
  show ‹$A *_S$ *kernel* $B \leq$ *kernel* $(B\ o_{CL}\ C)$›
    using *assms*(*1*) **by** (*simp add*: *leq-kernel-iff cblinfun-compose-image*)
  show ‹*kernel* $(B\ o_{CL}\ C) \leq A *_S$ *kernel* B›
  proof (*insert assms*(*2*), *transfer*, *intro subsetI*)
    fix $A :: ‹'a \Rightarrow 'b›$ **and** $B :: ‹'a \Rightarrow 'c›$ **and** $C :: ‹'b \Rightarrow 'a›$ **and** x
    assume ‹$x \in (B \circ C) -‘ \{0\}$›
    then have *BCx*: ‹$B\ (C\ x) = 0$›
      by *simp*
    assume ‹$A \circ C = (\lambda x.\ x)$›
    then have ‹$x = A\ (C\ x)$›
      apply (*simp add*: *o-def*)
      by *metis*
    then show ‹$x \in$ *closure* (A ‘ $B -‘ \{0\}$)›

```
    using ‹B (C x) = 0› closure-subset by fastforce
  qed
qed

lemma cblinfun-image-kernel-unitary:
  assumes ‹unitary U›
  shows ‹U *S kernel B = kernel (B oCL U*)›
  apply (rule cblinfun-image-kernel)
  using assms by (auto simp flip: cblinfun-compose-image)

lemma kernel-cblinfun-compose:
  assumes ‹kernel B = 0›
  shows ‹kernel A = kernel (B oCL A)›
  using assms apply transfer by auto


lemma eigenspace-0[simp]: ‹eigenspace 0 A = kernel A›
  by (simp add: eigenspace-def)

lemma kernel-isometry: ‹kernel U = 0› if ‹isometry U›
  by (simp add: kernel-compl-adj-range range-adjoint-isometry that)

lemma cblinfun-image-eigenspace-isometry:
  assumes [simp]: ‹isometry A› and ‹c ≠ 0›
  shows ‹A *S eigenspace c B = eigenspace c (sandwich A B)›
proof (rule antisym)
  show ‹A *S eigenspace c B ≤ eigenspace c (sandwich A B)›
  proof (unfold cblinfun-image-def2, rule ccspan-leqI, rule subsetI)
    fix x assume ‹x ∈ (*V) A ‘ space-as-set (eigenspace c B)›
    then obtain y where x-def: ‹x = A y› and ‹y ∈ space-as-set (eigenspace c
B)›
      by auto
    then have ‹B y = c *C y›
      by (simp add: eigenspace-memberD)
    then have ‹sandwich A B x = c *C x›
      apply (simp add: sandwich-apply x-def cblinfun-compose-assoc
          flip: cblinfun-apply-cblinfun-compose)
      by (simp add: cblinfun.scaleC-right)
    then show ‹x ∈ space-as-set (eigenspace c (sandwich A B))›
      by (simp add: eigenspace-memberI)
  qed
  show ‹eigenspace c (sandwich A *V B) ≤ A *S eigenspace c B›
  proof (rule ccsubspace-leI-unit)
    fix x
    assume ‹x ∈ space-as-set (eigenspace c (sandwich A B))›
    then have *: ‹sandwich A B x = c *C x›
      by (simp add: eigenspace-memberD)
    then have ‹c *C x ∈ range A›
      apply (simp add: sandwich-apply)
```

```
    by (metis rangeI)
  then have ‹(inverse c * c) *C x ∈ range A›
    apply (simp flip: scaleC-scaleC)
    by (metis (no-types, lifting) cblinfun.scaleC-right rangeE rangeI)
  with ‹c ≠ 0› have ‹x ∈ range A›
    by simp
  then obtain y where x-def: ‹x = A y›
    by auto
  have ‹B *V y = A* *V sandwich A B x›
    apply (simp add: sandwich-apply x-def)
   by (metis assms cblinfun-apply-cblinfun-compose id-cblinfun.rep-eq isometryD)
  also have ‹... = c *C y›
    apply (simp add: * cblinfun.scaleC-right)
    apply (simp add: x-def)
    by (metis assms(1) cblinfun-apply-cblinfun-compose id-cblinfun-apply isometry-def)
  finally have ‹y ∈ space-as-set (eigenspace c B)›
    by (simp add: eigenspace-memberI)
  then show ‹x ∈ space-as-set (A *S eigenspace c B) ›
    by (simp add: x-def cblinfun-apply-in-image')
 qed
qed

lemma cblinfun-image-eigenspace-unitary:
 assumes [simp]: ‹unitary A›
 shows ‹A *S eigenspace c B = eigenspace c (sandwich A B)›
 apply (cases ‹c = 0›)
  apply (simp add: sandwich-apply cblinfun-image-kernel-unitary kernel-isometry cblinfun-compose-assoc
    flip: kernel-cblinfun-compose)
 by (simp add: cblinfun-image-eigenspace-isometry)

lemma kernel-member-iff: ‹x ∈ space-as-set (kernel A) ⟷ A *V x = 0›
 using kernel-memberD kernel-memberI by blast

lemma kernel-square[simp]: ‹kernel (A* oCL A) = kernel A›
proof (intro ccsubspace-eqI iffI)
 fix x
 assume ‹x ∈ space-as-set (kernel A)›
 then show ‹x ∈ space-as-set (kernel (A* oCL A))›
   by (simp add: kernel.rep-eq)
next
 fix x
 assume ‹x ∈ space-as-set (kernel (A* oCL A))›
 then have ‹A* *V A *V x = 0›
   by (simp add: kernel.rep-eq)
 then have ‹(A *V x) •C (A *V x) = 0›
   by (metis cinner-adj-right cinner-zero-right)
 then have ‹A *V x = 0›
```

by *auto*
then show ‹$x \in$ *space-as-set* (*kernel A*)›
by (*simp add*: *kernel.rep-eq*)
qed

lemma *eq-on-ccsubspaces-Sup*:
fixes *a b* :: ‹'*a*::*complex-normed-vector* $\Rightarrow_{CL}$ '*b*::*complex-normed-vector*›
assumes ‹$\bigwedge i\ h.\ i \in I \Longrightarrow h \in$ *space-as-set* (*X i*) $\Longrightarrow$ *a h* = *b h*›
shows ‹$\bigwedge h.\ h \in$ *space-as-set* ($\bigsqcup i{\in}I.\ X\ i$) $\Longrightarrow$ *a h* = *b h*›
proof −
from *assms*
have ‹*X i* $\leq$ *kernel* (*a* − *b*)› **if** ‹$i \in I$› **for** *i*
using *that* **by** (*auto intro*!: *ccsubspace-leI simp*: *kernel.rep-eq minus-cblinfun.rep-eq*)
then have ‹($\bigsqcup i{\in}I.\ X\ i$) $\leq$ *kernel* (*a* − *b*)›
by (*simp add*: *SUP-least*)
then show ‹$h \in$ *space-as-set* ($\bigsqcup i{\in}I.\ X\ i$) $\Longrightarrow$ *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* $\Rightarrow_{CL}$ '*b*::*complex-normed-vector*›
assumes ‹$\bigwedge h\ i.\ h \in$ *space-as-set S* $\Longrightarrow$ *a h* = *b h*›
assumes ‹$\bigwedge h\ i.\ h \in$ *space-as-set T* $\Longrightarrow$ *a h* = *b h*›
shows ‹$\bigwedge h.\ h \in$ *space-as-set* (*S* $\sqcup$ *T*) $\Longrightarrow$ *a h* = *b h*›
apply (*rule eq-on-ccsubspaces-Sup*[**where** *I*=‹{*True*,*False*}› **and** *X*=‹$\lambda i.$ *if i then T else S*›])
using *assms*
apply *presburger*
by *fastforce*

13.12 Partial isometries

definition *partial-isometry* **where**
‹*partial-isometry A* $\longleftrightarrow$ ($\forall h \in$ *space-as-set* (− *kernel A*). *norm* (*A h*) = *norm h*)›

lemma *partial-isometryI*:
assumes ‹$\bigwedge h.\ h \in$ *space-as-set* (− *kernel A*) $\Longrightarrow$ *norm* (*A h*) = *norm h*›
shows ‹*partial-isometry A*›
using *assms partial-isometry-def* **by** *blast*

lemma
fixes *A* :: ‹'*a* :: *chilbert-space* $\Rightarrow_{CL}$ '*b* :: *complex-normed-vector*›
assumes *iso*: ‹$\bigwedge \psi.\ \psi \in$ *space-as-set V* $\Longrightarrow$ *norm* (*A* $*_V$ ψ) = *norm* ψ›
assumes *zero*: ‹$\bigwedge \psi.\ \psi \in$ *space-as-set* (− *V*) $\Longrightarrow$ *A* $*_V$ ψ = *0*›
shows *partial-isometryI'*: ‹*partial-isometry A*›
and *partial-isometry-initial*: ‹*kernel A* = − *V*›

proof −
from *zero*
have ‹− $V \leq$ *kernel A*›
by (*simp add*: *kernel-memberI less-eq-ccsubspace.rep-eq subsetI*)
moreover have ‹*kernel A* ≤ − *V*›
by (*smt* (*verit*, *ccfv-threshold*) *Proj-ortho-compl Proj-range assms*(1) *cblinfun.diff-left cblinfun.diff-right cblinfun-apply-in-image cblinfun-id-cblinfun-apply ccsubspace-leI kernel-Proj kernel-memberD kernel-memberI norm-eq-zero ortho-involution subsetI zero*)
ultimately show *kerA*: ‹*kernel A* = − *V*›
by *simp*

show ‹*partial-isometry A*›
apply (*rule partial-isometryI*)
by (*simp add*: *kerA iso*)
qed

lemma *Proj-partial-isometry*[*simp*]: ‹*partial-isometry* (*Proj S*)›
apply (*rule partial-isometryI*)
by (*simp add*: *Proj-fixes-image*)

lemma *is-Proj-partial-isometry*: ‹*is-Proj P* ⟹ *partial-isometry P*› **for** *P* :: ‹- :: *chilbert-space* ⇒$_{CL}$ -›
by (*metis Proj-on-own-range Proj-partial-isometry*)

lemma *isometry-partial-isometry*: ‹*isometry P* ⟹ *partial-isometry P*›
by (*simp add*: *isometry-preserves-norm partial-isometry-def*)

lemma *unitary-partial-isometry*: ‹*unitary P* ⟹ *partial-isometry P*›
using *isometry-partial-isometry unitary-isometry* **by** *blast*

lemma *norm-partial-isometry*:
fixes *A* :: ‹'*a* :: *chilbert-space* ⇒$_{CL}$ '*b*::*complex-normed-vector*›
assumes ‹*partial-isometry A*› **and** ‹*A* ≠ *0*›
shows ‹*norm A* = *1*›
proof −
from ‹*A* ≠ *0*›
have ‹− (*kernel A*) ≠ *0*›
by (*metis cblinfun-eqI diff-zero id-cblinfun-apply kernel-id kernel-memberD ortho-involution orthog-proj-exists orthogonal-complement-closed-subspace uminus-ccsubspace.rep-eq zero-cblinfun.rep-eq*)
then obtain *h* **where** ‹*h* ∈ *space-as-set* (− *kernel A*)› **and** ‹*h* ≠ *0*›
by (*metis cblinfun-id-cblinfun-apply ccsubspace-eqI closed-csubspace.subspace complex-vector.subspace-0 kernel-id kernel-memberD kernel-memberI orthogonal-complement-closed-subspace uminus-ccsubspace.rep-eq*)
with ‹*partial-isometry A*›
have ‹*norm* (*A h*) = *norm h*›
using *partial-isometry-def* **by** *blast*
then have ‹*norm A* ≥ *1*›

by (*smt* (*verit*) ‹$h \neq 0$› *mult-cancel-right1 mult-left-le-one-le norm-cblinfun norm-eq-zero norm-ge-zero*)

have ‹*norm A* ≤ *1*›
proof (*rule norm-cblinfun-bound*)
show ‹*0* ≤ (*1*::*real*)›
by *simp*
fix ψ :: $'a$
define *g h* **where** ‹*g* = *Proj* (*kernel A*) ψ› **and** ‹*h* = *Proj* (− *kernel A*) ψ›
have ‹*A g* = *0*›
by (*metis Proj-range cblinfun-apply-in-image g-def kernel-memberD*)
moreover from ‹*partial-isometry A*›
have ‹*norm* (*A h*) = *norm h*›
by (*metis Proj-range cblinfun-apply-in-image h-def partial-isometry-def*)
ultimately have ‹*norm* (*A* ψ) = *norm h*›
by (*simp add*: *Proj-ortho-compl cblinfun.diff-left cblinfun.diff-right g-def h-def*)
also have ‹*norm h* ≤ *norm* ψ›
by (*smt* (*verit*, *del-insts*) *h-def mult-left-le-one-le norm-Proj-leq1 norm-cblinfun norm-ge-zero*)
finally show ‹*norm* (*A* $*_V$ ψ) ≤ *1* ∗ *norm* ψ›
by *simp*
qed

from ‹*norm A* ≤ *1*› **and** ‹*norm A* ≥ *1*›
show ‹*norm A* = *1*›
by *auto*
qed

lemma *partial-isometry-adj-a-o-a*:
assumes ‹*partial-isometry a*›
shows ‹*a*∗ o_{CL} *a* = *Proj* (− *kernel a*)›
proof (*rule cblinfun-cinner-eqI*)
define *P* **where** ‹*P* = *Proj* (− *kernel a*)›
have *aP*: ‹*a* o_{CL} *P* = *a*›
by (*auto intro*!: *simp*: *cblinfun-compose-minus-right P-def Proj-ortho-compl*)
have *is-Proj-P*[*simp*]: ‹*is-Proj P*›
by (*simp add*: *P-def*)

fix ψ :: $'a$
have ‹ψ $\cdot_C$ ((*a*∗ o_{CL} *a*) $*_V$ ψ) = *a* ψ $\cdot_C$ *a* ψ›
by (*simp add*: *cinner-adj-right*)
also have ‹. . . = *a* (*P* ψ) $\cdot_C$ *a* (*P* ψ)›
by (*metis aP cblinfun-apply-cblinfun-compose*)
also have ‹. . . = *P* ψ $\cdot_C$ *P* ψ›
by (*metis P-def Proj-range assms cblinfun-apply-in-image cdot-square-norm partial-isometry-def*)
also have ‹. . . = ψ $\cdot_C$ *P* ψ›
by (*simp flip*: *cinner-adj-right add*: *is-proj-selfadj is-Proj-idempotent*[*THEN simp-a-oCL-b'*])

finally show ‹$\psi \cdot_C ((a* \ o_{CL} \ a) *_V \psi) = \psi \cdot_C P \ \psi$›
by −
qed

lemma *partial-isometry-square-proj*: ‹*is-Proj* $(a* \ o_{CL} \ a)$› **if** ‹*partial-isometry a*›
by (*simp add*: *partial-isometry-adj-a-o-a that*)

lemma *partial-isometry-adj*[*simp*]: ‹*partial-isometry* $(a*)$› **if** ‹*partial-isometry a*›
for a :: ‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*chilbert-space*›
proof −
have *ran-ker*: ‹$a *_S$ *top* = − *kernel* $(a*)$›
by (*simp add*: *kernel-compl-adj-range*)

have ‹*norm* $(a* *_V h) =$ *norm* h› **if** ‹$h \in$ *range* a› **for** h
proof −
from *that* **obtain** x **where** h: ‹$h = a \ x$›
by *auto*
have ‹*norm* $(a* *_V h) =$ *norm* $(a* *_V a *_V x)$›
by (*simp add*: h)
also have ‹$\ldots =$ *norm* $($*Proj* $(-$ *kernel* $a) *_V x)$›
by (*simp add*: ‹*partial-isometry a*› *partial-isometry-adj-a-o-a simp-a-oCL-b'*)
also have ‹$\ldots =$ *norm* $(a *_V$ *Proj* $(-$ *kernel* $a) *_V x)$›
by (*metis Proj-range* ‹*partial-isometry a*› *cblinfun-apply-in-image partial-isometry-def*)
also have ‹$\ldots =$ *norm* $(a *_V x)$›
by (*smt* (*verit*, *best*) *Proj-idempotent* ‹*partial-isometry a*› *adj-Proj cblinfun-apply-cblinfun-compose cinner-adj-right cnorm-eq partial-isometry-adj-a-o-a*)
also have ‹$\ldots =$ *norm* h›
using h **by** *auto*
finally show *?thesis*
by −
qed

then have *norm-pres*: ‹*norm* $(a* *_V h) =$ *norm* h› **if** ‹$h \in$ *closure* $($*range* $a)$›
for h
using *that* **apply** (*rule on-closure-eqI*)
by *assumption* (*intro continuous-intros*)+

show *?thesis*
apply (*rule partial-isometryI*)
by (*auto simp*: *cblinfun-image.rep-eq norm-pres simp flip*: *ran-ker*)
qed

13.13 Isomorphisms and inverses

definition *iso-cblinfun* :: ‹($'a$::*complex-normed-vector*, $'b$::*complex-normed-vector*) *cblinfun* ⇒ *bool*› **where**
‹*iso-cblinfun* $A = (\exists \ B. \ A \ o_{CL} \ B =$ *id-cblinfun* $\wedge \ B \ o_{CL} \ A =$ *id-cblinfun*)›

definition ‹*invertible-cblinfun* $A \longleftrightarrow (\exists B. \ B \ o_{CL} \ A =$ *id-cblinfun*)›

definition *cblinfun-inv* :: ‹('a::*complex-normed-vector*, 'b::*complex-normed-vector*) *cblinfun* ⇒ ('b,'a) *cblinfun*› **where**
‹*cblinfun-inv A* = (*if invertible-cblinfun A then SOME B*. *B* o_{CL} *A* = *id-cblinfun else 0*)›

lemma *cblinfun-inv-left*:
assumes ‹*invertible-cblinfun A*›
shows ‹*cblinfun-inv A* o_{CL} *A* = *id-cblinfun*›
apply (*simp add*: *assms cblinfun-inv-def*)
apply (*rule someI-ex*)
using *assms* **by** (*simp add*: *invertible-cblinfun-def*)

lemma *inv-cblinfun-invertible*: ‹*iso-cblinfun A* ⟹ *invertible-cblinfun A*›
by (*auto simp*: *iso-cblinfun-def invertible-cblinfun-def*)

lemma *cblinfun-inv-right*:
assumes ‹*iso-cblinfun A*›
shows ‹*A* o_{CL} *cblinfun-inv A* = *id-cblinfun*›
proof −
from *assms*
obtain *B* **where** *AB*: ‹*A* o_{CL} *B* = *id-cblinfun*› **and** *BA*: ‹*B* o_{CL} *A* = *id-cblinfun*›
using *iso-cblinfun-def* **by** *blast*
from *BA* **have** ‹*cblinfun-inv A* o_{CL} *A* = *id-cblinfun*›
by (*simp add*: *assms cblinfun-inv-left inv-cblinfun-invertible*)
with *AB BA* **have** ‹*cblinfun-inv A* = *B*›
by (*metis cblinfun-assoc-left*(*1*) *cblinfun-compose-id-right*)
with *AB* **show** ‹*A* o_{CL} *cblinfun-inv A* = *id-cblinfun*›
by *auto*
qed

lemma *cblinfun-inv-uniq*:
assumes *A* o_{CL} *B* = *id-cblinfun* **and** *B* o_{CL} *A* = *id-cblinfun*
shows *cblinfun-inv A* = *B*
using *assms* **by** (*metis inv-cblinfun-invertible cblinfun-compose-assoc cblinfun-compose-id-right cblinfun-inv-left iso-cblinfun-def*)

lemma *iso-cblinfun-unitary*: ‹*unitary A* ⟹ *iso-cblinfun A*›
using *iso-cblinfun-def unitary-def* **by** *blast*

lemma *invertible-cblinfun-isometry*: ‹*isometry A* ⟹ *invertible-cblinfun A*›
using *invertible-cblinfun-def isometryD* **by** *blast*

lemma *summable-cblinfun-apply-invertible*:
assumes ‹*invertible-cblinfun A*›
shows ‹(λ*x*. *A* $*_V$ *g x*) *summable-on S* ⟷ *g summable-on S*›
proof (*rule iffI*)
assume ‹*g summable-on S*›
then show ‹(λ*x*. *A* $*_V$ *g x*) *summable-on S*›

by (*rule summable-on-cblinfun-apply*)
next
assume ‹(λx. *A* ∗*V* *g x*) *summable-on S*›
then have ‹(λx. *cblinfun-inv A* ∗*V* *A* ∗*V* *g x*) *summable-on S*›
by (*rule summable-on-cblinfun-apply*)
then show ‹*g summable-on S*›
by (*simp add*: *cblinfun-inv-left assms flip*: *cblinfun-apply-cblinfun-compose*)
qed

lemma *infsum-cblinfun-apply-invertible*:
assumes ‹*invertible-cblinfun A*›
shows ‹(∑∞x∈*S*. *A* ∗*V* *g x*) = *A* ∗*V* (∑∞x∈*S*. *g x*)›
proof (*cases* ‹*g summable-on S*›)
case *True*
then show *?thesis*
by (*rule infsum-cblinfun-apply*)
next
case *False*
then have ‹¬ (λx. *A* ∗*V* *g x*) *summable-on S*›
using *assms* **by** (*simp add*: *summable-cblinfun-apply-invertible*)
with *False* **show** *?thesis*
by (*simp add*: *infsum-not-exists*)
qed

13.14 One-dimensional spaces

instantiation *cblinfun* :: (*one-dim*, *one-dim*) *complex-inner* **begin**

Once we have a theory for the trace, we could instead define the Hilbert-Schmidt inner product and relax the *one-dim*-sort constraint to (*cfinite-dim*,*complex-normed-vector*) or similar

definition *cinner-cblinfun* (*A*::′*a* ⇒*CL* ′*b*) (*B*::′*a* ⇒*CL* ′*b*)
= *cnj* (*one-dim-iso* (*A* ∗*V* *1*)) ∗ *one-dim-iso* (*B* ∗*V* *1*)
instance
proof *intro-classes*
fix *A B C* :: ′*a* ⇒*CL* ′*b*
and *c c′* :: *complex*
show (*A* •*C* *B*) = *cnj* (*B* •*C* *A*)
unfolding *cinner-cblinfun-def* **by** *auto*
show (*A* + *B*) •*C* *C* = (*A* •*C* *C*) + (*B* •*C* *C*)
by (*simp add*: *cinner-cblinfun-def algebra-simps plus-cblinfun.rep-eq*)
show (*c* ∗*C* *A* •*C* *B*) = *cnj c* ∗ (*A* •*C* *B*)
by (*simp add*: *cblinfun.scaleC-left cinner-cblinfun-def*)
show *0* ≤ (*A* •*C* *A*)
unfolding *cinner-cblinfun-def* **by** *auto*
have *bounded-clinear A* ⟹ *A 1* = *0* ⟹ *A* = (λ-. *0*)
for *A*::′*a* ⇒ ′*b*
proof (*rule one-dim-clinear-eqI* [**where** *x* = *1*] , *auto*)
show *clinear A*

```
      if bounded-clinear A
        and A 1 = 0
      for A :: 'a ⇒ 'b
      using that
      by (simp add: bounded-clinear.clinear)
    show clinear ((λ-. 0)::'a ⇒ 'b)
      if bounded-clinear A
        and A 1 = 0
      for A :: 'a ⇒ 'b
      using that
      by (simp add: complex-vector.module-hom-zero)
  qed
  hence A *_V 1 = 0 ⟹ A = 0
    by transfer
  hence one-dim-iso (A *_V 1) = 0 ⟹ A = 0
    by (metis one-dim-iso-of-zero one-dim-iso-inj)
  thus ((A ·_C A) = 0) = (A = 0)
    by (auto simp: cinner-cblinfun-def)

  show norm A = sqrt (cmod (A ·_C A))
    unfolding cinner-cblinfun-def
    by transfer (simp add: norm-mult abs-complex-def one-dim-onorm' cnj-x-x
power2-eq-square bounded-clinear.clinear)
qed
end

instantiation cblinfun :: (one-dim, one-dim) one-dim begin
lift-definition one-cblinfun :: 'a ⇒_CL 'b is one-dim-iso
  by (rule bounded-clinear-one-dim-iso)
lift-definition times-cblinfun :: 'a ⇒_CL 'b ⇒ 'a ⇒_CL 'b ⇒ 'a ⇒_CL 'b
  is λf g. f o one-dim-iso o g
  by (simp add: comp-bounded-clinear)
lift-definition inverse-cblinfun :: 'a ⇒_CL 'b ⇒ 'a ⇒_CL 'b is
  λf. ((*) (one-dim-iso (inverse (f 1)))) o one-dim-iso
  by (auto intro!: comp-bounded-clinear bounded-clinear-mult-right)
definition divide-cblinfun :: 'a ⇒_CL 'b ⇒ 'a ⇒_CL 'b ⇒ 'a ⇒_CL 'b where
  divide-cblinfun A B = A * inverse B
definition canonical-basis-cblinfun = [1 :: 'a ⇒_CL 'b]
definition ‹canonical-basis-length-cblinfun (- :: ('a ⇒_CL 'b) itself) = (1::nat)›
instance
proof intro-classes
  let ?basis = canonical-basis :: ('a ⇒_CL 'b) list
  fix A B C :: 'a ⇒_CL 'b
    and c c' :: complex
  show distinct ?basis
    unfolding canonical-basis-cblinfun-def by simp
  have (1::'a ⇒_CL 'b) ≠ (0::'a ⇒_CL 'b)
   by (metis cblinfun.zero-left one-cblinfun.rep-eq one-dim-iso-of-one zero-neq-one)
  thus cindependent (set ?basis)
```

```
    unfolding canonical-basis-cblinfun-def by simp

  have A ∈ cspan (set ?basis) for A
  proof −
    define c :: complex where c = one-dim-iso (A *V 1)
    have A x = one-dim-iso (A 1) *C one-dim-iso x for x
    by (smt (z3) cblinfun.scaleC-right complex-vector.scale-left-commute one-dim-iso-idem
one-dim-scaleC-1)
    hence A = one-dim-iso (A *V 1) *C 1
      by transfer metis
    thus A ∈ cspan (set ?basis)
      unfolding canonical-basis-cblinfun-def
     by (smt complex-vector.span-base complex-vector.span-scale list.set-intros(1))
  qed
  thus cspan (set ?basis) = UNIV by auto

  have A = (1::'a ⇒CL 'b) ⟹
    norm (1::'a ⇒CL 'b) = (1::real)
    by transfer simp
  thus A ∈ set ?basis ⟹ norm A = 1
    unfolding canonical-basis-cblinfun-def
    by simp

  show ?basis = [1]
    unfolding canonical-basis-cblinfun-def by simp
  show c *C 1 * c' *C 1 = (c * c') *C (1::'a⇒CL'b)
    by transfer auto
  have (1::'a ⇒CL 'b) = (0::'a ⇒CL 'b) ⟹ False
   by (metis cblinfun.zero-left one-cblinfun.rep-eq one-dim-iso-of-zero' zero-neq-neg-one)
  thus is-ortho-set (set ?basis)
    unfolding is-ortho-set-def canonical-basis-cblinfun-def by auto
  show A div B = A * inverse B
    by (simp add: divide-cblinfun-def)
  show inverse (c *C 1) = (1::'a⇒CL'b) /C c
    by transfer (simp add: o-def one-dim-inverse)
  show ‹canonical-basis-length TYPE('a ⇒CL 'b) = length (canonical-basis :: ('a
⇒CL 'b) list)›
    by (simp add: canonical-basis-length-cblinfun-def canonical-basis-cblinfun-def)
qed
end

lemma id-cblinfun-eq-1[simp]: ‹id-cblinfun = 1›
  by transfer auto

lemma one-dim-cblinfun-compose-is-times[simp]:
  fixes A :: 'a::one-dim ⇒CL 'a and B :: 'a ⇒CL 'a
  shows A oCL B = A * B
  by transfer simp
```

lemma *scaleC-one-dim-is-times*: ‹$r *_C x = one\text{-}dim\text{-}iso\ r * x$›
by *simp*

lemma *one-comp-one-cblinfun*[*simp*]: $1\ o_{CL}\ 1 = 1$
apply *transfer* **unfolding** *o-def* **by** *simp*

lemma *one-cblinfun-adj*[*simp*]: $1* = 1$
by *transfer simp*

lemma *scaleC-1-apply*[*simp*]: ‹$(x *_C 1) *_V y = x *_C y$›
by (*metis cblinfun.scaleC-left cblinfun-id-cblinfun-apply id-cblinfun-eq-1*)

lemma *cblinfun-apply-1-left*[*simp*]: ‹$1 *_V y = y$›
by (*metis cblinfun-id-cblinfun-apply id-cblinfun-eq-1*)

lemma *of-complex-cblinfun-apply*[*simp*]: ‹$of\text{-}complex\ x *_V y = one\text{-}dim\text{-}iso\ (x *_C y)$›
by (*metis of-complex-def cblinfun.scaleC-right one-cblinfun.rep-eq scaleC-cblinfun.rep-eq*)

lemma *cblinfun-compose-1-left*[*simp*]: ‹$1\ o_{CL}\ x = x$›
by *transfer auto*

lemma *cblinfun-compose-1-right*[*simp*]: ‹$x\ o_{CL}\ 1 = x$›
by *transfer auto*

lemma *one-dim-iso-id-cblinfun*: ‹$one\text{-}dim\text{-}iso\ id\text{-}cblinfun = id\text{-}cblinfun$›
by *simp*

lemma *one-dim-iso-id-cblinfun-eq-1*: ‹$one\text{-}dim\text{-}iso\ id\text{-}cblinfun = 1$›
by *simp*

lemma *one-dim-iso-comp-distr*[*simp*]: ‹$one\text{-}dim\text{-}iso\ (a\ o_{CL}\ b) = one\text{-}dim\text{-}iso\ a\ o_{CL}\ one\text{-}dim\text{-}iso\ b$›
by (*smt* (*z3*) *cblinfun-compose-scaleC-left cblinfun-compose-scaleC-right one-cinner-a-scaleC-one one-comp-one-cblinfun one-dim-iso-of-one one-dim-iso-scaleC*)

lemma *one-dim-iso-comp-distr-times*[*simp*]: ‹$one\text{-}dim\text{-}iso\ (a\ o_{CL}\ b) = one\text{-}dim\text{-}iso\ a * one\text{-}dim\text{-}iso\ b$›
by (*smt* (*verit*, *del-insts*) *mult.left-neutral mult-scaleC-left one-cinner-a-scaleC-one one-comp-one-cblinfun one-dim-iso-of-one one-dim-iso-scaleC cblinfun-compose-scaleC-right cblinfun-compose-scaleC-left*)

lemma *one-dim-iso-adjoint*[*simp*]: ‹$one\text{-}dim\text{-}iso\ (A*) = (one\text{-}dim\text{-}iso\ A)*$›
by (*smt* (*z3*) *one-cblinfun-adj one-cinner-a-scaleC-one one-dim-iso-of-one one-dim-iso-scaleC scaleC-adj*)

lemma *one-dim-iso-adjoint-complex*[*simp*]: ‹$one\text{-}dim\text{-}iso\ (A*) = cnj\ (one\text{-}dim\text{-}iso\ A)$›
by (*metis* (*mono-tags*, *lifting*) *one-cblinfun-adj one-dim-iso-idem one-dim-scaleC-1*

scaleC-adj)

lemma *one-dim-cblinfun-compose-commute*: ‹*a* o_{CL} *b* = *b* o_{CL} *a*› **for** *a b* :: ‹('*a*::*one-dim*,'*a*) *cblinfun*›
by (*simp add*: *one-dim-iso-inj*)

lemma *one-cblinfun-apply-one*[*simp*]: ‹*1* $*_V$ *1* = *1*›
by (*simp add*: *one-cblinfun.rep-eq*)

lemma *one-dim-cblinfun-apply-is-times*:
fixes *A* :: '*a*::*one-dim* $\Rightarrow_{CL}$ '*b*::*one-dim* **and** *b* :: '*a*
shows *A* $*_V$ *b* = *one-dim-iso A* * *one-dim-iso b*
apply (*subst one-dim-scaleC-1*[*of A*, *symmetric*])
apply (*subst one-dim-scaleC-1*[*of b*, *symmetric*])
apply (*simp only*: *cblinfun.scaleC-left cblinfun.scaleC-right*)
by *simp*

lemma *is-onb-one-dim*[*simp*]: ‹*norm x* = *1* $\Longrightarrow$ *is-onb* {*x*}› **for** *x* :: ‹- :: *one-dim*›
by (*auto simp*: *is-onb-def intro*!: *ccspan-one-dim*)

lemma *one-dim-iso-cblinfun-comp*: ‹*one-dim-iso* (*a* o_{CL} *b*) = *of-complex* (*cinner* (*a** $*_V$ *1*) (*b* $*_V$ *1*))›
for *a* :: ‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*b*::*one-dim*› **and** *b* :: ‹'*c*::*one-dim* $\Rightarrow_{CL}$ '*a*›
by (*simp add*: *cinner-adj-left cinner-cblinfun-def one-dim-iso-def*)

lemma *one-dim-iso-cblinfun-apply*[*simp*]: ‹*one-dim-iso* ψ $*_V$ φ = *one-dim-iso* (*one-dim-iso* ψ $*_C$ φ)›
by (*smt* (*verit*) *cblinfun.scaleC-left one-cblinfun.rep-eq one-dim-iso-of-one one-dim-iso-scaleC one-dim-scaleC-1*)

13.15 Loewner order

lift-definition *heterogenous-cblinfun-id* :: ‹'*a*::*complex-normed-vector* $\Rightarrow_{CL}$ '*b*::*complex-normed-vector*›
is ‹*if bounded-clinear* (*heterogenous-identity* :: '*a*::*complex-normed-vector* $\Rightarrow$ '*b*::*complex-normed-vector*) *then heterogenous-identity else* (λ-. *0*)›
by *auto*

lemma *heterogenous-cblinfun-id-def'*[*simp*]: *heterogenous-cblinfun-id* = *id-cblinfun*
by *transfer auto*

definition *heterogenous-same-type-cblinfun* (*x*::'*a*::*chilbert-space itself*) (*y*::'*b*::*chilbert-space itself*) $\longleftrightarrow$
unitary (*heterogenous-cblinfun-id* :: '*a* $\Rightarrow_{CL}$ '*b*) $\wedge$ *unitary* (*heterogenous-cblinfun-id* :: '*b* $\Rightarrow_{CL}$ '*a*)

lemma *heterogenous-same-type-cblinfun*[*simp*]: ‹*heterogenous-same-type-cblinfun* (*x*::'*a*::*chilbert-space itself*) (*y*::'*a*::*chilbert-space itself*)›
unfolding *heterogenous-same-type-cblinfun-def* **by** *auto*

instantiation *cblinfun* :: (*chilbert-space*, *chilbert-space*) *ord* **begin**
definition *less-eq-cblinfun-def-heterogenous*: ‹$A \leq B \longleftrightarrow$
(*if heterogenous-same-type-cblinfun TYPE('a) TYPE('b) then*
$\forall \psi$::'b. $\psi \cdot_C ((B-A) *_V$ *heterogenous-cblinfun-id* $*_V \psi) \geq 0$ *else* $(A=B)$)›
definition ‹$(A :: 'a \Rightarrow_{CL} 'b) < B \longleftrightarrow A \leq B \wedge \neg B \leq A$›
instance..
end

lemma *less-eq-cblinfun-def*: ‹$A \leq B \longleftrightarrow$
$(\forall \psi.\ \psi \cdot_C (A *_V \psi) \leq \psi \cdot_C (B *_V \psi))$›
unfolding *less-eq-cblinfun-def-heterogenous*
by (*auto simp del*: *less-eq-complex-def simp*: *cblinfun.diff-left cinner-diff-right*)

instantiation *cblinfun* :: (*chilbert-space*, *chilbert-space*) *ordered-complex-vector* **begin**
instance
proof *intro-classes*
fix *x y z* :: ‹'a $\Rightarrow_{CL}$ 'b›
fix *a b* :: *complex*

define *pos* **where** ‹*pos X* $\longleftrightarrow (\forall \psi.$ *cinner* ψ $(X *_V \psi) \geq 0)$› **for** $X ::$ ‹'b $\Rightarrow_{CL}$ 'b›
consider (*unitary*) ‹*heterogenous-same-type-cblinfun TYPE('a) TYPE('b)*›
‹$\bigwedge A\ B ::$ 'a $\Rightarrow_{CL}$ 'b. $A \leq B =$ *pos* $((B-A)\ o_{CL}$ (*heterogenous-cblinfun-id* :: 'b$\Rightarrow_{CL}$'a))›
| (*trivial*) ‹$\bigwedge A\ B ::$ 'a $\Rightarrow_{CL}$ 'b. $A \leq B \longleftrightarrow A = B$›
by *atomize-elim* (*auto simp*: *pos-def less-eq-cblinfun-def-heterogenous*)
note *cases* = *this*

have [*simp*]: ‹*pos* 0›
unfolding *pos-def* **by** *auto*

have *pos-nondeg*: ‹$X = 0$› **if** ‹*pos X*› **and** ‹*pos* $(-X)$› **for** *X*
apply (*rule cblinfun-cinner-eqI*, *simp*)
using *that* **by** (*metis* (*no-types*, *lifting*) *cblinfun.minus-left cinner-minus-right dual-order.antisym equation-minus-iff neg-le-0-iff-le pos-def*)

have *pos-add*: ‹*pos* $(X+Y)$› **if** ‹*pos X*› **and** ‹*pos Y*› **for** *X Y*
by (*smt* (*z3*) *pos-def cblinfun.diff-left cinner-minus-right cinner-simps*(*3*) *diff-ge-0-iff-ge diff-minus-eq-add neg-le-0-iff-le order-trans that*(*1*) *that*(*2*) *uminus-cblinfun.rep-eq*)

have *pos-scaleC*: ‹*pos* $(a *_C X)$› **if** ‹$a \geq 0$› **and** ‹*pos X*› **for** *X a*
using *that* **unfolding** *pos-def* **by** (*auto simp*: *cblinfun.scaleC-left*)

let *?id* = ‹*heterogenous-cblinfun-id* :: 'b $\Rightarrow_{CL}$ 'a›

show ‹$x \leq x$›
apply (*cases rule*:*cases*) **by** *auto*
show ‹$(x < y) \longleftrightarrow (x \leq y \wedge \neg\ y \leq x)$›

unfolding *less-cblinfun-def* **by** *simp*
show ‹$x \leq z$› **if** ‹$x \leq y$› **and** ‹$y \leq z$›
proof (*cases rule*:*cases*)
case *unitary*
define *a b* :: ‹'*b* $\Rightarrow_{CL}$ '*b*› **where** ‹$a = (y-x)$ o_{CL} *heterogenous-cblinfun-id*›
and ‹$b = (z-y)$ o_{CL} *heterogenous-cblinfun-id*›
with *unitary that* **have** ‹*pos a*› **and** ‹*pos b*›
by *auto*
then have ‹*pos* $(a + b)$›
by (*rule pos-add*)
moreover have ‹$a + b = (z - x)$ o_{CL} *heterogenous-cblinfun-id*›
unfolding *a-def b-def*
by (*metis* (*no-types*, *lifting*) *bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose diff-add-cancel ordered-field-class.sign-simps*(*2*) *ordered-field-class.sign-simps*(*8*))
ultimately show *?thesis*
using *unitary* **by** *auto*
next
case *trivial*
with *that* **show** *?thesis* **by** *auto*
qed
show ‹$x = y$› **if** ‹$x \leq y$› **and** ‹$y \leq x$›
proof (*cases rule*:*cases*)
case *unitary*
then have ‹*unitary ?id*›
by (*auto simp*: *heterogenous-same-type-cblinfun-def*)
define *a b* :: ‹'*b* $\Rightarrow_{CL}$ '*b*› **where** ‹$a = (y-x)$ o_{CL} *?id*›
and ‹$b = (x-y)$ o_{CL} *?id*›
with *unitary that* **have** ‹*pos a*› **and** ‹*pos b*›
by *auto*
then have ‹$a = 0$›
apply (*rule-tac pos-nondeg*)
apply (*auto simp*: *a-def b-def*)
by (*smt* (*verit*, *best*) *add.commute bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose cblinfun-compose-zero-left diff-0 diff-add-cancel group-cancel.rule0 group-cancel.sub1*)
then show *?thesis*
unfolding *a-def* **using** ‹*unitary ?id*›
by (*metis cblinfun-compose-assoc cblinfun-compose-id-right cblinfun-compose-zero-left eq-iff-diff-eq-0 unitaryD2*)
next
case *trivial*
with *that* **show** *?thesis* **by** *simp*
qed
show ‹$x + y \leq x + z$› **if** ‹$y \leq z$›
proof (*cases rule*:*cases*)
case *unitary*
with *that* **show** *?thesis*
by *auto*
next
case *trivial*

```
    with that show ?thesis
      by auto
  qed

  show ‹a *C x ≤ a *C y› if ‹x ≤ y› and ‹0 ≤ a›
  proof (cases rule:cases)
    case unitary
    with that pos-scaleC show ?thesis
      by (metis cblinfun-compose-scaleC-left complex-vector.scale-right-diff-distrib)
  next
    case trivial
    with that show ?thesis
      by auto
  qed

  show ‹a *C x ≤ b *C x› if ‹a ≤ b› and ‹0 ≤ x›
  proof (cases rule:cases)
    case unitary
    with that show ?thesis
      by (auto intro!: pos-scaleC simp flip: scaleC-diff-left)
  next
    case trivial
    with that show ?thesis
      by auto
  qed
qed
end

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

lemma positive-id-cblinfun[simp]: id-cblinfun ≥ 0
  unfolding less-eq-cblinfun-def using cinner-ge-zero by auto

lemma positive-selfadjointI: ‹selfadjoint A› if ‹A ≥ 0›
  apply (rule cinner-real-selfadjointI)
  using that by (auto simp: complex-is-real-iff-compare0 less-eq-cblinfun-def)

lemma cblinfun-leI:
  assumes ‹⋀x. norm x = 1 ⟹ x •C (A *V x) ≤ x •C (B *V x)›
  shows ‹A ≤ B›
proof (unfold less-eq-cblinfun-def, intro allI, case-tac ‹ψ = 0›)
  fix ψ :: 'a assume ‹ψ = 0›
  then show ‹ψ •C (A *V ψ) ≤ ψ •C (B *V ψ)›
    by simp
next
  fix ψ :: 'a assume ‹ψ ≠ 0›
  define φ where ‹φ = ψ /R norm ψ›
  have ‹φ •C (A *V φ) ≤ φ •C (B *V φ)›
```

apply (*rule assms*)
unfolding φ-*def*
by (*simp add*: ‹$\psi \neq 0$›)
with ‹$\psi \neq 0$› **show** ‹$\psi \cdot_C (A *_V \psi) \leq \psi \cdot_C (B *_V \psi)$›
unfolding φ-*def*
by (*smt* (*verit*) *cinner-adj-left cinner-scaleR-left cinner-simps*(6) *complex-of-real-nn-iff mult-cancel-right1 mult-left-mono norm-eq-zero norm-ge-zero of-real-1 right-inverse scaleR-scaleC scaleR-scaleR*)
qed

lemma *positive-cblinfunI*: ‹$A \geq 0$› **if** ‹$\bigwedge x.\ norm\ x = 1 \Longrightarrow cinner\ x\ (A *_V x) \geq 0$›
apply (*rule cblinfun-leI*)
using *that* **by** *simp*

lemma *less-eq-scaled-id-norm*:
assumes ‹*norm* $A \leq c$› **and** ‹*selfadjoint* A›
shows ‹$A \leq c *_R$ *id-cblinfun*›
proof −
have ‹$x \cdot_C (A *_V x) \leq$ *complex-of-real* c› **if** ‹*norm* $x = 1$› **for** x
proof −
have ‹*norm* $(x \cdot_C (A *_V x)) \leq$ *norm* $(A *_V x)$›
by (*metis complex-inner-class.Cauchy-Schwarz-ineq2 mult-cancel-right1 that*)
also have ‹$\ldots \leq$ *norm* A›
by (*metis more-arith-simps*(6) *norm-cblinfun that*)
also have ‹$\ldots \leq c$›
by (*rule assms*)
finally have ‹*norm* $(x \cdot_C (A *_V x)) \leq c$›
by −
moreover have ‹$x \cdot_C (A *_V x) \in \mathbb{R}$›
by (*metis assms*(2) *cinner-selfadjoint-real*)
ultimately show *?thesis*
by (*smt* (*verit*) *Re-complex-of-real Reals-cases complex-of-real-nn-iff less-eq-complex-def norm-of-real reals-zero-comparable*)
qed
then show *?thesis*
by (*smt* (*verit*) *cblinfun.scaleC-left cblinfun-id-cblinfun-apply cblinfun-leI cinner-scaleC-right cnorm-eq-1 mult-cancel-left2 scaleR-scaleC*)
qed

lemma *positive-cblinfun-squareI*: ‹$A = B* \circ_{CL} B \Longrightarrow A \geq 0$›
apply (*rule positive-cblinfunI*)
by (*metis cblinfun-apply-cblinfun-compose cinner-adj-right cinner-ge-zero*)

lemma *one-dim-loewner-order*: ‹$A \geq B \longleftrightarrow$ *one-dim-iso* $A \geq$ (*one-dim-iso* B :: *complex*)› **for** $A\ B$ :: ‹$'a \Rightarrow_{CL} 'a$::{*chilbert-space*, *one-dim*}›
proof −
have A: ‹$A =$ *one-dim-iso* $A *_C$ *id-cblinfun*›

by *simp*
have *B*: ‹B = *one-dim-iso B* $*_C$ *id-cblinfun*›
by *simp*
have ‹$A \geq B \longleftrightarrow (\forall \psi.$ *cinner* ψ (A ψ) $\geq$ *cinner* ψ (B ψ))›
by (*simp add*: *less-eq-cblinfun-def*)
also have ‹... $\longleftrightarrow (\forall \psi$::'*a*. *one-dim-iso B* $*$ (ψ $\cdot_C$ ψ) $\leq$ *one-dim-iso A* $*$ (ψ $\cdot_C$ ψ))›
apply (*subst A*, *subst B*)
by (*metis* (*no-types*, *opaque-lifting*) *cinner-scaleC-right id-cblinfun-apply scaleC-cblinfun.rep-eq*)
also have ‹... $\longleftrightarrow$ *one-dim-iso A* $\geq$ (*one-dim-iso B* :: *complex*)›
by (*auto intro*!: *mult-right-mono elim*!: *allE*[**where** *x=1*])
finally show *?thesis*
by −
qed

lemma *one-dim-positive*: ‹$A \geq 0 \longleftrightarrow$ *one-dim-iso A* $\geq$ (*0*::*complex*)› **for** A :: ‹'*a* $\Rightarrow_{CL}$ '*a*::{*chilbert-space*, *one-dim*}›
using *one-dim-loewner-order*[**where** *B=0*] **by** *auto*

lemma *op-square-nondegenerate*: ‹$a = 0$› **if** ‹$a* \ o_{CL}\ a = 0$›
proof (*rule cblinfun-eq-0-on-UNIV-span*[**where** *basis=UNIV*]; *simp*)
fix *s*
from *that* **have** ‹$s \cdot_C ((a* \ o_{CL}\ a) *_V\ s) = 0$›
by *simp*
then have ‹$(a *_V\ s) \cdot_C (a *_V\ s) = 0$›
by (*simp add*: *cinner-adj-right*)
then show ‹$a *_V\ s = 0$›
by *simp*
qed

lemma *comparable-selfadjoint*:
assumes ‹$a \leq b$›
assumes ‹*selfadjoint a*›
shows ‹*selfadjoint b*›
by (*smt* (*verit*, *best*) *assms*(*1*) *assms*(*2*) *cinner-selfadjoint-real cinner-real-selfadjointI comparable complex-is-real-iff-compare0 less-eq-cblinfun-def selfadjoint-def*)

lemma *comparable-selfadjoint'*:
assumes ‹$a \leq b$›
assumes ‹*selfadjoint b*›
shows ‹*selfadjoint a*›
by (*smt* (*verit*, *best*) *assms*(*1*) *assms*(*2*) *cinner-selfadjoint-real cinner-real-selfadjointI comparable complex-is-real-iff-compare0 less-eq-cblinfun-def selfadjoint-def*)

lemma *is-Proj-leq-id*: ‹*is-Proj P* $\Longrightarrow$ $P \leq$ *id-cblinfun*›
by (*metis diff-ge-0-iff-ge is-Proj-algebraic is-Proj-complement positive-cblinfun-squareI*)

lemma *Proj-mono*: ‹*Proj S* $\leq$ *Proj T* $\longleftrightarrow$ $S \leq T$›
proof (*rule iffI*)

assume ‹$S \leq T$›
define D **where** ‹$D = Proj\ T - Proj\ S$›
from ‹$S \leq T$› **have** *TS-S*[*simp*]: ‹$Proj\ T\ o_{CL}\ Proj\ S = Proj\ S$›
by (*smt* (*verit*, *ccfv-threshold*) *Proj-idempotent Proj-range cblinfun-apply-cblinfun-compose cblinfun-apply-in-image cblinfun-eqI cblinfun-fixes-range less-eq-ccsubspace.rep-eq subset-iff*)
then have *ST-S*[*simp*]: ‹$Proj\ S\ o_{CL}\ Proj\ T = Proj\ S$›
by (*metis adj-Proj adj-cblinfun-compose*)
have ‹$D* \ o_{CL}\ D = D$›
by (*simp add*: *D-def cblinfun-compose-minus-left cblinfun-compose-minus-right adj-minus adj-Proj*)
then have ‹$D \geq 0$›
by (*metis positive-cblinfun-squareI*)
then show ‹$Proj\ S \leq Proj\ T$›
by (*simp add*: *D-def*)
next
assume *PS-PT*: ‹$Proj\ S \leq Proj\ T$›
show ‹$S \leq T$›
proof (*rule ccsubspace-leI-unit*)
fix ψ **assume** ‹$\psi \in space\text{-}as\text{-}set\ S$› **and** [*simp*]: ‹$norm\ \psi = 1$›
then have ‹$1 = norm\ (Proj\ S *_V \psi)$›
by (*simp add*: *Proj-fixes-image*)
also from *PS-PT* **have** ‹$\ldots \leq norm\ (Proj\ T *_V \psi)$›
by (*metis* (*no-types*, *lifting*) *Proj-idempotent adj-Proj cblinfun-apply-cblinfun-compose cinner-adj-left cnorm-le less-eq-cblinfun-def*)
also have ‹$\ldots \leq 1$›
by (*metis Proj-is-Proj* ‹$norm\ \psi = 1$› *is-Proj-reduces-norm*)
ultimately have ‹$norm\ (Proj\ T *_V \psi) = 1$›
by *auto*
then show ‹$\psi \in space\text{-}as\text{-}set\ T$›
by (*simp add*: *norm-Proj-apply-1*)
qed
qed

lemma *has-sum-mono-neutral-cblinfun*:
fixes $f :: 'a \Rightarrow ('b{::}chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes ‹$(f\ has\text{-}sum\ a)\ A$› **and** $(g\ has\text{-}sum\ b)\ B$
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*
have *sum-hfh*: ‹$((\lambda x.\ h \cdot_C f\ x\ h)\ has\text{-}sum\ h \cdot_C a\ h)\ A$› **for** h
by (*intro has-sum-cinner-left has-sum-cblinfun-apply-left*)
from *assms*
have *sum-hgh*: ‹$((\lambda x.\ h \cdot_C g\ x\ h)\ has\text{-}sum\ h \cdot_C b\ h)\ B$› **for** h
by (*intro has-sum-cinner-left has-sum-cblinfun-apply-left*)
from *sum-hfh sum-hgh*

have ‹$h \cdot_C a\ h \leq h \cdot_C b\ h$› **for** h
apply (*rule has-sum-mono-neutral-complex*)
using *assms*
by (*auto intro*!: *simp*: *less-eq-cblinfun-def*)
then show ‹$a \leq b$›
by (*simp add*: *less-eq-cblinfun-def*)
qed

lemma *sums-mono-cblinfun*:
fixes $f :: nat \Rightarrow ('b::chilbert\text{-}space \Rightarrow_{CL} 'b)$
assumes ‹f *sums* a› **and** g *sums* b
assumes ‹$\bigwedge n.\ f\ n \leq g\ n$›
shows $a \leq b$
proof (*rule cblinfun-leI*)
fix h
from ‹f *sums* a›
have *sum1*: ‹$(\lambda n.\ h \cdot_C (f\ n *_V h))$ *sums* $(h \cdot_C (a *_V h))$›
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*: ‹$(\lambda n.\ h \cdot_C (g\ n *_V h))$ *sums* $(h \cdot_C (b *_V h))$›
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 ‹$h \cdot_C (f\ n *_V h) \leq h \cdot_C (g\ n *_V h)$› **for** n
using *assms*(*3*) *less-eq-cblinfun-def* **by** *auto*
with *sum1 sum2*
show ‹$h \cdot_C (a *_V h) \leq h \cdot_C (b *_V h)$›
by (*rule sums-le-complex*[*rotated*])
qed

13.16 Embedding vectors to operators

lift-definition *vector-to-cblinfun* :: ‹$'a::complex\text{-}normed\text{-}vector \Rightarrow 'b::one\text{-}dim \Rightarrow_{CL} 'a$› **is**
‹$\lambda\psi\ \varphi.\ one\text{-}dim\text{-}iso\ \varphi *_C \psi$›
by (*simp add*: *bounded-clinear-scaleC-const*)

lemma *vector-to-cblinfun-apply*[*simp*]: ‹*vector-to-cblinfun* $\psi *_V \varphi = one\text{-}dim\text{-}iso\ \psi *_C \varphi$›
apply (*transfer fixing*: $\psi\ \varphi$)
by *simp*

lemma *vector-to-cblinfun-cblinfun-compose*[*simp*]:
$A\ \ o_{CL}\ (vector\text{-}to\text{-}cblinfun\ \psi) = vector\text{-}to\text{-}cblinfun\ (A *_V \psi)$
apply *transfer*
unfolding *comp-def bounded-clinear-def clinear-def Vector-Spaces.linear-def module-hom-def module-hom-axioms-def*

by *simp*

lemma *vector-to-cblinfun-add*: ‹*vector-to-cblinfun* $(x + y)$ = *vector-to-cblinfun* x + *vector-to-cblinfun* y›
by *transfer* (*simp add*: *scaleC-add-right*)

lemma *norm-vector-to-cblinfun*[*simp*]: *norm* (*vector-to-cblinfun* x) = *norm* x
proof *transfer*
have *bounded-clinear* (*one-dim-iso*::$'a \Rightarrow$ *complex*)
by *simp*
moreover have *onorm* (*one-dim-iso*::$'a \Rightarrow$ *complex*) $*$ *norm* x = *norm* x
for x :: $'b$
by *simp*
ultimately show *onorm* ($\lambda\varphi$. *one-dim-iso* (φ::$'a$) $*_C$ x) = *norm* x
for x :: $'b$
by (*subst onorm-scaleC-left*)
qed

lemma *bounded-clinear-vector-to-cblinfun*[*bounded-clinear*]: *bounded-clinear vector-to-cblinfun*
apply (*rule bounded-clinearI*[**where** *K=1*])
apply (*transfer*, *simp add*: *scaleC-add-right*)
apply (*transfer*, *simp add*: *mult.commute*)
by *simp*

lemma *vector-to-cblinfun-scaleC*[*simp*]:
vector-to-cblinfun ($a *_C \psi$) = $a *_C$ *vector-to-cblinfun* ψ **for** a::*complex*
by (*intro clinear.scaleC bounded-clinear.clinear bounded-clinear-vector-to-cblinfun*)

lemma *vector-to-cblinfun-apply-one-dim*[*simp*]:
shows *vector-to-cblinfun* $\varphi *_V \gamma$ = *one-dim-iso* $\gamma *_C \varphi$
by *transfer* (*rule refl*)

lemma *vector-to-cblinfun-one-dim-iso*[*simp*]: ‹*vector-to-cblinfun* = *one-dim-iso*›
by (*auto intro*!: *ext cblinfun-eqI*)

lemma *vector-to-cblinfun-adj-apply*[*simp*]:
shows *vector-to-cblinfun* $\psi* *_V \varphi$ = *of-complex* (*cinner* ψ φ)
by (*simp add*: *cinner-adj-right one-dim-iso-def one-dim-iso-inj*)

lemma *vector-to-cblinfun-comp-one*[*simp*]:
(*vector-to-cblinfun* s :: $'a$::*one-dim* $\Rightarrow_{CL}$ -) o_{CL} *1*
= (*vector-to-cblinfun* s :: $'b$::*one-dim* $\Rightarrow_{CL}$ -)
apply (*transfer fixing*: s)
by *fastforce*

lemma *vector-to-cblinfun-0*[*simp*]: *vector-to-cblinfun 0 = 0*
by (*metis cblinfun.zero-left cblinfun-compose-zero-left vector-to-cblinfun-cblinfun-compose*)

lemma *image-vector-to-cblinfun*[*simp*]: *vector-to-cblinfun* $x *_S \top$ = *ccspan* $\{x\}$

— Not that the general case *vector-to-cblinfun x* $*_S$ *S* can be handled by using that $S = \top$ or $S = \bot$ by *one-dim-ccsubspace-all-or-nothing*
proof *transfer*
show *closure* (*range* ($\lambda\varphi$::'*b*. *one-dim-iso* φ $*_C$ *x*)) = *closure* (*cspan* {*x*})
for *x* :: '*a*
proof (*rule arg-cong* [**where** *f* = *closure*])
have *k* $*_C$ *x* ∈ *range* ($\lambda\varphi$. *one-dim-iso* φ $*_C$ *x*) **for** *k*
by (*smt* (*z3*) *id-apply one-dim-iso-id one-dim-iso-idem range-eqI*)
thus *range* ($\lambda\varphi$. *one-dim-iso* (φ::'*b*) $*_C$ *x*) = *cspan* {*x*}
unfolding *complex-vector.span-singleton*
by *auto*
qed
qed

lemma *vector-to-cblinfun-adj-comp-vector-to-cblinfun*[*simp*]:
shows *vector-to-cblinfun* $\psi*$ o_{CL} *vector-to-cblinfun* φ = *cinner* ψ φ $*_C$ *id-cblinfun*
proof −
have *one-dim-iso* γ $*_C$ *one-dim-iso* (*of-complex* (ψ $\cdot_C$ φ)) =
(ψ $\cdot_C$ φ) $*_C$ *one-dim-iso* γ
for γ :: '*c*::*one-dim*
by (*metis complex-vector.scale-left-commute of-complex-def one-dim-iso-of-one one-dim-iso-scaleC one-dim-scaleC-1*)
hence *one-dim-iso* ((*vector-to-cblinfun* $\psi*$ o_{CL} *vector-to-cblinfun* φ) $*_V$ γ)
= *one-dim-iso* ((*cinner* ψ φ $*_C$ *id-cblinfun*) $*_V$ γ)
for γ :: '*c*::*one-dim*
by *simp*
hence ((*vector-to-cblinfun* $\psi*$ o_{CL} *vector-to-cblinfun* φ) $*_V$ γ) = ((*cinner* ψ φ $*_C$ *id-cblinfun*) $*_V$ γ)
for γ :: '*c*::*one-dim*
by (*rule one-dim-iso-inj*)
thus *?thesis*
using *cblinfun-eqI*[**where** *x* = *vector-to-cblinfun* $\psi*$ o_{CL} *vector-to-cblinfun* φ
and *y* = (ψ $\cdot_C$ φ) $*_C$ *id-cblinfun*]
by *auto*
qed

lemma *isometry-vector-to-cblinfun*[*simp*]:
assumes *norm x* = *1*
shows *isometry* (*vector-to-cblinfun x*)
using *assms cnorm-eq-1 isometry-def* **by** *force*

lemma *image-vector-to-cblinfun-adj*:
assumes ‹ψ ∉ *space-as-set* (− *S*)›
shows ‹(*vector-to-cblinfun* ψ)* $*_S$ *S* = ⊤›
proof −
from *assms* **obtain** φ **where** ‹φ ∈ *space-as-set S*› **and** ‹¬ *is-orthogonal* ψ φ›
by (*metis orthogonal-complementI uminus-ccsubspace.rep-eq*)
have ‹((*vector-to-cblinfun* ψ)* $*_S$ *S* :: '*b ccsubspace*) ≥ (*vector-to-cblinfun* ψ)* $*_S$ *ccspan* {φ}› (**is** ‹- ≥ ...›)

by (*simp add*: ‹$\varphi \in$ *space-as-set* S› *cblinfun-image-mono ccspan-leqI*)
also have ‹$\ldots$ = *ccspan* {(*vector-to-cblinfun* ψ)* $*_V$ φ}›
by (*auto simp*: *cblinfun-image-ccspan*)
also have ‹$\ldots$ = *ccspan* {*of-complex* (ψ $\cdot_C$ φ)}›
by *auto*
also have ‹$\ldots$ > $\bot$›
by (*simp add*: ‹ψ $\cdot_C$ $\varphi \neq 0$› *flip*: *bot.not-eq-extremum*)
finally(*dual-order.strict-trans1*) **show** *?thesis*
using *one-dim-ccsubspace-all-or-nothing bot.not-eq-extremum* **by** *auto*
qed

lemma *image-vector-to-cblinfun-adj'*:
assumes ‹$\psi \neq 0$›
shows ‹(*vector-to-cblinfun* ψ)* $*_S$ $\top$ = $\top$›
apply (*rule image-vector-to-cblinfun-adj*)
using *assms* **by** *simp*

lemma *vector-to-cblinfun-inj*: ‹*inj-on* (*vector-to-cblinfun* :: 'a::*complex-normed-vector* ⇒ 'b::*one-dim* ⇒$_{CL}$ -) *X*›
proof (*rule inj-onI*)
fix *x y* :: 'a
assume ‹*vector-to-cblinfun x* = (*vector-to-cblinfun y* :: 'b ⇒$_{CL}$ -)›
then have ‹*vector-to-cblinfun x* (1::'b) = *vector-to-cblinfun y* (1::'b)›
by *simp*
then show ‹*x* = *y*›
by *simp*
qed

13.17 Rank-1 operators / butterflies

definition *rank1* **where** ‹*rank1 A* ⟷ (∃ψ. *A* $*_S$ $\top$ = *ccspan* {ψ})›
— This is not the usual definition of a rank-1 operator. The usual definition is an operator with 1-dim image. Here we define it as an operator with 0- or 1-dim image. This makes the definition simpler to use. The normal definition of rank-1 operators then corresponds to the non-zero *rank1* operators.

definition *butterfly* (*s*::'a::*complex-normed-vector*) (*t*::'b::*chilbert-space*)
= *vector-to-cblinfun s* o_{CL} (*vector-to-cblinfun t* :: *complex* ⇒$_{CL}$ -)*

abbreviation *selfbutter s* ≡ *butterfly s s*

lemma *butterfly-add-left*: ‹*butterfly* (*a* + *a'*) *b* = *butterfly a b* + *butterfly a' b*›
by (*simp add*: *butterfly-def vector-to-cblinfun-add cbilinear-add-left bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose*)

lemma *butterfly-add-right*: ‹*butterfly a* (*b* + *b'*) = *butterfly a b* + *butterfly a b'*›
by (*simp add*: *butterfly-def adj-plus vector-to-cblinfun-add cblinfun-compose-add-right*)

lemma *butterfly-def-one-dim*: *butterfly s t* = (*vector-to-cblinfun s* :: *'c*::*one-dim* $\Rightarrow_{CL}$ -)
o_{CL} (*vector-to-cblinfun t* :: *'c* $\Rightarrow_{CL}$ -)*
(**is** - = *?rhs*) **for** *s* :: *'a*::*complex-normed-vector* **and** *t* :: *'b*::*chilbert-space*
proof −
let *?isoAC* = *1* :: *'c* $\Rightarrow_{CL}$ *complex*
let *?isoCA* = *1* :: *complex* $\Rightarrow_{CL}$ *'c*
let *?vector* = *vector-to-cblinfun* :: - ⇒ (*'c* $\Rightarrow_{CL}$ -)

have *butterfly s t* =
(*?vector s* o_{CL} *?isoCA*) o_{CL} (*?vector t* o_{CL} *?isoCA*)*
unfolding *butterfly-def vector-to-cblinfun-comp-one* **by** *simp*
also have ... = *?vector s* o_{CL} (*?isoCA* o_{CL} *?isoCA**) o_{CL} (*?vector t*)*
by (*metis* (*no-types*, *lifting*) *cblinfun-compose-assoc adj-cblinfun-compose*)
also have ... = *?rhs*
by *simp*
finally show *?thesis*
by *simp*
qed

lemma *butterfly-comp-cblinfun*: *butterfly* ψ φ o_{CL} *a* = *butterfly* ψ (*a** $*_V$ φ)
unfolding *butterfly-def*
by (*simp add*: *cblinfun-compose-assoc flip*: *vector-to-cblinfun-cblinfun-compose*)

lemma *cblinfun-comp-butterfly*: *a* o_{CL} *butterfly* ψ φ = *butterfly* (*a* $*_V$ ψ) φ
unfolding *butterfly-def*
by (*simp add*: *cblinfun-compose-assoc flip*: *vector-to-cblinfun-cblinfun-compose*)

lemma *butterfly-apply*[*simp*]: *butterfly* ψ ψ' $*_V$ φ = (ψ' $\cdot_C$ φ) $*_C$ ψ
by (*simp add*: *butterfly-def scaleC-cblinfun.rep-eq*)

lemma *butterfly-scaleC-left*[*simp*]: *butterfly* (*c* $*_C$ ψ) φ = *c* $*_C$ *butterfly* ψ φ
unfolding *butterfly-def vector-to-cblinfun-scaleC scaleC-adj*
by (*simp add*: *cnj-x-x*)

lemma *butterfly-scaleC-right*[*simp*]: *butterfly* ψ (*c* $*_C$ φ) = *cnj c* $*_C$ *butterfly* ψ φ
unfolding *butterfly-def vector-to-cblinfun-scaleC scaleC-adj*
by (*simp add*: *cnj-x-x*)

lemma *butterfly-scaleR-left*[*simp*]: *butterfly* (*r* $*_R$ ψ) φ = *r* $*_C$ *butterfly* ψ φ
by (*simp add*: *scaleR-scaleC*)

lemma *butterfly-scaleR-right*[*simp*]: *butterfly* ψ (*r* $*_R$ φ) = *r* $*_C$ *butterfly* ψ φ
by (*simp add*: *butterfly-scaleC-right scaleR-scaleC*)

lemma *butterfly-adjoint*[*simp*]: (*butterfly* ψ φ)* = *butterfly* φ ψ
unfolding *butterfly-def* **by** *auto*

lemma *butterfly-comp-butterfly*[*simp*]: *butterfly* $\psi 1$ $\psi 2$ o_{CL} *butterfly* $\psi 3$ $\psi 4$ = ($\psi 2$

$\cdot_C$ $\psi 3$) $*_C$ *butterfly* $\psi 1$ $\psi 4$
by (*simp add*: *butterfly-comp-cblinfun*)

lemma *butterfly-0-left*[*simp*]: *butterfly* 0 *a* = 0
by (*simp add*: *butterfly-def*)

lemma *butterfly-0-right*[*simp*]: *butterfly a* 0 = 0
by (*simp add*: *butterfly-def*)

lemma *butterfly-is-rank1*:
assumes ‹$\varphi \neq 0$›
shows ‹*butterfly* ψ φ $*_S$ $\top$ = *ccspan* {ψ}›
using *assms* **by** (*simp add*: *butterfly-def cblinfun-compose-image image-vector-to-cblinfun-adj'*)

lemma *rank1-is-butterfly*:
— The restriction ψ is necessary. Consider, e.g., the space of all finite sequences (with sum-norm), and $A' f = (\sum x.\ f\, x)$. Then A' is not a butterfly.
assumes ‹A $*_S$ $\top$ = *ccspan* {ψ::-::*chilbert-space*}›
shows ‹$\exists \varphi.\ A$ = *butterfly* ψ φ›
proof (*rule exI*[*of* - ‹$A*$ $*_V$ (ψ /$_R$ (*norm* ψ)^2)›], *rule cblinfun-eqI*)
fix γ :: 'b
from *assms* **have** ‹A $*_V$ γ ∈ *space-as-set* (*ccspan* {ψ})›
by (*simp flip*: *assms*)
then obtain *c* **where** *c*: ‹A $*_V$ γ = *c* $*_C$ ψ›
apply *atomize-elim*
apply (*auto simp*: *ccspan.rep-eq*)
by (*metis complex-vector.span-breakdown-eq complex-vector.span-empty eq-iff-diff-eq-0 singletonD*)
have ‹A $*_V$ γ = *butterfly* ψ (ψ /$_R$ (*norm* ψ)^2) $*_V$ (A $*_V$ γ)›
apply (*auto simp*: *c simp flip*: *scaleC-scaleC*)
by (*metis cinner-eq-zero-iff divideC-field-simps*(1) *power2-norm-eq-cinner scaleC-left-commute scaleC-zero-right*)
also have ‹... = (*butterfly* ψ (ψ /$_R$ (*norm* ψ)^2) o_{CL} A) $*_V$ γ›
by *simp*
also have ‹... = *butterfly* ψ ($A*$ $*_V$ (ψ /$_R$ (*norm* ψ)^2)) $*_V$ γ›
by (*simp add*: *cinner-adj-left*)
finally show ‹A $*_V$ γ = ...›
by −
qed

lemma *rank1-0*[*simp*]: ‹*rank1* 0›
by (*metis ccspan-0 kernel-0 kernel-apply-self rank1-def*)

lemma *rank1-iff-butterfly*: ‹*rank1* A ⟷ ($\exists \psi\ \varphi.\ A$ = *butterfly* ψ φ)›
for A :: ‹-::*complex-inner* $\Rightarrow_{CL}$ -::*chilbert-space*›
proof (*rule iffI*)
assume ‹*rank1* A›
then obtain ψ **where** ‹A $*_S$ $\top$ = *ccspan* {ψ}›
using *rank1-def* **by** *auto*

```
  then have ‹∃φ. A = butterfly ψ φ›
    by (rule rank1-is-butterfly)
  then show ‹∃ψ φ. A = butterfly ψ φ›
    by auto
next
  assume asm: ‹∃ψ φ. A = butterfly ψ φ›
  show ‹rank1 A›
  proof (cases ‹A = 0›)
    case True
    then show ?thesis
      by simp
  next
    case False
    from asm obtain ψ φ where A: ‹A = butterfly ψ φ›
      by auto
    with False have ‹ψ ≠ 0› and ‹φ ≠ 0›
      by auto
    then have ‹butterfly ψ φ *_S ⊤ = ccspan {ψ}›
      by (rule-tac butterfly-is-rank1)
    with A ‹ψ ≠ 0› show ‹rank1 A›
      by (auto intro!: exI[of - ψ] simp: rank1-def)
  qed
qed

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

lemma norm-butterfly: norm (butterfly ψ φ) = norm ψ * norm φ
proof (cases φ=0)
  case True
  then show ?thesis by simp
next
  case False
  show ?thesis
    unfolding norm-cblinfun.rep-eq
  proof (rule onormI[OF - False])
    fix x

    have cmod (φ •_C x) * norm ψ ≤ norm ψ * norm φ * norm x
     by (metis ab-semigroup-mult-class.mult-ac(1) complex-inner-class.Cauchy-Schwarz-ineq2
mult.commute mult-left-mono norm-ge-zero)
    thus norm (butterfly ψ φ *_V x) ≤ norm ψ * norm φ * norm x
      by (simp add: power2-eq-square)

    show norm (butterfly ψ φ *_V φ) = norm ψ * norm φ * norm φ
     by (smt (z3) ab-semigroup-mult-class.mult-ac(1) butterfly-apply mult.commute
norm-eq-sqrt-cinner norm-ge-zero norm-scaleC power2-eq-square real-sqrt-abs real-sqrt-eq-iff)
  qed
```

qed

lemma *bounded-sesquilinear-butterfly*[*bounded-sesquilinear*]: ‹*bounded-sesquilinear* (*λ*(*b*::′*b*::*chilbert-space*) (*a*::′*a*::*chilbert-space*). *butterfly a b*)›
proof *standard*
fix *a a*′ :: ′*a* **and** *b b*′ :: ′*b* **and** *r* :: *complex*
show ‹*butterfly* (*a* + *a*′) *b* = *butterfly a b* + *butterfly a*′ *b*›
by (*rule butterfly-add-left*)
show ‹*butterfly a* (*b* + *b*′) = *butterfly a b* + *butterfly a b*′›
by (*rule butterfly-add-right*)
show ‹*butterfly* (*r* ∗*C a*) *b* = *r* ∗*C butterfly a b*›
by *simp*
show ‹*butterfly a* (*r* ∗*C b*) = *cnj r* ∗*C butterfly a b*›
by *simp*
show ‹∃ *K*. ∀ *b a*. *norm* (*butterfly a b*) ≤ *norm b* ∗ *norm a* ∗ *K* ›
apply (*rule exI*[*of* - *1*])
by (*simp add*: *norm-butterfly*)
qed

lemma *inj-selfbutter-upto-phase*:
assumes *selfbutter x* = *selfbutter y*
shows ∃ *c*. *cmod c* = *1* ∧ *x* = *c* ∗*C y*
proof (*cases x* = *0*)
case *True*
from *assms* **have** *y* = *0*
using *norm-butterfly*
by (*metis True butterfly-0-left divisors-zero norm-eq-zero*)
with *True* **show** *?thesis*
using *norm-one* **by** *fastforce*
next
case *False*
define *c* **where** *c* = (*y* •*C x*) / (*x* •*C x*)
have (*x* •*C x*) ∗*C x* = *selfbutter x* ∗*V x*
by (*simp add*: *butterfly-apply*)
also have ... = *selfbutter y* ∗*V x*
using *assms* **by** *simp*
also have ... = (*y* •*C x*) ∗*C y*
by (*simp add*: *butterfly-apply*)
finally have *xcy*: *x* = *c* ∗*C y*
by (*simp add*: *c-def ceq-vector-fraction-iff*)
have *cmod c* ∗ *norm x* = *cmod c* ∗ *norm y*
using *assms norm-butterfly*
by (*smt* (*verit*, *ccfv-SIG*) ‹(*x* •*C x*) ∗*C x* = *selfbutter x* ∗*V x*› ‹*selfbutter y* ∗*V x* = (*y* •*C x*) ∗*C y*› *cinner-scaleC-right complex-vector.scale-left-commute complex-vector.scale-right-imp-eq mult-cancel-left norm-eq-sqrt-cinner norm-eq-zero scaleC-scaleC xcy*)
also have *cmod c* ∗ *norm y* = *norm* (*c* ∗*C y*)
by *simp*
also have ... = *norm x*

```
    unfolding xcy[symmetric] by simp
  finally have c: cmod c = 1
    by (simp add: False)
  from c xcy show ?thesis
    by auto
qed

lemma butterfly-eq-proj:
  assumes norm x = 1
  shows selfbutter x = proj x
proof −
  define B and φ :: complex ⇒CL 'a
    where B = selfbutter x and φ = vector-to-cblinfun x
  then have B: B = φ oCL φ*
    unfolding butterfly-def by simp
  have φadjφ: φ* oCL φ = id-cblinfun
    using φ-def assms isometry-def isometry-vector-to-cblinfun by blast
  have B oCL B = φ oCL (φ* oCL φ) oCL φ*
    by (simp add: B cblinfun-assoc-left(1))
  also have ... = B
    unfolding φadjφ by (simp add: B)
  finally have idem: B oCL B = B.
  have herm: B = B*
    unfolding B by simp
  from idem herm have BProj: B = Proj (B *S top)
    by (rule Proj-on-own-range'[symmetric])
  have B *S top = ccspan {x}
    by (simp add: B φ-def assms cblinfun-compose-image range-adjoint-isometry)
  with BProj show B = proj x
    by simp
qed

lemma butterfly-sgn-eq-proj:
  shows selfbutter (sgn x) = proj x
proof (cases ‹x = 0›)
  case True
  then show ?thesis
    by simp
next
  case False
  then have ‹selfbutter (sgn x) = proj (sgn x)›
    by (simp add: butterfly-eq-proj norm-sgn)
  also have ‹ccspan {sgn x} = ccspan {x}›
    by (metis ccspan-singleton-scaleC scaleC-eq-0-iff scaleR-scaleC sgn-div-norm
sgn-zero-iff)
  finally show ?thesis
    by −
qed
```

lemma *butterfly-is-Proj*:
‹*norm x = 1 ⟹ is-Proj* (*selfbutter x*)›
by (*subst butterfly-eq-proj*, *simp-all*)

lemma *cspan-butterfly-UNIV*:
assumes ‹*cspan basisA = UNIV*›
assumes ‹*cspan basisB = UNIV*›
assumes ‹*is-ortho-set basisB*›
assumes ‹⋀*b*. *b* ∈ *basisB* ⟹ *norm b = 1*›
shows ‹*cspan* {*butterfly a b*| (*a*::′*a*::{*complex-normed-vector*}) (*b*::′*b*::{*chilbert-space*,*cfinite-dim*}).
a ∈ *basisA* ∧ *b* ∈ *basisB*} = *UNIV*›
proof −
have *F*: ‹∃ *F*∈{*butterfly a b* |*a b*. *a* ∈ *basisA* ∧ *b* ∈ *basisB*}. ∀ *b*′∈*basisB*. *F* ∗$_V$
b′ = (*if b*′ = *b then a else* 0)›
if ‹*a* ∈ *basisA*› **and** ‹*b* ∈ *basisB*› **for** *a b*
apply (*rule bexI*[**where** *x*=‹*butterfly a b*›])
using *assms that* **by** (*auto simp*: *is-ortho-set-def cnorm-eq-1*)
show *?thesis*
apply (*rule cblinfun-cspan-UNIV*[**where** *basisA*=*basisB* **and** *basisB*=*basisA*])
using *assms* **apply** *auto*[2]
using *F* **by** (*smt* (*verit*, *ccfv-SIG*) *image-iff*)
qed

lemma *cindependent-butterfly*:
fixes *basisA* :: ‹′*a*::*chilbert-space set*› **and** *basisB* :: ‹′*b*::*chilbert-space set*›
assumes ‹*is-ortho-set basisA*› ‹*is-ortho-set basisB*›
assumes *normA*: ‹⋀*a*. *a*∈*basisA* ⟹ *norm a = 1*› **and** *normB*: ‹⋀*b*. *b*∈*basisB*
⟹ *norm b = 1*›
shows ‹*cindependent* {*butterfly a b*| *a b*. *a*∈*basisA* ∧ *b*∈*basisB*}›
proof (*unfold complex-vector.independent-explicit-module*, *intro allI impI*, *rename-tac*
T f g)
fix *T* :: ‹(′*b* ⇒$_{CL}$ ′*a*) *set*› **and** *f* :: ‹′*b* ⇒$_{CL}$ ′*a* ⇒ *complex*› **and** *g* :: ‹′*b* ⇒$_{CL}$ ′*a*›
assume ‹*finite T*›
assume *T-subset*: ‹*T* ⊆ {*butterfly a b* |*a b*. *a* ∈ *basisA* ∧ *b* ∈ *basisB*}›
define *lin* **where** ‹*lin* = (∑ *g*∈*T*. *f g* ∗$_C$ *g*)›
assume ‹*lin = 0*›
assume ‹*g* ∈ *T*›

then obtain *a b* **where** *g*: ‹*g = butterfly a b*› **and** [*simp*]: ‹*a* ∈ *basisA*› ‹*b* ∈
basisB›
using *T-subset* **by** *auto*

have ∗: (*vector-to-cblinfun a*)∗ ∗$_V$ *f g* ∗$_C$ *g* ∗$_V$ *b = 0*
if ‹*g* ∈ *T* − {*butterfly a b*}› **for** *g*
proof −
from *that*
obtain *a*′ *b*′ **where** *g*: ‹*g = butterfly a*′ *b*′› **and** [*simp*]: ‹*a*′ ∈ *basisA*› ‹*b*′ ∈
basisB›
using *T-subset* **by** *auto*

```
    from that have ‹g ≠ butterfly a b› by auto
    with g consider (a) ‹a≠a'› | (b) ‹b≠b'›
      by auto
    then show ‹(vector-to-cblinfun a)* *V f g *C g *V b = 0›
    proof cases
      case a
      then show ?thesis
        using ‹is-ortho-set basisA› unfolding g
        by (auto simp: is-ortho-set-def butterfly-def scaleC-cblinfun.rep-eq)
    next
      case b
      then show ?thesis
        using ‹is-ortho-set basisB› unfolding g
        by (auto simp: is-ortho-set-def butterfly-def scaleC-cblinfun.rep-eq)
    qed
  qed

  have ‹0 = (vector-to-cblinfun a)* *V lin *V b›
    using ‹lin = 0› by auto
  also have ‹... = (∑ g∈T. (vector-to-cblinfun a)* *V (f g *C g) *V b)›
    unfolding lin-def
    apply (rule complex-vector.linear-sum)
   by (smt (z3) cblinfun.scaleC-left cblinfun.scaleC-right cblinfun.add-right clinearI
plus-cblinfun.rep-eq)
  also have ‹... = (∑ g∈{butterfly a b}. (vector-to-cblinfun a)* *V (f g *C g) *V
b)›
    apply (rule sum.mono-neutral-right)
    using ‹finite T› * ‹g ∈ T› g by auto
  also have ‹... = (vector-to-cblinfun a)* *V (f g *C g) *V b›
    by (simp add: g)
  also have ‹... = f g›
    unfolding g
   using normA normB by (auto simp: butterfly-def scaleC-cblinfun.rep-eq cnorm-eq-1)
  finally show ‹f g = 0›
    by simp
qed

lemma clinear-eq-butterflyI:
  fixes F G :: ‹('a::{chilbert-space,cfinite-dim} ⇒CL 'b::complex-inner) ⇒ 'c::complex-vector›
  assumes clinear F and clinear G
  assumes ‹cspan basisA = UNIV› ‹cspan basisB = UNIV›
  assumes ‹is-ortho-set basisA› ‹is-ortho-set basisB›
  assumes ⋀a b. a∈basisA ⟹ b∈basisB ⟹ F (butterfly a b) = G (butterfly a b)
  assumes ‹⋀b. b∈basisB ⟹ norm b = 1›
  shows F = G
  apply (rule complex-vector.linear-eq-on-span[where f=F, THEN ext, rotated 3])
     apply (subst cspan-butterfly-UNIV)
  using assms by auto
```

lemma *sum-butterfly-is-Proj*:
assumes ‹*is-ortho-set E*›
assumes ‹$\bigwedge e.\ e{\in}E \Longrightarrow norm\ e = 1$›
shows ‹*is-Proj* $(\sum e{\in}E.\ butterfly\ e\ e)$›
proof (*cases* ‹*finite E*›)
case *True*
show *?thesis*
proof (*rule is-Proj-I*)
show ‹$(\sum e{\in}E.\ butterfly\ e\ e)* = (\sum e{\in}E.\ butterfly\ e\ e)$›
by (*simp add*: *sum-adj*)
have *ortho*: ‹$f \neq e \Longrightarrow e \in E \Longrightarrow f \in E \Longrightarrow$ *is-orthogonal f e*› **for** *f e*
by (*meson assms*(*1*) *is-ortho-set-def*)
have *unit*: ‹$e \cdot_C e = 1$› **if** ‹$e \in E$› **for** *e*
using *assms*(*2*) *cnorm-eq-1 that* **by** *blast*
have $*$: ‹$(\sum f{\in}E.\ (f \cdot_C e) *_C butterfly\ f\ e) = butterfly\ e\ e$› **if** ‹$e \in E$› **for** *e*
apply (*subst sum-single*[**where** *i*=*e*])
by (*auto intro*!: *simp*: *that ortho unit True*)
show ‹$(\sum e{\in}E.\ butterfly\ e\ e)\ o_{CL}\ (\sum e{\in}E.\ butterfly\ e\ e) = (\sum e{\in}E.\ butterfly\ e\ e)$›
by (*auto simp*: $*$ *cblinfun-compose-sum-right cblinfun-compose-sum-left*)
qed
next
case *False*
then show *?thesis*
by *simp*
qed

lemma *rank1-compose-left*: ‹*rank1* $(a\ o_{CL}\ b)$› **if** ‹*rank1 b*›
proof −
from ‹*rank1 b*›
obtain ψ **where** ‹$b *_S \top = ccspan\ \{\psi\}$›
using *rank1-def* **by** *blast*
then have $*$: ‹$(a\ o_{CL}\ b) *_S \top = ccspan\ \{a\ \psi\}$›
by (*metis cblinfun-assoc-left*(*2*) *cblinfun-image-ccspan image-empty image-insert*)
then show ‹*rank1* $(a\ o_{CL}\ b)$›
using *rank1-def* **by** *blast*
qed

lemma *csubspace-of-1dim-space*:
assumes ‹$S \neq \{0\}$›
assumes ‹*csubspace S*›
assumes ‹$S \subseteq cspan\ \{\psi\}$›
shows ‹$S = cspan\ \{\psi\}$›
proof −
from ‹$S \neq \{0\}$› ‹*csubspace S*›
obtain φ **where** ‹$\varphi \in S$› **and** ‹$\varphi \neq 0$›
using *complex-vector.subspace-0* **by** *blast*
then have ‹$\varphi \in cspan\ \{\psi\}$›
using ‹$S \subseteq cspan\ \{\psi\}$› **by** *blast*

with ‹$\varphi \neq 0$› **obtain** c **where** ‹$\varphi = c *_C \psi$› **and** ‹$c \neq 0$›
by (*metis complex-vector.span-breakdown-eq complex-vector.span-empty right-minus-eq scaleC-eq-0-iff singletonD*)
have ‹*cspan* $\{\psi\}$ = *cspan* $\{inverse\ c *_C \varphi\}$›
by (*simp add*: ‹$\varphi = c *_C \psi$› ‹$c \neq 0$›)
also have ‹... $\subseteq$ *cspan* $\{\varphi\}$›
using ‹$c \neq 0$› **by** *auto*
also from ‹$\varphi = c *_C \psi$› ‹$\varphi \in S$› ‹$c \neq 0$› *assms*
have ‹... $\subseteq S$›
by (*metis complex-vector.span-subspace cspan-singleton-scaleC empty-subsetI insert-Diff insert-mono*)
finally have ‹*cspan* $\{\psi\} \subseteq S$›
by −
with ‹$S \subseteq$ *cspan* $\{\psi\}$› **show** *?thesis*
by *simp*
qed

lemma *subspace-of-1dim-ccspan*:
assumes ‹$S \neq 0$›
assumes ‹$S \leq$ *ccspan* $\{\psi\}$›
shows ‹$S =$ *ccspan* $\{\psi\}$›
using *assms* **apply** *transfer*
by (*simp add*: *csubspace-of-1dim-space*)

lemma *rank1-compose-right*: ‹*rank1* $(a\ o_{CL}\ b)$› **if** ‹*rank1* a›
proof −
have ‹$(a\ o_{CL}\ b) *_S \top \leq a *_S \top$›
by (*metis cblinfun-apply-cblinfun-compose cblinfun-apply-in-image cblinfun-image-ccspan-leqI ccspan-UNIV*)
also from ‹*rank1* a›
obtain ψ **where** ‹$a *_S \top =$ *ccspan* $\{\psi\}$›
using *rank1-def* **by** *blast*
finally have $*$: ‹$(a\ o_{CL}\ b) *_S \top \leq$ *ccspan* $\{\psi\}$›
by −
show ‹*rank1* $(a\ o_{CL}\ b)$›
proof (*cases* ‹$(a\ o_{CL}\ b) *_S \top = 0$›)
case *True*
then show *?thesis*
by *simp*
next
case *False*
with $*$ **have** ‹$(a\ o_{CL}\ b) *_S \top =$ *ccspan* $\{\psi\}$›
using *subspace-of-1dim-ccspan* **by** *blast*
then show *?thesis*
using *rank1-def* **by** *blast*
qed
qed

lemma *rank1-scaleC*: ‹*rank1* $(c *_C a)$› **if** ‹*rank1 a*› **and** ‹$c \neq 0$›
using *rank1-compose-left*[*OF* ‹*rank1 a*›, **where** *a*=‹$c *_C$ *id-cblinfun*›]
using *that* **by** *force*

lemma *rank1-scaleR*[*simp*]: ‹*rank1* $(c *_R a)$› **if** ‹*rank1 a*› **and** ‹$c \neq 0$›
by (*simp add*: *rank1-scaleC scaleR-scaleC that*(*1*) *that*(*2*))

lemma *rank1-uminus*: ‹*rank1* $(-a)$› **if** ‹*rank1 a*›
using *that rank1-scaleC*[**where** *c*=‹-1› **and** *a*=*a*] **by** *simp*

lemma *rank1-uminus-iff*[*simp*]: ‹*rank1* $(-a) \longleftrightarrow$ *rank1 a*›
using *rank1-uminus* **by** *force*

lemma *rank1-adj*: ‹*rank1* $(a*)$› **if** ‹*rank1 a*›
for *a* :: ‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*chilbert-space*›
by (*metis adj-0 butterfly-adjoint rank1-iff-butterfly that*)

lemma *rank1-adj-iff*[*simp*]: ‹*rank1* $(a*) \longleftrightarrow$ *rank1 a*›
for *a* :: ‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*chilbert-space*›
by (*metis double-adj rank1-adj*)

lemma *butterflies-sum-id-finite*: ‹*id-cblinfun* $= (\sum x{\in}B.$ *selfbutter x*)› **if** ‹*is-onb B*› **for** *B* :: ‹$'a$ :: {*cfinite-dim*, *chilbert-space*} *set*›
proof (*rule cblinfun-eq-gen-eqI*)
from ‹*is-onb B*› **show** ‹*ccspan B* $= \top$›
by (*simp add*: *is-onb-def*)
from ‹*is-onb B*› **have** ‹*cindependent B*›
by (*simp add*: *is-onb-def is-ortho-set-cindependent*)
then have [*simp*]: ‹*finite B*›
using *cindependent-cfinite-dim-finite* **by** *blast*
from ‹*is-onb B*›
have *1*: ‹$j \neq b \Longrightarrow j \in B \Longrightarrow$ *is-orthogonal j b*› **if** ‹$b \in B$› **for** *j b*
using *that* **by** (*auto simp add*: *is-onb-def is-ortho-set-def*)
from ‹*is-onb B*›
have *2*: ‹$b \bullet_C b = 1$› **if** ‹$b \in B$› **for** *b*
using *that* **by** (*simp add*: *is-onb-def cnorm-eq-1*)
fix *b* **assume** ‹$b \in B$›
then show ‹*id-cblinfun* $*_V$ *b* $= (\sum x{\in}B.$ *selfbutter x*) $*_V$ *b*›
using *1 2* **by** (*simp add*: *cblinfun.sum-left sum-single*[**where** *i*=*b*])
qed

lemma *butterfly-sum-left*: ‹*butterfly* $(\sum i{\in}M.\ \psi\ i)\ \varphi = (\sum i{\in}M.$ *butterfly* $(\psi\ i)$ $\varphi)$›
apply (*induction M rule*:*infinite-finite-induct*)
by (*auto simp add*: *butterfly-add-left*)

lemma *butterfly-sum-right*: ‹*butterfly* ψ $(\sum i{\in}M.\ \varphi\ i) = (\sum i{\in}M.$ *butterfly* ψ $(\varphi$ $i))$›
apply (*induction M rule*:*infinite-finite-induct*)

by (*auto simp add*: *butterfly-add-right*)

lemma *sum-butterfly-leq-id*:
assumes ‹*is-ortho-set E*›
assumes ‹$\bigwedge e.\ e{\in}E \Longrightarrow$ *norm e* = *1*›
shows ‹$(\sum i{\in}E.$ *butterfly i i*$) \leq$ *id-cblinfun*›
proof −
have ‹*is-Proj* $(\sum \psi{\in}E.$ *butterfly* ψ $\psi)$›
using *assms* **by** (*rule sum-butterfly-is-Proj*)
then show *?thesis*
by (*auto intro*!: *is-Proj-leq-id*)
qed

lemma *sandwich-butterfly*: ‹*sandwich a* (*butterfly x y*) = *butterfly* (*a x*) (*a y*)›
by (*simp add*: *sandwich-apply butterfly-comp-cblinfun cblinfun-comp-butterfly*)

13.18 Banach-Steinhaus

theorem *cbanach-steinhaus*:
fixes *F* :: ‹*'c* ⇒ *'a*::*cbanach* ⇒$_{CL}$ *'b*::*complex-normed-vector*›
assumes ‹$\bigwedge x.\ \exists M.\ \forall n.$ *norm* ((*F n*) $*_V$ *x*) ≤ *M*›
shows ‹$\exists M.\ \forall$ *n*. *norm* (*F n*) ≤ *M*›
using *cblinfun-blinfun-transfer*[*transfer-rule*]
apply *fail?*
proof (*use assms* **in** *transfer*)
fix *F* :: ‹*'c* ⇒ *'a* ⇒$_L$ *'b*› **assume** ‹$(\bigwedge x.\ \exists M.\ \forall n.$ *norm* (*F n* $*_v$ *x*) ≤ *M*)›
hence ‹$\bigwedge x.$ *bounded* (*range* ($\lambda n.$ *blinfun-apply* (*F n*) *x*))›
by (*metis* (*no-types*, *lifting*) *boundedI rangeE*)
hence ‹*bounded* (*range F*)›
by (*simp add*: *banach-steinhaus*)
thus ‹$\exists M.\ \forall n.$ *norm* (*F n*) ≤ *M*›
by (*simp add*: *bounded-iff*)
qed

13.19 Riesz-representation theorem

theorem *riesz-representation-cblinfun-existence*:
— Theorem 3.4 in [1]
fixes *f*::‹*'a*::*chilbert-space* ⇒$_{CL}$ *complex*›
shows ‹$\exists t.\ \forall x.$ *f* $*_V$ *x* = (*t* $\cdot_C$ *x*)›
by *transfer* (*rule riesz-representation-existence*)

lemma *riesz-representation-cblinfun-unique*:
— Theorem 3.4 in [1]
fixes *f*::‹*'a*::*complex-inner* ⇒$_{CL}$ *complex*›
assumes ‹$\bigwedge x.$ *f* $*_V$ *x* = (*t* $\cdot_C$ *x*)›
assumes ‹$\bigwedge x.$ *f* $*_V$ *x* = (*u* $\cdot_C$ *x*)›
shows ‹*t* = *u*›
using *assms* **by** (*rule riesz-representation-unique*)

theorem *riesz-representation-cblinfun-norm*:
includes *norm-syntax*
fixes f::‹$'a$::*chilbert-space* ⇒$_{CL}$ *complex*›
assumes ‹$\bigwedge x.\ f *_V x = (t \cdot_C x)$›
shows ‹$\|f\| = \|t\|$›
using *assms*
proof *transfer*
fix f::‹$'a$ ⇒ *complex*› **and** t
assume ‹*bounded-clinear f*› **and** ‹$\bigwedge x.\ f\ x = (t \cdot_C x)$›
from ‹$\bigwedge x.\ f\ x = (t \cdot_C x)$›
have ‹(*norm* (*f x*)) / (*norm x*) ≤ *norm t*›
for x
proof(*cases* ‹*norm x* = *0*›)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
have ‹*norm* (*f x*) = *norm* ((*t* ·$_C$ *x*))›
using ‹$\bigwedge x.\ f\ x = (t \cdot_C x)$› **by** *simp*
also have ‹*norm* (*t* ·$_C$ *x*) ≤ *norm t* * *norm x*›
by (*simp add*: *complex-inner-class.Cauchy-Schwarz-ineq2*)
finally have ‹*norm* (*f x*) ≤ *norm t* * *norm x*›
by *blast*
thus *?thesis*
by (*metis False linordered-field-class.divide-right-mono nonzero-mult-div-cancel-right norm-ge-zero*)
qed
moreover have ‹(*norm* (*f t*)) / (*norm t*) = *norm t*›
proof(*cases* ‹*norm t* = *0*›)
case *True*
thus *?thesis*
by *simp*
next
case *False*
have ‹*f t* = (*t* ·$_C$ *t*)›
using ‹$\bigwedge x.\ f\ x = (t \cdot_C x)$› **by** *blast*
also have ‹... = (*norm t*)^*2*›
by (*meson cnorm-eq-square*)
also have ‹... = (*norm t*)*(*norm t*)›
by (*simp add*: *power2-eq-square*)
finally have ‹*f t* = (*norm t*)*(*norm t*)›
by *blast*
thus *?thesis*
by (*metis* ‹*f t* = *t* ·$_C$ *t*› *norm-eq-sqrt-cinner norm-ge-zero real-div-sqrt*)
qed
ultimately have ‹*Sup* {(*norm* (*f x*)) / (*norm x*)| *x*. *True*} = *norm t*›
by (*smt cSup-eq-maximum mem-Collect-eq*)
moreover have ‹*Sup* {(*norm* (*f x*)) / (*norm x*)| *x*. *True*} = (*SUP x*. (*norm* (*f x*)) / (*norm x*))›

by (*simp add*: *full-SetCompr-eq*)
ultimately show ‹*onorm f* = *norm t*›
by (*simp add*: *onorm-def*)
qed

definition *the-riesz-rep* :: ‹('*a*::*chilbert-space* ⇒$_{CL}$ *complex*) ⇒ '*a*› **where**
‹*the-riesz-rep f* = (*SOME t*. ∀ *x*. *f* *$_V$ *x* = *t* ·$_C$ *x*)›

lemma *the-riesz-rep*[*simp*]: ‹*the-riesz-rep f* ·$_C$ *x* = *f* *$_V$ *x*›
unfolding *the-riesz-rep-def*
apply (*rule someI2-ex*)
by (*simp-all add*: *riesz-representation-cblinfun-existence*)

lemma *the-riesz-rep-unique*:
assumes ‹⋀*x*. *f* *$_V$ *x* = *t* ·$_C$ *x*›
shows ‹*t* = *the-riesz-rep f*›
using *assms riesz-representation-cblinfun-unique the-riesz-rep* **by** *metis*

lemma *the-riesz-rep-scaleC*: ‹*the-riesz-rep* (*c* *$_C$ *f*) = *cnj c* *$_C$ *the-riesz-rep f*›
apply (*rule the-riesz-rep-unique*[*symmetric*])
by *auto*

lemma *the-riesz-rep-add*: ‹*the-riesz-rep* (*f* + *g*) = *the-riesz-rep f* + *the-riesz-rep g*›
apply (*rule the-riesz-rep-unique*[*symmetric*])
by (*auto simp*: *cinner-add-left cblinfun.add-left*)

lemma *the-riesz-rep-norm*[*simp*]: ‹*norm* (*the-riesz-rep f*) = *norm f*›
apply (*rule riesz-representation-cblinfun-norm*[*symmetric*])
by *simp*

lemma *bounded-antilinear-the-riesz-rep*[*bounded-antilinear*]: ‹*bounded-antilinear the-riesz-rep*›
by (*metis* (*no-types*, *opaque-lifting*) *bounded-antilinear-intro dual-order.refl mult.commute mult-1 the-riesz-rep-add the-riesz-rep-norm the-riesz-rep-scaleC*)

lift-definition *the-riesz-rep-sesqui* :: ‹('*a*::*complex-normed-vector* ⇒ '*b*::*chilbert-space* ⇒ *complex*) ⇒ ('*a* ⇒$_{CL}$ '*b*)› **is**
‹λ*p*. *if bounded-sesquilinear p then the-riesz-rep o CBlinfun o p else* 0›
by (*metis* (*mono-tags*, *lifting*) *Cblinfun-comp-bounded-sesquilinear bounded-antilinear-o-bounded-antilinear' bounded-antilinear-the-riesz-rep bounded-clinear-0 fun.map-comp*)

lemma *the-riesz-rep-sesqui-apply*:
assumes ‹*bounded-sesquilinear p*›
shows ‹(*the-riesz-rep-sesqui p* *$_V$ *x*) ·$_C$ *y* = *p x y*›
apply (*transfer fixing*: *p x y*)
by (*auto simp add*: *CBlinfun-inverse bounded-sesquilinear-apply-bounded-clinear assms*)

13.20 Bidual

lift-definition *bidual-embedding* :: ‹'a::*complex-normed-vector* ⇒$_{CL}$ (('a ⇒$_{CL}$ *complex*) ⇒$_{CL}$ *complex*)›
  is ‹λx f. f *$_V$ x›
  by (*simp add*: *cblinfun.flip*)

lemma *bidual-embedding-apply*[*simp*]: ‹(*bidual-embedding* *$_V$ x) *$_V$ f = f *$_V$ x›
  by (*transfer fixing*: *x f*, *simp*)

lemma *bidual-embedding-isometric*[*simp*]: ‹*norm* (*bidual-embedding* *$_V$ x) = *norm* x› **for** *x* :: ‹'a::*complex-inner*›
proof −
  define *f* :: ‹'a ⇒$_{CL}$ *complex*› **where** ‹*f* = *CBlinfun* (λy. *cinner x y*)›
  then have [*simp*]: ‹*f* *$_V$ y = *cinner x y*› **for** *y*
    by (*simp add*: *bounded-clinear-CBlinfun-apply bounded-clinear-cinner-right*)
  then have [*simp*]: ‹*norm f* = *norm x*›
   apply (*auto intro*!: *Cauchy-Schwarz-ineq2 norm-cblinfun-eqI*[**where** *x*=*x*] *simp*: *power2-norm-eq-cinner*[*symmetric*])
    by (*simp add*: *norm-mult power2-eq-square*)
  show *?thesis*
    apply (*auto intro*!: *norm-cblinfun-eqI*[**where** *x*=*f*])
     apply (*metis norm-eq-sqrt-cinner norm-imp-pos-and-ge real-div-sqrt*)
    by (*metis mult.commute norm-cblinfun*)
qed

lemma *norm-bidual-embedding*[*simp*]: ‹*norm* (*bidual-embedding* :: 'a::{*complex-inner*, *not-singleton*} ⇒$_{CL}$ -) = *1*›
proof −
  obtain *x* :: 'a **where** [*simp*]: ‹*norm x* = *1*›
    by (*meson UNIV-not-singleton ex-norm1*)
  show *?thesis*
    by (*auto intro*!: *norm-cblinfun-eqI*[**where** *x*=*x*])
qed

lemma *isometry-bidual-embedding*[*simp*]: ‹*isometry bidual-embedding*›
  by (*simp add*: *norm-preserving-isometry*)

lemma *bidual-embedding-surj*[*simp*]: ‹*surj* (*bidual-embedding* :: 'a::*chilbert-space* ⇒$_{CL}$ -)›
proof −
  have ‹∃ y''. ∀ f. (*bidual-embedding* *$_V$ y'') *$_V$ f = y *$_V$ f›
    for *y* :: ‹('a ⇒$_{CL}$ *complex*) ⇒$_{CL}$ *complex*›
  proof −
    define *y'* **where** ‹*y'* = *CBlinfun* (λz. *cnj* (*y* (*cblinfun-cinner-right z*)))›
    have *y'-apply*: ‹*y' z* = *cnj* (*y* (*cblinfun-cinner-right z*))› **for** *z*
      unfolding *y'-def*
      apply (*subst CBlinfun-inverse*)
      by (*auto intro*!: *bounded-linear-intros*)
    obtain *y''* **where** ‹*y' z* = *y''* ·$_C$ *z*› **for** *z*

using *riesz-representation-cblinfun-existence* **by** *blast*
then have y'': ‹$z \cdot_C y'' = cnj\ (y'\ z)$› **for** z
by *auto*
have ‹(*bidual-embedding* $*_V\ y''$) $*_V\ f = y *_V f$› **for** $f :: \langle 'a \Rightarrow_{CL} complex\rangle$
proof −
obtain f' **where** f': ‹$f\ z = f' \cdot_C z$› **for** z
using *riesz-representation-cblinfun-existence* **by** *blast*
then have $f'2$: ‹$f = cblinfun\text{-}cinner\text{-}right\ f'$›
using *cblinfun-apply-inject* **by** *force*
show *?thesis*
by (*auto simp add*: $f'\ f'2\ y''\ y'$-*apply*)
qed
then show *?thesis*
by *blast*
qed
then show *?thesis*
by (*metis cblinfun-eqI surj-def*)
qed

13.21 Extension of complex bounded operators

definition *cblinfun-extension* **where**
cblinfun-extension $S\ \varphi = (SOME\ B.\ \forall x{\in}S.\ B *_V x = \varphi\ x)$

definition *cblinfun-extension-exists* **where**
cblinfun-extension-exists $S\ \varphi = (\exists B.\ \forall x{\in}S.\ B *_V x = \varphi\ x)$

lemma *cblinfun-extension-existsI*:
assumes $\bigwedge x.\ x{\in}S \Longrightarrow B *_V x = \varphi\ x$
shows *cblinfun-extension-exists* $S\ \varphi$
using *assms cblinfun-extension-exists-def* **by** *blast*

lemma *cblinfun-extension-exists-finite-dim*:
fixes φ::$'a$::{*complex-normed-vector*,*cfinite-dim*} ⇒ $'b$::*complex-normed-vector*
assumes *cindependent* S
and *cspan* $S = UNIV$
shows *cblinfun-extension-exists* $S\ \varphi$
proof−
define f::$'a$ ⇒ $'b$
where f = *complex-vector.construct* $S\ \varphi$
have *clinear* f
by (*simp add*: *complex-vector.linear-construct assms linear-construct f-def*)
have *bounded-clinear* f
using ‹*clinear* f› *assms* **by** *auto*
then obtain B::$'a \Rightarrow_{CL} 'b$
where $B *_V x = f\ x$ **for** x
using *cblinfun-apply-cases* **by** *blast*
have $B *_V x = \varphi\ x$
if *c1*: $x{\in}S$

```
    for x
  proof−
    have B *V x = f x
      by (simp add: ‹⋀x. B *V x = f x›)
    also have ... = φ x
      using assms complex-vector.construct-basis f-def that
      by (simp add: complex-vector.construct-basis)
    finally show ?thesis by blast
  qed
  thus ?thesis
    unfolding cblinfun-extension-exists-def
    by blast
qed

lemma cblinfun-extension-apply:
  assumes cblinfun-extension-exists S f
    and v ∈ S
  shows (cblinfun-extension S f) *V v = f v
  by (smt assms cblinfun-extension-def cblinfun-extension-exists-def tfl-some)

lemma
  fixes f :: ‹'a::complex-normed-vector ⇒ 'b::cbanach›
  assumes ‹csubspace S›
  assumes ‹closure S = UNIV›
  assumes f-add: ‹⋀x y. x ∈ S ⟹ y ∈ S ⟹ f (x + y) = f x + f y›
  assumes f-scale: ‹⋀c x y. x ∈ S ⟹ f (c *C x) = c *C f x›
  assumes bounded: ‹⋀x. x ∈ S ⟹ norm (f x) ≤ B * norm x›
  shows cblinfun-extension-exists-bounded-dense: ‹cblinfun-extension-exists S f›
   and cblinfun-extension-norm-bounded-dense: ‹B ≥ 0 ⟹ norm (cblinfun-extension
S f) ≤ B›
proof −
  define B' where ‹B' = (if B≤0 then 1 else B)›
  then have bounded': ‹⋀x. x ∈ S ⟹ norm (f x) ≤ B' * norm x›
    using bounded by (metis mult-1 mult-le-0-iff norm-ge-zero order-trans)
  have ‹B' > 0›
    by (simp add: B'-def)

  have ‹∃xi. (xi ⟶ x) ∧ (∀i. xi i ∈ S)› for x
    using assms(2) closure-sequential by blast
  then obtain seq :: ‹'a ⇒ nat ⇒ 'a› where seq-lim: ‹seq x ⟶ x› and seq-S:
‹seq x i ∈ S› for x i
    apply (atomize-elim, subst all-conj-distrib[symmetric])
    apply (rule choice)
    by auto
  define g where ‹g x = lim (λi. f (seq x i))› for x
  have ‹Cauchy (λi. f (seq x i))› for x
  proof (rule CauchyI)
    fix e :: real assume ‹e > 0›
    have ‹Cauchy (seq x)›
```

```
    using LIMSEQ-imp-Cauchy seq-lim by blast
   then obtain M where less-eB: ‹norm (seq x m − seq x n) < e/B'› if ‹n ≥
M› and ‹m ≥ M› for n m
   by atomize-elim (meson CauchyD ‹0 < B'› ‹0 < e› linordered-field-class.divide-pos-pos)
  have ‹norm (f (seq x m) − f (seq x n)) < e› if ‹n ≥ M› and ‹m ≥ M› for n
m
   proof −
    have ‹norm (f (seq x m) − f (seq x n)) = norm (f (seq x m − seq x n))›
     using f-add f-scale seq-S
    by (metis add-diff-cancel assms(1) complex-vector.subspace-diff diff-add-cancel)
    also have ‹... ≤ B' * norm (seq x m − seq x n)›
     apply (rule bounded')
     by (simp add: assms(1) complex-vector.subspace-diff seq-S)
    also from less-eB have ‹... < B' * (e/B')›
      by (meson ‹0 < B'› linordered-semiring-strict-class.mult-strict-left-mono
that)
    also have ‹... ≤ e›
     using ‹0 < B'› by auto
    finally show ?thesis
     by −
   qed
   then show ‹∃ M. ∀ m≥M. ∀ n≥M. norm (f (seq x m) − f (seq x n)) < e›
    by auto
 qed
 then have f-seq-lim: ‹(λi. f (seq x i)) ⟶ g x› for x
  by (simp add: Cauchy-convergent-iff convergent-LIMSEQ-iff g-def)
 have f-xi-lim: ‹(λi. f (xi i)) ⟶ g x› if ‹xi ⟶ x› and ‹⋀i. xi i ∈ S› for
xi x
 proof −
  from seq-lim that
  have ‹(λi. B' * norm (xi i − seq x i)) ⟶ 0›
     by (metis (no-types) ‹0 < B'› cancel-comm-monoid-add-class.diff-cancel
norm-not-less-zero norm-zero tendsto-diff tendsto-norm-zero-iff tendsto-zero-mult-left-iff)
  then have ‹(λi. f (xi i + (−1) *C seq x i)) ⟶ 0›
   apply (rule Lim-null-comparison[rotated])
    using bounded' by (simp add: assms(1) complex-vector.subspace-diff seq-S
that(2))
  then have ‹(λi. f (xi i) − f (seq x i)) ⟶ 0›
   apply (subst (asm) f-add)
    apply (auto simp: that ‹csubspace S› complex-vector.subspace-neg seq-S)[2]
   apply (subst (asm) f-scale)
   by (auto simp: that ‹csubspace S› complex-vector.subspace-neg seq-S)
  then show ‹(λi. f (xi i)) ⟶ g x›
   using Lim-transform f-seq-lim by fastforce
 qed
 have g-add: ‹g (x + y) = g x + g y› for x y
 proof −
  obtain xi :: ‹nat ⇒ 'a› where ‹xi ⟶ x› and ‹xi i ∈ S› for i
   using seq-S seq-lim by auto
```

obtain *yi* :: ‹*nat* ⇒ *'a*› **where** ‹*yi* ⟶ *y*› **and** ‹*yi i* ∈ *S*› **for** *i*
using *seq-S seq-lim* **by** *auto*
have ‹(λ*i*. *xi i* + *yi i*) ⟶ *x* + *y*›
using ‹*xi* ⟶ *x*› ‹*yi* ⟶ *y*› *tendsto-add* **by** *blast*
then have *lim1*: ‹(λ*i*. *f* (*xi i* + *yi i*)) ⟶ *g* (*x* + *y*)›
by (*simp add*: ‹⋀*i*. *xi i* ∈ *S*› ‹⋀*i*. *yi i* ∈ *S*› *assms*(1) *complex-vector.subspace-add f-xi-lim*)
have ‹(λ*i*. *f* (*xi i* + *yi i*)) = (λ*i*. *f* (*xi i*) + *f* (*yi i*))›
by (*simp add*: ‹⋀*i*. *xi i* ∈ *S*› ‹⋀*i*. *yi i* ∈ *S*› *f-add*)
also have ‹... ⟶ *g x* + *g y*›
by (*simp add*: ‹⋀*i*. *xi i* ∈ *S*› ‹⋀*i*. *yi i* ∈ *S*› ‹*xi* ⟶ *x*› ‹*yi* ⟶ *y*› *f-xi-lim tendsto-add*)
finally show *?thesis*
using *lim1 LIMSEQ-unique* **by** *blast*
qed
have *g-scale*: ‹*g* (*c* *$_C$ *x*) = *c* *$_C$ *g x*› **for** *c x*
proof −
obtain *xi* :: ‹*nat* ⇒ *'a*› **where** ‹*xi* ⟶ *x*› **and** ‹*xi i* ∈ *S*› **for** *i*
using *seq-S seq-lim* **by** *auto*
have ‹(λ*i*. *c* *$_C$ *xi i*) ⟶ *c* *$_C$ *x*›
using ‹*xi* ⟶ *x*› *bounded-clinear-scaleC-right clinear-continuous-at isCont-tendsto-compose* **by** *blast*
then have *lim1*: ‹(λ*i*. *f* (*c* *$_C$ *xi i*)) ⟶ *g* (*c* *$_C$ *x*)›
by (*simp add*: ‹⋀*i*. *xi i* ∈ *S*› *assms*(1) *complex-vector.subspace-scale f-xi-lim*)
have ‹(λ*i*. *f* (*c* *$_C$ *xi i*)) = (λ*i*. *c* *$_C$ *f* (*xi i*))›
by (*simp add*: ‹⋀*i*. *xi i* ∈ *S*› *f-scale*)
also have ‹... ⟶ *c* *$_C$ *g x*›
using ‹⋀*i*. *xi i* ∈ *S*› ‹*xi* ⟶ *x*› *bounded-clinear-scaleC-right clinear-continuous-at f-xi-lim isCont-tendsto-compose* **by** *blast*
finally show *?thesis*
using *lim1 LIMSEQ-unique* **by** *blast*
qed

have [*simp*]: ‹*f x* = *g x*› **if** ‹*x* ∈ *S*› **for** *x*
proof −
have ‹(λ-. *x*) ⟶ *x*›
by *auto*
then have ‹(λ-. *f x*) ⟶ *g x*›
using *that* **by** (*rule f-xi-lim*)
then show ‹*f x* = *g x*›
by (*simp add*: *LIMSEQ-const-iff*)
qed

have *g-bounded*: ‹*norm* (*g x*) ≤ *B'* ∗ *norm x*› **for** *x*
proof −
obtain *xi* :: ‹*nat* ⇒ *'a*› **where** ‹*xi* ⟶ *x*› **and** ‹*xi i* ∈ *S*› **for** *i*
using *seq-S seq-lim* **by** *auto*
then have ‹(λ*i*. *f* (*xi i*)) ⟶ *g x*›
using *f-xi-lim* **by** *presburger*

```
  then have ‹(λi. norm (f (xi i))) ⟶ norm (g x)›
    by (metis tendsto-norm)
  moreover have ‹(λi. B' * norm (xi i)) ⟶ B' * norm x›
    by (simp add: ‹xi ⟶ x› tendsto-mult-left tendsto-norm)
  ultimately show ‹norm (g x) ≤ B' * norm x›
    apply (rule lim-mono[rotated])
    using bounded' using ‹xi - ∈ S› by blast
qed

have ‹bounded-clinear g›
  using g-add g-scale apply (rule bounded-clinearI[where K=B'])
  using g-bounded by (simp add: ordered-field-class.sign-simps(5))
then have [simp]: ‹CBlinfun g *V x = g x› for x
  by (subst CBlinfun-inverse, auto)

show ‹cblinfun-extension-exists S f›
  apply (rule cblinfun-extension-existsI[where B=‹CBlinfun g›])
  by auto

then have ‹cblinfun-extension S f *V ψ = CBlinfun g *V ψ› if ‹ψ ∈ S› for ψ
  by (simp add: cblinfun-extension-apply that)

then have ext-is-g: ‹(*V) (cblinfun-extension S f) = g›
  apply (rule-tac ext)
  apply (rule on-closure-eqI[where S=S])
  using  ‹closure S = UNIV› ‹bounded-clinear g›
 by (auto simp add: continuous-at-imp-continuous-on clinear-continuous-within)

show ‹norm (cblinfun-extension S f) ≤ B› if ‹B ≥ 0›
proof (cases ‹B > 0›)
  case True
  then have ‹B = B'›
    unfolding B'-def
    by auto
  moreover have *: ‹norm (cblinfun-extension S f) ≤ B'›
    by (metis ext-is-g ‹0 < B'› g-bounded norm-cblinfun-bound order-le-less)
  ultimately show ?thesis
    by simp
next
  case False
  with bounded have ‹f x = 0› if ‹x ∈ S› for x
    by (smt (verit) mult-nonpos-nonneg norm-ge-zero norm-le-zero-iff that)
  then have ‹g x = (λ-. 0) x› if ‹x ∈ S› for x
    using that by simp
  then have ‹g x = 0› for x
    apply (rule on-closure-eqI[where S=S])
    using ‹closure S = UNIV› ‹bounded-clinear g›
  by (auto simp add: continuous-at-imp-continuous-on clinear-continuous-within)
  with ext-is-g have ‹cblinfun-extension S f = 0›
```

```
    by (simp add: cblinfun-eqI)
  then show ?thesis
    using that by simp
 qed
qed

lemma cblinfun-extension-cong:
 assumes ‹cspan A = cspan B›
 assumes ‹B ⊆ A›
 assumes fg: ‹⋀x. x∈B ⟹ f x = g x›
 assumes ‹cblinfun-extension-exists A f›
 shows ‹cblinfun-extension A f = cblinfun-extension B g›
proof −
 from ‹cblinfun-extension-exists A f› fg ‹B ⊆ A›
 have ‹cblinfun-extension-exists B g›
  by (metis assms(2) cblinfun-extension-exists-def subset-eq)

 have ‹(∀x∈A. C *V x = f x) ⟷ (∀x∈B. C *V x = f x)› for C
   by (smt (verit, ccfv-SIG) assms(1) assms(2) assms(4) cblinfun-eq-on-span
cblinfun-extension-exists-def complex-vector.span-eq subset-iff)
 also from fg have ‹... C ⟷ (∀x∈B. C *V x = g x)› for C
  by auto
 finally show ‹cblinfun-extension A f = cblinfun-extension B g›
  unfolding cblinfun-extension-def
  by auto
qed

lemma
 fixes f :: ‹'a::complex-inner ⇒ 'b::chilbert-space› and S
 assumes ‹is-ortho-set S› and ‹closure (cspan S) = UNIV›
 assumes ortho-f: ‹⋀x y. x∈S ⟹ y∈S ⟹ x≠y ⟹ is-orthogonal (f x) (f y)›
 assumes bounded: ‹⋀x. x ∈ S ⟹ norm (f x) ≤ B * norm x›
 shows cblinfun-extension-exists-ortho: ‹cblinfun-extension-exists S f›
  and cblinfun-extension-exists-ortho-norm: ‹B ≥ 0 ⟹ norm (cblinfun-extension
S f) ≤ B›
proof −
 define g where ‹g = cconstruct S f›
 have ‹cindependent S›
  using assms(1) is-ortho-set-cindependent by blast
 have g-f: ‹g x = f x› if ‹x∈S› for x
  unfolding g-def using ‹cindependent S› that by (rule complex-vector.construct-basis)
 have [simp]: ‹clinear g›
  unfolding g-def using ‹cindependent S› by (rule complex-vector.linear-construct)
 then have g-add: ‹g (x + y) = g x + g y› if ‹x ∈ cspan S› and ‹y ∈ cspan S›
for x y
  using clinear-iff by blast
 from ‹clinear g› have g-scale: ‹g (c *C x) = c *C g x› if ‹x ∈ cspan S› for x c
  by (simp add: complex-vector.linear-scale)
 moreover have g-bounded: ‹norm (g x) ≤ abs B * norm x› if ‹x ∈ cspan S›
```

for x
proof −
from *that* **obtain** t r **where** *x-sum*: ‹$x = (\sum a{\in}t.\ r\ a *_C\ a)$› **and** ‹*finite* t›
and ‹$t \subseteq S$›
unfolding *complex-vector.span-explicit* **by** *auto*
have ‹$(norm\ (g\ x))^2 = (norm\ (\sum a{\in}t.\ r\ a *_C\ g\ a))^2$›
by (*simp add*: *x-sum complex-vector.linear-sum clinear.scaleC*)
also have ‹$\ldots = (norm\ (\sum a{\in}t.\ r\ a *_C\ f\ a))^2$›
by (*smt* (*verit*) ‹$t \subseteq S$› *g-f in-mono sum.cong*)
also have ‹$\ldots = (\sum a{\in}t.\ (norm\ (r\ a *_C\ f\ a))^2)$›
using - ‹*finite* t› **apply** (*rule pythagorean-theorem-sum*)
using ‹$t \subseteq S$› *ortho-f in-mono* **by** *fastforce*
also have ‹$\ldots = (\sum a{\in}t.\ (cmod\ (r\ a) * norm\ (f\ a))^2)$›
by *simp*
also have ‹$\ldots \le (\sum a{\in}t.\ (cmod\ (r\ a) * B * norm\ a)^2)$›
apply (*rule sum-mono*)
by (*metis* ‹$t \subseteq S$› *assms*(*4*) *in-mono mult-left-mono mult-nonneg-nonneg norm-ge-zero power-mono vector-space-over-itself.scale-scale*)
also have ‹$\ldots = B^2 * (\sum a{\in}t.\ (norm\ (r\ a *_C\ a))^2)$›
by (*simp add*: *sum-distrib-left mult.commute vector-space-over-itself.scale-left-commute flip*: *power-mult-distrib*)
also have ‹$\ldots = B^2 * (norm\ (\sum a{\in}t.\ (r\ a *_C\ a)))^2$›
apply (*subst pythagorean-theorem-sum*)
using ‹*finite* t› **by** *auto* (*meson* ‹$t \subseteq S$› *assms*(*1*) *is-ortho-set-def subsetD*)
also have ‹$\ldots = (abs\ B * norm\ x)^2$›
by (*simp add*: *power-mult-distrib x-sum*)
finally show ‹$norm\ (g\ x) \le abs\ B * norm\ x$›
by *auto*
qed

from *g-add g-scale g-bounded*
have *extg-exists*: ‹*cblinfun-extension-exists* (*cspan S*) *g*›
apply (*rule-tac cblinfun-extension-exists-bounded-dense*[**where** B=‹*abs B*›])
using ‹*closure* (*cspan S*) = *UNIV*› **by** *auto*

then show ‹*cblinfun-extension-exists S f*›
by (*metis* (*mono-tags*, *opaque-lifting*) *g-f cblinfun-extension-apply cblinfun-extension-existsI complex-vector.span-base*)

have *norm-extg*: ‹*norm* (*cblinfun-extension* (*cspan S*) *g*) ≤ *B*› **if** ‹*B* ≥ *0*›
apply (*rule cblinfun-extension-norm-bounded-dense*)
using *g-add g-scale g-bounded* ‹*closure* (*cspan S*) = *UNIV*› *that* **by** *auto*

have *extg-extf*: ‹*cblinfun-extension* (*cspan S*) *g* = *cblinfun-extension S f*›
apply (*rule cblinfun-extension-cong*)
by (*auto simp add*: *complex-vector.span-base g-f extg-exists*)

from *norm-extg extg-extf*
show ‹*norm* (*cblinfun-extension S f*) ≤ *B*› **if** ‹*B* ≥ *0*›

using *that* **by** *simp*
qed

lemma *cblinfun-extension-exists-proj*:
fixes $f :: \langle 'a::complex\text{-}normed\text{-}vector \Rightarrow 'b::cbanach\rangle$
assumes $\langle csubspace\ S\rangle$
assumes *ex-P*: $\langle \exists P :: 'a \Rightarrow_{CL} 'a.\ is\text{-}Proj\ P \wedge range\ P = closure\ S\rangle$
assumes *f-add*: $\langle \bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow f\ (x + y) = f\ x + f\ y\rangle$
assumes *f-scale*: $\langle \bigwedge c\ x\ y.\ x \in S \Longrightarrow f\ (c *_C x) = c *_C f\ x\rangle$
assumes *bounded*: $\langle \bigwedge x.\ x \in S \Longrightarrow norm\ (f\ x) \leq B * norm\ x\rangle$
shows $\langle cblinfun\text{-}extension\text{-}exists\ S\ f\rangle$
— We cannot give a statement about the norm. While there is an extension with norm B, there is no guarantee that *cblinfun-extension S f* returns that specific extension since the extension is only determined on *ccspan S*.
proof (*cases* $\langle B \geq 0\rangle$)
case *True*
note *True*[*simp*]
obtain $P :: \langle 'a \Rightarrow_{CL} 'a\rangle$ **where** *P-proj*: $\langle is\text{-}Proj\ P\rangle$ **and** *P-im*: $\langle range\ P = closure\ S\rangle$
using *ex-P* **by** *blast*
define $f'\ S'$ **where** $\langle f'\ \psi = f\ (P\ \psi)\rangle$ **and** $\langle S' = S + space\text{-}as\text{-}set\ (kernel\ P)\rangle$ **for** ψ
have *PS'*: $\langle P *_V x \in S\rangle$ **if** $\langle x \in S'\rangle$ **for** x
proof −
obtain *x1 x2* **where** *x12*: $\langle x = x1 + x2\rangle$ **and** *x1*: $\langle x1 \in S\rangle$ **and** *x2*: $\langle x2 \in space\text{-}as\text{-}set\ (kernel\ P)\rangle$
using *that S'-def* $\langle x \in S'\rangle$ *set-plus-elim* **by** *blast*
have $\langle P *_V x = P *_V x1\rangle$
using *x2* **by** (*simp add*: *x12 cblinfun.add-right kernel-memberD*)
also have $\langle \ldots = x1\rangle$
by (*metis* (*no-types*, *opaque-lifting*) *P-im P-proj cblinfun-apply-cblinfun-compose closure-insert image-iff insertI1 insert-absorb is-Proj-idempotent x1*)
also have $\langle \ldots \in S\rangle$
by (*fact x1*)
finally show *?thesis*
by −
qed
have *SS'*: $\langle S \subseteq S'\rangle$
by (*metis S'-def ordered-field-class.sign-simps*(*2*) *set-zero-plus2 zero-space-as-set*)

have $\langle csubspace\ S'\rangle$
by (*simp add*: *S'-def assms*(*1*) *csubspace-set-plus*)
moreover have $\langle closure\ S' = UNIV\rangle$
proof *auto*
fix ψ
have $\langle \psi = P\ \psi + (id - P)\ \psi\rangle$
by *simp*
also have $\langle \ldots \in closure\ S + space\text{-}as\text{-}set\ (kernel\ P)\rangle$

by (*smt* (*verit*) *P-im P-proj calculation cblinfun.real.add-right eq-add-iff is-Proj-idempotent kernel-memberI rangeI set-plus-intro simp-a-oCL-b'*)
also have ‹... ⊆ *closure* (*closure S* + *space-as-set* (*kernel P*))›
using *closure-subset* **by** *blast*
also have ‹... = *closure* (*S* + *space-as-set* (*kernel P*))›
using *closed-sum-closure-left closed-sum-def* **by** *blast*
also have ‹... = *closure S'*›
using *S'-def* **by** *fastforce*
finally show ‹*ψ* ∈ *closure S'*›
by −
qed

moreover have ‹*f'* (*x* + *y*) = *f'* *x* + *f'* *y*› **if** ‹*x* ∈ *S'*› **and** ‹*y* ∈ *S'*› **for** *x y*
using *that* **by** (*auto simp*: *f'-def cblinfun.add-right f-add PS'*)
moreover have ‹*f'* (*c* *$_C$ *x*) = *c* *$_C$ *f'* *x*› **if** ‹*x* ∈ *S'*› **for** *c x*
using *that* **by** (*auto simp*: *f'-def cblinfun.scaleC-right f-scale PS'*)
moreover
have ‹*norm* (*f'* *x*) ≤ (*B* ∗ *norm P*) ∗ *norm x*› **if** ‹*x* ∈ *S'*› **for** *x*
proof −
have ‹*norm* (*f'* *x*) ≤ *B*∗ *norm* (*P x*)›
by (*auto simp*: *f'-def PS' bounded that*)
also have ‹... ≤ *B* ∗ *norm P* ∗ *norm x*›
by (*simp add*: *mult.assoc mult-left-mono norm-cblinfun*)
finally show *?thesis*
by *auto*
qed

ultimately have *F-ex*: ‹*cblinfun-extension-exists S' f'*›
by (*rule cblinfun-extension-exists-bounded-dense*)
define *F* **where** ‹*F* = *cblinfun-extension S' f'*›
have ‹*F* *ψ* = *f* *ψ*› **if** ‹*ψ* ∈ *S*› **for** *ψ*
proof −
from *F-ex that SS'* **have** ‹*F* *ψ* = *f'* *ψ*›
by (*auto simp add*: *F-def cblinfun-extension-apply*)
also have ‹... = *f* (*P* *$_V$ *ψ*)›
by (*simp add*: *f'-def*)
also have ‹... = *f* *ψ*›
using *P-proj P-im*
apply (*transfer fixing*: *ψ S f*)
by (*metis closure-subset in-mono is-projection-on-fixes-image is-projection-on-image that*)
finally show *?thesis*
by −
qed
then show ‹*cblinfun-extension-exists S f*›
using *cblinfun-extension-exists-def* **by** *blast*
next
case *False*
then have ‹*S* ⊆ {*0*}›

using *bounded* **by** *auto* (*meson norm-ge-zero norm-le-zero-iff order-trans zero-le-mult-iff*)
then show ‹*cblinfun-extension-exists S f*›
apply (*rule-tac cblinfun-extension-existsI*[**where** *B=0*])
apply *auto*
using *bounded* **by** *fastforce*
qed

lemma *cblinfun-extension-exists-hilbert*:
fixes *f* :: ‹*'a*::*chilbert-space* ⇒ *'b*::*cbanach*›
assumes ‹*csubspace S*›
assumes *f-add*: ‹⋀*x y*. *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *f* (*x* + *y*) = *f x* + *f y*›
assumes *f-scale*: ‹⋀*c x y*. *x* ∈ *S* ⟹ *f* (*c* ∗*C x*) = *c* ∗*C f x*›
assumes *bounded*: ‹⋀*x*. *x* ∈ *S* ⟹ *norm* (*f x*) ≤ *B* ∗ *norm x*›
shows ‹*cblinfun-extension-exists S f*›
— We cannot give a statement about the norm. While there is an extension with norm *B*, there is no guarantee that *cblinfun-extension S f* returns that specific extension since the extension is only determined on *ccspan S*.
proof −
have ‹∃*P*. *is-Proj P* ∧ *range* ((∗*V*) *P*) = *closure S*›
apply (*rule exI*[*of* - ‹*Proj* (*ccspan S*)›])
apply (*rule conjI*)
by *simp* (*metis Proj-is-Proj Proj-range Proj-range-closed assms*(*1*) *cblinfun-image.rep-eq ccspan.rep-eq closure-closed complex-vector.span-eq-iff space-as-set-top*)

from *assms*(*1*) *this assms*(*2*−*4*)
show *?thesis*
by (*rule cblinfun-extension-exists-proj*)
qed

lemma *cblinfun-extension-exists-restrict*:
assumes ‹*B* ⊆ *A*›
assumes ‹⋀*x*. *x*∈*B* ⟹ *f x* = *g x*›
assumes ‹*cblinfun-extension-exists A f*›
shows ‹*cblinfun-extension-exists B g*›
by (*metis assms cblinfun-extension-exists-def subset-eq*)

13.22 Bijections between different ONBs

Some of the theorems here logically belong into *Complex-Bounded-Operators.Complex-Inner-Product* but the proof uses some concepts from the present theory.

lemma *all-ortho-bases-same-card*:
— Follows [1], Proposition 4.14
fixes *E F* :: ‹*'a*::*chilbert-space set*›
assumes ‹*is-ortho-set E*› ‹*is-ortho-set F*› ‹*ccspan E* = ⊤› ‹*ccspan F* = ⊤›
shows ‹∃*f*. *bij-betw f E F*›
proof −
have ‹|*F*| ≤*o* |*E*|› **if** ‹*infinite E*› **and**
‹*is-ortho-set E*› ‹*is-ortho-set F*› ‹*ccspan E* = *top*› ‹*ccspan F* = *top*› **for** *E F* ::
‹*'a set*›

proof −
define F' **where** ‹$F'\ e = \{f{\in}F.\ \neg\ is\text{-}orthogonal\ f\ e\}$› **for** e
have ‹$\exists\, e{\in}E.\ cinner\ f\ e \neq 0$› **if** ‹$f \in F$› **for** f
proof (*rule ccontr*, *simp*)
assume ‹$\forall\, e{\in}E.\ is\text{-}orthogonal\ f\ e$›
then have ‹$f \in orthogonal\text{-}complement\ E$›
by (*simp add*: *orthogonal-complementI*)
also have ‹$\ldots = orthogonal\text{-}complement\ (closure\ (cspan\ E))$›
using *orthogonal-complement-of-closure orthogonal-complement-of-cspan* **by** *blast*
also have ‹$\ldots = space\text{-}as\text{-}set\ (-\ ccspan\ E)$›
by *transfer simp*
also have ‹$\ldots = space\text{-}as\text{-}set\ (-\ top)$›
by (*simp add*: ‹$ccspan\ E = top$›)
also have ‹$\ldots = \{0\}$›
by (*auto simp add*: *top-ccsubspace.rep-eq uminus-ccsubspace.rep-eq*)
finally have ‹$f = 0$›
by *simp*
with ‹$f \in F$› ‹$is\text{-}ortho\text{-}set\ F$› **show** *False*
by (*simp add*: *is-onb-def is-ortho-set-def*)
qed
then have *F′-union*: ‹$F = (\bigcup e{\in}E.\ F'\ e)$›
unfolding *F′-def* **by** *auto*
have ‹$countable\ (F'\ e)$› **for** e
proof −
have ‹$(\sum f{\in}M.\ (cmod\ (cinner\ (sgn\ f)\ e))^2) \leq (norm\ e)^2$› **if** ‹$finite\ M$› **and** ‹$M \subseteq F$› **for** M
proof −
have [*simp*]: ‹$is\text{-}ortho\text{-}set\ M$›
using ‹$is\text{-}ortho\text{-}set\ F$› *is-ortho-set-antimono that*(*2*) **by** *blast*
have [*simp*]: ‹$norm\ (sgn\ f) = 1$› **if** ‹$f \in M$› **for** f
by (*metis* ‹$is\text{-}ortho\text{-}set\ M$› *is-ortho-set-def norm-sgn that*)
have ‹$(\sum f{\in}M.\ (cmod\ (cinner\ (sgn\ f)\ e))^2) = (\sum f{\in}M.\ (norm\ ((cinner\ (sgn\ f)\ e) *_C sgn\ f))^2)$›
apply (*rule sum.cong*[*OF refl*])
by *simp*
also have ‹$\ldots = (norm\ (\sum f{\in}M.\ ((cinner\ (sgn\ f)\ e) *_C sgn\ f)))^2$›
apply (*rule pythagorean-theorem-sum*[*symmetric*])
using *that* **by** *auto* (*meson* ‹$is\text{-}ortho\text{-}set\ M$› *is-ortho-set-def*)
also have ‹$\ldots = (norm\ (\sum f{\in}M.\ proj\ f\ e))^2$›
by (*metis butterfly-apply butterfly-sgn-eq-proj*)
also have ‹$\ldots = (norm\ (Proj\ (ccspan\ M)\ e))^2$›
apply (*rule arg-cong*[**where** $f{=}\lambda x.\ (norm\ x)^2$›])
using ‹$finite\ M$› ‹$is\text{-}ortho\text{-}set\ M$› **apply** *induction*
by *simp* (*smt* (*verit*, *ccfv-threshold*) *Proj-orthog-ccspan-insert insertCI is-ortho-set-def plus-cblinfun.rep-eq sum.insert*)
also have ‹$\ldots \leq (norm\ (Proj\ (ccspan\ M)) * norm\ e)^2$›
by (*auto simp*: *norm-cblinfun intro*!: *power-mono*)
also have ‹$\ldots \leq (norm\ e)^2$›

```
      apply (rule power-mono)
       apply (metis norm-Proj-leq1 mult-left-le-one-le norm-ge-zero)
      by simp
    finally show ?thesis
      by −
  qed
  then have ‹(λf. (cmod (cinner (sgn f) e))^2) abs-summable-on F›
    apply (intro nonneg-bdd-above-summable-on bdd-aboveI)
    by auto
  then have ‹countable {f ∈ F. (cmod (sgn f ·_C e))^2 ≠ 0}›
    by (rule abs-summable-countable)
  then have ‹countable {f ∈ F. ¬ is-orthogonal (sgn f) e}›
    by force
  then have ‹countable {f ∈ F. ¬ is-orthogonal f e}›
    by force
  then show ?thesis
    unfolding F'-def by simp
qed
then have F'-leq: ‹|F' e| ≤o natLeq› for e
  using countable-leq-natLeq by auto

from F'-union have ‹|F| ≤o |Sigma E F'|› (is ‹- ≤o ...›)
  using card-of-UNION-Sigma by blast
also have ‹... ≤o |E × (UNIV::nat set)|› (is ‹- ≤o ...›)
  apply (rule card-of-Sigma-mono1)
  using F'-leq
  using card-of-nat ordIso-symmetric ordLeq-ordIso-trans by blast
also have ‹... =o |E| *c natLeq› (is ‹ordIso2 - ...›)
  by (metis Field-card-of Field-natLeq card-of-ordIso-subst cprod-def)
also have ‹... =o |E|›
  apply (rule card-prod-omega)
  using that by (simp add: cinfinite-def)
finally show ‹|F| ≤o |E|›
  by −
qed
then have infinite: ‹|E| =o |F|› if ‹infinite E› and ‹infinite F›
  by (simp add: assms ordIso-iff-ordLeq that(1) that(2))

have ‹|E| =o |F|› if ‹finite E› and ‹is-ortho-set E› ‹is-ortho-set F› ‹ccspan E
= top› ‹ccspan F = top›
  for E F :: ‹'a set›
proof −
  have ‹card E = card F›
    using that
  by (metis bij-betw-same-card ccspan.rep-eq closure-finite-cspan complex-vector.bij-if-span-eq-span-bases

    complex-vector.independent-span-bound is-ortho-set-cindependent top-ccsubspace.rep-eq
top-greatest)
  with ‹finite E› have ‹finite F›
```

by (*metis ccspan.rep-eq closure-finite-cspan complex-vector.independent-span-bound is-ortho-set-cindependent that(3) that(4) top-ccsubspace.rep-eq top-greatest*)
with ‹*card E = card F*› *that* **show** *?thesis*
apply (*rule-tac finite-card-of-iff-card*[*THEN iffD2*])
by *auto*
qed

with *infinite assms* **have** ‹|*E*| =*o* |*F*|›
by (*meson ordIso-symmetric*)

then show *?thesis*
by (*simp add*: *card-of-ordIso*)
qed

lemma *all-onbs-same-card*:
fixes *E F* :: ‹'*a*::*chilbert-space set*›
assumes ‹*is-onb E*› ‹*is-onb F*›
shows ‹∃*f*. *bij-betw f E F*›
apply (*rule all-ortho-bases-same-card*)
using *assms* **by** (*auto simp*: *is-onb-def*)

definition *bij-between-bases* **where** ‹*bij-between-bases E F* = (*SOME f*. *bij-betw f E F*)› **for** *E F* :: ‹'*a*::*chilbert-space set*›

lemma *bij-between-bases-bij*:
fixes *E F* :: ‹'*a*::*chilbert-space set*›
assumes ‹*is-onb E*› ‹*is-onb F*›
shows ‹*bij-betw* (*bij-between-bases E F*) *E F*›
using *all-onbs-same-card*
by (*metis assms*(*1*) *assms*(*2*) *bij-between-bases-def someI*)

definition *unitary-between* **where** ‹*unitary-between E F* = *cblinfun-extension E* (*bij-between-bases E F*)›

lemma *unitary-between-apply*:
fixes *E F* :: ‹'*a*::*chilbert-space set*›
assumes ‹*is-onb E*› ‹*is-onb F*› ‹*e* ∈ *E*›
shows ‹*unitary-between E F* ∗$_V$ *e* = *bij-between-bases E F e*›
proof −
from ‹*is-onb E*› ‹*is-onb F*›
have *EF*: ‹*bij-between-bases E F e* ∈ *F*› **if** ‹*e* ∈ *E*› **for** *e*
by (*meson bij-betwE bij-between-bases-bij that*)
have *ortho*: ‹*is-orthogonal* (*bij-between-bases E F x*) (*bij-between-bases E F y*)›
if ‹*x* ≠ *y*› **and** ‹*x* ∈ *E*› **and** ‹*y* ∈ *E*› **for** *x y*
by (*smt* (*verit*, *del-insts*) *assms*(*1*) *assms*(*2*) *bij-betw-iff-bijections bij-between-bases-bij is-onb-def is-ortho-set-def that*(*1*) *that*(*2*) *that*(*3*))
have *spanE*: ‹*closure* (*cspan E*) = *UNIV*›
by (*metis assms*(*1*) *ccspan.rep-eq is-onb-def top-ccsubspace.rep-eq*)
show *?thesis*

```
    unfolding unitary-between-def
    apply (rule cblinfun-extension-apply)
     apply (rule cblinfun-extension-exists-ortho[where B=1])
    using assms EF ortho spanE
    by (auto simp: is-onb-def)
qed

lemma unitary-between-unitary:
  fixes E F :: ‹'a::chilbert-space set›
  assumes ‹is-onb E› ‹is-onb F›
  shows ‹unitary (unitary-between E F)›
proof −
  have ‹(unitary-between E F *V b) •C (unitary-between E F *V c) = b •C c› if
‹b ∈ E› and ‹c ∈ E› for b c
  proof (cases ‹b = c›)
    case True
    from ‹is-onb E› that have 1: ‹b •C b = 1›
      using cnorm-eq-1 is-onb-def by blast
    from that have ‹unitary-between E F *V b ∈ F›
    by (metis assms(1) assms(2) bij-betw-apply bij-between-bases-bij unitary-between-apply)
    with ‹is-onb F› have 2: ‹(unitary-between E F *V b) •C (unitary-between E F
*V b) = 1›
      by (simp add: cnorm-eq-1 is-onb-def)
    from 1 2 True show ?thesis
      by simp
  next
    case False
    from ‹is-onb E› that have 1: ‹b •C c = 0›
      by (simp add: False is-onb-def is-ortho-set-def)
   from that have inF: ‹unitary-between E F *V b ∈ F› ‹unitary-between E F *V
c ∈ F›
    by (metis assms(1) assms(2) bij-betw-apply bij-between-bases-bij unitary-between-apply)+
    have neq: ‹unitary-between E F *V b ≠ unitary-between E F *V c›
      by (metis (no-types, lifting) False assms(1) assms(2) bij-betw-iff-bijections
bij-between-bases-bij that(1) that(2) unitary-between-apply)
   from inF neq ‹is-onb F› have 2: ‹(unitary-between E F *V b) •C (unitary-between
E F *V c) = 0›
      by (simp add: is-onb-def is-ortho-set-def)
    from 1 2 show ?thesis
      by simp
  qed
  then have iso: ‹isometry (unitary-between E F)›
    apply (rule-tac orthogonal-on-basis-is-isometry[where B=E])
    using assms(1) is-onb-def by auto
  have ‹unitary-between E F *S top = unitary-between E F *S ccspan E›
    by (metis assms(1) is-onb-def)
  also have ‹... ≥ ccspan (unitary-between E F ‘ E)› (is ‹- ≥ ...›)
    by (simp add: cblinfun-image-ccspan)
  also have ‹... = ccspan (bij-between-bases E F ‘ E)›
```

by (*metis assms(1) assms(2) image-cong unitary-between-apply*)
also have ‹... = *ccspan F*›
by (*metis assms(1) assms(2) bij-betw-imp-surj-on bij-between-bases-bij*)
also have ‹... = *top*›
using *assms(2) is-onb-def* **by** *blast*
finally have *surj*: ‹*unitary-between E F* $*_S$ *top* = *top*›
by (*simp add*: *top.extremum-unique*)
from *iso surj* **show** *?thesis*
by (*rule surj-isometry-is-unitary*)
qed

13.23 Notation

bundle *cblinfun-syntax* **begin**
notation *cblinfun-compose* (**infixl** ‹o_{CL}› *67*)
notation *cblinfun-apply* (**infixr** ‹$*_V$› *70*)
notation *cblinfun-image* (**infixr** ‹$*_S$› *70*)
notation *adj* (‹-*› [*99*] *100*)
type-notation *cblinfun* (‹(- $\Rightarrow_{CL}$ /-)› [*22*, *21*] *21*)
end

unbundle *no cblinfun-syntax* **and** *no lattice-syntax*

end

14 *Complex-L2* – Hilbert space of square-summable functions

theory *Complex-L2*
imports
Complex-Bounded-Linear-Function

HOL−Analysis.L2-Norm
HOL−Library.Rewrite
HOL−Analysis.Infinite-Sum
HOL−Library.Infinite-Typeclass
begin

unbundle *lattice-syntax* **and** *cblinfun-syntax* **and** *no blinfun-apply-syntax*

14.1 l2 norm of functions

definition ‹*has-ell2-norm* (*x*::-⇒*complex*) ⟷ (λ*i*. (*x i*)^2) *abs-summable-on UNIV*›

lemma *has-ell2-norm-bdd-above*: ‹*has-ell2-norm x* ⟷ *bdd-above* (*sum* (λ*xa*. *norm* ((*x xa*)^2)) ‘ *Collect finite*)›
by (*simp add*: *has-ell2-norm-def abs-summable-iff-bdd-above*)

lemma *has-ell2-norm-L2-set*: *has-ell2-norm x = bdd-above (L2-set (norm o x) ‘ Collect finite)*
proof (*rule iffI*)
have ‹*mono sqrt*›
using *monoI real-sqrt-le-mono* **by** *blast*
assume ‹*has-ell2-norm x*›
then have ∗: ‹*bdd-above (sum (λxa. norm ((x xa)^2)) ‘ Collect finite)*›
by (*subst* (*asm*) *has-ell2-norm-bdd-above*)
have ‹*bdd-above ((λF. sqrt (sum (λxa. norm ((x xa)^2)) F)) ‘ Collect finite)*›
using *bdd-above-image-mono*[*OF* ‹*mono sqrt*› ∗]
by (*auto simp*: *image-image*)
then show ‹*bdd-above (L2-set (norm o x) ‘ Collect finite)*›
by (*auto simp*: *L2-set-def norm-power*)
next
define *p2* **where** ‹*p2 x = (if x < 0 then 0 else x^2)*› **for** *x* :: *real*
have ‹*mono p2*›
by (*simp add*: *monoI p2-def*)
have [*simp*]: ‹*p2 (L2-set f F) = (∑ i∈F. (f i)^2)*› **for** *f* **and** *F* :: ‹*'a set*›
by (*smt* (*verit*) *L2-set-def L2-set-nonneg p2-def power2-less-0 real-sqrt-pow2 sum.cong sum-nonneg*)
assume ∗: ‹*bdd-above (L2-set (norm o x) ‘ Collect finite)*›
have ‹*bdd-above (p2 ‘ L2-set (norm o x) ‘ Collect finite)*›
using *bdd-above-image-mono*[*OF* ‹*mono p2*› ∗]
by *auto*
then show ‹*has-ell2-norm x*›
apply (*simp add*: *image-image has-ell2-norm-def abs-summable-iff-bdd-above*)
by (*simp add*: *norm-power*)
qed

definition *ell2-norm* :: ‹(*'a* ⇒ *complex*) ⇒ *real*› **where** ‹*ell2-norm f = sqrt (∑∞x. norm (f x)^2)*›

lemma *ell2-norm-SUP*:
assumes ‹*has-ell2-norm x*›
shows *ell2-norm x = sqrt (SUP F∈{F. finite F}. sum (λi. norm (x i)^2) F)*
using *assms* **apply** (*auto simp add*: *ell2-norm-def has-ell2-norm-def*)
apply (*subst infsum-nonneg-is-SUPREMUM-real*)
by (*auto simp*: *norm-power*)

lemma *ell2-norm-L2-set*:
assumes *has-ell2-norm x*
shows *ell2-norm x = (SUP F∈{F. finite F}. L2-set (norm o x) F)*
proof−
have *sqrt (⨆ (sum (λi. (cmod (x i))^2) ‘ Collect finite)) =*
(SUP F∈{F. finite F}. sqrt (∑ i∈F. (cmod (x i))^2))
proof (*subst continuous-at-Sup-mono*)
show *mono sqrt*
by (*simp add*: *mono-def*)
show *continuous (at-left (⨆ (sum (λi. (cmod (x i))^2) ‘ Collect finite))) sqrt*

using *continuous-at-split isCont-real-sqrt* **by** *blast*
show *sum* $(\lambda i.\ (cmod\ (x\ i))^2)$ ‘ *Collect finite* $\neq \{\}$
by *auto*
show *bdd-above* $(sum\ (\lambda i.\ (cmod\ (x\ i))^2)$ ‘ *Collect finite*)
using *has-ell2-norm-bdd-above*[*THEN iffD1*, *OF assms*] **by** (*auto simp*: *norm-power*)
show $\bigsqcup$ $(sqrt$ ‘ $sum\ (\lambda i.\ (cmod\ (x\ i))^2)$ ‘ *Collect finite*) = $(SUP\ F{\in}Collect$ *finite*. *sqrt* $(\sum i{\in}F.\ (cmod\ (x\ i))^2))$
by (*metis image-image*)
qed
thus *?thesis*
using *assms* **by** (*auto simp*: *ell2-norm-SUP L2-set-def*)
qed

lemma *has-ell2-norm-finite*[*simp*]: *has-ell2-norm* $(f::'a::finite{\Rightarrow}\text{-})$
unfolding *has-ell2-norm-def* **by** *simp*

lemma *ell2-norm-finite*:
ell2-norm $(f::'a::finite{\Rightarrow}complex) = sqrt\ (\sum x{\in}UNIV.\ (norm\ (f\ x))\ \hat{}\ 2)$
by (*simp add*: *ell2-norm-def*)

lemma *ell2-norm-finite-L2-set*: *ell2-norm* $(x::'a::finite{\Rightarrow}complex) = L2\text{-}set\ (norm\ o\ x)\ UNIV$
by (*simp add*: *ell2-norm-finite L2-set-def*)

lemma *ell2-norm-square*: ‹$(ell2\text{-}norm\ x)^2 = (\sum_\infty i.\ (cmod\ (x\ i))^2)$›
unfolding *ell2-norm-def*
apply (*subst real-sqrt-pow2*)
by (*simp-all add*: *infsum-nonneg*)

lemma *ell2-ket*:
fixes *a*
defines ‹$f \equiv (\lambda i.\ of\text{-}bool\ (a = i))$›
shows *has-ell2-norm-ket*: ‹*has-ell2-norm f*›
and *ell2-norm-ket*: ‹*ell2-norm f* = *1*›
proof −
have ‹$(\lambda x.\ (f\ x)^2)$ *abs-summable-on* $\{a\}$›
apply (*rule summable-on-finite*) **by** *simp*
then show ‹*has-ell2-norm f*›
unfolding *has-ell2-norm-def*
apply (*rule summable-on-cong-neutral*[*THEN iffD1*, *rotated* −*1*])
unfolding *f-def* **by** *auto*

have ‹$(\sum_\infty x{\in}\{a\}.\ (f\ x)^2) = 1$›
apply (*subst infsum-finite*)
by (*auto simp*: *f-def*)
then show ‹*ell2-norm f* = *1*›
unfolding *ell2-norm-def*
apply (*subst infsum-cong-neutral*[**where** T=‹$\{a\}$› **and** g=‹$\lambda x.\ (cmod\ (f\ x))^2$›])

```
    by (auto simp: f-def)
qed

lemma ell2-norm-geq0: ‹ell2-norm x ≥ 0›
  by (auto simp: ell2-norm-def intro!: infsum-nonneg)

lemma ell2-norm-point-bound:
  assumes ‹has-ell2-norm x›
  shows ‹ell2-norm x ≥ cmod (x i)›
proof −
  have ‹(cmod (x i))² = norm ((x i)²)›
    by (simp add: norm-power)
  also have ‹norm ((x i)²) = sum (λi. (norm ((x i)²))) {i}›
    by auto
  also have ‹... = infsum (λi. (norm ((x i)²))) {i}›
    by (rule infsum-finite[symmetric], simp)
  also have ‹... ≤ infsum (λi. (norm ((x i)²))) UNIV›
    apply (rule infsum-mono-neutral)
    using assms by (auto simp: has-ell2-norm-def)
  also have ‹... = (ell2-norm x)²›
   by (metis (no-types, lifting) ell2-norm-def ell2-norm-geq0 infsum-cong norm-power
real-sqrt-eq-iff real-sqrt-unique)
  finally show ?thesis
    using ell2-norm-geq0 power2-le-imp-le by blast
qed

lemma ell2-norm-0:
  assumes has-ell2-norm x
  shows ell2-norm x = 0 ⟷ x = (λ-. 0)
proof
  assume u1: x = (λ-. 0)
  have u2: (SUP x::'a set∈Collect finite. (0::real)) = 0
    if x = (λ-. 0)
    by (metis cSUP-const empty-Collect-eq finite.emptyI)
  show ell2-norm x = 0
    unfolding ell2-norm-def
    using u1 u2 by auto
next
  assume norm0: ell2-norm x = 0
  show x = (λ-. 0)
  proof
    fix i
    have ‹cmod (x i) ≤ ell2-norm x›
      using assms by (rule ell2-norm-point-bound)
    also have ‹... = 0›
      by (fact norm0)
    finally show x i = 0 by auto
  qed
qed
```

lemma *ell2-norm-smult*:
assumes *has-ell2-norm x*
shows *has-ell2-norm* (λ*i*. *c* ∗ *x i*) **and** *ell2-norm* (λ*i*. *c* ∗ *x i*) = *cmod c* ∗ *ell2-norm x*
proof −
have *L2-set-mul*: *L2-set* (*cmod* ∘ (λ*i*. *c* ∗ *x i*)) *F* = *cmod c* ∗ *L2-set* (*cmod* ∘ *x*) *F* **for** *F*
proof−
have *L2-set* (*cmod* ∘ (λ*i*. *c* ∗ *x i*)) *F* = *L2-set* (λ*i*. (*cmod c* ∗ (*cmod o x*) *i*)) *F*
by (*metis comp-def norm-mult*)
also have ... = *cmod c* ∗ *L2-set* (*cmod o x*) *F*
by (*metis norm-ge-zero L2-set-right-distrib*)
finally show *?thesis* **.**
qed

from *assms* **obtain** *M* **where** *M*: *M* ≥ *L2-set* (*cmod o x*) *F* **if** *finite F* **for** *F*
unfolding *has-ell2-norm-L2-set bdd-above-def* **by** *auto*
hence *cmod c* ∗ *M* ≥ *L2-set* (*cmod o* (λ*i*. *c* ∗ *x i*)) *F* **if** *finite F* **for** *F*
unfolding *L2-set-mul*
by (*simp add*: *ordered-comm-semiring-class.comm-mult-left-mono that*)
thus *has*: *has-ell2-norm* (λ*i*. *c* ∗ *x i*)
unfolding *has-ell2-norm-L2-set bdd-above-def* **using** *L2-set-mul*[*symmetric*] **by** *auto*
have *ell2-norm* (λ*i*. *c* ∗ *x i*) = (*SUP F* ∈ *Collect finite*. (*L2-set* (*cmod* ∘ (λ*i*. *c* ∗ *x i*)) *F*))
by (*simp add*: *ell2-norm-L2-set has*)
also have ... = (*SUP F* ∈ *Collect finite*. (*cmod c* ∗ *L2-set* (*cmod* ∘ *x*) *F*))
using *L2-set-mul* **by** *auto*
also have ... = *cmod c* ∗ *ell2-norm x*
proof (*subst ell2-norm-L2-set*)
show *has-ell2-norm x*
by (*simp add*: *assms*)
show (*SUP F*∈*Collect finite*. *cmod c* ∗ *L2-set* (*cmod* ∘ *x*) *F*) = *cmod c* ∗ ⨆ (*L2-set* (*cmod* ∘ *x*) ‘ *Collect finite*)
proof (*subst continuous-at-Sup-mono* [**where** *f* = λ*x*. *cmod c* ∗ *x*])
show *mono* ((∗) (*cmod c*))
by (*simp add*: *mono-def ordered-comm-semiring-class.comm-mult-left-mono*)
show *continuous* (*at-left* (⨆ (*L2-set* (*cmod* ∘ *x*) ‘ *Collect finite*))) ((∗) (*cmod c*))
proof (*rule continuous-mult*)
show *continuous* (*at-left* (⨆ (*L2-set* (*cmod* ∘ *x*) ‘ *Collect finite*))) (λ*x*. *cmod c*)
by *simp*
show *continuous* (*at-left* (⨆ (*L2-set* (*cmod* ∘ *x*) ‘ *Collect finite*))) (λ*x*. *x*)
by *simp*
qed
show *L2-set* (*cmod* ∘ *x*) ‘ *Collect finite* ≠ {}

```
      by auto
    show bdd-above (L2-set (cmod ∘ x) ‘ Collect finite)
      by (meson assms has-ell2-norm-L2-set)
    show (SUP F∈Collect finite. cmod c * L2-set (cmod ∘ x) F) = ⨆ ((*) (cmod
c) ‘ L2-set (cmod ∘ x) ‘ Collect finite)
      by (metis image-image)
  qed
 qed
 finally show ell2-norm (λi. c * x i) = cmod c * ell2-norm x.
qed


lemma ell2-norm-triangle:
 assumes has-ell2-norm x and has-ell2-norm y
 shows has-ell2-norm (λi. x i + y i) and ell2-norm (λi. x i + y i) ≤ ell2-norm
x + ell2-norm y
proof −
 have triangle: L2-set (cmod ∘ (λi. x i + y i)) F ≤ L2-set (cmod ∘ x) F + L2-set
(cmod ∘ y) F
   (is ?lhs≤?rhs)
   if finite F for F
 proof −
   have ?lhs ≤ L2-set (λi. (cmod o x) i + (cmod o y) i) F
   proof (rule L2-set-mono)
     show (cmod ∘ (λi. x i + y i)) i ≤ (cmod ∘ x) i + (cmod ∘ y) i
       if i ∈ F
       for i :: 'a
       using that norm-triangle-ineq by auto
     show 0 ≤ (cmod ∘ (λi. x i + y i)) i
       if i ∈ F
       for i :: 'a
       using that
       by simp
   qed
   also have ... ≤ ?rhs
     by (rule L2-set-triangle-ineq)
   finally show ?thesis .
 qed
 obtain Mx My where Mx: Mx ≥ L2-set (cmod o x) F and My: My ≥ L2-set
(cmod o y) F
   if finite F for F
   using assms unfolding has-ell2-norm-L2-set bdd-above-def by auto
 hence MxMy: Mx + My ≥ L2-set (cmod ∘ x) F + L2-set (cmod ∘ y) F if finite
F for F
   using that by fastforce
 hence bdd-plus: bdd-above ((λxa. L2-set (cmod ∘ x) xa + L2-set (cmod ∘ y) xa)
‘ Collect finite)
   unfolding bdd-above-def by auto
 from MxMy have MxMy': Mx + My ≥ L2-set (cmod ∘ (λi. x i + y i)) F if
```

finite F **for** *F*
using *triangle that* **by** *fastforce*
thus *has*: *has-ell2-norm* (λ*i*. *x i* + *y i*)
unfolding *has-ell2-norm-L2-set bdd-above-def* **by** *auto*
have *SUP-plus*: (*SUP x*∈*A*. *f x* + *g x*) ≤ (*SUP x*∈*A*. *f x*) + (*SUP x*∈*A*. *g x*)
if *notempty*: *A*≠{} **and** *bddf*: *bdd-above* (*f*‘*A*)**and** *bddg*: *bdd-above* (*g*‘*A*)
for *f g* :: *'a set* ⇒ *real* **and** *A*
proof−
have *xleq*: *x* ≤ (*SUP x*∈*A*. *f x*) + (*SUP x*∈*A*. *g x*) **if** *x*: *x* ∈ (λ*x*. *f x* + *g x*) ‘ *A* **for** *x*
proof −
obtain *a* **where** *aA*: *a*:*A* **and** *ax*: *x* = *f a* + *g a*
using *x* **by** *blast*
have *fa*: *f a* ≤ (*SUP x*∈*A*. *f x*)
by (*simp add*: *bddf aA cSUP-upper*)
moreover have *g a* ≤ (*SUP x*∈*A*. *g x*)
by (*simp add*: *bddg aA cSUP-upper*)
ultimately have *f a* + *g a* ≤ (*SUP x*∈*A*. *f x*) + (*SUP x*∈*A*. *g x*) **by** *simp*
with *ax* **show** *?thesis* **by** *simp*
qed
have (λ*x*. *f x* + *g x*) ‘ *A* ≠ {}
using *notempty* **by** *auto*
moreover have *x* ≤ ⨆ (*f* ‘ *A*) + ⨆ (*g* ‘ *A*)
if *x* ∈ (λ*x*. *f x* + *g x*) ‘ *A*
for *x* :: *real*
using *that*
by (*simp add*: *xleq*)
ultimately show *?thesis*
by (*meson bdd-above-def cSup-le-iff*)
qed
have *a2*: *bdd-above* (*L2-set* (*cmod* ∘ *x*) ‘ *Collect finite*)
by (*meson assms*(*1*) *has-ell2-norm-L2-set*)
have *a3*: *bdd-above* (*L2-set* (*cmod* ∘ *y*) ‘ *Collect finite*)
by (*meson assms*(*2*) *has-ell2-norm-L2-set*)
have *a1*: *Collect finite* ≠ {}
by *auto*
have *a4*: ⨆ (*L2-set* (*cmod* ∘ (λ*i*. *x i* + *y i*)) ‘ *Collect finite*)
≤ (*SUP xa*∈*Collect finite*.
L2-set (*cmod* ∘ *x*) *xa* + *L2-set* (*cmod* ∘ *y*) *xa*)
by (*metis* (*mono-tags*, *lifting*) *a1 bdd-plus cSUP-mono mem-Collect-eq triangle*)

have ∀ *r*. ⨆ (*L2-set* (*cmod* ∘ (λ*a*. *x a* + *y a*)) ‘ *Collect finite*) ≤ *r* ∨ ¬ (*SUP A*∈*Collect finite*. *L2-set* (*cmod* ∘ *x*) *A* + *L2-set* (*cmod* ∘ *y*) *A*) ≤ *r*
using *a4* **by** *linarith*
hence ⨆ (*L2-set* (*cmod* ∘ (λ*i*. *x i* + *y i*)) ‘ *Collect finite*)
≤ ⨆ (*L2-set* (*cmod* ∘ *x*) ‘ *Collect finite*) +
⨆ (*L2-set* (*cmod* ∘ *y*) ‘ *Collect finite*)
by (*metis* (*no-types*) *SUP-plus a1 a2 a3*)
hence ⨆ (*L2-set* (*cmod* ∘ (λ*i*. *x i* + *y i*)) ‘ *Collect finite*) ≤ *ell2-norm x* +

ell2-norm y
by (*simp add*: *assms*(*1*) *assms*(*2*) *ell2-norm-L2-set*)
thus *ell2-norm* (λ*i*. *x i* + *y i*) ≤ *ell2-norm x* + *ell2-norm y*
by (*simp add*: *ell2-norm-L2-set has*)
qed

lemma *ell2-norm-uminus*:
assumes *has-ell2-norm x*
shows ‹*has-ell2-norm* (λ*i*. − *x i*)› **and** ‹*ell2-norm* (λ*i*. − *x i*) = *ell2-norm x*›
using *assms* **by** (*auto simp*: *has-ell2-norm-def ell2-norm-def*)

14.2 The type *ell2* of square-summable functions

typedef *'a ell2* = ‹{*f*::*'a*⇒*complex*. *has-ell2-norm f*}›
unfolding *has-ell2-norm-def* **by** (*rule exI*[*of* - λ-.*0*], *auto*)
setup-lifting *type-definition-ell2*

instantiation *ell2* :: (*type*)*complex-vector* **begin**
lift-definition *zero-ell2* :: *'a ell2* **is** λ-. *0* **by** (*auto simp*: *has-ell2-norm-def*)
lift-definition *uminus-ell2* :: *'a ell2* ⇒ *'a ell2* **is** *uminus* **by** (*simp add*: *has-ell2-norm-def*)
lift-definition *plus-ell2* :: ‹*'a ell2* ⇒ *'a ell2* ⇒ *'a ell2*› **is** ‹λ*f g x*. *f x* + *g x*›
by (*rule ell2-norm-triangle*)
lift-definition *minus-ell2* :: *'a ell2* ⇒ *'a ell2* ⇒ *'a ell2* **is** λ*f g x*. *f x* − *g x*
apply (*subst add-uminus-conv-diff*[*symmetric*])
apply (*rule ell2-norm-triangle*)
by (*auto simp add*: *ell2-norm-uminus*)
lift-definition *scaleR-ell2* :: *real* ⇒ *'a ell2* ⇒ *'a ell2* **is** λ*r f x*. *complex-of-real r*
∗ *f x*
by (*rule ell2-norm-smult*)
lift-definition *scaleC-ell2* :: ‹*complex* ⇒ *'a ell2* ⇒ *'a ell2*› **is** ‹λ*c f x*. *c* ∗ *f x*›
by (*rule ell2-norm-smult*)

instance
proof
fix *a b c* :: *'a ell2*

show ((∗$_R$) *r*::*'a ell2* ⇒ -) = (∗$_C$) (*complex-of-real r*) **for** *r*
apply (*rule ext*) **apply** *transfer* **by** *auto*
show *a* + *b* + *c* = *a* + (*b* + *c*)
by (*transfer*; *rule ext*; *simp*)
show *a* + *b* = *b* + *a*
by (*transfer*; *rule ext*; *simp*)
show *0* + *a* = *a*
by (*transfer*; *rule ext*; *simp*)
show − *a* + *a* = *0*
by (*transfer*; *rule ext*; *simp*)
show *a* − *b* = *a* + − *b*
by (*transfer*; *rule ext*; *simp*)
show *r* ∗$_C$ (*a* + *b*) = *r* ∗$_C$ *a* + *r* ∗$_C$ *b* **for** *r*

apply (*transfer*; *rule ext*)
by (*simp add*: *vector-space-over-itself.scale-right-distrib*)
show $(r + r') *_C a = r *_C a + r' *_C a$ **for** $r\ r'$
apply (*transfer*; *rule ext*)
by (*simp add*: *ring-class.ring-distribs*(2))
show $r *_C r' *_C a = (r * r') *_C a$ **for** $r\ r'$
by (*transfer*; *rule ext*; *simp*)
show $1 *_C a = a$
by (*transfer*; *rule ext*; *simp*)
qed
end

instantiation *ell2* :: (*type*) *complex-normed-vector* **begin**
lift-definition *norm-ell2* :: $'a$ *ell2* $\Rightarrow$ *real* **is** *ell2-norm* .
declare *norm-ell2-def*[*code del*]
definition *dist x y* = *norm* $(x - y)$ **for** *x y*::$'a$ *ell2*
definition *sgn x* = *x* $/_R$ *norm x* **for** *x*::$'a$ *ell2*
definition [*code del*]: *uniformity* = (*INF e*∈{*0*<..}. *principal* {(*x*::$'a$ *ell2*, *y*). *norm* $(x - y) < e$})
definition [*code del*]: *open U* = (∀ *x*∈*U*. ∀$_F$ (*x'*, *y*) *in INF e*∈{*0*<..}. *principal* {(*x*, *y*). *norm* $(x - y) < e$}. $x' = x \longrightarrow y \in U$) **for** *U* :: $'a$ *ell2 set*
instance
proof
fix *a b* :: $'a$ *ell2*
show *dist a b* = *norm* $(a - b)$
by (*simp add*: *dist-ell2-def*)
show *sgn a* = *a* $/_R$ *norm a*
by (*simp add*: *sgn-ell2-def*)
show *uniformity* = (*INF e*∈{*0*<..}. *principal* {(*x*, *y*). *dist* (*x*::$'a$ *ell2*) *y* < *e*})
unfolding *dist-ell2-def uniformity-ell2-def* **by** *simp*
show *open U* = (∀ *x*∈*U*. ∀$_F$ (*x'*, *y*) *in uniformity*. (*x'*::$'a$ *ell2*) = *x* $\longrightarrow$ *y* ∈ *U*)
for *U* :: $'a$ *ell2 set*
unfolding *uniformity-ell2-def open-ell2-def* **by** *simp-all*
show (*norm a* = *0*) = (*a* = *0*)
apply *transfer* **by** (*fact ell2-norm-0*)
show *norm* $(a + b) \leq$ *norm a* + *norm b*
apply *transfer* **by** (*fact ell2-norm-triangle*)
show *norm* $(r *_R$ (*a*::$'a$ *ell2*)) = $|r| *$ *norm a* **for** *r*
and *a* :: $'a$ *ell2*
apply *transfer*
by (*simp add*: *ell2-norm-smult*(2))
show *norm* $(r *_C a)$ = *cmod r* * *norm a* **for** *r*
apply *transfer*
by (*simp add*: *ell2-norm-smult*(2))
qed
end

lemma *norm-point-bound-ell2*: *norm* (*Rep-ell2 x i*) $\leq$ *norm x*
apply *transfer*

by (*simp add*: *ell2-norm-point-bound*)

lemma *ell2-norm-finite-support*:
assumes ‹*finite S*› ‹$\bigwedge i.\ i \notin S \Longrightarrow$ *Rep-ell2 x i* $= 0$›
shows ‹*norm x* = *sqrt* ((*sum* ($\lambda i.$ (*cmod* (*Rep-ell2 x i*))^2)) *S*)›
proof (*insert assms*(*2*), *transfer fixing*: *S*)
fix *x* :: ‹$'a \Rightarrow$ *complex*›
assume *zero*: ‹$\bigwedge i.\ i \notin S \Longrightarrow x\ i = 0$›
have ‹*ell2-norm x* = *sqrt* ($\sum_\infty i.$ (*cmod* (*x i*))^2)›
by (*auto simp*: *ell2-norm-def*)
also have ‹$\ldots$ = *sqrt* ($\sum_\infty i{\in}S.$ (*cmod* (*x i*))^2)›
apply (*subst infsum-cong-neutral*[**where** *g*=‹$\lambda i.$ (*cmod* (*x i*))^2› **and** *S*=*UNIV* **and** *T*=*S*])
using *zero* **by** *auto*
also have ‹$\ldots$ = *sqrt* ($\sum i{\in}S.$ (*cmod* (*x i*))^2)›
using ‹*finite S*› **by** *simp*
finally show ‹*ell2-norm x* = *sqrt* ($\sum i{\in}S.$ (*cmod* (*x i*))^2)›
by −
qed

instantiation *ell2* :: (*type*) *complex-inner* **begin**
lift-definition *cinner-ell2* :: ‹$'a$ *ell2* $\Rightarrow$ $'a$ *ell2* $\Rightarrow$ *complex*› **is**
‹$\lambda f\ g.\ \sum_\infty x.$ *cnj* (*f x*) $*$ *g x*› .
declare *cinner-ell2-def*[*code del*]

instance
proof *standard*
fix *x y z* :: $'a$ *ell2* **fix** *c* :: *complex*
show *cinner x y* = *cnj* (*cinner y x*)
proof *transfer*
fix *x y* :: $'a{\Rightarrow}$*complex* **assume** *has-ell2-norm x* **and** *has-ell2-norm y*
have ($\sum_\infty i.$ *cnj* (*x i*) $*$ *y i*) = ($\sum_\infty i.$ *cnj* (*cnj* (*y i*) $*$ *x i*))
by (*metis complex-cnj-cnj complex-cnj-mult mult.commute*)
also have $\ldots$ = *cnj* ($\sum_\infty i.$ *cnj* (*y i*) $*$ *x i*)
by (*metis infsum-cnj*)
finally show ($\sum_\infty i.$ *cnj* (*x i*) $*$ *y i*) = *cnj* ($\sum_\infty i.$ *cnj* (*y i*) $*$ *x i*) .
qed

show *cinner* (*x* + *y*) *z* = *cinner x z* + *cinner y z*
proof *transfer*
fix *x y z* :: $'a \Rightarrow$ *complex*
assume *has-ell2-norm x*
hence *cnj-x*: ($\lambda i.$ *cnj* (*x i*) $*$ *cnj* (*x i*)) *abs-summable-on UNIV*
by (*simp del*: *complex-cnj-mult add*: *norm-mult*[*symmetric*] *complex-cnj-mult*[*symmetric*] *has-ell2-norm-def power2-eq-square*)
assume *has-ell2-norm y*
hence *cnj-y*: ($\lambda i.$ *cnj* (*y i*) $*$ *cnj* (*y i*)) *abs-summable-on UNIV*
by (*simp del*: *complex-cnj-mult add*: *norm-mult*[*symmetric*] *complex-cnj-mult*[*symmetric*] *has-ell2-norm-def power2-eq-square*)

assume *has-ell2-norm z*
hence *z*: $(\lambda i.\ z\ i * z\ i)$ *abs-summable-on UNIV*
by (*simp add*: *norm-mult*[*symmetric*] *has-ell2-norm-def power2-eq-square*)
have *cnj-x-z*:$(\lambda i.\ cnj\ (x\ i) * z\ i)$ *abs-summable-on UNIV*
using *cnj-x z* **by** (*rule abs-summable-product*)
have *cnj-y-z*:$(\lambda i.\ cnj\ (y\ i) * z\ i)$ *abs-summable-on UNIV*
using *cnj-y z* **by** (*rule abs-summable-product*)
show $(\sum_\infty i.\ cnj\ (x\ i + y\ i) * z\ i) = (\sum_\infty i.\ cnj\ (x\ i) * z\ i) + (\sum_\infty i.\ cnj\ (y\ i) * z\ i)$
apply (*subst infsum-add* [*symmetric*])
using *cnj-x-z cnj-y-z*
by (*auto simp add*: *summable-on-iff-abs-summable-on-complex distrib-left mult.commute*)
qed

show $cinner\ (c *_C x)\ y = cnj\ c * cinner\ x\ y$
proof *transfer*
fix $x\ y :: 'a \Rightarrow complex$ **and** $c :: complex$
assume *has-ell2-norm x*
hence *cnj-x*: $(\lambda i.\ cnj\ (x\ i) * cnj\ (x\ i))$ *abs-summable-on UNIV*
by (*simp del*: *complex-cnj-mult add*: *norm-mult*[*symmetric*] *complex-cnj-mult*[*symmetric*] *has-ell2-norm-def power2-eq-square*)
assume *has-ell2-norm y*
hence *y*: $(\lambda i.\ y\ i * y\ i)$ *abs-summable-on UNIV*
by (*simp add*: *norm-mult*[*symmetric*] *has-ell2-norm-def power2-eq-square*)
have *cnj-x-y*:$(\lambda i.\ cnj\ (x\ i) * y\ i)$ *abs-summable-on UNIV*
using *cnj-x y* **by** (*rule abs-summable-product*)
thus $(\sum_\infty i.\ cnj\ (c * x\ i) * y\ i) = cnj\ c * (\sum_\infty i.\ cnj\ (x\ i) * y\ i)$
by (*auto simp flip*: *infsum-cmult-right simp add*: *abs-summable-summable mult.commute vector-space-over-itself.scale-left-commute*)
qed

show $0 \le cinner\ x\ x$
proof *transfer*
fix $x :: 'a \Rightarrow complex$
assume *has-ell2-norm x*
hence $(\lambda i.\ cmod\ (cnj\ (x\ i) * x\ i))$ *abs-summable-on UNIV*
by (*simp add*: *norm-mult has-ell2-norm-def power2-eq-square*)
hence $(\lambda i.\ cnj\ (x\ i) * x\ i)$ *abs-summable-on UNIV*
by *auto*
hence *sum*: $(\lambda i.\ cnj\ (x\ i) * x\ i)$ *abs-summable-on UNIV*
unfolding *has-ell2-norm-def power2-eq-square*.
have $0 = (\sum_\infty i::'a.\ 0)$ **by** *auto*
also have $\ldots \le (\sum_\infty i.\ cnj\ (x\ i) * x\ i)$
apply (*rule infsum-mono-complex*)
by (*auto simp add*: *abs-summable-summable sum*)
finally show $0 \le (\sum_\infty i.\ cnj\ (x\ i) * x\ i)$ **by** *assumption*
qed

```
  show (cinner x x = 0) = (x = 0)
  proof (transfer, auto)
    fix x :: 'a ⇒ complex
    assume has-ell2-norm x
    hence (λi::'a. cmod (cnj (x i) * x i)) abs-summable-on UNIV
     by (smt (verit, del-insts) complex-mod-mult-cnj has-ell2-norm-def mult.commute
norm-ge-zero norm-power real-norm-def summable-on-cong)
    hence cmod-x2: (λi. cnj (x i) * x i) abs-summable-on UNIV
      unfolding has-ell2-norm-def power2-eq-square
      by simp
    assume eq0: (∑∞i. cnj (x i) * x i) = 0
    show x = (λ-. 0)
    proof (rule ccontr)
      assume x ≠ (λ-. 0)
      then obtain i where x i ≠ 0 by auto
      hence 0 < cnj (x i) * x i
      by (metis le-less cnj-x-x-geq0 complex-cnj-zero-iff vector-space-over-itself.scale-eq-0-iff)
      also have ... = (∑∞i∈{i}. cnj (x i) * x i) by auto
      also have ... ≤ (∑∞i. cnj (x i) * x i)
        apply (rule infsum-mono-neutral-complex)
        by (auto simp add: abs-summable-summable cmod-x2)
      also from eq0 have ... = 0 by assumption
      finally show False by simp
    qed
  qed

  show norm x = sqrt (cmod (cinner x x))
  proof transfer
    fix x :: 'a ⇒ complex
    assume x: has-ell2-norm x
    have (λi::'a. cmod (x i) * cmod (x i)) abs-summable-on UNIV ⟹
    (λi::'a. cmod (cnj (x i) * x i)) abs-summable-on UNIV
      by (simp add: norm-mult has-ell2-norm-def power2-eq-square)
    hence sum: (λi. cnj (x i) * x i) abs-summable-on UNIV
     by (metis (no-types, lifting) complex-mod-mult-cnj has-ell2-norm-def mult.commute
norm-power summable-on-cong x)
    from x have ell2-norm x = sqrt (∑∞i. (cmod (x i))²)
      unfolding ell2-norm-def by simp
    also have ... = sqrt (∑∞i. cmod (cnj (x i) * x i))
      unfolding norm-complex-def power2-eq-square by auto
    also have ... = sqrt (cmod (∑∞i. cnj (x i) * x i))
      by (auto simp: infsum-cmod abs-summable-summable sum)
   finally show ell2-norm x = sqrt (cmod (∑∞i. cnj (x i) * x i)) by assumption
  qed
qed
end

instance ell2 :: (type) chilbert-space
proof intro-classes
```

fix $X :: \langle nat \Rightarrow 'a\ ell2 \rangle$
define x **where** $\langle x\ n\ a = Rep\text{-}ell2\ (X\ n)\ a \rangle$ **for** $n\ a$
have [*simp*]: $\langle has\text{-}ell2\text{-}norm\ (x\ n) \rangle$ **for** n
using *Rep-ell2 x-def*[*abs-def*] **by** *simp*

assume $\langle Cauchy\ X \rangle$
moreover have $dist\ (x\ n\ a)\ (x\ m\ a) \leq dist\ (X\ n)\ (X\ m)$ **for** $n\ m\ a$
by (*metis Rep-ell2 x-def dist-norm ell2-norm-point-bound mem-Collect-eq minus-ell2.rep-eq norm-ell2.rep-eq*)
ultimately have $\langle Cauchy\ (\lambda n.\ x\ n\ a) \rangle$ **for** a
by (*meson Cauchy-def le-less-trans*)
then obtain l **where** *x-lim*: $\langle (\lambda n.\ x\ n\ a) \longrightarrow l\ a \rangle$ **for** a
apply *atomize-elim* **apply** (*rule choice*)
by (*simp add*: *convergent-eq-Cauchy*)
define L **where** $\langle L = Abs\text{-}ell2\ l \rangle$
define *normF* **where** $\langle normF\ F\ x = L2\text{-}set\ (cmod \circ x)\ F \rangle$ **for** $F :: \langle 'a\ set \rangle$ **and** x
have *normF-triangle*: $\langle normF\ F\ (\lambda a.\ x\ a + y\ a) \leq normF\ F\ x + normF\ F\ y \rangle$ **if** $\langle finite\ F \rangle$ **for** $F\ x\ y$
proof −
have $\langle normF\ F\ (\lambda a.\ x\ a + y\ a) = L2\text{-}set\ (\lambda a.\ cmod\ (x\ a + y\ a))\ F \rangle$
by (*metis* (*mono-tags*, *lifting*) *L2-set-cong comp-apply normF-def*)
also have $\langle \ldots \leq L2\text{-}set\ (\lambda a.\ cmod\ (x\ a) + cmod\ (y\ a))\ F \rangle$
by (*meson L2-set-mono norm-ge-zero norm-triangle-ineq*)
also have $\langle \ldots \leq L2\text{-}set\ (\lambda a.\ cmod\ (x\ a))\ F + L2\text{-}set\ (\lambda a.\ cmod\ (y\ a))\ F \rangle$
by (*simp add*: *L2-set-triangle-ineq*)
also have $\langle \ldots \leq normF\ F\ x + normF\ F\ y \rangle$
by (*smt* (*verit*, *best*) *L2-set-cong normF-def comp-apply*)
finally show *?thesis*
by −
qed
have *normF-negate*: $\langle normF\ F\ (\lambda a.\ -\ x\ a) = normF\ F\ x \rangle$ **if** $\langle finite\ F \rangle$ **for** $F\ x$
unfolding *normF-def o-def* **by** *simp*
have *normF-ell2norm*: $\langle normF\ F\ x \leq ell2\text{-}norm\ x \rangle$ **if** $\langle finite\ F \rangle$ **and** $\langle has\text{-}ell2\text{-}norm\ x \rangle$ **for** $F\ x$
apply (*auto intro*!: *cSUP-upper2*[**where** $x{=}F$] *simp*: *that normF-def ell2-norm-L2-set*)
by (*meson has-ell2-norm-L2-set that*(*2*))

note *Lim-bounded2*[*rotated*, *rule-format*, *trans*]

from $\langle Cauchy\ X \rangle$
obtain I **where** *cauchyX*: $\langle norm\ (X\ n - X\ m) \leq \varepsilon \rangle$ **if** $\langle \varepsilon{>}0 \rangle\ \langle n{\geq}I\ \varepsilon \rangle\ \langle m{\geq}I\ \varepsilon \rangle$ **for** $\varepsilon\ n\ m$
by (*metis Cauchy-def dist-norm less-eq-real-def*)
have *normF-xx*: $\langle normF\ F\ (\lambda a.\ x\ n\ a - x\ m\ a) \leq \varepsilon \rangle$ **if** $\langle finite\ F \rangle\ \langle \varepsilon{>}0 \rangle\ \langle n{\geq}I\ \varepsilon \rangle\ \langle m{\geq}I\ \varepsilon \rangle$ **for** $\varepsilon\ n\ m\ F$
apply (*subst asm-rl*[*of* $\langle (\lambda a.\ x\ n\ a - x\ m\ a) = Rep\text{-}ell2\ (X\ n - X\ m) \rangle$])
apply (*simp add*: *x-def minus-ell2.rep-eq*)
using *that cauchyX* **by** (*metis Rep-ell2 mem-Collect-eq normF-ell2norm norm-ell2.rep-eq*

order-trans)
have *normF-xl-lim*: ‹(λm. *normF F* (λa. *x m a* − *l a*)) ⟶ *0*› **if** ‹*finite F*› **for** *F*
proof −
have ‹(λxa. *cmod* (*x xa m* − *l m*)) ⟶ *0*› **for** *m*
using *x-lim* **by** (*simp add*: *LIM-zero-iff tendsto-norm-zero*)
then have ‹(λm. ∑ *i*∈*F*. ((*cmod* ∘ (λa. *x m a* − *l a*)) *i*)²) ⟶ *0*›
by (*auto intro*: *tendsto-null-sum*)
then show *?thesis*
unfolding *normF-def L2-set-def*
using *tendsto-real-sqrt* **by** *force*
qed
have *normF-xl*: ‹*normF F* (λa. *x n a* − *l a*) ≤ *ε*›
if ‹*n* ≥ *I ε*› **and** ‹*ε* > *0*› **and** ‹*finite F*› **for** *n ε F*
proof −
have ‹*normF F* (λa. *x n a* − *l a*) − *ε* ≤ *normF F* (λa. *x n a* − *x m a*) + *normF F* (λa. *x m a* − *l a*) − *ε*› **for** *m*
using *normF-triangle*[*OF* ‹*finite F*›, **where** *x*=‹(λa. *x n a* − *x m a*)› **and** *y*=‹(λa. *x m a* − *l a*)›]
by *auto*
also have ‹... *m* ≤ *normF F* (λa. *x m a* − *l a*)› **if** ‹*m* ≥ *I ε*› **for** *m*
using *normF-xx*[*OF* ‹*finite F*› ‹*ε*>*0*› ‹*n* ≥ *I ε*› ‹*m* ≥ *I ε*›]
by *auto*
also have ‹(λm. ... *m*) ⟶ *0*›
using ‹*finite F*› **by** (*rule normF-xl-lim*)
finally show *?thesis*
by *auto*
qed
have ‹*normF F l* ≤ *1* + *normF F* (*x* (*I 1*))› **if** [*simp*]: ‹*finite F*› **for** *F*
using *normF-xl*[**where** *F*=*F* **and** *ε*=*1* **and** *n*=‹*I 1*›]
using *normF-triangle*[**where** *F*=*F* **and** *x*=‹*x* (*I 1*)› **and** *y*=‹λa. *l a* − *x* (*I 1*) *a*›]
using *normF-negate*[**where** *F*=*F* **and** *x*=‹(λa. *x* (*I 1*) *a* − *l a*)›]
by *auto*
also have ‹... *F* ≤ *1* + *ell2-norm* (*x* (*I 1*))› **if** ‹*finite F*› **for** *F*
using *normF-ell2norm that* **by** *simp*
finally have [*simp*]: ‹*has-ell2-norm l*›
unfolding *has-ell2-norm-L2-set*
by (*auto intro*!: *bdd-aboveI simp flip*: *normF-def*)
then have ‹*l* = *Rep-ell2 L*›
by (*simp add*: *Abs-ell2-inverse L-def*)
have [*simp*]: ‹*has-ell2-norm* (λa. *x n a* − *l a*)› **for** *n*
apply (*subst diff-conv-add-uminus*)
apply (*rule ell2-norm-triangle*)
by (*auto intro*!: *ell2-norm-uminus*)
from *normF-xl* **have** *ell2norm-xl*: ‹*ell2-norm* (λa. *x n a* − *l a*) ≤ *ε*›
if ‹*n* ≥ *I ε*› **and** ‹*ε* > *0*› **for** *n ε*
apply (*subst ell2-norm-L2-set*)
using *that* **by** (*auto intro*!: *cSUP-least simp*: *normF-def*)

have ‹*norm* $(X\ n - L) \leq \varepsilon$› **if** ‹$n \geq I\ \varepsilon$› **and** ‹$\varepsilon > 0$› **for** $n\ \varepsilon$
using *ell2norm-xl*[*OF that*]
by (*simp add*: *x-def norm-ell2.rep-eq* ‹l = *Rep-ell2 L*› *minus-ell2.rep-eq*)
then have ‹$X \longrightarrow L$›
unfolding *tendsto-iff*
apply (*auto simp*: *dist-norm eventually-sequentially*)
by (*meson field-lbound-gt-zero le-less-trans*)
then show ‹*convergent X*›
by (*rule convergentI*)
qed

lemma *sum-ell2-transfer*[*transfer-rule*]:
includes *lifting-syntax*
shows ‹((=) ===> *pcr-ell2* (=)) ===> *rel-set* (=) ===> *pcr-ell2* (=))
(λ*f X x*. *sum* (λ*y*. *f y x*) *X*) *sum*›
proof (*intro rel-funI*, *rename-tac f f' X X'*)
fix *f* **and** *f'* :: ‹*'a* ⇒ *'b ell2*›
assume [*transfer-rule*]: ‹((=) ===> *pcr-ell2* (=)) *f f'*›
fix *X X'* :: ‹*'a set*›
assume ‹*rel-set* (=) *X X'*›
then have [*simp*]: ‹$X' = X$›
by (*simp add*: *rel-set-eq*)
show ‹*pcr-ell2* (=) (λ*x*. $\sum y{\in}X$. *f y x*) (*sum f' X'*)›
unfolding ‹$X' = X$›
proof (*induction X rule*: *infinite-finite-induct*)
case (*infinite X*)
show *?case*
apply (*simp add*: *infinite*)
by *transfer-prover*
next
case *empty*
show *?case*
apply (*simp add*: *empty*)
by *transfer-prover*
next
case (*insert x F*)
note [*transfer-rule*] = *insert.IH*
show *?case*
apply (*simp add*: *insert*)
by *transfer-prover*
qed
qed

lemma *clinear-Rep-ell2*[*simp*]: ‹*clinear* (λ*ψ*. *Rep-ell2 ψ i*)›
by (*simp add*: *clinearI plus-ell2.rep-eq scaleC-ell2.rep-eq*)

lemma *Abs-ell2-inverse-finite*[*simp*]: ‹*Rep-ell2* (*Abs-ell2 ψ*) = *ψ*› **for** *ψ* :: ‹-::*finite*
⇒ *complex*›
by (*simp add*: *Abs-ell2-inverse*)

14.3 Orthogonality

lemma *ell2-pointwise-ortho*:
assumes ‹$\bigwedge i.$ *Rep-ell2 x i* $= 0 \vee$ *Rep-ell2 y i* $= 0$›
shows ‹*is-orthogonal x y*›
using *assms* **apply** *transfer*
by (*simp add*: *infsum-0*)

14.4 Truncated vectors

lift-definition *trunc-ell2*:: ‹$'a$ *set* $\Rightarrow$ $'a$ *ell2* $\Rightarrow$ $'a$ *ell2*›
is ‹λ *S x*. (λ *i*. (*if i* $\in$ *S then x i else 0*))›
proof (*rename-tac S x*)
fix *x* :: ‹$'a \Rightarrow$ *complex*› **and** *S* :: ‹$'a$ *set*›
assume ‹*has-ell2-norm x*›
then have ‹($\lambda i.$ $(x\ i)^2$) *abs-summable-on UNIV*›
unfolding *has-ell2-norm-def* **by** −
then have ‹($\lambda i.$ $(x\ i)^2$) *abs-summable-on S*›
using *summable-on-subset-banach* **by** *blast*
then have ‹($\lambda xa.$ (*if xa* $\in$ *S then x xa else 0*)^2) *abs-summable-on UNIV*›
apply (*rule summable-on-cong-neutral*[*THEN iffD1*, *rotated* −*1*])
by *auto*
then show ‹*has-ell2-norm* ($\lambda i.$ *if i* $\in$ *S then x i else 0*)›
unfolding *has-ell2-norm-def* **by** −
qed

lemma *trunc-ell2-empty*[*simp*]: ‹*trunc-ell2* {} *x* $= 0$›
apply *transfer* **by** *simp*

lemma *trunc-ell2-UNIV*[*simp*]: ‹*trunc-ell2 UNIV* $\psi = \psi$›
apply *transfer* **by** *simp*

lemma *norm-id-minus-trunc-ell2*:
‹(*norm* (*x* − *trunc-ell2 S x*))^2 = (*norm x*)^2 − (*norm* (*trunc-ell2 S x*))^2›
proof−
have ‹*Rep-ell2* (*trunc-ell2 S x*) *i* $= 0 \vee$ *Rep-ell2* (*x* − *trunc-ell2 S x*) *i* $= 0$›
for *i*
apply *transfer*
by *auto*
hence ‹((*trunc-ell2 S x*) $\cdot_C$ (*x* − *trunc-ell2 S x*)) $= 0$›
using *ell2-pointwise-ortho* **by** *blast*
hence ‹(*norm x*)^2 = (*norm* (*trunc-ell2 S x*))^2 + (*norm* (*x* − *trunc-ell2 S x*))^2›
using *pythagorean-theorem* **by** *fastforce*
thus *?thesis* **by** *simp*
qed

lemma *norm-trunc-ell2-finite*:
‹*finite S* $\Longrightarrow$ (*norm* (*trunc-ell2 S x*)) = *sqrt* ((*sum* ($\lambda i.$ (*cmod* (*Rep-ell2 x i*))^2)) *S*)›

proof−
assume ‹*finite S*›
moreover have ‹⋀ *i*. *i* ∉ *S* ⟹ *Rep-ell2* ((*trunc-ell2 S x*)) *i* = *0*›
by (*simp add*: *trunc-ell2.rep-eq*)
ultimately have ‹(*norm* (*trunc-ell2 S x*)) = *sqrt* ((*sum* (λ*i*. (*cmod* (*Rep-ell2* ((*trunc-ell2 S x*)) *i*))^2)) *S*)›
using *ell2-norm-finite-support*
by *blast*
moreover have ‹⋀ *i*. *i* ∈ *S* ⟹ *Rep-ell2* ((*trunc-ell2 S x*)) *i* = *Rep-ell2 x i*›
by (*simp add*: *trunc-ell2.rep-eq*)
ultimately show *?thesis* **by** *simp*
qed

lemma *trunc-ell2-lim-at-UNIV*:
‹((λ*S*. *trunc-ell2 S ψ*) ⟶ *ψ*) (*finite-subsets-at-top UNIV*)›
proof −
define *f* **where** ‹*f i* = (*cmod* (*Rep-ell2 ψ i*))^2› **for** *i*

have *has*: ‹*has-ell2-norm* (*Rep-ell2 ψ*)›
using *Rep-ell2* **by** *blast*
then have *summable*: *f abs-summable-on UNIV*
by (*smt* (*verit*, *del-insts*) *f-def has-ell2-norm-def norm-ge-zero norm-power real-norm-def summable-on-cong*)

have ‹*norm ψ* = (*ell2-norm* (*Rep-ell2 ψ*))›
apply *transfer* **by** *simp*
also have ‹... = *sqrt* (*infsum f UNIV*)›
by (*simp add*: *ell2-norm-def f-def*[*symmetric*])
finally have *normψ*: ‹*norm ψ* = *sqrt* (*infsum f UNIV*)›
by −

have *norm-trunc*: ‹*norm* (*trunc-ell2 S ψ*) = *sqrt* (*sum f S*)› **if** ‹*finite S*› **for** *S*
using *f-def that norm-trunc-ell2-finite* **by** *fastforce*

have ‹(*sum f* ⟶ *infsum f UNIV*) (*finite-subsets-at-top UNIV*)›
using *f-def*[*abs-def*] *infsum-tendsto local.summable* **by** *fastforce*
then have ‹((λ*S*. *sqrt* (*sum f S*)) ⟶ *sqrt* (*infsum f UNIV*)) (*finite-subsets-at-top UNIV*)›
using *tendsto-real-sqrt* **by** *blast*
then have ‹((λ*S*. *norm* (*trunc-ell2 S ψ*)) ⟶ *norm ψ*) (*finite-subsets-at-top UNIV*)›
apply (*subst tendsto-cong*[**where** *g*=‹λ*S*. *sqrt* (*sum f S*)›])
by (*auto simp add*: *eventually-finite-subsets-at-top-weakI norm-trunc normψ*)
then have ‹((λ*S*. (*norm* (*trunc-ell2 S ψ*))^2) ⟶ (*norm ψ*)^2) (*finite-subsets-at-top UNIV*)›
by (*simp add*: *tendsto-power*)
then have ‹((λ*S*. (*norm ψ*)^2 − (*norm* (*trunc-ell2 S ψ*))^2) ⟶ *0*) (*finite-subsets-at-top UNIV*)›
apply (*rule tendsto-diff*[**where** *a*=‹(*norm ψ*)^2› **and** *b*=‹(*norm ψ*)^2›, *sim-*

plified, *rotated*])
by *auto*
then have ‹((λ*S*. (*norm* (ψ − *trunc-ell2 S* ψ))²) ⟶ 0) (*finite-subsets-at-top UNIV*)›
unfolding *norm-id-minus-trunc-ell2* **by** *simp*
then have ‹((λ*S*. *norm* (ψ − *trunc-ell2 S* ψ)) ⟶ 0) (*finite-subsets-at-top UNIV*)›
by *auto*
then have ‹((λ*S*. ψ − *trunc-ell2 S* ψ) ⟶ 0) (*finite-subsets-at-top UNIV*)›
by (*rule tendsto-norm-zero-cancel*)
then show *?thesis*
apply (*rule Lim-transform2*[**where** *f*=‹λ-. ψ›, *rotated*])
by *simp*
qed

lemma *trunc-ell2-lim-seq*: ‹((λ*n*. *trunc-ell2* {..<*n*} ψ) ⟶ ψ)›
using *trunc-ell2-lim-at-UNIV filterlim-lessThan-at-top*
by (*rule filterlim-compose*)

lemma *trunc-ell2-norm-mono*: ‹*M* ⊆ *N* ⟹ *norm* (*trunc-ell2 M* ψ) ≤ *norm* (*trunc-ell2 N* ψ)›
proof (*rule power2-le-imp-le*[*rotated*], *force*, *transfer*)
fix *M N* :: ‹'*a set*› **and** ψ :: ‹'*a* ⇒ *complex*›
assume ‹*M* ⊆ *N*› **and** ‹*has-ell2-norm* ψ›
have ‹(*ell2-norm* (λ*i*. *if i* ∈ *M then* ψ *i else* 0))² = (∑∞*i*∈*M*. (*cmod* (ψ *i*))²)›
unfolding *ell2-norm-square*
apply (*rule infsum-cong-neutral*)
by *auto*
also have ‹... ≤ (∑∞*i*∈*N*. (*cmod* (ψ *i*))²)›
apply (*rule infsum-mono2*)
using ‹*has-ell2-norm* ψ› ‹*M* ⊆ *N*›
by (*auto simp add*: *ell2-norm-square has-ell2-norm-def simp flip*: *norm-power intro*: *summable-on-subset-banach*)
also have ‹... = (*ell2-norm* (λ*i*. *if i* ∈ *N then* ψ *i else* 0))²›
unfolding *ell2-norm-square*
apply (*rule infsum-cong-neutral*)
by *auto*
finally show ‹(*ell2-norm* (λ*i*. *if i* ∈ *M then* ψ *i else* 0))² ≤ (*ell2-norm* (λ*i*. *if i* ∈ *N then* ψ *i else* 0))²›
by −
qed

lemma *trunc-ell2-reduces-norm*: ‹*norm* (*trunc-ell2 M* ψ) ≤ *norm* ψ›
by (*metis subset-UNIV trunc-ell2-UNIV trunc-ell2-norm-mono*)

lemma *trunc-ell2-twice*[*simp*]: ‹*trunc-ell2 M* (*trunc-ell2 N* ψ) = *trunc-ell2* (*M*∩*N*) ψ›
apply *transfer* **by** *auto*

lemma *trunc-ell2-union*: ‹*trunc-ell2* $(M \cup N)$ ψ = *trunc-ell2* M ψ + *trunc-ell2* N ψ − *trunc-ell2* $(M \cap N)$ ψ›
apply *transfer* **by** *auto*

lemma *trunc-ell2-union-disjoint*: ‹$M \cap N = \{\} \Longrightarrow$ *trunc-ell2* $(M \cup N)$ ψ = *trunc-ell2* M ψ + *trunc-ell2* N ψ›
by (*simp add*: *trunc-ell2-union*)

lemma *trunc-ell2-union-Diff*: ‹$M \subseteq N \Longrightarrow$ *trunc-ell2* $(N-M)$ ψ = *trunc-ell2* N ψ − *trunc-ell2* M ψ›
using *trunc-ell2-union-disjoint*[**where** M=‹$N-M$› **and** N=M **and** ψ=ψ]
by (*simp add*: *Un-commute inf.commute le-iff-sup*)

lemma *trunc-ell2-add*: ‹*trunc-ell2* M $(\psi + \varphi)$ = *trunc-ell2* M ψ + *trunc-ell2* M φ›
apply *transfer* **by** *auto*

lemma *trunc-ell2-scaleC*: ‹*trunc-ell2* M $(c *_C \psi)$ = $c *_C$ *trunc-ell2* M ψ›
apply *transfer* **by** *auto*

lemma *bounded-clinear-trunc-ell2*[*bounded-clinear*]: ‹*bounded-clinear* (*trunc-ell2* M)›
by (*auto intro*!: *bounded-clinearI*[**where** K=*1*] *trunc-ell2-reduces-norm*
simp: *trunc-ell2-add trunc-ell2-scaleC*)

lemma *trunc-ell2-lim*: ‹$((\lambda S.$ *trunc-ell2* S $\psi) \longrightarrow$ *trunc-ell2* M $\psi)$ (*finite-subsets-at-top* M)›
proof −
have ‹$((\lambda S.$ *trunc-ell2* S (*trunc-ell2* M $\psi)) \longrightarrow$ *trunc-ell2* M $\psi)$ (*finite-subsets-at-top* *UNIV*)›
using *trunc-ell2-lim-at-UNIV* **by** *blast*
then have ‹$((\lambda S.$ *trunc-ell2* $(S \cap M)$ $\psi) \longrightarrow$ *trunc-ell2* M $\psi)$ (*finite-subsets-at-top* *UNIV*)›
by *simp*
then show ‹$((\lambda S.$ *trunc-ell2* S $\psi) \longrightarrow$ *trunc-ell2* M $\psi)$ (*finite-subsets-at-top* M)›
unfolding *filterlim-def*
apply (*subst* (*asm*) *filtermap-filtermap*[**where** g=‹$\lambda S.\ S \cap M$›, *symmetric*])
apply (*subst* (*asm*) *finite-subsets-at-top-inter*[**where** A=M **and** B=*UNIV*])
by *auto*
qed

lemma *trunc-ell2-lim-general*:
assumes *big*: ‹$\bigwedge G.$ *finite* $G \Longrightarrow G \subseteq M \Longrightarrow (\forall_F\ H$ *in* $F.\ H \supseteq G)$›
assumes *small*: ‹$\forall_F\ H$ *in* $F.\ H \subseteq M$›
shows ‹$((\lambda S.$ *trunc-ell2* S $\psi) \longrightarrow$ *trunc-ell2* M $\psi)$ F›
proof (*rule tendstoI*)
fix e :: *real* **assume** ‹$e > 0$›
from *trunc-ell2-lim*[*THEN tendsto-iff*[*THEN iffD1*], *rule-format*, *OF* ‹$e > 0$›, **where** M=M **and** ψ=ψ]

```
  obtain G where ‹finite G› and ‹G ⊆ M› and
    close: ‹dist (trunc-ell2 G ψ) (trunc-ell2 M ψ) < e›
    apply atomize-elim
    unfolding eventually-finite-subsets-at-top
    by blast
  from ‹finite G› ‹G ⊆ M› and big
  have ‹∀⇩F H in F. H ⊇ G›
    by −
  with small have ‹∀⇩F H in F. H ⊆ M ∧ H ⊇ G›
    by (simp add: eventually-conj-iff)
  then show ‹∀⇩F H in F. dist (trunc-ell2 H ψ) (trunc-ell2 M ψ) < e›
  proof (rule eventually-mono)
    fix H assume GHM: ‹H ⊆ M ∧ H ⊇ G›
    have ‹dist (trunc-ell2 H ψ) (trunc-ell2 M ψ) = norm (trunc-ell2 (M−H) ψ)›
     by (simp add: GHM dist-ell2-def norm-minus-commute trunc-ell2-union-Diff)
    also have ‹... ≤ norm (trunc-ell2 (M−G) ψ)›
      by (simp add: Diff-mono GHM trunc-ell2-norm-mono)
    also have ‹...  = dist (trunc-ell2 G ψ) (trunc-ell2 M ψ)›
    by (simp add: ‹G ⊆ M› dist-ell2-def norm-minus-commute trunc-ell2-union-Diff)
    also have ‹... < e›
      using close by simp
    finally show ‹dist (trunc-ell2 H ψ) (trunc-ell2 M ψ) < e›
      by −
  qed
qed

lemma norm-ell2-bound-trunc:
  assumes ‹⋀M. finite M ⟹ norm (trunc-ell2 M ψ) ≤ B›
  shows ‹norm ψ ≤ B›
proof −
  note trunc-ell2-lim-at-UNIV[of ψ]
  then have ‹((λS. norm (trunc-ell2 S ψ)) ⟶ norm ψ) (finite-subsets-at-top
UNIV)›
    using tendsto-norm by auto
  then show ‹norm ψ ≤ B›
    apply (rule tendsto-upperbound)
    using assms apply (rule eventually-finite-subsets-at-top-weakI)
    by auto
qed

lemma trunc-ell2-uminus: ‹trunc-ell2 (−M) ψ = ψ − trunc-ell2 M ψ›
 by (metis Int-UNIV-left boolean-algebra-class.diff-eq subset-UNIV trunc-ell2-UNIV
trunc-ell2-union-Diff)
```

14.5 Kets and bras

```
lift-definition ket :: ‹'a ⇒ 'a ell2› is ‹λx y. of-bool (x=y)›
  by (rule has-ell2-norm-ket)
```

abbreviation *bra* :: *'a* ⇒ (-,*complex*) *cblinfun* **where** *bra i* ≡ *vector-to-cblinfun* (*ket i*)* **for** *i*

instance *ell2* :: (*type*) *not-singleton*
proof *standard*
  have *ket undefined* ≠ (*0*::*'a ell2*)
  proof *transfer*
    show ‹(λ*y*. *of-bool* ((*undefined*::*'a*) = *y*)) ≠ (λ-. *0*)›
      by (*metis* (*mono-tags*) *of-bool-eq*(*2*) *zero-neq-one*)
  qed
  thus ‹∃ *x y*::*'a ell2*. *x* ≠ *y*›
    by *blast*
qed

lemma *cinner-ket-left*: ‹*ket i* ·$_C$ *ψ* = *Rep-ell2 ψ i*›
  apply (*transfer fixing*: *i*)
  apply (*subst infsum-cong-neutral*[**where** *T*=‹{*i*}›])
  by *auto*

lemma *cinner-ket-right*: ‹(*ψ* ·$_C$ *ket i*) = *cnj* (*Rep-ell2 ψ i*)›
  apply (*transfer fixing*: *i*)
  apply (*subst infsum-cong-neutral*[**where** *T*=‹{*i*}›])
  by *auto*

lemma *bounded-clinear-Rep-ell2*[*simp*, *bounded-clinear*]: ‹*bounded-clinear* (λ*ψ*. *Rep-ell2 ψ x*)›
  apply (*subst asm-rl*[*of* ‹(λ*ψ*. *Rep-ell2 ψ x*) = (λ*ψ*. *ket x* ·$_C$ *ψ*)›])
   apply (*auto simp*: *cinner-ket-left*)
  by (*simp add*: *bounded-clinear-cinner-right*)

lemma *cinner-ket-eqI*:
  assumes ‹⋀*i*. *ket i* ·$_C$ *ψ* = *ket i* ·$_C$ *φ*›
  shows ‹*ψ* = *φ*›
  by (*metis Rep-ell2-inject assms cinner-ket-left ext*)

lemma *norm-ket*[*simp*]: *norm* (*ket i*) = *1*
  apply *transfer* **by** (*rule ell2-norm-ket*)

lemma *cinner-ket-same*[*simp*]:
  ‹(*ket i* ·$_C$ *ket i*) = *1*›
proof−
  have ‹*norm* (*ket i*) = *1*›
    by *simp*
  hence ‹*sqrt* (*cmod* (*ket i* ·$_C$ *ket i*)) = *1*›
    by (*metis norm-eq-sqrt-cinner*)
  hence ‹*cmod* (*ket i* ·$_C$ *ket i*) = *1*›
    using *real-sqrt-eq-1-iff* **by** *blast*
  moreover have ‹(*ket i* ·$_C$ *ket i*) = *cmod* (*ket i* ·$_C$ *ket i*)›

```
  proof−
    have ‹(ket i •C ket i) ∈ ℝ›
      by (simp add: cinner-real)
    thus ?thesis
      by (metis ‹norm (ket i) = 1› cnorm-eq norm-one of-real-1 one-cinner-one)
  qed
  ultimately show ?thesis by simp
qed

lemma orthogonal-ket[simp]:
  ‹is-orthogonal (ket i) (ket j) ⟷ i ≠ j›
  by (simp add: cinner-ket-left ket.rep-eq of-bool-def)

lemma cinner-ket: ‹(ket i •C ket j) = of-bool (i=j)›
  by (simp add: cinner-ket-left ket.rep-eq)

lemma ket-injective[simp]: ‹ket i = ket j ⟷ i = j›
  by (metis cinner-ket one-neq-zero of-bool-def)

lemma inj-ket[simp]: ‹inj-on ket M›
  by (simp add: inj-on-def)

lemma trunc-ell2-ket-cspan:
  ‹trunc-ell2 S x ∈ cspan (range ket)› if ‹finite S›
proof (use that in induction)
  case empty
  then show ?case
    by (auto intro: complex-vector.span-zero)
next
  case (insert a F)
  from insert.hyps have ‹trunc-ell2 (insert a F) x = trunc-ell2 F x + Rep-ell2 x
a *C ket a›
    apply (transfer fixing: F a)
    by auto
  with insert.IH
  show ?case
   by (simp add: complex-vector.span-add-eq complex-vector.span-base complex-vector.span-scale)
qed

lemma closed-cspan-range-ket[simp]:
  ‹closure (cspan (range ket)) = UNIV›
proof (intro set-eqI iffI UNIV-I closure-approachable[THEN iffD2] allI impI)
  fix ψ :: ‹'a ell2›
  fix e :: real assume ‹e > 0›
  have ‹((λS. trunc-ell2 S ψ) ⟶⟶ ψ) (finite-subsets-at-top UNIV)›
    by (rule trunc-ell2-lim-at-UNIV)
  then obtain F where ‹finite F› and ‹dist (trunc-ell2 F ψ) ψ < e›
    apply (drule-tac tendstoD[OF - ‹e > 0›])
    by (auto dest: simp: eventually-finite-subsets-at-top)
```

moreover have ‹*trunc-ell2 F ψ ∈ cspan* (*range ket*)›
using ‹*finite F*› *trunc-ell2-ket-cspan* **by** *blast*
ultimately show ‹∃ *φ*∈*cspan* (*range ket*). *dist φ ψ* < *e*›
by *auto*
qed

lemma *ccspan-range-ket*[*simp*]: *ccspan* (*range ket*) = (*top*::(′*a ell2 ccsubspace*))
proof−
have ‹*closure* (*complex-vector.span* (*range ket*)) = (*UNIV*::′*a ell2 set*)›
using *Complex-L2.closed-cspan-range-ket* **by** *blast*
thus *?thesis*
by (*simp add*: *ccspan.abs-eq top-ccsubspace.abs-eq*)
qed

lemma *cspan-range-ket-finite*[*simp*]: *cspan* (*range ket* :: ′*a*::*finite ell2 set*) = *UNIV*
by (*metis closed-cspan-range-ket closure-finite-cspan finite-class.finite-UNIV finite-imageI*)

instance *ell2* :: (*finite*) *cfinite-dim*
proof
define *basis* :: ‹′*a ell2 set*› **where** ‹*basis* = *range ket*›
have ‹*finite basis*›
unfolding *basis-def* **by** *simp*
moreover have ‹*cspan basis* = *UNIV*›
by (*simp add*: *basis-def*)
ultimately show ‹∃ *basis*::′*a ell2 set*. *finite basis* ∧ *cspan basis* = *UNIV*›
by *auto*
qed

instantiation *ell2* :: (*enum*) *onb-enum* **begin**
definition *canonical-basis-ell2* = *map ket Enum.enum*
definition ‹*canonical-basis-length-ell2* (- :: ′*a ell2 itself*) = *length* (*Enum.enum* :: ′*a list*)›
instance
proof
show *distinct* (*canonical-basis*::′*a ell2 list*)
proof−
have ‹*finite* (*UNIV*::′*a set*)›
by *simp*
have ‹*distinct* (*enum-class.enum*::′*a list*)›
using *enum-distinct* **by** *blast*
moreover have ‹*inj-on ket* (*set enum-class.enum*)›
by (*meson inj-onI ket-injective*)
ultimately show *?thesis*
unfolding *canonical-basis-ell2-def*
using *distinct-map*
by *blast*
qed

```
  show is-ortho-set (set (canonical-basis::'a ell2 list))
    apply (auto simp: canonical-basis-ell2-def enum-UNIV)
    by (smt (z3) norm-ket f-inv-into-f is-ortho-set-def orthogonal-ket norm-zero)

  show cindependent (set (canonical-basis::'a ell2 list))
    apply (auto simp: canonical-basis-ell2-def enum-UNIV)
   by (smt (verit, best) norm-ket f-inv-into-f is-ortho-set-def is-ortho-set-cindependent
orthogonal-ket norm-zero)

  show cspan (set (canonical-basis::'a ell2 list)) = UNIV
    by (auto simp: canonical-basis-ell2-def enum-UNIV)

  show norm (x::'a ell2) = 1
    if (x::'a ell2) ∈ set canonical-basis
    for x :: 'a ell2
    using that unfolding canonical-basis-ell2-def
    by auto

  show ‹canonical-basis-length TYPE('a ell2) = length (canonical-basis :: 'a ell2
list)›
    by (simp add: canonical-basis-length-ell2-def canonical-basis-ell2-def)
  qed
end

lemma canonical-basis-length-ell2[code-unfold, simp]:
  length (canonical-basis ::'a::enum ell2 list) = CARD('a)
  unfolding canonical-basis-ell2-def apply simp
  using card-UNIV-length-enum by metis

lemma ket-canonical-basis: ket x = canonical-basis ! enum-idx x
proof−
  have x = (enum-class.enum::'a list) ! enum-idx x
    using enum-idx-correct[where i = x] by simp
  hence p1: ket x = ket ((enum-class.enum::'a list) ! enum-idx x)
    by simp
  have enum-idx x < length (enum-class.enum::'a list)
    using enum-idx-bound[where x = x] card-UNIV-length-enum
    by metis
  hence (map ket (enum-class.enum::'a list)) ! enum-idx x
        = ket ((enum-class.enum::'a list) ! enum-idx x)
    by auto
  thus ?thesis
    unfolding canonical-basis-ell2-def using p1 by auto
qed

lemma clinear-equal-ket:
  fixes f g :: ‹'a::finite ell2 ⇒ -›
  assumes ‹clinear f›
  assumes ‹clinear g›
```

assumes ‹$\bigwedge i.\ f\ (ket\ i) = g\ (ket\ i)$›
shows ‹$f = g$›
apply (*rule ext*)
apply (*rule complex-vector.linear-eq-on-span*[**where** f=f **and** g=g **and** B=‹*range ket*›])
using *assms* **by** *auto*

lemma *equal-ket*:
fixes $A\ B$ :: ‹('*a ell2*, '*b*::*complex-normed-vector*) *cblinfun*›
assumes ‹$\bigwedge\ x.\ A *_V ket\ x = B *_V ket\ x$›
shows ‹$A = B$›
apply (*rule cblinfun-eq-gen-eqI*[**where** G=‹*range ket*›])
using *assms* **by** *auto*

lemma *antilinear-equal-ket*:
fixes $f\ g$ :: ‹'*a*::*finite ell2* ⇒ -›
assumes ‹*antilinear f*›
assumes ‹*antilinear g*›
assumes ‹$\bigwedge i.\ f\ (ket\ i) = g\ (ket\ i)$›
shows ‹$f = g$›
proof −
have [*simp*]: ‹*clinear* (*f* ∘ *from-conjugate-space*)›
apply (*rule antilinear-o-antilinear*)
using *assms* **by** (*simp-all add*: *antilinear-from-conjugate-space*)
have [*simp*]: ‹*clinear* (*g* ∘ *from-conjugate-space*)›
apply (*rule antilinear-o-antilinear*)
using *assms* **by** (*simp-all add*: *antilinear-from-conjugate-space*)
have [*simp*]: ‹*cspan* (*to-conjugate-space* ‘ (*range ket* :: '*a ell2 set*)) = *UNIV*›
by *simp*
have *f o from-conjugate-space* = *g o from-conjugate-space*
apply (*rule ext*)
apply (*rule complex-vector.linear-eq-on-span*[**where** f=*f o from-conjugate-space* **and** g=*g o from-conjugate-space* **and** B=‹*to-conjugate-space* ‘ *range ket*›])
apply (*simp*, *simp*)
using *assms*(*3*) **by** (*auto simp*: *to-conjugate-space-inverse*)
then show $f = g$
by (*smt* (*verit*) *UNIV-I from-conjugate-space-inverse surj-def surj-fun-eq to-conjugate-space-inject*)

qed

lemma *cinner-ket-adjointI*:
fixes F::'*a ell2* ⇒$_{CL}$ - **and** G::'*b ell2* ⇒$_{CL}$-
assumes $\bigwedge\ i\ j.\ (F *_V ket\ i) \cdot_C ket\ j = ket\ i \cdot_C (G *_V ket\ j)$
shows $F = G*$
proof −
from *assms*
have ‹$(F *_V x) \cdot_C y = x \cdot_C (G *_V y)$› **if** ‹$x \in$ *range ket*› **and** ‹$y \in$ *range ket*›
for $x\ y$
using *that* **by** *auto*

then have ‹$(F *_V x) \cdot_C y = x \cdot_C (G *_V y)$› **if** ‹$x \in$ *range ket*› **for** x y
apply (*rule bounded-clinear-eq-on-closure*[**where** G=‹*range ket*› **and** t=y, *rotated 2*])
using *that* **by** (*auto intro*!: *bounded-linear-intros*)
then have ‹$(F *_V x) \cdot_C y = x \cdot_C (G *_V y)$› **for** x y
apply (*rule bounded-antilinear-eq-on*[**where** G=‹*range ket*› **and** t=x, *rotated 2*])
by (*auto intro*!: *bounded-linear-intros*)
then show *?thesis*
by (*rule adjoint-eqI*)
qed

lemma *ket-nonzero*[*simp*]: *ket* $i \neq 0$
using *norm-ket*[*of i*] **by** *force*

lemma *cindependent-ket*[*simp*]:
cindependent (*range* (*ket*::'*a*⇒-))
proof−
define S **where** S = *range* (*ket*::'*a*⇒-)
have *is-ortho-set S*
unfolding *S-def is-ortho-set-def* **by** *auto*
moreover have $0 \notin S$
unfolding *S-def*
using *ket-nonzero*
by (*simp add*: *image-iff*)
ultimately show *?thesis*
using *is-ortho-set-cindependent*[**where** A = S] **unfolding** *S-def*
by *blast*
qed

lemma *cdim-UNIV-ell2*[*simp*]: ‹*cdim* (*UNIV*::'*a*::*finite ell2 set*) = *CARD*('*a*)›
apply (*subst cspan-range-ket-finite*[*symmetric*])
by (*metis card-image cindependent-ket complex-vector.dim-span-eq-card-independent inj-ket*)

lemma *is-ortho-set-ket*[*simp*]: ‹*is-ortho-set* (*range ket*)›
using *is-ortho-set-def* **by** *fastforce*

lemma *bounded-clinear-equal-ket*:
fixes f g :: ‹'*a ell2* ⇒ -›
assumes ‹*bounded-clinear f*›
assumes ‹*bounded-clinear g*›
assumes ‹$\bigwedge i.$ f (*ket* i) = g (*ket* i)›
shows ‹$f = g$›
apply (*rule ext*)
apply (*rule bounded-clinear-eq-on-closure*[*of f g* ‹*range ket*›])
using *assms* **by** *auto*

lemma *bounded-antilinear-equal-ket*:

fixes $f\ g :: \langle 'a\ ell2 \Rightarrow -\rangle$
assumes ‹*bounded-antilinear f*›
assumes ‹*bounded-antilinear g*›
assumes $\langle \bigwedge i.\ f\ (ket\ i) = g\ (ket\ i)\rangle$
shows ‹$f = g$›
apply (*rule ext*)
apply (*rule bounded-antilinear-eq-on*[*of f g* ‹*range ket*›])
using *assms* **by** *auto*

lemma *is-onb-ket*[*simp*]: ‹*is-onb* (*range ket*)›
by (*auto simp*: *is-onb-def*)

lemma *ell2-sum-ket*: $\langle \psi = (\sum i{\in}UNIV.\ Rep\text{-}ell2\ \psi\ i *_C ket\ i)\rangle$ **for** $\psi :: \langle -::finite\ ell2\rangle$
apply *transfer* **apply** (*rule ext*)
apply (*subst sum-single*)
by *auto*

lemma *trunc-ell2-singleton*: $\langle trunc\text{-}ell2\ \{x\}\ \psi = Rep\text{-}ell2\ \psi\ x *_C ket\ x\rangle$
apply *transfer* **by** *auto*

lemma *trunc-ell2-insert*: $\langle trunc\text{-}ell2\ (insert\ x\ M)\ \varphi = Rep\text{-}ell2\ \varphi\ x *_C ket\ x + trunc\text{-}ell2\ M\ \varphi\rangle$
if ‹$x \notin M$›
using *trunc-ell2-union-disjoint*[**where** $M{=}\langle\{x\}\rangle$ **and** $N{=}M$]
using *that* **by** (*auto simp*: *trunc-ell2-singleton*)

lemma *trunc-ell2-finite-sum*: $\langle trunc\text{-}ell2\ M\ \psi = (\sum i{\in}M.\ Rep\text{-}ell2\ \psi\ i *_C ket\ i)\rangle$
if ‹*finite M*›
using *that* **apply** *induction* **by** (*auto simp*: *trunc-ell2-insert*)

lemma *is-orthogonal-trunc-ell2*: ‹*is-orthogonal* (*trunc-ell2 M ψ*) (*trunc-ell2 N φ*)›
if ‹$M \cap N = \{\}$›
proof −
have $*$: $\langle cnj\ (if\ i \in M\ then\ a\ else\ 0) * (if\ i \in N\ then\ b\ else\ 0) = 0\rangle$ **for** *a b i*
using *that* **by** *auto*
show *?thesis*
apply (*transfer fixing*: *M N*)
by (*simp add*: $*$)
qed

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

14.6 Butterflies

lemma *cspan-butterfly-ket*: $\langle cspan\ \{butterfly\ (ket\ i)\ (ket\ j) |\ (i::'b::finite)\ (j::'a::finite).\ True\} = UNIV\rangle$
proof −

have $*$: ‹$\{butterfly\ (ket\ i)\ (ket\ j) |\ (i::'b::finite)\ (j::'a::finite).\ True\} = \{butterfly\ a\ b\ |a\ b.\ a \in range\ ket \wedge b \in range\ ket\}$›
by *auto*
show *?thesis*
apply (*subst* $*$)
apply (*rule cspan-butterfly-UNIV*)
by *auto*
qed

lemma *cindependent-butterfly-ket*: ‹*cindependent* $\{butterfly\ (ket\ i)\ (ket\ j) |\ (i::'b)\ (j::'a).\ True\}$›
proof −
have $*$: ‹$\{butterfly\ (ket\ i)\ (ket\ j) |\ (i::'b)\ (j::'a).\ True\} = \{butterfly\ a\ b\ |a\ b.\ a \in range\ ket \wedge b \in range\ ket\}$›
by *auto*
show *?thesis*
apply (*subst* $*$)
apply (*rule cindependent-butterfly*)
by *auto*
qed

lemma *clinear-eq-butterfly-ketI*:
fixes $F\ G :: ‹('a::finite\ ell2 \Rightarrow_{C\,L}\ 'b::finite\ ell2) \Rightarrow 'c::complex\text{-}vector›$
assumes *clinear F* **and** *clinear G*
assumes $\bigwedge i\ j.\ F\ (butterfly\ (ket\ i)\ (ket\ j)) = G\ (butterfly\ (ket\ i)\ (ket\ j))$
shows $F = G$
apply (*rule complex-vector.linear-eq-on-span*[**where** $f{=}F$, *THEN ext*, *rotated 3*])
apply (*subst cspan-butterfly-ket*)
using *assms* **by** *auto*

lemma *sum-butterfly-ket*[*simp*]: ‹$(\sum (i::'a::finite)\in UNIV.\ butterfly\ (ket\ i)\ (ket\ i)) = id\text{-}cblinfun$›
apply (*rule equal-ket*)
apply (*subst complex-vector.linear-sum*[**where** $f{=}‹\lambda y.\ y *_V ket\ \text{-}›$])
apply (*auto simp add*: *scaleC-cblinfun.rep-eq cblinfun.add-left clinearI butterfly-def cblinfun-compose-image cinner-ket*)
apply (*subst sum.mono-neutral-cong-right*[**where** $S{=}‹\{\text{-}\}›$])
by *auto*

lemma *ell2-decompose-has-sum*: ‹$((\lambda x.\ Rep\text{-}ell2\ \varphi\ x *_C ket\ x)\ has\text{-}sum\ \varphi)\ UNIV$›
proof (*unfold has-sum-def*)
have $*$: ‹$trunc\text{-}ell2\ M\ \varphi = (\sum x{\in}M.\ Rep\text{-}ell2\ \varphi\ x *_C ket\ x)$› **if** ‹*finite M*› **for** *M*
using *that* **apply** *induction*
by (*auto simp*: *trunc-ell2-insert*)
show ‹$(sum\ (\lambda x.\ Rep\text{-}ell2\ \varphi\ x *_C ket\ x) \longrightarrow \varphi)\ (finite\text{-}subsets\text{-}at\text{-}top\ UNIV)$›
apply (*rule Lim-transform-eventually*)
apply (*rule trunc-ell2-lim-at-UNIV*)
using $*$ **by** (*rule eventually-finite-subsets-at-top-weakI*)

qed

lemma *ell2-decompose-infsum*: ‹$\varphi = (\sum_\infty x.\ \textit{Rep-ell2}\ \varphi\ x *_C \textit{ket}\ x)$›
by (*metis ell2-decompose-has-sum infsumI*)

lemma *ell2-decompose-summable*: ‹$(\lambda x.\ \textit{Rep-ell2}\ \varphi\ x *_C \textit{ket}\ x)\ \textit{summable-on}\ \textit{UNIV}$›
using *ell2-decompose-has-sum summable-on-def* **by** *blast*

lemma *Rep-ell2-cblinfun-apply-sum*: ‹$\textit{Rep-ell2}\ (A *_V \varphi)\ y = (\sum_\infty x.\ \textit{Rep-ell2}\ \varphi\ x * \textit{Rep-ell2}\ (A *_V \textit{ket}\ x)\ y)$›
proof −
have *1*: ‹$\textit{bounded-linear}\ (\lambda z.\ \textit{Rep-ell2}\ (A *_V z)\ y)$›
by (*auto intro*!: *bounded-clinear-compose*[*unfolded o-def*, *OF bounded-clinear-Rep-ell2*] *cblinfun.bounded-clinear-right bounded-clinear.bounded-linear*)
have *2*: ‹$(\lambda x.\ \textit{Rep-ell2}\ \varphi\ x *_C \textit{ket}\ x)\ \textit{summable-on}\ \textit{UNIV}$›
by (*simp add*: *ell2-decompose-summable*)
have ‹$\textit{Rep-ell2}\ (A *_V \varphi)\ y = \textit{Rep-ell2}\ (A *_V (\sum_\infty x.\ \textit{Rep-ell2}\ \varphi\ x *_C \textit{ket}\ x))\ y$›
by (*simp flip*: *ell2-decompose-infsum*)
also have ‹$\ldots = (\sum_\infty x.\ \textit{Rep-ell2}\ (A *_V (\textit{Rep-ell2}\ \varphi\ x *_C \textit{ket}\ x))\ y)$›
apply (*subst infsum-bounded-linear*[*symmetric*, **where** h=‹$\lambda z.\ \textit{Rep-ell2}\ (A *_V z)\ y$›])
using *1 2* **by** (*auto simp*: *o-def*)
also have ‹$\ldots = (\sum_\infty x.\ \textit{Rep-ell2}\ \varphi\ x * \textit{Rep-ell2}\ (A *_V \textit{ket}\ x)\ y)$›
by (*simp add*: *cblinfun.scaleC-right scaleC-ell2.rep-eq*)
finally show *?thesis*
by −
qed

14.7 One-dimensional spaces

instantiation *ell2* :: (*CARD-1*) *one* **begin**
lift-definition *one-ell2* :: *'a ell2* **is** *λ-. 1* **by** *simp*
instance..
end

lemma *ket-CARD-1-is-1*: ‹$\textit{ket}\ x = 1$› **for** x :: ‹*'a*::*CARD-1*›
apply *transfer* **by** *simp*

instantiation *ell2* :: (*CARD-1*) *times* **begin**
lift-definition *times-ell2* :: *'a ell2* ⇒ *'a ell2* ⇒ *'a ell2* **is** *λa b x. a x * b x*
by *simp*
instance..
end

instantiation *ell2* :: (*CARD-1*) *divide* **begin**
lift-definition *divide-ell2* :: *'a ell2* ⇒ *'a ell2* ⇒ *'a ell2* **is** *λa b x. a x / b x*
by *simp*
instance..

end

instantiation *ell2* :: (*CARD-1*) *inverse* **begin**
lift-definition *inverse-ell2* :: *'a ell2* ⇒ *'a ell2* **is** λ*a x*. *inverse* (*a x*)
by *simp*
instance..
end

instance *ell2* :: ({*enum*,*CARD-1*}) *one-dim*

Note: enum is not needed logically, but without it this instantiation clashes with *instantiation ell2* :: (*enum*) *onb-enum*

proof *intro-classes*
show *canonical-basis* = [*1*::*'a ell2*]
unfolding *canonical-basis-ell2-def*
apply *transfer*
by (*simp add*: *enum-CARD-1*[*of undefined*])
show *a* *$_C$ *1* * *b* *$_C$ *1* = (*a* * *b*) *$_C$ (*1*::*'a ell2*) **for** *a b*
apply (*transfer fixing*: *a b*) **by** *simp*
show *x* / *y* = *x* * *inverse y* **for** *x y* :: *'a ell2*
apply *transfer*
by (*simp add*: *divide-inverse*)
show *inverse* (*c* *$_C$ *1*) = *inverse c* *$_C$ (*1*::*'a ell2*) **for** *c* :: *complex*
apply *transfer* **by** *auto*
qed

14.8 Explicit bounded operators

definition *explicit-cblinfun* :: ‹(*'a* ⇒ *'b* ⇒ *complex*) ⇒ (*'b ell2*, *'a ell2*) *cblinfun*›
where
‹*explicit-cblinfun M* = *cblinfun-extension* (*range ket*) (λ*a*. *Abs-ell2* (λ*j*. *M j* (*inv ket a*)))›

definition *explicit-cblinfun-exists* :: ‹(*'a* ⇒ *'b* ⇒ *complex*) ⇒ *bool*› **where**
‹*explicit-cblinfun-exists M* ⟷
(∀ *a*. *has-ell2-norm* (λ*j*. *M j a*)) ∧
cblinfun-extension-exists (*range ket*) (λ*a*. *Abs-ell2* (λ*j*. *M j* (*inv ket a*)))›

lemma *explicit-cblinfun-exists-bounded*:
assumes ‹⋀*S T ψ*. *finite S* ⟹ *finite T* ⟹ (⋀*a*. *a*∉*T* ⟹ *ψ a* = *0*) ⟹
(∑ *b*∈*S*. (*cmod* (∑ *a*∈*T*. *ψ a* *$_C$ *M b a*))^2) ≤ *B* * (∑ *a*∈*T*. (*cmod* (*ψ a*))^2)›
shows ‹*explicit-cblinfun-exists M*›
proof −
define *F f* **where** ‹*F* = *complex-vector.construct* (*range ket*) *f*›
and ‹*f* = (λ*a*. *Abs-ell2* (λ*j*. *M j* (*inv ket a*)))›
from *assms*[**where** *S*=‹{}› **and** *T*=‹{*undefined*}› **and** *ψ*=‹λ*x*. *of-bool* (*x*=*undefined*)›]
have ‹*B* ≥ *0*›
by *auto*

```
  have has-norm: ‹has-ell2-norm (λb. M b a)› for a
 proof (unfold has-ell2-norm-def, intro nonneg-bdd-above-summable-on bdd-aboveI)
   show ‹0 ≤ cmod ((M x a)²)› for x
     by simp
   fix B'
   assume ‹B' ∈ sum (λx. cmod ((M x a)²)) ` {F. F ⊆ UNIV ∧ finite F}›
  then obtain S where [simp]: ‹finite S› and B'-def: ‹B' = (∑ x∈S. cmod ((M
x a)²))›
     by blast
   from assms[where S=S and T=‹{a}› and ψ=‹λx. of-bool (x=a)›]
   show ‹B' ≤ B›
     by (simp add: norm-power B'-def)
 qed
 have ‹clinear F›
   by (auto intro!: complex-vector.linear-construct simp: F-def)
 have F-B: ‹norm (F ψ) ≤ (sqrt B) * norm ψ› if ψ-range-ket: ‹ψ ∈ cspan (range
ket)› for ψ
 proof −
   from that
   obtain T' where ‹finite T'› and ‹T' ⊆ range ket› and ψT': ‹ψ ∈ cspan T'›
     by (meson vector-finitely-spanned)

   then obtain T where T'-def: ‹T' = ket ` T›
     by (meson subset-image-iff)
   have ‹finite T›
    by (metis T'-def ‹finite T'› finite-image-iff inj-ket inj-on-subset subset-UNIV)
   have ψT: ‹ψ ∈ cspan (ket ` T)›
     using T'-def ψT' by blast
   have Rep-ψ: ‹Rep-ell2 ψ x = 0› if ‹x ∉ T› for x
     using - - ψT apply (rule complex-vector.linear-eq-0-on-span)
      apply auto
     by (metis ket.rep-eq that of-bool-def)
   have ‹norm (trunc-ell2 S (F ψ)) ≤ sqrt B * norm ψ› if ‹finite S› for S
   proof −
     have ∗: ‹cconstruct (range ket) f ψ = (∑ x∈T. Rep-ell2 ψ x ∗C f (ket x))›
     proof (rule complex-vector.linear-eq-on[where x=ψ and B=‹ket ` T›])
       show ‹clinear (cconstruct (range ket) f)›
         using F-def ‹clinear F› by blast
       show ‹clinear (λa. ∑ x∈T. Rep-ell2 a x ∗C f (ket x))›
           by (auto intro!: clinear-compose[unfolded o-def, OF clinear-Rep-ell2]
complex-vector.linear-compose-sum)
       show ‹ψ ∈ cspan (ket ` T)›
         by (simp add: ψT)
       have ‹f b = (∑ x∈T. Rep-ell2 b x ∗C f (ket x))›
         if ‹b ∈ ket ` T› for b
       proof −
         define b' where ‹b' = inv ket b›
         have bb': ‹b = ket b'›
           using b'-def that by force
```

```
        show ?thesis
          apply (subst sum-single[where i=b'])
          using that by (auto simp add: ‹finite T› bb' ket.rep-eq)
      qed
      then show ‹cconstruct (range ket) f b = (∑x∈T. Rep-ell2 b x *C f (ket
x))›
        if ‹b ∈ ket ` T› for b
        apply (subst complex-vector.construct-basis)
        using that by auto
    qed
    have ‹(norm (trunc-ell2 S (F ψ)))² = (norm (trunc-ell2 S (∑x∈T. Rep-ell2
ψ x *C f (ket x))))²›
      apply (rule arg-cong[where f=‹λx. (norm (trunc-ell2 - x))²›])
      by (simp add: F-def * )
    also have ‹... = (norm (trunc-ell2 S (∑x∈T. Rep-ell2 ψ x *C Abs-ell2 (λb.
M b x))))²›
      by (simp add: f-def)
    also have ‹... = (∑i∈S. (cmod (Rep-ell2 (∑x∈T. Rep-ell2 ψ x *C Abs-ell2
(λb. M b x)) i))²)›
      by (simp add: that norm-trunc-ell2-finite real-sqrt-pow2 sum-nonneg)
    also have ‹... = (∑i∈S. (cmod (∑x∈T. Rep-ell2 ψ x *C Rep-ell2 (Abs-ell2
(λb. M b x)) i))²)›
      by (simp add: complex-vector.linear-sum[OF clinear-Rep-ell2]
          clinear.scaleC[OF clinear-Rep-ell2])
    also have ‹... = (∑i∈S. (cmod (∑x∈T. Rep-ell2 ψ x *C M i x))²)›
      using has-norm by (simp add: Abs-ell2-inverse)
    also have ‹... ≤ B * (∑x∈T. (cmod (Rep-ell2 ψ x))²)›
      using ‹finite S› ‹finite T› Rep-ψ by (rule assms)
    also have ‹... = B * ((norm (trunc-ell2 T ψ))²)›
      by (simp add: ‹finite T› norm-trunc-ell2-finite sum-nonneg)
    also have ‹... ≤ B * (norm ψ)²›
      by (simp add: mult-left-mono ‹B ≥ 0› trunc-ell2-reduces-norm)
    finally show ?thesis
      apply (rule-tac power2-le-imp-le)
      by (simp-all add: ‹0 ≤ B› power-mult-distrib)
  qed
  then show ?thesis
    by (rule norm-ell2-bound-trunc)
qed
then have ‹cblinfun-extension-exists (cspan (range ket)) F›
  apply (rule cblinfun-extension-exists-hilbert[rotated -1])
 by (auto intro: ‹clinear F› complex-vector.linear-add complex-vector.linear-scale)
then have ex: ‹cblinfun-extension-exists (range ket) f›
  apply (rule cblinfun-extension-exists-restrict[rotated -1])
 by (simp-all add: F-def complex-vector.span-superset complex-vector.construct-basis)
from ex has-norm
show ?thesis
  using explicit-cblinfun-exists-def f-def by blast
qed
```

lemma *explicit-cblinfun-exists-finite-dim*[*simp*]: ‹*explicit-cblinfun-exists m*› **for** *m* :: -::*finite* ⇒ -::*finite* ⇒ -
by (*auto simp*: *explicit-cblinfun-exists-def cblinfun-extension-exists-finite-dim*)

lemma *explicit-cblinfun-ket*: ‹*explicit-cblinfun M* $*_V$ *ket a* = *Abs-ell2* (λ*b*. *M b a*)› **if** ‹*explicit-cblinfun-exists M*›
using *that* **by** (*auto simp*: *explicit-cblinfun-exists-def explicit-cblinfun-def cblinfun-extension-apply*)

lemma *Rep-ell2-explicit-cblinfun-ket*[*simp*]: ‹*Rep-ell2* (*explicit-cblinfun M* $*_V$ *ket a*) *b* = *M b a*› **if** ‹*explicit-cblinfun-exists M*›
using *that* **apply** (*simp add*: *explicit-cblinfun-ket*)
by (*simp add*: *Abs-ell2-inverse explicit-cblinfun-exists-def*)

lemma *bounded-extension-counterexample-1*: ‹∃*f*. ∀*x*. *f* (*ket x*) = *ket 0*›
— First part of counterexample showing that not every linear function can be extended to a bounded operator.
by *auto*

lemma *bounded-extension-counterexample-2*:
— Second part of counterexample showing that not every linear function can be extended to a bounded operator.
assumes ‹∀*x*::′*a*::*infinite*. *f* (*ket x*) = *ket 0*›
shows ‹¬ *cblinfun-extension-exists* (*range ket*) *f*›
proof (*rule ccontr*, *unfold not-not*)
assume ‹*cblinfun-extension-exists* (*range ket*) *f*›
moreover define *F* **where** ‹*F* = *cblinfun-extension* (*range ket*) *f*›
ultimately have *F*: ‹*F* (*ket x*) = *ket 0*› **for** *x*
by (*simp add*: *assms cblinfun-extension-apply*)
have *F-geq*: ‹*norm F* ≥ *sqrt B*› **for** *B* :: *nat*
proof −
obtain *S* :: ‹′*a set*› **where** *card-S*: ‹*card S* = *B*› **and** *fin-S*: ‹*finite S*›
using *arb-finite-subset*[*of* ‹{}› *B*]
by (*meson finite.emptyI obtain-subset-with-card-n*)
define ψ **where** ‹ψ = (∑ *i*∈*S*. *ket i*)›
have ‹(*norm* ψ)^2 = *B*›
by (*simp add*: ψ-*def pythagorean-theorem-sum card-S fin-S*)
then have *norm-ψ*: ‹*norm* ψ = *sqrt B*›
by (*smt* (*verit*, *best*) *norm-ge-zero real-sqrt-abs*)
have ‹*F* ψ = *B* $*_R$ *ket 0*›
by (*simp add*: ψ-*def cblinfun.sum-right real-vector.sum-constant-scale F card-S*)
then have ‹*norm* (*F* ψ) = *B*›
by *simp*
with *norm-ψ* **have** ‹*norm F* ≥ *B* / *sqrt B*›
using *norm-cblinfun*[*of F* ψ]
by (*simp add*: *divide-le-eq*)
then show *?thesis*
by (*simp add*: *real-div-sqrt*)

```
  qed
  then show False
  proof −
    obtain B :: nat where B: ‹B > (norm F)^2›
      apply atomize-elim
      apply (rule exI[of - ‹nat (ceiling ((norm F)^2 + 1))›])
      by linarith
    with F-geq show False
      by (smt (verit, ccfv-threshold) B sqrt-le-D)
  qed
qed
```

14.9 Diagonal operators

```
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 ‹$(\sum b{\in}S.\ (cmod\ (\sum a{\in}T.\ \psi\ a *_C\ (of\text{-}bool\ (b = a) * f\ b)))^2)$
$\leq B^2 * (\sum a{\in}T.\ (cmod\ (\psi\ a))^2)$› .
qed
qed

lemma *diagonal-operator-ket*:
assumes ‹*bdd-above* (*range* ($\lambda x.$ *cmod* (*f x*)))›
shows ‹*diagonal-operator f* (*ket x*) = *f x* $*_C$ *ket x*›
proof −
have [*simp*]: ‹*has-ell2-norm* ($\lambda b.$ *of-bool* (*b* = *x*) * *f b*)›
by (*auto intro*!: *finite-nonzero-values-imp-summable-on simp*: *has-ell2-norm-def*)
have ‹*Abs-ell2* ($\lambda b.$ *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* ($\lambda x.$ *cmod* (*f x*)))›
shows ‹*diagonal-operator f* = *0*›
by (*simp add*: *assms diagonal-operator-def*)

lemma *diagonal-operator-adj*: ‹*diagonal-operator f*$*$ = *diagonal-operator* ($\lambda x.$ *cnj* (*f x*))›
by (*cases* ‹*bdd-above* (*range* ($\lambda x.$ *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* ($\lambda x.$ *cmod* (*f x*)))›
assumes ‹*bdd-above* (*range* ($\lambda x.$ *cmod* (*g x*)))›
shows ‹*diagonal-operator f* o_{CL} *diagonal-operator g* = *diagonal-operator* ($\lambda x.$ (*f x* * *g x*))›
proof −
have ‹*bdd-above* (*range* ($\lambda x.$ *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-mu*
then show *?thesis*
by *fast*
qed

then show *?thesis*
by (*auto intro*!: *equal-ket simp*: *diagonal-operator-ket assms cblinfun.scaleC-right*)
qed

14.10 Classical operators

We call an operator mapping *ket x* to *ket* (π *x*) or *0* "classical". (The meaning is inspired by the fact that in quantum mechanics, such operators usually correspond to operations with classical interpretation (such as Pauli-X, CNOT, measurement in the computational basis, etc.))

definition *classical-operator* :: ('*a*⇒'*b option*) ⇒ '*a ell2* ⇒$_{CL}$ '*b ell2* **where**
classical-operator π =
(*let f* = (λ*t*. (*case* π (*inv* (*ket*::'*a*⇒-) *t*)
of None ⇒ (*0*::'*b ell2*)
| *Some i* ⇒ *ket i*))
in
cblinfun-extension (*range* (*ket*::'*a*⇒-)) *f*)

definition *classical-operator-exists* π ⟷
cblinfun-extension-exists (*range ket*)
(λ*t*. *case* π (*inv ket t*) *of None* ⇒ *0* | *Some i* ⇒ *ket i*)

lemma *classical-operator-existsI*:
assumes ⋀*x*. *B* *$_V$ (*ket x*) = (*case* π *x of Some i* ⇒ *ket i* | *None* ⇒ *0*)
shows *classical-operator-exists* π
unfolding *classical-operator-exists-def*
apply (*rule cblinfun-extension-existsI*[*of* - *B*])
using *assms*
by (*auto simp*: *inv-f-f*[*OF inj-ket*])

lemma
assumes *inj-map* π
shows *classical-operator-exists-inj*: *classical-operator-exists* π
and *classical-operator-norm-inj*: ‹*norm* (*classical-operator* π) ≤ *1*›
proof −
have ‹*is-orthogonal* (*case* π *x of None* ⇒ *0* | *Some x'* ⇒ *ket x'*)
(*case* π *y of None* ⇒ *0* | *Some y'* ⇒ *ket y'*)›
if ‹*x* ≠ *y*› **for** *x y*
apply (*cases* ‹π *x*›; *cases* ‹π *y*›)
using *that assms*
by (*auto simp add*: *inj-map-def*)
then have *1*: ‹*is-orthogonal* (*case* π (*inv ket x*) *of None* ⇒ *0* | *Some x'* ⇒ *ket x'*)
(*case* π (*inv ket y*) *of None* ⇒ *0* | *Some y'* ⇒ *ket y'*)›
if ‹*x* ∈ *range ket*› **and** ‹*y* ∈ *range ket*› **and** ‹*x* ≠ *y*› **for** *x y*
using *that* **by** *auto*

have ‹*norm* (*case* π *x of None* ⇒ *0* | *Some x* ⇒ *ket x*) ≤ *1* * *norm* (*ket x*)› **for**

```
x
    apply (cases ‹π x›) by auto
  then have 2: ‹norm (case π (inv ket x) of None ⇒ 0 | Some x ⇒ ket x) ≤ 1 *
norm x›
    if ‹x ∈ range ket› for x
    using that by auto

  show ‹classical-operator-exists π›
    unfolding classical-operator-exists-def
    using - - 1 2 apply (rule cblinfun-extension-exists-ortho)
    by simp-all

  show ‹norm (classical-operator π) ≤ 1›
    unfolding classical-operator-def Let-def
    using - - 1 2 apply (rule cblinfun-extension-exists-ortho-norm)
    by simp-all
qed

lemma classical-operator-exists-finite[simp]: classical-operator-exists (π :: -::finite
⇒ -)
  unfolding classical-operator-exists-def
  using cindependent-ket cspan-range-ket-finite
  by (rule cblinfun-extension-exists-finite-dim)

lemma classical-operator-ket:
  assumes classical-operator-exists π
  shows (classical-operator π) *V (ket x) = (case π x of Some i ⇒ ket i | None
⇒ 0)
  unfolding classical-operator-def
  using f-inv-into-f ket-injective rangeI
  by (metis assms cblinfun-extension-apply classical-operator-exists-def)

lemma classical-operator-ket-finite:
  (classical-operator π) *V (ket (x::'a::finite)) = (case π x of Some i ⇒ ket i | None
⇒ 0)
  by (rule classical-operator-ket, simp)

lemma classical-operator-adjoint[simp]:
  fixes π :: 'a ⇒ 'b option
  assumes a1: inj-map π
  shows  (classical-operator π)* = classical-operator (inv-map π)
proof−
  define F where F = classical-operator (inv-map π)
  define G where G = classical-operator π
  have (F *V ket i) •C ket j = ket i •C (G *V ket j) for i j
  proof−
    have w1: (classical-operator (inv-map π)) *V (ket i)
     = (case inv-map π i of Some k ⇒ ket k | None ⇒ 0)
      by (simp add: classical-operator-ket classical-operator-exists-inj)
```

have *w2*: (*classical-operator* π) $*_V$ (*ket* j)
 = (*case* π j *of Some* $k \Rightarrow$ *ket* k | *None* $\Rightarrow$ *0*)
  by (*simp add*: *assms classical-operator-ket classical-operator-exists-inj*)
have (*F* $*_V$ *ket i*) $\cdot_C$ *ket j* = (*classical-operator* (*inv-map* π) $*_V$ *ket i*) $\cdot_C$ *ket j*
  unfolding *F-def* **by** *blast*
also have ... = ((*case inv-map* π *i of Some k* $\Rightarrow$ *ket k* | *None* $\Rightarrow$ *0*) $\cdot_C$ *ket j*)
  using *w1* **by** *simp*
also have ... = (*ket i* $\cdot_C$ (*case* π *j of Some k* $\Rightarrow$ *ket k* | *None* $\Rightarrow$ *0*))
proof(*induction inv-map* π *i*)
  case *None*
  hence *pi1*: *None* = *inv-map* π *i*.
  show *?case*
  proof (*induction* π *j*)
    case *None*
    thus *?case*
      using *pi1* **by** *auto*
  next
    case (*Some c*)
    have $c \neq i$
    proof(*rule classical*)
      assume $\neg(c \neq i)$
      hence $c = i$
        by *blast*
      hence *inv-map* π *c* = *inv-map* π *i*
        by *simp*
      hence *inv-map* π *c* = *None*
        by (*simp add*: *pi1*)
      moreover have *inv-map* π *c* = *Some j*
        using *Some.hyps* **unfolding** *inv-map-def*
        apply *auto*
        by (*metis a1 f-inv-into-f inj-map-def option.distinct*(*1*) *rangeI*)
      ultimately show *?thesis* **by** *simp*
    qed
    thus *?thesis*
   by (*metis None.hyps Some.hyps cinner-zero-left orthogonal-ket option.simps*(*4*)

        option.simps(*5*))
  qed
next
  case (*Some d*)
  hence *s1*: *Some d* = *inv-map* π *i*.
  show (*case inv-map* π *i of None* $\Rightarrow$ *0*| *Some a* $\Rightarrow$ *ket a*) $\cdot_C$ *ket j*
      = *ket i* $\cdot_C$ (*case* π *j of None* $\Rightarrow$ *0* | *Some a* $\Rightarrow$ *ket a*)
  proof(*induction* π *j*)
    case *None*
    have $d \neq j$
    proof(*rule classical*)
      assume $\neg(d \neq j)$
      hence $d = j$

```
        by blast
      hence π d = π j
        by simp
      hence π d = None
        by (simp add: None.hyps)
      moreover have π d = Some i
        using Some.hyps unfolding inv-map-def
        apply auto
        by (metis f-inv-into-f option.distinct(1) option.inject)
      ultimately show ?thesis
        by simp
    qed
    thus ?case
        by (metis None.hyps Some.hyps cinner-zero-right orthogonal-ket option.case-eq-if
        option.simps(5))
  next
    case (Some c)
    hence s2: π j = Some c by simp
    have (ket d ·C ket j) = (ket i ·C ket c)
    proof(cases π j = Some i)
      case True
      hence ij: Some j = inv-map π i
        unfolding inv-map-def apply auto
        apply (metis a1 f-inv-into-f inj-map-def option.discI range-eqI)
        by (metis range-eqI)
      have i = c
        using True s2 by auto
      moreover have j = d
        by (metis option.inject s1 ij)
      ultimately show ?thesis
        by (simp add: cinner-ket-same)
    next
      case False
      moreover have π d = Some i
        using s1 unfolding inv-map-def
        by (metis f-inv-into-f option.distinct(1) option.inject)
      ultimately have j ≠ d
        by auto
      moreover have i ≠ c
        using False s2 by auto
      ultimately show ?thesis
        by (metis orthogonal-ket)
    qed
    hence (case Some d of None ⇒ 0 | Some a ⇒ ket a) ·C ket j
        = ket i ·C (case Some c of None ⇒ 0 | Some a ⇒ ket a)
      by simp
    thus (case inv-map π i of None ⇒ 0 | Some a ⇒ ket a) ·C ket j
        = ket i ·C (case π j of None ⇒ 0 | Some a ⇒ ket a)
```

```
      by (simp add: Some.hyps s1)
    qed
   qed
   also have ... = ket i ·C (classical-operator π *V ket j)
    by (simp add: w2)
   also have ... = ket i ·C (G *V ket j)
    unfolding G-def by blast
   finally show ?thesis .
  qed
  hence G* = F
   using cinner-ket-adjointI
   by auto
  thus ?thesis unfolding G-def F-def .
qed

lemma
  fixes π::'b ⇒ 'c option and ϱ::'a ⇒ 'b option
  assumes classical-operator-exists π
  assumes classical-operator-exists ϱ
  shows classical-operator-exists-comp[simp]: classical-operator-exists (π ∘m ϱ)
   and classical-operator-mult[simp]: classical-operator π oCL classical-operator ϱ
= classical-operator (π ∘m ϱ)
proof −
 define Cπ Cϱ Cπϱ where Cπ = classical-operator π and Cϱ = classical-operator
ϱ
   and Cπϱ = classical-operator (π ∘m ϱ)
  have Cπx: Cπ *V (ket x) = (case π x of Some i ⇒ ket i | None ⇒ 0) for x
   unfolding Cπ-def using ‹classical-operator-exists π› by (rule classical-operator-ket)
  have Cϱx: Cϱ *V (ket x) = (case ϱ x of Some i ⇒ ket i | None ⇒ 0) for x
   unfolding Cϱ-def using ‹classical-operator-exists ϱ› by (rule classical-operator-ket)
  have Cπϱx': (Cπ oCL Cϱ) *V (ket x) = (case (π ∘m ϱ) x of Some i ⇒ ket i |
None ⇒ 0) for x
   apply (simp add: scaleC-cblinfun.rep-eq Cϱx)
   apply (cases ϱ x)
   by (auto simp: Cπx)
  thus ‹classical-operator-exists (π ∘m ϱ)›
   by (rule classical-operator-existsI)
  hence Cπϱ *V (ket x) = (case (π ∘m ϱ) x of Some i ⇒ ket i | None ⇒ 0) for x
   unfolding Cπϱ-def
   by (rule classical-operator-ket)
  with Cπϱx' have (Cπ oCL Cϱ) *V (ket x) = Cπϱ *V (ket x) for x
   by simp
  thus Cπ oCL Cϱ = Cπϱ
   by (simp add: equal-ket)
qed

lemma classical-operator-Some[simp]: classical-operator (Some::'a⇒-) = id-cblinfun
proof−
  have (classical-operator Some) *V (ket i)  = id-cblinfun *V (ket i)
```

for *i*::*'a*
apply (*subst classical-operator-ket*)
apply (*rule classical-operator-exists-inj*)
by *auto*
thus *?thesis*
using *equal-ket*[**where** *A* = *classical-operator* (*Some*::*'a* ⇒ - *option*)
and *B* = *id-cblinfun*::*'a ell2* ⇒$_{CL}$ -]
by *blast*
qed

lemma *isometry-classical-operator*[*simp*]:
fixes *π*::*'a* ⇒ *'b*
assumes *a1*: *inj π*
shows *isometry* (*classical-operator* (*Some o π*))
proof −
have *b0*: *inj-map* (*Some* ∘ *π*)
by (*simp add*: *a1*)
have *b0'*: *inj-map* (*inv-map* (*Some* ∘ *π*))
by *simp*
have *b1*: *inv-map* (*Some* ∘ *π*) ∘$_m$ (*Some* ∘ *π*) = *Some*
apply (*rule ext*) **unfolding** *inv-map-def o-def*
using *assms* **unfolding** *inj-def inv-def* **by** *auto*
have *b3*: *classical-operator* (*inv-map* (*Some* ∘ *π*)) *o*$_{CL}$
classical-operator (*Some* ∘ *π*) = *classical-operator* (*inv-map* (*Some* ∘ *π*)
∘$_m$ (*Some* ∘ *π*))
by (*metis b0 b0' b1 classical-operator-Some classical-operator-exists-inj*
classical-operator-mult)
show *?thesis*
unfolding *isometry-def*
apply (*subst classical-operator-adjoint*)
using *b0* **by** (*auto simp add*: *b1 b3*)
qed

lemma *unitary-classical-operator*[*simp*]:
fixes *π*::*'a* ⇒ *'b*
assumes *a1*: *bij π*
shows *unitary* (*classical-operator* (*Some o π*))
proof (*unfold unitary-def*, *rule conjI*)
have *inj π*
using *a1 bij-betw-imp-inj-on* **by** *auto*
hence *isometry* (*classical-operator* (*Some o π*))
by *simp*
hence *classical-operator* (*Some* ∘ *π*)∗ *o*$_{CL}$ *classical-operator* (*Some* ∘ *π*) =
id-cblinfun
unfolding *isometry-def* **by** *simp*
thus ‹*classical-operator* (*Some* ∘ *π*)∗ *o*$_{CL}$ *classical-operator* (*Some* ∘ *π*) = *id-cblinfun*›
by *simp*
next
have *inj π*

by (*simp add*: *assms bij-is-inj*)
have *comp*: *Some* ∘ *π* ∘$_m$ *inv-map* (*Some* ∘ *π*) = *Some*
apply (*rule ext*)
unfolding *inv-map-def o-def map-comp-def*
unfolding *inv-def* **apply** *auto*
apply (*metis* ‹*inj π*› *inv-def inv-f-f*)
using *bij-def image-iff range-eqI*
by (*metis a1*)
have *classical-operator* (*Some* ∘ *π*) *o*$_{CL}$ *classical-operator* (*Some* ∘ *π*)∗
= *classical-operator* (*Some* ∘ *π*) *o*$_{CL}$ *classical-operator* (*inv-map* (*Some* ∘ *π*))
by (*simp add*: ‹*inj π*›)
also have ... = *classical-operator* ((*Some* ∘ *π*) ∘$_m$ (*inv-map* (*Some* ∘ *π*)))
by (*simp add*: ‹*inj π*› *classical-operator-exists-inj*)
also have ... = *classical-operator* (*Some*::′*b*⇒-)
using *comp*
by *simp*
also have ... = (*id-cblinfun*:: ′*b ell2* ⇒$_{CL}$ -)
by *simp*
finally show *classical-operator* (*Some* ∘ *π*) *o*$_{CL}$ *classical-operator* (*Some* ∘ *π*)∗
= *id-cblinfun*.
qed

unbundle *no lattice-syntax* **and** *no cblinfun-syntax*

end

15 *Extra-Jordan-Normal-Form* – Additional results for `Jordan_Normal_Form`

theory *Extra-Jordan-Normal-Form*
imports
Jordan-Normal-Form.Matrix Jordan-Normal-Form.Schur-Decomposition
begin

We define bundles to activate/deactivate the notation from `Jordan_Normal_Form`.

Reactivate the notation locally via "**includes** *jnf-syntax*" in a lemma statement. (Or sandwich a declaration using that notation between "**unbundle** *jnf-syntax* **...** **unbundle** *no jnf-syntax*.)

open-bundle *jnf-syntax*
begin
notation *transpose-mat* (‹(-^{T})› [*1000*])
notation *cscalar-prod* (**infix** ‹•*c*› *70*)
notation *vec-index* (**infixl** ‹$› *100*)
notation *smult-vec* (**infixl** ‹·$_v$› *70*)
notation *scalar-prod* (**infix** ‹•› *70*)
notation *index-mat* (**infixl** ‹$$› *100*)
notation *smult-mat* (**infixl** ‹·$_m$› *70*)
notation *mult-mat-vec* (**infixl** ‹∗$_v$› *70*)

notation *pow-mat* (**infixr** ‹$\widehat{}_m$› *75*)
notation *append-vec* (**infixr** ‹$@_v$› *65*)
notation *append-rows* (**infixr** ‹$@_r$› *65*)
end

lemma *mat-entry-explicit*:
fixes *M* :: *'a*::*field mat*
assumes *M* ∈ *carrier-mat m n* **and** *i* < *m* **and** *j* < *n*
shows *vec-index* (*M* $*_v$ *unit-vec n j*) *i* = *M* $$ (*i*,*j*)
using *assms* **by** *auto*

lemma *mat-adjoint-def'*: *mat-adjoint M* = *transpose-mat* (*map-mat conjugate M*)
apply (*rule mat-eq-iff*[*THEN iffD2*])
apply (*auto simp*: *mat-adjoint-def transpose-mat-def*)
apply (*subst mat-of-rows-index*)
by *auto*

lemma *mat-adjoint-swap*:
fixes *M* ::*complex mat*
assumes *M* ∈ *carrier-mat nB nA* **and** *iA* < *dim-row M* **and** *iB* < *dim-col M*
shows (*mat-adjoint M*)$$(*iB*,*iA*) = *cnj* (*M*$$(*iA*,*iB*))
unfolding *transpose-mat-def map-mat-def*
by (*simp add*: *assms*(*2*) *assms*(*3*) *mat-adjoint-def'*)

lemma *cscalar-prod-adjoint*:
fixes *M*:: *complex mat*
assumes *M* ∈ *carrier-mat nB nA*
and *dim-vec v* = *nA*
and *dim-vec u* = *nB*
shows *v* ·*c* ((*mat-adjoint M*) $*_v$ *u*) = (*M* $*_v$ *v*) ·*c u*
unfolding *mat-adjoint-def* **using** *assms*(*1*) *assms*(*2*,*3*)[*symmetric*]
apply (*simp add*: *scalar-prod-def sum-distrib-left field-simps*)
by (*intro sum.swap*)

lemma *scaleC-minus1-left-vec*: −*1* $·_v$ *v* = − *v* **for** *v* :: -::*ring-1 vec*
unfolding *smult-vec-def uminus-vec-def* **by** *auto*

lemma *square-nneg-complex*:
fixes *x* :: *complex*
assumes *x* ∈ ℝ **shows** *x*^*2* ≥ *0*
apply (*cases x*) **using** *assms* **unfolding** *Reals-def less-eq-complex-def* **by** *auto*

definition *vec-is-zero n v* = (∀ *i*<*n*. *v* $ *i* = *0*)

lemma *vec-is-zero*: *dim-vec v* = *n* ⟹ *vec-is-zero n v* ⟷ *v* = 0_v *n*
unfolding *vec-is-zero-def* **apply** *auto*
by (*metis index-zero-vec*(*1*))

```
fun gram-schmidt-sub0
  where gram-schmidt-sub0 n us [] = us
  | gram-schmidt-sub0 n us (w # ws) =
     (let w' = adjuster n w us + w in
      if vec-is-zero n w' then gram-schmidt-sub0 n us ws
                          else gram-schmidt-sub0 n (w' # us) ws)

lemma (in cof-vec-space) adjuster-already-in-span:
  assumes w ∈ carrier-vec n
  assumes us-carrier: set us ⊆ carrier-vec n
  assumes corthogonal us
  assumes w ∈ span (set us)
  shows adjuster n w us + w = 0_v n
proof −
  define v U where v = adjuster n w us + w and U = set us
  have span: v ∈ span U
    unfolding v-def U-def
    apply (rule adjust-preserves-span[THEN iffD1])
    using assms corthogonal-distinct by simp-all
  have v-carrier: v ∈ carrier-vec n
    by (simp add: v-def assms corthogonal-distinct)
  have v ·c us!i = 0 if i < length us for i
    unfolding v-def
    apply (rule adjust-zero)
    using that assms by simp-all
  hence v ·c u = 0 if u ∈ U for u
    by (metis assms(3) U-def corthogonal-distinct distinct-Ex1 that)
  hence ortho: u ·c v = 0 if u ∈ U for u
    apply (subst conjugate-zero-iff[symmetric])
    apply (subst conjugate-vec-sprod-comm)
    using that us-carrier v-carrier apply (auto simp: U-def)[2]
    apply (subst conjugate-conjugate-sprod)
    using that us-carrier v-carrier by (auto simp: U-def)
  from span obtain a where v: lincomb a U = v
    apply atomize-elim apply (rule finite-in-span[simplified])
    unfolding U-def using us-carrier by auto
  have v ·c v = (∑ u∈U. (a u ·_v u) ·c v)
    apply (subst v[symmetric])
    unfolding lincomb-def
    apply (subst finsum-scalar-prod-sum)
    using U-def span us-carrier by auto
  also have ... = (∑ u∈U. a u * (u ·c v))
    using U-def assms(1) in-mono us-carrier v-def by fastforce
  also have ... = (∑ u∈U. a u * conjugate 0)
    apply (rule sum.cong, simp)
    using span span-closed U-def us-carrier ortho by auto
  also have ... = 0
    by auto
  finally have v ·c v = 0
```

```
    by −
  thus v = 0_v n
    using U-def conjugate-square-eq-0-vec span span-closed us-carrier by blast
qed


lemma (in cof-vec-space) gram-schmidt-sub0-result:
  assumes gram-schmidt-sub0 n us ws = us'
    and set ws ⊆ carrier-vec n
    and set us ⊆ carrier-vec n
    and distinct us
    and ~ lin-dep (set us)
    and corthogonal us
  shows set us' ⊆ carrier-vec n ∧
        distinct us' ∧
        corthogonal us' ∧
        span (set (us @ ws)) = span (set us')
  using assms
proof (induct ws arbitrary: us us')
  case (Cons w ws)
  show ?case
  proof (cases w ∈ span (set us))
    case False
    let ?v = adjuster n w us
    have wW[simp]: set (w#ws) ⊆ carrier-vec n using Cons by simp
    hence W[simp]: set ws ⊆ carrier-vec n
      and w[simp]: w : carrier-vec n by auto
    have U[simp]: set us ⊆ carrier-vec n using Cons by simp
    have UW: set (us@ws) ⊆ carrier-vec n by simp
    have wU: set (w#us) ⊆ carrier-vec n by simp
    have dist-U: distinct us using Cons by simp
    have w-U: w ∉ set us using False using span-mem by auto
    have ind-U: ~ lin-dep (set us)
      using Cons by simp
    have ind-wU: ~ lin-dep (insert w (set us))
      apply (subst lin-dep-iff-in-span[simplified, symmetric])
      using w-U ind-U False by auto
    thm lin-dep-iff-in-span[simplified, symmetric]
    have corth: corthogonal us using Cons by simp
    have ?v + w ≠ 0_v n
      by (simp add: False adjust-nonzero dist-U)
    hence ¬ vec-is-zero n (?v + w)
      by (simp add: vec-is-zero)
    hence U'def: gram-schmidt-sub0 n ((?v + w)#us) ws = us'
      using Cons by simp
    have v: ?v : carrier-vec n using dist-U by auto
    hence vw: ?v + w : carrier-vec n by auto
    hence vwU: set ((?v + w) # us) ⊆ carrier-vec n by auto
    have vsU: ?v : span (set us)
```

```
    apply (rule adjuster-in-span[OF w])
    using Cons by simp-all
  hence vsUW: ?v : span (set (us @ ws))
    using span-is-monotone[of set us set (us@ws)] by auto
  have wsU: w ∉ span (set us)
    using lin-dep-iff-in-span[OF U ind-U w w-U] ind-wU by auto
  hence vwU: ?v + w ∉ span (set us) using adjust-not-in-span[OF w U dist-U]
by auto

  have span: ?v + w ∉ span (set us)
    apply (subst span-add[symmetric])
    by (simp-all add: False vsU)
  hence vwUS: ?v + w ∉ set us using span-mem by auto

  have vwU: set ((?v + w) # us) ⊆ carrier-vec n
    using U w vw by simp
  have dist2: distinct (((?v + w) # us))
    using vwUS
    by (simp add: dist-U)

  have orth2: corthogonal ((adjuster n w us + w) # us)
    using adjust-orthogonal[OF U corth w wsU].

  have ind-vwU: ~ lin-dep (set ((adjuster n w us + w) # us))
    apply simp
    apply (subst lin-dep-iff-in-span[simplified, symmetric])
    by (simp-all add: ind-U vw vwUS span)

  have span-UwW-U': span (set (us @ w # ws)) = span (set us')
    using Cons(1)[OF U'def W vwU dist2 ind-vwU orth2]
     using span-Un[OF vwU wU gram-schmidt-sub-span[OF w U dist-U] W W
refl]
    by simp

  show ?thesis
    apply (intro conjI)
     using Cons(1)[OF U'def W vwU dist2 ind-vwU orth2] span-UwW-U' by
simp-all
 next
  case True

  let ?v = adjuster n w us
  have ?v + w = 0_v n
    apply (rule adjuster-already-in-span)
    using True Cons by auto
  hence vec-is-zero n (?v + w)
    by (simp add: vec-is-zero)
  hence U'-def: us' = gram-schmidt-sub0 n us ws
    using Cons by simp
```

```
    have span: span (set (us @ w # ws)) = span (set us')
    proof −
      have wU-U: span (set (w # us)) = span (set us)
        apply (subst already-in-span[OF - True, simplified])
        using Cons by auto
      have span (set (us @ w # ws)) = span (set (w # us) ∪ set ws)
        by simp
      also have ... = span (set us ∪ set ws)
        apply (rule span-Un) using wU-U Cons by auto
      also have ... = local.span (set us')
        using Cons U'-def by auto
      finally show ?thesis
        by −
    qed
    moreover have set us' ⊆ carrier-vec n ∧ distinct us' ∧ corthogonal us'
      unfolding U'-def using Cons by simp
    ultimately show ?thesis
      by auto
  qed
qed simp
```

This is a variant of *gram-schmidt* that does not require the input vectors *ws* to be distinct or linearly independent. (In comparison to *gram-schmidt*, our version also returns the result in reversed order.)

```
definition gram-schmidt0 n ws = gram-schmidt-sub0 n [] ws

lemma (in cof-vec-space) gram-schmidt0-result:
  fixes ws
  defines us' ≡ gram-schmidt0 n ws
  assumes ws: set ws ⊆ carrier-vec n
  shows set us' ⊆ carrier-vec n        (is ?thesis1)
    and distinct us'                   (is ?thesis2)
    and corthogonal us'                (is ?thesis3)
    and span (set ws) = span (set us')  (is ?thesis4)
proof −
  have carrier-empty: set [] ⊆ carrier-vec n by auto
  have distinct-empty: distinct [] by simp
  have indep-empty: lin-indpt (set [])
    using basis-def subset-li-is-li unit-vecs-basis by auto
  have ortho-empty: corthogonal [] by auto
  note gram-schmidt-sub0-result' = gram-schmidt-sub0-result
    [OF us'-def[symmetric, THEN meta-eq-to-obj-eq, unfolded gram-schmidt0-def]
ws
      carrier-empty distinct-empty indep-empty ortho-empty]
  thus ?thesis1 ?thesis2 ?thesis3 ?thesis4
    by auto
qed

locale complex-vec-space = cof-vec-space n TYPE(complex) for n :: nat
```

lemma *gram-schmidt0-corthogonal*:
assumes *a1*: *corthogonal R*
and *a2*: $\bigwedge x.\ x \in set\ R \Longrightarrow dim\text{-}vec\ x = d$
shows *gram-schmidt0 d R* = *rev R*
proof −
have *gram-schmidt-sub0 d U R* = *rev R* @ *U*
if *corthogonal* ((*rev U*) @ *R*)
and $\bigwedge x.\ x \in set\ U \cup set\ R \Longrightarrow dim\text{-}vec\ x = d$ **for** *U*
proof (*insert that*, *induction R arbitrary*: *U*)
case *Nil*
show *?case*
by *auto*
next
case (*Cons a R*)
have $a \in set\ (rev\ U\ @\ a\ \#\ R)$
by *simp*
moreover have *uar*: *corthogonal* (*rev U* @ *a* # *R*)
by (*simp add*: *Cons.prems*(*1*))
ultimately have ‹$a \neq 0_v\ d$›
unfolding *corthogonal-def*
by (*metis conjugate-zero-vec in-set-conv-nth scalar-prod-right-zero zero-carrier-vec*)
then have *nonzero-a*: ¬ *vec-is-zero d a*
by (*simp add*: *Cons.prems*(*2*) *vec-is-zero*)
define *T* **where** *T* = *rev U* @ *a* # *R*
have *T* ! *length* (*rev U*) = *a*
unfolding *T-def*
by (*meson nth-append-length*)
moreover have (*T* ! *i* ·*c T* ! *j* = *0*) = (*i* ≠ *j*) **if** *i*<*length T* **and** *j*<*length T*
for *i j*
using *uar that*
unfolding *corthogonal-def T-def*
by *auto*
moreover have *length* (*rev U*) < *length T*
by (*simp add*: *T-def*)
ultimately have (*T* ! (*length* (*rev U*)) ·*c T* ! *j* = *0*) = (*length* (*rev U*) ≠ *j*)
if *j*<*length T* **for** *j*
using *that* **by** *blast*
hence *T* ! (*length* (*rev U*)) ·*c T* ! *j* = *0*
if *j*<*length T* **and** *j* ≠ *length* (*rev U*) **for** *j*
using *that* **by** *blast*
hence *a* ·*c T* ! *j* = *0* **if** *j* < *length* (*rev U*) **for** *j*
using ‹*T* ! *length* (*rev U*) = *a*› *that*(*1*)
‹*length* (*rev U*) < *length T*› *dual-order.strict-trans* **by** *blast*
moreover have *T* ! *j* = (*rev U*) ! *j* **if** *j* < *length* (*rev U*) **for** *j*
by (*smt T-def* ‹*length* (*rev U*) < *length T*› *dual-order.strict-trans list-update-append1*
list-update-id nth-list-update-eq that)
ultimately have *a* ·*c u* = *0* **if** *u* ∈ *set* (*rev U*) **for** *u*
by (*metis in-set-conv-nth that*)

```
    hence a ·c u = 0 if u ∈ set U for u
      by (simp add: that)
    moreover have ⋀x. x ∈ set U ⟹ dim-vec x = d
      by (simp add: Cons.prems(2))
    ultimately have adjuster d a U = 0_v d
    proof(induction U)
      case Nil
      then show ?case by simp
    next
      case (Cons u U)
      moreover have 0 ·_v u + 0_v d = 0_v d
      proof−
        have dim-vec u = d
          by (simp add: calculation(3))
        thus ?thesis
          by auto
      qed
      ultimately show ?case by auto
    qed
    hence adjuster-a: adjuster d a U + a = a
      by (simp add: Cons.prems(2) carrier-vecI)
    have gram-schmidt-sub0 d U (a # R) = gram-schmidt-sub0 d (a # U) R
      by (simp add: adjuster-a nonzero-a)
    also have ... = rev (a # R) @ U
      apply (subst Cons.IH)
      using Cons.prems by simp-all
    finally show ?case
      by −
  qed
  from this[where U=[]] show ?thesis
    unfolding gram-schmidt0-def using assms by auto
qed

lemma adjuster-carrier′:
  assumes w: (w :: 'a::conjugatable-field vec) : carrier-vec n
    and us: set (us :: 'a vec list) ⊆ carrier-vec n
  shows adjuster n w us ∈ carrier-vec n
  by (insert us, induction us, auto)

lemma eq-mat-on-vecI:
  fixes M N :: ‹'a::field mat›
  assumes eq: ‹⋀v. v∈carrier-vec nA ⟹ M *_v v = N *_v v›
  assumes [simp]: ‹M ∈ carrier-mat nB nA› ‹N ∈ carrier-mat nB nA›
  shows ‹M = N›
proof (rule eq-matI)
  show [simp]: ‹dim-row M = dim-row N› ‹dim-col M = dim-col N›
    using assms(2) assms(3) by blast+
  fix i j
  assume [simp]: ‹i < dim-row N› ‹j < dim-col N›
```

show ‹M $$ $(i, j) = N$ $$ (i, j)›
thm *mat-entry-explicit*[**where** M=M]
apply (*subst mat-entry-explicit*[*symmetric*])
using *assms* **apply** *auto*[*3*]
apply (*subst mat-entry-explicit*[*symmetric*])
using *assms* **apply** *auto*[*3*]
apply (*subst eq*)
apply *auto* **using** *assms*(*3*) *unit-vec-carrier* **by** *blast*
qed

lemma *list-of-vec-plus*:
fixes *v1 v2* :: ‹*complex vec*›
assumes ‹*dim-vec v1* = *dim-vec v2*›
shows ‹*list-of-vec* (*v1* + *v2*) = *map2* (+) (*list-of-vec v1*) (*list-of-vec v2*)›
proof−
have ‹i < *dim-vec v1* ⟹ (*list-of-vec* (*v1* + *v2*)) ! i = (*map2* (+) (*list-of-vec v1*) (*list-of-vec v2*)) ! i›
for i
by (*simp add*: *assms*)
thus *?thesis*
by (*metis assms index-add-vec*(*2*) *length-list-of-vec length-map map-fst-zip nth-equalityI*)
qed

lemma *list-of-vec-mult*:
fixes v :: ‹*complex vec*›
shows ‹*list-of-vec* (c ·$_v$ v) = *map* ((∗) c) (*list-of-vec v*)›
by (*metis* (*mono-tags*, *lifting*) *index-smult-vec*(*1*) *index-smult-vec*(*2*) *length-list-of-vec length-map nth-equalityI nth-list-of-vec nth-map*)

lemma *map-map-vec-cols*: ‹*map* (*map-vec f*) (*cols m*) = *cols* (*map-mat f m*)›
by (*simp add*: *cols-def*)

lemma *map-vec-conjugate*: ‹*map-vec conjugate v* = *conjugate v*›
by *fastforce*

unbundle *no jnf-syntax*

end

16 *Cblinfun-Matrix* – Matrix representation of bounded operators

theory *Cblinfun-Matrix*
imports
Complex-L2

Jordan-Normal-Form.Gram-Schmidt

HOL−Analysis.Starlike
Complex-Bounded-Operators.Extra-Jordan-Normal-Form
begin

hide-const (**open**) *Order.bottom Order.top*
hide-type (**open**) *Finite-Cartesian-Product.vec*
hide-const (**open**) *Finite-Cartesian-Product.mat*
hide-fact (**open**) *Finite-Cartesian-Product.mat-def*
hide-const (**open**) *Finite-Cartesian-Product.vec*
hide-fact (**open**) *Finite-Cartesian-Product.vec-def*
hide-const (**open**) *Finite-Cartesian-Product.row*
hide-fact (**open**) *Finite-Cartesian-Product.row-def*
no-notation *Finite-Cartesian-Product.vec-nth* (**infixl** ‹$› *90*)

unbundle *jnf-syntax*
unbundle *cblinfun-syntax*

16.1 Isomorphism between vectors

We define the canonical isomorphism between vectors in some complex vector space $'a$ and the complex n-dimensional vectors (where n is the dimension of $'a$). This is possible if $'a$, $'b$ are of class *basis-enum* since that class fixes a finite canonical basis. Vector are represented using the *complex vec* type from `Jordan_Normal_Form`. (The isomorphism will be called *vec-of-onb-basis* below.)

definition *vec-of-basis-enum* :: ‹$'a$::*basis-enum* ⇒ *complex vec*› **where**
— Maps v to a $'a$ *vec* represented in basis *canonical-basis*
‹*vec-of-basis-enum v* = *vec-of-list* (*map* (*crepresentation* (*set canonical-basis*) *v*) *canonical-basis*)›

lemma *dim-vec-of-basis-enum'*[*simp*]:
‹*dim-vec* (*vec-of-basis-enum* (*v*::$'a$)) = *length* (*canonical-basis*::$'a$::*basis-enum list*)›
unfolding *vec-of-basis-enum-def*
by *simp*

definition *basis-enum-of-vec* :: ‹*complex vec* ⇒ $'a$::*basis-enum*› **where**
‹*basis-enum-of-vec v* =
(*if dim-vec v* = *length* (*canonical-basis* :: $'a$ *list*)
then sum-list (*map2* (∗$_C$) (*list-of-vec v*) (*canonical-basis* :: $'a$ *list*))
else 0)›

lemma *vec-of-basis-enum-inverse*[*simp*]:
fixes ψ :: $'a$::*basis-enum*
shows *basis-enum-of-vec* (*vec-of-basis-enum* ψ) = ψ
unfolding *vec-of-basis-enum-def basis-enum-of-vec-def*
unfolding *list-vec zip-map1 zip-same-conv-map map-map*
apply (*simp add*: *o-def*)
apply (*subst sum.distinct-set-conv-list*[*symmetric*], *simp*)

apply (*rule complex-vector.sum-representation-eq*)
using *is-generator-set* **by** *auto*

lemma *basis-enum-of-vec-inverse*[*simp*]:
fixes *v* :: *complex vec*
defines *n* ≡ *length* (*canonical-basis* :: *'a*::*basis-enum list*)
assumes *f1*: *dim-vec v* = *n*
shows *vec-of-basis-enum* ((*basis-enum-of-vec v*)::*'a*) = *v*
proof (*rule eq-vecI*)
show ‹*dim-vec* (*vec-of-basis-enum* (*basis-enum-of-vec v* :: *'a*)) = *dim-vec v*›
by (*auto simp*: *vec-of-basis-enum-def f1 n-def*)
next
fix *j* **assume** *j-v*: ‹*j* < *dim-vec v*›
define *w* **where** *w* = *list-of-vec v*
define *basis* **where** *basis* = (*canonical-basis*::*'a list*)
have [*simp*]: *length w* = *n length basis* = *n* ‹*dim-vec v* = *n*› ‹*length* (*canonical-basis*::*'a list*) = *n*›
‹*j* < *n*›
using *j-v* **by** (*auto simp*: *f1 basis-def w-def n-def*)
have [*simp*]: ‹*cindependent* (*set basis*)› ‹*cspan* (*set basis*) = *UNIV*›
by (*auto simp*: *basis-def is-cindependent-set is-generator-set*)

have ‹*vec-of-basis-enum* ((*basis-enum-of-vec v*)::*'a*) $ *j*
= *map* (*crepresentation* (*set basis*) (*sum-list* (*map2* (∗$_C$) *w basis*))) *basis* ! *j*›
by (*auto simp*: *vec-of-list-index vec-of-basis-enum-def basis-enum-of-vec-def simp flip*: *w-def basis-def*)
also have ‹... = *crepresentation* (*set basis*) (*sum-list* (*map2* (∗$_C$) *w basis*)) (*basis*!*j*)›
by *simp*
also have ‹... = *crepresentation* (*set basis*) (∑ *i*<*n*. (*w*!*i*) ∗$_C$ (*basis*!*i*)) (*basis*!*j*)›
by (*auto simp*: *sum-list-sum-nth atLeast0LessThan*)
also have ‹... = (∑ *i*<*n*. (*w*!*i*) ∗$_C$ *crepresentation* (*set basis*) (*basis*!*i*) (*basis*!*j*))›
by (*auto simp*: *complex-vector.representation-sum complex-vector.representation-scale*)
also have ‹... = *w*!*j*›
apply (*subst sum-single*[**where** *i*=*j*])
apply (*auto simp*: *complex-vector.representation-basis*)
using ‹*j* < *n*› ‹*length basis* = *n*› *basis-def distinct-canonical-basis nth-eq-iff-index-eq*
by *blast*
also have ‹... = *v* $ *j*›
by (*simp add*: *w-def*)
finally show ‹*vec-of-basis-enum* (*basis-enum-of-vec v* :: *'a*) $ *j* = *v* $ *j*›
by −
qed

lemma *basis-enum-eq-vec-of-basis-enumI*:
fixes *a b* :: -::*basis-enum*
assumes *vec-of-basis-enum a* = *vec-of-basis-enum b*
shows *a* = *b*
by (*metis assms vec-of-basis-enum-inverse*)

lemma *vec-of-basis-enum-carrier-vec*[*simp*]: ‹*vec-of-basis-enum v* ∈ *carrier-vec* (*canonical-basis-length TYPE*('*a*))› **for** *v* :: ‹'*a*::*basis-enum*›
apply (*simp only*: *dim-vec-of-basis-enum'* *carrier-vec-def* *vec-of-basis-enum-def*)
by (*simp add*: *canonical-basis-length*)

lemma *vec-of-basis-enum-inj*: *inj vec-of-basis-enum*
by (*simp add*: *basis-enum-eq-vec-of-basis-enumI injI*)

lemma *basis-enum-of-vec-inj*: *inj-on* (*basis-enum-of-vec* :: *complex vec* ⇒ '*a*)
(*carrier-vec* (*length* (*canonical-basis* :: '*a*::{*basis-enum*,*complex-normed-vector*} *list*)))
by (*metis basis-enum-of-vec-inverse carrier-dim-vec inj-on-inverseI*)

16.2 Operations on vectors

lemma *basis-enum-of-vec-add*:
assumes [*simp*]: ‹*dim-vec v1* = *length* (*canonical-basis* :: '*a*::*basis-enum list*)›
‹*dim-vec v2* = *length* (*canonical-basis* :: '*a list*)›
shows ‹((*basis-enum-of-vec* (*v1* + *v2*)) :: '*a*) = *basis-enum-of-vec v1* + *basis-enum-of-vec v2*›
proof −
have ‹*length* (*list-of-vec v1*) = *length* (*list-of-vec v2*)› **and** ‹*length* (*list-of-vec v2*) = *length* (*canonical-basis* :: '*a list*)›
by *simp-all*
then have ‹*sum-list* (*map2* (∗$_C$) (*map2* (+) (*list-of-vec v1*) (*list-of-vec v2*)) (*canonical-basis*::'*a list*))
= *sum-list* (*map2* (∗$_C$) (*list-of-vec v1*) *canonical-basis*) + *sum-list* (*map2* (∗$_C$) (*list-of-vec v2*) *canonical-basis*)›
apply (*induction rule*: *list-induct3*)
by (*auto simp*: *scaleC-add-left*)
then show *?thesis*
using *assms* **by** (*auto simp*: *basis-enum-of-vec-def list-of-vec-plus*)
qed

lemma *basis-enum-of-vec-mult*:
assumes [*simp*]: ‹*dim-vec v* = *length* (*canonical-basis* :: '*a*::*basis-enum list*)›
shows ‹((*basis-enum-of-vec* (*c* ·$_v$ *v*)) :: '*a*) = *c* ∗$_C$ *basis-enum-of-vec v*›
proof −
have ∗: ‹*monoid-add-hom* ((∗$_C$) *c* :: '*a* ⇒ -)›
by (*simp add*: *monoid-add-hom-def plus-hom.intro scaleC-add-right semigroup-add-hom.intro zero-hom.intro*)
show *?thesis*
apply (*auto simp*: *basis-enum-of-vec-def list-of-vec-mult map-zip-map*
monoid-add-hom.hom-sum-list[*OF* ∗])
by (*metis case-prod-unfold comp-apply scaleC-scaleC*)
qed

lemma *vec-of-basis-enum-add*:

‹*vec-of-basis-enum* (*a* + *b*) = *vec-of-basis-enum a* + *vec-of-basis-enum b*›
by (*auto simp*: *vec-of-basis-enum-def complex-vector.representation-add*)

lemma *vec-of-basis-enum-scaleC*:
vec-of-basis-enum (*c* *$_C$ *b*) = *c* ·$_v$ (*vec-of-basis-enum b*)
by (*auto simp*: *vec-of-basis-enum-def complex-vector.representation-scale*)

lemma *vec-of-basis-enum-scaleR*:
vec-of-basis-enum (*r* *$_R$ *b*) = *complex-of-real r* ·$_v$ (*vec-of-basis-enum b*)
by (*simp add*: *scaleR-scaleC vec-of-basis-enum-scaleC*)

lemma *vec-of-basis-enum-uminus*:
vec-of-basis-enum (− *b2*) = − *vec-of-basis-enum b2*
unfolding *scaleC-minus1-left*[*symmetric*, *of b2*]
unfolding *scaleC-minus1-left-vec*[*symmetric*]
by (*rule vec-of-basis-enum-scaleC*)

lemma *vec-of-basis-enum-minus*:
vec-of-basis-enum (*b1* − *b2*) = *vec-of-basis-enum b1* − *vec-of-basis-enum b2*
by (*metis* (*mono-tags*, *opaque-lifting*) *carrier-vec-dim-vec diff-conv-add-uminus diff-zero index-add-vec*(2) *minus-add-uminus-vec vec-of-basis-enum-add vec-of-basis-enum-uminus*)

lemma *cinner-basis-enum-of-vec*:
defines *n* ≡ *length* (*canonical-basis* :: *'a*::*onb-enum list*)
assumes [*simp*]: *dim-vec x* = *n dim-vec y* = *n*
shows (*basis-enum-of-vec x* :: *'a*) ·$_C$ *basis-enum-of-vec y* = *y* ·*c x*
proof −
have ‹(*basis-enum-of-vec x* :: *'a*) ·$_C$ *basis-enum-of-vec y*
= (∑ *i*<*n*. *x*$*i* *$_C$ *canonical-basis* ! *i* :: *'a*) ·$_C$ (∑ *i*<*n*. *y*$*i* *$_C$ *canonical-basis* ! *i*)›
by (*auto simp*: *basis-enum-of-vec-def sum-list-sum-nth atLeast0LessThan simp flip*: *n-def*)
also have ‹... = (∑ *i*<*n*. ∑ *j*<*n*. *cnj* (*x*$*i*) *$_C$ *y*$*j* *$_C$ ((*canonical-basis* ! *i* :: *'a*) ·$_C$ (*canonical-basis* ! *j*)))›
apply (*subst cinner-sum-left*)
apply (*subst cinner-sum-right*)
by (*auto simp*: *mult-ac*)
also have ‹... = (∑ *i*<*n*. ∑ *j*<*n*. *cnj* (*x*$*i*) *$_C$ *y*$*j* *$_C$ (*if i*=*j then 1 else 0*))›
apply (*rule sum.cong*[*OF refl*])
apply (*rule sum.cong*[*OF refl*])
by (*auto simp*: *cinner-canonical-basis n-def*)
also have ‹... = (∑ *i*<*n*. *cnj* (*x*$*i*) *$_C$ *y*$*i*)›
apply (*rule sum.cong*[*OF refl*])
apply (*subst sum-single*)
by *auto*
also have ‹... = *y* ·*c x*›
by (*smt* (*z3*) *assms*(2) *complex-scaleC-def conjugate-complex-def dim-vec-conjugate lessThan-atLeast0 lessThan-iff mult.commute scalar-prod-def sum.cong vec-index-conjugate*)
finally show *?thesis*

by −
qed

lemma *cscalar-prod-vec-of-basis-enum*: *cscalar-prod* (*vec-of-basis-enum* φ) (*vec-of-basis-enum* ψ) = *cinner* ψ φ
for ψ :: 'a::*onb-enum*
apply (*subst cinner-basis-enum-of-vec*[*symmetric*, **where** 'a='a])
by *simp-all*

definition ‹*norm-vec* ψ = *sqrt* ($\sum$ $i \in \{0$ ..< *dim-vec* $\psi\}$. *let* z = *vec-index* ψ i *in* (*Re* z)^2 + (*Im* z)^2)›

lemma *norm-vec-of-basis-enum*: ‹*norm* ψ = *norm-vec* (*vec-of-basis-enum* ψ)› **for** ψ :: 'a::*onb-enum*
proof −
have *norm* ψ = *sqrt* (*cmod* ($\sum i$ = 0..<*dim-vec* (*vec-of-basis-enum* ψ).
vec-of-basis-enum ψ $ i * *conjugate* (*vec-of-basis-enum* ψ) $ i))
unfolding *norm-eq-sqrt-cinner*[**where** 'a='a] *cscalar-prod-vec-of-basis-enum*[*symmetric*] *scalar-prod-def dim-vec-conjugate*
by *rule*
also have ... = *sqrt* (*cmod* ($\sum x$ = 0..<*dim-vec* (*vec-of-basis-enum* ψ).
let z = *vec-of-basis-enum* ψ $ x *in* (*Re* z)^2 + (*Im* z)^2))
apply (*subst sum.cong*, *rule refl*)
apply (*subst vec-index-conjugate*)
by (*auto simp*: *Let-def complex-mult-cnj*)
also have ... = *norm-vec* (*vec-of-basis-enum* ψ)
unfolding *Let-def norm-of-real norm-vec-def*
apply (*subst abs-of-nonneg*)
apply (*rule sum-nonneg*)
by *auto*
finally show *?thesis*
by −
qed

lemma *basis-enum-of-vec-unit-vec*:
defines *basis* ≡ (*canonical-basis*::'a::*basis-enum list*)
and n ≡ *length* (*canonical-basis* :: 'a *list*)
assumes *a3*: $i < n$
shows *basis-enum-of-vec* (*unit-vec* n i) = *basis*!i
proof−
define L::*complex list* **where** L = *list-of-vec* (*unit-vec* n i)
define I::*nat list* **where** I = [0..<n]
have *length* L = n
by (*simp add*: *L-def*)
moreover have *length basis* = n
by (*simp add*: *basis-def n-def*)
ultimately have *map2* ($*_C$) L *basis* = *map* (λj. $L!j$ $*_C$ *basis*!j) I
by (*simp add*: *I-def list-eq-iff-nth-eq*)
hence *sum-list* (*map2* ($*_C$) L *basis*) = *sum-list* (*map* (λj. $L!j$ $*_C$ *basis*!j) I)

by *simp*
also have ... = *sum* (λ*j*. *L*!*j* ∗$_C$ *basis*!*j*) {*0*..*n*−*1*}
proof−
have *set I* = {*0*..*n*−*1*}
using *I-def a3* **by** *auto*
thus *?thesis*
using *Groups-List.sum-code*[**where** *xs* = *I* **and** *g* = (λ*j*. *L*!*j* ∗$_C$ *basis*!*j*)]
by (*simp add*: *I-def*)
qed
also have ... = *sum* (λ*j*. (*list-of-vec* (*unit-vec n i*))!*j* ∗$_C$ *basis*!*j*) {*0*..*n*−*1*}
unfolding *L-def* **by** *blast*
finally have *sum-list* (*map2* (∗$_C$) (*list-of-vec* (*unit-vec n i*)) *basis*)
= *sum* (λ*j*. (*list-of-vec* (*unit-vec n i*))!*j* ∗$_C$ *basis*!*j*) {*0*..*n*−*1*}
using *L-def* **by** *blast*
also have ... = *basis* ! *i*
proof−
have (∑*j* = *0*..*n* − *1*. *list-of-vec* (*unit-vec n i*) ! *j* ∗$_C$ *basis* ! *j*) =
(∑*j* ∈ {*0*..*n* − *1*}. *list-of-vec* (*unit-vec n i*) ! *j* ∗$_C$ *basis* ! *j*)
by *simp*
also have ... = *list-of-vec* (*unit-vec n i*) ! *i* ∗$_C$ *basis* ! *i*
+ (∑*j* ∈ {*0*..*n* − *1*}−{*i*}. *list-of-vec* (*unit-vec n i*) ! *j* ∗$_C$ *basis* ! *j*)
proof−
define *a* **where** *a j* = *list-of-vec* (*unit-vec n i*) ! *j* ∗$_C$ *basis* ! *j* **for** *j*
define *S* **where** *S* = {*0*..*n* − *1*}
have *finite S*
by (*simp add*: *S-def*)
hence (∑*j* ∈ *insert i S*. *a j*) = *a i* + (∑*j*∈*S*−{*i*}. *a j*)
using *Groups-Big.comm-monoid-add-class.sum.insert-remove*
by *auto*
moreover have *S*−{*i*} = {*0*..*n*−*1*}−{*i*}
unfolding *S-def*
by *blast*
moreover have *insert i S* = {*0*..*n*−*1*}
using *S-def Suc-diff-1 a3 atLeastAtMost-iff diff-is-0-eq*′ *le-SucE le-numeral-extra*(*4*)
less-imp-le not-gr-zero
by *fastforce*
ultimately show *?thesis*
using ‹*a* ≡ λ*j*. *list-of-vec* (*unit-vec n i*) ! *j* ∗$_C$ *basis* ! *j*›
by *simp*
qed
also have ... = *list-of-vec* (*unit-vec n i*) ! *i* ∗$_C$ *basis* ! *i*
proof−
have *j* ∈ {*0*..*n* − *1*}−{*i*} ⟹ *list-of-vec* (*unit-vec n i*) ! *j* = *0*
for *j*
using *a3 atMost-atLeast0 atMost-iff diff-Suc-less index-unit-vec*(*1*) *le-less-trans*
list-of-vec-index zero-le **by** *fastforce*
hence *j* ∈ {*0*..*n* − *1*}−{*i*} ⟹ *list-of-vec* (*unit-vec n i*) ! *j* ∗$_C$ *basis* ! *j* = *0*
for *j*
by *auto*

```
    hence (∑ j ∈ {0..n − 1}−{i}. list-of-vec (unit-vec n i) ! j *C basis ! j) = 0
      by (simp add: ⟨⋀j. j ∈ {0..n − 1} − {i} ⟹ list-of-vec (unit-vec n i) ! j
*C basis ! j = 0⟩)
    thus ?thesis by simp
  qed
  also have ... = basis ! i
    by (simp add: a3)
  finally show ?thesis
    using ⟨(∑ j = 0..n − 1. list-of-vec (unit-vec n i) ! j *C basis ! j)
          = list-of-vec (unit-vec n i) ! i *C basis ! i + (∑ j∈{0..n − 1} − {i}.
list-of-vec (unit-vec n i) ! j *C basis ! j)⟩
      ⟨list-of-vec (unit-vec n i) ! i *C basis ! i + (∑ j∈{0..n − 1} − {i}. list-of-vec
(unit-vec n i) ! j *C basis ! j)
         = list-of-vec (unit-vec n i) ! i *C basis ! i⟩
      ⟨list-of-vec (unit-vec n i) ! i *C basis ! i = basis ! i⟩
    by auto
 qed
 finally have sum-list (map2 (*C) (list-of-vec (unit-vec n i)) basis)
    = basis ! i
  by (simp add: assms)
 hence sum-list (map2 scaleC (list-of-vec (unit-vec n i)) (canonical-basis::'a list))
    = (canonical-basis::'a list) ! i
  by (simp add: assms)
 thus ?thesis
  unfolding basis-enum-of-vec-def
  by (simp add: assms)
qed

lemma vec-of-basis-enum-ket:
 vec-of-basis-enum (ket i) = unit-vec (CARD('a)) (enum-idx i)
 for i::'a::enum
proof−
 have dim-vec (vec-of-basis-enum (ket i))
    = dim-vec (unit-vec (CARD('a)) (enum-idx i))
 proof−
  have dim-vec (unit-vec (CARD('a)) (enum-idx i))
    = CARD('a)
    by simp
  moreover have dim-vec (vec-of-basis-enum (ket i)) = CARD('a)
    unfolding vec-of-basis-enum-def vec-of-basis-enum-def by auto
  ultimately show ?thesis by simp
 qed
 moreover have vec-of-basis-enum (ket i) $ j =
  (unit-vec (CARD('a)) (enum-idx i)) $ j
  if j < dim-vec (vec-of-basis-enum (ket i))
  for j
 proof−
  have j-bound: j < length (canonical-basis::'a ell2 list)
    by (metis dim-vec-of-basis-enum' that)
```

```
    have y1: cindependent (set (canonical-basis::'a ell2 list))
      using is-cindependent-set by blast
    have y2: canonical-basis ! j ∈ set (canonical-basis::'a ell2 list)
      using j-bound by auto
    have p1: enum-class.enum ! enum-idx i = i
      using enum-idx-correct by blast
   moreover have p2: (canonical-basis::'a ell2 list) ! t = ket ((enum-class.enum::'a
list) ! t)
      if t < length (enum-class.enum::'a list)
      for t
      unfolding canonical-basis-ell2-def
      using that by auto
    moreover have p3: enum-idx i < length (enum-class.enum::'a list)
    proof−
      have set (enum-class.enum::'a list) = UNIV
        using UNIV-enum by blast
      hence i ∈ set (enum-class.enum::'a list)
        by blast
      thus ?thesis
        unfolding enum-idx-def
        by (metis index-of-bound length-greater-0-conv length-pos-if-in-set)
    qed
    ultimately have p4: (canonical-basis::'a ell2 list) ! (enum-idx i)  = ket i
      by auto
    have enum-idx i < length (enum-class.enum::'a list)
      using p3
      by auto
   moreover have length (enum-class.enum::'a list) = dim-vec (vec-of-basis-enum
(ket i))
      unfolding vec-of-basis-enum-def canonical-basis-ell2-def
      using dim-vec-of-basis-enum'[where v = ket i]
      unfolding canonical-basis-ell2-def by simp
   ultimately have enum-i-dim-vec: enum-idx i < dim-vec (unit-vec (CARD('a))
(enum-idx i))
      using ‹dim-vec (vec-of-basis-enum (ket i)) = dim-vec (unit-vec (CARD('a))
(enum-idx i))› by auto
    hence r1: (unit-vec (CARD('a)) (enum-idx i)) $ j
        = (if enum-idx i = j then 1 else 0)
      using ‹dim-vec (vec-of-basis-enum (ket i)) = dim-vec (unit-vec (CARD('a))
(enum-idx i))› that by auto
    moreover have vec-of-basis-enum (ket i) $ j = (if enum-idx i = j then 1 else
0)
    proof(cases enum-idx i = j)
      case True
      have crepresentation (set (canonical-basis::'a ell2 list))
         ((canonical-basis::'a ell2 list) ! j) ((canonical-basis::'a ell2 list) ! j) = 1
        using y1 y2 complex-vector.representation-basis[where
          basis = set (canonical-basis::'a ell2 list)
          and b = (canonical-basis::'a ell2 list) ! j]
```

```
      by smt

    hence vec-of-basis-enum ((canonical-basis::'a ell2 list) ! j) $ j = 1
      unfolding vec-of-basis-enum-def
      by (metis j-bound nth-map vec-of-list-index)
    hence vec-of-basis-enum ((canonical-basis::'a ell2 list) ! (enum-idx i))
        $ enum-idx i = 1
      using True by simp
    hence vec-of-basis-enum (ket i) $ enum-idx i = 1
      using p4
      by simp
    thus ?thesis using True unfolding vec-of-basis-enum-def by auto
  next
    case False
    have crepresentation (set (canonical-basis::'a ell2 list))
       ((canonical-basis::'a ell2 list) ! (enum-idx i)) ((canonical-basis::'a ell2 list)
! j) = 0
      using y1 y2 complex-vector.representation-basis[where
        basis = set (canonical-basis::'a ell2 list)
        and b = (canonical-basis::'a ell2 list) ! j]
    by (metis (mono-tags, opaque-lifting) False enum-i-dim-vec basis-enum-of-vec-inverse
          basis-enum-of-vec-unit-vec canonical-basis-length-ell2 index-unit-vec(3)
j-bound
          list-of-vec-index list-vec nth-map r1 vec-of-basis-enum-def)
    hence vec-of-basis-enum ((canonical-basis::'a ell2 list) ! (enum-idx i)) $ j = 0
      unfolding vec-of-basis-enum-def by (smt j-bound nth-map vec-of-list-index)
    hence vec-of-basis-enum ((canonical-basis::'a ell2 list) ! (enum-idx i)) $ j = 0
      by auto
    hence vec-of-basis-enum (ket i) $ j = 0
      using p4
      by simp
    thus ?thesis using False unfolding vec-of-basis-enum-def by simp
  qed
  ultimately show ?thesis by auto
 qed
 ultimately show ?thesis
  using Matrix.eq-vecI
  by auto
qed

lemma vec-of-basis-enum-zero:
 defines nA ≡ length (canonical-basis :: 'a::basis-enum list)
 shows vec-of-basis-enum (0::'a) = 0_v nA
 by (metis assms carrier-vecI dim-vec-of-basis-enum' minus-cancel-vec right-minus-eq
vec-of-basis-enum-minus)

lemma (in complex-vec-space) vec-of-basis-enum-cspan:
 fixes X :: 'a::basis-enum set
 assumes length (canonical-basis :: 'a list) = n
```

```
  shows vec-of-basis-enum ‘ cspan X = span (vec-of-basis-enum ‘ X)
proof −
  have carrier: vec-of-basis-enum ‘ X ⊆ carrier-vec n
    by (metis assms carrier-vecI dim-vec-of-basis-enum′ image-subsetI)
  have lincomb-sum: lincomb c A = vec-of-basis-enum (∑ b∈B. c′ b ∗C b)
    if B-def: B = basis-enum-of-vec ‘ A and ‹finite A›
    and AX: A ⊆ vec-of-basis-enum ‘ X and c′-def: ⋀b. c′ b = c (vec-of-basis-enum
b)
    for c c′ A and B::′a set
    unfolding B-def using ‹finite A› AX
  proof induction
    case empty
    then show ?case
      by (simp add: assms vec-of-basis-enum-zero)
  next
    case (insert x F)
    then have IH: lincomb c F = vec-of-basis-enum (∑ b∈basis-enum-of-vec ‘ F.
c′ b ∗C b)
      (is - = ?sum)
      by simp
    have xx: vec-of-basis-enum (basis-enum-of-vec x :: ′a) = x
      apply (rule basis-enum-of-vec-inverse)
      using assms carrier carrier-vecD insert.prems by auto
    have lincomb c (insert x F) = c x ·v x + lincomb c F
      apply (rule lincomb-insert2)
      using insert.hyps insert.prems carrier by auto
   also have c x ·v x = vec-of-basis-enum (c′ (basis-enum-of-vec x) ∗C (basis-enum-of-vec
x :: ′a))
      by (simp add: xx vec-of-basis-enum-scaleC c′-def)
    also note IH
    also have
      vec-of-basis-enum (c′ (basis-enum-of-vec x) ∗C (basis-enum-of-vec x :: ′a)) +
?sum
        = vec-of-basis-enum (∑ b∈basis-enum-of-vec ‘ insert x F. c′ b ∗C b)
      apply simp apply (rule sym)
      apply (subst sum.insert)
      using ‹finite F› ‹x ∉ F› dim-vec-of-basis-enum′ insert.prems
        vec-of-basis-enum-add c′-def by auto
    finally show ?case
      by −
  qed

  show ?thesis
  proof auto
    fix x assume x ∈ local.span (vec-of-basis-enum ‘ X)
    then obtain c A where xA: x = lincomb c A and finite A and AX: A ⊆
vec-of-basis-enum ‘ X
      unfolding span-def by auto
    define B::′a set and c′ where B = basis-enum-of-vec ‘ A
```

```
    and c' b = c (vec-of-basis-enum b) for b::'a
   note xA
   also have lincomb c A = vec-of-basis-enum (∑ b∈B. c' b *C b)
    using B-def ‹finite A› AX c'-def by (rule lincomb-sum)
   also have ... ∈ vec-of-basis-enum ‘ cspan X
    unfolding complex-vector.span-explicit
    apply (rule imageI) apply (rule CollectI)
    apply (rule exI) apply (rule exI)
    using ‹finite A› AX by (auto simp: B-def)
   finally show x ∈ vec-of-basis-enum ‘ cspan X
    by −
  next
   fix x assume x ∈ cspan X
   then obtain c' B where x: x = (∑ b∈B. c' b *C b) and finite B and BX: B
⊆ X
    unfolding complex-vector.span-explicit by auto
   define A and c where A = vec-of-basis-enum ‘ B
    and c b = c' (basis-enum-of-vec b) for b
   have vec-of-basis-enum x = vec-of-basis-enum (∑ b∈B. c' b *C b)
    using x by simp
   also have ... = lincomb c A
    apply (rule lincomb-sum[symmetric])
    unfolding A-def c-def using BX ‹finite B›
    by (auto simp: image-image)
   also have ... ∈ span (vec-of-basis-enum ‘ X)
    unfolding span-def apply (rule CollectI)
    apply (rule exI, rule exI)
    unfolding A-def using ‹finite B› BX by auto
   finally show vec-of-basis-enum x ∈ local.span (vec-of-basis-enum ‘ X)
    by −
  qed
qed

lemma (in complex-vec-space) basis-enum-of-vec-span:
  assumes length (canonical-basis :: 'a list) = n
  assumes Y ⊆ carrier-vec n
 shows basis-enum-of-vec ‘ local.span Y = cspan (basis-enum-of-vec ‘ Y :: 'a::basis-enum
set)
proof −
  define X::'a set where X = basis-enum-of-vec ‘ Y
 then have cspan (basis-enum-of-vec ‘ Y :: 'a set) = basis-enum-of-vec ‘ vec-of-basis-enum
‘ cspan X
   by (simp add: image-image)
  also have ... = basis-enum-of-vec ‘ local.span (vec-of-basis-enum ‘ X)
   apply (subst vec-of-basis-enum-cspan)
   using assms by simp-all
  also have vec-of-basis-enum ‘ X = Y
   unfolding X-def image-image
   apply (rule image-cong[where g=id and M=Y and N=Y, simplified])
```

```
    using assms(1) assms(2) by auto
  finally show ?thesis
    by simp
qed

lemma vec-of-basis-enum-canonical-basis:
  assumes i < length (canonical-basis :: 'a list)
  shows vec-of-basis-enum (canonical-basis!i :: 'a)
      = unit-vec (length (canonical-basis :: 'a::basis-enum list)) i
 by (metis assms basis-enum-of-vec-inverse basis-enum-of-vec-unit-vec index-unit-vec(3))

lemma vec-of-basis-enum-times:
  fixes ψ φ :: 'a::one-dim
  shows vec-of-basis-enum (ψ * φ)
   = vec-of-list [vec-index (vec-of-basis-enum ψ) 0 * vec-index (vec-of-basis-enum
φ) 0]
proof −
  have [simp]: ‹crepresentation {1} x 1 = one-dim-iso x› for x :: 'a
    apply (subst one-dim-scaleC-1[where x=x, symmetric])
    apply (subst complex-vector.representation-scale)
   by (auto simp add: complex-vector.span-base complex-vector.representation-basis)
  show ?thesis
    apply (rule eq-vecI)
    by (auto simp: vec-of-basis-enum-def)
qed

lemma vec-of-basis-enum-to-inverse:
  fixes ψ :: 'a::one-dim
 shows vec-of-basis-enum (inverse ψ) = vec-of-list [inverse (vec-index (vec-of-basis-enum
ψ) 0)]
proof −
  have [simp]: ‹crepresentation {1} x 1 = one-dim-iso x› for x :: 'a
    apply (subst one-dim-scaleC-1[where x=x, symmetric])
    apply (subst complex-vector.representation-scale)
   by (auto simp add: complex-vector.span-base complex-vector.representation-basis)
  show ?thesis
    apply (rule eq-vecI)
     apply (auto simp: vec-of-basis-enum-def)
     by (metis complex-vector.scale-cancel-right one-dim-inverse one-dim-scaleC-1
zero-neq-one)
qed

lemma vec-of-basis-enum-divide:
  fixes ψ φ :: 'a::one-dim
  shows vec-of-basis-enum (ψ / φ)
   = vec-of-list [vec-index (vec-of-basis-enum ψ) 0 / vec-index (vec-of-basis-enum
φ) 0]
 by (simp add: divide-inverse vec-of-basis-enum-to-inverse vec-of-basis-enum-times)
```

lemma *vec-of-basis-enum-1*: *vec-of-basis-enum* (*1* :: *'a*::*one-dim*) = *vec-of-list* [*1*]
proof −
have [*simp*]: ‹*crepresentation* {*1*} *x 1* = *one-dim-iso x*› **for** *x* :: *'a*
apply (*subst one-dim-scaleC-1*[**where** *x*=*x*, *symmetric*])
apply (*subst complex-vector.representation-scale*)
by (*auto simp add*: *complex-vector.span-base complex-vector.representation-basis*)
show *?thesis*
apply (*rule eq-vecI*)
by (*auto simp*: *vec-of-basis-enum-def*)
qed

lemma *vec-of-basis-enum-ell2-component*:
fixes *ψ* :: ‹*'a*::*enum ell2*›
assumes [*simp*]: ‹*i* < *CARD*(*'a*)›
shows ‹*vec-of-basis-enum ψ* $ *i* = *Rep-ell2 ψ* (*Enum.enum* ! *i*)›
proof −
let *?i* = ‹*Enum.enum* ! *i*›
have ‹*Rep-ell2 ψ* (*Enum.enum* ! *i*) = *ket ?i* ∙*C ψ*›
by (*simp add*: *cinner-ket-left*)
also have ‹. . . = *vec-of-basis-enum ψ* ∙*c vec-of-basis-enum* (*ket ?i* :: *'a ell2*)›
by (*rule cscalar-prod-vec-of-basis-enum*[*symmetric*])
also have ‹. . . = *vec-of-basis-enum ψ* ∙*c unit-vec* (*CARD*(*'a*)) *i*›
by (*simp add*: *vec-of-basis-enum-ket enum-idx-enum*)
also have ‹. . . = *vec-of-basis-enum ψ* ∙ *unit-vec* (*CARD*(*'a*)) *i*›
by (*smt* (*verit*, *best*) *assms carrier-vecI conjugate-conjugate-sprod conjugate-id conjugate-vec-sprod-comm dim-vec-conjugate eq-vecI index-unit-vec*(*3*) *scalar-prod-left-unit vec-index-conjugate*)
also have ‹. . . = *vec-of-basis-enum ψ* $ *i*›
by *simp*
finally show *?thesis*
by *simp*
qed

lemma *corthogonal-vec-of-basis-enum*:
fixes *S* :: *'a*::*onb-enum list*
shows *corthogonal* (*map vec-of-basis-enum S*) ⟷ *is-ortho-set* (*set S*) ∧ *distinct S*
proof *auto*
assume *assm*: ‹*corthogonal* (*map vec-of-basis-enum S*)›
then show ‹*is-ortho-set* (*set S*)›
by (*smt* (*verit*, *ccfv-SIG*) *cinner-eq-zero-iff corthogonal-def cscalar-prod-vec-of-basis-enum in-set-conv-nth is-ortho-set-def length-map nth-map*)
show ‹*distinct S*›
using *assm corthogonal-distinct distinct-map* **by** *blast*
next
assume ‹*is-ortho-set* (*set S*)› **and** ‹*distinct S*›
then show ‹*corthogonal* (*map vec-of-basis-enum S*)›
by (*smt* (*verit*, *ccfv-threshold*) *cinner-eq-zero-iff corthogonalI cscalar-prod-vec-of-basis-enum*

is-ortho-set-def length-map length-map nth-eq-iff-index-eq nth-map nth-map nth-mem nth-mem)
qed

16.3 Isomorphism between bounded linear functions and matrices

We define the canonical isomorphism between $'a \Rightarrow_{CL} 'b$ and the complex $n * m$-matrices (where n,m are the dimensions of $'a$, $'b$, respectively). This is possible if $'a$, $'b$ are of class *basis-enum* since that class fixes a finite canonical basis. Matrices are represented using the *complex mat* type from `Jordan_Normal_Form`. (The isomorphism will be called *mat-of-cblinfun* below.)

definition *mat-of-cblinfun* :: ‹'a::{*basis-enum*,*complex-normed-vector*} ⇒$_{CL}$ 'b::{*basis-enum*,*complex-normed-v*
⇒ *complex mat*› **where**
‹*mat-of-cblinfun f* =
mat (*length* (*canonical-basis* :: 'b *list*)) (*length* (*canonical-basis* :: 'a *list*)) (
λ (*i*, *j*). *crepresentation* (*set* (*canonical-basis*::'b *list*)) (*f* *$_V$ ((*canonical-basis*::'a *list*)!*j*)) ((*canonical-basis*::'b *list*)!*i*))›
for *f*

lift-definition *cblinfun-of-mat* :: ‹*complex mat* ⇒ 'a::{*basis-enum*,*complex-normed-vector*}
⇒$_{CL}$ 'b::{*basis-enum*,*complex-normed-vector*}› **is**
‹λ*M*. *if M*∈*carrier-mat* (*length* (*canonical-basis* :: 'b *list*)) (*length* (*canonical-basis* :: 'a *list*))
then λ*v*. *basis-enum-of-vec* (*M* *$_v$ *vec-of-basis-enum v*)
else (λ*v*. *0*)›
proof (*intro bounded-clinear-finite-dim clinearI*)
fix *M* :: *complex mat*
define *m* **where** *m* = *length* (*canonical-basis* :: 'b *list*)
define *n* **where** *n* = *length* (*canonical-basis* :: 'a *list*)
define *f*::*complex mat* ⇒ 'a ⇒ 'b
where *f M* = (*if M*∈*carrier-mat m n*
then λ*v*. *basis-enum-of-vec* (*M* *$_v$ *vec-of-basis-enum* (*v*::'a))
else (λ*v*. *0*))
for *M*::*complex mat*

show *add*: ‹*f M* (*b1* + *b2*) = *f M b1* + *f M b2*› **for** *b1 b2*
apply (*auto simp*: *f-def*)
by (*metis* (*mono-tags*, *lifting*) *carrier-matD*(*1*) *carrier-vec-dim-vec dim-mult-mat-vec dim-vec-of-basis-enum'* *m-def mult-add-distrib-mat-vec n-def basis-enum-of-vec-add vec-of-basis-enum-add*)

show *scale*: ‹*f M* (*c* *$_C$ *b*) = *c* *$_C$ *f M b*› **for** *c b*
apply (*auto simp*: *f-def*)
by (*metis carrier-matD*(*1*) *carrier-vec-dim-vec dim-mult-mat-vec dim-vec-of-basis-enum'* *m-def mult-mat-vec n-def basis-enum-of-vec-mult vec-of-basis-enum-scaleC*)
qed

lemma *cblinfun-of-mat-invalid*:
assumes ‹$M \notin$ *carrier-mat* (*canonical-basis-length* $TYPE('b::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\}))$ (*canonical-basis-length* $TYPE('a::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\})$)›
shows ‹(*cblinfun-of-mat* $M :: 'a \Rightarrow_{CL} 'b) = 0$›
apply (*transfer fixing*: M)
using *assms* **by** (*simp add*: *canonical-basis-length*)

lemma *dim-row-mat-of-cblinfun*[*simp*]: ‹*dim-row* (*mat-of-cblinfun* ($a::'a::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\} \Rightarrow_{CL} 'b::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\}$)) = *canonical-basis-length* $TYPE('b)$›
by (*simp add*: *mat-of-cblinfun-def canonical-basis-length*)

lemma *dim-col-mat-of-cblinfun*[*simp*]: ‹*dim-col* (*mat-of-cblinfun* ($a::'a::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\} \Rightarrow_{CL} 'b::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\}$)) = *canonical-basis-length* $TYPE('a)$›
by (*simp add*: *mat-of-cblinfun-def canonical-basis-length*)

lemma *mat-of-cblinfun-ell2-carrier*[*simp*]: ‹*mat-of-cblinfun* ($a::'a::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\} \Rightarrow_{CL} 'b::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\}$) ∈ *carrier-mat* (*canonical-basis-length* $TYPE('b)$) (*canonical-basis-length* $TYPE('a)$)›
by (*auto intro*!: *carrier-matI*)

lemma *basis-enum-of-vec-cblinfun-apply*:
fixes M :: *complex mat*
defines $nA \equiv$ *length* (*canonical-basis* :: $'a::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\}$ *list*)
and $nB \equiv$ *length* (*canonical-basis* :: $'b::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\}$ *list*)
assumes $M \in$ *carrier-mat* nB nA **and** *dim-vec* $x = nA$
shows *basis-enum-of-vec* ($M *_v x$) = (*cblinfun-of-mat* $M :: 'a \Rightarrow_{CL} 'b$) $*_V$ *basis-enum-of-vec* x
by (*metis assms basis-enum-of-vec-inverse cblinfun-of-mat.rep-eq*)

lemma *mat-of-cblinfun-cblinfun-apply*:
‹*vec-of-basis-enum* ($F *_V u$) = *mat-of-cblinfun* $F *_v$ *vec-of-basis-enum* u›
for $F::'a::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\} \Rightarrow_{CL} 'b::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\}$ **and** $u::'a$
proof (*rule eq-vecI*)
show ‹*dim-vec* (*vec-of-basis-enum* ($F *_V u$)) = *dim-vec* (*mat-of-cblinfun* $F *_v$ *vec-of-basis-enum* u)›
by (*simp add*: *dim-vec-of-basis-enum'* *mat-of-cblinfun-def*)
next
fix i
define *BasisA* **where** *BasisA* = (*canonical-basis*::$'a$ *list*)
define *BasisB* **where** *BasisB* = (*canonical-basis*::$'b$ *list*)
define nA **where** nA = *length* (*canonical-basis* :: $'a$ *list*)
define nB **where** nB = *length* (*canonical-basis* :: $'b$ *list*)
assume ‹$i <$ *dim-vec* (*mat-of-cblinfun* $F *_v$ *vec-of-basis-enum* u)›
then have [*simp*]: ‹$i < nB$›
by (*simp add*: *mat-of-cblinfun-def nB-def*)

define *v* **where** ‹*v* = *BasisB* ! *i*›

have *dim-row-F*[*simp*]: ‹*dim-row* (*mat-of-cblinfun F*) = *nB*›
by (*simp add*: *mat-of-cblinfun-def nB-def*)
have [*simp*]: ‹*length BasisB* = *nB*›
by (*simp add*: *nB-def BasisB-def*)
have [*simp*]: ‹*length BasisA* = *nA*›
using *BasisA-def nA-def* **by** *auto*
have [*simp*]: ‹*cindependent* (*set BasisA*)›
using *BasisA-def is-cindependent-set* **by** *auto*
have [*simp*]: ‹*cindependent* (*set BasisB*)›
using *BasisB-def is-cindependent-set* **by** *blast*
have [*simp*]: ‹*cspan* (*set BasisB*) = *UNIV*›
by (*simp add*: *BasisB-def is-generator-set*)
have [*simp*]: ‹*cspan* (*set BasisA*) = *UNIV*›
by (*simp add*: *BasisA-def is-generator-set*)

have ‹(*mat-of-cblinfun F* $*_v$ *vec-of-basis-enum u*) $ *i* =
(∑ *j* = *0*..<*nA*. *row* (*mat-of-cblinfun F*) *i* $ *j* * *crepresentation* (*set BasisA*) *u* (*vec-of-list BasisA* $ *j*))›
using *dim-row-F* **by** (*auto simp*: *mult-mat-vec-def vec-of-basis-enum-def scalar-prod-def simp flip*: *BasisA-def*)
also have ‹... = (∑ *j* = *0*..<*nA*. *crepresentation* (*set BasisB*) (*F* $*_V$ *BasisA* ! *j*) *v*
* *crepresentation* (*set BasisA*) *u* (*BasisA* ! *j*))›
apply (*rule sum.cong*[*OF refl*])
by (*auto simp*: *vec-of-list-index mat-of-cblinfun-def scalar-prod-def v-def simp flip*: *BasisA-def BasisB-def*)
also have ‹... = *crepresentation* (*set BasisB*) (*F* $*_V$ *u*) *v*› (**is** ‹(∑ *j*=-..<-. *?lhs v j*) = -›)
proof (*rule complex-vector.representation-eqI*[*symmetric*, *THEN fun-cong*])
show ‹*cindependent* (*set BasisB*)› ‹*F* $*_V$ *u* ∈ *cspan* (*set BasisB*)›
by *simp-all*
show *only-basis*: ‹(∑ *j* = *0*..<*nA*. *?lhs b j*) ≠ *0* ⟹ *b* ∈ *set BasisB*› **for** *b*
by (*metis* (*mono-tags*, *lifting*) *complex-vector.representation-ne-zero mult-not-zero sum.not-neutral-contains-not-neutral*)
then show ‹*finite* {*b*. (∑ *j* = *0*..<*nA*. *?lhs b j*) ≠ *0*}›
by (*smt* (*z3*) *List.finite-set finite-subset mem-Collect-eq subsetI*)
have ‹(∑ *b* | (∑ *j* = *0*..<*nA*. *?lhs b j*) ≠ *0*. (∑ *j* = *0*..<*nA*. *?lhs b j*) $*_C$ *b*) =
(∑ *b*∈*set BasisB*. (∑ *j* = *0*..<*nA*. *?lhs b j*) $*_C$ *b*)›
apply (*rule sum.mono-neutral-left*)
using *only-basis* **by** *auto*
also have ‹... = (∑ *b*∈*set BasisB*. (∑ *a*∈*set BasisA*. *crepresentation* (*set BasisB*) (*F* $*_V$ *a*) *b* * *crepresentation* (*set BasisA*) *u a*) $*_C$ *b*)›
apply (*subst sum.reindex-bij-betw*[**where** *h*=‹*nth BasisA*› **and** *T*=‹*set BasisA*›])
apply (*metis BasisA-def* ‹*length BasisA* = *nA*› *atLeast0LessThan bij-betw-nth distinct-canonical-basis*)
by *simp*

```
    also have ‹... = (∑ a∈set BasisA. crepresentation (set BasisA) u a *C
(∑ b∈set BasisB. crepresentation (set BasisB) (F *V a) b *C b))›
     apply (simp add: scaleC-sum-left scaleC-sum-right)
     apply (subst sum.swap)
     by (metis (no-types, lifting) mult.commute sum.cong)
   also have ‹... = (∑ a∈set BasisA. crepresentation (set BasisA) u a *C (F *V
a))›
     apply (subst complex-vector.sum-representation-eq)
     by auto
   also have ‹... = F *V (∑ a∈set BasisA. crepresentation (set BasisA) u a *C
a)›
     by (simp flip: cblinfun.scaleC-right cblinfun.sum-right)
   also have ‹... = F *V u›
     apply (subst complex-vector.sum-representation-eq)
     by auto
   finally show ‹(∑ b | (∑ j = 0..<nA. ?lhs b j) ≠ 0. (∑ j = 0..<nA. ?lhs b j)
*C b) = F *V u›
     by auto
  qed
  also have ‹crepresentation (set BasisB) (F *V u) v = vec-of-basis-enum (F *V
u) $ i›
   by (auto simp: vec-of-list-index vec-of-basis-enum-def v-def simp flip: BasisB-def)
  finally show ‹vec-of-basis-enum (F *V u) $ i = (mat-of-cblinfun F *v vec-of-basis-enum
u) $ i›
   by simp
qed

lemma mat-of-cblinfun-inverse:
  cblinfun-of-mat (mat-of-cblinfun B) = B
 for B::'a::{basis-enum,complex-normed-vector} ⇒CL 'b::{basis-enum,complex-normed-vector}
proof (rule cblinfun-eqI)
  fix x :: 'a define y where ‹y = vec-of-basis-enum x›
 then have ‹cblinfun-of-mat (mat-of-cblinfun B) *V x = ((cblinfun-of-mat (mat-of-cblinfun
B) :: 'a⇒CL'b) *V basis-enum-of-vec y)›
   by simp
  also have ‹... = basis-enum-of-vec (mat-of-cblinfun B *v vec-of-basis-enum
(basis-enum-of-vec y :: 'a))›
   apply (transfer fixing: B)
   by (simp add: mat-of-cblinfun-def)
  also have ‹... = basis-enum-of-vec (vec-of-basis-enum (B *V x))›
   by (simp add: mat-of-cblinfun-cblinfun-apply y-def)
  also have ‹... = B *V x›
   by simp
  finally show ‹cblinfun-of-mat (mat-of-cblinfun B) *V x = B *V x›
   by −
qed

lemma mat-of-cblinfun-inj: inj mat-of-cblinfun
  by (metis inj-on-def mat-of-cblinfun-inverse)
```

lemma *cblinfun-of-mat-inverse*:
fixes *M*::*complex mat*
defines *nA* ≡ *length* (*canonical-basis* :: *'a*::{*basis-enum*,*complex-normed-vector*} *list*)
and *nB* ≡ *length* (*canonical-basis* :: *'b*::{*basis-enum*,*complex-normed-vector*} *list*)
assumes *M* ∈ *carrier-mat nB nA*
shows *mat-of-cblinfun* (*cblinfun-of-mat M* :: *'a* ⇒$_{CL}$ *'b*) = *M*
by (*smt* (*verit*) *assms*(*3*) *basis-enum-of-vec-inverse carrier-matD*(*1*) *carrier-vecD cblinfun-of-mat.rep-eq dim-mult-mat-vec eq-mat-on-vecI mat-carrier mat-of-cblinfun-def mat-of-cblinfun-cblinfun-apply nA-def nB-def*)

lemma *cblinfun-of-mat-inj*: *inj-on* (*cblinfun-of-mat*::*complex mat* ⇒ *'a* ⇒$_{CL}$ *'b*)
(*carrier-mat* (*length* (*canonical-basis* :: *'b*::{*basis-enum*,*complex-normed-vector*} *list*))
(*length* (*canonical-basis* :: *'a*::{*basis-enum*,*complex-normed-vector*} *list*)))
using *cblinfun-of-mat-inverse*
by (*metis inj-onI*)

lemma *cblinfun-eq-mat-of-cblinfunI*:
assumes *mat-of-cblinfun a* = *mat-of-cblinfun b*
shows *a* = *b*
by (*metis assms mat-of-cblinfun-inverse*)

16.4 Operations on matrices

lemma *cblinfun-of-mat-plus*:
defines *nA* ≡ *length* (*canonical-basis* :: *'a*::{*basis-enum*,*complex-normed-vector*} *list*)
and *nB* ≡ *length* (*canonical-basis* :: *'b*::{*basis-enum*,*complex-normed-vector*} *list*)
assumes [*simp*,*intro*]: *M* ∈ *carrier-mat nB nA* **and** [*simp*,*intro*]: *N* ∈ *carrier-mat nB nA*
shows (*cblinfun-of-mat* (*M* + *N*) :: *'a* ⇒$_{CL}$ *'b*) = ((*cblinfun-of-mat M* + *cblinfun-of-mat N*))
proof −
have [*simp*]: ‹*vec-of-basis-enum* (*v* :: *'a*) ∈ *carrier-vec nA*› **for** *v*
by (*auto simp add*: *carrier-dim-vec dim-vec-of-basis-enum' nA-def*)
have [*simp*]: ‹*dim-row M* = *nB*› ‹*dim-row N* = *nB*›
using *carrier-matD*(*1*) **by** *auto*
show *?thesis*
apply (*transfer fixing*: *M N*)
by (*auto intro*!: *ext simp add*: *add-mult-distrib-mat-vec nA-def*[*symmetric*] *nB-def*[*symmetric*]
add-mult-distrib-mat-vec[**where** *nr*=*nB* **and** *nc*=*nA*] *basis-enum-of-vec-add*)
qed

lemma *mat-of-cblinfun-zero*:
mat-of-cblinfun (*0* :: ('*a*::{*basis-enum*,*complex-normed-vector*} ⇒$_{CL}$ '*b*::{*basis-enum*,*complex-normed-vector*}
= *0*$_m$ (*length* (*canonical-basis* :: '*b list*)) (*length* (*canonical-basis* :: '*a list*))
unfolding *mat-of-cblinfun-def*
by (*auto simp*: *complex-vector.representation-zero*)

lemma *mat-of-cblinfun-plus*:
mat-of-cblinfun (*F* + *G*) = *mat-of-cblinfun F* + *mat-of-cblinfun G*
for *F G*::'*a*::{*basis-enum*,*complex-normed-vector*} ⇒$_{CL}$ '*b*::{*basis-enum*,*complex-normed-vector*}
by (*auto simp add*: *mat-of-cblinfun-def cblinfun.add-left complex-vector.representation-add*)

lemma *mat-of-cblinfun-id*:
mat-of-cblinfun (*id-cblinfun* :: ('*a*::{*basis-enum*,*complex-normed-vector*} ⇒$_{CL}$ '*a*))
= *1*$_m$ (*length* (*canonical-basis* :: '*a list*))
apply (*rule eq-matI*)
by (*auto simp*: *mat-of-cblinfun-def complex-vector.representation-basis is-cindependent-set nth-eq-iff-index-eq*)

lemma *mat-of-cblinfun-1*:
mat-of-cblinfun (*1* :: ('*a*::*one-dim* ⇒$_{CL}$ '*b*::*one-dim*)) = *1*$_m$ *1*
apply (*rule eq-matI*)
by (*auto simp*: *mat-of-cblinfun-def complex-vector.representation-basis nth-eq-iff-index-eq*)

lemma *mat-of-cblinfun-uminus*:
mat-of-cblinfun (− *M*) = − *mat-of-cblinfun M*
for *M*::'*a*::{*basis-enum*,*complex-normed-vector*} ⇒$_{CL}$ '*b*::{*basis-enum*,*complex-normed-vector*}
proof−
define *nA* **where** *nA* = *length* (*canonical-basis* :: '*a list*)
define *nB* **where** *nB* = *length* (*canonical-basis* :: '*b list*)
have *M1*: *mat-of-cblinfun M* ∈ *carrier-mat nB nA*
unfolding *nB-def nA-def*
by (*metis add.right-neutral add-carrier-mat mat-of-cblinfun-plus mat-of-cblinfun-zero nA-def*
nB-def zero-carrier-mat)
have *M2*: *mat-of-cblinfun* (−*M*) ∈ *carrier-mat nB nA*
by (*metis add-carrier-mat mat-of-cblinfun-plus mat-of-cblinfun-zero diff-0 nA-def nB-def*
uminus-add-conv-diff zero-carrier-mat)
have *mat-of-cblinfun* (*M* − *M*) = *0*$_m$ *nB nA*
unfolding *nA-def nB-def*
by (*simp add*: *mat-of-cblinfun-zero*)
moreover have *mat-of-cblinfun* (*M* − *M*) = *mat-of-cblinfun M* + *mat-of-cblinfun* (− *M*)
by (*metis mat-of-cblinfun-plus pth-2*)
ultimately have *mat-of-cblinfun M* + *mat-of-cblinfun* (− *M*) = *0*$_m$ *nB nA*
by *simp*
thus *?thesis*
using *M1 M2*
by (*smt add-uminus-minus-mat assoc-add-mat comm-add-mat left-add-zero-mat*

minus-r-inv-mat
uminus-carrier-mat)
qed

lemma *mat-of-cblinfun-minus*:
mat-of-cblinfun (*M* − *N*) = *mat-of-cblinfun M* − *mat-of-cblinfun N*
for *M*::′*a*::{*basis-enum*,*complex-normed-vector*} ⇒$_{CL}$ ′*b*::{*basis-enum*,*complex-normed-vector*}
and *N*::′*a* ⇒$_{CL}$ ′*b*
by (*smt* (*z3*) *add-uminus-minus-mat mat-of-cblinfun-uminus mat-carrier mat-of-cblinfun-def mat-of-cblinfun-plus pth-2*)

lemma *cblinfun-of-mat-uminus*:
defines *nA* ≡ *length* (*canonical-basis* :: ′*a*::{*basis-enum*,*complex-normed-vector*} *list*)
and *nB* ≡ *length* (*canonical-basis* :: ′*b*::{*basis-enum*,*complex-normed-vector*} *list*)
assumes *M* ∈ *carrier-mat nB nA*
shows (*cblinfun-of-mat* (−*M*) :: ′*a* ⇒$_{CL}$ ′*b*) = − *cblinfun-of-mat M*
by (*smt assms add.group-axioms add.right-neutral add-minus-cancel add-uminus-minus-mat cblinfun-of-mat-plus group.inverse-inverse mat-of-cblinfun-inverse mat-of-cblinfun-zero minus-r-inv-mat uminus-carrier-mat*)

lemma *cblinfun-of-mat-minus*:
fixes *M*::*complex mat*
defines *nA* ≡ *length* (*canonical-basis* :: ′*a*::{*basis-enum*,*complex-normed-vector*} *list*)
and *nB* ≡ *length* (*canonical-basis* :: ′*b*::{*basis-enum*,*complex-normed-vector*} *list*)
assumes *M* ∈ *carrier-mat nB nA* **and** *N* ∈ *carrier-mat nB nA*
shows (*cblinfun-of-mat* (*M* − *N*) :: ′*a* ⇒$_{CL}$ ′*b*) = *cblinfun-of-mat M* − *cblinfun-of-mat N*
by (*metis assms add-uminus-minus-mat cblinfun-of-mat-plus cblinfun-of-mat-uminus pth-2 uminus-carrier-mat*)

lemma *cblinfun-of-mat-times*:
fixes *M N* ::*complex mat*
defines *nA* ≡ *length* (*canonical-basis* :: ′*a*::{*basis-enum*,*complex-normed-vector*} *list*)
and *nB* ≡ *length* (*canonical-basis* :: ′*b*::{*basis-enum*,*complex-normed-vector*} *list*)
and *nC* ≡ *length* (*canonical-basis* :: ′*c*::{*basis-enum*,*complex-normed-vector*} *list*)
assumes *a1*: *M* ∈ *carrier-mat nC nB* **and** *a2*: *N* ∈ *carrier-mat nB nA*
shows *cblinfun-of-mat* (*M* ∗ *N*) = ((*cblinfun-of-mat M*)::′*b* ⇒$_{CL}$ ′*c*) *o*$_{CL}$ ((*cblinfun-of-mat N*)::′*a* ⇒$_{CL}$ ′*b*)
proof −
have *b1*: ((*cblinfun-of-mat M*)::′*b* ⇒$_{CL}$ ′*c*) *v* = *basis-enum-of-vec* (*M* ∗$_v$ *vec-of-basis-enum v*)
for *v*

by (*metis assms*(4) *cblinfun-of-mat.rep-eq nB-def nC-def*)
have *b2*: ((*cblinfun-of-mat N*)::'*a* ⇒$_{CL}$ '*b*) *v* = *basis-enum-of-vec* (*N* ∗$_v$ *vec-of-basis-enum v*)
for *v*
by (*metis assms*(5) *cblinfun-of-mat.rep-eq nA-def nB-def*)
have *b3*: ((*cblinfun-of-mat* (*M* ∗ *N*))::'*a* ⇒$_{CL}$ '*c*) *v*
= *basis-enum-of-vec* ((*M* ∗ *N*) ∗$_v$ *vec-of-basis-enum v*)
for *v*
by (*metis assms*(4) *assms*(5) *cblinfun-of-mat.rep-eq mult-carrier-mat nA-def nC-def*)
have (*basis-enum-of-vec* ((*M* ∗ *N*) ∗$_v$ *vec-of-basis-enum v*)::'*c*)
= (*basis-enum-of-vec* (*M* ∗$_v$ (*vec-of-basis-enum* ((*basis-enum-of-vec* (*N* ∗$_v$ *vec-of-basis-enum v*))::'*b*))))
for *v*::'*a*
proof−
have *c1*: *vec-of-basis-enum* (*basis-enum-of-vec x* :: '*b*) = *x*
if *dim-vec x* = *nB*
for *x*::*complex vec*
using *that* **unfolding** *nB-def*
by *simp*
have *c2*: *vec-of-basis-enum v* ∈ *carrier-vec nA*
by (*metis* (*mono-tags*, *opaque-lifting*) *add.commute carrier-vec-dim-vec index-add-vec*(2)
index-zero-vec(2) *nA-def vec-of-basis-enum-add basis-enum-of-vec-inverse*)
have (*M* ∗ *N*) ∗$_v$ *vec-of-basis-enum v* = *M* ∗$_v$ (*N* ∗$_v$ *vec-of-basis-enum v*)
using *Matrix.assoc-mult-mat-vec a1 a2 c2* **by** *blast*
hence (*basis-enum-of-vec* ((*M* ∗ *N*) ∗$_v$ *vec-of-basis-enum v*)::'*c*)
= (*basis-enum-of-vec* (*M* ∗$_v$ (*N* ∗$_v$ *vec-of-basis-enum v*))::'*c*)
by *simp*
also have ... =
(*basis-enum-of-vec* (*M* ∗$_v$ (*vec-of-basis-enum* ((*basis-enum-of-vec* (*N* ∗$_v$ *vec-of-basis-enum v*))::'*b*))))
using *c1 a2* **by** *auto*
finally show *?thesis* **by** *simp*
qed
thus *?thesis* **using** *b1 b2 b3*
by (*simp add*: *cblinfun-eqI scaleC-cblinfun.rep-eq*)
qed

lemma *cblinfun-of-mat-adjoint*:
defines *nA* ≡ *length* (*canonical-basis* :: '*a*::*onb-enum list*)
and *nB* ≡ *length* (*canonical-basis* :: '*b*::*onb-enum list*)
fixes *M*:: *complex mat*
assumes *M* ∈ *carrier-mat nB nA*
shows ((*cblinfun-of-mat* (*mat-adjoint M*)) :: '*b* ⇒$_{CL}$ '*a*) = (*cblinfun-of-mat M*)∗
proof (*rule adjoint-eqI*)
show (*cblinfun-of-mat* (*mat-adjoint M*) ∗$_V$ *x*) ∙$_C$ *y* = *x* ∙$_C$ (*cblinfun-of-mat M* ∗$_V$ *y*)
for *x*::'*b* **and** *y*::'*a*

proof −
define u **where** $u = vec\text{-}of\text{-}basis\text{-}enum\ x$
define v **where** $v = vec\text{-}of\text{-}basis\text{-}enum\ y$
have *c1*: $vec\text{-}of\text{-}basis\text{-}enum\ ((cblinfun\text{-}of\text{-}mat\ (mat\text{-}adjoint\ M)\ *_V\ x)::'a) = (mat\text{-}adjoint\ M)\ *_v\ u$
unfolding *u-def*
by (*metis* (*mono-tags*, *lifting*) *assms*(*3*) *cblinfun-of-mat-inverse map-carrier-mat mat-adjoint-def' mat-of-cblinfun-cblinfun-apply nA-def nB-def transpose-carrier-mat*)
have *c2*: $(vec\text{-}of\text{-}basis\text{-}enum\ ((cblinfun\text{-}of\text{-}mat\ M\ *_V\ y)::'b)) = M\ *_v\ v$
by (*metis assms*(*3*) *cblinfun-of-mat-inverse mat-of-cblinfun-cblinfun-apply nA-def nB-def v-def*)
have *c3*: $dim\text{-}vec\ v = nA$
unfolding *v-def nA-def vec-of-basis-enum-def*
by (*simp add*:)
have *c4*: $dim\text{-}vec\ u = nB$
unfolding *u-def nB-def vec-of-basis-enum-def*
by (*simp add*:)
have $v \cdot_c ((mat\text{-}adjoint\ M)\ *_v\ u) = (M\ *_v\ v) \cdot_c u$
using *c3 c4 cscalar-prod-adjoint assms*(*3*) **by** *blast*
hence $v \cdot_c (vec\text{-}of\text{-}basis\text{-}enum\ ((cblinfun\text{-}of\text{-}mat\ (mat\text{-}adjoint\ M)\ *_V\ x)::'a)) = (vec\text{-}of\text{-}basis\text{-}enum\ ((cblinfun\text{-}of\text{-}mat\ M\ *_V\ y)::'b)) \cdot_c u$
using *c1 c2* **by** *simp*
thus $(cblinfun\text{-}of\text{-}mat\ (mat\text{-}adjoint\ M)\ *_V\ x) \cdot_C y = x \cdot_C (cblinfun\text{-}of\text{-}mat\ M\ *_V\ y)$
unfolding *u-def v-def*
by (*simp add*: *cscalar-prod-vec-of-basis-enum*)
qed
qed

lemma *mat-of-cblinfun-compose*:
$mat\text{-}of\text{-}cblinfun\ (F\ o_{CL}\ G) = mat\text{-}of\text{-}cblinfun\ F * mat\text{-}of\text{-}cblinfun\ G$
for $F::'b::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\} \Rightarrow_{CL} 'c::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\}$
and $G::'a::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\} \Rightarrow_{CL} 'b$
by (*smt* (*verit*, *del-insts*) *cblinfun-of-mat-inverse mat-carrier mat-of-cblinfun-def mat-of-cblinfun-inverse cblinfun-of-mat-times mult-carrier-mat*)

lemma *mat-of-cblinfun-scaleC*:
$mat\text{-}of\text{-}cblinfun\ ((a::complex)\ *_C\ F) = a \cdot_m (mat\text{-}of\text{-}cblinfun\ F)$
for $F :: 'a::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\} \Rightarrow_{CL} 'b::\{basis\text{-}enum, complex\text{-}normed\text{-}vector\}$
by (*auto simp add*: *complex-vector.representation-scale mat-of-cblinfun-def*)

lemma *mat-of-cblinfun-scaleR*:
$mat\text{-}of\text{-}cblinfun\ ((a::real)\ *_R\ F) = (complex\text{-}of\text{-}real\ a) \cdot_m (mat\text{-}of\text{-}cblinfun\ F)$
unfolding *scaleR-scaleC* **by** (*rule mat-of-cblinfun-scaleC*)

lemma *mat-of-cblinfun-adj*:
$mat\text{-}of\text{-}cblinfun\ (F*) = mat\text{-}adjoint\ (mat\text{-}of\text{-}cblinfun\ F)$
for $F :: 'a::onb\text{-}enum \Rightarrow_{CL} 'b::onb\text{-}enum$

by (*metis* (*no-types*, *lifting*) *cblinfun-of-mat-inverse map-carrier-mat mat-adjoint-def' mat-carrier cblinfun-of-mat-adjoint mat-of-cblinfun-def mat-of-cblinfun-inverse transpose-carrier-mat*)

lemma *mat-of-cblinfun-vector-to-cblinfun*:
mat-of-cblinfun (*vector-to-cblinfun* ψ)
$=$ *mat-of-cols* (*length* (*canonical-basis* :: $'a$ *list*)) [*vec-of-basis-enum* ψ]
for ψ::$'a$::{*basis-enum*,*complex-normed-vector*}
by (*auto simp*: *mat-of-cols-Cons-index-0 mat-of-cblinfun-def vec-of-basis-enum-def vec-of-list-index*)

lemma *mat-of-cblinfun-proj*:
fixes *a*::$'a$::*onb-enum*
defines $d \equiv$ *length* (*canonical-basis* :: $'a$ *list*)
and *norm2* $\equiv$ (*vec-of-basis-enum a*) $\cdot_c$ (*vec-of-basis-enum a*)
shows *mat-of-cblinfun* (*proj a*) =
1 / norm2 $\cdot_m$ (*mat-of-cols d* [*vec-of-basis-enum a*]
$*$ *mat-of-rows d* [*conjugate* (*vec-of-basis-enum a*)]) (**is** ‹- = *?rhs*›)
proof (*cases a = 0*)
case *False*
have *norm2*: ‹*norm2* = (*norm a*)^2›
by (*simp add*: *cscalar-prod-vec-of-basis-enum norm2-def cdot-square-norm*[*symmetric*, *simplified*])
have [*simp*]: ‹*vec-of-basis-enum a* ∈ *carrier-vec d*›
by (*simp add*: *carrier-vecI d-def*)

have ‹*mat-of-cblinfun* (*proj a*) = *mat-of-cblinfun* (*proj* (*a* /$_R$ *norm a*))›
by (*metis* (*mono-tags*, *opaque-lifting*) *ccspan-singleton-scaleC complex-vector.scale-eq-0-iff nonzero-imp-inverse-nonzero norm-eq-zero scaleR-scaleC scale-left-imp-eq*)
also have ‹... = *mat-of-cblinfun* (*selfbutter* (*a* /$_R$ *norm a*))›
apply (*subst butterfly-eq-proj*)
by (*auto simp add*: *False*)
also have ‹... = *1*/*norm2* $\cdot_m$ *mat-of-cblinfun* (*selfbutter a*)›
apply (*simp add*: *mat-of-cblinfun-scaleC norm2*)
by (*simp add*: *inverse-eq-divide power2-eq-square*)
also have ‹... = *1 / norm2* $\cdot_m$ (*mat-of-cblinfun* (*vector-to-cblinfun a* :: *complex* $\Rightarrow_{CL}$ $'a$) $*$ *mat-adjoint* (*mat-of-cblinfun* (*vector-to-cblinfun a* :: *complex* $\Rightarrow_{CL}$ $'a$)))›
by (*simp add*: *butterfly-def mat-of-cblinfun-compose mat-of-cblinfun-adj*)
also have ‹... = *?rhs*›
by (*simp add*: *mat-of-cblinfun-vector-to-cblinfun mat-adjoint-def flip*: *d-def*)
finally show *?thesis*
by −
next
case *True*
show *?thesis*
apply (*rule eq-matI*)
by (*auto simp*: *True mat-of-cblinfun-zero vec-of-basis-enum-zero scalar-prod-def mat-of-rows-index*

simp flip: *d-def*)
qed

lemma *mat-of-cblinfun-ell2-component*:
fixes *a* :: ‹'*a*::*enum ell2* ⇒$_{CL}$ '*b*::*enum ell2*›
assumes [*simp*]: ‹*i* < *CARD*('*b*)› ‹*j* < *CARD*('*a*)›
shows ‹*mat-of-cblinfun a* $$ (*i*,*j*) = *Rep-ell2* (*a* *$_V$ *ket* (*Enum.enum* ! *j*)) (*Enum.enum* ! *i*)›
proof −
let *?i* = ‹*Enum.enum* ! *i*› **and** *?j* = ‹*Enum.enum* ! *j*› **and** *?aj* = ‹*a* *$_V$ *ket* (*Enum.enum* ! *j*)›
have ‹*Rep-ell2 ?aj* (*Enum.enum* ! *i*) = *vec-of-basis-enum ?aj* $ *i*›
by (*rule vec-of-basis-enum-ell2-component*[*symmetric*], *simp*)
also have ‹... = (*mat-of-cblinfun a* *$_v$ *vec-of-basis-enum* (*ket* (*enum-class.enum* ! *j*) :: '*a ell2*)) $ *i*›
by (*simp add*: *mat-of-cblinfun-cblinfun-apply*)
also have ‹... = (*mat-of-cblinfun a* *$_v$ *unit-vec CARD*('*a*) *j*) $ *i*›
by (*simp add*: *vec-of-basis-enum-ket enum-idx-enum*)
also have ‹... = *mat-of-cblinfun a* $$ (*i*, *j*)›
apply (*subst mat-entry-explicit*[**where** *m*=‹*CARD*('*b*)›])
by (*auto intro*!: *simp*: *canonical-basis-length*)
finally show *?thesis*
by *auto*
qed

lemma *cblinfun-of-mat-mat*:
shows ‹*cblinfun-of-mat* (*mat* (*CARD*('*b*)) (*CARD*('*a*)) *f*) = *explicit-cblinfun* (λ(*r*::'*b*::*enum*) (*c*::'*a*::*enum*). *f* (*enum-idx r*, *enum-idx c*))›
apply (*intro equal-ket basis-enum-eq-vec-of-basis-enumI*)
by (*auto intro*!: *eq-vecI simp*: *enum-idx-correct enum-idx-enum vec-of-basis-enum-ket mat-of-cblinfun-ell2-component canonical-basis-length cblinfun-of-mat-inverse index-row mat-of-cblinfun-cblinfun-apply*)

lemma *mat-of-cblinfun-explicit-cblinfun*:
fixes *f* :: ‹'*a*::*enum* ⇒ '*b*::*enum* ⇒ *complex*›
shows ‹*mat-of-cblinfun* (*explicit-cblinfun f*) = *mat* (*CARD*('*a*)) (*CARD*('*b*)) (λ(*r*,*c*). *f* (*Enum.enum*!*r*) (*Enum.enum*!*c*))›
by (*auto intro*!: *cblinfun-of-mat-inj*[**where** '*a*=‹'*b ell2*› **and** '*b*=‹'*a ell2*›, *THEN inj-onD*]
simp: *cblinfun-of-mat-mat canonical-basis-length enum-idx-correct mat-of-cblinfun-inverse*)

lemma *mat-of-cblinfun-classical-operator*:
fixes *f*::'*a*::*enum* ⇒ '*b*::*enum option*
shows *mat-of-cblinfun* (*classical-operator f*) = *mat* (*CARD*('*b*)) (*CARD*('*a*)) (λ(*r*,*c*). *if f* (*Enum.enum*!*c*) = *Some* (*Enum.enum*!*r*) *then 1 else 0*)
proof −
define *nA* **where** *nA* = *CARD*('*a*)
define *nB* **where** *nB* = *CARD*('*b*)

```
  define BasisA where BasisA = (canonical-basis::'a ell2 list)
  define BasisB where BasisB = (canonical-basis::'b ell2 list)
  have mat-of-cblinfun (classical-operator f) ∈ carrier-mat nB nA
    by (auto simp: canonical-basis-length nA-def nB-def)
  moreover have nA = CARD ('a)
    unfolding nA-def
    by (simp add:)
  moreover have nB = CARD ('b)
    unfolding nB-def
    by (simp add:)
 ultimately have mat-of-cblinfun (classical-operator f) ∈ carrier-mat (CARD('b))
(CARD('a))
    unfolding nA-def nB-def
    by simp
  moreover have (mat-of-cblinfun (classical-operator f))$$(r,c)
  = (mat (CARD('b)) (CARD('a))
    (λ(r,c). if f (Enum.enum!c) = Some (Enum.enum!r) then 1 else 0))$$(r,c)
    if a1: r < CARD('b) and a2: c < CARD('a)
    for r c
  proof−
    have CARD('a) = length (enum-class.enum::'a list)
      using card-UNIV-length-enum[where 'a = 'a] .
    hence x1: BasisA!c = ket ((Enum.enum::'a list)!c)
      unfolding BasisA-def using a2 canonical-basis-ell2-def
        nth-map[where n = c and xs = Enum.enum::'a list and f = ket] by metis
    have cardb: CARD('b) = length (enum-class.enum::'b list)
      using card-UNIV-length-enum[where 'a = 'b] .
    hence x2: BasisB!r = ket ((Enum.enum::'b list)!r)
      unfolding BasisB-def using a1 canonical-basis-ell2-def
        nth-map[where n = r and xs = Enum.enum::'b list and f = ket] by metis
    have inj (map (ket::'b ⇒-))
      by (meson injI ket-injective list.inj-map)
   hence length (Enum.enum::'b list) = length (map (ket::'b ⇒-) (Enum.enum::'b
list))
      by simp
    hence lengthBasisB: CARD('b) = length BasisB
      unfolding BasisB-def canonical-basis-ell2-def using cardb
      by smt
    have v1: (mat-of-cblinfun (classical-operator f))$$(r,c) = 0
      if c1: f (Enum.enum!c) = None
    proof−
      have (classical-operator f) *V ket (Enum.enum!c)
          = (case f (Enum.enum!c) of None ⇒ 0 | Some i ⇒ ket i)
        using classical-operator-ket-finite .
      also have ... = 0
        using c1 by simp
      finally have (classical-operator f) *V ket (Enum.enum!c) = 0 .
      hence *: (classical-operator f) *V BasisA!c = 0
        using x1 by simp
```

hence *is-orthogonal* (*BasisB*!*r*) (*classical-operator f* $*_V$ *BasisA*!*c*)
by *simp*
thus *?thesis*
unfolding *mat-of-cblinfun-def BasisA-def BasisB-def*
by (*smt* (*verit*, *del-insts*) *BasisA-def* ∗ *a1 a2 canonical-basis-length-ell2 complex-vector.representation-zero index-mat*(*1*) *old.prod.case*)
qed
have *v2*: (*mat-of-cblinfun* (*classical-operator f*))$$(*r*,*c*) = *0*
if *c1*: *f* (*Enum.enum*!*c*) = *Some* (*Enum.enum*!*r'*) **and** *c2*: *r*≠*r'*
and *c3*: *r'* < *CARD*('*b*)
for *r'*
proof−
have *x3*: *BasisB*!*r'* = *ket* ((*Enum.enum*::'*b list*)!*r'*)
unfolding *BasisB-def* **using** *cardb c3 canonical-basis-ell2-def*
nth-map[**where** *n* = *r'* **and** *xs* = *Enum.enum*::'*b list* **and** *f* = *ket*]
by *smt*
have *distinct BasisB*
unfolding *BasisB-def*
by *simp*
moreover have *r* < *length BasisB*
using *a1 lengthBasisB* **by** *simp*
moreover have *r'* < *length BasisB*
using *c3 lengthBasisB* **by** *simp*
ultimately have *h1*: *BasisB*!*r* ≠ *BasisB*!*r'*
using *nth-eq-iff-index-eq*[**where** *xs* = *BasisB* **and** *i* = *r* **and** *j* = *r'*] *c2*
by *blast*
have (*classical-operator f*) $*_V$ *ket* (*Enum.enum*!*c*)
= (*case f* (*Enum.enum*!*c*) *of None* ⇒ *0* | *Some i* ⇒ *ket i*)
using *classical-operator-ket-finite* .
also have ... = *ket* (*Enum.enum*!*r'*)
using *c1* **by** *simp*
finally have (*classical-operator f*) $*_V$ *ket* (*Enum.enum*!*c*) = *ket* (*Enum.enum*!*r'*)
.
hence ∗: (*classical-operator f*) $*_V$ *BasisA*!*c* = *BasisB*!*r'*
using *x1 x3* **by** *simp*
moreover have *is-orthogonal* (*BasisB*!*r*) (*BasisB*!*r'*)
using *h1*
using *BasisB-def* ‹*r* < *length BasisB*› ‹*r'* < *length BasisB*› *is-ortho-set-def is-orthonormal nth-mem*
by *metis*
ultimately have *is-orthogonal* (*BasisB*!*r*) (*classical-operator f* $*_V$ *BasisA*!*c*)
by *simp*
thus *?thesis*
unfolding *mat-of-cblinfun-def BasisA-def BasisB-def*
by (*smt* (*z3*) *BasisA-def BasisB-def* ∗ ‹*r* < *length BasisB*› ‹*r'* < *length BasisB*› *a2 canonical-basis-length-ell2 case-prod-conv complex-vector.representation-basis h1 index-mat*(*1*) *is-cindependent-set nth-mem*)
qed
have (*mat-of-cblinfun* (*classical-operator f*))$$(*r*,*c*) = *0*

```
  if b1: f (Enum.enum!c) ≠ Some (Enum.enum!r)
 proof (cases f (Enum.enum!c) = None)
  case True thus ?thesis using v1 by blast
 next
  case False
  hence ∃ R. f (Enum.enum!c) = Some R
   apply (induction f (Enum.enum!c))
    apply simp
   by simp
  then obtain R where R0: f (Enum.enum!c) = Some R
   by blast
  have R ∈ set (Enum.enum::'b list)
   using UNIV-enum by blast
  hence ∃ r'. R = (Enum.enum::'b list)!r' ∧ r' < length (Enum.enum::'b list)
   by (metis in-set-conv-nth)
  then obtain r' where u1: R = (Enum.enum::'b list)!r'
   and u2: r' < length (Enum.enum::'b list)
   by blast
  have R1: f (Enum.enum!c) = Some (Enum.enum!r')
   using R0 u1 by blast
  have Some ((Enum.enum::'b list)!r') ≠ Some ((Enum.enum::'b list)!r)
  proof(rule classical)
   assume ¬(Some ((Enum.enum::'b list)!r') ≠ Some ((Enum.enum::'b list)!r))
   hence Some ((Enum.enum::'b list)!r') = Some ((Enum.enum::'b list)!r)
    by blast
   hence f (Enum.enum!c) = Some ((Enum.enum::'b list)!r)
    using R1 by auto
   thus ?thesis
    using b1 by blast
  qed
  hence ((Enum.enum::'b list)!r') ≠ ((Enum.enum::'b list)!r)
   by simp
  hence r' ≠ r
   by blast
  moreover have r' < CARD('b)
   using u2 cardb by simp
  ultimately show ?thesis using R1 v2[where r' = r'] by blast
 qed
 moreover have (mat-of-cblinfun (classical-operator f))$$(r,c) = 1
  if b1: f (Enum.enum!c) = Some (Enum.enum!r)
 proof−
  have CARD('b) = length (enum-class.enum::'b list)
   using card-UNIV-length-enum[where 'a = 'b].
  hence length (map (ket::'b⇒-) enum-class.enum) = CARD('b)
   by simp
  hence w0: map (ket::'b⇒-) enum-class.enum ! r = ket (enum-class.enum !
r)
   by (simp add: a1)
  have CARD('a) = length (enum-class.enum::'a list)
```

```
    using card-UNIV-length-enum[where 'a = 'a].
  hence length (map (ket::'a⇒-) enum-class.enum) = CARD('a)
    by simp
  hence map (ket::'a⇒-) enum-class.enum ! c = ket (enum-class.enum ! c)
    by (simp add: a2)
  hence (classical-operator f) *V (BasisA!c) = (classical-operator f) *V (ket
(Enum.enum!c))
    unfolding BasisA-def canonical-basis-ell2-def by simp
  also have ... = (case f (enum-class.enum ! c) of None ⇒ 0 | Some x ⇒ ket
x)
    by (rule classical-operator-ket-finite)
  also have ... = BasisB!r
    using w0 b1 by (simp add: BasisB-def canonical-basis-ell2-def)
  finally have w1: (classical-operator f) *V (BasisA!c) = BasisB!r
    by simp
  have (mat-of-cblinfun (classical-operator f))$$(r,c)
    = (BasisB!r) •C (classical-operator f *V (BasisA!c))
    unfolding BasisB-def BasisA-def mat-of-cblinfun-def
     using ‹nA = CARD('a)› ‹nB = CARD('b)› a1 a2 nA-def nB-def apply
auto
   by (metis BasisA-def BasisB-def canonical-basis-length-ell2 cinner-canonical-basis
complex-vector.representation-basis is-cindependent-set nth-mem w1)
  also have ... = (BasisB!r) •C (BasisB!r)
    using w1 by simp
  also have ... = 1
    unfolding BasisB-def
    using ‹nB = CARD('b)› a1 nB-def
    by (simp add: cinner-canonical-basis)
  finally show ?thesis by blast
 qed
 ultimately show ?thesis
  by (simp add: a1 a2)
qed
ultimately show ?thesis
 apply (rule-tac eq-matI) by auto
qed


lemma mat-of-cblinfun-sandwich:
 fixes a :: (-::onb-enum, -::onb-enum) cblinfun
 shows ‹mat-of-cblinfun (sandwich a *V b) = (let a' = mat-of-cblinfun a in a' *
mat-of-cblinfun b * mat-adjoint a')›
 by (simp add: mat-of-cblinfun-compose sandwich-apply Let-def mat-of-cblinfun-adj)


lemma mat-of-cblinfun-one-dim-iso:
 ‹mat-of-cblinfun (one-dim-iso f :: 'a::one-dim⇒CL 'b::one-dim) = mat-of-rows-list
1 [[one-dim-iso f]]›
proof −
```

define c :: *complex* **where** ‹c = *one-dim-iso* f›
have ‹*mat-of-cblinfun* (*one-dim-iso* f :: $'a \Rightarrow_{CL} 'b$) = *mat-of-cblinfun* (c *$_C$ 1 :: $'a \Rightarrow_{CL} 'b$)›
by (*simp add*: *c-def*)
also have ‹... = *smult-mat* c (*mat-of-cblinfun* (1 :: $'a \Rightarrow_{CL} 'b$))›
using *mat-of-cblinfun-scaleC* **by** *blast*
also have ‹... = *smult-mat* c (1_m 1)›
by (*simp add*: *mat-of-cblinfun-1*)
also have ‹... = *mat-of-rows-list* 1 [[c]]›
by (*auto simp*: *mat-of-rows-list-def*)
finally show *?thesis*
by (*simp add*: *c-def*)
qed

lemma *mat-of-cblinfun-times*:
fixes F G :: ‹$'a$::*one-dim* $\Rightarrow_{CL}$ $'b$::*one-dim*›
shows ‹*mat-of-cblinfun* (F * G) = *mat-of-rows-list* 1 [[(*one-dim-iso* F) * (*one-dim-iso* G)]]›
proof −
have ‹*mat-of-cblinfun* (F * G) = *mat-of-cblinfun* (*one-dim-iso* (*one-dim-iso* F * *one-dim-iso* G :: *complex*) :: $'a \Rightarrow_{CL} 'b$)›
by (*metis id-apply mat-of-cblinfun-one-dim-iso one-dim-iso-id one-dim-iso-times*)
also have ‹... = *mat-of-rows-list* 1 [[*one-dim-iso* (*one-dim-iso* F * *one-dim-iso* G :: *complex*)]]›
using *mat-of-cblinfun-one-dim-iso* **by** *blast*
also have ‹... = *mat-of-rows-list* 1 [[*one-dim-iso* F * *one-dim-iso* G :: *complex*]]›
by *simp*
finally show *?thesis*
by −
qed

16.5 Operations on subspaces

lemma *ccspan-gram-schmidt0-invariant*:
defines *basis-enum* ≡ (*basis-enum-of-vec* :: - ⇒ $'a$::{*basis-enum*,*complex-normed-vector*})
defines n ≡ *length* (*canonical-basis* :: $'a$ *list*)
assumes *set ws* ⊆ *carrier-vec n*
shows *ccspan* (*set* (*map basis-enum* (*gram-schmidt0 n ws*))) = *ccspan* (*set* (*map basis-enum ws*))
proof (*transfer fixing*: *n ws basis-enum*)
interpret *complex-vec-space*.
define *gs* **where** *gs* = *gram-schmidt0 n ws*
have *closure* (*cspan* (*set* (*map basis-enum gs*)))
= *cspan* (*set* (*map basis-enum gs*))
apply (*rule closure-finite-cspan*)
by *simp*
also have ... = *cspan* (*basis-enum* ‘ *set gs*)
by *simp*

```
  also have ... = basis-enum ‘ span (set gs)
    unfolding basis-enum-def
    apply (rule basis-enum-of-vec-span[symmetric])
    using n-def apply simp
    by (simp add: assms(3) cof-vec-space.gram-schmidt0-result(1) gs-def)
  also have ... = basis-enum ‘ span (set ws)
    unfolding gs-def
    apply (subst gram-schmidt0-result(4)[where ws=ws, symmetric])
    using assms by auto
  also have ... = cspan (basis-enum ‘ set ws)
    unfolding basis-enum-def
    apply (rule basis-enum-of-vec-span)
    using n-def apply simp
    by (simp add: assms(3))
  also have ... = cspan (set (map basis-enum ws))
    by simp
  also have ... = closure (cspan (set (map basis-enum ws)))
    apply (rule closure-finite-cspan[symmetric])
    by simp
  finally show closure (cspan (set (map basis-enum gs)))
            = closure (cspan (set (map basis-enum ws))).
qed

definition is-subspace-of-vec-list n vs ws =
  (let ws' = gram-schmidt0 n ws in
    ∀ v∈set vs. adjuster n v ws' = − v)

lemma ccspan-leq-using-vec:
  fixes A B :: ‹'a::{basis-enum,complex-normed-vector} list›
  shows ‹(ccspan (set A) ≤ ccspan (set B)) ⟷
    is-subspace-of-vec-list (length (canonical-basis :: 'a list))
      (map vec-of-basis-enum A) (map vec-of-basis-enum B)›
proof −
  define d Av Bv Bo
    where d = length (canonical-basis :: 'a list)
      and Av = map vec-of-basis-enum A
      and Bv = map vec-of-basis-enum B
      and Bo = gram-schmidt0 d Bv
  interpret complex-vec-space d.

  have Av-carrier: set Av ⊆ carrier-vec d
    unfolding Av-def apply auto
    by (simp add: carrier-vecI d-def dim-vec-of-basis-enum')
  have Bv-carrier: set Bv ⊆ carrier-vec d
    unfolding Bv-def apply auto
    by (simp add: carrier-vecI d-def dim-vec-of-basis-enum')
  have Bo-carrier: set Bo ⊆ carrier-vec d
    apply (simp add: Bo-def)
    using Bv-carrier by (rule gram-schmidt0-result(1))
```

```
  have orth-Bo: corthogonal Bo
    apply (simp add: Bo-def)
    using Bv-carrier by (rule gram-schmidt0-result(3))
  have distinct-Bo: distinct Bo
    apply (simp add: Bo-def)
    using Bv-carrier by (rule gram-schmidt0-result(2))

  have ccspan (set A) ≤ ccspan (set B) ⟷ cspan (set A) ⊆ cspan (set B)
    apply (transfer fixing: A B)
    apply (subst closure-finite-cspan, simp)
    by (subst closure-finite-cspan, simp-all)
  also have ... ⟷ span (set Av) ⊆ span (set Bv)
    apply (simp add: Av-def Bv-def)
    apply (subst vec-of-basis-enum-cspan[symmetric], simp add: d-def)
    apply (subst vec-of-basis-enum-cspan[symmetric], simp add: d-def)
    by (metis inj-image-subset-iff inj-on-def vec-of-basis-enum-inverse)
  also have ... ⟷ span (set Av) ⊆ span (set Bo)
    unfolding Bo-def Av-def Bv-def
    apply (subst gram-schmidt0-result(4)[symmetric])
   by (simp-all add: carrier-dim-vec d-def dim-vec-of-basis-enum' image-subset-iff)
  also have ... ⟷ (∀ v∈set Av. adjuster d v Bo = - v)
  proof (intro iffI ballI)
    fix v assume v ∈ set Av and span (set Av) ⊆ span (set Bo)
    then have v ∈ span (set Bo)
      using Av-carrier span-mem by auto
    have adjuster d v Bo + v = 0_v d
      apply (rule adjuster-already-in-span)
      using Av-carrier ‹v ∈ set Av› Bo-carrier orth-Bo
        ‹v ∈ span (set Bo)› by simp-all
    then show adjuster d v Bo = - v
      using Av-carrier Bo-carrier ‹v ∈ set Av›
       by (metis (no-types, lifting) add.inv-equality adjuster-carrier' local.vec-neg
subsetD)
  next
    fix v
    assume adj-minusv: ∀ v∈set Av. adjuster d v Bo = - v
    have ∗: adjuster d v Bo ∈ span (set Bo) if v ∈ set Av for v
      apply (rule adjuster-in-span)
      using Bo-carrier that Av-carrier distinct-Bo by auto
    have v ∈ span (set Bo) if v ∈ set Av for v
      using ∗[OF that] adj-minusv[rule-format, OF that]
      apply auto
      by (metis (no-types, lifting) Av-carrier Bo-carrier adjust-nonzero distinct-Bo
subsetD that uminus-l-inv-vec)
    then show span (set Av) ⊆ span (set Bo)
      apply auto
      by (meson Bo-carrier in-mono span-subsetI subsetI)
  qed
  also have ... = is-subspace-of-vec-list d Av Bv
```

by (*simp add*: *is-subspace-of-vec-list-def d-def Bo-def*)
finally show *ccspan* (*set A*) ≤ *ccspan* (*set B*) ⟷ *is-subspace-of-vec-list d Av Bv*
by *simp*
qed

lemma *cblinfun-image-ccspan-using-vec*:
A ∗$_S$ *ccspan* (*set S*) = *ccspan* (*basis-enum-of-vec* ‘ *set* (*map* ((∗$_v$) (*mat-of-cblinfun A*)) (*map vec-of-basis-enum S*)))
apply (*auto simp*: *cblinfun-image-ccspan image-image*)
by (*metis mat-of-cblinfun-cblinfun-apply vec-of-basis-enum-inverse*)

mk-projector-orthog d L takes a list L of d-dimensional vectors and returns the projector onto the span of L. (Assuming that all vectors in L are orthogonal and nonzero.)

fun *mk-projector-orthog* :: *nat* ⇒ *complex vec list* ⇒ *complex mat* **where**
mk-projector-orthog d [] = *zero-mat d d*
| *mk-projector-orthog d* [*v*] = (*let norm2* = *cscalar-prod v v in smult-mat* (*1*/*norm2*) (*mat-of-cols d* [*v*] ∗ *mat-of-rows d* [*conjugate v*]))
| *mk-projector-orthog d* (*v*#*vs*) = (*let norm2* = *cscalar-prod v v in smult-mat* (*1*/*norm2*) (*mat-of-cols d* [*v*] ∗ *mat-of-rows d* [*conjugate v*]) + *mk-projector-orthog d vs*)

lemma *mk-projector-orthog-correct*:
fixes *S* :: *'a*::*onb-enum list*
defines *d* ≡ *length* (*canonical-basis* :: *'a list*)
assumes *ortho*: *is-ortho-set* (*set S*) **and** *distinct*: *distinct S*
shows *mk-projector-orthog d* (*map vec-of-basis-enum S*)
= *mat-of-cblinfun* (*Proj* (*ccspan* (*set S*)))
proof −
define *Snorm* **where** *Snorm* = *map* (λ*s*. *s* /$_R$ *norm s*) *S*

have *distinct Snorm*
proof (*insert ortho distinct*, *unfold Snorm-def*, *induction S*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons s S*)
then have *is-ortho-set* (*set S*) **and** *distinct S*
unfolding *is-ortho-set-def* **by** *auto*
note *IH* = *Cons.IH*[*OF this*]
have *s* /$_R$ *norm s* ∉ (λ*s*. *s* /$_R$ *norm s*) ‘ *set S*
proof *auto*
fix *s'* **assume** *s'* ∈ *set S* **and** *same*: *s* /$_R$ *norm s* = *s'* /$_R$ *norm s'*
with *Cons.prems* **have** *s* ≠ *s'* **by** *auto*
have *s* ≠ *0*
by (*metis Cons.prems*(*1*) *is-ortho-set-def list.set-intros*(*1*))

```
    then have 0 ≠ (s /R norm s) ·C (s /R norm s)
      by simp
    also have ‹(s /R norm s) ·C (s /R norm s) = (s /R norm s) ·C (s' /R norm
s')›
      by (simp add: same)
    also have ‹(s /R norm s) ·C (s' /R norm s') = (s ·C s') / (norm s * norm
s')›
      by (simp add: scaleR-scaleC divide-inverse-commute)
    also from ‹s' ∈ set S› ‹s ≠ s'› have ... = 0
      using Cons.prems unfolding is-ortho-set-def by simp
    finally show False
      by simp
  qed
  then show ?case
    using IH by simp
 qed

 have norm-Snorm: norm s = 1 if s ∈ set Snorm for s
  using that ortho unfolding Snorm-def is-ortho-set-def apply auto
  by (metis left-inverse norm-eq-zero)

 have ortho-Snorm: is-ortho-set (set Snorm)
  unfolding is-ortho-set-def
 proof (intro conjI ballI impI)
  fix x y
  show 0 ∉ set Snorm
    using norm-Snorm[of 0] by auto
  assume x ∈ set Snorm and y ∈ set Snorm and x ≠ y
  from ‹x ∈ set Snorm›
  obtain x' where x: x = x' /R norm x' and x': x' ∈ set S
    unfolding Snorm-def by auto
  from ‹y ∈ set Snorm›
  obtain y' where y: y = y' /R norm y' and y': y' ∈ set S
    unfolding Snorm-def by auto
  from ‹x ≠ y› x y have ‹x' ≠ y'› by auto
  with x' y' ortho have cinner x' y' = 0
    unfolding is-ortho-set-def by auto
  then show cinner x y = 0
    unfolding x y scaleR-scaleC by auto
 qed

 have inj-butter: inj-on selfbutter (set Snorm)
 proof (rule inj-onI)
  fix x y
  assume x ∈ set Snorm and y ∈ set Snorm
  assume selfbutter x = selfbutter y
  then obtain c where xcy: x = c *C y and cmod c = 1
    using inj-selfbutter-upto-phase by auto
  have 0 ≠ cmod (cinner x x)
```

```
    using ‹x ∈ set Snorm› norm-Snorm
    by force
  also have cmod (cinner x x) = cmod (c * (x ·C y))
    apply (subst (2) xcy) by simp
  also have ... = cmod (x ·C y)
    using ‹cmod c = 1› by (simp add: norm-mult)
  finally have (x ·C y) ≠ 0
    by simp
  then show x = y
    using ortho-Snorm ‹x ∈ set Snorm› ‹y ∈ set Snorm›
    unfolding is-ortho-set-def by auto
qed

from ‹distinct Snorm› inj-butter
have distinct': distinct (map selfbutter Snorm)
  unfolding distinct-map by simp

have Span-Snorm: ccspan (set Snorm) = ccspan (set S)
  apply (transfer fixing: Snorm S)
  apply (simp add: scaleR-scaleC Snorm-def)
  apply (subst complex-vector.span-image-scale)
  using is-ortho-set-def ortho by fastforce+

have mk-projector-orthog d (map vec-of-basis-enum S)
    = mat-of-cblinfun (sum-list (map selfbutter Snorm))
  unfolding Snorm-def
proof (induction S)
  case Nil
  show ?case
    by (simp add: d-def mat-of-cblinfun-zero)
next
  case (Cons a S)
  define sumS where sumS = sum-list (map selfbutter (map (λs. s /R norm s)
S))
  with Cons have IH: mk-projector-orthog d (map vec-of-basis-enum S)
            = mat-of-cblinfun sumS
    by simp

  define factor where factor = inverse ((complex-of-real (norm a))²)
  have factor': factor = 1 / (vec-of-basis-enum a ·c vec-of-basis-enum a)
    unfolding factor-def cscalar-prod-vec-of-basis-enum
    by (simp add: inverse-eq-divide power2-norm-eq-cinner)

  have mk-projector-orthog d (map vec-of-basis-enum (a # S))
      = factor ·m (mat-of-cols d [vec-of-basis-enum a]
            * mat-of-rows d [conjugate (vec-of-basis-enum a)])
        + mat-of-cblinfun sumS
    apply (cases S)
     apply (auto simp add: factor' sumS-def d-def mat-of-cblinfun-zero)[1]
```

```
    by (auto simp add: IH[symmetric] factor' d-def)

  also have ... = factor ·_m (mat-of-cols d [vec-of-basis-enum a] *
      mat-adjoint (mat-of-cols d [vec-of-basis-enum a])) + mat-of-cblinfun sumS
    apply (rule arg-cong[where f=λx. - ·_m (- * x) + -])
    apply (rule mat-eq-iff[THEN iffD2])
    apply (auto simp add: mat-adjoint-def)
    apply (subst mat-of-rows-index) apply auto
    apply (subst mat-of-rows-index) apply auto
    apply (subst mat-of-cols-index) apply auto
    by (simp add: assms(1) dim-vec-of-basis-enum')
  also have ... = mat-of-cblinfun (selfbutter (a /_R norm a)) + mat-of-cblinfun
sumS
      apply (simp add: butterfly-scaleR-left butterfly-scaleR-right power-inverse
mat-of-cblinfun-scaleR factor-def)
    apply (simp add: butterfly-def mat-of-cblinfun-compose
        mat-of-cblinfun-adj mat-of-cblinfun-vector-to-cblinfun d-def)
   by (simp add: mat-of-cblinfun-compose mat-of-cblinfun-adj mat-of-cblinfun-vector-to-cblinfun
mat-of-cblinfun-scaleC power2-eq-square)
  finally show ?case
    by (simp add: mat-of-cblinfun-plus sumS-def)
 qed
 also have ... = mat-of-cblinfun (∑ s∈set Snorm. selfbutter s)
   by (metis distinct' distinct-map sum.distinct-set-conv-list)
 also have ... = mat-of-cblinfun (∑ s∈set Snorm. proj s)
   apply (rule arg-cong[where f=mat-of-cblinfun])
   apply (rule sum.cong[OF refl])
   apply (rule butterfly-eq-proj)
   using norm-Snorm by simp
 also have ... = mat-of-cblinfun (Proj (ccspan (set Snorm)))
   apply (rule arg-cong[where f=mat-of-cblinfun])
   using ortho-Snorm ‹distinct Snorm› apply (induction Snorm)
    apply auto
   apply (subst Proj-orthog-ccspan-insert[where Y=‹set -›])
   by (auto simp: is-ortho-set-def)
 also have ... = mat-of-cblinfun (Proj (ccspan (set S)))
   unfolding Span-Snorm by simp
 finally show ?thesis
   by −
qed

definition ‹mk-projector d vs = mk-projector-orthog d (gram-schmidt0 d vs)›

lemma mat-of-cblinfun-Proj-ccspan:
 fixes S :: ‹'a::onb-enum list›
 shows ‹mat-of-cblinfun (Proj (ccspan (set S))) = mk-projector (length (canonical-basis
:: 'a list)) (map vec-of-basis-enum S)›
proof−
 define d gs
```

```
    where d = length (canonical-basis :: 'a list)
      and gs = gram-schmidt0 d (map vec-of-basis-enum S)
  interpret complex-vec-space d.
  have gs-dim: x ∈ set gs ⟹ dim-vec x = d for x
   by (smt carrier-vecD carrier-vec-dim-vec d-def dim-vec-of-basis-enum' ex-map-conv
gram-schmidt0-result(1) gs-def subset-code(1))
  have ortho-gs: is-ortho-set (set (map basis-enum-of-vec gs :: 'a list))
    apply (subst corthogonal-vec-of-basis-enum[THEN iffD1], auto)
   by (smt carrier-dim-vec cof-vec-space.gram-schmidt0-result(1) d-def dim-vec-of-basis-enum'
gram-schmidt0-result(3) gs-def imageE map-idI map-map o-apply set-map subset-code(1)
basis-enum-of-vec-inverse)
  have distinct-gs: distinct (map basis-enum-of-vec gs :: 'a list)
   by (metis (mono-tags, opaque-lifting) carrier-vec-dim-vec cof-vec-space.gram-schmidt0-result(2)
d-def dim-vec-of-basis-enum' distinct-map gs-def gs-dim image-iff inj-on-inverseI
set-map subsetI basis-enum-of-vec-inverse)

  have mk-projector-orthog d gs
      = mk-projector-orthog d (map vec-of-basis-enum (map basis-enum-of-vec gs ::
'a list))
    apply simp
    apply (subst map-cong[where ys=gs and g=id], simp)
    using gs-dim by (auto intro!: vec-of-basis-enum-inverse simp: d-def)
  also have ... = mat-of-cblinfun (Proj (ccspan (set (map basis-enum-of-vec gs ::
'a list))))
    unfolding d-def
    apply (subst mk-projector-orthog-correct)
    using ortho-gs distinct-gs by auto
  also have ... = mat-of-cblinfun (Proj (ccspan (set S)))
    apply (rule arg-cong[where f=λx. mat-of-cblinfun (Proj x)])
    unfolding gs-def d-def
    apply (subst ccspan-gram-schmidt0-invariant)
    by (auto simp add: carrier-vecI dim-vec-of-basis-enum')
  finally show ?thesis
    by (simp add: d-def gs-def mk-projector-def)
qed

unbundle no jnf-syntax and no cblinfun-syntax

end
```

17 *Cblinfun-Code* – Support for code generation

This theory provides support for code generation involving on complex vector spaces and bounded operators (e.g., types *cblinfun* and *ell2*). To fully support code generation, in addition to importing this theory, one need to activate support for code generation (import theory *Jordan-Normal-Form.Matrix-Impl*) and for real and complex numbers (import theory *Real-Impl.Real-Impl* for support of reals of the form $a + b * sqrt\ c$ or *Algebraic-Numbers.Real-Factorization*

(much slower) for support of algebraic reals; support of complex numbers comes "for free").

The builtin support for real and complex numbers (in *Complex-Main*) is not sufficient because it does not support the computation of square-roots which are used in the setup below.

It is also recommended to import *HOL−Library.Code-Target-Numeral* for faster support of nats and integers.

theory *Cblinfun-Code*
  imports
    Cblinfun-Matrix Containers.Set-Impl Jordan-Normal-Form.Matrix-Kernel
begin

no-notation *Lattice.meet* (**infixl** ‹⊓ı› *70*)
no-notation *Lattice.join* (**infixl** ‹⊔ı› *65*)
hide-const (**open**) *Coset.kernel*
hide-const (**open**) *Matrix-Kernel.kernel*
hide-const (**open**) *Order.bottom Order.top*

unbundle *lattice-syntax*
unbundle *jnf-syntax*
unbundle *cblinfun-syntax*

17.1 Code equations for cblinfun operators

In this subsection, we define the code for all operations involving only operators (no combinations of operators/vectors/subspaces)

The following lemma registers cblinfun as an abstract datatype with constructor *cblinfun-of-mat*. That means that in generated code, all cblinfun operators will be represented as *cblinfun-of-mat X* where X is a matrix. In code equations for operations involving operators (e.g., +), we can then write the equation directly in terms of matrices by writing, e.g., *mat-of-cblinfun* $(A + B)$ in the lhs, and in the rhs we define the matrix that corresponds to the sum of A,B. In the rhs, we can access the matrices corresponding to A,B by writing *mat-of-cblinfun B*. (See, e.g., lemma *mat-of-cblinfun-plus*.)
See [2] for more information on [*code abstype*].

declare *mat-of-cblinfun-inverse* [*code abstype*]

declare *mat-of-cblinfun-plus*[*code*]
  — Code equation for addition of cblinfuns

declare *mat-of-cblinfun-id*[*code*]
  — Code equation for computing the identity operator

declare *mat-of-cblinfun-1*[*code*]
  — Code equation for computing the one-dimensional identity

declare *mat-of-cblinfun-zero*[*code*]
— Code equation for computing the zero operator

declare *mat-of-cblinfun-uminus*[*code*]
— Code equation for computing the unary minus on cblinfun's

declare *mat-of-cblinfun-minus*[*code*]
— Code equation for computing the difference of cblinfun's

declare *mat-of-cblinfun-classical-operator*[*code*]
— Code equation for computing the "classical operator"

declare *mat-of-cblinfun-explicit-cblinfun*[*code*]
— Code equation for computing the *explicit-cblinfun*

declare *mat-of-cblinfun-compose*[*code*]
— Code equation for computing the composition/product of cblinfun's

declare *mat-of-cblinfun-scaleC*[*code*]
— Code equation for multiplication with complex scalar

declare *mat-of-cblinfun-scaleR*[*code*]
— Code equation for multiplication with real scalar

declare *mat-of-cblinfun-adj*[*code*]
— Code equation for computing the adj

This instantiation defines a code equation for equality tests for cblinfun.

instantiation *cblinfun* :: (*onb-enum*,*onb-enum*) *equal* **begin**
definition [*code*]: *equal-cblinfun M N* $\longleftrightarrow$ *mat-of-cblinfun M* = *mat-of-cblinfun N*

for *M N* :: $'a \Rightarrow_{CL} 'b$
instance
apply *intro-classes*
unfolding *equal-cblinfun-def*
using *mat-of-cblinfun-inj injD* **by** *fastforce*
end

17.2 Vectors

In this section, we define code for operations on vectors. As with operators above, we do this by using an isomorphism between finite vectors (i.e., types T of sort *complex-vector*) and the type *complex vec* from `Jordan_Normal_Form`. We have developed such an isomorphism in theory *Cblinfun-Matrix* for any type T of sort *onb-enum* (i.e., any type with a

finite canonical orthonormal basis) as was done above for bounded operators. Unfortunately, we cannot declare code equations for a type class, code equations must be related to a specific type constructor. So we give code definition only for vectors of type *'a ell2* (where *'a* must be of sort *enum* to make make sure that *'a ell2* is finite dimensional).

The isomorphism between *'a ell2* is given by the constants *ell2-of-vec* and *vec-of-ell2* which are copies of the more general *basis-enum-of-vec* and *vec-of-basis-enum* but with a more restricted type to be usable in our code equations.

definition *ell2-of-vec* :: *complex vec* ⇒ *'a*::*enum ell2* **where** *ell2-of-vec* = *basis-enum-of-vec*
definition *vec-of-ell2* :: *'a*::*enum ell2* ⇒ *complex vec* **where** *vec-of-ell2* = *vec-of-basis-enum*

The following theorem registers the isomorphism *ell2-of-vec*/*vec-of-ell2* for code generation. From now on, code for operations on - *ell2* can be expressed by declarations such as *vec-of-ell2* (*f a b*) = *g* (*vec-of-ell2 a*) (*vec-of-ell2 b*) if the operation f on - *ell2* corresponds to the operation g on *complex vec*.

lemma *vec-of-ell2-inverse* [*code abstype*]:
ell2-of-vec (*vec-of-ell2 B*) = *B*
unfolding *ell2-of-vec-def vec-of-ell2-def*
by (*rule vec-of-basis-enum-inverse*)

This instantiation defines a code equation for equality tests for ell2.

instantiation *ell2* :: (*enum*) *equal* **begin**
definition [*code*]: *equal-ell2 M N* ⟷ *vec-of-ell2 M* = *vec-of-ell2 N*
for *M N* :: *'a*::*enum ell2*
instance
apply *intro-classes*
unfolding *equal-ell2-def*
by (*metis vec-of-ell2-inverse*)
end

lemma *vec-of-ell2-carrier-vec*[*simp*]: ‹*vec-of-ell2 v* ∈ *carrier-vec CARD*('*a*)› **for** *v* :: ‹*'a*::*enum ell2*›
using *vec-of-basis-enum-carrier-vec*[*of v*]
apply (*simp only*: *canonical-basis-length canonical-basis-length-ell2*)
by (*simp add*: *vec-of-ell2-def*)

lemma *vec-of-ell2-zero*[*code*]:
— Code equation for computing the zero vector
vec-of-ell2 (*0*::*'a*::*enum ell2*) = *zero-vec* (*CARD*('*a*))
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-zero*)

lemma *vec-of-ell2-ket*[*code*]:
— Code equation for computing a standard basis vector
vec-of-ell2 (*ket i*) = *unit-vec* (*CARD*('*a*)) (*enum-idx i*)
for *i*::*'a*::*enum*
using *vec-of-ell2-def vec-of-basis-enum-ket* **by** *metis*

lemma *vec-of-ell2-scaleC*[*code*]:
— Code equation for multiplying a vector with a complex scalar
vec-of-ell2 (*scaleC a* ψ) = *smult-vec a* (*vec-of-ell2* ψ)
for ψ :: *'a*::*enum ell2*
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-scaleC*)

lemma *vec-of-ell2-scaleR*[*code*]:
— Code equation for multiplying a vector with a real scalar
vec-of-ell2 (*scaleR a* ψ) = *smult-vec* (*complex-of-real a*) (*vec-of-ell2* ψ)
for ψ :: *'a*::*enum ell2*
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-scaleR*)

lemma *ell2-of-vec-plus*[*code*]:
— Code equation for adding vectors
vec-of-ell2 (*x* + *y*) = (*vec-of-ell2 x*) + (*vec-of-ell2 y*) **for** *x y* :: *'a*::*enum ell2*
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-add*)

lemma *ell2-of-vec-minus*[*code*]:
— Code equation for subtracting vectors
vec-of-ell2 (*x* − *y*) = (*vec-of-ell2 x*) − (*vec-of-ell2 y*) **for** *x y* :: *'a*::*enum ell2*
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-minus*)

lemma *ell2-of-vec-uminus*[*code*]:
— Code equation for negating a vector
vec-of-ell2 (− *y*) = − (*vec-of-ell2 y*) **for** *y* :: *'a*::*enum ell2*
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-uminus*)

lemma *cinner-ell2-code* [*code*]: *cinner* ψ φ = *cscalar-prod* (*vec-of-ell2* φ) (*vec-of-ell2* ψ)
— Code equation for the inner product of vectors
by (*simp add*: *cscalar-prod-vec-of-basis-enum vec-of-ell2-def*)

lemma *norm-ell2-code* [*code*]:
— Code equation for the norm of a vector
norm ψ = *norm-vec* (*vec-of-ell2* ψ)
by (*simp add*: *norm-vec-of-basis-enum vec-of-ell2-def*)

lemma *times-ell2-code*[*code*]:
— Code equation for the product in the algebra of one-dimensional vectors
fixes ψ φ :: *'a*::{*CARD-1*,*enum*} *ell2*
shows *vec-of-ell2* (ψ ∗ φ)
= *vec-of-list* [*vec-index* (*vec-of-ell2* ψ) *0* ∗ *vec-index* (*vec-of-ell2* φ) *0*]
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-times*)

lemma *divide-ell2-code*[*code*]:
— Code equation for the product in the algebra of one-dimensional vectors
fixes ψ φ :: *'a*::{*CARD-1*,*enum*} *ell2*
shows *vec-of-ell2* (ψ / φ)

= *vec-of-list* [*vec-index* (*vec-of-ell2* ψ) *0* / *vec-index* (*vec-of-ell2* φ) *0*]
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-divide*)

lemma *inverse-ell2-code*[*code*]:
— Code equation for the product in the algebra of one-dimensional vectors
fixes ψ :: *'a*::{*CARD-1*,*enum*} *ell2*
shows *vec-of-ell2* (*inverse* ψ)
= *vec-of-list* [*inverse* (*vec-index* (*vec-of-ell2* ψ) *0*)]
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-to-inverse*)

lemma *one-ell2-code*[*code*]:
— Code equation for the unit in the algebra of one-dimensional vectors
vec-of-ell2 (*1* :: *'a*::{*CARD-1*,*enum*} *ell2*) = *vec-of-list* [*1*]
by (*simp add*: *vec-of-ell2-def vec-of-basis-enum-1*)

17.3 Vector/Matrix

We proceed to give code equations for operations involving both operators (cblinfun) and vectors. As explained above, we have to restrict the equations to vectors of type *'a ell2* even though the theory is available for any type of class *onb-enum*. As a consequence, we run into an additional technicality now. For example, to define a code equation for applying an operator to a vector, we might try to give the following lemma:

lemma *cblinfun-apply-ell2*[*code*]: *vec-of-ell2* (*M* $*_V$ *x*) = (*mult-mat-vec* (*mat-of-cblinfun M*) (*vec-of-ell2 x*)) **by** (*simp add*: *mat-of-cblinfun-cblinfun-apply vec-of-ell2-def*)

Unfortunately, this does not work, Isabelle produces the warning "Projection as head in equation", most likely due to the fact that the type of ($*_V$) in the equation is less general than the type of ($*_V$) (it is restricted to *ell2*). We overcome this problem by defining a constant *cblinfun-apply-ell2* which is equal to ($*_V$) but has a more restricted type. We then instruct the code generation to replace occurrences of ($*_V$) by *cblinfun-apply-ell2* (where possible), and we add code generation for *cblinfun-apply-ell2* instead of ($*_V$).

definition *cblinfun-apply-ell2* :: *'a ell2* $\Rightarrow_{CL}$ *'b ell2* $\Rightarrow$ *'a ell2* $\Rightarrow$ *'b ell2*
where [*code del*, *code-abbrev*]: *cblinfun-apply-ell2* = ($*_V$)
— *code-abbrev* instructs the code generation to replace the rhs ($*_V$) by the lhs *cblinfun-apply-ell2* before starting the actual code generation.

lemma *cblinfun-apply-ell2*[*code*]:
— Code equation for *cblinfun-apply-ell2*, i.e., for applying an operator to an *ell2* vector
vec-of-ell2 (*cblinfun-apply-ell2 M x*) = (*mult-mat-vec* (*mat-of-cblinfun M*) (*vec-of-ell2 x*))
by (*simp add*: *cblinfun-apply-ell2-def mat-of-cblinfun-cblinfun-apply vec-of-ell2-def*)

For the constant *vector-to-cblinfun* (canonical isomorphism from vectors to operators), we have the same problem and define a constant *vector-to-cblinfun-code*

with more restricted type

definition *vector-to-cblinfun-code* :: *'a ell2* ⇒ *'b*::*one-dim* ⇒$_{CL}$ *'a ell2* **where**
[*code del*,*code-abbrev*]: *vector-to-cblinfun-code* = *vector-to-cblinfun*
— *code-abbrev* instructs the code generation to replace the rhs *vector-to-cblinfun* by the lhs *vector-to-cblinfun-code* before starting the actual code generation.

lemma *vector-to-cblinfun-code*[*code*]:
— Code equation for translating a vector into an operation (single-column matrix)
mat-of-cblinfun (*vector-to-cblinfun-code* *ψ*) = *mat-of-cols* (*CARD*(*'a*)) [*vec-of-ell2* *ψ*]
for *ψ*::*'a*::*enum ell2*
by (*simp add*: *mat-of-cblinfun-vector-to-cblinfun vec-of-ell2-def vector-to-cblinfun-code-def*)

definition *butterfly-code* :: ‹*'a ell2* ⇒ *'b ell2* ⇒ *'b ell2* ⇒$_{CL}$ *'a ell2*›
where [*code del*, *code-abbrev*]: ‹*butterfly-code* = *butterfly*›
lemma *butterfly-code*[*code*]: ‹*mat-of-cblinfun* (*butterfly-code s t*)
= *mat-of-cols* (*CARD*(*'a*)) [*vec-of-ell2 s*] ∗ *mat-of-rows* (*CARD*(*'b*)) [*map-vec cnj* (*vec-of-ell2 t*)]›
for *s* :: ‹*'a*::*enum ell2*› **and** *t* :: ‹*'b*::*enum ell2*›
by (*simp add*: *butterfly-code-def butterfly-def vector-to-cblinfun-code mat-of-cblinfun-compose mat-of-cblinfun-adj mat-adjoint-def map-map-vec-cols flip*: *vector-to-cblinfun-code-def map-vec-conjugate*[*abs-def*])

17.4 Subspaces

In this section, we define code equations for handling subspaces, i.e., values of type *'a ccsubspace*. We choose to computationally represent a subspace by a list of vectors that span the subspace. That is, if *vecs* are vectors (type *complex vec*), *SPAN vecs* is defined to be their span. Then the code generation can simply represent all subspaces in this form, and we need to define the operations on subspaces in terms of list of vectors (e.g., the closed union of two subspaces would be computed as the concatenation of the two lists, to give one of the simplest examples).

To support this, *SPAN* is declared as a "*code-datatype*". (Not as an abstract datatype like *cblinfun-of-mat*/*mat-of-cblinfun* because that would require *SPAN* to be injective.)

Then all code equations for different operations need to be formulated as functions of values of the form *SPAN x*. (E.g., *SPAN x* + *SPAN y* = *SPAN* (. . .).)

definition [*code del*]: *SPAN x* = (*let n* = *length* (*canonical-basis* :: *'a*::*onb-enum list*) *in*
ccspan (*basis-enum-of-vec* ‘ *Set.filter* (λ*v*. *dim-vec v* = *n*) (*set x*)) :: *'a ccsubspace*)
— The SPAN of vectors x, as a *ccsubspace*. We filter out vectors of the wrong dimension because *SPAN* needs to have well-defined behavior even in cases that would not actually occur in an execution.
code-datatype *SPAN*

We first declare code equations for *Proj*, i.e., for turning a subspace into a projector. This means, we would need a code equation of the form *mat-of-cblinfun* (*Proj* (*SPAN S*)) = ... However, this equation is not accepted by the code generation for reasons we do not understand. But if we define an auxiliary constant *mat-of-cblinfun-Proj-code* that stands for *mat-of-cblinfun* (*Proj* -), define a code equation for *mat-of-cblinfun-Proj-code*, and then define a code equation for *mat-of-cblinfun* (*Proj S*) in terms of *mat-of-cblinfun-Proj-code*, Isabelle accepts the code equations.

definition *mat-of-cblinfun-Proj-code S = mat-of-cblinfun* (*Proj S*)
declare *mat-of-cblinfun-Proj-code-def*[*symmetric*, *code*]

lemma *mat-of-cblinfun-Proj-code-code*[*code*]:
— Code equation for computing a projector onto a set S of vectors. We first make the vectors S into an orthonormal basis using the Gram-Schmidt procedure and then compute the projector as the sum of the "butterflies" $x * x*$ of the vectors $x \in S$ (done by *mk-projector*).
mat-of-cblinfun-Proj-code (*SPAN S* :: *'a*::*onb-enum ccsubspace*) =
(*let d = length* (*canonical-basis* :: *'a list*) *in mk-projector d* (*filter* (*λv. dim-vec v = d*) *S*))
proof −
have ∗: *map-option vec-of-basis-enum* (*if dim-vec x = length* (*canonical-basis* :: *'a list*) *then Some* (*basis-enum-of-vec x* :: *'a*) *else None*)
= (*if dim-vec x = length* (*canonical-basis* :: *'a list*) *then Some x else None*)
for *x*
by *auto*
show *?thesis*
unfolding *SPAN-def mat-of-cblinfun-Proj-code-def*
using *mat-of-cblinfun-Proj-ccspan*[**where** *S* =
map basis-enum-of-vec (*filter* (*λv. dim-vec v = length* (*canonical-basis* :: *'a list*))) *S*) :: *'a list*]
apply (*simp only*: *Let-def map-filter-map-filter filter-set image-set map-map-filter o-def*)
unfolding ∗
by (*simp add*: *map-filter-map-filter*[*symmetric*])
qed

lemma *top-ccsubspace-code*[*code*]:
— Code equation for ⊤, the subspace containing everything. Top is represented as the span of the standard basis vectors.
(*top*::*'a ccsubspace*) =
(*let n = length* (*canonical-basis* :: *'a*::*onb-enum list*) *in SPAN* (*unit-vecs n*))
unfolding *SPAN-def*
apply (*simp only*: *index-unit-vec Let-def map-filter-map-filter filter-set image-set map-map-filter*
map-filter-map o-def unit-vecs-def)
apply (*simp add*: *basis-enum-of-vec-unit-vec*)
apply (*subst nth-image*)
by *auto*

lemma *bot-as-span*[*code*]:
— Code equation for $\bot$, the subspace containing everything. Top is represented as the span of the standard basis vectors.
(*bot*::'*a*::*onb-enum ccsubspace*) = *SPAN* []
unfolding *SPAN-def* **by** (*auto simp*:)

lemma *sup-spans*[*code*]:
— Code equation for the join (lub) of two subspaces (union of the generating lists)
SPAN A ⊔ *SPAN B* = *SPAN* (*A* @ *B*)
by (*simp add*: *SPAN-def ccspan-union image-Un filter-Un Let-def del*: *Set.filter-eq*)

We do not need an equation for (+) because (+) is defined in terms of (⊔) (for *ccsubspace*), thus the code generation automatically computes (+) in terms of the code for (⊔)

definition [*code del*,*code-abbrev*]: *Span-code* (*S*::'*a*::*enum ell2 set*) = (*ccspan S*)
— A copy of *ccspan* with restricted type. For analogous reasons as *cblinfun-apply-ell2*, see there for explanations

lemma *span-Set-Monad*[*code*]: *Span-code* (*Set-Monad l*) = (*SPAN* (*map vec-of-ell2 l*))
— Code equation for the span of a finite set. (*Set-Monad* is a datatype constructor that represents sets as lists in the computation.)
apply (*simp add*: *Span-code-def SPAN-def Let-def flip*: *ell2-of-vec-def*)
apply (*rule arg-cong* [*of* - - *ccspan*])
apply (*auto simp add*: *vec-of-ell2-inverse*)
apply (*metis* (*mono-tags*, *lifting*) *carrier-vecD imageI mem-Collect-eq vec-of-ell2-carrier-vec vec-of-ell2-inverse*)
done

This instantiation defines a code equation for equality tests for *ccsubspace*. The actual code for equality tests is given below (lemma *equal-ccsubspace-code*).

instantiation *ccsubspace* :: (*onb-enum*) *equal* **begin**
definition [*code del*]: *equal-ccsubspace* (*A*::'*a ccsubspace*) *B* = (*A*=*B*)
instance apply *intro-classes* **unfolding** *equal-ccsubspace-def* **by** *simp*
end

lemma *leq-ccsubspace-code*[*code*]:
— Code equation for deciding inclusion of one space in another. Uses the constant *is-subspace-of-vec-list* which implements the actual computation by checking for each generator of A whether it is in the span of B (by orthogonal projection onto an orthonormal basis of B which is computed using Gram-Schmidt).
SPAN A ≤ (*SPAN B* :: '*a*::*onb-enum ccsubspace*)
⟷ (*let d* = *length* (*canonical-basis* :: '*a list*) *in*
is-subspace-of-vec-list d
(*filter* (λ*v*. *dim-vec v* = *d*) *A*)
(*filter* (λ*v*. *dim-vec v* = *d*) *B*))

proof −
define d A' B' **where** d = *length* (*canonical-basis* :: *'a list*)
and A' = *filter* (λ*v*. *dim-vec* v = d) A
and B' = *filter* (λ*v*. *dim-vec* v = d) B

show *?thesis*
unfolding *SPAN-def d-def*[*symmetric*] *filter-set Let-def*
A'-def[*symmetric*] *B'-def*[*symmetric*] *image-set*
apply (*subst ccspan-leq-using-vec*)
unfolding *d-def*[*symmetric*] *map-map o-def*
apply (*subst map-cong*[**where** *xs*=A', *OF refl*])
apply (*rule basis-enum-of-vec-inverse*)
apply (*simp add*: *A'-def d-def*)
apply (*subst map-cong*[**where** *xs*=B', *OF refl*])
apply (*rule basis-enum-of-vec-inverse*)
by (*simp-all add*: *B'-def d-def*)
qed

lemma *equal-ccsubspace-code*[*code*]:
— Code equation for equality test. By checking mutual inclusion (for which we have code by the preceding code equation).
HOL.equal (*A*::- *ccsubspace*) B = ($A \leq B \wedge B \leq A$)
unfolding *equal-ccsubspace-def* **by** *auto*

lemma *cblinfun-image-code*[*code*]:
— Code equation for applying an operator A to a subspace. Simply by multiplying each generator with A
$A *_S$ *SPAN* S = (*let* d = *length* (*canonical-basis* :: *'a list*) *in*
SPAN (*map* (*mult-mat-vec* (*mat-of-cblinfun* A))
(*filter* (λ*v*. *dim-vec* v = d) S)))
for *A*::*'a*::*onb-enum* ⇒$_{CL}$ *'b*::*onb-enum*
proof −
define *dA dB S'*
where *dA* = *length* (*canonical-basis* :: *'a list*)
and *dB* = *length* (*canonical-basis* :: *'b list*)
and *S'* = *filter* (λ*v*. *dim-vec* v = *dA*) S

have *cblinfun-image* A (*SPAN* S) = $A *_S$ *ccspan* (*set* (*map basis-enum-of-vec* *S'*))
unfolding *SPAN-def dA-def*[*symmetric*] *Let-def S'-def filter-set*
by *simp*
also have ... = *ccspan* ((λ*x*. *basis-enum-of-vec*
(*mat-of-cblinfun* $A *_v$ *vec-of-basis-enum* (*basis-enum-of-vec* x :: *'a*))) ‘ *set* *S'*)
apply (*subst cblinfun-image-ccspan-using-vec*)
by (*simp add*: *image-image*)
also have ... = *ccspan* ((λ*x*. *basis-enum-of-vec* (*mat-of-cblinfun* $A *_v$ x)) ‘ *set* *S'*)
apply (*subst image-cong*[*OF refl*])

```
    apply (subst basis-enum-of-vec-inverse)
   by (auto simp add: S'-def dA-def)
 also have ... = SPAN (map (mult-mat-vec (mat-of-cblinfun A)) S')
   unfolding SPAN-def dB-def[symmetric] Let-def filter-set
   apply (subst filter-True)
   by (simp-all add: dB-def mat-of-cblinfun-def image-image)

 finally show ?thesis
   unfolding dA-def[symmetric] S'-def[symmetric] Let-def
   by simp
qed

definition [code del, code-abbrev]: range-cblinfun-code A = A *_S top
  — A new constant for the special case of applying an operator to the subspace ⊤
(i.e., for computing the range of the operator). We do this to be able to give more
specialized code for this specific situation. (The generic code for (*_S) would work
but is less efficient because it involves repeated matrix multiplications. code-abbrev
makes sure occurrences of A *_S ⊤ are replaced before starting the actual code
generation.

lemma range-cblinfun-code[code]:
  — Code equation for computing the range of an operator A. Returns the columns
of the matrix representation of A.
  fixes A :: 'a::onb-enum ⇒_CL 'b::onb-enum
  shows range-cblinfun-code A = SPAN (cols (mat-of-cblinfun A))
proof −
  define dA dB
    where dA = length (canonical-basis :: 'a list)
      and dB = length (canonical-basis :: 'b list)
  have carrier-A: mat-of-cblinfun A ∈ carrier-mat dB dA
    unfolding mat-of-cblinfun-def dA-def dB-def by simp

  have range-cblinfun-code A = A *_S SPAN (unit-vecs dA)
    unfolding range-cblinfun-code-def
    by (metis dA-def top-ccsubspace-code)
  also have ... = SPAN (map (λi. mat-of-cblinfun A *_v unit-vec dA i) [0..<dA])
    unfolding cblinfun-image-code dA-def[symmetric] Let-def
    apply (subst filter-True)
     apply (meson carrier-vecD subset-code(1) unit-vecs-carrier)
    by (simp add: unit-vecs-def o-def)
  also have ... = SPAN (map (λx. mat-of-cblinfun A *_v col (1_m dA) x) [0..<dA])
    apply (subst map-cong[OF refl])
    by auto
  also have ... = SPAN (map (col (mat-of-cblinfun A * 1_m dA)) [0..<dA])
    apply (subst map-cong[OF refl])
     apply (subst col-mult2[symmetric])
       apply (rule carrier-A)
    by auto
  also have ... = SPAN (cols (mat-of-cblinfun A))
```

unfolding *cols-def dA-def*[*symmetric*]
apply (*subst right-mult-one-mat*[*OF carrier-A*])
using *carrier-A* **by** *blast*
finally show *?thesis*
by −
qed

lemma *uminus-Span-code*[*code*]: − *X* = *range-cblinfun-code* (*id-cblinfun* − *Proj X*)
— Code equation for the orthogonal complement of a subspace *X*. Computed as the range of one minus the projector on *X*
unfolding *range-cblinfun-code-def*
by (*metis Proj-ortho-compl Proj-range*)

lemma *kernel-code*[*code*]:
— Computes the kernel of an operator *A*. This is implemented using the existing functions for transforming a matrix into row echelon form (*gauss-jordan-single*) and for computing a basis of the kernel of such a matrix (*find-base-vectors*)
kernel A = *SPAN* (*find-base-vectors* (*gauss-jordan-single* (*mat-of-cblinfun A*)))
for *A*::('*a*::*onb-enum*,'*b*::*onb-enum*) *cblinfun*
proof −
define *dA dB Am Ag base*
where *dA* = *length* (*canonical-basis* :: '*a list*)
and *dB* = *length* (*canonical-basis* :: '*b list*)
and *Am* = *mat-of-cblinfun A*
and *Ag* = *gauss-jordan-single Am*
and *base* = *find-base-vectors Ag*

interpret *complex-vec-space dA*.

have *Am-carrier*: *Am* ∈ *carrier-mat dB dA*
unfolding *Am-def mat-of-cblinfun-def dA-def dB-def* **by** *simp*

have *row-echelon*: *row-echelon-form Ag*
unfolding *Ag-def*
using *Am-carrier refl* **by** (*rule gauss-jordan-single*)

have *Ag-carrier*: *Ag* ∈ *carrier-mat dB dA*
unfolding *Ag-def*
using *Am-carrier refl* **by** (*rule gauss-jordan-single*(*2*))

have *base-carrier*: *set base* ⊆ *carrier-vec dA*
unfolding *base-def*
using *find-base-vectors*(*1*)[*OF row-echelon Ag-carrier*]
using *Ag-carrier mat-kernel-def* **by** *blast*

interpret *k*: *kernel dB dA Ag*
apply *standard* **using** *Ag-carrier* **by** *simp*

have *basis-base*: *kernel.basis dA Ag* (*set base*)
using *row-echelon Ag-carrier* **unfolding** *base-def*
by (*rule find-base-vectors*(*3*))

have *space-as-set* (*SPAN base*)
= *space-as-set* (*ccspan* (*basis-enum-of-vec* ‘ *set base* :: *'a set*))
unfolding *SPAN-def dA-def*[*symmetric*] *Let-def filter-set*
apply (*subst filter-True*)
using *base-carrier* **by** *auto*

also have ... = *cspan* (*basis-enum-of-vec* ‘ *set base*)
apply *transfer* **apply** (*subst closure-finite-cspan*)
by *simp-all*

also have ... = *basis-enum-of-vec* ‘ *span* (*set base*)
apply (*subst basis-enum-of-vec-span*)
using *base-carrier dA-def* **by** *auto*

also have ... = *basis-enum-of-vec* ‘ *mat-kernel Ag*
using *basis-base k.Ker.basis-def k.span-same* **by** *auto*

also have ... = *basis-enum-of-vec* ‘ {*v* ∈ *carrier-vec dA*. *Ag* ∗$_v$ *v* = 0$_v$ *dB*}
apply (*rule arg-cong*[**where** *f*=λ*x*. *basis-enum-of-vec* ‘ *x*])
unfolding *mat-kernel-def* **using** *Ag-carrier*
by *simp*

also have ... = *basis-enum-of-vec* ‘ {*v* ∈ *carrier-vec dA*. *Am* ∗$_v$ *v* = 0$_v$ *dB*}
using *gauss-jordan-single*(*1*)[*OF Am-carrier Ag-def*[*symmetric*]]
by *auto*

also have ... = {*w*. *A* ∗$_V$ *w* = *0*}
proof −
have *basis-enum-of-vec* ‘ {*v* ∈ *carrier-vec dA*. *Am* ∗$_v$ *v* = 0$_v$ *dB*}
= *basis-enum-of-vec* ‘ {*v* ∈ *carrier-vec dA*. *A* ∗$_V$ *basis-enum-of-vec v* = *0*}
apply (*rule arg-cong*[**where** *f*=λ*t*. *basis-enum-of-vec* ‘ *t*])
apply (*rule Collect-cong*)
apply (*simp add*: *Am-def*)
by (*metis Am-carrier Am-def carrier-matD*(*2*) *carrier-vecD dB-def mat-carrier*
mat-of-cblinfun-def mat-of-cblinfun-cblinfun-apply vec-of-basis-enum-inverse
basis-enum-of-vec-inverse vec-of-basis-enum-zero)
also have ... = {*w* ∈ *basis-enum-of-vec* ‘ *carrier-vec dA*. *A* ∗$_V$ *w* = *0*}
apply (*subst Compr-image-eq*[*symmetric*])
by *simp*
also have ... = {*w*. *A* ∗$_V$ *w* = *0*}
apply *auto*

by (*metis* (*no-types*, *lifting*) *Am-carrier Am-def carrier-matD*(*2*) *carrier-vec-dim-vec dim-vec-of-basis-enum' image-iff mat-carrier mat-of-cblinfun-def vec-of-basis-enum-inverse*)
finally show *?thesis*
by −
qed

also have ... = *space-as-set* (*kernel A*)
apply *transfer* **by** *auto*

finally have *SPAN base* = *kernel A*
by (*simp add*: *space-as-set-inject*)
then show *?thesis*
by (*simp add*: *base-def Ag-def Am-def*)
qed

lemma *inf-ccsubspace-code*[*code*]:
— Code equation for intersection of subspaces. Reduced to orthogonal complement and sum of subspaces for which we already have code equations.
(*A*::'*a*::*onb-enum ccsubspace*) ⊓ *B* = − (− *A* ⊔ − *B*)
by (*subst ortho-involution*[*symmetric*], *subst compl-inf*, *simp*)

lemma *Sup-ccsubspace-code*[*code*]:
— Supremum (sum) of a set of subspaces. Implemented by repeated pairwise sum.
Sup (*Set-Monad l* :: '*a*::*onb-enum ccsubspace set*) = *fold sup l bot*
unfolding *Set-Monad-def*
by (*simp add*: *Sup-set-fold*)

lemma *Inf-ccsubspace-code*[*code*]:
— Infimum (intersection) of a set of subspaces. Implemented by the orthogonal complement of the supremum.
Inf (*Set-Monad l* :: '*a*::*onb-enum ccsubspace set*)
= − *Sup* (*Set-Monad* (*map uminus l*))
unfolding *Set-Monad-def*
apply (*induction l*)
by *auto*

17.5 Miscellanea

This is a hack to circumvent a bug in the code generation. The automatically generated code for the class *uniformity* has a type that is different from what the generated code later assumes, leading to compilation errors (in ML at least) in any expression involving - *ell2* (even if the constant *uniformity* is not actually used).

The fragment below circumvents this by forcing Isabelle to use the right type. (The logically useless fragment "*let x* = ((=)::'*a*⇒-⇒-)" achieves this.)

lemma *uniformity-ell2-code*[*code*]: (*uniformity* :: ('*a ell2* ∗ -) *filter*) = *Filter.abstract-filter*

(%-.
Code.abort STR ''no uniformity'' (%-.
let x = ((=)::'a⇒-⇒-) in uniformity))
by *simp*

Code equation for *UNIV*. It is now implemented via type class *enum* (which provides a list of all values).

declare *enum-class.UNIV-enum*[*code*]

Setup for code generation involving sets of *ell2*/*ccsubspace*. This configures to use lists for representing sets in code.

derive (*eq*) *ceq ccsubspace*
derive (*no*) *ccompare ccsubspace*
derive (*monad*) *set-impl ccsubspace*
derive (*eq*) *ceq ell2*
derive (*no*) *ccompare ell2*
derive (*monad*) *set-impl ell2*

unbundle *no lattice-syntax* **and** *no jnf-syntax* **and** *no cblinfun-syntax*

end

18 *Cblinfun-Code-Examples* – Examples and test cases for code generation

theory *Cblinfun-Code-Examples*
imports
Complex-Bounded-Operators.Extra-Pretty-Code-Examples
Jordan-Normal-Form.Matrix-Impl
HOL−Library.Code-Target-Numeral
Cblinfun-Code
begin

hide-const (**open**) *Order.bottom Order.top*
no-notation *Lattice.join* (**infixl** ‹⊔ı› *65*)
no-notation *Lattice.meet* (**infixl** ‹⊓ı› *70*)

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

19 Examples

19.1 Operators

value *id-cblinfun* :: *bool ell2* $\Rightarrow_{CL}$ *bool ell2*

value *1* :: *unit ell2* $\Rightarrow_{CL}$ *unit ell2*

value *id-cblinfun* + *id-cblinfun* :: *bool ell2* $\Rightarrow_{CL}$ *bool ell2*

value *0* :: (*bool ell2* $\Rightarrow_{CL}$ *Enum.finite-3 ell2*)

value − *id-cblinfun* :: *bool ell2* $\Rightarrow_{CL}$ *bool ell2*

value *id-cblinfun* − *id-cblinfun* :: *bool ell2* $\Rightarrow_{CL}$ *bool ell2*

value *classical-operator* (λ*b*. *Some* (¬ *b*))

value ‹*explicit-cblinfun* (λ*x y* :: *bool*. *of-bool* (*x* ∧ *y*))›

value *id-cblinfun* = (*0* :: *bool ell2* $\Rightarrow_{CL}$ *bool ell2*)

value *2* $*_R$ *id-cblinfun* :: *bool ell2* $\Rightarrow_{CL}$ *bool ell2*

value *imaginary-unit* $*_C$ *id-cblinfun* :: *bool ell2* $\Rightarrow_{CL}$ *bool ell2*

value *id-cblinfun* o_{CL} *0* :: *bool ell2* $\Rightarrow_{CL}$ *bool ell2*

value *id-cblinfun*∗ :: *bool ell2* $\Rightarrow_{CL}$ *bool ell2*

19.2 Vectors

value *0* :: *bool ell2*

value *1* :: *unit ell2*

value *ket False*

value *2* $*_C$ *ket False*

value *2* $*_R$ *ket False*

value *ket True* + *ket False*

value *ket True* − *ket True*

value *ket True* = *ket True*

value − *ket True*

value *cinner* (*ket True*) (*ket True*)

value *norm* (*ket True*)

value *ket* () ∗ *ket* ()

value *1* :: *unit ell2*

value (*1*::*unit ell2*) ∗ (*1*::*unit ell2*)

19.3 Vector/Matrix

value *id-cblinfun* $*_V$ *ket True*

value ‹*vector-to-cblinfun* (*ket True*) :: *unit ell2* $\Rightarrow_{CL}$ -›

19.4 Subspaces

value *ccspan* {*ket False*}

value *Proj* (*ccspan* {*ket False*})

value *top* :: *bool ell2 ccsubspace*

value *bot* :: *bool ell2 ccsubspace*

value *0* :: *bool ell2 ccsubspace*

value *ccspan* {*ket False*} ⊔ *ccspan* {*ket True*}

value *ccspan* {*ket False*} + *ccspan* {*ket True*}

value *ccspan* {*ket False*} ⊓ *ccspan* {*ket True*}

value *id-cblinfun* $*_S$ *ccspan* {*ket False*}

value *id-cblinfun* $*_S$ (*top* :: *bool ell2 ccsubspace*)

value − *ccspan* {*ket False*}

value *ccspan* {*ket False*, *ket True*} = *top*

value *ccspan* {*ket False*} ≤ *ccspan* {*ket True*}

value *cblinfun-image id-cblinfun* (*ccspan* {*ket True*})

value *kernel id-cblinfun* :: *bool ell2 ccsubspace*

value *eigenspace 1 id-cblinfun* :: *bool ell2 ccsubspace*

value *Inf* {*ccspan* {*ket False*}, *top*}

value *Sup* {*ccspan* {*ket False*}, *top*}

end

References

[1] J. B. Conway. *A course in functional analysis*, volume 96. Springer Science & Business Media, 2013.

[2] F. Haftmann. Code generation from Isabelle/HOL theories. https://isabelle.in.tum.de/website-Isabelle2019/dist/Isabelle2019/doc/codegen.pdf, 2019.