Complex Bounded Operators*

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

[1]University of Tartu

May 26, 2024

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, and the Estonian Centre of Exellence in IT (EXCITE) funded by ERDF.

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*
⟨*proof*⟩

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*

⟨*proof*⟩

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*
⟨*proof*⟩

translations *x* ↼ *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*∈ℝ
shows *x* ≤ *0* ∨ *x* ≥ *0*
⟨*proof*⟩

lemma *unique-choice*: $\forall x.\ \exists!y.\ Q\ x\ y \Longrightarrow \exists!f.\ \forall x.\ Q\ x\ (f\ x)$
⟨*proof*⟩

lemma *image-set-plus*:
  assumes ‹*linear U*›
  shows ‹*U* ‘ (*A* + *B*) = *U* ‘ *A* + *U* ‘ *B*›
  ⟨*proof*⟩

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 L* **and** *a2*: *K* ≤ *L*
  shows *L2-set f K* ≤ *L2-set f L*
⟨*proof*⟩

lemma *Sup-real-close*:
  fixes *e* :: *real*
  assumes *0* < *e*
    and *S*: *bdd-above S S* ≠ {}
  shows ∃ *x*∈*S*. *Sup S* − *e* < *x*
⟨*proof*⟩

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

⟨*ML*⟩

lemma *card-prod-omega*: ‹*X* ∗*c natLeq* =*o X*› **if** ‹*Cinfinite X*›
  ⟨*proof*⟩

lemma *countable-leq-natLeq*: ‹|*X*| ≤*o natLeq*› **if** ‹*countable X*›
  ⟨*proof*⟩

lemma *set-Times-plus-distrib*: ‹(*A* × *B*) + (*C* × *D*) = (*A* + *C*) × (*B* + *D*)›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

lemma *not-not-singleton-zero*:
  ‹*x* = *0*› **if** ‹¬ *class.not-singleton TYPE*(′*a*)› **for** *x* :: ‹′*a*::*zero*›
  ⟨*proof*⟩

lemma *UNIV-not-singleton*[*simp*]: (*UNIV*::-::*not-singleton set*) ≠ {*x*}

⟨*proof*⟩

lemma *UNIV-not-singleton-converse*:
assumes $\bigwedge x::'a.\ UNIV \neq \{x\}$
shows $\exists\, x::'a.\ \exists\, y.\ x \neq y$
⟨*proof*⟩

subclass (**in** *card2*) *not-singleton*
⟨*proof*⟩

subclass (**in** *perfect-space*) *not-singleton*
⟨*proof*⟩

lemma *class-not-singletonI-monoid-add*:
assumes $(UNIV::'a\ set) \neq \{0\}$
shows *class.not-singleton TYPE('a::monoid-add)*
⟨*proof*⟩

lemma *not-singleton-vs-CARD-1*:
assumes ‹¬ *class.not-singleton TYPE('a)*›
shows ‹*class.CARD-1 TYPE('a)*›
⟨*proof*⟩

2.3 *CARD-1*

context *CARD-1* **begin**

lemma *everything-the-same*[*simp*]: $(x::'a)=y$
⟨*proof*⟩

lemma *CARD-1-UNIV*: $UNIV = \{x::'a\}$
⟨*proof*⟩

lemma *CARD-1-ext*: $x\ (a::'a) = y\ b \Longrightarrow x = y$
⟨*proof*⟩

end

instance *unit* :: *CARD-1*
⟨*proof*⟩

instance *prod* :: (*CARD-1*, *CARD-1*) *CARD-1*
⟨*proof*⟩

instance *fun* :: (*CARD-1*, *CARD-1*) *CARD-1*
⟨*proof*⟩

lemma *enum-CARD-1*: (*Enum.enum* :: *'a*::{*CARD-1*,*enum*} *list*) = [*a*]

⟨*proof*⟩

lemma *card-not-singleton*: ‹*CARD*(*'a*::*not-singleton*) ≠ *1*›
⟨*proof*⟩

2.4 Topology

lemma *cauchy-filter-metricI*:
fixes *F* :: *'a*::*metric-space filter*
assumes $\bigwedge e.\ e>0 \Longrightarrow \exists P.\ eventually\ P\ F \wedge (\forall x\ y.\ P\ x \wedge P\ y \longrightarrow dist\ x\ y < e)$
shows *cauchy-filter F*
⟨*proof*⟩

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*⟩

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.
$((\lambda x.\ c + f\ x :: 'a::topological\text{-}group\text{-}add) \longrightarrow c + d)\ F \longleftrightarrow (f \longrightarrow d)\ F$
⟨*proof*⟩

lemma *finite-subsets-at-top-minus*:
assumes $A \subseteq B$
shows $finite\text{-}subsets\text{-}at\text{-}top\ (B - A) \le filtermap\ (\lambda F.\ F - A)\ (finite\text{-}subsets\text{-}at\text{-}top\ B)$
⟨*proof*⟩

lemma *finite-subsets-at-top-inter*:
assumes $A \subseteq B$
shows $filtermap\ (\lambda F.\ F \cap A)\ (finite\text{-}subsets\text{-}at\text{-}top\ B) = finite\text{-}subsets\text{-}at\text{-}top\ A$
⟨*proof*⟩

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

lemma *complete-singleton*:
complete {*s*::*'a*::*uniform-space*}
⟨*proof*⟩

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*⟩

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*⟩

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*: ‹$\forall_F$ *w in F*. $f\ w \in S$›
shows $((g \circ f) \longrightarrow z)\ F$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *has-sum-comm-additive-general*:
— This is a strengthening of *has-sum-comm-additive-general*.
fixes *f* :: ‹'*b* :: {*comm-monoid-add*,*topological-space*} ⇒ '*c* :: {*comm-monoid-add*,*topological-space*}›
assumes *f-sum*: ‹$\bigwedge F$. *finite F* $\Longrightarrow F \subseteq S \Longrightarrow$ *sum* (*f o 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-sum x*) *S*›
shows ‹((*f o g*) *has-sum* (*f x*)) *S*›
⟨*proof*⟩

lemma *summable-on-comm-additive-general*:
— This is a strengthening of *summable-on-comm-additive-general*.
fixes *g* :: ‹'*a* ⇒ '*b* :: {*comm-monoid-add*,*topological-space*}› **and** *f* :: ‹'*b* ⇒ '*c* :: {*comm-monoid-add*,*topological-space*}›
assumes ‹$\bigwedge F$. *finite F* $\Longrightarrow F \subseteq S \Longrightarrow$ *sum* (*f o 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-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 o g*) *summable-on S*›
⟨*proof*⟩

lemma *has-sum-metric*:
fixes *l* :: ‹*'a* :: {*metric-space*, *comm-monoid-add*}›
shows ‹(*f has-sum l*) *A* ⟷ (∀ *e*. *e* > *0* ⟶ (∃ *X*. *finite X* ∧ *X* ⊆ *A* ∧ (∀ *Y*. *finite Y* ∧ *X* ⊆ *Y* ∧ *Y* ⊆ *A* ⟶ *dist* (*sum f Y*) *l* < *e*)))›
⟨*proof*⟩

lemma *summable-on-product-finite-left*:
fixes *f* :: ‹*'a*×*'b* ⇒ *'c*::{*topological-comm-monoid-add*}›
assumes *sum*: ‹⋀*x*. *x*∈*X* ⟹ (λ*y*. *f*(*x*,*y*)) *summable-on Y*›
assumes ‹*finite X*›
shows ‹*f summable-on* (*X*×*Y*)›
⟨*proof*⟩

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*⟩

2.6 Complex numbers

lemma *cmod-Re*:
assumes *x* ≥ *0*
shows *cmod x* = *Re x*
⟨*proof*⟩

lemma *abs-complex-real*[*simp*]: *abs x* ∈ ℝ **for** *x* :: *complex*
⟨*proof*⟩

lemma *Im-abs*[*simp*]: *Im* (*abs x*) = *0*
⟨*proof*⟩

lemma *cnj-x-x*: *cnj x* ∗ *x* = (*abs x*)^2
⟨*proof*⟩

lemma *cnj-x-x-geq0*[*simp*]: ‹*cnj x* ∗ *x* ≥ *0*›
⟨*proof*⟩

lemma *complex-of-real-leq-1-iff*[*iff*]: ‹*complex-of-real x* ≤ *1* ⟷ *x* ≤ *1*›
⟨*proof*⟩

lemma *x-cnj-x*: ‹*x* ∗ *cnj x* = (*abs x*)^2›

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *index-of-correct*:
assumes *x* ∈ *set y*
shows *y* ! *index-of x y* = *x*
⟨*proof*⟩

lemma *enum-idx-correct*:
Enum.enum ! *enum-idx i* = *i*
⟨*proof*⟩

lemma *index-of-bound*:
assumes *y* ≠ [] **and** *x* ∈ *set y*
shows *index-of x y* < *length y*
⟨*proof*⟩

lemma *enum-idx-bound*[*simp*]: *enum-idx x* < *CARD*(′*a*) **for** *x* :: ′*a*::*enum*
⟨*proof*⟩

lemma *index-of-nth*:
assumes *distinct xs*
assumes *i* < *length xs*
shows *index-of* (*xs* ! *i*) *xs* = *i*
⟨*proof*⟩

lemma *enum-idx-enum*:
assumes ‹*i* < *CARD*(′*a*::*enum*)›
shows ‹*enum-idx* (*enum-class.enum* ! *i* :: ′*a*) = *i*›
⟨*proof*⟩

2.8 Filtering lists/sets

lemma *map-filter-map*: *List.map-filter f* (*map g l*) = *List.map-filter* (*f o g*) *l*
⟨*proof*⟩

lemma *map-filter-Some*[*simp*]: *List.map-filter* (λ*x*. *Some* (*f x*)) *l* = *map f l*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *Set-filter-unchanged*: *Set.filter P X* = *X* **if** $\bigwedge x.\ x \in X \Longrightarrow P\ x$ **for** *P* **and** *X* :: *'z set*
⟨*proof*⟩

2.9 Maps

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

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

lemma *inj-map-total*[*simp*]: *inj-map* (*Some o* π) = *inj* π
⟨*proof*⟩

lemma *inj-map-Some*[*simp*]: *inj-map Some*
⟨*proof*⟩

lemma *inv-map-total*:
assumes *surj* π
shows *inv-map* (*Some o* π) = *Some o inv* π
⟨*proof*⟩

lemma *inj-map-map-comp*[*simp*]:
assumes *a1*: *inj-map f* **and** *a2*: *inj-map g*
shows *inj-map* ($f \circ_m g$)
⟨*proof*⟩

lemma *inj-map-inv-map*[*simp*]: *inj-map* (*inv-map* π)
⟨*proof*⟩

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*: $-$ ($\bigsqcap A$) = $\bigsqcup$ (*uminus* ‘ *A*) **for** *A* :: ‹*'a*::*complete-orthocomplemented-lattice set*›
⟨*proof*⟩

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

lemma *uminus-INF*: $-$ (*INF* $x \in A$. *B x*) = (*SUP* $x \in A$. $-$ *B x*) **for** *B* :: ‹*'a* $\Rightarrow$ *'b*::*complete-orthocomplemented-lattice*›
⟨*proof*⟩

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

lemma *uminus-Sup*: $-(\bigsqcup A) = \bigsqcap(uminus \text{ ` } A)$ **for** *A* :: ‹′*a*::*complete-orthocomplemented-lattice set*›
⟨*proof*⟩

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

lemma *uminus-SUP*: $-$ (*SUP* $x \in A$. *B x*) = (*INF* $x \in A$. $-$ *B x*) **for** *B* :: ‹′*a* ⇒ ′*b*::*complete-orthocomplemented-lattice*›
⟨*proof*⟩

lemma *has-sumI-metric*:
fixes *l* :: ‹′*a* :: {*metric-space*, *comm-monoid-add*}›
assumes ‹$\bigwedge e.\ e > 0 \Longrightarrow \exists X.$ *finite* $X \wedge X \subseteq A \wedge (\forall Y.$ *finite* $Y \wedge X \subseteq Y \wedge Y \subseteq A \longrightarrow$ *dist* (*sum f Y*) $l < e)$›
shows ‹(*f has-sum l*) *A*›
⟨*proof*⟩

lemma *limitin-pullback-topology*:
‹*limitin* (*pullback-topology A g T*) *f l F* ⟷ *l*∈*A* ∧ ($\forall_F$ *x in F*. *f x* ∈ *A*) ∧ *limitin T* (*g o f*) (*g l*) *F*›
⟨*proof*⟩

lemma *tendsto-coordinatewise*: ‹(*f* ⟶ *l*) *F* ⟷ (∀ *x*. ((λ*i*. *f i x*) ⟶ *l x*) *F*)›
⟨*proof*⟩

lemma *limitin-closure-of*:
assumes *limit*: ‹*limitin T f c F*›
assumes *in-S*: ‹$\forall_F$ *x in F*. *f x* ∈ *S*›
assumes *nontrivial*: ‹¬ *trivial-limit F*›
shows ‹*c* ∈ *T closure-of S*›
⟨*proof*⟩

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* ⟨*proof*⟩
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 λ*f g x*. *f x* + *g x* ⟨*proof*⟩
lift-definition *zero-euclidean-space* :: *'a euclidean-space* **is** λ-. *0* ⟨*proof*⟩
lift-definition *uminus-euclidean-space* ::
  'a euclidean-space ⇒ *'a euclidean-space*
  is λ*f x*. − *f x* ⟨*proof*⟩
lift-definition *minus-euclidean-space* ::
  'a euclidean-space ⇒ *'a euclidean-space* ⇒ *'a euclidean-space*
  is λ*f g x*. *f x* − *g x*⟨*proof*⟩
lift-definition *scaleR-euclidean-space* ::
  real ⇒ *'a euclidean-space* ⇒ *'a euclidean-space*
  is λ*c f x*. *c* ∗ *f x* ⟨*proof*⟩
instance
  ⟨*proof*⟩
end

instantiation *euclidean-space* :: (*finite*) *real-inner* **begin**
lift-definition *inner-euclidean-space* :: *'a euclidean-space* ⇒ *'a euclidean-space* ⇒ *real*
  is λ*f g*. ∑*x*∈*UNIV*. *f x* ∗ *g x* :: *real* ⟨*proof*⟩
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*. $\forall_F$ (*x'*::*'a euclidean-space*, *y*) *in uniformity*. *x'* = *x* ⟶ *y* ∈ *U*)
instance
⟨*proof*⟩
end

instantiation *euclidean-space* :: (*finite*) *euclidean-space* **begin**
lift-definition *euclidean-space-basis-vector* :: *'a* ⇒ *'a euclidean-space* **is**
  λ*x*. *indicator* {*x*} ⟨*proof*⟩
definition *Basis-euclidean-space* == (*euclidean-space-basis-vector* ‘ *UNIV*)
instance
⟨*proof*⟩
end

3.2 Misc

lemma *closure-bounded-linear-image-subset-eq*:
  assumes *f*: *bounded-linear f*

shows *closure* (*f* ‘ *closure S*) = *closure* (*f* ‘ *S*)
⟨*proof*⟩

lemma *not-singleton-real-normed-is-perfect-space*[*simp*]: ‹*class.perfect-space* (*open* :: ′*a*::{*not-singleton*,*real-normed-vector*} *set* ⇒ *bool*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

lemma *norm-plus-leq-norm-prod*: ‹*norm* (*a* + *b*) ≤ *sqrt 2* ∗ *norm* (*a*, *b*)›
⟨*proof*⟩

lemma *ex-norm1*:
assumes ‹(*UNIV*::′*a*::*real-normed-vector set*) ≠ {*0*}›
shows ‹∃ *x*::′*a*. *norm x* = *1*›
⟨*proof*⟩

lemma *bdd-above-norm-f*:
assumes *bounded-linear f*
shows ‹*bdd-above* {*norm* (*f x*) |*x*. *norm x* = *1*}›
⟨*proof*⟩

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

lemma *abs-summable-on-scaleR-left* [*intro*]:
fixes *c* :: ‹′*a* :: *real-normed-vector*›
assumes *c* ≠ *0* ⟹ *f abs-summable-on A*
shows (λ*x*. *f x* $*_R$ *c*) *abs-summable-on A*
⟨*proof*⟩

lemma *abs-summable-on-scaleR-right* [*intro*]:
fixes *f* :: ‹′*a* ⇒ ′*b* :: *real-normed-vector*›
assumes *c* ≠ *0* ⟹ *f abs-summable-on A*
shows (λ*x*. *c* $*_R$ *f x*) *abs-summable-on A*

⟨*proof*⟩

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 Orderings.thy

This class is analogous to *unbounded-dense-linorder*, except that it does not require a total order

class *unbounded-dense-order* = *dense-order* + *no-top* + *no-bot*

instance *unbounded-dense-linorder* ⊆ *unbounded-dense-order* ⟨*proof*⟩

4.3 Missing from Rings.thy

The existing class *abs-if* requires $|a| = (\mathit{if}\ a < (0::'a)\ \mathit{then}\ -\ a\ \mathit{else}\ a)$. However, if $(<)$ is not a total order, this condition is too strong when a is incomparable with $0::'a$. (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 $0::'a$.

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

class *ordered-semiring-1* = *ordered-semiring* + *semiring-1*
— missing class analogous to *linordered-semiring-1* without requiring a total order
begin

lemma *convex-bound-le*:
assumes $x \leq a$ **and** $y \leq a$ **and** $0 \leq u$ **and** $0 \leq v$ **and** $u + v = 1$
shows $u * x + v * y \leq a$
⟨*proof*⟩

end

subclass (**in** *linordered-semiring-1*) *ordered-semiring-1* ⟨*proof*⟩

class *ordered-semiring-strict* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add* +
— missing class analogous to *linordered-semiring-strict* without requiring a total order
assumes *mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
assumes *mult-strict-right-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *ordered-semiring*
⟨*proof*⟩

lemma *mult-pos-pos*[*simp*]: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a * b$
⟨*proof*⟩

lemma *mult-pos-neg*: $0 < a \Longrightarrow b < 0 \Longrightarrow a * b < 0$
⟨*proof*⟩

lemma *mult-neg-pos*: $a < 0 \Longrightarrow 0 < b \Longrightarrow a * b < 0$
⟨*proof*⟩

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes *t1*: $a < b$ **and** *t2*: $c < d$ **and** *t3*: $0 < b$ **and** *t4*: $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

This weaker variant has more natural premises

lemma *mult-strict-mono'*:
assumes $a < b$ **and** $c < d$ **and** $0 \leq a$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

lemma *mult-less-le-imp-less*:

assumes *t1*: $a < b$ **and** *t2*: $c \leq d$ **and** *t3*: $0 \leq a$ **and** *t4*: $0 < c$
shows $a * c < b * d$
⟨*proof*⟩

lemma *mult-le-less-imp-less*:
assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

end

subclass (**in** *linordered-semiring-strict*) *ordered-semiring-strict*
⟨*proof*⟩

class *ordered-semiring-1-strict* = *ordered-semiring-strict* + *semiring-1*
— missing class analogous to *linordered-semiring-1-strict* without requiring a total order
begin

subclass *ordered-semiring-1* ⟨*proof*⟩

lemma *convex-bound-lt*:
assumes $x < a$ **and** $y < a$ **and** $0 \leq u$ **and** $0 \leq v$ **and** $u + v = 1$
shows $u * x + v * y < a$
⟨*proof*⟩

end

subclass (**in** *linordered-semiring-1-strict*) *ordered-semiring-1-strict* ⟨*proof*⟩

class *ordered-comm-semiring-strict* = *comm-semiring-0* + *ordered-cancel-ab-semigroup-add* +
— missing class analogous to *linordered-comm-semiring-strict* without requiring a total order
assumes *comm-mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *ordered-semiring-strict*
⟨*proof*⟩

subclass *ordered-cancel-comm-semiring*
⟨*proof*⟩

end

subclass (**in** *linordered-comm-semiring-strict*) *ordered-comm-semiring-strict*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *mult-strict-left-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow c * a < c * b$
⟨*proof*⟩

lemma *mult-strict-right-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow a * c < b * c$
⟨*proof*⟩

lemma *mult-neg-neg*: $a < 0 \Longrightarrow b < 0 \Longrightarrow 0 < a * b$
⟨*proof*⟩

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

4.4 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*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x / y < w / z \longleftrightarrow (x * z - w * y) / (y * z) < 0$
⟨*proof*⟩

lemma *frac-le-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x / y \leq w / z \longleftrightarrow (x * z - w * y) / (y * z) \leq 0$
⟨*proof*⟩

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*]: $0 = 1 / a \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *one-divide-eq-0-iff* [*simp*]: $1 / a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

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*]:
$(1 = b/a) = ((a \neq 0 \ \& \ a = b))$
⟨*proof*⟩

lemma *divide-eq-eq-1* [*simp*]:
$(b/a = 1) = ((a \neq 0 \ \& \ a = b))$
⟨*proof*⟩

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*: $0 < a \Longrightarrow 0 < inverse\ a$
and *inverse-le-imp-le*: $inverse\ a \leq inverse\ b \Longrightarrow 0 < a \Longrightarrow b \leq a$
and *dense-le*: $(\bigwedge x.\ x < y \Longrightarrow x \leq z) \Longrightarrow y \leq z$
and *nn-comparable*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a \leq b \lor b \leq a$
and *abs-nn*: $|x| \geq 0$
begin

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

lemma *comparable*:
assumes *h1*: $a \leq c \lor a \geq c$
and *h2*: $b \leq c \lor b \geq c$
shows $a \leq b \lor b \leq a$
⟨*proof*⟩

lemma *negative-imp-inverse-negative*:
$a < 0 \Longrightarrow inverse\ a < 0$
⟨*proof*⟩

lemma *inverse-positive-imp-positive*:
assumes *inv-gt-0*: $0 < inverse\ a$ **and** *nz*: $a \neq 0$
shows $0 < a$
⟨*proof*⟩

lemma *inverse-negative-imp-negative*:
assumes *inv-less-0*: $inverse\ a < 0$ **and** *nz*: $a \neq 0$
shows $a < 0$
⟨*proof*⟩

lemma *linordered-field-no-lb*:

$\forall x. \exists y. y < x$
⟨*proof*⟩

lemma *linordered-field-no-ub*:
$\forall x. \exists y. y > x$
⟨*proof*⟩

lemma *less-imp-inverse-less*:
assumes *less*: $a < b$ **and** *apos*: $0 < a$
shows *inverse* $b <$ *inverse* a
⟨*proof*⟩

lemma *inverse-less-imp-less*:
inverse $a <$ *inverse* $b \Longrightarrow 0 < a \Longrightarrow b < a$
⟨*proof*⟩

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow$ *inverse* $a <$ *inverse* $b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *le-imp-inverse-le*:
$a \leq b \Longrightarrow 0 < a \Longrightarrow$ *inverse* $b \leq$ *inverse* a
⟨*proof*⟩

lemma *inverse-le-iff-le* [*simp*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow$ *inverse* $a \leq$ *inverse* $b \longleftrightarrow b \leq a$
⟨*proof*⟩

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 $a \leq$ *inverse* $b \Longrightarrow b < 0 \Longrightarrow b \leq a$
⟨*proof*⟩

lemma *inverse-less-imp-less-neg*:
inverse $a <$ *inverse* $b \Longrightarrow b < 0 \Longrightarrow b < a$
⟨*proof*⟩

lemma *inverse-less-iff-less-neg* [*simp*]:
$a < 0 \Longrightarrow b < 0 \Longrightarrow$ *inverse* $a <$ *inverse* $b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *le-imp-inverse-le-neg*:
$a \leq b \Longrightarrow b < 0 \Longrightarrow$ *inverse* $b \leq$ *inverse* a
⟨*proof*⟩

lemma *inverse-le-iff-le-neg* [*simp*]:
$a < 0 \Longrightarrow b < 0 \Longrightarrow$ *inverse* $a \leq$ *inverse* $b \longleftrightarrow b \leq a$

⟨*proof*⟩

lemma *one-less-inverse*:
$0 < a \Longrightarrow a < 1 \Longrightarrow 1 < \mathit{inverse}\ a$
⟨*proof*⟩

lemma *one-le-inverse*:
$0 < a \Longrightarrow a \leq 1 \Longrightarrow 1 \leq \mathit{inverse}\ a$
⟨*proof*⟩

lemma *pos-le-divide-eq* [*field-simps*]:
assumes $0 < c$
shows $a \leq b\ /\ c \longleftrightarrow a * c \leq b$
⟨*proof*⟩

lemma *pos-less-divide-eq* [*field-simps*]:
assumes $0 < c$
shows $a < b\ /\ c \longleftrightarrow a * c < b$
⟨*proof*⟩

lemma *neg-less-divide-eq* [*field-simps*]:
assumes $c < 0$
shows $a < b\ /\ c \longleftrightarrow b < a * c$
⟨*proof*⟩

lemma *neg-le-divide-eq* [*field-simps*]:
assumes $c < 0$
shows $a \leq b\ /\ c \longleftrightarrow b \leq a * c$
⟨*proof*⟩

lemma *pos-divide-le-eq* [*field-simps*]:
assumes $0 < c$
shows $b\ /\ c \leq a \longleftrightarrow b \leq a * c$
⟨*proof*⟩

lemma *pos-divide-less-eq* [*field-simps*]:
assumes $0 < c$
shows $b\ /\ c < a \longleftrightarrow b < a * c$
⟨*proof*⟩

lemma *neg-divide-le-eq* [*field-simps*]:
assumes $c < 0$
shows $b\ /\ c \leq a \longleftrightarrow a * c \leq b$
⟨*proof*⟩

lemma *neg-divide-less-eq* [*field-simps*]:
assumes $c < 0$
shows $b\ /\ c < a \longleftrightarrow a * c < b$
⟨*proof*⟩

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*]: $0 < c \Longrightarrow a \leq - (b / c) \longleftrightarrow a * c \leq - b$
⟨*proof*⟩

lemma *neg-le-minus-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow a \leq - (b / c) \longleftrightarrow - b \leq a * c$
⟨*proof*⟩

lemma *pos-less-minus-divide-eq* [*field-simps*]: $0 < c \Longrightarrow a < - (b / c) \longleftrightarrow a * c < - b$
⟨*proof*⟩

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

lemma *pos-minus-divide-less-eq* [*field-simps*]: $0 < c \Longrightarrow - (b / c) < a \longleftrightarrow - b < a * c$
⟨*proof*⟩

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

lemma *pos-minus-divide-le-eq* [*field-simps*]: $0 < c \Longrightarrow - (b / c) \leq a \longleftrightarrow - b \leq a * c$
⟨*proof*⟩

lemma *neg-minus-divide-le-eq* [*field-simps*]: $c < 0 \Longrightarrow - (b / c) \leq a \longleftrightarrow a * c \leq - b$
⟨*proof*⟩

lemma *frac-less-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x / y < w / z \longleftrightarrow (x * z - w * y) / (y * z) < 0$
⟨*proof*⟩

lemma *frac-le-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x / y \leq w / z \longleftrightarrow (x * z - w * y) / (y * z) \leq 0$
⟨*proof*⟩

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*]:
$0 < x \Longrightarrow 0 < y \Longrightarrow 0 < x / y$
⟨*proof*⟩

lemma *divide-nonneg-pos*:
$0 \leq x \Longrightarrow 0 < y \Longrightarrow 0 \leq x \ / \ y$
⟨*proof*⟩

lemma *divide-neg-pos*:
$x < 0 \Longrightarrow 0 < y \Longrightarrow x \ / \ y < 0$
⟨*proof*⟩

lemma *divide-nonpos-pos*:
$x \leq 0 \Longrightarrow 0 < y \Longrightarrow x \ / \ y \leq 0$
⟨*proof*⟩

lemma *divide-pos-neg*:
$0 < x \Longrightarrow y < 0 \Longrightarrow x \ / \ y < 0$
⟨*proof*⟩

lemma *divide-nonneg-neg*:
$0 \leq x \Longrightarrow y < 0 \Longrightarrow x \ / \ y \leq 0$
⟨*proof*⟩

lemma *divide-neg-neg*:
$x < 0 \Longrightarrow y < 0 \Longrightarrow 0 < x \ / \ y$
⟨*proof*⟩

lemma *divide-nonpos-neg*:
$x \leq 0 \Longrightarrow y < 0 \Longrightarrow 0 \leq x \ / \ y$
⟨*proof*⟩

lemma *divide-strict-right-mono*:
$a < b \Longrightarrow 0 < c \Longrightarrow a \ / \ c < b \ / \ c$
⟨*proof*⟩

lemma *divide-strict-right-mono-neg*:
$b < a \Longrightarrow c < 0 \Longrightarrow a \ / \ c < b \ / \ c$
⟨*proof*⟩

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

lemma *divide-strict-left-mono*:
$b < a \Longrightarrow 0 < c \Longrightarrow 0 < a*b \Longrightarrow c \ / \ a < c \ / \ b$
⟨*proof*⟩

lemma *divide-left-mono*:
$b \leq a \Longrightarrow 0 \leq c \Longrightarrow 0 < a*b \Longrightarrow c \ / \ a \leq c \ / \ b$
⟨*proof*⟩

lemma *divide-strict-left-mono-neg*:
$a < b \Longrightarrow c < 0 \Longrightarrow 0 < a*b \Longrightarrow c \ / \ a < c \ / \ b$
⟨*proof*⟩

lemma *mult-imp-div-pos-le*: $0 < y \Longrightarrow x \leq z * y \Longrightarrow x \,/\, y \leq z$
⟨*proof*⟩

lemma *mult-imp-le-div-pos*: $0 < y \Longrightarrow z * y \leq x \Longrightarrow z \leq x \,/\, y$
⟨*proof*⟩

lemma *mult-imp-div-pos-less*: $0 < y \Longrightarrow x < z * y \Longrightarrow x \,/\, y < z$
⟨*proof*⟩

lemma *mult-imp-less-div-pos*: $0 < y \Longrightarrow z * y < x \Longrightarrow z < x \,/\, y$
⟨*proof*⟩

lemma *frac-le*: $0 \leq x \Longrightarrow x \leq y \Longrightarrow 0 < w \Longrightarrow w \leq z \Longrightarrow x \,/\, z \leq y \,/\, w$
⟨*proof*⟩

lemma *frac-less*: $0 \leq x \Longrightarrow x < y \Longrightarrow 0 < w \Longrightarrow w \leq z \Longrightarrow x \,/\, z < y \,/\, w$
⟨*proof*⟩

lemma *frac-less2*: $0 < x \Longrightarrow x \leq y \Longrightarrow 0 < w \Longrightarrow w < z \Longrightarrow x \,/\, z < y \,/\, w$
⟨*proof*⟩

lemma *less-half-sum*: $a < b \Longrightarrow a < (a+b) \,/\, (1+1)$
⟨*proof*⟩

lemma *gt-half-sum*: $a < b \Longrightarrow (a+b)/(1+1) < b$
⟨*proof*⟩

subclass *unbounded-dense-order*
⟨*proof*⟩

lemma *dense-le-bounded*:
fixes $x\ y\ z$ :: $'a$
assumes $x < y$
and $*$: $\bigwedge w.\ [\![x < w ; w < y]\!] \Longrightarrow w \leq z$
shows $y \leq z$
⟨*proof*⟩

subclass *field-abs-sgn* ⟨*proof*⟩

lemma *nonzero-abs-inverse*:
$a \neq 0 \Longrightarrow |inverse\ a| = inverse\ |a|$
⟨*proof*⟩

lemma *nonzero-abs-divide*:
$b \neq 0 \Longrightarrow |a \,/\, b| = |a| \,/\, |b|$

⟨*proof*⟩

lemma *field-le-epsilon*:
assumes *e*: $\bigwedge e.\ 0 < e \Longrightarrow x \leq y + e$
shows $x \leq y$
⟨*proof*⟩

lemma *inverse-positive-iff-positive* [*simp*]:
$(0 < inverse\ a) = (0 < a)$
⟨*proof*⟩

lemma *inverse-negative-iff-negative* [*simp*]:
$(inverse\ a < 0) = (a < 0)$
⟨*proof*⟩

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
$0 \leq inverse\ a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
$inverse\ a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *one-less-inverse-iff*: $1 < inverse\ x \longleftrightarrow 0 < x \wedge x < 1$
⟨*proof*⟩

lemma *one-le-inverse-iff*: $1 \leq inverse\ x \longleftrightarrow 0 < x \wedge x \leq 1$
⟨*proof*⟩

lemma *inverse-less-1-iff*: $inverse\ x < 1 \longleftrightarrow x \leq 0 \vee 1 < x$
⟨*proof*⟩

lemma *inverse-le-1-iff*: $inverse\ x \leq 1 \longleftrightarrow x \leq 0 \vee 1 \leq x$
⟨*proof*⟩

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

lemma *zero-le-divide-1-iff* [*simp*]:
$0 \leq 1\ /\ a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *zero-less-divide-1-iff* [*simp*]:
$0 < 1\ /\ a \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *divide-le-0-1-iff* [*simp*]:
$1\ /\ a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *divide-less-0-1-iff* [*simp*]:

$1 \mathbin{/} a < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *divide-right-mono*:
$a \leq b \Longrightarrow 0 \leq c \Longrightarrow a/c \leq b/c$
⟨*proof*⟩

lemma *divide-right-mono-neg*: $a \leq b$
$\Longrightarrow c \leq 0 \Longrightarrow b \mathbin{/} c \leq a \mathbin{/} c$
⟨*proof*⟩

lemma *divide-left-mono-neg*: $a \leq b$
$\Longrightarrow c \leq 0 \Longrightarrow 0 < a * b \Longrightarrow c \mathbin{/} a \leq c \mathbin{/} b$
⟨*proof*⟩

lemma *divide-nonneg-nonneg* [*simp*]:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x \mathbin{/} y$
⟨*proof*⟩

lemma *divide-nonpos-nonpos*:
$x \leq 0 \Longrightarrow y \leq 0 \Longrightarrow 0 \leq x \mathbin{/} y$
⟨*proof*⟩

lemma *divide-nonneg-nonpos*:
$0 \leq x \Longrightarrow y \leq 0 \Longrightarrow x \mathbin{/} y \leq 0$
⟨*proof*⟩

lemma *divide-nonpos-nonneg*:
$x \leq 0 \Longrightarrow 0 \leq y \Longrightarrow x \mathbin{/} y \leq 0$
⟨*proof*⟩

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (1 \leq b/a) = (a \leq b)$
⟨*proof*⟩

lemma *le-divide-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (1 \leq b/a) = (b \leq a)$
⟨*proof*⟩

lemma *divide-le-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (b/a \leq 1) = (b \leq a)$
⟨*proof*⟩

lemma *divide-le-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (b/a \leq 1) = (a \leq b)$
⟨*proof*⟩

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

$0 < a \Longrightarrow (1 < b/a) = (a < b)$
⟨*proof*⟩

lemma *less-divide-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (1 < b/a) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (b/a < 1) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow b/a < 1 \longleftrightarrow a < b$
⟨*proof*⟩

lemma *abs-div-pos*: $0 < y \Longrightarrow$
$|x| / y = |x / y|$
⟨*proof*⟩

lemma *zero-le-divide-abs-iff* [*simp*]: $(0 \leq a / |b|) = (0 \leq a \mid b = 0)$
⟨*proof*⟩

lemma *divide-le-0-abs-iff* [*simp*]: $(a / |b| \leq 0) = (a \leq 0 \mid b = 0)$
⟨*proof*⟩

For creating values between u and v.

lemma *scaling-mono*:
assumes $u \leq v$ **and** $0 \leq r$ **and** $r \leq s$
shows $u + r * (v - u) / s \leq v$
⟨*proof*⟩

end

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

4.5 Ordering on complex numbers

instantiation *complex* :: *nice-ordered-field* **begin**
instance
⟨*proof*⟩
end

lemma *less-eq-complexI*: $Re\ x \leq Re\ y \Longrightarrow Im\ x = Im\ y \Longrightarrow x \leq y$ ⟨*proof*⟩
lemma *less-complexI*: $Re\ x < Re\ y \Longrightarrow Im\ x = Im\ y \Longrightarrow x < y$ ⟨*proof*⟩

lemma *complex-of-real-mono*:
x ≤ *y* ⟹ *complex-of-real x* ≤ *complex-of-real y*
⟨*proof*⟩

lemma *complex-of-real-mono-iff* [*simp*]:
complex-of-real x ≤ *complex-of-real y* ⟷ *x* ≤ *y*
⟨*proof*⟩

lemma *complex-of-real-strict-mono-iff* [*simp*]:
complex-of-real x < *complex-of-real y* ⟷ *x* < *y*
⟨*proof*⟩

lemma *complex-of-real-nn-iff* [*simp*]:
0 ≤ *complex-of-real y* ⟷ *0* ≤ *y*
⟨*proof*⟩

lemma *complex-of-real-pos-iff* [*simp*]:
0 < *complex-of-real y* ⟷ *0* < *y*
⟨*proof*⟩

lemma *Re-mono*: *x* ≤ *y* ⟹ *Re x* ≤ *Re y*
⟨*proof*⟩

lemma *comp-Im-same*: *x* ≤ *y* ⟹ *Im x* = *Im y*
⟨*proof*⟩

lemma *Re-strict-mono*: *x* < *y* ⟹ *Re x* < *Re y*
⟨*proof*⟩

lemma *complex-of-real-cmod*: ‹*complex-of-real* (*cmod x*) = *abs x*›
⟨*proof*⟩

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-normed-vector*, *not-singleton*} ⇒ ′*b*::*real-normed-vector*

assumes *a1*: *bounded-linear f*
shows ‹*onorm f* = *Sup* {*norm* (*f x*) | *x*. *norm x* = *1*}›
⟨*proof*⟩

lemma *onormI*:
assumes $\bigwedge x$. *norm* (*f x*) ≤ *b* ∗ *norm x*
and *x* ≠ *0* **and** *norm* (*f x*) = *b* ∗ *norm x*
shows *onorm f* = *b*
⟨*proof*⟩

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 $x /_C c$ ≡ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

subclass (**in** *complex-algebra-1*) *real-algebra-1* ⟨*proof*⟩

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

subclass (**in** *complex-div-algebra*) *real-div-algebra* ⟨*proof*⟩

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

subclass (**in** *complex-field*) *real-field* ⟨*proof*⟩

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

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

instance
⟨*proof*⟩

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*.
⟨*proof*⟩

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*⟩

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*⟩

lemma *clinear-compose*: *clinear f* ⟹ *clinear g* ⟹ *clinear* (*g* ∘ *f*)
⟨*proof*⟩

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*]:
$c \neq 0 \Longrightarrow a = b /_C\ c \longleftrightarrow c *_C\ a = b$
$c \neq 0 \Longrightarrow b /_C\ c = a \longleftrightarrow b = c *_C\ a$
$c \neq 0 \Longrightarrow a + b /_C\ c = (c *_C\ a + b) /_C\ c$
$c \neq 0 \Longrightarrow a /_C\ c + b = (a + c *_C\ b) /_C\ c$
$c \neq 0 \Longrightarrow a - b /_C\ c = (c *_C\ a - b) /_C\ c$
$c \neq 0 \Longrightarrow a /_C\ c - b = (a - c *_C\ b) /_C\ c$
$c \neq 0 \Longrightarrow - (a /_C\ c) + b = (- a + c *_C\ b) /_C\ c$
$c \neq 0 \Longrightarrow - (a /_C\ c) - b = (- a - c *_C\ b) /_C\ c$
for *a b* :: ′*a* :: *complex-vector*
⟨*proof*⟩

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-vector*
⟨*proof*⟩

lemma *scaleC-2*:
fixes *x* :: ′*a*::*complex-vector*
shows *scaleC 2 x* = *x* + *x*
⟨*proof*⟩

lemma *scaleC-half-double* [*simp*]:
fixes *a* :: ′*a*::*complex-vector*
shows (*1* / *2*) $*_C$ (*a* + *a*) = *a*
⟨*proof*⟩

lemma *clinear-scale-complex*:
fixes *c*::*complex* **shows** *clinear f* ⟹ *f* (*c* ∗ *b*) = *c* ∗ *f b*
⟨*proof*⟩

interpretation *scaleC-left*: *additive* (λ*a*. *scaleC a x* :: ′*a*::*complex-vector*)
⟨*proof*⟩

interpretation *scaleC-right*: *additive* (λ*x*. *scaleC a x* :: ′*a*::*complex-vector*)
⟨*proof*⟩

lemma *nonzero-inverse-scaleC-distrib*:
a ≠ *0* ⟹ *x* ≠ *0* ⟹ *inverse* (*scaleC a x*) = *scaleC* (*inverse a*) (*inverse x*)
for *x* :: ′*a*::*complex-div-algebra*
⟨*proof*⟩

lemma *inverse-scaleC-distrib*: *inverse* (*scaleC a x*) = *scaleC* (*inverse a*) (*inverse x*)
for *x* :: ′*a*::{*complex-div-algebra*,*division-ring*}
⟨*proof*⟩

lemma *complex-add-divide-simps*[*vector-add-divide-simps*]:
$v + (b / z) *_C w = (\textit{if } z = 0 \textit{ then } v \textit{ else } (z *_C v + b *_C w) /_C z)$
$a *_C v + (b / z) *_C w = (\textit{if } z = 0 \textit{ then } a *_C v \textit{ else } ((a * z) *_C v + b *_C w) /_C z)$
$(a / z) *_C v + w = (\textit{if } z = 0 \textit{ then } w \textit{ else } (a *_C v + z *_C w) /_C z)$
$(a / z) *_C v + b *_C w = (\textit{if } z = 0 \textit{ then } b *_C w \textit{ else } (a *_C v + (b * z) *_C w) /_C z)$
$v - (b / z) *_C w = (\textit{if } z = 0 \textit{ then } v \textit{ else } (z *_C v - b *_C w) /_C z)$
$a *_C v - (b / z) *_C w = (\textit{if } z = 0 \textit{ then } a *_C v \textit{ else } ((a * z) *_C v - b *_C w) /_C z)$
$(a / z) *_C v - w = (\textit{if } z = 0 \textit{ then } -w \textit{ else } (a *_C v - z *_C w) /_C z)$
$(a / z) *_C v - b *_C w = (\textit{if } z = 0 \textit{ then } -b *_C w \textit{ else } (a *_C v - (b * z) *_C w) /_C z)$
for *v* :: *'a* :: *complex-vector*
⟨*proof*⟩

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

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

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* ⟷ *?rhs*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *scaleC-eq-iff* [*simp*]: $b + u *_C a = a + u *_C b \longleftrightarrow a = b \lor u = 1$
for *a* :: *'a::complex-vector*
⟨*proof*⟩

lemma *scaleC-collapse* [*simp*]: $(1 - u) *_C a + u *_C a = a$
for *a* :: *'a::complex-vector*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *of-complex-0* [*simp*]: *of-complex 0* = *0*
  ⟨*proof*⟩

lemma *of-complex-1* [*simp*]: *of-complex 1* = *1*
  ⟨*proof*⟩

lemma *of-complex-add* [*simp*]: *of-complex* (*x* + *y*) = *of-complex x* + *of-complex y*
  ⟨*proof*⟩

lemma *of-complex-minus* [*simp*]: *of-complex* (− *x*) = − *of-complex x*
  ⟨*proof*⟩

lemma *of-complex-diff* [*simp*]: *of-complex* (*x* − *y*) = *of-complex x* − *of-complex y*
  ⟨*proof*⟩

lemma *of-complex-mult* [*simp*]: *of-complex* (*x* ∗ *y*) = *of-complex x* ∗ *of-complex y*
  ⟨*proof*⟩

lemma *of-complex-sum*[*simp*]: *of-complex* (*sum f s*) = (∑ *x*∈*s*. *of-complex* (*f x*))
  ⟨*proof*⟩

lemma *of-complex-prod*[*simp*]: *of-complex* (*prod f s*) = (∏ *x*∈*s*. *of-complex* (*f x*))
  ⟨*proof*⟩

lemma *nonzero-of-complex-inverse*:
  x ≠ *0* ⟹ *of-complex* (*inverse x*) = *inverse* (*of-complex x* :: *'a*::*complex-div-algebra*)
  ⟨*proof*⟩

lemma *of-complex-inverse* [*simp*]:
  of-complex (*inverse x*) = *inverse* (*of-complex x* :: *'a*::{*complex-div-algebra*,*division-ring*})
  ⟨*proof*⟩

lemma *nonzero-of-complex-divide*:
  y ≠ *0* ⟹ *of-complex* (*x* / *y*) = (*of-complex x* / *of-complex y* :: *'a*::*complex-field*)
  ⟨*proof*⟩

lemma *of-complex-divide* [*simp*]:
  of-complex (*x* / *y*) = (*of-complex x* / *of-complex y* :: *'a*::*complex-div-algebra*)
  ⟨*proof*⟩

lemma *of-complex-power* [*simp*]:
of-complex $(x \,\hat{}\, n)$ = (*of-complex* x :: $'a$::{*complex-algebra-1*}) $\hat{}\, n$
⟨*proof*⟩

lemma *of-complex-power-int* [*simp*]:
of-complex (*power-int* x n) = *power-int* (*of-complex* x :: $'a$ :: {*complex-div-algebra*,*division-ring*}) n
⟨*proof*⟩

lemma *of-complex-eq-iff* [*simp*]: *of-complex* x = *of-complex* y ⟷ $x = y$
⟨*proof*⟩

lemma *inj-of-complex*: *inj of-complex*
⟨*proof*⟩

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 ⟷ $-x = y$
⟨*proof*⟩

lemma *of-complex-eq-minus-of-complex-iff* [*simp*]: *of-complex* x = −*of-complex* y ⟷ $x = -y$
⟨*proof*⟩

lemma *of-complex-eq-id* [*simp*]: *of-complex* = (*id* :: *complex* ⇒ *complex*)
⟨*proof*⟩

Collapse nested embeddings.

lemma *of-complex-of-nat-eq* [*simp*]: *of-complex* (*of-nat* n) = *of-nat* n
⟨*proof*⟩

lemma *of-complex-of-int-eq* [*simp*]: *of-complex* (*of-int* z) = *of-int* z
⟨*proof*⟩

lemma *of-complex-numeral* [*simp*]: *of-complex* (*numeral* w) = *numeral* w
⟨*proof*⟩

lemma *of-complex-neg-numeral* [*simp*]: *of-complex* (− *numeral* w) = − *numeral* w
⟨*proof*⟩

lemma *numeral-power-int-eq-of-complex-cancel-iff* [*simp*]:
power-int (*numeral* x) n = (*of-complex* y :: $'a$ :: {*complex-div-algebra*, *division-ring*}) ⟷
power-int (*numeral* x) $n = y$
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *of-complex-in-Ints-iff* [*simp*]: *of-complex* $x \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-of-complex* [*intro*]: $x \in \mathbb{Z} \Longrightarrow$ *of-complex* $x \in \mathbb{Z}$
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *scaleC-times* [*simp*]:
fixes a :: $'a$::*complex-algebra-1*
shows (*numeral* u) $*_C$ (*numeral* w $*$ a) = (*numeral* u $*$ *numeral* w) $*_C$ a
⟨*proof*⟩

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* $r \in \mathbb{C}$
⟨*proof*⟩

lemma *Complexs-of-int* [*simp*]: *of-int* $z \in \mathbb{C}$
⟨*proof*⟩

lemma *Complexs-of-nat* [*simp*]: *of-nat* $n \in \mathbb{C}$
⟨*proof*⟩

lemma *Complexs-numeral* [*simp*]: *numeral* $w \in \mathbb{C}$

⟨*proof*⟩

lemma *Complexs-0* [*simp*]: $0 \in \mathbb{C}$ **and** *Complexs-1* [*simp*]: $1 \in \mathbb{C}$
⟨*proof*⟩

lemma *Complexs-add* [*simp*]: $a \in \mathbb{C} \Longrightarrow b \in \mathbb{C} \Longrightarrow a + b \in \mathbb{C}$
⟨*proof*⟩

lemma *Complexs-minus* [*simp*]: $a \in \mathbb{C} \Longrightarrow - a \in \mathbb{C}$
⟨*proof*⟩

lemma *Complexs-minus-iff* [*simp*]: $- a \in \mathbb{C} \longleftrightarrow a \in \mathbb{C}$
⟨*proof*⟩

lemma *Complexs-diff* [*simp*]: $a \in \mathbb{C} \Longrightarrow b \in \mathbb{C} \Longrightarrow a - b \in \mathbb{C}$
⟨*proof*⟩

lemma *Complexs-mult* [*simp*]: $a \in \mathbb{C} \Longrightarrow b \in \mathbb{C} \Longrightarrow a * b \in \mathbb{C}$
⟨*proof*⟩

lemma *nonzero-Complexs-inverse*: $a \in \mathbb{C} \Longrightarrow a \neq 0 \Longrightarrow$ *inverse* $a \in \mathbb{C}$
for a :: *'a::complex-div-algebra*
⟨*proof*⟩

lemma *Complexs-inverse*: $a \in \mathbb{C} \Longrightarrow$ *inverse* $a \in \mathbb{C}$
for a :: *'a::{complex-div-algebra,division-ring}*
⟨*proof*⟩

lemma *Complexs-inverse-iff* [*simp*]: *inverse* $x \in \mathbb{C} \longleftrightarrow x \in \mathbb{C}$
for x :: *'a::{complex-div-algebra,division-ring}*
⟨*proof*⟩

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*
⟨*proof*⟩

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}*
⟨*proof*⟩

lemma *Complexs-power* [*simp*]: $a \in \mathbb{C} \Longrightarrow a \,\hat{}\, n \in \mathbb{C}$
for a :: *'a::complex-algebra-1*
⟨*proof*⟩

lemma *Complexs-cases* [*cases set*: *Complexs*]:
assumes $q \in \mathbb{C}$
obtains (*of-complex*) c **where** $q =$ *of-complex* c
⟨*proof*⟩

lemma *sum-in-Complexs* [*intro*,*simp*]: $(\bigwedge i.\ i \in s \Longrightarrow f\ i \in \mathbb{C}) \Longrightarrow sum\ f\ s \in \mathbb{C}$
⟨*proof*⟩

lemma *prod-in-Complexs* [*intro*,*simp*]: $(\bigwedge i.\ i \in s \Longrightarrow f\ i \in \mathbb{C}) \Longrightarrow prod\ f\ s \in \mathbb{C}$
⟨*proof*⟩

lemma *Complexs-induct* [*case-names of-complex*, *induct set*: *Complexs*]:
$q \in \mathbb{C} \Longrightarrow (\bigwedge r.\ P\ (of\text{-}complex\ r)) \Longrightarrow P\ q$
⟨*proof*⟩

6.4 Ordered complex vector spaces

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

subclass (**in** *ordered-complex-vector*) *ordered-real-vector*
⟨*proof*⟩

lemma *scaleC-mono*:
$a \le b \Longrightarrow x \le y \Longrightarrow 0 \le b \Longrightarrow 0 \le x \Longrightarrow a *_C x \le b *_C y$
⟨*proof*⟩

lemma *scaleC-mono′*:
$a \le b \Longrightarrow c \le d \Longrightarrow 0 \le a \Longrightarrow 0 \le c \Longrightarrow a *_C c \le b *_C d$
⟨*proof*⟩

lemma *pos-le-divideC-eq* [*field-simps*]:
$a \le b /_C c \longleftrightarrow c *_C a \le b$ (**is** $?P \longleftrightarrow ?Q$) **if** $0 < c$
⟨*proof*⟩

lemma *pos-less-divideC-eq* [*field-simps*]:
$a < b /_C c \longleftrightarrow c *_C a < b$ **if** $c > 0$
⟨*proof*⟩

lemma *pos-divideC-le-eq* [*field-simps*]:
$b /_C c \le a \longleftrightarrow b \le c *_C a$ **if** $c > 0$
⟨*proof*⟩

lemma *pos-divideC-less-eq* [*field-simps*]:
$b /_C c < a \longleftrightarrow b < c *_C a$ **if** $c > 0$
⟨*proof*⟩

lemma *pos-le-minus-divideC-eq* [*field-simps*]:
$a \le -\ (b /_C c) \longleftrightarrow c *_C a \le -\ b$ **if** $c > 0$
⟨*proof*⟩

lemma *pos-less-minus-divideC-eq* [*field-simps*]:

$a < - (b /_C c) \longleftrightarrow c *_C a < - b$ **if** $c > 0$
⟨*proof*⟩

lemma *pos-minus-divideC-le-eq* [*field-simps*]:
$- (b /_C c) \leq a \longleftrightarrow - b \leq c *_C a$ **if** $c > 0$
⟨*proof*⟩

lemma *pos-minus-divideC-less-eq* [*field-simps*]:
$- (b /_C c) < a \longleftrightarrow - b < c *_C a$ **if** $c > 0$
⟨*proof*⟩

lemma *scaleC-image-atLeastAtMost*: $c > 0 \Longrightarrow scaleC\ c\ `\ \{x..y\} = \{c *_C x..c *_C y\}$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *divideC-field-splits-simps-1* [*field-split-simps*]:
$a = b\ /_C\ c \longleftrightarrow (\mathit{if}\ c = 0\ \mathit{then}\ a = 0\ \mathit{else}\ c *_C a = b)$
$b\ /_C\ c = a \longleftrightarrow (\mathit{if}\ c = 0\ \mathit{then}\ a = 0\ \mathit{else}\ b = c *_C a)$
$a + b\ /_C\ c = (\mathit{if}\ c = 0\ \mathit{then}\ a\ \mathit{else}\ (c *_C a + b)\ /_C\ c)$
$a\ /_C\ c + b = (\mathit{if}\ c = 0\ \mathit{then}\ b\ \mathit{else}\ (a + c *_C b)\ /_C\ c)$
$a - b\ /_C\ c = (\mathit{if}\ c = 0\ \mathit{then}\ a\ \mathit{else}\ (c *_C a - b)\ /_C\ c)$
$a\ /_C\ c - b = (\mathit{if}\ c = 0\ \mathit{then}\ -\ b\ \mathit{else}\ (a - c *_C b)\ /_C\ c)$
$-\ (a\ /_C\ c) + b = (\mathit{if}\ c = 0\ \mathit{then}\ b\ \mathit{else}\ (-\ a + c *_C b)\ /_C\ c)$
$-\ (a\ /_C\ c) - b = (\mathit{if}\ c = 0\ \mathit{then}\ -\ b\ \mathit{else}\ (-\ a - c *_C b)\ /_C\ c)$
for $a\ b$ :: $'a$ :: *complex-vector*
⟨*proof*⟩

lemma *divideC-field-splits-simps-2* [*field-split-simps*]:
$0 < c \Longrightarrow a \le b\ /_C\ c \longleftrightarrow (\mathit{if}\ c > 0\ \mathit{then}\ c *_C a \le b\ \mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ b \le c *_C a\ \mathit{else}\ a \le 0)$
$0 < c \Longrightarrow a < b\ /_C\ c \longleftrightarrow (\mathit{if}\ c > 0\ \mathit{then}\ c *_C a < b\ \mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ b < c *_C a\ \mathit{else}\ a < 0)$
$0 < c \Longrightarrow b\ /_C\ c \le a \longleftrightarrow (\mathit{if}\ c > 0\ \mathit{then}\ b \le c *_C a\ \mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ c *_C a \le b\ \mathit{else}\ a \ge 0)$
$0 < c \Longrightarrow b\ /_C\ c < a \longleftrightarrow (\mathit{if}\ c > 0\ \mathit{then}\ b < c *_C a\ \mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ c *_C a < b\ \mathit{else}\ a > 0)$
$0 < c \Longrightarrow a \le -\ (b\ /_C\ c) \longleftrightarrow (\mathit{if}\ c > 0\ \mathit{then}\ c *_C a \le -\ b\ \mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ -\ b \le c *_C a\ \mathit{else}\ a \le 0)$
$0 < c \Longrightarrow a < -\ (b\ /_C\ c) \longleftrightarrow (\mathit{if}\ c > 0\ \mathit{then}\ c *_C a < -\ b\ \mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ -\ b < c *_C a\ \mathit{else}\ a < 0)$
$0 < c \Longrightarrow -\ (b\ /_C\ c) \le a \longleftrightarrow (\mathit{if}\ c > 0\ \mathit{then}\ -\ b \le c *_C a\ \mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ c *_C a \le -\ b\ \mathit{else}\ a \ge 0)$
$0 < c \Longrightarrow -\ (b\ /_C\ c) < a \longleftrightarrow (\mathit{if}\ c > 0\ \mathit{then}\ -\ b < c *_C a\ \mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ c *_C a < -\ b\ \mathit{else}\ a > 0)$
for $a\ b$ :: $'a$ :: *ordered-complex-vector*
⟨*proof*⟩

lemma *scaleC-nonneg-nonneg*: $0 \le a \Longrightarrow 0 \le x \Longrightarrow 0 \le a *_C x$
for x :: $'a$::*ordered-complex-vector*
⟨*proof*⟩

lemma *scaleC-nonneg-nonpos*: $0 \le a \Longrightarrow x \le 0 \Longrightarrow a *_C x \le 0$
for x :: $'a$::*ordered-complex-vector*
⟨*proof*⟩

lemma *scaleC-nonpos-nonneg*: $a \le 0 \Longrightarrow 0 \le x \Longrightarrow a *_C x \le 0$

for x :: $'a$::*ordered-complex-vector*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *scaleC-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a *_C b$
for b :: $'a$::*ordered-complex-vector*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ⟨*proof*⟩

instance *complex-normed-div-algebra* < *complex-normed-algebra-1* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *norm-of-complex* [*simp*]: *norm* (*of-complex c* :: *'a::complex-normed-algebra-1*)
= *cmod c*
⟨*proof*⟩

lemma *norm-of-complex-add1* [*simp*]: *norm* (*of-complex x* + *1* :: *'a* :: *complex-normed-div-algebra*)
= *cmod* (*x* + *1*)
⟨*proof*⟩

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

lemma *norm-of-complex-diff* [*simp*]:
norm (*of-complex b* − *of-complex a* :: *'a::complex-normed-algebra-1*) ≤ *cmod* (*b*
− *a*)
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *closed-complex-atMost*: ‹*closed* {..*a*::*complex*}›
⟨*proof*⟩

lemma *closed-complex-atLeast*: ‹*closed* {*a*::*complex*..}›
⟨*proof*⟩

lemma *closed-complex-atLeastAtMost*: ‹*closed* {*a*::*complex* .. *b*}›
⟨*proof*⟩

6.8 Sign function

lemma *sgn-scaleC*: *sgn* (*scaleC r x*) = *scaleC* (*sgn r*) (*sgn x*)
for *x* :: *'a*::*complex-normed-vector*
⟨*proof*⟩

lemma *sgn-of-complex*: *sgn* (*of-complex r* :: *'a*::*complex-normed-algebra-1*) = *of-complex* (*sgn r*)
⟨*proof*⟩

lemma *complex-sgn-eq*: *sgn x* = *x* / |*x*|
for *x* :: *complex*
⟨*proof*⟩

lemma *czero-le-sgn-iff* [*simp*]: *0* ≤ *sgn x* ⟷ *0* ≤ *x*
for *x* :: *complex*
⟨*proof*⟩

lemma *csgn-le-0-iff* [*simp*]: *sgn x* ≤ *0* ⟷ *x* ≤ *0*
for *x* :: *complex*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *clinear-times-of-complex*: *clinear* (λ*x*. *a* ∗ *of-complex x*)
⟨*proof*⟩

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*.
⟨*proof*⟩

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

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

lemma *clinear*: *clinear f*
⟨*proof*⟩

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*
⟨*proof*⟩

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*.
⟨*proof*⟩

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* (λ*a*. *a* ∗∗ *b*)
⟨*proof*⟩

lemma *bounded-clinear-right*: *bounded-clinear* (λ*b*. *a* ∗∗ *b*)
⟨*proof*⟩

lemma *flip*: *bounded-cbilinear* (λ*x y*. *y* ∗∗ *x*)
⟨*proof*⟩

lemma *comp1*:
  assumes *bounded-clinear g*
  shows *bounded-cbilinear* (λ*x*. (∗∗) (*g x*))
⟨*proof*⟩

lemma *comp*: *bounded-clinear f* ⟹ *bounded-clinear g* ⟹ *bounded-cbilinear* (λ*x y*. *f x* ∗∗ *g y*)
  ⟨*proof*⟩

end

lemma *bounded-clinear-ident*[*simp*]: *bounded-clinear* (λ*x*. *x*)
  ⟨*proof*⟩

lemma *bounded-clinear-zero*[*simp*]: *bounded-clinear* (λ*x*. *0*)
  ⟨*proof*⟩

lemma *bounded-clinear-add*:
  assumes *bounded-clinear f*
    and *bounded-clinear g*
  shows *bounded-clinear* (λ*x*. *f x* + *g x*)
⟨*proof*⟩

lemma *bounded-clinear-minus*:
  assumes *bounded-clinear f*

shows *bounded-clinear* ($\lambda x.$ $-$ f x)
⟨*proof*⟩

lemma *bounded-clinear-sub*: *bounded-clinear f* $\Longrightarrow$ *bounded-clinear g* $\Longrightarrow$ *bounded-clinear* ($\lambda x.$ f x $-$ g x)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *bounded-clinear-compose*:
assumes *bounded-clinear f*
and *bounded-clinear g*
shows *bounded-clinear* ($\lambda x.$ f (g x))
⟨*proof*⟩

lemma *bounded-cbilinear-mult*: *bounded-cbilinear* (($*$) :: $'a \Rightarrow 'a \Rightarrow 'a$::*complex-normed-algebra*)
⟨*proof*⟩

lemma *bounded-clinear-mult-left*: *bounded-clinear* (λx::$'a$::*complex-normed-algebra*. $x * y$)
⟨*proof*⟩

lemma *bounded-clinear-mult-right*: *bounded-clinear* (λy::$'a$::*complex-normed-algebra*. $x * y$)
⟨*proof*⟩

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* ($\lambda x.$ x / y)
for y :: $'a$::*complex-normed-field*
⟨*proof*⟩

lemma *bounded-cbilinear-scaleC*: *bounded-cbilinear scaleC*
⟨*proof*⟩

lemma *bounded-clinear-scaleC-left*: *bounded-clinear* ($\lambda c.$ *scaleC* c x)
⟨*proof*⟩

lemma *bounded-clinear-scaleC-right*: *bounded-clinear* ($\lambda x.$ *scaleC* c x)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *complex-bounded-clinear*: *bounded-clinear f* ⟷ (∃ *c*::*complex*. *f* = (λx. *x* ∗ *c*))
for *f* :: *complex* ⇒ *complex*
⟨*proof*⟩

6.9.1 Limits of Sequences

6.10 Cauchy sequences

lemma *cCauchy-iff2*: *Cauchy X* ⟷ (∀ *j*. (∃ *M*. ∀ *m* ≥ *M*. ∀ *n* ≥ *M*. *cmod* (*X m* − *X n*) < *inverse* (*real* (*Suc j*))))
⟨*proof*⟩

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* ⇒ *complex*
assumes *inc*: ⋀*n*. *f n* ≤ *f* (*Suc n*)
and *bdd*: ⋀*n*. *f n* ≤ *l*
and *en*: ⋀*e*. *0* < *e* ⟹ ∃ *n*. *l* ≤ *f n* + *e*
shows *f* ⟶ *l*
⟨*proof*⟩

lemma *complex-Cauchy-convergent*:
fixes *X* :: *nat* ⇒ *complex*
assumes *X*: *Cauchy X*
shows *convergent X*
⟨*proof*⟩

instance *complex* :: *complete-space*
⟨*proof*⟩

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

subclass (**in** *cbanach*) *banach* ⟨*proof*⟩

instance *complex* :: *banach* ⟨*proof*⟩

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 *notation-norm* **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*⟩

lemma (**in** *scaleC*) *scaleC-real*: **assumes** $r \in \mathbb{R}$ **shows** $r *_C x = Re\ r *_R x$
  ⟨*proof*⟩

lemma *of-complex-of-real-eq* [*simp*]: $of\text{-}complex\ (of\text{-}real\ n) = of\text{-}real\ n$

⟨proof⟩

lemma *Complexs-of-real* [*simp*]: *of-real* $r \in \mathbb{C}$
⟨proof⟩

lemma *Reals-in-Complexs*: $\mathbb{R} \subseteq \mathbb{C}$
⟨proof⟩

lemma (**in** *bounded-clinear*) *bounded-linear*: *bounded-linear f*
⟨proof⟩

lemma *clinear-times*: *clinear* ($\lambda x.\ c * x$)
for *c* :: *'a::complex-algebra*
⟨proof⟩

lemma (**in** *clinear*) *linear*: ‹*linear f*›
⟨proof⟩

lemma *bounded-clinearI*:
assumes ‹$\bigwedge b1\ b2.\ f\ (b1 + b2) = f\ b1 + f\ b2$›
assumes ‹$\bigwedge r\ b.\ f\ (r *_C\ b) = r *_C\ f\ b$›
assumes ‹$\bigwedge x.\ norm\ (f\ x) \le norm\ x * K$›
shows *bounded-clinear f*
⟨proof⟩

lemma *bounded-clinear-id*[*simp*]: ‹*bounded-clinear id*›
⟨proof⟩

lemma *bounded-clinear-0*[*simp*]: ‹*bounded-clinear 0*›
⟨proof⟩

definition *cbilinear* :: ‹(*'a::complex-vector* ⇒ *'b::complex-vector* ⇒ *'c::complex-vector*) ⇒ *bool*›
where ‹$cbilinear = (\lambda\ f.\ (\forall\ \ y.\ clinear\ (\lambda\ x.\ f\ x\ y)) \land (\forall\ \ x.\ clinear\ (\lambda\ y.\ f\ x\ y))$
)›

lemma *cbilinear-add-left*:
assumes ‹*cbilinear f*›
shows ‹$f\ (a + b)\ c = f\ a\ c + f\ b\ c$›
⟨proof⟩

lemma *cbilinear-add-right*:
assumes ‹*cbilinear f*›
shows ‹$f\ a\ (b + c) = f\ a\ b + f\ a\ c$›
⟨proof⟩

lemma *cbilinear-times*:
fixes g' :: ‹*'a::complex-vector* ⇒ *complex*› **and** *g* :: ‹*'b::complex-vector* ⇒ *complex*›
assumes ‹$\bigwedge\ x\ y.\ h\ x\ y = (g'\ x)*(g\ y)$› **and** ‹*clinear g*› **and** ‹*clinear g'*›

shows ‹*cbilinear h*›
⟨*proof*⟩

lemma *csubspace-is-subspace*: *csubspace A* $\Longrightarrow$ *subspace A*
⟨*proof*⟩

lemma *span-subset-cspan*: *span A* $\subseteq$ *cspan A*
⟨*proof*⟩

lemma *cindependent-implies-independent*:
assumes *cindependent* (*S*::*'a*::*complex-vector set*)
shows *independent S*
⟨*proof*⟩

lemma *cspan-singleton*: *cspan* $\{x\} = \{\alpha *_C x| \ \alpha.\ True\}$
⟨*proof*⟩

lemma *cspan-as-span*:
cspan (*B*::*'a*::*complex-vector set*) = *span* (*B* ∪ *scaleC* i ‘ *B*)
⟨*proof*⟩

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*⟩

lemma *cindependent-inter-scaleC-cindependent*:
assumes *a1*: *cindependent* (*B*::*'a*::*complex-vector set*) **and** *a3*: $c \neq 1$
shows $B \cap (*_C)\ c$ ‘ $B = \{\}$
⟨*proof*⟩

lemma *real-independent-from-complex-independent*:
assumes *cindependent* (*B*::*'a*::*complex-vector set*)
defines $B' == ((*_C)$ i ‘ $B)$
shows *independent* $(B \cup B')$
⟨*proof*⟩

lemma *crepresentation-from-representation*:
assumes *a1*: *cindependent B* **and** *a2*: $b \in B$ **and** *a3*: *finite B*
shows *crepresentation B* ψ *b* = (*representation* $(B \cup (*_C)$ i ‘ $B)$ ψ *b*)
+ i $*_C$ (*representation* $(B \cup (*_C)$ i ‘ $B)$ ψ (i $*_C$ *b*))
⟨*proof*⟩

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

lemma *scaleC-cindependent*:
assumes *a1*: *cindependent* (*B*::′*a*::*complex-vector set*) **and** *a3*: $c \neq 0$
shows *cindependent* (($*_C$) *c* ‘ *B*)
⟨*proof*⟩

lemma *cspan-eqI*:
assumes ‹$\bigwedge a.\ a \in A \Longrightarrow a \in$ *cspan B*›
assumes ‹$\bigwedge b.\ b \in B \Longrightarrow b \in$ *cspan A*›
shows ‹*cspan A* = *cspan B*›
⟨*proof*⟩

lemma (**in** *bounded-cbilinear*) *bounded-bilinear*[*simp*]: *bounded-bilinear prod*
⟨*proof*⟩

lemma *norm-scaleC-sgn*[*simp*]: ‹*complex-of-real* (*norm* ψ) $*_C$ *sgn* $\psi = \psi$› **for** ψ :: ′*a*::*complex-normed-vector*
⟨*proof*⟩

lemma *scaleC-of-complex*[*simp*]: ‹*scaleC x* (*of-complex y*) = *of-complex* ($x * y$)›
⟨*proof*⟩

lemma *bounded-clinear-inv*:
assumes [*simp*]: ‹*bounded-clinear f*›
assumes *b*: ‹$b > 0$›
assumes *bound*: ‹$\bigwedge x.$ *norm* (*f x*) $\geq b *$ *norm x*›
assumes ‹*surj f*›
shows ‹*bounded-clinear* (*inv f*)›
⟨*proof*⟩

lemma *range-is-csubspace*[*simp*]:
assumes *a1*: *clinear f*
shows *csubspace* (*range f*)
⟨*proof*⟩

lemma *csubspace-is-convex*[*simp*]:
assumes *a1*: *csubspace M*
shows *convex M*
⟨*proof*⟩

lemma *kernel-is-csubspace*[*simp*]:
assumes *a1*: *clinear f*
shows *csubspace* (*f* −‘ {*0*})
⟨*proof*⟩

lemma *bounded-cbilinear-0*[*simp*]: ‹*bounded-cbilinear* (λ- -. *0*)›
⟨*proof*⟩
lemma *bounded-cbilinear-0′*[*simp*]: ‹*bounded-cbilinear 0*›
⟨*proof*⟩

lemma *bounded-cbilinear-apply-bounded-clinear*: ‹*bounded-clinear* (*f x*)› **if** ‹*bounded-cbilinear f*›
⟨*proof*⟩

lemma *clinear-scaleR*[*simp*]: ‹*clinear* (*scaleR x*)›
⟨*proof*⟩

lemma *abs-summable-on-scaleC-left* [*intro*]:
fixes *c* :: ‹′*a* :: *complex-normed-vector*›
assumes $c \neq 0 \Longrightarrow$ *f abs-summable-on A*
shows $(\lambda x.\ f\ x *_C c)$ *abs-summable-on A*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *antilinear-imp-scaleC*:
fixes *D* :: *complex* ⇒ ′*a*::*complex-vector*
assumes *antilinear D*
obtains *d* **where** $D = (\lambda x.\ cnj\ x *_C d)$
⟨*proof*⟩

corollary *complex-antilinearD*:
fixes *f* :: *complex* ⇒ *complex*
assumes *antilinear f* **obtains** *c* **where** $f = (\lambda x.\ c * cnj\ x)$
⟨*proof*⟩

lemma *antilinearI*:
assumes $\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y$
and $\bigwedge c\ x.\ f\ (c *_C x) = cnj\ c *_C f\ x$
shows *antilinear f*
⟨*proof*⟩

lemma *antilinear-o-antilinear*: *antilinear f* $\Longrightarrow$ *antilinear g* $\Longrightarrow$ *clinear* (*g o f*)
⟨*proof*⟩

lemma *clinear-o-antilinear*: *antilinear f* $\Longrightarrow$ *clinear g* $\Longrightarrow$ *antilinear* (*g o f*)
⟨*proof*⟩

lemma *antilinear-o-clinear*: *clinear f* $\Longrightarrow$ *antilinear g* $\Longrightarrow$ *antilinear* (*g o f*)
⟨*proof*⟩

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

lemma *bounded-antilinearI*:
assumes ‹⋀*b1 b2*. *f* (*b1* + *b2*) = *f b1* + *f b2*›
assumes ‹⋀*r b*. *f* (*r* $*_C$ *b*) = *cnj r* $*_C$ *f b*›
assumes ‹∀ *x*. *norm* (*f x*) ≤ *norm x* ∗ *K*›
shows *bounded-antilinear f*
⟨*proof*⟩

sublocale *bounded-antilinear* ⊆ *real*: *bounded-linear*
— Gives access to all lemmas from *Real-Vector-Spaces.linear* using prefix *real*.
⟨*proof*⟩

lemma (**in** *bounded-antilinear*) *bounded-linear*: *bounded-linear f*
⟨*proof*⟩

lemma (**in** *bounded-antilinear*) *antilinear*: *antilinear f*
⟨*proof*⟩

lemma *bounded-antilinear-intro*:
assumes ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y*
and ⋀*r x*. *f* (*scaleC r x*) = *scaleC* (*cnj r*) (*f x*)
and ⋀*x*. *norm* (*f x*) ≤ *norm x* ∗ *K*
shows *bounded-antilinear f*
⟨*proof*⟩

lemma *bounded-antilinear-0*[*simp*]: ‹*bounded-antilinear* (λ-. *0*)›
⟨*proof*⟩

lemma *bounded-antilinear-0'*[*simp*]: ‹*bounded-antilinear 0*›
⟨*proof*⟩

lemma *cnj-bounded-antilinear*[*simp*]: *bounded-antilinear cnj*
⟨*proof*⟩

lemma *bounded-antilinear-o-bounded-antilinear*:
assumes *bounded-antilinear f*

and *bounded-antilinear g*
shows *bounded-clinear* $(\lambda x.\ f\ (g\ x))$
⟨*proof*⟩

lemma *bounded-antilinear-o-bounded-antilinear'*:
assumes *bounded-antilinear f*
and *bounded-antilinear g*
shows *bounded-clinear* $(g\ o\ f)$
⟨*proof*⟩

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

lemma *bounded-antilinear-o-bounded-clinear'*:
assumes *bounded-clinear f*
and *bounded-antilinear g*
shows *bounded-antilinear* $(g\ o\ f)$
⟨*proof*⟩

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

lemma *bounded-clinear-o-bounded-antilinear'*:
assumes *bounded-antilinear f*
and *bounded-clinear g*
shows *bounded-antilinear* $(g\ o\ f)$
⟨*proof*⟩

lemma *bij-clinear-imp-inv-clinear*: *clinear* (*inv f*)
if *a1*: *clinear f* **and** *a2*: *bij f*
⟨*proof*⟩

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*: $\exists K.\ \forall a\ b.\ norm\ (prod\ a\ b) \leq norm\ a * norm\ b * K$

sublocale *bounded-sesquilinear* ⊆ *real*: *bounded-bilinear*
— Gives access to all lemmas from *Real-Vector-Spaces.linear* using prefix *real*.
⟨*proof*⟩

lemma (**in** *bounded-sesquilinear*) *bounded-bilinear*[*simp*]: *bounded-bilinear prod*
⟨*proof*⟩

lemma (**in** *bounded-sesquilinear*) *bounded-antilinear-left*: *bounded-antilinear* (*λa. prod a b*)
⟨*proof*⟩

lemma (**in** *bounded-sesquilinear*) *bounded-clinear-right*: *bounded-clinear* (*λb. prod a b*)
⟨*proof*⟩

lemma (**in** *bounded-sesquilinear*) *comp1*:
assumes ‹*bounded-clinear g*›
shows ‹*bounded-sesquilinear* (*λx. prod* (*g x*))›
⟨*proof*⟩

lemma (**in** *bounded-sesquilinear*) *comp2*:
assumes ‹*bounded-clinear g*›
shows ‹*bounded-sesquilinear* (*λx y. prod x* (*g y*))›
⟨*proof*⟩

lemma (**in** *bounded-sesquilinear*) *comp*: *bounded-clinear f* ⟹ *bounded-clinear g* ⟹ *bounded-sesquilinear* (*λx y. prod* (*f x*) (*g y*))
⟨*proof*⟩

lemma *bounded-clinear-const-scaleR*:
fixes *c* :: *real*
assumes ‹*bounded-clinear f*›
shows ‹*bounded-clinear* (*λ x. c* $*_R$ *f x*)›
⟨*proof*⟩

lemma *bounded-linear-bounded-clinear*:
‹*bounded-linear A* ⟹ ∀ *c x. A* (*c* $*_C$ *x*) = *c* $*_C$ *A x* ⟹ *bounded-clinear A*›
⟨*proof*⟩

lemma *comp-bounded-clinear*:
fixes *A* :: ‹*'b*::*complex-normed-vector* ⇒ *'c*::*complex-normed-vector*›
and *B* :: ‹*'a*::*complex-normed-vector* ⇒ *'b*›
assumes ‹*bounded-clinear A*› **and** ‹*bounded-clinear B*›
shows ‹*bounded-clinear* (*A* ∘ *B*)›
⟨*proof*⟩

lemma *bounded-sesquilinear-add*:
‹*bounded-sesquilinear* (*λ x y. A x y* + *B x y*)› **if** ‹*bounded-sesquilinear A*› **and**

‹*bounded-sesquilinear B*›
⟨*proof*⟩

lemma *bounded-sesquilinear-uminus*:
‹*bounded-sesquilinear* (λ *x y*. − *A x y*)› **if** ‹*bounded-sesquilinear A*›
⟨*proof*⟩

lemma *bounded-sesquilinear-diff*:
‹*bounded-sesquilinear* (λ *x y*. *A x y* − *B x y*)› **if** ‹*bounded-sesquilinear A*› **and** ‹*bounded-sesquilinear B*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *bounded-sesquilinear-0′*[*simp*]: ‹*bounded-sesquilinear 0*›
⟨*proof*⟩

lemma *bounded-sesquilinear-apply-bounded-clinear*: ‹*bounded-clinear* (*f x*)› **if** ‹*bounded-sesquilinear f*›
⟨*proof*⟩

7.3 Misc 2

lemma *summable-on-scaleC-left* [*intro*]:
fixes *c* :: ‹′*a* :: *complex-normed-vector*›
assumes $c \neq 0 \Longrightarrow$ *f summable-on A*
shows (λ*x*. *f x* $*_C$ *c*) *summable-on A*
⟨*proof*⟩

lemma *summable-on-scaleC-right* [*intro*]:
fixes *f* :: ‹′*a* ⇒ ′*b* :: *complex-normed-vector*›
assumes $c \neq 0 \Longrightarrow$ *f summable-on A*
shows (λ*x*. *c* $*_C$ *f x*) *summable-on A*
⟨*proof*⟩

lemma *infsum-scaleC-left*:
fixes *c* :: ‹′*a* :: *complex-normed-vector*›
assumes $c \neq 0 \Longrightarrow$ *f summable-on A*
shows *infsum* (λ*x*. *f x* $*_C$ *c*) *A* = *infsum f A* $*_C$ *c*
⟨*proof*⟩

lemma *infsum-scaleC-right*:
fixes *f* :: ‹′*a* ⇒ ′*b* :: *complex-normed-vector*›
shows *infsum* (λ*x*. *c* $*_C$ *f x*) *A* = *c* $*_C$ *infsum f A*
⟨*proof*⟩

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*⟩

lemma *closure-scaleC*:
  fixes *S*::‹*'a*::*complex-normed-vector set*›
  shows ‹*closure* ((*$*_C$) *a* ‘ *S*) = (*$*_C$) *a* ‘ *closure S*›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

7.4 Finite dimension and canonical basis

lemma *vector-finitely-spanned*:
assumes ‹*z* ∈ *cspan T*›
shows ‹∃ *S*. *finite S* ∧ *S* ⊆ *T* ∧ *z* ∈ *cspan S*›
⟨*proof*⟩

⟨*ML*⟩

class *cfinite-dim* = *complex-vector* +
assumes *cfinitely-spanned*: ∃ *S*::*'a set*. *finite S* ∧ *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*›

⟨*ML*⟩

instantiation *complex* :: *basis-enum* **begin**
definition *canonical-basis* = [*1*::*complex*]
definition ‹*canonical-basis-length* (-::*complex itself*) = *1*›
instance
⟨*proof*⟩
end

lemma *cdim-UNIV-basis-enum*[*simp*]: ‹*cdim* (*UNIV*::*'a*::*basis-enum set*) = *length* (*canonical-basis*::*'a list*)›
⟨*proof*⟩

lemma *finite-basis*: ∃ *basis*::*'a*::*cfinite-dim set*. *finite basis* ∧ *cindependent basis* ∧ *cspan basis* = *UNIV*
⟨*proof*⟩

instance *basis-enum* ⊆ *cfinite-dim*
⟨*proof*⟩

lemma *cindependent-cfinite-dim-finite*:
assumes ‹*cindependent* (*S*::*'a*::*cfinite-dim set*)›
shows ‹*finite S*›
⟨*proof*⟩

lemma *cfinite-dim-finite-subspace-basis*:
assumes ‹*csubspace X*›
shows ∃ *basis*::*'a*::*cfinite-dim set. finite basis* ∧ *cindependent basis* ∧ *cspan basis* = *X*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*)
  ⟨*proof*⟩

lemma *finite-span-closed*[*simp*]:
  fixes *B* :: *'a::real-normed-vector set*
  assumes *finite B*
  shows *closed* (*real-vector.span B*)
  ⟨*proof*⟩

lemma *finite-cspan-closed*[*simp*]:
  fixes *S*::‹*'a::complex-normed-vector set*›
  assumes *a1*: ‹*finite S*›
  shows ‹*closed* (*cspan S*)›
  ⟨*proof*⟩

lemma *closure-finite-cspan*:
  fixes *T*::‹*'a::complex-normed-vector set*›
  assumes ‹*finite T*›
  shows ‹*closure* (*cspan T*) = *cspan T*›
  ⟨*proof*⟩

lemma *finite-cspan-crepresentation-bounded*:
  fixes *B* :: *'a::complex-normed-vector set*
  assumes *a1*: *finite B* **and** *a2*: *cindependent B*
  shows ∃ *D*>*0*. ∀ ψ *b*. *cmod* (*crepresentation B* ψ *b*) ≤ *norm* ψ ∗ *D*
⟨*proof*⟩

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 *notation-norm*
  ⟨*proof*⟩

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 :: ‹$'b \Rightarrow$ *real*›
and c :: ‹$'a$ :: *real-normed-vector*›
assumes ‹$c \neq 0$›
assumes ‹($\lambda x.\ f\ x *_R\ c$) *summable-on* A›
shows ‹f *summable-on* A›
⟨*proof*⟩

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* ($\lambda x.\ f\ x *_R\ c$) A = *infsum* f $A *_R\ c$
⟨*proof*⟩

lemma *infsum-of-real*:
shows ‹($\sum_\infty x \in A$. *of-real* ($f\ x$) :: $'b$::{*real-normed-vector*, *real-algebra-1*}) = *of-real* ($\sum_\infty x \in A.\ f\ x$)›
— This result has nothing to do with the bounded operator library but it uses *finite-span-closed* so it is proven here.
⟨*proof*⟩

7.5 Closed subspaces

lemma *csubspace-INF*[*simp*]: ($\bigwedge x.\ x \in A \Longrightarrow$ *csubspace* x) $\Longrightarrow$ *csubspace* ($\bigcap A$)
⟨*proof*⟩

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*⟩

lemma *csubspace-set-plus*:
assumes ‹*csubspace* A› **and** ‹*csubspace* B›
shows ‹*csubspace* ($A + B$)›
⟨*proof*⟩

lemma *closed-csubspace-0*[*simp*]:
closed-csubspace ({*0*} :: (*'a*::{*complex-vector*,*t1-space*}) *set*)
⟨*proof*⟩

lemma *closed-csubspace-UNIV*[*simp*]: *closed-csubspace* (*UNIV*::(*'a*::{*complex-vector*,*topological-space*}) *set*)
⟨*proof*⟩

lemma *closed-csubspace-inter*[*simp*]:
assumes *closed-csubspace A* **and** *closed-csubspace B*
shows *closed-csubspace* (*A*∩*B*)
⟨*proof*⟩

lemma *closed-csubspace-INF*[*simp*]:
assumes *a1*: ∀ *A*∈$\mathcal{A}$. *closed-csubspace A*
shows *closed-csubspace* ($\bigcap \mathcal{A}$)
⟨*proof*⟩

typedef (**overloaded**) (*'a*::{*complex-vector*,*topological-space*})
ccsubspace = ‹{*S*::*'a set*. *closed-csubspace S*}›
morphisms *space-as-set Abs-ccsubspace*
⟨*proof*⟩

setup-lifting *type-definition-ccsubspace*

lemma *csubspace-space-as-set*[*simp*]: ‹*csubspace* (*space-as-set S*)›
⟨*proof*⟩

lemma *closed-space-as-set*[*simp*]: ‹*closed* (*space-as-set S*)›
⟨*proof*⟩

lemma *zero-space-as-set*[*simp*]: ‹*0* ∈ *space-as-set A*›
⟨*proof*⟩

instantiation *ccsubspace* :: (*complex-normed-vector*) *scaleC* **begin**
lift-definition *scaleC-ccsubspace* :: *complex* ⇒ *'a ccsubspace* ⇒ *'a ccsubspace* **is**
$\lambda c\ S.\ (*_C)\ c$ ‘ *S*
⟨*proof*⟩

lift-definition *scaleR-ccsubspace* :: *real* ⇒ *'a ccsubspace* ⇒ *'a ccsubspace* **is**
$\lambda c\ S.\ (*_R)\ c$ ‘ *S*
⟨*proof*⟩

instance
⟨*proof*⟩
end

instantiation *ccsubspace* :: ({*complex-vector*,*t1-space*}) *bot* **begin**
lift-definition *bot-ccsubspace* :: ‹*'a ccsubspace*› **is** ‹{*0*}›
⟨*proof*⟩
instance⟨*proof*⟩
end

lemma *zero-cblinfun-image*[*simp*]: $0 *_C S = bot$ **for** *S* :: *- ccsubspace*
⟨*proof*⟩

lemma *csubspace-scaleC-invariant*:
fixes *a S*
assumes ‹$a \neq 0$› **and** ‹*csubspace S*›
shows ‹$(*_C)$ *a* ‘ $S = S$›
⟨*proof*⟩

lemma *ccsubspace-scaleC-invariant*[*simp*]: $a \neq 0 \Longrightarrow a *_C S = S$ **for** *S* :: *-
ccsubspace*
⟨*proof*⟩

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

instance ⟨*proof*⟩
end

lemma *space-as-set-bot*[*simp*]: ‹*space-as-set bot* = {*0*}›
⟨*proof*⟩

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

⟨*proof*⟩

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

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *Inf*
begin
lift-definition *Inf-ccsubspace*::‹*'a ccsubspace set* ⇒ *'a ccsubspace*›
is ‹λ *S*. $\bigcap$ *S*›
⟨*proof*⟩

instance ⟨*proof*⟩
end

lift-definition *ccspan* :: *'a::complex-normed-vector set* ⇒ *'a ccsubspace*
is λG. *closure* (*cspan* *G*)
⟨*proof*⟩

lemma *ccspan-superset*:
‹*A* ⊆ *space-as-set* (*ccspan A*)›
for *A* :: ‹*'a::complex-normed-vector set*›
⟨*proof*⟩

lemma *ccspan-superset'*: ‹$x \in X \Longrightarrow x \in$ *space-as-set* (*ccspan X*)›
⟨*proof*⟩

lemma *ccspan-canonical-basis*[*simp*]: *ccspan* (*set canonical-basis*) = *top*
⟨*proof*⟩

lemma *ccspan-Inf-def*: ‹*ccspan A* = *Inf* {*S*. *A* ⊆ *space-as-set S*}›
for *A*::‹(*'a::cbanach*) *set*›
⟨*proof*⟩

lemma *cspan-singleton-scaleC*[*simp*]: (*a::complex*)≠*0* ⟹ *cspan* { $a *_C \psi$ } = *cspan* {ψ}
for ψ::*'a::complex-vector*
⟨*proof*⟩

lemma *closure-is-closed-csubspace*[*simp*]:
fixes *S*::‹*'a::complex-normed-vector set*›
assumes ‹*csubspace S*›
shows ‹*closed-csubspace* (*closure S*)›
⟨*proof*⟩

lemma *ccspan-singleton-scaleC*[*simp*]: (*a::complex*)≠*0* ⟹ *ccspan* {$a *_C \psi$} = *ccspan* {ψ}
⟨*proof*⟩

lemma *clinear-continuous-at*:
assumes ‹*bounded-clinear f*›
shows ‹*isCont f x*›
⟨*proof*⟩

lemma *clinear-continuous-within*:
assumes ‹*bounded-clinear f*›
shows ‹*continuous* (*at x within s*) *f*›
⟨*proof*⟩

lemma *antilinear-continuous-at*:
assumes ‹*bounded-antilinear f*›
shows ‹*isCont f x*›
⟨*proof*⟩

lemma *antilinear-continuous-within*:
assumes ‹*bounded-antilinear f*›
shows ‹*continuous* (*at x within s*) *f*›
⟨*proof*⟩

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*⟩

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *order*
begin
lift-definition *less-eq-ccsubspace* :: ‹*'a ccsubspace* ⇒ *'a ccsubspace* ⇒ *bool*›
is ‹(⊆)›⟨*proof*⟩
declare *less-eq-ccsubspace-def*[*code del*]
lift-definition *less-ccsubspace* :: ‹*'a ccsubspace* ⇒ *'a ccsubspace* ⇒ *bool*›
is ‹(⊂)›⟨*proof*⟩
declare *less-ccsubspace-def*[*code del*]
instance
⟨*proof*⟩
end

lemma *ccspan-leqI*:
assumes ‹*M* ⊆ *space-as-set S*›
shows ‹*ccspan M* ≤ *S*›
⟨*proof*⟩

lemma *ccspan-mono*:
assumes ‹*A* ⊆ *B*›
shows ‹*ccspan A* ≤ *ccspan B*›
⟨*proof*⟩

lemma *ccsubspace-leI*:
assumes *t1*: *space-as-set A* ⊆ *space-as-set B*
shows *A* ≤ *B*
⟨*proof*⟩

lemma *ccspan-of-empty*[*simp*]: *ccspan* {} = *bot*
⟨*proof*⟩

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

lemma *space-as-set-inf*[*simp*]: *space-as-set* ($A \sqcap B$) = *space-as-set* $A \cap$ *space-as-set* B
⟨*proof*⟩

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *order-top* **begin**
instance
⟨*proof*⟩
end

instantiation *ccsubspace* :: ({*complex-vector*,*t1-space*}) *order-bot* **begin**
instance
⟨*proof*⟩
end

instantiation *ccsubspace* :: ({*complex-vector*,*topological-space*}) *semilattice-inf* **begin**
instance
⟨*proof*⟩
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*›
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *ccspan-0*[*simp*]: ‹*ccspan* {*0*} = *0*›
⟨*proof*⟩

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*⟩

lemma *right-unique-rel-ccsubspace*[*transfer-rule*]: ‹*right-unique* (*rel-ccsubspace R*)› **if** ‹*right-unique R*›
⟨*proof*⟩

lemma *bi-unique-rel-ccsubspace*[*transfer-rule*]: ‹*bi-unique* (*rel-ccsubspace R*)› **if** ‹*bi-unique R*›
⟨*proof*⟩

lemma *converse-rel-ccsubspace*: ‹*conversep* (*rel-ccsubspace R*) = *rel-ccsubspace* (*conversep*

R)›
⟨*proof*⟩

lemma *space-as-set-top*[*simp*]: ‹*space-as-set top* = *UNIV*›
⟨*proof*⟩

lemma *ccsubspace-eqI*:
assumes ‹⋀*x*. *x* ∈ *space-as-set S* ⟷ *x* ∈ *space-as-set T*›
shows ‹*S* = *T*›
⟨*proof*⟩

lemma *ccspan-remove-0*: ‹*ccspan* (*A* − {*0*}) = *ccspan A*›
⟨*proof*⟩

lemma *sgn-in-spaceD*: ‹*ψ* ∈ *space-as-set A*› **if** ‹*sgn ψ* ∈ *space-as-set A*› **and** ‹*ψ* ≠ *0*›
for *ψ* :: ‹- :: *complex-normed-vector*›
⟨*proof*⟩

lemma *sgn-in-spaceI*: ‹*sgn ψ* ∈ *space-as-set A*› **if** ‹*ψ* ∈ *space-as-set A*›
for *ψ* :: ‹- :: *complex-normed-vector*›
⟨*proof*⟩

lemma *ccsubspace-leI-unit*:
fixes *A B* :: ‹- :: *complex-normed-vector ccsubspace*›
assumes ⋀*ψ*. *norm ψ* = *1* ⟹ *ψ* ∈ *space-as-set A* ⟹ *ψ* ∈ *space-as-set B*
shows *A* ≤ *B*
⟨*proof*⟩

lemma *kernel-is-closed-csubspace*[*simp*]:
assumes *a1*: *bounded-clinear f*
shows *closed-csubspace* (*f* −‘ {*0*})
⟨*proof*⟩

lemma *ccspan-closure*[*simp*]: ‹*ccspan* (*closure X*) = *ccspan X*›
⟨*proof*⟩

lemma *ccspan-finite*: ‹*space-as-set* (*ccspan X*) = *cspan X*› **if** ‹*finite X*›
⟨*proof*⟩

lemma *ccspan-UNIV*[*simp*]: ‹*ccspan UNIV* = ⊤›
⟨*proof*⟩

lemma *infsum-in-closed-csubspaceI*:
assumes ‹⋀*x*. *x*∈*X* ⟹ *f x* ∈ *A*›
assumes ‹*closed-csubspace A*›
shows ‹*infsum f X* ∈ *A*›
⟨*proof*⟩

lemma *closed-csubspace-space-as-set*[*simp*]: ‹*closed-csubspace* (*space-as-set X*)›
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *closed-sum-left-subset*: ‹$0 \in B \Longrightarrow A \subseteq A +_M B$› **for** *A B* :: -::*monoid-add*
⟨*proof*⟩

lemma *closed-sum-right-subset*: ‹$0 \in A \Longrightarrow B \subseteq A +_M B$› **for** *A B* :: -::*monoid-add*
⟨*proof*⟩

lemma *finite-cspan-closed-csubspace*:
assumes *finite* (*S*::'*a*::*complex-normed-vector set*)
shows *closed-csubspace* (*cspan S*)
⟨*proof*⟩

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*⟩

lemma *closed-subspace-closed-sum*:
fixes *A B*::('*a*::*complex-normed-vector*) *set*
assumes *a1*: ‹*csubspace A*› **and** *a2*: ‹*csubspace B*›
shows ‹*closed-csubspace* $(A +_M B)$›
⟨*proof*⟩

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*⟩

lemma *closed-sum-zero-left*[*simp*]:
fixes *A* :: ‹('*a*::{*monoid-add*, *topological-space*}) *set*›
shows ‹$\{0\} +_M A$ = *closure A*›
⟨*proof*⟩

lemma *closed-sum-zero-right*[*simp*]:
fixes A :: ‹('a::{*monoid-add*, *topological-space*}) *set*›
shows ‹$A +_M \{0\}$ = *closure* A›
⟨*proof*⟩

lemma *closed-sum-closure-right*[*simp*]:
fixes A B :: ‹'a::*real-normed-vector set*›
shows ‹$A +_M$ *closure* $B = A +_M B$›
⟨*proof*⟩

lemma *closed-sum-closure-left*[*simp*]:
fixes A B :: ‹'a::*real-normed-vector set*›
shows ‹*closure* $A +_M B = A +_M B$›
⟨*proof*⟩

lemma *closed-sum-mono-left*:
assumes ‹$A \subseteq B$›
shows ‹$A +_M C \subseteq B +_M C$›
⟨*proof*⟩

lemma *closed-sum-mono-right*:
assumes ‹$A \subseteq B$›
shows ‹$C +_M A \subseteq C +_M B$›
⟨*proof*⟩

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 λA B::'a *set*. $A +_M B$
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *closed-sum-cspan*[*simp*]:
shows ‹*cspan* $X +_M$ *cspan* Y = *closure* (*cspan* $(X \cup Y)$)›
⟨*proof*⟩

lemma *closure-image-closed-sum*:
assumes ‹*bounded-linear* U›
shows ‹*closure* (U ‘ $(A +_M B)$) = *closure* (U ‘ A) $+_M$ *closure* (U ‘ B)›
⟨*proof*⟩

lemma *ccspan-union*: *ccspan* $A \sqcup$ *ccspan* B = *ccspan* $(A \cup B)$
⟨*proof*⟩

instantiation *ccsubspace* :: (*complex-normed-vector*) *Sup*
begin
lift-definition *Sup-ccsubspace*::‹*'a ccsubspace set* ⇒ *'a ccsubspace*›
is ‹$\lambda S.$ *closure* (*complex-vector.span* (*Union S*))›
⟨*proof*⟩

instance⟨*proof*⟩
end

instance *ccsubspace* :: ({*complex-normed-vector*}) *semilattice-sup*
⟨*proof*⟩

instance *ccsubspace* :: (*complex-normed-vector*) *complete-lattice*
⟨*proof*⟩

instantiation *ccsubspace* :: (*complex-normed-vector*) *comm-monoid-add* **begin**
definition *plus-ccsubspace* :: *'a ccsubspace* ⇒ - ⇒ -
where [*simp*]: *plus-ccsubspace* = *sup*
instance
⟨*proof*⟩
end

lemma *SUP-ccspan*: ‹(*SUP x*∈*X. ccspan* (*S x*)) = *ccspan* (⋃*x*∈*X. S x*)›
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ccsubspace-Sup-empty*: *Sup* {} = (*0*::- *ccsubspace*)
⟨*proof*⟩

lemma *ccsubspace-add-right-incr*[*simp*]: $a \leq a + c$ **for** *a*::- *ccsubspace*
⟨*proof*⟩

lemma *ccsubspace-add-left-incr*[*simp*]: $a \leq c + a$ **for** *a*::- *ccsubspace*
⟨*proof*⟩

lemma *sum-bot-ccsubspace*[*simp*]: ‹(∑*x*∈*X*. ⊥) = (⊥ :: - *ccsubspace*)›
⟨*proof*⟩

7.7 Conjugate space

typedef *'a conjugate-space* = *UNIV* :: *'a set*
morphisms *from-conjugate-space to-conjugate-space* ⟨*proof*⟩
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** ‹$\lambda c\ x.\ cnj\ c *_C x$›⟨*proof*⟩
lift-definition *scaleR-conjugate-space* :: ‹*real* ⇒ *'a conjugate-space* ⇒ *'a conjugate-space*› **is** ‹$\lambda r\ x.\ r *_R x$›⟨*proof*⟩
lift-definition *plus-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* ⇒ *'a conjugate-space* **is** (+)⟨*proof*⟩
lift-definition *uminus-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* **is** ‹$\lambda x.\ -x$›⟨*proof*⟩
lift-definition *zero-conjugate-space* :: *'a conjugate-space* **is** *0*⟨*proof*⟩
lift-definition *minus-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* ⇒ *'a conjugate-space* **is** (−)⟨*proof*⟩
instance
⟨*proof*⟩
end

instantiation *conjugate-space* :: (*complex-normed-vector*) *complex-normed-vector*
begin
lift-definition *sgn-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* **is** *sgn*⟨*proof*⟩
lift-definition *norm-conjugate-space* :: *'a conjugate-space* ⇒ *real* **is** *norm*⟨*proof*⟩
lift-definition *dist-conjugate-space* :: *'a conjugate-space* ⇒ *'a conjugate-space* ⇒ *real* **is** *dist*⟨*proof*⟩
lift-definition *uniformity-conjugate-space* :: (*'a conjugate-space* × *'a conjugate-space*) *filter* **is** *uniformity*⟨*proof*⟩
lift-definition *open-conjugate-space* :: *'a conjugate-space set* ⇒ *bool* **is** *open*⟨*proof*⟩
instance
⟨*proof*⟩
end

instantiation *conjugate-space* :: (*cbanach*) *cbanach* **begin**
instance
⟨*proof*⟩
end

lemma *bounded-antilinear-to-conjugate-space*[*simp*]: ‹*bounded-antilinear to-conjugate-space*›
⟨*proof*⟩

lemma *bounded-antilinear-from-conjugate-space*[*simp*]: ‹*bounded-antilinear from-conjugate-space*›
⟨*proof*⟩

lemma *antilinear-to-conjugate-space*[*simp*]: ‹*antilinear to-conjugate-space*›
⟨*proof*⟩

lemma *antilinear-from-conjugate-space*[*simp*]: ‹*antilinear from-conjugate-space*›
⟨*proof*⟩

lemma *cspan-to-conjugate-space*[*simp*]: *cspan* (*to-conjugate-space* ‘ *X*) = *to-conjugate-space* ‘ *cspan X*
⟨*proof*⟩

lemma *surj-to-conjugate-space*[*simp*]: *surj to-conjugate-space*
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *norm-from-conjugate-space*[*simp*]: ‹*norm* (*from-conjugate-space x*) = *norm*
x›
⟨*proof*⟩

lemma *closure-to-conjugate-space*: ‹*closure* (*to-conjugate-space* ‘ *X*) = *to-conjugate-space*
‘ *closure X*›
⟨*proof*⟩

lemma *closure-from-conjugate-space*: ‹*closure* (*from-conjugate-space* ‘ *X*) = *from-conjugate-space*
‘ *closure X*›
⟨*proof*⟩

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: ‹$\bigwedge x.\ x \in G \Longrightarrow A\ x = B\ x$› **and** *t*: ‹$t \in$ *closure* (*cspan G*)›
shows ‹$A\ t = B\ t$›
⟨*proof*⟩

7.8 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*)
⟨*proof*⟩

lemma *snd-scaleC* [*simp*]: *snd* (*scaleC r A*) = *scaleC r* (*snd A*)
⟨*proof*⟩

proposition *scaleC-Pair* [*simp*]: *scaleC r* (*a*, *b*) = (*scaleC r a*, *scaleC r b*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *module-prod-scale-eq-scaleC*: *module-prod.scale* $(*_C)$ $(*_C)$ = *scaleC*
⟨*proof*⟩

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*
⟨*proof*⟩

instance *prod* :: (*complex-normed-vector*, *complex-normed-vector*) *complex-normed-vector*

⟨*proof*⟩

lemma *cspan-Times*: ‹*cspan* (*S* × *T*) = *cspan S* × *cspan T*› **if** ‹*0* ∈ *S*› **and** ‹*0* ∈ *T*›
⟨*proof*⟩

lemma *onorm-case-prod-plus*: ‹*onorm* (*case-prod plus* :: - ⇒ ′*a*::{*real-normed-vector*, *not-singleton*}) = *sqrt 2*›
⟨*proof*⟩

7.9 Copying existing theorems into sublocales

context *bounded-clinear* **begin**
interpretation *bounded-linear f* ⟨*proof*⟩
lemmas *continuous* = *real.continuous*
lemmas *uniform-limit* = *real.uniform-limit*
lemmas *Cauchy* = *real.Cauchy*
end

context *bounded-antilinear* **begin**
interpretation *bounded-linear f* ⟨*proof*⟩
lemmas *continuous* = *real.continuous*
lemmas *uniform-limit* = *real.uniform-limit*
end

context *bounded-cbilinear* **begin**
interpretation *bounded-bilinear prod* ⟨*proof*⟩
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* ⟨*proof*⟩
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*.

⟨*ML*⟩

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*
⟨*proof*⟩

lemma *cinner-minus-left* [*simp*]: *cinner* (− *x*) *y* = − *cinner x y*
⟨*proof*⟩

lemma *cinner-diff-left*: *cinner* (*x* − *y*) *z* = *cinner x z* − *cinner y z*
⟨*proof*⟩

lemma *cinner-sum-left*: *cinner* (∑ *x*∈*A*. *f x*) *y* = (∑ *x*∈*A*. *cinner* (*f x*) *y*)
⟨*proof*⟩

lemma *call-zero-iff* [*simp*]: (∀ *u*. *cinner x u* = *0*) ⟷ (*x* = *0*)
⟨*proof*⟩

Transfer distributivity rules to right argument.

lemma *cinner-add-right*: *cinner x* (*y* + *z*) = *cinner x y* + *cinner x z*
⟨*proof*⟩

lemma *cinner-scaleC-right* [*simp*]: *cinner x* (*scaleC r y*) = *r* ∗ (*cinner x y*)
⟨*proof*⟩

lemma *cinner-zero-right* [*simp*]: *cinner x 0* = *0*
⟨*proof*⟩

lemma *cinner-minus-right* [*simp*]: *cinner x* (− *y*) = − *cinner x y*
⟨*proof*⟩

lemma *cinner-diff-right*: *cinner x* (*y* − *z*) = *cinner x y* − *cinner x z*
⟨*proof*⟩

lemma *cinner-sum-right*: *cinner x* (∑ *y*∈*A*. *f y*) = (∑ *y*∈*A*. *cinner x* (*f y*))
⟨*proof*⟩

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* ⟷ *x* ≠ *0*
⟨*proof*⟩

lemma *power2-norm-eq-cinner*:
shows (*complex-of-real* (*norm x*))^2 = (*cinner x x*)
⟨*proof*⟩

lemma *power2-norm-eq-cinner′*:
shows (*norm x*)^2 = *Re* (*cinner x x*)

⟨*proof*⟩

Identities involving real multiplication and division.

lemma *cinner-mult-left*: *cinner* (*of-complex m* ∗ *a*) *b* = *cnj m* ∗ (*cinner a b*)
⟨*proof*⟩

lemma *cinner-mult-right*: *cinner a* (*of-complex m* ∗ *b*) = *m* ∗ (*cinner a b*)
⟨*proof*⟩

lemma *cinner-mult-left'*: *cinner* (*a* ∗ *of-complex m*) *b* = *cnj m* ∗ (*cinner a b*)
⟨*proof*⟩

lemma *cinner-mult-right'*: *cinner a* (*b* ∗ *of-complex m*) = (*cinner a b*) ∗ *m*
⟨*proof*⟩

lemma *Cauchy-Schwarz-ineq*:
(*cinner x y*) ∗ (*cinner y x*) ≤ *cinner x x* ∗ *cinner y y*
⟨*proof*⟩

lemma *Cauchy-Schwarz-ineq2*:
shows *norm* (*cinner x y*) ≤ *norm x* ∗ *norm y*
⟨*proof*⟩

subclass *complex-normed-vector*
⟨*proof*⟩

end

lemma *csquare-continuous*:
fixes *e* :: *real*
shows *e* > *0* ⟹ ∃ *d*. *0* < *d* ∧ (∀ *y*. *cmod* (*y* − *x*) < *d* ⟶ *cmod* (*y* ∗ *y* − *x* ∗ *x*) < *e*)
⟨*proof*⟩

lemma *cnorm-le*: *norm x* ≤ *norm y* ⟷ *cinner x x* ≤ *cinner y y*
⟨*proof*⟩

lemma *cnorm-lt*: *norm x* < *norm y* ⟷ *cinner x x* < *cinner y y*
⟨*proof*⟩

lemma *cnorm-eq*: *norm x = norm y ⟷ cinner x x = cinner y y*
⟨*proof*⟩

lemma *cnorm-eq-1*: *norm x = 1 ⟷ cinner x x = 1*
⟨*proof*⟩

lemma *cinner-divide-left*:
fixes *a* :: *'a* :: {*complex-inner*,*complex-div-algebra*}
shows *cinner* (*a* / *of-complex m*) *b* = (*cinner a b*) / *cnj m*
⟨*proof*⟩

lemma *cinner-divide-right*:
fixes *a* :: *'a* :: {*complex-inner*,*complex-div-algebra*}
shows *cinner a* (*b* / *of-complex m*) = (*cinner a b*) / *m*
⟨*proof*⟩

Re-enable constraints for *open*, *uniformity*, *dist*, and *norm*.

⟨*ML*⟩

lemma *bounded-sesquilinear-cinner*:
bounded-sesquilinear (*cinner*::*'a*::*complex-inner* ⇒ *'a* ⇒ *complex*)
⟨*proof*⟩

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*) $\Longrightarrow$ *g differentiable at x within s* $\Longrightarrow$ (λx. *cinner* (*f x*) (*g x*)) *differentiable at x within s*
⟨*proof*⟩

8.2 Class instances

instantiation *complex* :: *complex-inner*
begin

definition *cinner-complex-def* [*simp*]: *cinner x y* = *cnj x* ∗ *y*

instance
⟨*proof*⟩

end

lemma
shows *complex-inner-1-left*[*simp*]: *cinner 1 x* = *x*
and *complex-inner-1-right*[*simp*]: *cinner x 1* = *cnj x*
⟨*proof*⟩

lemma *cdot-square-norm*: *cinner x x* = *complex-of-real* ((*norm x*)^2)
⟨*proof*⟩

lemma *cnorm-eq-square*: *norm x* = *a* ⟷ *0* ≤ *a* ∧ *cinner x x* = *complex-of-real* (a^2)
⟨*proof*⟩

lemma *cnorm-le-square*: *norm x* ≤ *a* ⟷ *0* ≤ *a* ∧ *cinner x x* ≤ *complex-of-real* (a^2)
⟨*proof*⟩

lemma *cnorm-ge-square*: *norm x* ≥ *a* ⟷ *a* ≤ *0* ∨ *cinner x x* ≥ *complex-of-real* (a^2)
⟨*proof*⟩

lemma *norm-lt-square*: *norm x* < *a* ⟷ *0* < *a* ∧ *cinner x x* < *complex-of-real* (a^2)
⟨*proof*⟩

lemma *norm-gt-square*: *norm x* > *a* ⟷ *a* < *0* ∨ *cinner x x* > *complex-of-real* (a^2)
⟨*proof*⟩

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*: $cinner\ x\ y = ((norm\ (x+y))^2 - (norm\ (x-y))^2 - \mathrm{i} * (norm\ (x + \mathrm{i} *_C\ y))^2 + \mathrm{i} * (norm\ (x - \mathrm{i} *_C\ y))^2)\ /\ 4$
⟨*proof*⟩

lemma *of-complex-inner-1* [*simp*]:
cinner (*of-complex x*) (*1* :: *'a* :: {*complex-inner*, *complex-normed-algebra-1*}) = *cnj x*
⟨*proof*⟩

lemma *summable-of-complex-iff*:
summable (λ*x*. *of-complex* (*f x*) :: *'a* :: {*complex-normed-algebra-1*,*complex-inner*}) ⟷ *summable f*
⟨*proof*⟩

8.3 Gradient derivative

definition
cgderiv :: [*'a*::*complex-inner* ⇒ *complex*, *'a*, *'a*] ⇒ *bool*
((*cGDERIV* (-)/ (-)/ :> (-)) [*1000*, *1000*, *60*] *60*)
where

cGDERIV f x :> *D* ⟷ *FDERIV f x* :> *cinner D*

lemma *cgderiv-deriv* [*simp*]: *cGDERIV f x* :> *D* ⟷ *DERIV f x* :> *cnj D*
⟨*proof*⟩

lemma *cGDERIV-DERIV-compose*:
assumes *cGDERIV f x* :> *df* **and** *DERIV g* (*f x*) :> *cnj dg*
shows *cGDERIV* (λ*x*. *g* (*f x*)) *x* :> *scaleC dg df*
⟨*proof*⟩

lemma *cGDERIV-subst*: ⟦*cGDERIV f x* :> *df*; *df* = *d*⟧ ⟹ *cGDERIV f x* :> *d*
⟨*proof*⟩

lemma *cGDERIV-const*: *cGDERIV* (λ*x*. *k*) *x* :> *0*
⟨*proof*⟩

lemma *cGDERIV-add*:
⟦*cGDERIV f x* :> *df*; *cGDERIV g x* :> *dg*⟧
⟹ *cGDERIV* (λ*x*. *f x* + *g x*) *x* :> *df* + *dg*
⟨*proof*⟩

lemma *cGDERIV-minus*:
  $cGDERIV\ f\ x :> df \Longrightarrow cGDERIV\ (\lambda x.\ -\ f\ x)\ x :> -\ df$
  ⟨*proof*⟩

lemma *cGDERIV-diff*:
  $[\![cGDERIV\ f\ x :> df;\ cGDERIV\ g\ x :> dg]\!]$
  $\Longrightarrow cGDERIV\ (\lambda x.\ f\ x - g\ x)\ x :> df - dg$
  ⟨*proof*⟩

lemma *cGDERIV-scaleC*:
  $[\![DERIV\ f\ x :> df;\ cGDERIV\ g\ x :> dg]\!]$
  $\Longrightarrow cGDERIV\ (\lambda x.\ scaleC\ (f\ x)\ (g\ x))\ x$
  $:> (scaleC\ (cnj\ (f\ x))\ dg + scaleC\ (cnj\ df)\ (cnj\ (g\ x)))$
  ⟨*proof*⟩

lemma *GDERIV-mult*:
  $[\![cGDERIV\ f\ x :> df;\ cGDERIV\ g\ x :> dg]\!]$
  $\Longrightarrow cGDERIV\ (\lambda x.\ f\ x * g\ x)\ x :> cnj\ (f\ x) *_C dg + cnj\ (g\ x) *_C df$
  ⟨*proof*⟩

lemma *cGDERIV-inverse*:
  $[\![cGDERIV\ f\ x :> df;\ f\ x \neq 0]\!]$
  $\Longrightarrow cGDERIV\ (\lambda x.\ inverse\ (f\ x))\ x :> -\ cnj\ ((inverse\ (f\ x))^2) *_C df$
  ⟨*proof*⟩

lemma *has-derivative-norm*[*derivative-intros*]:
  fixes *x* :: *'a*::*complex-inner*
  assumes $x \neq 0$
  shows (*norm has-derivative* ($\lambda h.$ *Re* (*cinner* (*sgn x*) *h*))) (*at x*)
  thm *has-derivative-norm*
⟨*proof*⟩

bundle *cinner-syntax* **begin**
notation *cinner* (**infix** $\cdot_C$ *70*)
end

bundle *no-cinner-syntax* **begin**
no-notation *cinner* (**infix** $\cdot_C$ *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* ∈ ℝ
  ⟨*proof*⟩

lemmas *cinner-commute′* [*simp*] = *cinner-commute*[*symmetric*]

lemma (**in** *complex-inner*) *cinner-eq-flip*: ‹(*cinner x y* = *cinner z w*) ⟷ (*cinner y x* = *cinner w z*)›
  ⟨*proof*⟩

lemma *Im-cinner-x-x*[*simp*]: *Im* ($x \cdot_C x$) = *0*
  ⟨*proof*⟩

lemma *of-complex-inner-1′* [*simp*]:
  cinner (*1* :: ′*a* :: {*complex-inner*, *complex-normed-algebra-1*}) (*of-complex x*) = *x*
  ⟨*proof*⟩

class *chilbert-space* = *complex-inner* + *complete-space*
begin
subclass *cbanach* ⟨*proof*⟩
end

instantiation *complex* :: *chilbert-space* **begin**
instance ⟨*proof*⟩
end

9.2 Misc facts

lemma *cinner-scaleR-left* [*simp*]: *cinner* (*scaleR r x*) *y* = *of-real r* ∗ (*cinner x y*)
  ⟨*proof*⟩

lemma *cinner-scaleR-right* [*simp*]: *cinner x* (*scaleR r y*) = *of-real r* ∗ (*cinner x y*)
  ⟨*proof*⟩

This is a useful rule for establishing the equality of vectors

lemma *cinner-extensionality*:

assumes ‹$\bigwedge\gamma.\ \gamma \cdot_C \psi = \gamma \cdot_C \varphi$›
shows ‹$\psi = \varphi$›
⟨*proof*⟩

lemma *polar-identity*:
includes *notation-norm*
shows ‹$\|x + y\|\hat{}2 = \|x\|\hat{}2 + \|y\|\hat{}2 + 2 * Re\ (x \cdot_C y)$›
— Shown in the proof of Corollary 1.5 in [1]
⟨*proof*⟩

lemma *polar-identity-minus*:
includes *notation-norm*
shows ‹$\|x - y\|\hat{}2 = \|x\|\hat{}2 + \|y\|\hat{}2 - 2 * Re\ (x \cdot_C y)$›
⟨*proof*⟩

proposition *parallelogram-law*:
includes *notation-norm*
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]
⟨*proof*⟩

theorem *pythagorean-theorem*:
includes *notation-norm*
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]
⟨*proof*⟩

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*⟩

lemma *Cauchy-cinner-Cauchy*:
fixes $x\ y$ :: ‹$nat \Rightarrow 'a$::*complex-inner*›
assumes *a1*: ‹*Cauchy x*› **and** *a2*: ‹*Cauchy y*›
shows ‹$Cauchy\ (\lambda\ n.\ x\ n \cdot_C y\ n)$›
⟨*proof*⟩

lemma *cinner-sup-norm*: ‹$norm\ \psi = (SUP\ \varphi.\ cmod\ (cinner\ \varphi\ \psi)\ /\ norm\ \varphi)$›
⟨*proof*⟩

lemma *cinner-sup-onorm*:
fixes A :: ‹$'a$::{*real-normed-vector*,*not-singleton*} $\Rightarrow$ $'b$::*complex-inner*›
assumes ‹*bounded-linear A*›

shows ‹*onorm A* = (*SUP* (*ψ*,*φ*). *cmod* (*cinner ψ* (*A φ*)) / (*norm ψ* ∗ *norm φ*))›
⟨*proof*⟩

lemma *sum-cinner*:
fixes *f* :: *'a* ⇒ *'b*::*complex-inner*
shows *cinner* (*sum f A*) (*sum g B*) = (∑ *i*∈*A*. ∑ *j*∈*B*. *cinner* (*f i*) (*g j*))
⟨*proof*⟩

lemma *Cauchy-cinner-product-summable'*:
fixes *a b* :: *nat* ⇒ *'a*::*complex-inner*
shows ‹(λ(*x*, *y*). *cinner* (*a x*) (*b y*)) *summable-on UNIV* ⟷ (λ(*x*, *y*). *cinner* (*a y*) (*b* (*x* − *y*))) *summable-on* {(*k*, *i*). *i* ≤ *k*}›
⟨*proof*⟩

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*⟩

end

lemma *sgn-cinner*[*simp*]: ‹*sgn ψ* •$_C$ *ψ* = *norm ψ*›
⟨*proof*⟩

instance *prod* :: (*chilbert-space*, *chilbert-space*) *chilbert-space*⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *orthogonal-complement-orthoI'*:
‹*x* ∈ *M* ⟹ *y* ∈ *orthogonal-complement M* ⟹ *x* •$_C$ *y* = *0*›
⟨*proof*⟩

lemma *orthogonal-complementI*:
‹(⋀*x*. *x* ∈ *M* ⟹ *y* •$_C$ *x* = *0*) ⟹ *y* ∈ *orthogonal-complement M*›
⟨*proof*⟩

abbreviation *is-orthogonal*::‹*'a*::*complex-inner* ⇒ *'a* ⇒ *bool*› **where**
‹*is-orthogonal x y* ≡ *x* •$_C$ *y* = *0*›

bundle *orthogonal-notation* **begin**
notation *is-orthogonal* (**infixl** ⊥ *69*)
end

bundle *no-orthogonal-notation* **begin**
no-notation *is-orthogonal* (**infixl** ⊥ *69*)
end

lemma *is-orthogonal-sym*: *is-orthogonal* ψ φ = *is-orthogonal* φ ψ
⟨*proof*⟩

lemma *is-orthogonal-sgn-right*[*simp*]: ‹*is-orthogonal e* (*sgn f*) ⟷ *is-orthogonal e f*›
⟨*proof*⟩

lemma *is-orthogonal-sgn-left*[*simp*]: ‹*is-orthogonal* (*sgn e*) *f* ⟷ *is-orthogonal e f*›
⟨*proof*⟩

lemma *orthogonal-complement-closed-subspace*[*simp*]:
closed-csubspace (*orthogonal-complement A*)
for *A* :: ‹(′*a*::*complex-inner*) *set*›
⟨*proof*⟩

lemma *orthogonal-complement-zero-intersection*:
assumes *0*∈*M*
shows ‹*M* ∩ (*orthogonal-complement M*) = {*0*}›
⟨*proof*⟩

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*⟩

instantiation *ccsubspace* :: (*complex-inner*) *uminus*
begin
lift-definition *uminus-ccsubspace*::‹′*a ccsubspace* ⇒ ′*a ccsubspace*›
is ‹*orthogonal-complement*›
⟨*proof*⟩

instance ⟨*proof*⟩
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.
⟨*proof*⟩

instantiation *ccsubspace* :: (*complex-inner*) *minus* **begin**
lift-definition *minus-ccsubspace* :: *'a ccsubspace* ⇒ *'a ccsubspace* ⇒ *'a ccsubspace*
is $\lambda A\ B.\ A \cap$ (*orthogonal-complement B*)
⟨*proof*⟩
instance⟨*proof*⟩
end

definition *is-ortho-set* :: *'a::complex-inner set* ⇒ *bool* **where**
— Orthogonal set
‹*is-ortho-set S* ⟷ (∀ *x*∈*S*. ∀ *y*∈*S*. *x* ≠ *y* ⟶ (*x* $\bullet_C$ *y*) = *0*) ∧ *0* ∉ *S*›

definition *is-onb* **where** ‹*is-onb E* ⟷ *is-ortho-set E* ∧ (∀ *b*∈*E*. *norm b* = *1*) ∧ *ccspan E* = *top*›

lemma *is-ortho-set-empty*[*simp*]: *is-ortho-set* {}
⟨*proof*⟩

lemma *is-ortho-set-antimono*: ‹*A* ⊆ *B* ⟹ *is-ortho-set B* ⟹ *is-ortho-set A*›
⟨*proof*⟩

lemma *orthogonal-complement-of-closure*:
fixes *A* ::(*'a::complex-inner*) *set*
shows *orthogonal-complement A* = *orthogonal-complement* (*closure A*)
⟨*proof*⟩

lemma *is-orthogonal-closure*:
assumes ‹⋀*s*. *s* ∈ *S* ⟹ *is-orthogonal a s*›
assumes ‹*x* ∈ *closure S*›
shows ‹*is-orthogonal a x*›
⟨*proof*⟩

lemma *is-orthogonal-cspan*:
assumes *a1*: ⋀*s*. *s* ∈ *S* ⟹ *is-orthogonal a s* **and** *a3*: *x* ∈ *cspan S*
shows *is-orthogonal a x*
⟨*proof*⟩

lemma *ccspan-leq-ortho-ccspan*:
assumes ⋀*s t*. *s*∈*S* ⟹ *t*∈*T* ⟹ *is-orthogonal s t*
shows *ccspan S* ≤ − (*ccspan T*)
⟨*proof*⟩

lemma *double-orthogonal-complement-increasing*[*simp*]:
shows *M* ⊆ *orthogonal-complement* (*orthogonal-complement M*)
⟨*proof*⟩

lemma *orthonormal-basis-of-cspan*:
fixes $S::'a::complex\text{-}inner\ set$
assumes *finite S*
shows $\exists A.\ is\text{-}ortho\text{-}set\ A \wedge (\forall x{\in}A.\ norm\ x = 1) \wedge cspan\ A = cspan\ S \wedge finite\ A$
⟨*proof*⟩

lemma *is-ortho-set-cindependent*:
assumes *is-ortho-set A*
shows *cindependent A*
⟨*proof*⟩

lemma *onb-expansion-finite*:
includes *notation-norm*
fixes $T::\langle 'a::\{complex\text{-}inner,cfinite\text{-}dim\}\ set\rangle$
assumes *a1*: ‹$cspan\ T = UNIV$› **and** *a3*: ‹$is\text{-}ortho\text{-}set\ T$›
and *a4*: ‹$\bigwedge t.\ t{\in}T \Longrightarrow \|t\| = 1$›
shows ‹$x = (\sum t{\in}T.\ (t \bullet_C x) *_C t)$›
⟨*proof*⟩

lemma *is-ortho-set-singleton*[*simp*]: ‹$is\text{-}ortho\text{-}set\ \{x\} \longleftrightarrow x \neq 0$›
⟨*proof*⟩

lemma *orthogonal-complement-antimono*[*simp*]:
fixes $A\ B :: \langle('a::complex\text{-}inner)\ set\rangle$
assumes $A \supseteq B$
shows ‹$orthogonal\text{-}complement\ A \subseteq orthogonal\text{-}complement\ B$›
⟨*proof*⟩

lemma *orthogonal-complement-UNIV*[*simp*]:
$orthogonal\text{-}complement\ UNIV = \{0\}$
⟨*proof*⟩

lemma *orthogonal-complement-zero*[*simp*]:
$orthogonal\text{-}complement\ \{0\} = UNIV$
⟨*proof*⟩

lemma *mem-ortho-ccspanI*:
assumes ‹$\bigwedge y.\ y \in S \Longrightarrow is\text{-}orthogonal\ x\ y$›
shows ‹$x \in space\text{-}as\text{-}set\ (-\ ccspan\ S)$›
⟨*proof*⟩

9.4 Projections

lemma *smallest-norm-exists*:
— Theorem 2.5 in [1] (inside the proof)

includes *notation-norm*
fixes M :: ‹′*a*::*chilbert-space set*›
assumes *q1*: ‹*convex M*› **and** *q2*: ‹*closed M*› **and** *q3*: ‹$M \neq \{\}$›
shows ‹$\exists k$. *is-arg-min* (λ x. $\|x\|$) (λ t. $t \in M$) k›
⟨*proof*⟩

lemma *smallest-norm-unique*:
— Theorem 2.5 in [1] (inside the proof)
includes *notation-norm*
fixes M :: ‹′*a*::*complex-inner set*›
assumes *q1*: ‹*convex M*›
assumes *r*: ‹*is-arg-min* (λ x. $\|x\|$) (λ t. $t \in M$) r›
assumes *s*: ‹*is-arg-min* (λ x. $\|x\|$) (λ t. $t \in M$) s›
shows ‹$r = s$›
⟨*proof*⟩

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*: ‹$M \neq \{\}$›
shows ‹$\exists k$. *is-arg-min* (λ x. *dist x h*) (λ x. $x \in M$) k›
⟨*proof*⟩

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. $x \in M$) r›
assumes ‹*is-arg-min* (λ x. *dist x h*) (λ x. $x \in M$) s›
shows ‹$r = s$›
⟨*proof*⟩
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. $x \in M$) k) $\longleftrightarrow$
$h - k \in$ *orthogonal-complement* $M \wedge k \in M$›
⟨*proof*⟩
include *notation-norm*
⟨*proof*⟩

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*⟩

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 ‹$r = s$›
⟨*proof*⟩

definition *is-projection-on*::‹$('a \Rightarrow 'a) \Rightarrow ('a$::*metric-space*) *set* $\Rightarrow$ *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* $\wedge$ π $h \in M$)›
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\pi_1 = \pi_2$
⟨*proof*⟩

definition *projection* :: ‹*'a::metric-space set* $\Rightarrow$ $('a \Rightarrow 'a)$› **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*
⟨*proof*⟩

lemma *projection-is-projection-on'*[*simp*]:
— Common special case of ⟦*convex ?M*; *closed ?M*; $?M \neq \{\}$⟧ $\Longrightarrow$ *is-projection-on* (*projection ?M*) *?M*
fixes *M* :: ‹*'a::chilbert-space set*›
assumes ‹*closed-csubspace M*›
shows *is-projection-on* (*projection M*) *M*
⟨*proof*⟩

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*›

⟨proof⟩

lemma *is-projection-on-in-image*:
assumes *is-projection-on* π *M*
shows π *h* $\in$ *M*
⟨proof⟩

lemma *is-projection-on-image*:
assumes *is-projection-on* π *M*
shows *range* π = *M*
⟨proof⟩

lemma *projection-in-image*[*simp*]:
fixes *M* :: ‹′a::*chilbert-space set*›
assumes ‹*convex M*› **and** ‹*closed M*› **and** ‹*M* ≠ {}›
shows ‹*projection M h* ∈ *M*›
⟨proof⟩

lemma *projection-image*[*simp*]:
fixes *M* :: ‹′a::*chilbert-space set*›
assumes ‹*convex M*› **and** ‹*closed M*› **and** ‹*M* ≠ {}›
shows ‹*range* (*projection M*) = *M*›
⟨proof⟩

lemma *projection-eqI′*:
fixes *M* :: ‹′a::*complex-inner set*›
assumes ‹*convex M*›
assumes ‹*is-projection-on f M*›
shows ‹*projection M* = *f*›
⟨proof⟩

lemma *is-projection-on-eqI*:
fixes *M* :: ‹′a::*complex-inner set*›
assumes *a1*: ‹*closed-csubspace M*› **and** *a2*: ‹*h* − *x* ∈ *orthogonal-complement M*›
and *a3*: ‹*x* ∈ *M*›
and *a4*: ‹*is-projection-on* π *M*›
shows ‹π *h* = *x*›
⟨proof⟩

lemma *projection-eqI*:
fixes *M* :: ‹(′a::*chilbert-space*) *set*›
assumes ‹*closed-csubspace M*› **and** ‹*h* − *x* ∈ *orthogonal-complement M*› **and** ‹*x* ∈ *M*›
shows ‹*projection M h* = *x*›
⟨proof⟩

lemma *is-projection-on-fixes-image*:
fixes *M* :: ‹′a::*metric-space set*›
assumes *a1*: *is-projection-on* π *M* **and** *a3*: *x* ∈ *M*

shows $\pi\ x = x$
⟨*proof*⟩

lemma *projection-fixes-image*:
fixes M :: ‹(′*a*::*chilbert-space*) *set*›
assumes *closed-csubspace M* **and** $x \in M$
shows *projection M x* = *x*
⟨*proof*⟩

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*⟩

proposition *is-projection-on-reduces-norm*:
includes *notation-norm*
fixes M :: ‹(′*a*::*complex-inner*) *set*›
assumes ‹*is-projection-on* π *M*› **and** ‹*closed-csubspace M*›
shows ‹$\|\ \pi\quad h\ \| \leq \|\ h\ \|$›
⟨*proof*⟩

proposition *projection-reduces-norm*:
includes *notation-norm*
fixes M :: ‹′*a*::*chilbert-space set*›
assumes *a1*: *closed-csubspace M*
shows ‹‖ *projection M h* ‖ ≤ ‖ *h* ‖›
⟨*proof*⟩
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*⟩

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]
⟨*proof*⟩

proposition *is-projection-on-idem*:
fixes M :: ‹(′*a*::*complex-inner*) *set*›
assumes *is-projection-on* π *M*
shows $\pi\ (\pi\ x) = \pi\ x$
⟨*proof*⟩

proposition *projection-idem*:
fixes M :: ′*a*::*chilbert-space set*

assumes *a1*: *closed-csubspace M*
shows *projection M* (*projection M x*) = *projection M x*
⟨*proof*⟩

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*⟩
proposition *projection-kernel-is-orthogonal-complement*:
fixes *M* :: ‹′*a*::*chilbert-space set*›
assumes *closed-csubspace M*
shows (*projection M*) −‘ {*0*} = (*orthogonal-complement M*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-projection-on-zero*:
is-projection-on (λ-. *0*) {*0*}
⟨*proof*⟩

lemma *projection-zero*[*simp*]:
projection {*0*} = (λ-. *0*)
⟨*proof*⟩

lemma *is-projection-on-rank1*:
fixes *t* :: ‹′*a*::*complex-inner*›
shows ‹*is-projection-on* (λ*x*. ((*t* $\bullet_C$ *x*) / (*t* $\bullet_C$ *t*)) $*_C$ *t*) (*cspan* {*t*})›
⟨*proof*⟩

lemma *projection-rank1*:
fixes *t x* :: ‹′*a*::*complex-inner*›
shows ‹*projection* (*cspan* {*t*}) *x* = ((*t* $\bullet_C$ *x*) / (*t* $\bullet_C$ *t*)) $*_C$ *t*›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *de-morgan-orthogonal-complement-inter*:
  fixes *A B*::′*a*::*chilbert-space set*
  assumes *a1*: ‹*closed-csubspace A*› **and** *a2*: ‹*closed-csubspace B*›
  shows ‹*orthogonal-complement* (*A* ∩ *B*) = *orthogonal-complement A* $+_M$ *orthogonal-complement B*›
⟨*proof*⟩

lemma *orthogonal-complement-of-cspan*: ‹*orthogonal-complement A* = *orthogonal-complement* (*cspan A*)›
  ⟨*proof*⟩

lemma *orthogonal-complement-orthogonal-complement-closure-cspan*:
  ‹*orthogonal-complement* (*orthogonal-complement S*) = *closure* (*cspan S*)› **for** *S* :: ‹′*a*::*chilbert-space set*›
⟨*proof*⟩

instance *ccsubspace* :: (*chilbert-space*) *complete-orthomodular-lattice*
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *orthogonal-bot*[*simp*]: ‹*orthogonal-spaces S bot*›
⟨*proof*⟩

lemma *orthogonal-spaces-sym*: ‹*orthogonal-spaces S T* ⟹ *orthogonal-spaces T S*›
⟨*proof*⟩

lemma *orthogonal-sup*: ‹*orthogonal-spaces S T1* ⟹ *orthogonal-spaces S T2* ⟹ *orthogonal-spaces S* (*sup T1 T2*)›
⟨*proof*⟩

lemma *orthogonal-sum*:
assumes ‹*finite F*› **and** ‹⋀*x*. *x*∈*F* ⟹ *orthogonal-spaces S* (*T x*)›
shows ‹*orthogonal-spaces S* (*sum T F*)›
⟨*proof*⟩

lemma *orthogonal-spaces-ccspan*: ‹(∀ *x*∈*S*. ∀ *y*∈*T*. *is-orthogonal x y*) ⟷ *orthogonal-spaces* (*ccspan S*) (*ccspan T*)›
⟨*proof*⟩

9.7 Orthonormal bases

lemma *ortho-basis-exists*:
fixes *S* :: ‹′*a*::*chilbert-space set*›
assumes ‹*is-ortho-set S*›
shows ‹∃ *B*. *B* ⊇ *S* ∧ *is-ortho-set B* ∧ *closure* (*cspan B*) = *UNIV*›
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

lemma *is-ortho-set-some-chilbert-basis*[*simp*]: ‹*is-ortho-set some-chilbert-basis*›
⟨*proof*⟩

lemma *is-normal-some-chilbert-basis*: ‹⋀*x*. *x* ∈ *some-chilbert-basis* ⟹ *norm x* = *1*›

⟨*proof*⟩

lemma *ccspan-some-chilbert-basis*[*simp*]: ‹*ccspan some-chilbert-basis* = *top*›
⟨*proof*⟩

lemma *span-some-chilbert-basis*[*simp*]: ‹*closure* (*cspan some-chilbert-basis*) = *UNIV*›
⟨*proof*⟩

lemma *cindependent-some-chilbert-basis*[*simp*]: ‹*cindependent some-chilbert-basis*›
⟨*proof*⟩

lemma *finite-some-chilbert-basis*[*simp*]: ‹*finite* (*some-chilbert-basis* :: *'a* :: {*chilbert-space*, *cfinite-dim*} *set*)›
⟨*proof*⟩

lemma *some-chilbert-basis-nonempty*: ‹(*some-chilbert-basis* :: *'a*::{*chilbert-space*, *not-singleton*} *set*) ≠ {}›
⟨*proof*⟩

lemma *basis-projections-reconstruct-has-sum*:
assumes ‹*is-ortho-set B*› **and** *normB*: ‹⋀*b*. *b*∈*B* ⟹ *norm b* = *1*› **and** *ψB*: ‹*ψ* ∈ *space-as-set* (*ccspan B*)›
shows ‹((*λb*. (*b* $\cdot_C$ *ψ*) $*_C$ *b*) *has-sum ψ*) *B*›
⟨*proof*⟩

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*∈*B*. (*b* $\cdot_C$ *ψ*) $*_C$ *b*) = *ψ*›
⟨*proof*⟩

lemma *basis-projections-reconstruct-summable*:
assumes ‹*is-ortho-set B*› **and** ‹⋀*b*. *b*∈*B* ⟹ *norm b* = *1*› **and** ‹*ψ* ∈ *space-as-set* (*ccspan B*)›
shows ‹(*λb*. (*b* $\cdot_C$ *ψ*) $*_C$ *b*) *summable-on B*›
⟨*proof*⟩

lemma *parseval-identity-has-sum*:
assumes ‹*is-ortho-set B*› **and** *normB*: ‹⋀*b*. *b*∈*B* ⟹ *norm b* = *1*› **and** ‹*ψ* ∈ *space-as-set* (*ccspan B*)›
shows ‹((*λb*. (*norm* (*b* $\cdot_C$ *ψ*))^2) *has-sum* (*norm ψ*)^2) *B*›
⟨*proof*⟩

lemma *parseval-identity-summable*:
assumes ‹*is-ortho-set B*› **and** ‹⋀*b*. *b*∈*B* ⟹ *norm b* = *1*› **and** ‹*ψ* ∈ *space-as-set* (*ccspan B*)›
shows ‹(*λb*. (*norm* (*b* $\cdot_C$ *ψ*))^2) *summable-on B*›
⟨*proof*⟩

lemma *parseval-identity*:
assumes ‹*is-ortho-set B*› **and** ‹$\bigwedge$*b. b*∈*B* ⟹ *norm b* = *1*› **and** ‹ψ ∈ *space-as-set* (*ccspan B*)›
shows ‹($\sum_\infty$*b*∈*B*. (*norm* (*b* $\cdot_C$ ψ))^2) = (*norm* ψ)^2›
⟨*proof*⟩

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*⟩

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* $\cdot_C$ *x*›
⟨*proof*⟩

lemma *riesz-representation-unique*:
— Theorem 3.4 in [1]
fixes *f*::‹*'a*::*complex-inner* ⇒ *complex*›
assumes ‹$\bigwedge$*x*. *f x* = (*t* $\cdot_C$ *x*)›
assumes ‹$\bigwedge$*x*. *f x* = (*u* $\cdot_C$ *x*)›
shows ‹*t* = *u*›
⟨*proof*⟩

9.9 Adjoints

definition *is-cadjoint F G* ⟷ (∀ *x*. ∀ *y*. (*F x* $\cdot_C$ *y*) = (*x* $\cdot_C$ *G y*))

lemma *is-adjoint-sym*:
‹*is-cadjoint F G* ⟹ *is-cadjoint G F*›
⟨*proof*⟩

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 *notation-norm*
⟨*proof*⟩

lemma *cadjoint-is-cadjoint*[*simp*]:
fixes *G* :: *'b*::*chilbert-space* ⇒ *'a*::*complex-inner*

assumes [*simp*]: ‹*bounded-clinear G*›
shows ‹*is-cadjoint* (*cadjoint G*) *G*›
⟨*proof*⟩

lemma *is-cadjoint-unique*:
assumes ‹*is-cadjoint F1 G*›
assumes ‹*is-cadjoint F2 G*›
shows ‹*F1* = *F2*›
⟨*proof*⟩

lemma *cadjoint-univ-prop*:
fixes *G* :: *'b::chilbert-space* ⇒ *'a::complex-inner*
assumes *a1*: ‹*bounded-clinear G*›
shows ‹*cadjoint G x* $\cdot_C$ *y* = *x* $\cdot_C$ *G y*›
⟨*proof*⟩

lemma *cadjoint-univ-prop'*:
fixes *G* :: *'b::chilbert-space* ⇒ *'a::complex-inner*
assumes *a1*: ‹*bounded-clinear G*›
shows ‹*x* $\cdot_C$ *cadjoint G y* = *G x* $\cdot_C$ *y*›
⟨*proof*⟩

notation *cadjoint* (-† [*99*] *100*)

lemma *cadjoint-eqI*:
fixes *G*:: ‹*'b::complex-inner* ⇒ *'a::complex-inner*›
and *F*:: ‹*'a* ⇒ *'b*›
assumes ‹⋀*x y*. (*F x* $\cdot_C$ *y*) = (*x* $\cdot_C$ *G y*)›
shows ‹$G^\dagger$ = *F*›
⟨*proof*⟩

lemma *cadjoint-bounded-clinear*:
fixes *A* :: *'a::chilbert-space* ⇒ *'b::complex-inner*
assumes *a1*: *bounded-clinear A*
shows ‹*bounded-clinear* ($A^\dagger$)›
⟨*proof*⟩
include *notation-norm*
⟨*proof*⟩

proposition *double-cadjoint*:
fixes *U* :: ‹*'a::chilbert-space* ⇒ *'b::complex-inner*›
assumes *a1*: *bounded-clinear U*
shows $U^{\dagger\dagger}$ = *U*
⟨*proof*⟩

lemma *cadjoint-id*[*simp*]: ‹$id^\dagger$ = *id*›
⟨*proof*⟩

lemma *scaleC-cadjoint*:

fixes A::$'a$::*chilbert-space* $\Rightarrow$ $'b$::*complex-inner*
assumes *bounded-clinear* A
shows ‹$(\lambda t.\ a *_C\ A\ t)^\dagger = (\lambda s.\ cnj\ a *_C\ (A^\dagger)\ s)$›
⟨*proof*⟩

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›
⟨*proof*⟩

lemma *is-projection-on-cadjoint*:
fixes M :: ‹$'a$::*complex-inner set*›
assumes ‹*is-projection-on* π M› **and** ‹*closed-csubspace* M›
shows ‹$\pi^\dagger = \pi$›
⟨*proof*⟩

lemma *projection-cadjoint*:
fixes M :: ‹$'a$::*chilbert-space set*›
assumes ‹*closed-csubspace* M›
shows ‹$(projection\ M)^\dagger = projection\ M$›
⟨*proof*⟩

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*⟩

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*⟩

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* ($\lambda x.$ πa x + π x) (*closure* (*cspan* (*insert a S*)))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

9.11 Canonical basis (*onb-enum*)

⟨*ML*⟩

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*

⟨*ML*⟩

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*)›
⟨*proof*⟩

lemma *canonical-basis-is-onb*[*simp*]: ‹*is-onb* (*set canonical-basis* :: ′*a*::*onb-enum set*)›
⟨*proof*⟩

instance *onb-enum* ⊆ *chilbert-space*
⟨*proof*⟩

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*›⟨*proof*⟩
instance
⟨*proof*⟩
end

instance *conjugate-space* :: (*chilbert-space*) *chilbert-space*⟨*proof*⟩

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*⟩

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 additionally 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*
 ⟨*proof*⟩

lemma *one-cinner-one*[*simp*]: ‹(*1*::(′*a*::*one-dim*)) $\bullet_C$ *1* = *1*›
⟨*proof*⟩
 include *notation-norm*
 ⟨*proof*⟩

lemma *one-cinner-a-scaleC-one*[*simp*]: ‹((*1*::′*a*::*one-dim*) $\bullet_C$ *a*) $*_C$ *1* = *a*›
⟨*proof*⟩

lemma *one-dim-apply-is-times-def*:

$\psi * \varphi = ((1 \cdot_C \psi) * (1 \cdot_C \varphi)) *_C 1$ **for** ψ :: ‹'a::one-dim›
⟨proof⟩

instance *one-dim* ⊆ *complex-algebra-1*
⟨proof⟩

instance *one-dim* ⊆ *complex-normed-algebra*
⟨proof⟩

instance *one-dim* ⊆ *complex-normed-algebra-1*
⟨proof⟩

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* $(1 \cdot_C a)$

lemma *one-dim-iso-idem*[*simp*]: *one-dim-iso* (*one-dim-iso x*) = *one-dim-iso x*
⟨proof⟩

lemma *one-dim-iso-id*[*simp*]: *one-dim-iso* = *id*
⟨proof⟩

lemma *one-dim-iso-adjoint*[*simp*]: ‹*cadjoint one-dim-iso* = *one-dim-iso*›
⟨proof⟩

lemma *one-dim-iso-is-of-complex*[*simp*]: *one-dim-iso* = *of-complex*
⟨proof⟩

lemma *of-complex-one-dim-iso*[*simp*]: *of-complex* (*one-dim-iso* ψ) = *one-dim-iso* ψ
⟨proof⟩

lemma *one-dim-iso-of-complex*[*simp*]: *one-dim-iso* (*of-complex c*) = *of-complex c*
⟨proof⟩

lemma *one-dim-iso-add*[*simp*]:
‹*one-dim-iso* (*a* + *b*) = *one-dim-iso a* + *one-dim-iso b*›
⟨proof⟩

lemma *one-dim-iso-minus*[*simp*]:
‹*one-dim-iso* (*a* − *b*) = *one-dim-iso a* − *one-dim-iso b*›
⟨proof⟩

lemma *one-dim-iso-scaleC*[*simp*]: *one-dim-iso* $(c *_C \psi) = c *_C$ *one-dim-iso* ψ
⟨proof⟩

lemma *clinear-one-dim-iso*[*simp*]: *clinear one-dim-iso*
⟨*proof*⟩

lemma *bounded-clinear-one-dim-iso*[*simp*]: *bounded-clinear one-dim-iso*
⟨*proof*⟩

lemma *one-dim-iso-of-one*[*simp*]: *one-dim-iso 1 = 1*
⟨*proof*⟩

lemma *onorm-one-dim-iso*[*simp*]: *onorm one-dim-iso = 1*
⟨*proof*⟩

lemma *one-dim-iso-times*[*simp*]: *one-dim-iso* $(\psi * \varphi)$ = *one-dim-iso* ψ * *one-dim-iso* φ
⟨*proof*⟩

lemma *one-dim-iso-of-zero*[*simp*]: *one-dim-iso 0 = 0*
⟨*proof*⟩

lemma *one-dim-iso-of-zero'*: *one-dim-iso x = 0* $\Longrightarrow$ *x = 0*
⟨*proof*⟩

lemma *one-dim-scaleC-1*[*simp*]: *one-dim-iso x* $*_C$ *1 = x*
⟨*proof*⟩

lemma *one-dim-clinear-eqI*:
assumes (*x*::'*a*::*one-dim*) $\neq$ *0* **and** *clinear f* **and** *clinear g* **and** *f x = g x*
shows *f = g*
⟨*proof*⟩

lemma *one-dim-norm*: *norm x = cmod* (*one-dim-iso x*)
⟨*proof*⟩

lemma *norm-one-dim-iso*[*simp*]: ‹*norm* (*one-dim-iso x*) = *norm x*›
⟨*proof*⟩

lemma *one-dim-onorm*:
fixes *f* :: '*a*::*one-dim* ⇒ '*b*::*complex-normed-vector*
assumes *clinear f*
shows *onorm f = norm* (*f 1*)
⟨*proof*⟩

lemma *one-dim-onorm'*:
fixes *f* :: '*a*::*one-dim* ⇒ '*b*::*one-dim*
assumes *clinear f*
shows *onorm f = cmod* (*one-dim-iso* (*f 1*))
⟨*proof*⟩

instance *one-dim* ⊆ *zero-neq-one* ⟨*proof*⟩

lemma *one-dim-iso-inj*: *one-dim-iso x* = *one-dim-iso y* ⟹ *x* = *y*
⟨*proof*⟩

instance *one-dim* ⊆ *comm-ring*
⟨*proof*⟩

instance *one-dim* ⊆ *field*
⟨*proof*⟩

instance *one-dim* ⊆ *complex-normed-field*
⟨*proof*⟩

instance *one-dim* ⊆ *chilbert-space*⟨*proof*⟩

lemma *ccspan-one-dim*[*simp*]: ‹*ccspan* {*x*} = *top*› **if** ‹*x* ≠ *0*› **for** *x* :: ‹- :: *one-dim*›
⟨*proof*⟩

lemma *one-dim-ccsubspace-all-or-nothing*: ‹*A* = *bot* ∨ *A* = *top*› **for** *A* :: ‹-::*one-dim ccsubspace*›
⟨*proof*⟩

lemma *scaleC-1-right*[*simp*]: ‹*scaleC x* (*1*::′*a*::*one-dim*) = *of-complex x*›
⟨*proof*⟩

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*:

$(\forall\, u{\in}CBasis.\ cinner\ x\ u = 0) \longleftrightarrow (x = 0)$

syntax *-type-cdimension* :: *type* ⇒ *nat* ((*1CDIM*/(*1*′(-′))))
translations *CDIM*(′*a*) ⇀ *CONST card* (*CONST CBasis* :: ′*a set*)
⟨*ML*⟩

lemma (**in** *ceuclidean-space*) *norm-CBasis*[*simp*]: $u \in CBasis \Longrightarrow norm\ u = 1$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *cinner-same-CBasis*[*simp*]: $u \in CBasis \Longrightarrow cinner\ u\ u = 1$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *cinner-not-same-CBasis*: $u \in CBasis \Longrightarrow v \in CBasis \Longrightarrow u \neq v \Longrightarrow cinner\ u\ v = 0$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *sgn-CBasis*: $u \in CBasis \Longrightarrow sgn\ u = u$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *CBasis-zero* [*simp*]: $0 \notin CBasis$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *nonzero-CBasis*: $u \in CBasis \Longrightarrow u \neq 0$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *SOME-CBasis*: $(SOME\ i.\ i \in CBasis) \in CBasis$
⟨*proof*⟩

lemma *norm-some-CBasis* [*simp*]: $norm\ (SOME\ i.\ i \in CBasis) = 1$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *cinner-sum-left-CBasis*[*simp*]:
$b \in CBasis \Longrightarrow cinner\ (\sum i{\in}CBasis.\ f\ i *_C\ i)\ b = cnj\ (f\ b)$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *ceuclidean-eqI*:
assumes *b*: $\bigwedge b.\ b \in CBasis \Longrightarrow cinner\ x\ b = cinner\ y\ b$ **shows** $x = y$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *ceuclidean-eq-iff*:
$x = y \longleftrightarrow (\forall\, b{\in}CBasis.\ cinner\ x\ b = cinner\ y\ b)$
⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *ceuclidean-representation-sum*:
$(\sum i{\in}CBasis.\ f\ i *_C\ i) = b \longleftrightarrow (\forall\, i{\in}CBasis.\ f\ i = cnj\ (cinner\ b\ i))$

⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *ceuclidean-representation-sum'*:
  $b = (\sum i \in CBasis.\ f\ i *_C\ i) \longleftrightarrow (\forall i \in CBasis.\ f\ i = cinner\ i\ b)$
  ⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *ceuclidean-representation*: $(\sum b \in CBasis.\ cinner\ b\ x *_C\ b) = x$
  ⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *ceuclidean-cinner*: $cinner\ x\ y = (\sum b \in CBasis.\ cinner\ x\ b * cnj\ (cinner\ y\ b))$
  ⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *choice-CBasis-iff*:
  fixes $P :: 'a \Rightarrow complex \Rightarrow bool$
  shows $(\forall i \in CBasis.\ \exists x.\ P\ i\ x) \longleftrightarrow (\exists x.\ \forall i \in CBasis.\ P\ i\ (cinner\ x\ i))$
  ⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *bchoice-CBasis-iff*:
  fixes $P :: 'a \Rightarrow complex \Rightarrow bool$
  shows $(\forall i \in CBasis.\ \exists x \in A.\ P\ i\ x) \longleftrightarrow (\exists x.\ \forall i \in CBasis.\ cinner\ x\ i \in A \land P\ i\ (cinner\ x\ i))$
  ⟨*proof*⟩

lemma (**in** *ceuclidean-space*) *ceuclidean-representation-sum-fun*:
  $(\lambda x.\ \sum b \in CBasis.\ cinner\ b\ (f\ x) *_C\ b) = f$
  ⟨*proof*⟩

lemma *euclidean-isCont*:
  assumes $\bigwedge b.\ b \in CBasis \Longrightarrow isCont\ (\lambda x.\ (cinner\ b\ (f\ x)) *_C\ b)\ x$
  shows *isCont f x*
  ⟨*proof*⟩

lemma *CDIM-positive* [*simp*]: $0 < CDIM('a::ceuclidean\text{-}space)$
  ⟨*proof*⟩

lemma *CDIM-ge-Suc0* [*simp*]: $Suc\ 0 \leq card\ CBasis$
  ⟨*proof*⟩

lemma *sum-cinner-CBasis-scaleC* [*simp*]:
  fixes $f :: 'a::ceuclidean\text{-}space \Rightarrow 'b::complex\text{-}vector$
  assumes $b \in CBasis$ **shows** $(\sum i \in CBasis.\ (cinner\ i\ b) *_C\ f\ i) = f\ b$
  ⟨*proof*⟩

lemma *sum-cinner-CBasis-eq* [*simp*]:
  assumes $b \in CBasis$ **shows** $(\sum i \in CBasis.\ (cinner\ i\ b) * f\ i) = f\ b$
  ⟨*proof*⟩

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*⟩

lemma *norm-le-componentwise*:
$(\bigwedge b.\ b \in CBasis \Longrightarrow cmod(cinner\ x\ b) \le cmod(cinner\ y\ b)) \Longrightarrow norm\ x \le norm\ y$
⟨*proof*⟩

lemma *CBasis-le-norm*: $b \in CBasis \Longrightarrow cmod\ (cinner\ x\ b) \le norm\ x$
⟨*proof*⟩

lemma *norm-bound-CBasis-le*: $b \in CBasis \Longrightarrow norm\ x \le e \Longrightarrow cmod\ (inner\ x\ b) \le e$
⟨*proof*⟩

lemma *norm-bound-CBasis-lt*: $b \in CBasis \Longrightarrow norm\ x < e \Longrightarrow cmod\ (inner\ x\ b) < e$
⟨*proof*⟩

lemma *cnorm-le-l1*: $norm\ x \le (\sum b{\in}CBasis.\ cmod\ (cinner\ x\ b))$
⟨*proof*⟩

11.2 Class instances

11.2.1 Type *complex*

instantiation *complex* :: *ceuclidean-space*
begin

definition
[*simp*]: *CBasis* = {*1*::*complex*}

instance
⟨*proof*⟩

end

lemma *CDIM-complex*[*simp*]: *CDIM*(*complex*) = *1*
⟨*proof*⟩

11.2.2 Type $'a \times 'b$

lemma *cinner-Pair* [*simp*]: *cinner* (*a*, *b*) (*c*, *d*) = *cinner a c* + *cinner b d*
⟨*proof*⟩

lemma *cinner-Pair-0*: *cinner x* (*0*, *b*) = *cinner* (*snd x*) *b cinner x* (*a*, *0*) = *cinner* (*fst x*) *a*
⟨*proof*⟩

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*⟩

instance ⟨*proof*⟩

lemma *CDIM-prod*[*simp*]: *CDIM*(′*a* × ′*b*) = *CDIM*(′*a*) + *CDIM*(′*b*)
⟨*proof*⟩

end

11.3 Locale instances

lemma *finite-dimensional-vector-space-euclidean*:
finite-dimensional-vector-space ($*_C$) *CBasis*
⟨*proof*⟩

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*)

⟨*proof*⟩

interpretation *ceucl*: *finite-dimensional-vector-space-pair-1*
scaleC::*complex*⇒′*a*::*ceuclidean-space*⇒′*a CBasis*
scaleC::*complex*⇒′*b*::*complex-vector* ⇒ ′*b*
⟨*proof*⟩

interpretation *ceucl?*: *finite-dimensional-vector-space-prod scaleC scaleC CBasis CBasis*

rewrites *Basis-pair* = *CBasis*
and *module-prod.scale* $(*_C)$ $(*_C)$ = (*scaleC*::-⇒-⇒('*a* × '*b*))
⟨*proof*⟩

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* ≤ (∑ *i*∈*CBasis. norm* (*f i*))
⟨*proof*⟩

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* (λ*x. r x* $\cdot_C$ *f*) ≤ *onorm r* ∗ *norm f*
⟨*proof*⟩

lemma *onorm-cinner-right*:
assumes *bounded-linear r*
shows *onorm* (λ*x. f* $\cdot_C$ *r x*) ≤ *norm f* ∗ *onorm r*
⟨*proof*⟩

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

⟨*ML*⟩

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*
⟨*proof*⟩

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* (*λx. cblinfun-apply f* (*g x*))
⟨*proof*⟩

setup-lifting *type-definition-cblinfun*

lemma *cblinfun-eqI*: (⋀*i. cblinfun-apply x i* = *cblinfun-apply y i*) ⟹ *x* = *y*
⟨*proof*⟩

lemma *bounded-clinear-CBlinfun-apply*: *bounded-clinear f* ⟹ *cblinfun-apply* (*CBlinfun f*) = *f*
⟨*proof*⟩

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* ⇒ *real* **is** *onorm* ⟨*proof*⟩

lift-definition *minus-cblinfun* :: *'a* $\Rightarrow_{CL}$ *'b* ⇒ *'a* $\Rightarrow_{CL}$ *'b* ⇒ *'a* $\Rightarrow_{CL}$ *'b*
is *λf g x. f x* − *g x*
⟨*proof*⟩

definition *dist-cblinfun* :: *'a* $\Rightarrow_{CL}$ *'b* ⇒ *'a* $\Rightarrow_{CL}$ *'b* ⇒ *real*
where *dist-cblinfun a b* = *norm* (*a* − *b*)

definition [*code del*]:
($uniformity$ :: (($'a \Rightarrow_{CL} 'b$) × ($'a \Rightarrow_{CL} 'b$)) *filter*) = (*INF* $e \in \{0 <..\}$. *principal* $\{(x, y).\ dist\ x\ y < e\}$)

definition *open-cblinfun* :: ($'a \Rightarrow_{CL} 'b$) *set* ⇒ *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** $\lambda f\ x.\ -\ f\ x$
⟨*proof*⟩

lift-definition *zero-cblinfun* :: $'a \Rightarrow_{CL} 'b$ **is** $\lambda x.\ 0$
⟨*proof*⟩

lift-definition *plus-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$
is $\lambda f\ g\ x.\ f\ x + g\ x$
⟨*proof*⟩

lift-definition *scaleC-cblinfun*::*complex* ⇒ $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$ **is** $\lambda r\ f\ x.\ r *_C f\ x$
⟨*proof*⟩
lift-definition *scaleR-cblinfun*::*real* ⇒ $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$ **is** $\lambda r\ f\ x.\ r *_R f\ x$
⟨*proof*⟩

definition *sgn-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$
where *sgn-cblinfun* x = *scaleC* (*inverse* (*norm* x)) x

instance
⟨*proof*⟩

end

declare *uniformity-Abort*[**where** $'a$=($'a$ :: *complex-normed-vector*) $\Rightarrow_{CL}$ ($'b$ :: *complex-normed-vector*), *code*]

lemma *norm-cblinfun-eqI*:
assumes $n \leq$ *norm* (*cblinfun-apply* $f\ x$) / *norm* x
assumes $\bigwedge x.$ *norm* (*cblinfun-apply* $f\ x$) $\leq n *$ *norm* x
assumes $0 \leq n$
shows *norm* $f = n$
⟨*proof*⟩

lemma *norm-cblinfun*: *norm* (*cblinfun-apply* $f\ x$) $\leq$ *norm* $f\ *$ *norm* x
⟨*proof*⟩

lemma *norm-cblinfun-bound*: $0 \leq b \Longrightarrow (\bigwedge x.$ *norm* (*cblinfun-apply* $f\ x$) $\leq b *$ *norm* x) $\Longrightarrow$ *norm* $f \leq b$
⟨*proof*⟩

lemma *bounded-cbilinear-cblinfun-apply*[*bounded-cbilinear*]: *bounded-cbilinear cblinfun-apply*
⟨*proof*⟩

interpretation *cblinfun*: *bounded-cbilinear cblinfun-apply*
⟨*proof*⟩

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*⟩

12.5 On Euclidean Space

lemma *norm-cblinfun-ceuclidean-le*:
fixes $a::'a::ceuclidean\text{-}space \Rightarrow_{CL} 'b::complex\text{-}normed\text{-}vector$
shows $norm\ a \le sum\ (\lambda x.\ norm\ (a\ x))\ CBasis$
⟨*proof*⟩

lemma *ctendsto-componentwise1*:
fixes $a::'a::ceuclidean\text{-}space \Rightarrow_{CL} 'b::complex\text{-}normed\text{-}vector$
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*⟩

lift-definition
cblinfun-of-matrix::(′*b*::*ceuclidean-space* ⇒ ′*a*::*ceuclidean-space* ⇒ *complex*) ⇒ ′*a* $\Rightarrow_{CL}$ ′*b*
is $\lambda a\ x.\ \sum i{\in}CBasis.\ \sum j{\in}CBasis.\ ((j \bullet_C x) * a\ i\ j) *_C i$
⟨*proof*⟩

lemma *cblinfun-of-matrix-works*:
fixes *f*::′*a*::*ceuclidean-space* $\Rightarrow_{CL}$ ′*b*::*ceuclidean-space*
shows *cblinfun-of-matrix* $(\lambda i\ j.\ i \bullet_C (f\ j)) = f$
⟨*proof*⟩

lemma *cblinfun-of-matrix-apply*:
cblinfun-of-matrix $a\ x = (\sum i{\in}CBasis.\ \sum j{\in}CBasis.\ ((j \bullet_C x) * a\ i\ j) *_C i)$
⟨*proof*⟩

lemma *cblinfun-of-matrix-minus*: *cblinfun-of-matrix* x − *cblinfun-of-matrix* y = *cblinfun-of-matrix* $(x - y)$
⟨*proof*⟩

lemma *norm-cblinfun-of-matrix*:
norm (*cblinfun-of-matrix* a) $\le (\sum i{\in}CBasis.\ \sum j{\in}CBasis.\ cmod\ (a\ i\ j))$
⟨*proof*⟩

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-of-matrix* $(b\ n)) \longrightarrow$ *cblinfun-of-matrix* $a)\ F$
⟨*proof*⟩

lemma *ctendsto-componentwise*:
fixes *a*::′*a*::*ceuclidean-space* $\Rightarrow_{CL}$ ′*b*::*ceuclidean-space*
and *b*::′*c* ⇒ ′*a* $\Rightarrow_{CL}$ ′*b*
shows $(\bigwedge i\ j.\ i \in CBasis \Longrightarrow j \in CBasis \Longrightarrow ((\lambda n.\ b\ n\ j \bullet_C i) \longrightarrow a\ j \bullet_C i)\ F) \Longrightarrow (b \longrightarrow a)\ F$
⟨*proof*⟩

lemma
continuous-cblinfun-componentwiseI:
fixes *f*:: ′*b*::*t2-space* ⇒ ′*a*::*ceuclidean-space* $\Rightarrow_{CL}$ ′*c*::*ceuclidean-space*
assumes $\bigwedge i\ j.\ i \in CBasis \Longrightarrow j \in CBasis \Longrightarrow$ *continuous* $F\ (\lambda x.\ (f\ x)\ j \bullet_C i)$
shows *continuous* $F\ f$
⟨*proof*⟩

lemma
continuous-cblinfun-componentwiseI1:

fixes f:: $'b$::*t2-space* ⇒ $'a$::*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*
⟨*proof*⟩

lemma
continuous-on-cblinfun-componentwise:
fixes f:: $'d$::*t2-space* ⇒ $'e$::*ceuclidean-space* $\Rightarrow_{CL}$ $'f$::*complex-normed-vector*
assumes $\bigwedge i.\ i \in CBasis \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ f\ x\ i)$
shows *continuous-on s f*
⟨*proof*⟩

lemma *bounded-antilinear-cblinfun-matrix*: *bounded-antilinear* $(\lambda x.\ (x::\text{-}\Rightarrow_{CL}\ \text{-})\ j \bullet_C\ i)$
⟨*proof*⟩

lemma *continuous-cblinfun-matrix*:
fixes f:: $'b$::*t2-space* ⇒ $'a$::*complex-normed-vector* $\Rightarrow_{CL}$ $'c$::*complex-inner*
assumes *continuous F f*
shows *continuous F* $(\lambda x.\ (f\ x)\ j \bullet_C\ i)$
⟨*proof*⟩

lemma *continuous-on-cblinfun-matrix*:
fixes f::$'a$::*t2-space* ⇒ $'b$::*complex-normed-vector* $\Rightarrow_{CL}$ $'c$::*complex-inner*
assumes *continuous-on S f*
shows *continuous-on S* $(\lambda x.\ (f\ x)\ j \bullet_C\ i)$
⟨*proof*⟩

lemma *continuous-on-cblinfun-of-matrix*[*continuous-intros*]:
assumes $\bigwedge i\ j.\ i \in CBasis \Longrightarrow j \in CBasis \Longrightarrow continuous\text{-}on\ S\ (\lambda s.\ g\ s\ i\ j)$
shows *continuous-on S* $(\lambda s.\ cblinfun\text{-}of\text{-}matrix\ (g\ s))$
⟨*proof*⟩

lemma *cblinfun-euclidean-eqI*: $(\bigwedge i.\ i \in CBasis \Longrightarrow cblinfun\text{-}apply\ x\ i = cblinfun\text{-}apply\ y\ i) \Longrightarrow x = y$
⟨*proof*⟩

lemma *CBlinfun-eq-matrix*: *bounded-clinear f* $\Longrightarrow$ *CBlinfun f* = *cblinfun-of-matrix* $(\lambda i\ j.\ i \bullet_C\ f\ j)$
⟨*proof*⟩

12.6 concrete bounded linear functions

lemma *transfer-bounded-cbilinear-bounded-clinearI*:
assumes $g = (\lambda i\ x.\ (cblinfun\text{-}apply\ (f\ i)\ x))$
shows *bounded-cbilinear* g = *bounded-clinear* f
⟨*proof*⟩

lemma *transfer-bounded-cbilinear-bounded-clinear*[*transfer-rule*]:
(*rel-fun* (*rel-fun* (=) (*pcr-cblinfun* (=) (=))) (=)) *bounded-cbilinear bounded-clinear*
⟨*proof*⟩

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*⟩

lemma *transfer-bounded-sesquilinear-bounded-antilinear*[*transfer-rule*]:
(*rel-fun* (*rel-fun* (=) (*pcr-cblinfun* (=) (=))) (=)) *bounded-sesquilinear bounded-antilinear*
⟨*proof*⟩

context *bounded-cbilinear*
begin

lift-definition *prod-left*::$'b \Rightarrow 'a \Rightarrow_{CL} 'c$ **is** $(\lambda b\ a.\ prod\ a\ b)$
⟨*proof*⟩
declare *prod-left.rep-eq*[*simp*]

lemma *bounded-clinear-prod-left*[*bounded-clinear*]: *bounded-clinear prod-left*
⟨*proof*⟩

lift-definition *prod-right*::$'a \Rightarrow 'b \Rightarrow_{CL} 'c$ **is** $(\lambda a\ b.\ prod\ a\ b)$
⟨*proof*⟩
declare *prod-right.rep-eq*[*simp*]

lemma *bounded-clinear-prod-right*[*bounded-clinear*]: *bounded-clinear prod-right*
⟨*proof*⟩

end

lift-definition *id-cblinfun*::$'a$::*complex-normed-vector* $\Rightarrow_{CL} 'a$ **is** $\lambda x.\ x$
⟨*proof*⟩

lemmas *cblinfun-id-cblinfun-apply*[*simp*] = *id-cblinfun.rep-eq*

lemma *norm-cblinfun-id*[*simp*]:
norm (*id-cblinfun*::$'a$::{*complex-normed-vector*, *not-singleton*} $\Rightarrow_{CL} 'a$) = *1*
⟨*proof*⟩

lemma *norm-cblinfun-id-le*:
norm (*id-cblinfun*::'*a*::*complex-normed-vector* $\Rightarrow_{CL}$ '*a*) $\leq$ *1*
⟨*proof*⟩

lift-definition *cblinfun-compose*::
'*a*::*complex-normed-vector* $\Rightarrow_{CL}$ '*b*::*complex-normed-vector* $\Rightarrow$
'*c*::*complex-normed-vector* $\Rightarrow_{CL}$ '*a* $\Rightarrow$
'*c* $\Rightarrow_{CL}$ '*b* (**infixl** o_{CL} *67*) **is** (o)

parametric *comp-transfer*
⟨*proof*⟩

lemma *cblinfun-apply-cblinfun-compose*[*simp*]: (a o_{CL} b) c = a (b c)
⟨*proof*⟩

lemma *norm-cblinfun-compose*:
norm (f o_{CL} g) $\leq$ *norm* f $*$ *norm* g
⟨*proof*⟩

lemma *bounded-cbilinear-cblinfun-compose*[*bounded-cbilinear*]: *bounded-cbilinear* (o_{CL})
⟨*proof*⟩

lemma *cblinfun-compose-zero*[*simp*]:
blinfun-compose *0* = (λ-. *0*)
blinfun-compose x *0* = *0*
⟨*proof*⟩

lemma *cblinfun-bij2*:
fixes f::'*a* $\Rightarrow_{CL}$ '*a*::*ceuclidean-space*
assumes f o_{CL} g = *id-cblinfun*
shows *bij* (*cblinfun-apply* g)
⟨*proof*⟩

lemma *cblinfun-bij1*:
fixes f::'*a* $\Rightarrow_{CL}$ '*a*::*ceuclidean-space*
assumes f o_{CL} g = *id-cblinfun*
shows *bij* (*cblinfun-apply* f)
⟨*proof*⟩

lift-definition *cblinfun-cinner-right*::'*a*::*complex-inner* $\Rightarrow$ '*a* $\Rightarrow_{CL}$ *complex* **is** ($\cdot_C$)

⟨*proof*⟩
declare *cblinfun-cinner-right.rep-eq*[*simp*]

lemma *bounded-antilinear-cblinfun-cinner-right*[*bounded-antilinear*]: *bounded-antilinear cblinfun-cinner-right*
⟨*proof*⟩

lift-definition *cblinfun-scaleC-right*::*complex* ⇒ *'a* $\Rightarrow_{CL}$ *'a*::*complex-normed-vector* **is** $(*_C)$
⟨*proof*⟩
declare *cblinfun-scaleC-right.rep-eq*[*simp*]

lemma *bounded-clinear-cblinfun-scaleC-right*[*bounded-clinear*]: *bounded-clinear cblinfun-scaleC-right*
⟨*proof*⟩

lift-definition *cblinfun-scaleC-left*::*'a*::*complex-normed-vector* ⇒ *complex* $\Rightarrow_{CL}$ *'a* **is** $\lambda x\ y.\ y *_C x$
⟨*proof*⟩
lemmas [*simp*] = *cblinfun-scaleC-left.rep-eq*

lemma *bounded-clinear-cblinfun-scaleC-left*[*bounded-clinear*]: *bounded-clinear cblinfun-scaleC-left*
⟨*proof*⟩

lift-definition *cblinfun-mult-right*::*'a* ⇒ *'a* $\Rightarrow_{CL}$ *'a*::*complex-normed-algebra* **is** (∗)
⟨*proof*⟩
declare *cblinfun-mult-right.rep-eq*[*simp*]

lemma *bounded-clinear-cblinfun-mult-right*[*bounded-clinear*]: *bounded-clinear cblinfun-mult-right*
⟨*proof*⟩

lift-definition *cblinfun-mult-left*::*'a*::*complex-normed-algebra* ⇒ *'a* $\Rightarrow_{CL}$ *'a* **is** $\lambda x\ y.\ y * x$
⟨*proof*⟩
lemmas [*simp*] = *cblinfun-mult-left.rep-eq*

lemma *bounded-clinear-cblinfun-mult-left*[*bounded-clinear*]: *bounded-clinear cblinfun-mult-left*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

lemma *cstrong-operator-topology-continuous-evaluation*:
  continuous-map cstrong-operator-topology euclidean (λf. *cblinfun-apply f x*)
⟨*proof*⟩

lemma *continuous-on-cstrong-operator-topo-iff-coordinatewise*:
  continuous-map T cstrong-operator-topology f

$\longleftrightarrow$ ($\forall$ *x*. *continuous-map T euclidean* (λ*y*. *cblinfun-apply* (*f y*) *x*))
⟨*proof*⟩

lemma *cstrong-operator-topology-weaker-than-euclidean*:
continuous-map euclidean cstrong-operator-topology (λ*f*. *f*)
⟨*proof*⟩
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
Banach-Steinhaus.Banach-Steinhaus
Complex-Inner-Product
One-Dimensional-Spaces
Complex-Bounded-Linear-Function0
HOL−Library.Function-Algebras
begin

unbundle *lattice-syntax*

13.1 Misc basic facts and declarations

notation *cblinfun-apply* (**infixr** $*_V$ *70*)

lemma *id-cblinfun-apply*[*simp*]: *id-cblinfun* $*_V$ $\psi = \psi$
⟨*proof*⟩

lemma *apply-id-cblinfun*[*simp*]: ‹($*_V$) *id-cblinfun* = *id*›
⟨*proof*⟩

lemma *isCont-cblinfun-apply*[*simp*]: *isCont* (($*_V$) *A*) ψ
⟨*proof*⟩

declare *cblinfun.scaleC-left*[*simp*]

lemma *cblinfun-apply-clinear*[*simp*]: ‹*clinear* (*cblinfun-apply A*)›
⟨*proof*⟩

lemma *cblinfun-cinner-eqI*:
fixes *A B* :: ‹*'a*::*chilbert-space* $\Rightarrow_{CL}$ *'a*›
assumes ‹$\bigwedge\psi$. *norm* ψ = *1* $\Longrightarrow$ *cinner* ψ (*A* $*_V$ ψ) = *cinner* ψ (*B* $*_V$ ψ)›
shows ‹*A* = *B*›
⟨*proof*⟩

lemma *id-cblinfun-not-0*[*simp*]: ‹(*id-cblinfun* :: *'a*::{*complex-normed-vector*, *not-singleton*}

$\Rightarrow_{CL}$ -) ≠ 0›
⟨proof⟩

lemma *cblinfun-norm-geqI*:
assumes ‹norm (f $*_V$ x) / norm x ≥ K›
shows ‹norm f ≥ K›
⟨proof⟩

declare *scaleC-conv-of-complex*[*simp*]

lemma *cblinfun-eq-0-on-span*:
fixes S::‹'a::complex-normed-vector set›
assumes x ∈ cspan S
and ⋀s. s∈S ⟹ F $*_V$ s = 0
shows ‹F $*_V$ x = 0›
⟨proof⟩

lemma *cblinfun-eq-on-span*:
fixes S::‹'a::complex-normed-vector set›
assumes x ∈ cspan S
and ⋀s. s∈S ⟹ F $*_V$ s = G $*_V$ s
shows ‹F $*_V$ x = G $*_V$ x›
⟨proof⟩

lemma *cblinfun-eq-0-on-UNIV-span*:
fixes basis::‹'a::complex-normed-vector set›
assumes cspan basis = UNIV
and ⋀s. s∈basis ⟹ F $*_V$ s = 0
shows ‹F = 0›
⟨proof⟩

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 ⋀s. s∈basis ⟹ F $*_V$ s = G $*_V$ s
shows ‹F = G›
⟨proof⟩

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 ⋀u. u ∈ basis ⟹ f $*_V$ u = g $*_V$ u
shows f = g
⟨proof⟩

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 $f = 0$
$\langle proof \rangle$

lemma *cinner-canonical-basis-eq-0*:
defines $basisA == set\ (canonical\text{-}basis{::}'a{::}onb\text{-}enum\ list)$
and $basisB == set\ (canonical\text{-}basis{::}'b{::}onb\text{-}enum\ list)$
assumes $\bigwedge u\ v.\ u{\in}basisA \Longrightarrow v{\in}basisB \Longrightarrow is\text{-}orthogonal\ v\ (F *_V\ u)$
shows $F = 0$
$\langle proof \rangle$

lemma *cinner-canonical-basis-eq*:
defines $basisA == set\ (canonical\text{-}basis{::}'a{::}onb\text{-}enum\ list)$
and $basisB == set\ (canonical\text{-}basis{::}'b{::}onb\text{-}enum\ list)$
assumes $\bigwedge u\ v.\ u{\in}basisA \Longrightarrow v{\in}basisB \Longrightarrow v \bullet_C\ (F *_V\ u) = v \bullet_C\ (G *_V\ u)$
shows $F = G$
$\langle proof \rangle$

lemma *cinner-canonical-basis-eq'*:
defines $basisA == set\ (canonical\text{-}basis{::}'a{::}onb\text{-}enum\ list)$
and $basisB == set\ (canonical\text{-}basis{::}'b{::}onb\text{-}enum\ list)$
assumes $\bigwedge u\ v.\ u{\in}basisA \Longrightarrow v{\in}basisB \Longrightarrow (F *_V\ u) \bullet_C\ v = (G *_V\ u) \bullet_C\ v$
shows $F = G$
$\langle proof \rangle$

lemma *not-not-singleton-cblinfun-zero*:
‹$x = 0$› **if** ‹$\neg$ *class.not-singleton TYPE('a)*› **for** x :: ‹*'a::complex-normed-vector* $\Rightarrow_{CL}$ *'b::complex-normed-vector*›
$\langle proof \rangle$

lemma *cblinfun-norm-approx-witness*:
fixes A :: ‹*'a::{not-singleton,complex-normed-vector}* $\Rightarrow_{CL}$ *'b::complex-normed-vector*›
assumes ‹$\varepsilon > 0$›
shows ‹$\exists\psi.\ norm\ (A *_V\ \psi) \geq norm\ A - \varepsilon \wedge norm\ \psi = 1$›
$\langle proof \rangle$

lemma *cblinfun-norm-approx-witness-mult*:
fixes A :: ‹*'a::{not-singleton,complex-normed-vector}* $\Rightarrow_{CL}$ *'b::complex-normed-vector*›
assumes ‹$\varepsilon < 1$›
shows ‹$\exists\psi.\ norm\ (A *_V\ \psi) \geq norm\ A * \varepsilon \wedge norm\ \psi = 1$›
$\langle proof \rangle$

lemma *cblinfun-norm-approx-witness'*:
fixes A :: ‹*'a::complex-normed-vector* $\Rightarrow_{CL}$ *'b::complex-normed-vector*›
assumes ‹$\varepsilon > 0$›
shows ‹$\exists\psi.\ norm\ (A *_V\ \psi)\ /\ norm\ \psi \geq norm\ A - \varepsilon$›
$\langle proof \rangle$

lemma *cblinfun-to-CARD-1-0*[*simp*]: ‹(*A* :: - $\Rightarrow_{CL}$ -::*CARD-1*) = *0*›
⟨*proof*⟩

lemma *cblinfun-from-CARD-1-0*[*simp*]: ‹(*A* :: -::*CARD-1* $\Rightarrow_{CL}$ -) = *0*›
⟨*proof*⟩

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*: ‹⋀*a b*. *a*∈*basisA* ⟹ *b*∈*basisB* ⟹ ∃ *F*∈*basis*. ∀ *a'*∈*basisA*. *F* $*_V$ *a'* = (*if a'*=*a then b else 0*)›
shows ‹*cspan basis* = *UNIV*›
⟨*proof*⟩

instance *cblinfun* :: (‹{*cfinite-dim*,*complex-normed-vector*}›, ‹{*cfinite-dim*,*complex-normed-vector*}›) *cfinite-dim*
⟨*proof*⟩

lemma *norm-cblinfun-bound-dense*:
assumes ‹*0* ≤ *b*›
assumes *S*: ‹*closure S* = *UNIV*›
assumes *bound*: ‹⋀*x*. *x*∈*S* ⟹ *norm* (*cblinfun-apply f x*) ≤ *b* ∗ *norm x*›
shows ‹*norm f* ≤ *b*›
⟨*proof*⟩

lemma *infsum-cblinfun-apply*:
assumes ‹*g summable-on S*›
shows ‹*infsum* (λ*x*. *A* $*_V$ *g x*) *S* = *A* $*_V$ (*infsum g S*)›
⟨*proof*⟩

lemma *has-sum-cblinfun-apply*:
assumes ‹(*g has-sum x*) *S*›
shows ‹((λ*x*. *A* $*_V$ *g x*) *has-sum* (*A* $*_V$ *x*)) *S*›
⟨*proof*⟩

lemma *abs-summable-on-cblinfun-apply*:
assumes ‹*g abs-summable-on S*›
shows ‹(λ*x*. *A* $*_V$ *g x*) *abs-summable-on S*›
⟨*proof*⟩

lemma *summable-on-cblinfun-apply*:
assumes ‹*g summable-on S*›
shows ‹(λ*x*. *A* $*_V$ *g x*) *summable-on S*›
⟨*proof*⟩

lemma *summable-on-cblinfun-apply-left*:
assumes ‹*A summable-on S*›
shows ‹$(\lambda x.\ A\ x *_V\ g)$ *summable-on S*›
⟨*proof*⟩

lemma *abs-summable-on-cblinfun-apply-left*:
assumes ‹*A abs-summable-on S*›
shows ‹$(\lambda x.\ A\ x *_V\ g)$ *abs-summable-on S*›
⟨*proof*⟩
lemma *infsum-cblinfun-apply-left*:
assumes ‹*A summable-on S*›
shows ‹*infsum* $(\lambda x.\ A\ x *_V\ g)\ S =$ (*infsum A S*) $*_V\ g$›
⟨*proof*⟩
lemma *has-sum-cblinfun-apply-left*:
assumes ‹(*A has-sum x*) *S*›
shows ‹($(\lambda x.\ A\ x *_V\ g)$ *has-sum* $(x *_V\ g)$) *S*›
⟨*proof*⟩

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 ‹(($\lambda i.$ *cinner a* (*f i*)) *has-sum cinner a x*) *I*›
⟨*proof*⟩

lemma *summable-on-cinner-left*:
assumes ‹*f summable-on I*›
shows ‹($\lambda i.$ *cinner a* (*f i*)) *summable-on I*›
⟨*proof*⟩

lemma *infsum-cinner-left*:
assumes ‹φ *summable-on I*›
shows ‹*cinner* ψ $(\sum_\infty i{\in}I.\ \varphi\ i) = (\sum_\infty i{\in}I.$ *cinner* $\psi\ (\varphi\ i))$›
⟨*proof*⟩

lemma *has-sum-cinner-right*:
assumes ‹(*f has-sum x*) *I*›
shows ‹(($\lambda i.\ f\ i \cdot_C\ a$) *has-sum* $(x \cdot_C\ a)$) *I*›
⟨*proof*⟩

lemma *summable-on-cinner-right*:
assumes ‹*f summable-on I*›
shows ‹($\lambda i.\ f\ i \cdot_C\ a$) *summable-on I*›
⟨*proof*⟩

lemma *infsum-cinner-right*:
assumes ‹φ *summable-on I*›
shows ‹$(\sum_\infty i{\in}I.\ \varphi\ i) \cdot_C\ \psi = (\sum_\infty i{\in}I.\ \varphi\ i \cdot_C\ \psi)$›

⟨*proof*⟩

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* $(a\ x)\ (b\ y) \geq 0$›
shows *absum*: ‹$(\lambda(x,\ y).$ *cinner* $(a\ x)\ (b\ y))$ *summable-on UNIV*›
⟨*proof*⟩

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*: ‹$(\lambda(x,\ y).$ *cinner* $(a\ x)\ (b\ y))$ *summable-on UNIV*›

shows *Cauchy-cinner-product-infsum*: ‹$(\sum_\infty k.\ \sum i{\leq}k.$ *cinner* $(a\ i)\ (b\ (k - i)))$ $=$ *cinner* $(\sum_\infty k.\ a\ k)\ (\sum_\infty k.\ b\ k)$›
and *Cauchy-cinner-product-infsum-exists*: ‹$(\lambda k.\ \sum i{\leq}k.$ *cinner* $(a\ i)\ (b\ (k - i)))$ *summable-on UNIV*›
⟨*proof*⟩

lemma *CBlinfun-plus*:
assumes [*simp*]: ‹*bounded-clinear f*› ‹*bounded-clinear g*›
shows ‹*CBlinfun* $(f + g) =$ *CBlinfun f* $+$ *CBlinfun g*›
⟨*proof*⟩

lemma *CBlinfun-scaleC*:
assumes ‹*bounded-clinear f*›
shows ‹*CBlinfun* $(\lambda y.\ c *_C f\ y) = c *_C$ *CBlinfun f*›
⟨*proof*⟩

lemma *cinner-sup-norm-cblinfun*:
fixes *A* :: ‹*'a::{complex-normed-vector,not-singleton}* $\Rightarrow_{CL}$ *'b::complex-inner*›
shows ‹*norm A* $=$ (*SUP* (ψ,φ). *cmod* (*cinner* ψ $(A *_V \varphi)$) / (*norm* ψ $*$ *norm* φ))›
⟨*proof*⟩

lemma *norm-cblinfun-Sup*: ‹*norm* A = (*SUP* ψ. *norm* ($A *_V \psi$) / *norm* ψ)›
⟨*proof*⟩

lemma *cblinfun-eq-on*:
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** ‹$t \in$ *closure* (*cspan* G)›
shows $A *_V t = B *_V t$
⟨*proof*⟩

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 $A = B$
⟨*proof*⟩

declare *cnj-bounded-antilinear*[*bounded-antilinear*]

lemma *Cblinfun-comp-bounded-cbilinear*: ‹*bounded-clinear* (*CBlinfun o p*)› **if** ‹*bounded-cbilinear p*›
⟨*proof*⟩

lemma *Cblinfun-comp-bounded-sesquilinear*: ‹*bounded-antilinear* (*CBlinfun o p*)›
if ‹*bounded-sesquilinear p*›
⟨*proof*⟩

13.2 Relationship to real bounded operators (- $\Rightarrow_L$ -)

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*⟩

instance
⟨*proof*⟩
end

lemma *clinear-blinfun-compose-left*: ‹*clinear* (λx. *blinfun-compose x y*)›
⟨*proof*⟩

instance *blinfun* :: (*real-normed-vector*, *cbanach*) *cbanach*⟨*proof*⟩

lemma *blinfun-compose-assoc*: $(A\ o_L\ B)\ o_L\ C = A\ o_L\ (B\ \ o_L\ C)$
⟨*proof*⟩

lift-definition *blinfun-of-cblinfun*::‹′*a*::*complex-normed-vector* $\Rightarrow_{CL}$ ′*b*::*complex-normed-vector* ⇒ ′*a* $\Rightarrow_L$ ′*b*› **is** *id*
⟨*proof*⟩

lift-definition *blinfun-cblinfun-eq* ::
‹′*a* $\Rightarrow_L$ ′*b* ⇒ ′*a*::*complex-normed-vector* $\Rightarrow_{CL}$ ′*b*::*complex-normed-vector* ⇒ *bool*›
is (=) ⟨*proof*⟩

lemma *blinfun-cblinfun-eq-bi-unique*[*transfer-rule*]: ‹*bi-unique blinfun-cblinfun-eq*›
⟨*proof*⟩

lemma *blinfun-cblinfun-eq-right-total*[*transfer-rule*]: ‹*right-total blinfun-cblinfun-eq*›
⟨*proof*⟩

named-theorems *cblinfun-blinfun-transfer*

lemma *cblinfun-blinfun-transfer-0*[*cblinfun-blinfun-transfer*]:
blinfun-cblinfun-eq (*0*::(-,-) *blinfun*) (*0*::(-,-) *cblinfun*)
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-plus*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (+) (+)
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-minus*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (−) (−)
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-uminus*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (*uminus*) (*uminus*)
⟨*proof*⟩

definition *real-complex-eq r c* ⟷ *complex-of-real r* = *c*

lemma *bi-unique-real-complex-eq*[*transfer-rule*]: ‹*bi-unique real-complex-eq*›
⟨*proof*⟩

lemma *left-total-real-complex-eq*[*transfer-rule*]: ‹*left-total real-complex-eq*›
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-scaleC*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*real-complex-eq* ===> *blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (*scaleR*) (*scaleC*)

⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-CBlinfun*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*eq-onp bounded-clinear* ===> *blinfun-cblinfun-eq*) *Blinfun CBlinfun*
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-norm*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> (=)) *norm norm*
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-dist*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq* ===> (=)) *dist dist*
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-sgn*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) *sgn sgn*
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-Cauchy*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (((=) ===> *blinfun-cblinfun-eq*) ===> (=)) *Cauchy Cauchy*
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-tendsto*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (((=) ===> *blinfun-cblinfun-eq*) ===> *blinfun-cblinfun-eq* ===> (=) ===> (=)) *tendsto tendsto*
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-compose*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq* ===> *blinfun-cblinfun-eq*) (o_L) (o_{CL})
⟨*proof*⟩

lemma *cblinfun-blinfun-transfer-apply*[*cblinfun-blinfun-transfer*]:
includes *lifting-syntax*
shows (*blinfun-cblinfun-eq* ===> (=) ===> (=)) *blinfun-apply cblinfun-apply*
⟨*proof*⟩

lemma *blinfun-of-cblinfun-inj*:
‹*blinfun-of-cblinfun f* = *blinfun-of-cblinfun g* ⟹ *f* = *g*›
⟨*proof*⟩

lemma *blinfun-of-cblinfun-inv*:

assumes $\bigwedge c. \bigwedge x.\ f *_v (c *_C x) = c *_C (f *_v x)$
shows $\exists\, g.$ *blinfun-of-cblinfun* $g = f$
⟨*proof*⟩

lemma *blinfun-of-cblinfun-zero*:
‹*blinfun-of-cblinfun 0 = 0*›
⟨*proof*⟩

lemma *blinfun-of-cblinfun-uminus*:
‹*blinfun-of-cblinfun* $(-\ f) = -$ (*blinfun-of-cblinfun* f)›
⟨*proof*⟩

lemma *blinfun-of-cblinfun-minus*:
‹*blinfun-of-cblinfun* $(f - g) =$ *blinfun-of-cblinfun* $f -$ *blinfun-of-cblinfun* g›
⟨*proof*⟩

lemma *blinfun-of-cblinfun-scaleC*:
‹*blinfun-of-cblinfun* $(c *_C f) = c *_C$ (*blinfun-of-cblinfun* f)›
⟨*proof*⟩

lemma *blinfun-of-cblinfun-scaleR*:
‹*blinfun-of-cblinfun* $(c *_R f) = c *_R$ (*blinfun-of-cblinfun* f)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

13.3 Composition

lemma *cblinfun-compose-assoc*:
shows $(A\ o_{CL}\ B)\ o_{CL}\ C = A\ o_{CL}\ (B\ o_{CL}\ C)$
⟨*proof*⟩

lemma *cblinfun-compose-zero-right*[*simp*]: $U\ o_{CL}\ 0 = 0$
⟨*proof*⟩

lemma *cblinfun-compose-zero-left*[*simp*]: $0\ o_{CL}\ U = 0$
⟨*proof*⟩

lemma *cblinfun-compose-scaleC-left*[*simp*]:

fixes $A::'b::complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'c::complex\text{-}normed\text{-}vector$
and $B::'a::complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'b$
shows ‹$(a *_C A) \ o_{CL} \ B = a *_C (A \ o_{CL} \ B)$›
⟨*proof*⟩

lemma *cblinfun-compose-scaleR-left*[*simp*]:
fixes $A::'b::complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'c::complex\text{-}normed\text{-}vector$
and $B::'a::complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'b$
shows ‹$(a *_R A) \ o_{CL} \ B = a *_R (A \ o_{CL} \ B)$›
⟨*proof*⟩

lemma *cblinfun-compose-scaleC-right*[*simp*]:
fixes $A::'b::complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'c::complex\text{-}normed\text{-}vector$
and $B::'a::complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'b$
shows ‹$A \ o_{CL} \ (a *_C B) = a *_C (A \ o_{CL} \ B)$›
⟨*proof*⟩

lemma *cblinfun-compose-scaleR-right*[*simp*]:
fixes $A::'b::complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'c::complex\text{-}normed\text{-}vector$
and $B::'a::complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'b$
shows ‹$A \ o_{CL} \ (a *_R B) = a *_R (A \ o_{CL} \ B)$›
⟨*proof*⟩

lemma *cblinfun-compose-id-right*[*simp*]:
shows $U \ o_{CL} \ id\text{-}cblinfun = U$
⟨*proof*⟩

lemma *cblinfun-compose-id-left*[*simp*]:
shows $id\text{-}cblinfun \ o_{CL} \ U = U$
⟨*proof*⟩

lemma *cblinfun-compose-add-left*: ‹$(a + b) \ o_{CL} \ c = (a \ o_{CL} \ c) + (b \ o_{CL} \ c)$›
⟨*proof*⟩

lemma *cblinfun-compose-add-right*: ‹$a \ o_{CL} \ (b + c) = (a \ o_{CL} \ b) + (a \ o_{CL} \ c)$›
⟨*proof*⟩

lemma *cbilinear-cblinfun-compose*[*simp*]: *cbilinear cblinfun-compose*
⟨*proof*⟩

lemma *cblinfun-compose-sum-left*: ‹$(\sum i\in S. \ g \ i) \ o_{CL} \ x = (\sum i\in S. \ g \ i \ o_{CL} \ x)$›
⟨*proof*⟩

lemma *cblinfun-compose-sum-right*: ‹$x \ o_{CL} \ (\sum i\in S. \ g \ i) = (\sum i\in S. \ x \ o_{CL} \ g \ i)$›
⟨*proof*⟩

lemma *cblinfun-compose-minus-right*: ‹$a \ o_{CL} \ (b - c) = (a \ o_{CL} \ b) - (a \ o_{CL} \ c)$›
⟨*proof*⟩
lemma *cblinfun-compose-minus-left*: ‹$(a - b) \ o_{CL} \ c = (a \ o_{CL} \ c) - (b \ o_{CL} \ c)$›

⟨*proof*⟩

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 $adj\ x\ o_{CL}\ (x\ o_{CL}\ y) = id\text{-}cblinfun\ o_{CL}\ y$.
⟨*proof*⟩

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 $adj\ x *_V x *_V y = id\text{-}cblinfun *_V y$.
⟨*proof*⟩

lemma *cblinfun-compose-uminus-left*: ‹$(- a)\ o_{CL}\ b = - (a\ o_{CL}\ b)$›
⟨*proof*⟩

lemma *cblinfun-compose-uminus-right*: ‹$a\ o_{CL}\ (- b) = - (a\ o_{CL}\ b)$›
⟨*proof*⟩

lemma *bounded-clinear-cblinfun-compose-left*: ‹*bounded-clinear* ($\lambda x.\ x\ o_{CL}\ y$)›
⟨*proof*⟩
lemma *bounded-clinear-cblinfun-compose-right*: ‹*bounded-clinear* ($\lambda y.\ x\ o_{CL}\ y$)›
⟨*proof*⟩
lemma *clinear-cblinfun-compose-left*: ‹*clinear* ($\lambda x.\ x\ o_{CL}\ y$)›
⟨*proof*⟩
lemma *clinear-cblinfun-compose-right*: ‹*clinear* ($\lambda y.\ x\ o_{CL}\ y$)›
⟨*proof*⟩

lemma *additive-cblinfun-compose-left*[*simp*]: ‹*Modules.additive* ($\lambda x.\ x\ o_{CL}\ a$)›
⟨*proof*⟩
lemma *additive-cblinfun-compose-right*[*simp*]: ‹*Modules.additive* ($\lambda x.\ a\ o_{CL}\ x$)›
⟨*proof*⟩
lemma *isCont-cblinfun-compose-left*: ‹*isCont* ($\lambda x.\ x\ o_{CL}\ a$) *y*›
⟨*proof*⟩
lemma *isCont-cblinfun-compose-right*: ‹*isCont* ($\lambda x.\ a\ o_{CL}\ x$) *y*›
⟨*proof*⟩

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*⟩

13.4 Adjoint

lift-definition
adj :: *′a::chilbert-space* $\Rightarrow_{CL}$ *′b::complex-inner* ⇒ *′b* $\Rightarrow_{CL}$ *′a* (-* [*99*] *100*)

is *cadjoint* ⟨*proof*⟩

definition *selfadjoint* :: ‹('*a*::*chilbert-space* $\Rightarrow_{CL}$ '*a*) ⇒ *bool*› **where**
‹*selfadjoint a* ⟷ *a*∗ = *a*›

lemma *id-cblinfun-adjoint*[*simp*]: *id-cblinfun*∗ = *id-cblinfun*
⟨*proof*⟩

lemma *double-adj*[*simp*]: (*A*∗)∗ = *A*
⟨*proof*⟩

lemma *adj-cblinfun-compose*[*simp*]:
fixes *B*::‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*b*::*chilbert-space*›
and *A*::‹'*b* $\Rightarrow_{CL}$ '*c*::*complex-inner*›
shows (*A* o_{CL} *B*)∗ = (*B*∗) o_{CL} (*A*∗)
⟨*proof*⟩

lemma *scaleC-adj*[*simp*]: (*a* $*_C$ *A*)∗ = (*cnj a*) $*_C$ (*A*∗)
⟨*proof*⟩

lemma *scaleR-adj*[*simp*]: (*a* $*_R$ *A*)∗ = *a* $*_R$ (*A*∗)
⟨*proof*⟩

lemma *adj-plus*: ‹(*A* + *B*)∗ = (*A*∗) + (*B*∗)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *adj-0*[*simp*]: ‹*0*∗ = *0*›
⟨*proof*⟩

lemma *selfadjoint-0*[*simp*]: ‹*selfadjoint 0*›
⟨*proof*⟩

lemma *norm-adj*[*simp*]: ‹*norm* (*A*∗) = *norm A*›
for *A* :: ‹'*b*::*chilbert-space* $\Rightarrow_{CL}$ '*c*::*complex-inner*›
⟨*proof*⟩

lemma *antilinear-adj*[*simp*]: ‹*antilinear adj*›
⟨*proof*⟩

lemma *bounded-antilinear-adj*[*bounded-antilinear*, *simp*]: ‹*bounded-antilinear adj*›
⟨*proof*⟩

lemma *adjoint-eqI*:
fixes *G*:: ‹*'b*::*chilbert-space* $\Rightarrow_{CL}$ *'a*::*complex-inner*›
and *F*:: ‹*'a* $\Rightarrow_{CL}$ *'b*›
assumes ‹$\bigwedge x\ y.$ ((*cblinfun-apply F*) $x \bullet_C y$) = ($x \bullet_C$ (*cblinfun-apply G*) y)›
shows ‹$F = G*$›
⟨*proof*⟩

lemma *adj-uminus*: ‹$(-A)* = -\ (A*)$›
⟨*proof*⟩

lemma *cinner-real-hermiteanI*:
— Prop. II.2.12 in [1]
assumes ‹$\bigwedge\psi.\ \psi \bullet_C (A *_V \psi) \in \mathbb{R}$›
shows ‹$A* = A$›
⟨*proof*⟩

lemma *norm-AAadj*[*simp*]: ‹*norm* ($A\ o_{CL}\ A*$) = (*norm* $A)^2$› **for** A :: ‹*'a*::*chilbert-space* $\Rightarrow_{CL}$ *'b*::{*complex-inner*}›
⟨*proof*⟩

lemma *sum-adj*: ‹(*sum a F*)∗ = *sum* (λ*i*. (*a i*)∗) *F*›
⟨*proof*⟩

lemma *has-sum-adj*:
assumes ‹(*f has-sum x*) *I*›
shows ‹((λ*x*. *adj* (*f x*)) *has-sum adj x*) *I*›

⟨*proof*⟩

lemma *adj-minus*: ‹$(A - B)* = (A*) - (B*)$›
⟨*proof*⟩

lemma *cinner-hermitian-real*: ‹$x \bullet_C (A *_V x) \in \mathbb{R}$› **if** ‹*selfadjoint A*›
⟨*proof*⟩

lemma *adj-inject*: ‹*adj a* = *adj b* ⟷ *a* = *b*›
⟨*proof*⟩

lemma *norm-AadjA*[*simp*]: ‹*norm* ($A*\ o_{CL}\ A$) = (*norm* $A)^2$› **for** A :: ‹*'a*::*chilbert-space* $\Rightarrow_{CL}$ *'b*::*chilbert-space*›
⟨*proof*⟩

lemma *cspan-adj-closed*:
assumes ‹$\bigwedge a.\ a \in A \Longrightarrow a* \in A$›

assumes ‹$a \in$ *cspan* A›
shows ‹$a*$ $\in$ *cspan* A›
⟨*proof*⟩

13.5 Powers of operators

lift-definition *cblinfun-power* :: ‹$'a$::*complex-normed-vector* $\Rightarrow_{CL}$ $'a$ $\Rightarrow$ *nat* $\Rightarrow$ $'a$ $\Rightarrow_{CL}$ $'a$› **is**
‹λ(a::$'a$$\Rightarrow$$'a$) n. a ^^ n›
⟨*proof*⟩

lemma *cblinfun-power-0*[*simp*]: ‹*cblinfun-power* A *0* = *id-cblinfun*›
⟨*proof*⟩

lemma *cblinfun-power-Suc'*: ‹*cblinfun-power* A (*Suc* n) = A o_{CL} *cblinfun-power* A n›
⟨*proof*⟩

lemma *cblinfun-power-Suc*: ‹*cblinfun-power* A (*Suc* n) = *cblinfun-power* A n o_{CL} A›
⟨*proof*⟩

lemma *cblinfun-power-compose*[*simp*]: ‹*cblinfun-power* A n o_{CL} *cblinfun-power* A m = *cblinfun-power* A (n+m)›
⟨*proof*⟩

lemma *cblinfun-power-scaleC*: ‹*cblinfun-power* ($c *_C a$) n = c^n $*_C$ *cblinfun-power* a n›
⟨*proof*⟩

lemma *cblinfun-power-scaleR*: ‹*cblinfun-power* ($c *_R a$) n = c^n $*_R$ *cblinfun-power* a n›
⟨*proof*⟩

lemma *cblinfun-power-uminus*: ‹*cblinfun-power* ($-a$) n = ($-$*1*)^n $*_R$ *cblinfun-power* a n›
⟨*proof*⟩

lemma *cblinfun-power-adj*: ‹(*cblinfun-power* S n)$*$ = *cblinfun-power* ($S*$) n›
⟨*proof*⟩

13.6 Unitaries / isometries

definition *isometry*::‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*complex-inner* $\Rightarrow$ *bool*› **where**
‹*isometry* U $\longleftrightarrow$ $U*$ o_{CL} U = *id-cblinfun*›

definition *unitary*::‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*complex-inner* $\Rightarrow$ *bool*› **where**
‹*unitary* U $\longleftrightarrow$ ($U*$ o_{CL} U = *id-cblinfun*) $\wedge$ (U o_{CL} $U*$ = *id-cblinfun*)›

lemma *unitaryI*: ‹*unitary a*› **if** ‹$a*$ o_{CL} a = *id-cblinfun*› **and** ‹a o_{CL} $a*$ = *id-cblinfun*›
⟨*proof*⟩

lemma *unitary-twosided-isometry*: *unitary U* ⟷ *isometry U* ∧ *isometry* ($U*$)
⟨*proof*⟩

lemma *isometryD*[*simp*]: *isometry U* ⟹ $U*$ o_{CL} U = *id-cblinfun*
⟨*proof*⟩

lemma *unitaryD1*: *unitary U* ⟹ $U*$ o_{CL} U = *id-cblinfun*
⟨*proof*⟩

lemma *unitaryD2*[*simp*]: *unitary U* ⟹ U o_{CL} $U*$ = *id-cblinfun*
⟨*proof*⟩

lemma *unitary-isometry*[*simp*]: *unitary U* ⟹ *isometry U*
⟨*proof*⟩

lemma *unitary-adj*[*simp*]: *unitary* ($U*$) = *unitary U*
⟨*proof*⟩

lemma *isometry-cblinfun-compose*[*simp*]:
assumes *isometry A* **and** *isometry B*
shows *isometry* (A o_{CL} B)
⟨*proof*⟩

lemma *unitary-cblinfun-compose*[*simp*]: *unitary* (A o_{CL} B)
if *unitary A* **and** *unitary B*
⟨*proof*⟩

lemma *unitary-surj*:
assumes *unitary U*
shows *surj* (*cblinfun-apply U*)
⟨*proof*⟩

lemma *unitary-id*[*simp*]: *unitary id-cblinfun*
⟨*proof*⟩

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*⟩

lemma *isometry-preserves-norm*: ‹*isometry U* ⟹ *norm* ($U *_V \psi$) = *norm* ψ›
⟨*proof*⟩

lemma *norm-isometry-compose*:
  assumes ‹*isometry U*›
  shows ‹*norm* (U o_{CL} A) = *norm A*›
⟨*proof*⟩

lemma *norm-isometry*:
  fixes U :: ‹′*a*::{*chilbert-space*,*not-singleton*} $\Rightarrow_{CL}$ ′*b*::*complex-inner*›
  assumes ‹*isometry U*›
  shows ‹*norm U* = *1*›
  ⟨*proof*⟩

lemma *norm-preserving-isometry*: ‹*isometry U*› **if** ‹$\bigwedge\psi$. *norm* (U $*_V$ ψ) = *norm* ψ›
  ⟨*proof*⟩

lemma *norm-isometry-compose′*: ‹*norm* (A o_{CL} U) = *norm A*› **if** ‹*isometry* (U*)›
  ⟨*proof*⟩

lemma *unitary-nonzero*[*simp*]: ‹¬ *unitary* (*0* :: ′*a*::{*chilbert-space*, *not-singleton*} $\Rightarrow_{CL}$ -)›
  ⟨*proof*⟩

lemma *isometry-inj*:
  assumes ‹*isometry U*›
  shows ‹*inj-on U X*›
  ⟨*proof*⟩

lemma *unitary-inj*:
  assumes ‹*unitary U*›
  shows ‹*inj-on U X*›
  ⟨*proof*⟩

lemma *unitary-adj-inv*: ‹*unitary U* ⟹ *cblinfun-apply* (U*) = *inv* (*cblinfun-apply* U)›
  ⟨*proof*⟩

lemma *isometry-cinner-both-sides*:
  assumes ‹*isometry U*›
  shows ‹*cinner* (U x) (U y) = *cinner x y*›
  ⟨*proof*⟩

lemma *isometry-image-is-ortho-set*:
  assumes ‹*is-ortho-set A*›
  assumes ‹*isometry U*›
  shows ‹*is-ortho-set* (U ‘ A)›
  ⟨*proof*⟩

13.7 Product spaces

lift-definition *cblinfun-left* :: ‹*'a::complex-normed-vector* $\Rightarrow_{CL}$ (*'a*×*'b::complex-normed-vector*)›
is ‹(λ*x*. (*x*,*0*))›
⟨*proof*⟩
lift-definition *cblinfun-right* :: ‹*'b::complex-normed-vector* $\Rightarrow_{CL}$ (*'a::complex-normed-vector*×*'b*)›
is ‹(λ*x*. (*0*,*x*))›
⟨*proof*⟩

lemma *isometry-cblinfun-left*[*simp*]: ‹*isometry cblinfun-left*›
⟨*proof*⟩

lemma *isometry-cblinfun-right*[*simp*]: ‹*isometry cblinfun-right*›
⟨*proof*⟩

lemma *cblinfun-left-right-ortho*[*simp*]: ‹*cblinfun-left*∗ o_{CL} *cblinfun-right* = *0*›
⟨*proof*⟩

lemma *cblinfun-right-left-ortho*[*simp*]: ‹*cblinfun-right*∗ o_{CL} *cblinfun-left* = *0*›
⟨*proof*⟩

lemma *cblinfun-left-apply*[*simp*]: ‹*cblinfun-left* $*_V$ ψ = (ψ,*0*)›
⟨*proof*⟩

lemma *cblinfun-left-adj-apply*[*simp*]: ‹*cblinfun-left*∗ $*_V$ ψ = *fst* ψ›
⟨*proof*⟩

lemma *cblinfun-right-apply*[*simp*]: ‹*cblinfun-right* $*_V$ ψ = (*0*,ψ)›
⟨*proof*⟩

lemma *cblinfun-right-adj-apply*[*simp*]: ‹*cblinfun-right*∗ $*_V$ ψ = *snd* ψ›
⟨*proof*⟩

lift-definition *ccsubspace-Times* :: ‹*'a::complex-normed-vector ccsubspace* ⇒ *'b::complex-normed-vector ccsubspace* ⇒ (*'a*×*'b*) *ccsubspace*› **is**
Product-Type.Times
⟨*proof*⟩

lemma *ccspan-Times*: ‹*ccspan* (*S* × *T*) = *ccsubspace-Times* (*ccspan S*) (*ccspan T*)› **if** ‹*0* ∈ *S*› **and** ‹*0* ∈ *T*›
⟨*proof*⟩

lemma *ccspan-Times-sing1*: ‹*ccspan* ({*0*::*'a::complex-normed-vector*} × *B*) = *ccsubspace-Times 0* (*ccspan B*)›
⟨*proof*⟩

lemma *ccspan-Times-sing2*: ‹*ccspan* (*B* × {*0*::*'a::complex-normed-vector*}) = *ccsubspace-Times* (*ccspan B*) *0*›
⟨*proof*⟩

lemma *ccsubspace-Times-sup*: ‹*sup* (*ccsubspace-Times A B*) (*ccsubspace-Times C D*) = *ccsubspace-Times* (*sup A C*) (*sup B D*)›
⟨*proof*⟩

lemma *ccsubspace-Times-top-top*[*simp*]: ‹*ccsubspace-Times top top* = *top*›
⟨*proof*⟩

lemma *is-ortho-set-prod*:
assumes ‹*is-ortho-set B*› ‹*is-ortho-set B′*›
shows ‹*is-ortho-set* ((*B* × {*0*}) ∪ ({*0*} × *B′*))›
⟨*proof*⟩

lemma *ccsubspace-Times-ccspan*:
assumes ‹*ccspan B* = *S*› **and** ‹*ccspan B′* = *S′*›
shows ‹*ccspan* ((*B* × {*0*}) ∪ ({*0*} × *B′*)) = *ccsubspace-Times S S′*›
⟨*proof*⟩

lemma *is-onb-prod*:
assumes ‹*is-onb B*› ‹*is-onb B′*›
shows ‹*is-onb* ((*B* × {*0*}) ∪ ({*0*} × *B′*))›
⟨*proof*⟩

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* $\Rightarrow_{CL}$ ′*b*::*complex-normed-vector* ⇒ ′*a ccsubspace* ⇒ ′*b ccsubspace*› (**infixr** $*_S$ *70*)
is λ*A S*. *closure* (*A* ‘ *S*)
⟨*proof*⟩

lemma *cblinfun-image-mono*:
assumes *a1*: $S \leq T$
shows $A *_S S \leq A *_S T$
⟨*proof*⟩

lemma *cblinfun-image-0*[*simp*]:
shows $U *_S 0 = 0$
thm *zero-ccsubspace-def*
⟨*proof*⟩

lemma *cblinfun-image-bot*[*simp*]: $U *_S$ *bot* = *bot*
⟨*proof*⟩

lemma *cblinfun-image-sup*[*simp*]:
fixes A B :: ‹′*a*::*chilbert-space ccsubspace*› **and** U :: ′*a* $\Rightarrow_{CL}$′*b*::*chilbert-space*
shows ‹$U *_S$ (*sup* A B) = *sup* ($U *_S A$) ($U *_S B$)›
⟨*proof*⟩

lemma *scaleC-cblinfun-image*[*simp*]:
fixes A :: ‹′*a*::*chilbert-space* $\Rightarrow_{CL}$ ′*b* :: *chilbert-space*›
and S :: ‹′*a ccsubspace*› **and** α :: *complex*
shows ‹($\alpha *_C A$) $*_S$ S = $\alpha *_C$ ($A *_S S$)›
⟨*proof*⟩

lemma *cblinfun-image-id*[*simp*]:
id-cblinfun $*_S$ $\psi = \psi$
⟨*proof*⟩

lemma *cblinfun-compose-image*:
‹(A o_{CL} B) $*_S$ S = $A *_S$ ($B *_S S$)›
⟨*proof*⟩

lemmas *cblinfun-assoc-left* = *cblinfun-compose-assoc*[*symmetric*] *cblinfun-compose-image*[*symmetric*]
add.assoc[**where** *?′a*=′*a*::*chilbert-space* $\Rightarrow_{CL}$ ′*b*::*chilbert-space*, *symmetric*]
lemmas *cblinfun-assoc-right* = *cblinfun-compose-assoc cblinfun-compose-image*
add.assoc[**where** *?′a*=′*a*::*chilbert-space* $\Rightarrow_{CL}$ ′*b*::*chilbert-space*]

lemma *cblinfun-image-INF-leq*[*simp*]:
fixes U :: ′*b*::*complex-normed-vector* $\Rightarrow_{CL}$ ′*c*::*complex-normed-vector*
and V :: ′*a* ⇒ ′*b ccsubspace*
shows ‹$U *_S$ (*INF* $i \in X$. V i) ≤ (*INF* $i \in X$. $U *_S$ (V i))›
⟨*proof*⟩

lemma *isometry-cblinfun-image-inf-distrib′*:
fixes U::‹′*a*::*complex-normed-vector* $\Rightarrow_{CL}$ ′*b*::*cbanach*› **and** B C::′*a ccsubspace*
shows $U *_S$ (*inf* B C) ≤ *inf* ($U *_S B$) ($U *_S C$)
⟨*proof*⟩

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*⟩

lemma *cblinfun-fixes-range*:
assumes A o_{CL} $B = B$ **and** $\psi \in$ *space-as-set* ($B *_S$ *top*)
shows $A *_V \psi = \psi$

⟨*proof*⟩

lemma *zero-cblinfun-image*[*simp*]: *0* $*_S$ *S* = (*0*::- *ccsubspace*)
⟨*proof*⟩

lemma *cblinfun-image-INF-eq-general*:
fixes *V* :: *'a* ⇒ *'b*::*chilbert-space ccsubspace*
and *U* :: *'b* $\Rightarrow_{CL}$*'c*::*chilbert-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*: ⋀*i*. *V i* ≤ *Uinv* $*_S$ *top*
and ‹*X* ≠ {}›
shows *U* $*_S$ (*INF i*∈*X*. *V i*) = (*INF i*∈*X*. *U* $*_S$ *V i*)
⟨*proof*⟩

lemma *unitary-range*[*simp*]:
assumes *unitary U*
shows *U* $*_S$ *top* = *top*
⟨*proof*⟩

lemma *range-adjoint-isometry*:
assumes *isometry U*
shows *U*∗ $*_S$ *top* = *top*
⟨*proof*⟩

lemma *cblinfun-image-INF-eq*[*simp*]:
fixes *V* :: *'a* ⇒ *'b*::*chilbert-space ccsubspace*
and *U* :: *'b* $\Rightarrow_{CL}$ *'c*::*chilbert-space*
assumes ‹*isometry U*› ‹*X* ≠ {}›
shows *U* $*_S$ (*INF i*∈*X*. *V i*) = (*INF i*∈*X*. *U* $*_S$ *V i*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cblinfun-image-ccspan*:
shows *A* $*_S$ *ccspan G* = *ccspan* (($*_V$) *A* ‘ *G*)
⟨*proof*⟩

lemma *cblinfun-apply-in-image*[*simp*]: *A* $*_V$ *ψ* ∈ *space-as-set* (*A* $*_S$ ⊤)
⟨*proof*⟩

lemma *cblinfun-plus-image-distr*:
‹$(A + B) *_S S \leq A *_S S \sqcup B *_S S$›
⟨*proof*⟩

lemma *cblinfun-sum-image-distr*:
‹$(\sum i{\in}I.\ A\ i) *_S S \leq (SUP\ i{\in}I.\ A\ i *_S S)$›
⟨*proof*⟩

lemma *space-as-set-image-commute*:
assumes *UV*: ‹$U\ o_{CL}\ V = id\text{-}cblinfun$› **and** *VU*: ‹$V\ o_{CL}\ U = id\text{-}cblinfun$›

shows ‹$(*_V)\ U$ ‘ *space-as-set* T = *space-as-set* $(U *_S T)$›
⟨*proof*⟩

lemma *right-total-rel-ccsubspace*:
fixes R :: ‹$'a$::*complex-normed-vector* ⇒ $'b$::*complex-normed-vector* ⇒ *bool*›
assumes *UV*: ‹$U\ o_{CL}\ V = id\text{-}cblinfun$›
assumes *VU*: ‹$V\ o_{CL}\ U = id\text{-}cblinfun$›
assumes *R-def*: ‹$\bigwedge x\ y.\ R\ x\ y \longleftrightarrow x = U *_V y$›
shows ‹*right-total* (*rel-ccsubspace* R)›
⟨*proof*⟩

lemma *left-total-rel-ccsubspace*:
fixes R :: ‹$'a$::*complex-normed-vector* ⇒ $'b$::*complex-normed-vector* ⇒ *bool*›
assumes *UV*: ‹$U\ o_{CL}\ V = id\text{-}cblinfun$›
assumes *VU*: ‹$V\ o_{CL}\ U = id\text{-}cblinfun$›
assumes *R-def*: ‹$\bigwedge x\ y.\ R\ x\ y \longleftrightarrow y = U *_V x$›
shows ‹*left-total* (*rel-ccsubspace* R)›
⟨*proof*⟩

lemma *cblinfun-image-bot-zero*[*simp*]: ‹$A *_S top = bot \longleftrightarrow A = 0$›
⟨*proof*⟩

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 :: ‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*chilbert-space*›
assumes ‹*isometry* U›
assumes ‹$U *_S \top = \top$›
shows ‹*unitary* U›
⟨*proof*⟩

lemma *cblinfun-apply-in-image'*: $A *_V \psi \in$ *space-as-set* $(A *_S S)$ **if** ‹$\psi \in$ *space-as-set* S›
⟨*proof*⟩

lemma *cblinfun-image-ccspan-leqI*:
assumes ‹$\bigwedge v.\ v \in M \Longrightarrow A *_V v \in$ *space-as-set* T›

shows ‹$A *_S$ *ccspan* $M \leq T$›
⟨*proof*⟩

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*⟩

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 ‹$a *_V (b *_V x) = c *_V x$›
⟨*proof*⟩

lemma *cblinfun-image-def2*: ‹$A *_S S =$ *ccspan* $((*_V)\ A$ ‘ *space-as-set* $S)$›
⟨*proof*⟩

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* $(U$ ‘ $A)$›
⟨*proof*⟩

13.9 Sandwiches

lift-definition *sandwich* :: ‹$('a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*complex-inner*$)$ $\Rightarrow$ $(('a \Rightarrow_{CL} 'a) \Rightarrow_{CL} ('b \Rightarrow_{CL} 'b))$› **is**
‹$\lambda(A::'a \Rightarrow_{CL} 'b)\ B.\ A\ o_{CL}\ B\ o_{CL}\ A*$›
⟨*proof*⟩

lemma *sandwich-0*[*simp*]: ‹*sandwich* *0* = *0*›
⟨*proof*⟩

lemma *sandwich-apply*: ‹*sandwich* $A *_V B = A\ o_{CL}\ B\ o_{CL}\ A*$›
⟨*proof*⟩

lemma *sandwich-arg-compose*:
assumes ‹*isometry* U›
shows ‹*sandwich* $U\ x\ o_{CL}$ *sandwich* $U\ y =$ *sandwich* $U\ (x\ o_{CL}\ y)$›
⟨*proof*⟩

lemma *norm-sandwich*: ‹*norm* (*sandwich* A) = (*norm* $A)^2$› **for** A :: ‹$'a$::{*chilbert-space*} $\Rightarrow_{CL}$ $'b$::{*complex-inner*}›

⟨*proof*⟩

lemma *sandwich-apply-adj*: ‹*sandwich A* (*B*∗) = (*sandwich A B*)∗›
⟨*proof*⟩

lemma *sandwich-id*[*simp*]: *sandwich id-cblinfun* = *id-cblinfun*
⟨*proof*⟩

lemma *sandwich-compose*: ‹*sandwich* (*A* o_{CL} *B*) = *sandwich A* o_{CL} *sandwich B*›
⟨*proof*⟩

lemma *inj-sandwich-isometry*: ‹*inj* (*sandwich U*)› **if** [*simp*]: ‹*isometry U*› **for** *U* :: ‹′*a*::*chilbert-space* $\Rightarrow_{CL}$ ′*b*::*chilbert-space*›
⟨*proof*⟩

lemma *sandwich-isometry-id*: ‹*isometry* (*U*∗) ⟹ *sandwich U id-cblinfun* = *id-cblinfun*›
⟨*proof*⟩

13.10 Projectors

lift-definition *Proj* :: (′*a*::*chilbert-space*) *ccsubspace* ⇒ ′*a* $\Rightarrow_{CL}$′*a*
is ‹*projection*›
⟨*proof*⟩

lemma *Proj-range*[*simp*]: *Proj S* $*_S$ *top* = *S*
⟨*proof*⟩

lemma *adj-Proj*: ‹(*Proj M*)∗ = *Proj M*›
⟨*proof*⟩

lemma *Proj-idempotent*[*simp*]: ‹*Proj M* o_{CL} *Proj M* = *Proj M*›
⟨*proof*⟩

lift-definition *is-Proj* :: ‹′*a*::*complex-normed-vector* $\Rightarrow_{CL}$ ′*a* ⇒ *bool*› **is**
‹λ*P*. ∃ *M*. *is-projection-on P M*› ⟨*proof*⟩

lemma *is-Proj-id*[*simp*]: ‹*is-Proj id-cblinfun*›
⟨*proof*⟩

lemma *Proj-top*[*simp*]: ‹*Proj* ⊤ = *id-cblinfun*›
⟨*proof*⟩

lemma *Proj-on-own-range*′:
fixes *P* :: ‹′*a*::*chilbert-space* $\Rightarrow_{CL}$′*a*›
assumes ‹*P* o_{CL} *P* = *P*› **and** ‹*P* = *P*∗›
shows ‹*Proj* (*P* $*_S$ *top*) = *P*›
⟨*proof*⟩

lemma *Proj-range-closed*:

assumes *is-Proj P*
shows *closed* (*range* (*cblinfun-apply P*))
⟨*proof*⟩

lemma *Proj-is-Proj*[*simp*]:
fixes *M*::‹′*a*::*chilbert-space ccsubspace*›
shows ‹*is-Proj* (*Proj M*)›
⟨*proof*⟩

lemma *is-Proj-algebraic*:
fixes *P*::‹′*a*::*chilbert-space* $\Rightarrow_{CL}$ ′*a*›
shows ‹*is-Proj P* ⟷ *P* o_{CL} *P* = *P* ∧ *P* = *P*∗›
⟨*proof*⟩

lemma *Proj-on-own-range*:
fixes *P* :: ‹′*a*::*chilbert-space* $\Rightarrow_{CL}$′*a*›
assumes ‹*is-Proj P*›
shows ‹*Proj* (*P* $*_S$ *top*) = *P*›
⟨*proof*⟩

lemma *Proj-image-leq*: (*Proj S*) $*_S$ *A* ≤ *S*
⟨*proof*⟩

lemma *Proj-sandwich*:
fixes *A*::′*a*::*chilbert-space* $\Rightarrow_{CL}$ ′*b*::*chilbert-space*
assumes *isometry A*
shows *sandwich A* $*_V$ *Proj S* = *Proj* (*A* $*_S$ *S*)
⟨*proof*⟩

lemma *Proj-orthog-ccspan-union*:
assumes ⋀*x y*. *x* ∈ *X* ⟹ *y* ∈ *Y* ⟹ *is-orthogonal x y*
shows ‹*Proj* (*ccspan* (*X* ∪ *Y*)) = *Proj* (*ccspan X*) + *Proj* (*ccspan Y*)›
⟨*proof*⟩

abbreviation *proj* :: ′*a*::*chilbert-space* ⇒ ′*a* $\Rightarrow_{CL}$ ′*a* **where** *proj* ψ ≡ *Proj* (*ccspan* {ψ})

lemma *proj-0*[*simp*]: ‹*proj 0* = *0*›
⟨*proof*⟩

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*⟩

lemma *is-Proj-idempotent*:
assumes *is-Proj P*
shows *P* o_{CL} *P* = *P*

⟨*proof*⟩

lemma *is-proj-selfadj*:
assumes *is-Proj P*
shows $P* = P$
⟨*proof*⟩

lemma *is-Proj-I*:
assumes $P\ o_{CL}\ P = P$ **and** $P* = P$
shows *is-Proj P*
⟨*proof*⟩

lemma *is-Proj-0*[*simp*]: *is-Proj 0*
⟨*proof*⟩

lemma *is-Proj-complement*[*simp*]:
fixes *P* :: ‹*'a*::*chilbert-space* $\Rightarrow_{CL}$ *'a*›
assumes *a1*: *is-Proj P*
shows *is-Proj* (*id-cblinfun* − *P*)
⟨*proof*⟩

lemma *Proj-bot*[*simp*]: *Proj bot* = *0*
⟨*proof*⟩

lemma *Proj-ortho-compl*:
Proj (− *X*) = *id-cblinfun* − *Proj X*
⟨*proof*⟩

lemma *Proj-inj*:
assumes *Proj X* = *Proj Y*
shows *X* = *Y*
⟨*proof*⟩

lemma *norm-Proj-leq1*: ‹*norm* (*Proj M*) ≤ *1*› **for** *M* :: ‹*'a* :: *chilbert-space ccsub-space*›
⟨*proof*⟩

lemma *Proj-orthog-ccspan-insert*:
assumes ⋀*y*. *y* ∈ *Y* ⟹ *is-orthogonal x y*
shows ‹*Proj* (*ccspan* (*insert x Y*)) = *proj x* + *Proj* (*ccspan Y*)›
⟨*proof*⟩

lemma *Proj-fixes-image*: ‹*Proj S* $*_V$ ψ = ψ› **if** ‹ψ ∈ *space-as-set S*›
⟨*proof*⟩

lemma *norm-is-Proj*: ‹*norm P* ≤ *1*› **if** ‹*is-Proj P*› **for** *P* :: ‹*'a* :: *chilbert-space* $\Rightarrow_{CL}$ *'a*›
⟨*proof*⟩

lemma *Proj-sup*: ‹*orthogonal-spaces* S T $\Longrightarrow$ *Proj* (*sup* S T) = *Proj* S + *Proj* T›
⟨*proof*⟩

lemma *Proj-sum-spaces*:
assumes ‹*finite* X›
assumes ‹$\bigwedge x\ y.\ x{\in}X \Longrightarrow y{\in}X \Longrightarrow x{\neq}y \Longrightarrow$ *orthogonal-spaces* ($J\ x$) ($J\ y$)›
shows ‹*Proj* ($\sum x{\in}X.\ J\ x$) = ($\sum x{\in}X.$ *Proj* ($J\ x$))›
⟨*proof*⟩

lemma *is-Proj-reduces-norm*:
fixes P :: ‹$'a$::*complex-inner* $\Rightarrow_{CL}$ $'a$›
assumes ‹*is-Proj* P›
shows ‹*norm* ($P *_V \psi$) $\leq$ *norm* ψ›
⟨*proof*⟩

lemma *norm-Proj-apply*: ‹*norm* (*Proj* $T *_V \psi$) = *norm* $\psi \longleftrightarrow \psi \in$ *space-as-set* T›
⟨*proof*⟩

lemma *norm-Proj-apply-1*: ‹*norm* ψ = *1* $\Longrightarrow$ *norm* (*Proj* $T *_V \psi$) = *1* $\longleftrightarrow \psi \in$ *space-as-set* T›
⟨*proof*⟩

lemma *norm-is-Proj-nonzero*: ‹*norm* P = *1*› **if** ‹*is-Proj* P› **and** ‹$P \neq$ *0*› **for** P :: ‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'a$›
⟨*proof*⟩

lemma *Proj-compose-cancelI*:
assumes ‹$A *_S \top \leq S$›
shows ‹*Proj* S o_{CL} A = A›
⟨*proof*⟩

lemma *space-as-setI-via-Proj*:
assumes ‹*Proj* $M *_V x = x$›
shows ‹$x \in$ *space-as-set* M›
⟨*proof*⟩

lemma *unitary-image-ortho-compl*:
— Logically, this lemma belongs in an earlier section but its proof uses projectors.
fixes U :: ‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*chilbert-space*›
assumes [*simp*]: ‹*unitary* U›
shows ‹$U *_S (- A) = - (U *_S A)$›
⟨*proof*⟩

lemma *Proj-on-image* [*simp*]: ‹*Proj* $S *_S S = S$›
⟨*proof*⟩

13.11 Kernel / eigenspaces

lift-definition *kernel* :: $'a$::*complex-normed-vector* $\Rightarrow_{CL}$ $'b$::*complex-normed-vector* $\Rightarrow$ $'a$ *ccsubspace*
is $\lambda f.\ f -`\ \{0\}$
⟨*proof*⟩

definition *eigenspace* :: *complex* $\Rightarrow$ $'a$::*complex-normed-vector* $\Rightarrow_{CL}$$'a$ $\Rightarrow$ $'a$ *ccsubspace* **where**
eigenspace a A = *kernel* ($A - a *_C$ *id-cblinfun*)

lemma *kernel-scaleC*[*simp*]: $a \neq 0 \Longrightarrow$ *kernel* ($a *_C A$) = *kernel* A
for a :: *complex* **and** A :: (-,-) *cblinfun*
⟨*proof*⟩

lemma *kernel-0*[*simp*]: *kernel* 0 = *top*
⟨*proof*⟩

lemma *kernel-id*[*simp*]: *kernel id-cblinfun* = 0
⟨*proof*⟩

lemma *eigenspace-scaleC*[*simp*]:
assumes *a1*: $a \neq 0$
shows *eigenspace* b ($a *_C A$) = *eigenspace* (b/a) A
⟨*proof*⟩

lemma *eigenspace-memberD*:
assumes $x \in$ *space-as-set* (*eigenspace* e A)
shows $A *_V x = e *_C x$
⟨*proof*⟩

lemma *kernel-memberD*:
assumes $x \in$ *space-as-set* (*kernel* A)
shows $A *_V x = 0$
⟨*proof*⟩

lemma *eigenspace-memberI*:
assumes $A *_V x = e *_C x$
shows $x \in$ *space-as-set* (*eigenspace* e A)
⟨*proof*⟩

lemma *kernel-memberI*:
assumes $A *_V x = 0$
shows $x \in$ *space-as-set* (*kernel* A)
⟨*proof*⟩

lemma *kernel-Proj*[*simp*]: ‹*kernel* (*Proj* S) = $-$ S›
⟨*proof*⟩

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*⟩

lemma *cblinfun-compose-Proj-kernel*[*simp*]: ‹*a* o_{CL} *Proj* (*kernel a*) = *0*›
⟨*proof*⟩

lemma *kernel-compl-adj-range*:
shows ‹*kernel a* = − (*a*∗ $*_S$ *top*)›
⟨*proof*⟩

lemma *kernel-apply-self*: ‹*A* $*_S$ *kernel A* = *0*›
⟨*proof*⟩

lemma *leq-kernel-iff*:
shows ‹*A* ≤ *kernel B* ⟷ *B* $*_S$ *A* = *0*›
⟨*proof*⟩

lemma *cblinfun-image-kernel*:
assumes ‹*C* $*_S$ *A* $*_S$ *kernel B* ≤ *kernel B*›
assumes ‹*A* o_{CL} *C* = *id-cblinfun*›
shows ‹*A* $*_S$ *kernel B* = *kernel* (*B* o_{CL} *C*)›
⟨*proof*⟩

lemma *cblinfun-image-kernel-unitary*:
assumes ‹*unitary U*›
shows ‹*U* $*_S$ *kernel B* = *kernel* (*B* o_{CL} *U*∗)›
⟨*proof*⟩

lemma *kernel-cblinfun-compose*:
assumes ‹*kernel B* = *0*›
shows ‹*kernel A* = *kernel* (*B* o_{CL} *A*)›
⟨*proof*⟩

lemma *eigenspace-0*[*simp*]: ‹*eigenspace 0 A* = *kernel A*›
⟨*proof*⟩

lemma *kernel-isometry*: ‹*kernel U* = *0*› **if** ‹*isometry U*›
⟨*proof*⟩

lemma *cblinfun-image-eigenspace-isometry*:
assumes [*simp*]: ‹*isometry A*› **and** ‹*c* ≠ *0*›
shows ‹*A* $*_S$ *eigenspace c B* = *eigenspace c* (*sandwich A B*)›
⟨*proof*⟩

lemma *cblinfun-image-eigenspace-unitary*:

assumes [*simp*]: ‹*unitary A*›
shows ‹$A *_S$ *eigenspace c B* = *eigenspace c* (*sandwich A B*)›
⟨*proof*⟩

lemma *kernel-member-iff*: ‹$x \in$ *space-as-set* (*kernel A*) $\longleftrightarrow A *_V x = 0$›
⟨*proof*⟩

lemma *kernel-square*[*simp*]: ‹*kernel* ($A* \ o_{CL}\ A$) = *kernel A*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma
fixes *A* :: ‹*'a* :: *chilbert-space* $\Rightarrow_{CL}$ *'b* :: *complex-normed-vector*›
assumes *iso*: ‹$\bigwedge\psi$. $\psi \in$ *space-as-set V* $\Longrightarrow$ *norm* ($A *_V \psi$) = *norm* ψ›
assumes *zero*: ‹$\bigwedge\psi$. $\psi \in$ *space-as-set* (− *V*) $\Longrightarrow A *_V \psi = 0$›
shows *partial-isometryI'*: ‹*partial-isometry A*›
and *partial-isometry-initial*: ‹*kernel A* = − *V*›
⟨*proof*⟩

lemma *Proj-partial-isometry*[*simp*]: ‹*partial-isometry* (*Proj S*)›
⟨*proof*⟩

lemma *is-Proj-partial-isometry*: ‹*is-Proj P* $\Longrightarrow$ *partial-isometry P*› **for** *P* :: ‹- :: *chilbert-space* $\Rightarrow_{CL}$ -›
⟨*proof*⟩

lemma *isometry-partial-isometry*: ‹*isometry P* $\Longrightarrow$ *partial-isometry P*›
⟨*proof*⟩

lemma *unitary-partial-isometry*: ‹*unitary P* $\Longrightarrow$ *partial-isometry P*›
⟨*proof*⟩

lemma *norm-partial-isometry*:
fixes *A* :: ‹*'a* :: *chilbert-space* $\Rightarrow_{CL}$ *'b*::*complex-normed-vector*›
assumes ‹*partial-isometry A*› **and** ‹$A \neq 0$›
shows ‹*norm A* = *1*›
⟨*proof*⟩

lemma *partial-isometry-adj-a-o-a*:

assumes ‹*partial-isometry a*›
shows ‹$a* \; o_{CL} \; a = Proj \; (- \; kernel \; a)$›
⟨*proof*⟩

lemma *partial-isometry-square-proj*: ‹*is-Proj* $(a* \; o_{CL} \; a)$› **if** ‹*partial-isometry a*›
⟨*proof*⟩

lemma *partial-isometry-adj*[*simp*]: ‹*partial-isometry* $(a*)$› **if** ‹*partial-isometry a*›
for *a* :: ‹$'a$::*chilbert-space* $\Rightarrow_{CL}$ $'b$::*chilbert-space*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *inv-cblinfun-invertible*: ‹*iso-cblinfun A* ⟹ *invertible-cblinfun A*›
⟨*proof*⟩

lemma *cblinfun-inv-right*:
assumes ‹*iso-cblinfun A*›
shows ‹$A \; o_{CL}$ *cblinfun-inv* $A =$ *id-cblinfun*›
⟨*proof*⟩

lemma *cblinfun-inv-uniq*:
assumes $A \; o_{CL} \; B =$ *id-cblinfun* **and** $B \; o_{CL} \; A =$ *id-cblinfun*
shows *cblinfun-inv* $A = B$
⟨*proof*⟩

lemma *iso-cblinfun-unitary*: ‹*unitary A* ⟹ *iso-cblinfun A*›
⟨*proof*⟩

lemma *invertible-cblinfun-isometry*: ‹*isometry A* ⟹ *invertible-cblinfun A*›
⟨*proof*⟩

lemma *summable-cblinfun-apply-invertible*:

assumes ‹*invertible-cblinfun A*›
shows ‹$(\lambda x.\ A *_V\ g\ x)$ *summable-on S* $\longleftrightarrow$ *g summable-on S*›
⟨*proof*⟩

lemma *infsum-cblinfun-apply-invertible*:
assumes ‹*invertible-cblinfun A*›
shows ‹$(\sum_\infty x{\in}S.\ A *_V\ g\ x) = A *_V\ (\sum_\infty x{\in}S.\ g\ x)$›
⟨*proof*⟩

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* $(A{::}'a \Rightarrow_{CL} 'b)$ $(B{::}'a \Rightarrow_{CL} 'b)$
= *cnj* (*one-dim-iso* $(A *_V\ 1)$) $*$ *one-dim-iso* $(B *_V\ 1)$
instance
⟨*proof*⟩
end

instantiation *cblinfun* :: (*one-dim*, *one-dim*) *one-dim* **begin**
lift-definition *one-cblinfun* :: $'a \Rightarrow_{CL} 'b$ **is** *one-dim-iso*
⟨*proof*⟩
lift-definition *times-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$
is $\lambda f\ g.\ f$ *o one-dim-iso o g*
⟨*proof*⟩
lift-definition *inverse-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$ **is**
$\lambda f.$ $((*)$ (*one-dim-iso* (*inverse* (*f 1*)))) *o one-dim-iso*
⟨*proof*⟩
definition *divide-cblinfun* :: $'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b \Rightarrow 'a \Rightarrow_{CL} 'b$ **where**
divide-cblinfun A B = *A* $*$ *inverse B*
definition *canonical-basis-cblinfun* = $[1 :: 'a \Rightarrow_{CL} 'b]$
definition ‹*canonical-basis-length-cblinfun* (- :: $('a \Rightarrow_{CL} 'b)$ *itself*) = (*1*::*nat*)›
instance
⟨*proof*⟩
end

lemma *id-cblinfun-eq-1*[*simp*]: ‹*id-cblinfun* = *1*›
⟨*proof*⟩

lemma *one-dim-cblinfun-compose-is-times*[*simp*]:
fixes $A :: 'a{::}one\text{-}dim \Rightarrow_{CL} 'a$ **and** $B :: 'a \Rightarrow_{CL} 'a$
shows $A\ o_{CL}\ B = A * B$
⟨*proof*⟩

lemma *scaleC-one-dim-is-times*: ‹$r *_C\ x$ = *one-dim-iso* $r * x$›
⟨*proof*⟩

lemma *one-comp-one-cblinfun*[*simp*]: *1* o_{CL} *1* = *1*
⟨*proof*⟩

lemma *one-cblinfun-adj*[*simp*]: *1*∗ = *1*
⟨*proof*⟩

lemma *scaleC-1-apply*[*simp*]: ‹(*x* $*_C$ *1*) $*_V$ *y* = *x* $*_C$ *y*›
⟨*proof*⟩

lemma *cblinfun-apply-1-left*[*simp*]: ‹*1* $*_V$ *y* = *y*›
⟨*proof*⟩

lemma *of-complex-cblinfun-apply*[*simp*]: ‹*of-complex x* $*_V$ *y* = *one-dim-iso* (*x* $*_C$ *y*)›
⟨*proof*⟩

lemma *cblinfun-compose-1-left*[*simp*]: ‹*1* o_{CL} *x* = *x*›
⟨*proof*⟩

lemma *cblinfun-compose-1-right*[*simp*]: ‹*x* o_{CL} *1* = *x*›
⟨*proof*⟩

lemma *one-dim-iso-id-cblinfun*: ‹*one-dim-iso id-cblinfun* = *id-cblinfun*›
⟨*proof*⟩

lemma *one-dim-iso-id-cblinfun-eq-1*: ‹*one-dim-iso id-cblinfun* = *1*›
⟨*proof*⟩

lemma *one-dim-iso-comp-distr*[*simp*]: ‹*one-dim-iso* (*a* o_{CL} *b*) = *one-dim-iso a* o_{CL} *one-dim-iso b*›
⟨*proof*⟩

lemma *one-dim-iso-comp-distr-times*[*simp*]: ‹*one-dim-iso* (*a* o_{CL} *b*) = *one-dim-iso a* ∗ *one-dim-iso b*›
⟨*proof*⟩

lemma *one-dim-iso-adjoint*[*simp*]: ‹*one-dim-iso* (*A*∗) = (*one-dim-iso A*)∗›
⟨*proof*⟩

lemma *one-dim-iso-adjoint-complex*[*simp*]: ‹*one-dim-iso* (*A*∗) = *cnj* (*one-dim-iso A*)›
⟨*proof*⟩

lemma *one-dim-cblinfun-compose-commute*: ‹*a* o_{CL} *b* = *b* o_{CL} *a*› **for** *a b* :: ‹(′*a*::*one-dim*,′*a*) *cblinfun*›
⟨*proof*⟩

lemma *one-cblinfun-apply-one*[*simp*]: ‹*1* $*_V$ *1* = *1*›

⟨*proof*⟩

lemma *one-dim-cblinfun-apply-is-times*:
fixes $A :: 'a::one\text{-}dim \Rightarrow_{CL} 'b::one\text{-}dim$ **and** $b :: 'a$
shows $A *_V b = one\text{-}dim\text{-}iso\ A * one\text{-}dim\text{-}iso\ b$
⟨*proof*⟩

lemma *is-onb-one-dim*[*simp*]: ‹$norm\ x = 1 \Longrightarrow is\text{-}onb\ \{x\}$› **for** x :: ‹- :: *one-dim*›
⟨*proof*⟩

lemma *one-dim-iso-cblinfun-comp*: ‹$one\text{-}dim\text{-}iso\ (a\ o_{CL}\ b) = of\text{-}complex\ (cinner\ (a* *_V 1)\ (b *_V 1))$›
for a :: ‹$'a::chilbert\text{-}space \Rightarrow_{CL} 'b::one\text{-}dim$› **and** b :: ‹$'c::one\text{-}dim \Rightarrow_{CL} 'a$›
⟨*proof*⟩

lemma *one-dim-iso-cblinfun-apply*[*simp*]: ‹$one\text{-}dim\text{-}iso\ \psi *_V \varphi = one\text{-}dim\text{-}iso\ (one\text{-}dim\text{-}iso\ \psi *_C \varphi)$›
⟨*proof*⟩

13.15 Loewner order

lift-definition *heterogenous-cblinfun-id* :: ‹$'a::complex\text{-}normed\text{-}vector \Rightarrow_{CL} 'b::complex\text{-}normed\text{-}vector$›
is ‹*if bounded-clinear* (*heterogenous-identity* :: $'a::complex\text{-}normed\text{-}vector \Rightarrow 'b::complex\text{-}normed\text{-}vector$) *then heterogenous-identity else* (λ-. *0*)›
⟨*proof*⟩

lemma *heterogenous-cblinfun-id-def'*[*simp*]: *heterogenous-cblinfun-id* = *id-cblinfun*
⟨*proof*⟩

definition *heterogenous-same-type-cblinfun* ($x::'a::chilbert\text{-}space$ *itself*) ($y::'b::chilbert\text{-}space$ *itself*) $\longleftrightarrow$
unitary (*heterogenous-cblinfun-id* :: $'a \Rightarrow_{CL} 'b$) ∧ *unitary* (*heterogenous-cblinfun-id* :: $'b \Rightarrow_{CL} 'a$)

lemma *heterogenous-same-type-cblinfun*[*simp*]: ‹*heterogenous-same-type-cblinfun* ($x::'a::chilbert\text{-}space$ *itself*) ($y::'a::chilbert\text{-}space$ *itself*)›
⟨*proof*⟩

instantiation *cblinfun* :: (*chilbert-space*, *chilbert-space*) *ord* **begin**
definition *less-eq-cblinfun-def-heterogenous*: ‹$A \le B \longleftrightarrow$
(*if heterogenous-same-type-cblinfun TYPE*('a) *TYPE*('b) *then*
$\forall \psi::'b.\ cinner\ \psi\ ((B-A) *_V heterogenous\text{-}cblinfun\text{-}id *_V \psi) \ge 0$ *else* (A=B))›
definition ‹$(A :: 'a \Rightarrow_{CL} 'b) < B \longleftrightarrow A \le B \land \neg B \le A$›
instance⟨*proof*⟩
end

lemma *less-eq-cblinfun-def*: ‹$A \le B \longleftrightarrow$
$(\forall \psi.\ cinner\ \psi\ (A *_V \psi) \le cinner\ \psi\ (B *_V \psi))$›
⟨*proof*⟩

instantiation *cblinfun* :: (*chilbert-space*, *chilbert-space*) *ordered-complex-vector* **begin**
instance
⟨*proof*⟩
end

lemma *positive-id-cblinfun*[*simp*]: *id-cblinfun* ≥ *0*
⟨*proof*⟩

lemma *positive-hermitianI*: ‹*A*∗ = *A*› **if** ‹*A* ≥ *0*›
⟨*proof*⟩

lemma *cblinfun-leI*:
assumes ‹⋀*x*. *norm x* = *1* ⟹ *x* $\cdot_C$ (*A* $*_V$ *x*) ≤ *x* $\cdot_C$ (*B* $*_V$ *x*)›
shows ‹*A* ≤ *B*›
⟨*proof*⟩

lemma *positive-cblinfunI*: ‹*A* ≥ *0*› **if** ‹⋀*x*. *norm x* = *1* ⟹ *cinner x* (*A* $*_V$ *x*) ≥ *0*›
⟨*proof*⟩

lemma *less-eq-scaled-id-norm*:
assumes ‹*norm A* ≤ *c*› **and** ‹*selfadjoint A*›
shows ‹*A* ≤ *c* $*_R$ *id-cblinfun*›
⟨*proof*⟩

lemma *positive-cblinfun-squareI*: ‹*A* = *B*∗ o_{CL} *B* ⟹ *A* ≥ *0*›
⟨*proof*⟩

lemma *one-dim-loewner-order*: ‹*A* ≥ *B* ⟷ *one-dim-iso A* ≥ (*one-dim-iso B* :: *complex*)› **for** *A B* :: ‹'*a* $\Rightarrow_{CL}$ '*a*::{*chilbert-space*, *one-dim*}›
⟨*proof*⟩

lemma *one-dim-positive*: ‹*A* ≥ *0* ⟷ *one-dim-iso A* ≥ (*0*::*complex*)› **for** *A* :: ‹'*a* $\Rightarrow_{CL}$ '*a*::{*chilbert-space*, *one-dim*}›
⟨*proof*⟩

lemma *op-square-nondegenerate*: ‹*a* = *0*› **if** ‹*a*∗ o_{CL} *a* = *0*›
⟨*proof*⟩

lemma *comparable-hermitean*:
assumes ‹*a* ≤ *b*›
assumes ‹*selfadjoint a*›
shows ‹*selfadjoint b*›
⟨*proof*⟩

lemma *comparable-hermitean'*:
  assumes ‹$a \leq b$›
  assumes ‹*selfadjoint b*›
  shows ‹*selfadjoint a*›
  ⟨*proof*⟩

lemma *Proj-mono*: ‹*Proj S* ≤ *Proj T* ⟷ *S* ≤ *T*›
⟨*proof*⟩

13.16 Embedding vectors to operators

lift-definition *vector-to-cblinfun* :: ‹*'a::complex-normed-vector* ⇒ *'b::one-dim* $\Rightarrow_{CL}$ *'a*› **is**
  ‹*λψ φ. one-dim-iso φ* $*_C$ *ψ*›
  ⟨*proof*⟩

lemma *vector-to-cblinfun-cblinfun-compose*[*simp*]:
  A o_{CL} (*vector-to-cblinfun ψ*) = *vector-to-cblinfun* (*A* $*_V$ *ψ*)
  ⟨*proof*⟩

lemma *vector-to-cblinfun-add*: ‹*vector-to-cblinfun* (*x* + *y*) = *vector-to-cblinfun x* + *vector-to-cblinfun y*›
  ⟨*proof*⟩

lemma *norm-vector-to-cblinfun*[*simp*]: *norm* (*vector-to-cblinfun x*) = *norm x*
⟨*proof*⟩

lemma *bounded-clinear-vector-to-cblinfun*[*bounded-clinear*]: *bounded-clinear vector-to-cblinfun*
  ⟨*proof*⟩

lemma *vector-to-cblinfun-scaleC*[*simp*]:
  vector-to-cblinfun (*a* $*_C$ *ψ*) = *a* $*_C$ *vector-to-cblinfun ψ* **for** *a::complex*
  ⟨*proof*⟩

lemma *vector-to-cblinfun-apply-one-dim*[*simp*]:
  shows *vector-to-cblinfun φ* $*_V$ *γ* = *one-dim-iso γ* $*_C$ *φ*
  ⟨*proof*⟩

lemma *vector-to-cblinfun-one-dim-iso*[*simp*]: ‹*vector-to-cblinfun* = *one-dim-iso*›
  ⟨*proof*⟩

lemma *vector-to-cblinfun-adj-apply*[*simp*]:
  shows *vector-to-cblinfun ψ*∗ $*_V$ *φ* = *of-complex* (*cinner ψ φ*)
  ⟨*proof*⟩

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* $\Rightarrow_{CL}$ -)

⟨*proof*⟩

lemma *vector-to-cblinfun-0*[*simp*]: *vector-to-cblinfun 0 = 0*
⟨*proof*⟩

lemma *image-vector-to-cblinfun*[*simp*]: *vector-to-cblinfun x* $*_S$ ⊤ = *ccspan* {*x*}
— Not that the general case *vector-to-cblinfun x* $*_S$ *S* can be handled by using that *S* = ⊤ or *S* = ⊥ by *one-dim-ccsubspace-all-or-nothing*
⟨*proof*⟩

lemma *vector-to-cblinfun-adj-comp-vector-to-cblinfun*[*simp*]:
shows *vector-to-cblinfun ψ*∗ o_{CL} *vector-to-cblinfun φ* = *cinner ψ φ* $*_C$ *id-cblinfun*
⟨*proof*⟩

lemma *isometry-vector-to-cblinfun*[*simp*]:
assumes *norm x = 1*
shows *isometry* (*vector-to-cblinfun x*)
⟨*proof*⟩

lemma *image-vector-to-cblinfun-adj*:
assumes ‹*ψ* ∉ *space-as-set* (− *S*)›
shows ‹(*vector-to-cblinfun ψ*)∗ $*_S$ *S* = ⊤›
⟨*proof*⟩

lemma *image-vector-to-cblinfun-adj′*:
assumes ‹*ψ* ≠ *0*›
shows ‹(*vector-to-cblinfun ψ*)∗ $*_S$ ⊤ = ⊤›
⟨*proof*⟩

13.17 Rank-1 operators / butterflies

definition *rank1* **where** ‹*rank1 A* ⟷ (∃ *ψ*. *A* $*_S$ ⊤ = *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* $\Rightarrow_{CL}$ -)∗

abbreviation *selfbutter s* ≡ *butterfly s s*

lemma *butterfly-add-left*: ‹*butterfly* (*a* + *a′*) *b* = *butterfly a b* + *butterfly a′ b*›
⟨*proof*⟩

lemma *butterfly-add-right*: ‹*butterfly a* (*b* + *b′*) = *butterfly a b* + *butterfly a b′*›
⟨*proof*⟩

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*⟩

lemma *butterfly-comp-cblinfun*: *butterfly* ψ φ o_{CL} *a* = *butterfly* ψ (*a*∗ $*_V$ φ)
⟨*proof*⟩

lemma *cblinfun-comp-butterfly*: *a* o_{CL} *butterfly* ψ φ = *butterfly* (*a* $*_V$ ψ) φ
⟨*proof*⟩

lemma *butterfly-apply*[*simp*]: *butterfly* ψ ψ' $*_V$ φ = (ψ' $\bullet_C$ φ) $*_C$ ψ
⟨*proof*⟩

lemma *butterfly-scaleC-left*[*simp*]: *butterfly* (*c* $*_C$ ψ) φ = *c* $*_C$ *butterfly* ψ φ
⟨*proof*⟩

lemma *butterfly-scaleC-right*[*simp*]: *butterfly* ψ (*c* $*_C$ φ) = *cnj c* $*_C$ *butterfly* ψ φ
⟨*proof*⟩

lemma *butterfly-scaleR-left*[*simp*]: *butterfly* (*r* $*_R$ ψ) φ = *r* $*_C$ *butterfly* ψ φ
⟨*proof*⟩

lemma *butterfly-scaleR-right*[*simp*]: *butterfly* ψ (*r* $*_R$ φ) = *r* $*_C$ *butterfly* ψ φ
⟨*proof*⟩

lemma *butterfly-adjoint*[*simp*]: (*butterfly* ψ φ)∗ = *butterfly* φ ψ
⟨*proof*⟩

lemma *butterfly-comp-butterfly*[*simp*]: *butterfly* ψ*1* ψ*2* o_{CL} *butterfly* ψ*3* ψ*4* = (ψ*2* $\bullet_C$ ψ*3*) $*_C$ *butterfly* ψ*1* ψ*4*
⟨*proof*⟩

lemma *butterfly-0-left*[*simp*]: *butterfly 0 a* = *0*
⟨*proof*⟩

lemma *butterfly-0-right*[*simp*]: *butterfly a 0* = *0*
⟨*proof*⟩

lemma *butterfly-is-rank1*:
assumes ‹φ ≠ *0*›
shows ‹*butterfly* ψ φ $*_S$ ⊤ = *ccspan* {ψ}›
⟨*proof*⟩

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* $*_S$ ⊤ = *ccspan* {*ψ*::-::*chilbert-space*}›
shows ‹∃*φ*. *A* = *butterfly ψ φ*›
⟨*proof*⟩

lemma *rank1-0*[*simp*]: ‹*rank1 0*›
⟨*proof*⟩

lemma *rank1-iff-butterfly*: ‹*rank1 A* ⟷ (∃ *ψ φ*. *A* = *butterfly ψ φ*)›
for *A* :: ‹-::*complex-inner* $\Rightarrow_{CL}$ -::*chilbert-space*›
⟨*proof*⟩

lemma *norm-butterfly*: *norm* (*butterfly ψ φ*) = *norm ψ* ∗ *norm φ*
⟨*proof*⟩

lemma *bounded-sesquilinear-butterfly*[*bounded-sesquilinear*]: ‹*bounded-sesquilinear* (λ(*b*::'*b*::*chilbert-space*) (*a*::'*a*::*chilbert-space*). *butterfly a b*)›
⟨*proof*⟩

lemma *inj-selfbutter-upto-phase*:
assumes *selfbutter x* = *selfbutter y*
shows ∃ *c*. *cmod c* = *1* ∧ *x* = *c* $*_C$ *y*
⟨*proof*⟩

lemma *butterfly-eq-proj*:
assumes *norm x* = *1*
shows *selfbutter x* = *proj x*
⟨*proof*⟩

lemma *butterfly-sgn-eq-proj*:
shows *selfbutter* (*sgn x*) = *proj x*
⟨*proof*⟩

lemma *butterfly-is-Proj*:
‹*norm x* = *1* ⟹ *is-Proj* (*selfbutter x*)›
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *sum-butterfly-is-Proj*:
assumes ‹*is-ortho-set E*›
assumes ‹⋀*e*. *e*∈*E* ⟹ *norm e* = *1*›
shows ‹*is-Proj* (∑*e*∈*E*. *butterfly e e*)›
⟨*proof*⟩

lemma *rank1-compose-left*: ‹*rank1* (*a* o_{CL} *b*)› **if** ‹*rank1 b*›
⟨*proof*⟩

lemma *csubspace-of-1dim-space*:
assumes ‹*S* ≠ {*0*}›
assumes ‹*csubspace S*›
assumes ‹*S* ⊆ *cspan* {ψ}›
shows ‹*S* = *cspan* {ψ}›
⟨*proof*⟩

lemma *subspace-of-1dim-ccspan*:
assumes ‹*S* ≠ *0*›
assumes ‹*S* ≤ *ccspan* {ψ}›
shows ‹*S* = *ccspan* {ψ}›
⟨*proof*⟩

lemma *rank1-compose-right*: ‹*rank1* (*a* o_{CL} *b*)› **if** ‹*rank1 a*›
⟨*proof*⟩

lemma *rank1-scaleC*: ‹*rank1* (*c* $*_C$ *a*)› **if** ‹*rank1 a*› **and** ‹*c* ≠ *0*›
⟨*proof*⟩

lemma *rank1-uminus*: ‹*rank1* (−*a*)› **if** ‹*rank1 a*›
⟨*proof*⟩

lemma *rank1-uminus-iff*[*simp*]: ‹*rank1* (−*a*) ⟷ *rank1 a*›
⟨*proof*⟩

lemma *rank1-adj*: ‹*rank1* (*a*∗)› **if** ‹*rank1 a*›
for *a* :: ‹′*a*::*chilbert-space* $\Rightarrow_{CL}$ ′*b*::*chilbert-space*›
⟨*proof*⟩

lemma *rank1-adj-iff* [*simp*]: ‹*rank1* (*a*∗) ⟷ *rank1 a*›
for *a* :: ‹′*a*::*chilbert-space* $\Rightarrow_{CL}$ ′*b*::*chilbert-space*›
⟨*proof*⟩

lemma *butterflies-sum-id-finite*: ‹*id-cblinfun* = (∑ *x*∈*B*. *selfbutter x*)› **if** ‹*is-onb B*› **for** *B* :: ‹′*a* :: {*cfinite-dim*, *chilbert-space*} *set*›
⟨*proof*⟩

lemma *butterfly-sum-left*: ‹*butterfly* (∑ *i*∈*M*. ψ *i*) φ = (∑ *i*∈*M*. *butterfly* (ψ *i*) φ)›
⟨*proof*⟩

lemma *butterfly-sum-right*: ‹*butterfly* ψ (∑ *i*∈*M*. φ *i*) = (∑ *i*∈*M*. *butterfly* ψ (φ *i*))›
⟨*proof*⟩

13.18 Banach-Steinhaus

theorem *cbanach-steinhaus*:
fixes *F* :: ‹′*c* ⇒ ′*a*::*cbanach* $\Rightarrow_{CL}$ ′*b*::*complex-normed-vector*›
assumes ‹⋀*x*. ∃ *M*. ∀ *n*. *norm* ((*F n*) $*_V$ *x*) ≤ *M*›
shows ‹∃ *M*. ∀ *n*. *norm* (*F n*) ≤ *M*›
⟨*proof*⟩

13.19 Riesz-representation theorem

theorem *riesz-representation-cblinfun-existence*:
— Theorem 3.4 in [1]
fixes *f*::‹′*a*::*chilbert-space* $\Rightarrow_{CL}$ *complex*›
shows ‹∃ *t*. ∀ *x*. *f* $*_V$ *x* = (*t* $\bullet_C$ *x*)›
⟨*proof*⟩

lemma *riesz-representation-cblinfun-unique*:
— Theorem 3.4 in [1]
fixes *f*::‹′*a*::*complex-inner* $\Rightarrow_{CL}$ *complex*›
assumes ‹⋀*x*. *f* $*_V$ *x* = (*t* $\bullet_C$ *x*)›
assumes ‹⋀*x*. *f* $*_V$ *x* = (*u* $\bullet_C$ *x*)›
shows ‹*t* = *u*›
⟨*proof*⟩

theorem *riesz-representation-cblinfun-norm*:
includes *notation-norm*
fixes *f*::‹′*a*::*chilbert-space* $\Rightarrow_{CL}$ *complex*›
assumes ‹⋀*x*. *f* $*_V$ *x* = (*t* $\bullet_C$ *x*)›
shows ‹‖*f*‖ = ‖*t*‖›

⟨proof⟩

definition *the-riesz-rep* :: ‹(′a::chilbert-space $\Rightarrow_{CL}$ complex) ⇒ ′a› **where**
‹*the-riesz-rep f* = (*SOME t*. ∀ *x*. *f* $*_V$ *x* = *t* $\bullet_C$ *x*)›

lemma *the-riesz-rep*[*simp*]: ‹*the-riesz-rep f* $\bullet_C$ *x* = *f* $*_V$ *x*›
⟨proof⟩

lemma *the-riesz-rep-unique*:
assumes ‹⋀*x*. *f* $*_V$ *x* = *t* $\bullet_C$ *x*›
shows ‹*t* = *the-riesz-rep f*›
⟨proof⟩

lemma *the-riesz-rep-scaleC*: ‹*the-riesz-rep* (*c* $*_C$ *f*) = *cnj c* $*_C$ *the-riesz-rep f*›
⟨proof⟩

lemma *the-riesz-rep-add*: ‹*the-riesz-rep* (*f* + *g*) = *the-riesz-rep f* + *the-riesz-rep* *g*›
⟨proof⟩

lemma *the-riesz-rep-norm*[*simp*]: ‹*norm* (*the-riesz-rep f*) = *norm f*›
⟨proof⟩

lemma *bounded-antilinear-the-riesz-rep*[*bounded-antilinear*]: ‹*bounded-antilinear the-riesz-rep*›
⟨proof⟩

lift-definition *the-riesz-rep-sesqui* :: ‹(′a::complex-normed-vector ⇒ ′b::chilbert-space ⇒ complex) ⇒ (′a $\Rightarrow_{CL}$ ′b)› **is**
‹λ*p*. *if bounded-sesquilinear p then the-riesz-rep o CBlinfun o p else 0*›
⟨proof⟩

lemma *the-riesz-rep-sesqui-apply*:
assumes ‹*bounded-sesquilinear p*›
shows ‹(*the-riesz-rep-sesqui p* $*_V$ *x*) $\bullet_C$ *y* = *p x y*›
⟨proof⟩

13.20 Bidual

lift-definition *bidual-embedding* :: ‹′a::complex-normed-vector $\Rightarrow_{CL}$ ((′a $\Rightarrow_{CL}$ complex) $\Rightarrow_{CL}$ complex)›
is ‹λ*x f*. *f* $*_V$ *x*›
⟨proof⟩

lemma *bidual-embedding-apply*[*simp*]: ‹(*bidual-embedding* $*_V$ *x*) $*_V$ *f* = *f* $*_V$ *x*›
⟨proof⟩

lemma *bidual-embedding-isometric*[*simp*]: ‹*norm* (*bidual-embedding* $*_V$ *x*) = *norm* *x*› **for** *x* :: ‹′a::complex-inner›
⟨proof⟩

lemma *norm-bidual-embedding*[*simp*]: ‹*norm* (*bidual-embedding* :: *'a*::{*complex-inner*, *not-singleton*} $\Rightarrow_{CL}$ -) = *1*›
⟨*proof*⟩

lemma *isometry-bidual-embedding*[*simp*]: ‹*isometry bidual-embedding*›
⟨*proof*⟩

lemma *bidual-embedding-surj*[*simp*]: ‹*surj* (*bidual-embedding* :: *'a*::*chilbert-space* $\Rightarrow_{CL}$ -)›
⟨*proof*⟩

13.21 Extension of complex bounded operators

definition *cblinfun-extension* **where**
cblinfun-extension S φ = (*SOME B*. $\forall$ *x*$\in$*S*. *B* $*_V$ *x* = φ *x*)

definition *cblinfun-extension-exists* **where**
cblinfun-extension-exists S φ = ($\exists$ *B*. $\forall$ *x*$\in$*S*. *B* $*_V$ *x* = φ *x*)

lemma *cblinfun-extension-existsI*:
assumes $\bigwedge$*x*. *x*$\in$*S* $\Longrightarrow$ *B* $*_V$ *x* = φ *x*
shows *cblinfun-extension-exists S* φ
⟨*proof*⟩

lemma *cblinfun-extension-exists-finite-dim*:
fixes φ::*'a*::{*complex-normed-vector*,*cfinite-dim*} $\Rightarrow$ *'b*::*complex-normed-vector*
assumes *cindependent S*
and *cspan S* = *UNIV*
shows *cblinfun-extension-exists S* φ
⟨*proof*⟩

lemma *cblinfun-extension-apply*:
assumes *cblinfun-extension-exists S f*
and *v* $\in$ *S*
shows (*cblinfun-extension S f*) $*_V$ *v* = *f v*
⟨*proof*⟩

lemma
fixes *f* :: ‹*'a*::*complex-normed-vector* $\Rightarrow$ *'b*::*cbanach*›
assumes ‹*csubspace S*›
assumes ‹*closure S* = *UNIV*›
assumes *f-add*: ‹$\bigwedge$*x y*. *x* $\in$ *S* $\Longrightarrow$ *y* $\in$ *S* $\Longrightarrow$ *f* (*x* + *y*) = *f x* + *f y*›
assumes *f-scale*: ‹$\bigwedge$*c x y*. *x* $\in$ *S* $\Longrightarrow$ *f* (*c* $*_C$ *x*) = *c* $*_C$ *f x*›
assumes *bounded*: ‹$\bigwedge$*x*. *x* $\in$ *S* $\Longrightarrow$ *norm* (*f x*) $\leq$ *B* $*$ *norm x*›
shows *cblinfun-extension-exists-bounded-dense*: ‹*cblinfun-extension-exists S f*›
and *cblinfun-extension-norm-bounded-dense*: ‹*B* $\geq$ *0* $\Longrightarrow$ *norm* (*cblinfun-extension S f*) $\leq$ *B*›
⟨*proof*⟩

lemma *cblinfun-extension-cong*:
assumes ‹*cspan A = cspan B*›
assumes ‹$B \subseteq A$›
assumes *fg*: ‹$\bigwedge x.\ x{\in}B \Longrightarrow f\ x = g\ x$›
assumes ‹*cblinfun-extension-exists A f*›
shows ‹*cblinfun-extension A f = cblinfun-extension B g*›
⟨*proof*⟩

lemma
fixes *f* :: ‹*'a::complex-inner ⇒ 'b::chilbert-space*› **and** *S*
assumes ‹*is-ortho-set S*› **and** ‹*closure* (*cspan S*) = *UNIV*›
assumes *ortho-f*: ‹$\bigwedge x\ y.\ x{\in}S \Longrightarrow y{\in}S \Longrightarrow x{\neq}y \Longrightarrow$ *is-orthogonal* (*f x*) (*f y*)›
assumes *bounded*: ‹$\bigwedge x.\ x \in S \Longrightarrow$ *norm* (*f x*) $\leq B *$ *norm x*›
shows *cblinfun-extension-exists-ortho*: ‹*cblinfun-extension-exists S f*›
and *cblinfun-extension-exists-ortho-norm*: ‹$B \geq 0 \Longrightarrow$ *norm* (*cblinfun-extension S f*) $\leq B$›
⟨*proof*⟩

lemma *cblinfun-extension-exists-proj*:
fixes *f* :: ‹*'a::complex-normed-vector ⇒ 'b::cbanach*›
assumes ‹*csubspace S*›
assumes *ex-P*: ‹$\exists P :: 'a \Rightarrow_{CL} 'a.$ *is-Proj P* ∧ *range P = closure S*›
assumes *f-add*: ‹$\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow f\ (x + y) = f\ x + f\ y$›
assumes *f-scale*: ‹$\bigwedge c\ x\ y.\ x \in S \Longrightarrow f\ (c *_C x) = c *_C f\ x$›
assumes *bounded*: ‹$\bigwedge x.\ x \in S \Longrightarrow$ *norm* (*f x*) $\leq B *$ *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*⟩

lemma *cblinfun-extension-exists-hilbert*:
fixes *f* :: ‹*'a::chilbert-space ⇒ 'b::cbanach*›
assumes ‹*csubspace S*›
assumes *f-add*: ‹$\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow f\ (x + y) = f\ x + f\ y$›
assumes *f-scale*: ‹$\bigwedge c\ x\ y.\ x \in S \Longrightarrow f\ (c *_C x) = c *_C f\ x$›
assumes *bounded*: ‹$\bigwedge x.\ x \in S \Longrightarrow$ *norm* (*f x*) $\leq B *$ *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*⟩

lemma *cblinfun-extension-exists-restrict*:
assumes ‹$B \subseteq A$›
assumes ‹$\bigwedge x.\ x{\in}B \Longrightarrow f\ x = g\ x$›
assumes ‹*cblinfun-extension-exists A f*›

shows ‹*cblinfun-extension-exists B g*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*⟩

13.23 Notation

bundle *cblinfun-notation* **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

bundle *no-cblinfun-notation* **begin**
no-notation *cblinfun-compose* (**infixl** o_{CL} *67*)
no-notation *cblinfun-apply* (**infixr** $*_V$ *70*)
no-notation *cblinfun-image* (**infixr** $*_S$ *70*)
no-notation *adj* (-$*$ [*99*] *100*)
no-type-notation *cblinfun* ((- $\Rightarrow_{CL}$ /-) [*22*, *21*] *21*)
end

unbundle *no-cblinfun-notation*
unbundle *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
begin

unbundle *lattice-syntax*
unbundle *cblinfun-notation*
unbundle *no-notation-blinfun-apply*

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*)›
⟨*proof*⟩

lemma *has-ell2-norm-L2-set*: *has-ell2-norm x* = *bdd-above* (*L2-set* (*norm o x*) ‘ *Collect finite*)
⟨*proof*⟩

definition *ell2-norm* :: ‹(′*a* ⇒ *complex*) ⇒ *real*› **where** ‹*ell2-norm f* = *sqrt* ($\sum_\infty$*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)*
⟨*proof*⟩

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*⟩

lemma *has-ell2-norm-finite*[*simp*]: *has-ell2-norm (f::'a::finite⇒-)*
⟨*proof*⟩

lemma *ell2-norm-finite*:
ell2-norm (f::'a::finite⇒complex) = sqrt ($\sum$ *x∈UNIV. (norm (f x))^2)*
⟨*proof*⟩

lemma *ell2-norm-finite-L2-set*: *ell2-norm (x::'a::finite⇒complex) = L2-set (norm o x) UNIV*
⟨*proof*⟩

lemma *ell2-norm-square*: ‹$(ell2\text{-}norm\ x)^2 = (\sum_\infty i.\ (cmod\ (x\ i))^2)$›
⟨*proof*⟩

lemma *ell2-ket*:
fixes *a*
defines ‹*f ≡ (λi. of-bool (a = i))*›
shows *has-ell2-norm-ket*: ‹*has-ell2-norm f*›
and *ell2-norm-ket*: ‹*ell2-norm f = 1*›
⟨*proof*⟩

lemma *ell2-norm-geq0*: ‹*ell2-norm x ≥ 0*›
⟨*proof*⟩

lemma *ell2-norm-point-bound*:
assumes ‹*has-ell2-norm x*›
shows ‹*ell2-norm x ≥ cmod (x i)*›
⟨*proof*⟩

lemma *ell2-norm-0*:
assumes *has-ell2-norm x*
shows *ell2-norm x = 0 ⟷ x = (λ-. 0)*
⟨*proof*⟩

lemma *ell2-norm-smult*:
assumes *has-ell2-norm x*
shows *has-ell2-norm (λi. c * x i)* **and** *ell2-norm (λi. c * x i) = cmod c **

ell2-norm x
⟨*proof*⟩

lemma *ell2-norm-triangle*:
assumes *has-ell2-norm x* **and** *has-ell2-norm y*
shows *has-ell2-norm* (λ*i. x i* + *y i*) **and** *ell2-norm* (λ*i. x i* + *y i*) ≤ *ell2-norm x* + *ell2-norm y*
⟨*proof*⟩

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*›
⟨*proof*⟩

14.2 The type *ell2* of square-summable functions

typedef *'a ell2* = ‹{*f*::*'a*⇒*complex. has-ell2-norm f*}›
⟨*proof*⟩
setup-lifting *type-definition-ell2*

instantiation *ell2* :: (*type*)*complex-vector* **begin**
lift-definition *zero-ell2* :: *'a ell2* **is** λ-. *0* ⟨*proof*⟩
lift-definition *uminus-ell2* :: *'a ell2* ⇒ *'a ell2* **is** *uminus* ⟨*proof*⟩
lift-definition *plus-ell2* :: ‹*'a ell2* ⇒ *'a ell2* ⇒ *'a ell2*› **is** ‹λ*f g x. f x* + *g x*›
⟨*proof*⟩
lift-definition *minus-ell2* :: *'a ell2* ⇒ *'a ell2* ⇒ *'a ell2* **is** λ*f g x. f x* − *g x*
⟨*proof*⟩
lift-definition *scaleR-ell2* :: *real* ⇒ *'a ell2* ⇒ *'a ell2* **is** λ*r f x. complex-of-real r* ∗ *f x*
⟨*proof*⟩
lift-definition *scaleC-ell2* :: ‹*complex* ⇒ *'a ell2* ⇒ *'a ell2*› **is** ‹λ*c f x. c* ∗ *f x*›
⟨*proof*⟩

instance
⟨*proof*⟩
end

instantiation *ell2* :: (*type*)*complex-normed-vector* **begin**
lift-definition *norm-ell2* :: *'a ell2* ⇒ *real* **is** *ell2-norm* ⟨*proof*⟩
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*. $\forall_F$ (*x'*, *y*) *in INF e*∈{*0*<..}. *principal* {(*x*, *y*). *norm* (*x* − *y*) < *e*}. *x'* = *x* ⟶ *y* ∈ *U*) **for** *U* :: *'a ell2 set*
instance
⟨*proof*⟩

end

lemma *norm-point-bound-ell2*: *norm* (*Rep-ell2 x i*) ≤ *norm x*
⟨*proof*⟩

lemma *ell2-norm-finite-support*:
assumes ‹*finite S*› ‹⋀ *i*. *i* ∉ *S* ⟹ *Rep-ell2 x i* = *0*›
shows ‹*norm x* = *sqrt* ((*sum* (λ*i*. (*cmod* (*Rep-ell2 x i*))^2)) *S*)›
⟨*proof*⟩

instantiation *ell2* :: (*type*) *complex-inner* **begin**
lift-definition *cinner-ell2* :: ‹′*a ell2* ⇒ ′*a ell2* ⇒ *complex*› **is**
‹λ*f g*. $\sum_\infty$*x*. (*cnj* (*f x*) ∗ *g x*)› ⟨*proof*⟩
declare *cinner-ell2-def*[*code del*]

instance
⟨*proof*⟩
end

instance *ell2* :: (*type*) *chilbert-space*
⟨*proof*⟩

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*⟩

lemma *clinear-Rep-ell2*[*simp*]: ‹*clinear* (λψ. *Rep-ell2* ψ *i*)›
⟨*proof*⟩

lemma *Abs-ell2-inverse-finite*[*simp*]: ‹*Rep-ell2* (*Abs-ell2* ψ) = ψ› **for** ψ :: ‹-::*finite* ⇒ *complex*›
⟨*proof*⟩

14.3 Orthogonality

lemma *ell2-pointwise-ortho*:
assumes ‹⋀ *i*. *Rep-ell2 x i* = *0* ∨ *Rep-ell2 y i* = *0*›
shows ‹*is-orthogonal x y*›
⟨*proof*⟩

14.4 Truncated vectors

lift-definition *trunc-ell2*:: ‹′*a set* ⇒ ′*a ell2* ⇒ ′*a ell2*›
is ‹λ *S x*. (λ *i*. (*if i* ∈ *S then x i else 0*))›
⟨*proof*⟩

lemma *trunc-ell2-empty*[*simp*]: ‹*trunc-ell2* {} *x* = *0*›
⟨*proof*⟩

lemma *trunc-ell2-UNIV*[*simp*]: ‹*trunc-ell2 UNIV ψ = ψ*›
⟨*proof*⟩

lemma *norm-id-minus-trunc-ell2*:
‹(*norm* (*x* − *trunc-ell2 S x*))^2 = (*norm x*)^2 − (*norm* (*trunc-ell2 S x*))^2›
⟨*proof*⟩

lemma *norm-trunc-ell2-finite*:
‹*finite S* ⟹ (*norm* (*trunc-ell2 S x*)) = *sqrt* ((*sum* (λ*i*. (*cmod* (*Rep-ell2 x i*))^2)) *S*)›
⟨*proof*⟩

lemma *trunc-ell2-lim-at-UNIV*:
‹((λ*S*. *trunc-ell2 S ψ*) ⟶ *ψ*) (*finite-subsets-at-top UNIV*)›
⟨*proof*⟩

lemma *trunc-ell2-norm-mono*: ‹*M* ⊆ *N* ⟹ *norm* (*trunc-ell2 M ψ*) ≤ *norm* (*trunc-ell2 N ψ*)›
⟨*proof*⟩

lemma *trunc-ell2-reduces-norm*: ‹*norm* (*trunc-ell2 M ψ*) ≤ *norm ψ*›
⟨*proof*⟩

lemma *trunc-ell2-twice*[*simp*]: ‹*trunc-ell2 M* (*trunc-ell2 N ψ*) = *trunc-ell2* (*M*∩*N*) *ψ*›
⟨*proof*⟩

lemma *trunc-ell2-union*: ‹*trunc-ell2* (*M* ∪ *N*) *ψ* = *trunc-ell2 M ψ* + *trunc-ell2 N ψ* − *trunc-ell2* (*M*∩*N*) *ψ*›
⟨*proof*⟩

lemma *trunc-ell2-union-disjoint*: ‹*M* ∩ *N* = {} ⟹ *trunc-ell2* (*M* ∪ *N*) *ψ* = *trunc-ell2 M ψ* + *trunc-ell2 N ψ*›
⟨*proof*⟩

lemma *trunc-ell2-union-Diff*: ‹*M* ⊆ *N* ⟹ *trunc-ell2* (*N*−*M*) *ψ* = *trunc-ell2 N ψ* − *trunc-ell2 M ψ*›
⟨*proof*⟩

lemma *trunc-ell2-add*: ‹*trunc-ell2 M* (*ψ* + *φ*) = *trunc-ell2 M ψ* + *trunc-ell2 M φ*›
⟨*proof*⟩

lemma *trunc-ell2-scaleC*: ‹*trunc-ell2 M* (*c* $*_C$ *ψ*) = *c* $*_C$ *trunc-ell2 M ψ*›
⟨*proof*⟩

lemma *bounded-clinear-trunc-ell2*[*bounded-clinear*]: ‹*bounded-clinear* (*trunc-ell2 M*)›
⟨*proof*⟩

lemma *trunc-ell2-lim*: ‹((λS. *trunc-ell2 S* ψ) $\longrightarrow$ *trunc-ell2 M* ψ) (*finite-subsets-at-top M*)›
⟨*proof*⟩

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 ‹((λS. *trunc-ell2 S* ψ) $\longrightarrow$ *trunc-ell2 M* ψ) *F*›
⟨*proof*⟩

lemma *norm-ell2-bound-trunc*:
assumes ‹$\bigwedge M$. *finite M* $\Longrightarrow$ *norm* (*trunc-ell2 M* ψ) $\leq B$›
shows ‹*norm* $\psi \leq B$›
⟨*proof*⟩

lemma *trunc-ell2-uminus*: ‹*trunc-ell2* ($-M$) $\psi = \psi -$ *trunc-ell2 M* ψ›
⟨*proof*⟩

14.5 Kets and bras

lift-definition *ket* :: ‹*'a* $\Rightarrow$ *'a ell2*› **is** ‹$\lambda x\ y$. *of-bool* (x=y)›
⟨*proof*⟩

abbreviation *bra* :: *'a* $\Rightarrow$ (-,*complex*) *cblinfun* **where** *bra i* $\equiv$ *vector-to-cblinfun* (*ket i*)$*$ **for** *i*

instance *ell2* :: (*type*) *not-singleton*
⟨*proof*⟩

lemma *cinner-ket-left*: ‹*ket i* $\bullet_C$ ψ = *Rep-ell2* ψ *i*›
⟨*proof*⟩

lemma *cinner-ket-right*: ‹(ψ $\bullet_C$ *ket i*) = *cnj* (*Rep-ell2* ψ *i*)›
⟨*proof*⟩

lemma *bounded-clinear-Rep-ell2*[*simp*, *bounded-clinear*]: ‹*bounded-clinear* ($\lambda\psi$. *Rep-ell2* ψ *x*)›
⟨*proof*⟩

lemma *cinner-ket-eqI*:
assumes ‹$\bigwedge i$. *ket i* $\bullet_C$ ψ = *ket i* $\bullet_C$ φ›
shows ‹$\psi = \varphi$›
⟨*proof*⟩

lemma *norm-ket*[*simp*]: *norm* (*ket i*) = *1*
⟨*proof*⟩

lemma *cinner-ket-same*[*simp*]:
‹(*ket i* $\bullet_C$ *ket i*) = *1*›
⟨*proof*⟩

lemma *orthogonal-ket*[*simp*]:
‹*is-orthogonal* (*ket i*) (*ket j*) ⟷ *i* ≠ *j*›
⟨*proof*⟩

lemma *cinner-ket*: ‹(*ket i* $\bullet_C$ *ket j*) = *of-bool* (*i*=*j*)›
⟨*proof*⟩

lemma *ket-injective*[*simp*]: ‹*ket i* = *ket j* ⟷ *i* = *j*›
⟨*proof*⟩

lemma *inj-ket*[*simp*]: ‹*inj-on ket M*›
⟨*proof*⟩

lemma *trunc-ell2-ket-cspan*:
‹*trunc-ell2 S x* ∈ *cspan* (*range ket*)› **if** ‹*finite S*›
⟨*proof*⟩

lemma *closed-cspan-range-ket*[*simp*]:
‹*closure* (*cspan* (*range ket*)) = *UNIV*›
⟨*proof*⟩

lemma *ccspan-range-ket*[*simp*]: *ccspan* (*range ket*) = (*top*::(′*a ell2 ccsubspace*))
⟨*proof*⟩

lemma *cspan-range-ket-finite*[*simp*]: *cspan* (*range ket* :: ′*a*::*finite ell2 set*) = *UNIV*
⟨*proof*⟩

instance *ell2* :: (*finite*) *cfinite-dim*
⟨*proof*⟩

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*⟩
end

lemma *canonical-basis-length-ell2*[*code-unfold*, *simp*]:
length (*canonical-basis* ::′*a*::*enum ell2 list*) = *CARD*(′*a*)
⟨*proof*⟩

lemma *ket-canonical-basis*: *ket x* = *canonical-basis* ! *enum-idx x*
⟨*proof*⟩

lemma *clinear-equal-ket*:
  fixes *f g* :: ‹′*a*::*finite ell2* ⇒ -›
  assumes ‹*clinear f*›
  assumes ‹*clinear g*›
  assumes ‹⋀*i*. *f* (*ket i*) = *g* (*ket i*)›
  shows ‹*f* = *g*›
  ⟨*proof*⟩

lemma *equal-ket*:
  fixes *A B* :: ‹(′*a ell2*, ′*b*::*complex-normed-vector*) *cblinfun*›
  assumes ‹⋀ *x*. $A *_V$ *ket x* = $B *_V$ *ket x*›
  shows ‹*A* = *B*›
  ⟨*proof*⟩

lemma *antilinear-equal-ket*:
  fixes *f g* :: ‹′*a*::*finite ell2* ⇒ -›
  assumes ‹*antilinear f*›
  assumes ‹*antilinear g*›
  assumes ‹⋀*i*. *f* (*ket i*) = *g* (*ket i*)›
  shows ‹*f* = *g*›
⟨*proof*⟩

lemma *cinner-ket-adjointI*:
  fixes *F*::′*a ell2* $\Rightarrow_{CL}$ - **and** *G*::′*b ell2* $\Rightarrow_{CL}$ -
  assumes ⋀ *i j*. ($F *_V$ *ket i*) $\cdot_C$ *ket j* = *ket i* $\cdot_C$ ($G *_V$ *ket j*)
  shows *F* = *G*∗
⟨*proof*⟩

lemma *ket-nonzero*[*simp*]: *ket i* ≠ *0*
  ⟨*proof*⟩

lemma *cindependent-ket*[*simp*]:
  cindependent (*range* (*ket*::′*a*⇒-))
⟨*proof*⟩

lemma *cdim-UNIV-ell2*[*simp*]: ‹*cdim* (*UNIV*::′*a*::*finite ell2 set*) = *CARD*(′*a*)›
  ⟨*proof*⟩

lemma *is-ortho-set-ket*[*simp*]: ‹*is-ortho-set* (*range ket*)›
  ⟨*proof*⟩

lemma *bounded-clinear-equal-ket*:
  fixes *f g* :: ‹′*a ell2* ⇒ -›
  assumes ‹*bounded-clinear f*›
  assumes ‹*bounded-clinear g*›
  assumes ‹⋀*i*. *f* (*ket i*) = *g* (*ket i*)›
  shows ‹*f* = *g*›
  ⟨*proof*⟩

lemma *bounded-antilinear-equal-ket*:
fixes $f\ g$ :: ‹$'a\ ell2 \Rightarrow$ -›
assumes ‹*bounded-antilinear f*›
assumes ‹*bounded-antilinear g*›
assumes ‹$\bigwedge i.\ f\ (ket\ i) = g\ (ket\ i)$›
shows ‹$f = g$›
⟨*proof*⟩

lemma *is-onb-ket*[*simp*]: ‹*is-onb* (*range ket*)›
⟨*proof*⟩

lemma *ell2-sum-ket*: ‹$\psi = (\sum i{\in}UNIV.\ Rep\text{-}ell2\ \psi\ i *_C ket\ i)$› **for** ψ :: ‹-::*finite ell2*›
⟨*proof*⟩

lemma *trunc-ell2-singleton*: ‹$trunc\text{-}ell2\ \{x\}\ \psi = Rep\text{-}ell2\ \psi\ x *_C ket\ x$›
⟨*proof*⟩

lemma *trunc-ell2-insert*: ‹$trunc\text{-}ell2\ (insert\ x\ M)\ \varphi = Rep\text{-}ell2\ \varphi\ x *_C ket\ x + trunc\text{-}ell2\ M\ \varphi$›
if ‹$x \notin M$›
⟨*proof*⟩

lemma *trunc-ell2-finite-sum*: ‹$trunc\text{-}ell2\ M\ \psi = (\sum i{\in}M.\ Rep\text{-}ell2\ \psi\ i *_C ket\ i)$›
if ‹*finite M*›
⟨*proof*⟩

lemma *is-orthogonal-trunc-ell2*: ‹*is-orthogonal* (*trunc-ell2 M* ψ) (*trunc-ell2 N* φ)›
if ‹$M \cap N = \{\}$›
⟨*proof*⟩

14.6 Butterflies

lemma *cspan-butterfly-ket*: ‹$cspan\ \{butterfly\ (ket\ i)\ (ket\ j)|\ (i::'b::finite)\ (j::'a::finite).\ True\} = UNIV$›
⟨*proof*⟩

lemma *cindependent-butterfly-ket*: ‹$cindependent\ \{butterfly\ (ket\ i)\ (ket\ j)|\ (i::'b)\ (j::'a).\ True\}$›
⟨*proof*⟩

lemma *clinear-eq-butterfly-ketI*:
fixes $F\ G$ :: ‹$('a::finite\ ell2 \Rightarrow_{CL} 'b::finite\ ell2) \Rightarrow 'c::complex\text{-}vector$›
assumes *clinear F* **and** *clinear G*
assumes $\bigwedge i\ j.\ F\ (butterfly\ (ket\ i)\ (ket\ j)) = G\ (butterfly\ (ket\ i)\ (ket\ j))$
shows $F = G$
⟨*proof*⟩

lemma *sum-butterfly-ket*[*simp*]: ‹$(\sum (i::'a::finite){\in}UNIV.\ butterfly\ (ket\ i)\ (ket\ i))$

= *id-cblinfun*›
⟨*proof*⟩

lemma *ell2-decompose-has-sum*: ‹((λx. *Rep-ell2* φ *x* $*_C$ *ket x*) *has-sum* φ) *UNIV*›
⟨*proof*⟩

lemma *ell2-decompose-infsum*: ‹φ = ($\sum_\infty x$. *Rep-ell2* φ *x* $*_C$ *ket x*)›
⟨*proof*⟩

lemma *ell2-decompose-summable*: ‹(λx. *Rep-ell2* φ *x* $*_C$ *ket x*) *summable-on UNIV*›
⟨*proof*⟩

lemma *Rep-ell2-cblinfun-apply-sum*: ‹*Rep-ell2* (*A* $*_V$ φ) *y* = ($\sum_\infty x$. *Rep-ell2* φ *x* $*$ *Rep-ell2* (*A* $*_V$ *ket x*) *y*)›
⟨*proof*⟩

14.7 One-dimensional spaces

instantiation *ell2* :: (*CARD-1*) *one* **begin**
lift-definition *one-ell2* :: *'a ell2* **is** λ-. *1* ⟨*proof*⟩
instance⟨*proof*⟩
end

lemma *ket-CARD-1-is-1*: ‹*ket x* = *1*› **for** *x* :: ‹*'a*::*CARD-1*›
⟨*proof*⟩

instantiation *ell2* :: (*CARD-1*) *times* **begin**
lift-definition *times-ell2* :: *'a ell2* ⇒ *'a ell2* ⇒ *'a ell2* **is** $\lambda a\ b\ x.\ a\ x * b\ x$
⟨*proof*⟩
instance⟨*proof*⟩
end

instantiation *ell2* :: (*CARD-1*) *divide* **begin**
lift-definition *divide-ell2* :: *'a ell2* ⇒ *'a ell2* ⇒ *'a ell2* **is** $\lambda a\ b\ x.\ a\ x\ /\ b\ x$
⟨*proof*⟩
instance⟨*proof*⟩
end

instantiation *ell2* :: (*CARD-1*) *inverse* **begin**
lift-definition *inverse-ell2* :: *'a ell2* ⇒ *'a ell2* **is** $\lambda a\ x$. *inverse* (*a x*)
⟨*proof*⟩
instance⟨*proof*⟩
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*⟩

14.8 Explicit bounded operators

definition *explicit-cblinfun* :: ‹($'a \Rightarrow 'b \Rightarrow$ *complex*) $\Rightarrow$ ($'b$ *ell2*, $'a$ *ell2*) *cblinfun*›
where
‹*explicit-cblinfun M* = *cblinfun-extension* (*range ket*) (λa. *Abs-ell2* (λj. *M j* (*inv ket a*)))›

definition *explicit-cblinfun-exists* :: ‹($'a \Rightarrow 'b \Rightarrow$ *complex*) $\Rightarrow$ *bool*› **where**
‹*explicit-cblinfun-exists M* $\longleftrightarrow$
($\forall$ *a*. *has-ell2-norm* (λj. *M j a*)) $\wedge$
cblinfun-extension-exists (*range ket*) (λa. *Abs-ell2* (λj. *M j* (*inv ket a*)))›

lemma *explicit-cblinfun-exists-bounded*:
assumes ‹$\bigwedge S\ T\ \psi$. *finite S* $\Longrightarrow$ *finite T* $\Longrightarrow$ ($\bigwedge a$. $a \notin T \Longrightarrow \psi\ a = 0$) $\Longrightarrow$
($\sum b \in S$. (*cmod* ($\sum a \in T$. $\psi\ a *_C M\ b\ a$))^2) $\leq B *$ ($\sum a \in T$. (*cmod* (ψ *a*))^2)›
shows ‹*explicit-cblinfun-exists M*›
⟨*proof*⟩

lemma *explicit-cblinfun-exists-finite-dim*[*simp*]: ‹*explicit-cblinfun-exists m*› **for** *m* :: -::*finite* $\Rightarrow$ -::*finite* $\Rightarrow$ -
⟨*proof*⟩

lemma *explicit-cblinfun-ket*: ‹*explicit-cblinfun M* $*_V$ *ket a* = *Abs-ell2* (λb. *M b a*)›
if ‹*explicit-cblinfun-exists M*›
⟨*proof*⟩

lemma *Rep-ell2-explicit-cblinfun-ket*[*simp*]: ‹*Rep-ell2* (*explicit-cblinfun M* $*_V$ *ket a*) *b* = *M b a*› **if** ‹*explicit-cblinfun-exists M*›
⟨*proof*⟩

14.9 Classical operators

We call an operator mapping *ket x* to *ket* (π *x*) or *0*::$'a$ ”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* :: ($'a \Rightarrow 'b$ *option*) $\Rightarrow$ $'a$ *ell2* $\Rightarrow_{CL}$ $'b$ *ell2* **where**
classical-operator π =
(*let f* = (λt. (*case* π (*inv* (*ket*::$'a \Rightarrow$-) *t*)
of None $\Rightarrow$ (*0*::$'b$ *ell2*)
| *Some i* $\Rightarrow$ *ket i*))
in
cblinfun-extension (*range* (*ket*::$'a \Rightarrow$-)) *f*)

definition *classical-operator-exists* π $\longleftrightarrow$
cblinfun-extension-exists (*range ket*)
(λt. *case* π (*inv ket t*) *of None* $\Rightarrow$ *0* | *Some i* $\Rightarrow$ *ket i*)

lemma *classical-operator-existsI*:
assumes $\bigwedge x.\ B *_V (ket\ x) = (case\ \pi\ x\ of\ Some\ i \Rightarrow ket\ i \mid None \Rightarrow 0)$
shows *classical-operator-exists* π
⟨*proof*⟩

lemma
assumes *inj-map* π
shows *classical-operator-exists-inj*: *classical-operator-exists* π
and *classical-operator-norm-inj*: ‹*norm* (*classical-operator* π) $\leq$ *1*›
⟨*proof*⟩

lemma *classical-operator-exists-finite*[*simp*]: *classical-operator-exists* (π :: -::*finite* $\Rightarrow$ -)
⟨*proof*⟩

lemma *classical-operator-ket*:
assumes *classical-operator-exists* π
shows $(classical\text{-}operator\ \pi) *_V (ket\ x) = (case\ \pi\ x\ of\ Some\ i \Rightarrow ket\ i \mid None \Rightarrow 0)$
⟨*proof*⟩

lemma *classical-operator-ket-finite*:
$(classical\text{-}operator\ \pi) *_V (ket\ (x::'a::finite)) = (case\ \pi\ x\ of\ Some\ i \Rightarrow ket\ i \mid None \Rightarrow 0)$
⟨*proof*⟩

lemma *classical-operator-adjoint*[*simp*]:
fixes $\pi :: 'a \Rightarrow 'b\ option$
assumes *a1*: *inj-map* π
shows $(classical\text{-}operator\ \pi)* = classical\text{-}operator\ (inv\text{-}map\ \pi)$
⟨*proof*⟩

lemma
fixes $\pi::'b \Rightarrow 'c\ option$ **and** $\varrho::'a \Rightarrow 'b\ option$
assumes *classical-operator-exists* π
assumes *classical-operator-exists* ϱ
shows *classical-operator-exists-comp*[*simp*]: *classical-operator-exists* $(\pi \circ_m \varrho)$
and *classical-operator-mult*[*simp*]: $classical\text{-}operator\ \pi\ o_{CL}\ classical\text{-}operator\ \varrho = classical\text{-}operator\ (\pi \circ_m \varrho)$
⟨*proof*⟩

lemma *classical-operator-Some*[*simp*]: *classical-operator* (*Some*::'*a*⇒-) = *id-cblinfun*
⟨*proof*⟩

lemma *isometry-classical-operator*[*simp*]:
fixes $\pi::'a \Rightarrow 'b$
assumes *a1*: *inj* π
shows *isometry* (*classical-operator* (*Some o* π))
⟨*proof*⟩

lemma *unitary-classical-operator*[*simp*]:
  fixes π::'*a* ⇒ '*b*
  assumes *a1*: *bij* π
  shows *unitary* (*classical-operator* (*Some o* π))
⟨*proof*⟩

unbundle *no-lattice-syntax*
unbundle *no-cblinfun-notation*

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-notation*" in a lemma statement. (Or sandwich a declaration using that notation between "**unbundle** *jnf-notation* **...** **unbundle** *no-jnf-notation*.)

bundle *jnf-notation* **begin**
notation *transpose-mat* ((-^{T}) [*1000*])
notation *cscalar-prod* (**infix** ∙c *70*)
notation *vec-index* (**infixl** $ *100*)
notation *smult-vec* (**infixl** $\cdot_v$ *70*)
notation *scalar-prod* (**infix** ∙ *70*)
notation *index-mat* (**infixl** $$ *100*)
notation *smult-mat* (**infixl** $\cdot_m$ *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

bundle *no-jnf-notation* **begin**
no-notation *transpose-mat* ((-^{T}) [*1000*])
no-notation *cscalar-prod* (**infix** ∙c *70*)
no-notation *vec-index* (**infixl** $ *100*)
no-notation *smult-vec* (**infixl** $\cdot_v$ *70*)
no-notation *scalar-prod* (**infix** ∙ *70*)
no-notation *index-mat* (**infixl** $$ *100*)
no-notation *smult-mat* (**infixl** $\cdot_m$ *70*)

no-notation *mult-mat-vec* (**infixl** $*_v$ *70*)
no-notation *pow-mat* (**infixr** $\widehat{\ }_m$ *75*)
no-notation *append-vec* (**infixr** $@_v$ *65*)
no-notation *append-rows* (**infixr** $@_r$ *65*)
end

unbundle *jnf-notation*

lemma *mat-entry-explicit*:
  fixes *M* :: *'a*::*field mat*
  assumes $M \in$ *carrier-mat m n* **and** $i < m$ **and** $j < n$
  shows *vec-index* ($M *_v$ *unit-vec n j*) $i = M$ $$ (*i*,*j*)
  ⟨*proof*⟩

lemma *mat-adjoint-def'*: *mat-adjoint M* = *transpose-mat* (*map-mat conjugate M*)
  ⟨*proof*⟩

lemma *mat-adjoint-swap*:
  fixes *M* ::*complex mat*
  assumes $M \in$ *carrier-mat nB nA* **and** *iA* < *dim-row M* **and** *iB* < *dim-col M*
  shows (*mat-adjoint M*)$$(*iB*,*iA*) = *cnj* (*M*$$(*iA*,*iB*))
  ⟨*proof*⟩

lemma *cscalar-prod-adjoint*:
  fixes *M*:: *complex mat*
  assumes $M \in$ *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*
  ⟨*proof*⟩

lemma *scaleC-minus1-left-vec*: $-1 \cdot_v v = - v$ **for** *v* :: *-*::*ring-1 vec*
  ⟨*proof*⟩

lemma *square-nneg-complex*:
  fixes *x* :: *complex*
  assumes $x \in \mathbb{R}$ **shows** $x\widehat{\ }2 \geq 0$
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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′*)
⟨*proof*⟩

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*⟩

locale *complex-vec-space = cof-vec-space n TYPE*(*complex*) **for** *n* :: *nat*

lemma *gram-schmidt0-corthogonal*:
assumes *a1*: *corthogonal R*
and *a2*: ⋀*x. x ∈ set R ⟹ dim-vec x = d*
shows *gram-schmidt0 d R = rev R*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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*⟩

lemma *list-of-vec-mult*:
  fixes *v* :: ‹*complex vec*›
  shows ‹*list-of-vec* ($c \cdot_v v$) = *map* ((∗) *c*) (*list-of-vec v*)›
  ⟨*proof*⟩

lemma *map-map-vec-cols*: ‹*map* (*map-vec f*) (*cols m*) = *cols* (*map-mat f m*)›
  ⟨*proof*⟩

lemma *map-vec-conjugate*: ‹*map-vec conjugate v* = *conjugate v*›
  ⟨*proof*⟩

unbundle *no-jnf-notation*

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-notation*
unbundle *cblinfun-notation*

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*)›
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *vec-of-basis-enum-inj*: *inj vec-of-basis-enum*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*⟩

lemma *basis-enum-of-vec-mult*:
assumes [*simp*]: ‹*dim-vec v* = *length* (*canonical-basis* :: ′*a*::*basis-enum list*)›
shows ‹((*basis-enum-of-vec* ($c \cdot_v$ *v*)) :: ′*a*) = $c *_C$ *basis-enum-of-vec v*›
⟨*proof*⟩

lemma *vec-of-basis-enum-add*:
‹*vec-of-basis-enum* (*a* + *b*) = *vec-of-basis-enum a* + *vec-of-basis-enum b*›
⟨*proof*⟩

lemma *vec-of-basis-enum-scaleC*:
vec-of-basis-enum ($c *_C$ *b*) = $c \cdot_v$ (*vec-of-basis-enum b*)
⟨*proof*⟩

lemma *vec-of-basis-enum-scaleR*:
vec-of-basis-enum ($r *_R$ *b*) = *complex-of-real r* $\cdot_v$ (*vec-of-basis-enum b*)
⟨*proof*⟩

lemma *vec-of-basis-enum-uminus*:
vec-of-basis-enum (− *b2*) = − *vec-of-basis-enum b2*
⟨*proof*⟩

lemma *vec-of-basis-enum-minus*:
vec-of-basis-enum (*b1* − *b2*) = *vec-of-basis-enum b1* − *vec-of-basis-enum b2*

⟨*proof*⟩

lemma *cinner-basis-enum-of-vec*:
defines $n \equiv$ *length* (*canonical-basis* :: *'a*::*onb-enum list*)
assumes [*simp*]: *dim-vec* $x = n$ *dim-vec* $y = n$
shows (*basis-enum-of-vec* x :: *'a*) $\cdot_C$ *basis-enum-of-vec* $y = y \cdot c\ x$
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *basis-enum-of-vec-unit-vec*:
defines *basis* $\equiv$ (*canonical-basis*::*'a*::*basis-enum list*)
and $n \equiv$ *length* (*canonical-basis* :: *'a list*)
assumes *a3*: $i < n$
shows *basis-enum-of-vec* (*unit-vec* n i) = *basis*!i
⟨*proof*⟩

lemma *vec-of-basis-enum-ket*:
vec-of-basis-enum (*ket* i) = *unit-vec* (*CARD*(*'a*)) (*enum-idx* i)
for i::*'a*::*enum*
⟨*proof*⟩

lemma *vec-of-basis-enum-zero*:
defines $nA \equiv$ *length* (*canonical-basis* :: *'a*::*basis-enum list*)
shows *vec-of-basis-enum* (*0*::*'a*) = 0_v nA
⟨*proof*⟩

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*⟩

lemma (**in** *complex-vec-space*) *basis-enum-of-vec-span*:
assumes *length* (*canonical-basis* :: *'a list*) = n
assumes $Y \subseteq$ *carrier-vec* n
shows *basis-enum-of-vec* ‘ *local.span* Y = *cspan* (*basis-enum-of-vec* ‘ Y :: *'a*::*basis-enum set*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *vec-of-basis-enum-times*:
  fixes ψ φ :: *'a*::*one-dim*
  shows *vec-of-basis-enum* ($\psi * \varphi$)
  = *vec-of-list* [*vec-index* (*vec-of-basis-enum* ψ) *0* $*$ *vec-index* (*vec-of-basis-enum* φ) *0*]
⟨*proof*⟩

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*⟩

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*]
  ⟨*proof*⟩

lemma *vec-of-basis-enum-1*: *vec-of-basis-enum* (*1* :: *'a*::*one-dim*) = *vec-of-list* [*1*]
⟨*proof*⟩

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*⟩

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*⟩

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*} $\Rightarrow_{CL}$ *'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*} $\Rightarrow_{CL}$*'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*⟩

lemma *cblinfun-of-mat-invalid*:
assumes ‹*M* ∉ *carrier-mat* (*canonical-basis-length TYPE*(*'b*::{*basis-enum*,*complex-normed-vector*})) (*canonical-basis-length TYPE*(*'a*::{*basis-enum*,*complex-normed-vector*}))›
shows ‹(*cblinfun-of-mat M* :: *'a* $\Rightarrow_{CL}$ *'b*) = *0*›
⟨*proof*⟩

lemma *dim-row-mat-of-cblinfun*[*simp*]: ‹*dim-row* (*mat-of-cblinfun* (*a*::*'a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ *'b*::{*basis-enum*,*complex-normed-vector*})) = *canonical-basis-length TYPE*(*'b*)›
⟨*proof*⟩

lemma *dim-col-mat-of-cblinfun*[*simp*]: ‹*dim-col* (*mat-of-cblinfun* (*a*::*'a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ *'b*::{*basis-enum*,*complex-normed-vector*})) = *canonical-basis-length TYPE*(*'a*)›
⟨*proof*⟩

lemma *mat-of-cblinfun-ell2-carrier*[*simp*]: ‹*mat-of-cblinfun* (*a*::*'a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ *'b*::{*basis-enum*,*complex-normed-vector*}) ∈ *carrier-mat* (*canonical-basis-length TYPE*(*'b*)) (*canonical-basis-length TYPE*(*'a*))›
⟨*proof*⟩

lemma *basis-enum-of-vec-cblinfun-apply*:
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** *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*
⟨*proof*⟩

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-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ '*b*::{*basis-enum*,*complex-normed-vector*}
and u::'*a*
⟨*proof*⟩

lemma *mat-of-cblinfun-inverse*:
cblinfun-of-mat (*mat-of-cblinfun* B) = B
for B::'*a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ '*b*::{*basis-enum*,*complex-normed-vector*}
⟨*proof*⟩

lemma *mat-of-cblinfun-inj*: *inj mat-of-cblinfun*
⟨*proof*⟩

lemma *cblinfun-of-mat-inverse*:
fixes M::*complex mat*
defines $nA \equiv$ *length* (*canonical-basis* :: '*a*::{*basis-enum*,*complex-normed-vector*} *list*)
and $nB \equiv$ *length* (*canonical-basis* :: '*b*::{*basis-enum*,*complex-normed-vector*} *list*)
assumes $M \in$ *carrier-mat nB nA*
shows *mat-of-cblinfun* (*cblinfun-of-mat* M :: '*a* $\Rightarrow_{CL}$ '*b*) = M
⟨*proof*⟩

lemma *cblinfun-of-mat-inj*: *inj-on* (*cblinfun-of-mat*::*complex mat* ⇒ '*a* $\Rightarrow_{CL}$ '*b*)
(*carrier-mat* (*length* (*canonical-basis* :: '*b*::{*basis-enum*,*complex-normed-vector*} *list*))
(*length* (*canonical-basis* :: '*a*::{*basis-enum*,*complex-normed-vector*} *list*)))
⟨*proof*⟩

lemma *cblinfun-eq-mat-of-cblinfunI*:
assumes *mat-of-cblinfun a* = *mat-of-cblinfun b*
shows $a = b$
⟨*proof*⟩

16.4 Operations on matrices

lemma *cblinfun-of-mat-plus*:
defines $nA \equiv$ *length* (*canonical-basis* :: '*a*::{*basis-enum*,*complex-normed-vector*} *list*)
and $nB \equiv$ *length* (*canonical-basis* :: '*b*::{*basis-enum*,*complex-normed-vector*} *list*)
assumes [*simp*,*intro*]: $M \in$ *carrier-mat nB nA* **and** [*simp*,*intro*]: $N \in$ *carrier-mat nB nA*
shows (*cblinfun-of-mat* $(M + N)$ :: '*a* $\Rightarrow_{CL}$ '*b*) = ((*cblinfun-of-mat* M + *cblinfun-of-mat* N))
⟨*proof*⟩

lemma *mat-of-cblinfun-zero*:
mat-of-cblinfun (*0* :: ('*a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ '*b*::{*basis-enum*,*complex-normed-vector*}))
= 0_m (*length* (*canonical-basis* :: '*b list*)) (*length* (*canonical-basis* :: '*a list*))
⟨*proof*⟩

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*} $\Rightarrow_{CL}$'*b*::{*basis-enum*,*complex-normed-vector*}
⟨*proof*⟩

lemma *mat-of-cblinfun-id*:
mat-of-cblinfun (*id-cblinfun* :: ('*a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$'*a*))
= 1_m (*length* (*canonical-basis* :: '*a list*))
⟨*proof*⟩

lemma *mat-of-cblinfun-1*:
mat-of-cblinfun (*1* :: ('*a*::*one-dim* $\Rightarrow_{CL}$'*b*::*one-dim*)) = 1_m *1*
⟨*proof*⟩

lemma *mat-of-cblinfun-uminus*:
mat-of-cblinfun (− *M*) = − *mat-of-cblinfun M*
for *M*::'*a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$'*b*::{*basis-enum*,*complex-normed-vector*}
⟨*proof*⟩

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*} $\Rightarrow_{CL}$ '*b*::{*basis-enum*,*complex-normed-vector*}
and *N*::'*a* $\Rightarrow_{CL}$'*b*
⟨*proof*⟩

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* $\Rightarrow_{CL}$ '*b*) = − *cblinfun-of-mat M*
⟨*proof*⟩

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* $\Rightarrow_{CL}$ '*b*) = *cblinfun-of-mat M* − *cblinfun-of-mat N*
⟨*proof*⟩

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* $\Rightarrow_{CL}$′*c*) o_{CL} ((*cblinfun-of-mat N*)::′*a* $\Rightarrow_{CL}$′*b*)
⟨*proof*⟩

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* $\Rightarrow_{CL}$ ′*a*) = (*cblinfun-of-mat M*)∗
⟨*proof*⟩

lemma *mat-of-cblinfun-compose*:
  mat-of-cblinfun (*F* o_{CL} *G*) = *mat-of-cblinfun F* ∗ *mat-of-cblinfun G*
  for *F*::′*b*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ ′*c*::{*basis-enum*,*complex-normed-vector*}
    and *G*::′*a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ ′*b*
  ⟨*proof*⟩

lemma *mat-of-cblinfun-scaleC*:
  mat-of-cblinfun ((*a*::*complex*) $*_C$ *F*) = *a* $\cdot_m$ (*mat-of-cblinfun F*)
  for *F* :: ′*a*::{*basis-enum*,*complex-normed-vector*} $\Rightarrow_{CL}$ ′*b*::{*basis-enum*,*complex-normed-vector*}
  ⟨*proof*⟩

lemma *mat-of-cblinfun-scaleR*:
  mat-of-cblinfun ((*a*::*real*) $*_R$ *F*) = (*complex-of-real a*) $\cdot_m$ (*mat-of-cblinfun F*)
  ⟨*proof*⟩

lemma *mat-of-cblinfun-adj*:
  mat-of-cblinfun (*F*∗) = *mat-adjoint* (*mat-of-cblinfun F*)
  for *F* :: ′*a*::*onb-enum* $\Rightarrow_{CL}$ ′*b*::*onb-enum*
  ⟨*proof*⟩

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*}
  ⟨*proof*⟩

lemma *mat-of-cblinfun-proj*:

fixes *a*::′*a*::*onb-enum*
defines *d* ≡ *length* (*canonical-basis* :: ′*a list*)
and *norm2* ≡ (*vec-of-basis-enum a*) ·*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*⟩

lemma *mat-of-cblinfun-ell2-component*:
fixes *a* :: ‹′*a*::*enum ell2* $\Rightarrow_{CL}$ ′*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*⟩

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*))›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*⟩

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*′)›
⟨*proof*⟩

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*⟩

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*⟩

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*)))
⟨*proof*⟩

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*⟩

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*⟩

unbundle *no-jnf-notation*
unbundle *no-cblinfun-notation*

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-notation*
unbundle *cblinfun-notation*

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
⟨*proof*⟩
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*
⟨*proof*⟩

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
⟨*proof*⟩
end

lemma *vec-of-ell2-carrier-vec*[*simp*]: ‹*vec-of-ell2 v* ∈ *carrier-vec CARD*(*'a*)› **for** *v* :: ‹*'a*::*enum ell2*›
⟨*proof*⟩

lemma *vec-of-ell2-zero*[*code*]:
— Code equation for computing the zero vector
vec-of-ell2 (*0*::*'a*::*enum ell2*) = *zero-vec* (*CARD*(*'a*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *norm-ell2-code* [*code*]:
— Code equation for the norm of a vector
norm ψ = *norm-vec* (*vec-of-ell2* ψ)
⟨*proof*⟩

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*]
⟨*proof*⟩

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*]
⟨*proof*⟩

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*)]
⟨*proof*⟩

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*]
⟨*proof*⟩

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*))

⟨*proof*⟩

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* $\Rightarrow_{CL}$ $'a$ *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*
⟨*proof*⟩

definition *butterfly-code* :: ‹$'a$ *ell2* ⇒ $'b$ *ell2* ⇒ $'b$ *ell2* $\Rightarrow_{CL}$ $'a$ *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*›
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *bot-as-span*[*code*]:
— Code equation for ⊥, the subspace containing everything. Top is represented as the span of the standard basis vectors.
(*bot*::*'a*::*onb-enum ccsubspace*) = *SPAN* []
⟨*proof*⟩

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*)
⟨*proof*⟩

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.)
⟨*proof*⟩

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 ⟨*proof*⟩
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*⟩

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*≤*B* ∧ *B*≤*A*)
⟨*proof*⟩

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* $\Rightarrow_{CL}$ '*b*::*onb-enum*
⟨*proof*⟩

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 $A *_S \top$ 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* $\Rightarrow_{CL}$ *'b::onb-enum*
shows *range-cblinfun-code A* = *SPAN* (*cols* (*mat-of-cblinfun A*))
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
 ⟨*proof*⟩

Code equation for *UNIV*. It is now implemented via type class *enum* (which provides a list of all values).

declare [[*code drop*: *UNIV*]]
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*
unbundle *no-jnf-notation*
unbundle *no-cblinfun-notation*

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-notation*

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.