Kraus Maps*

Dominique Unruh

October 13, 2025

Abstract

We formalize Kraus maps [1, Section 3], i.e., quantum channels of the form $\rho \mapsto \sum_x M_x \rho M_x^\dagger$ for suitable families of "Kraus operators" M_x. (In the finite-dimensional setting and the setting of separable Hilbert spaces, those are known to be equivalent to completely-positive maps, another common formalization of quantum channels.) Our results hold for arbitrary (i.e., not necessarily finite-dimensional or separable) Hilbert spaces.

Specifically, in theory `Kraus_Families`, we formalize the type (α, β, ξ) `kraus_family` of families of Kraus operators M_x (Kraus families for short), from trace-class operators on Hilbert space α to those on β, indexed by x of type ξ. This induces both a Kraus map $\rho \mapsto \sum_x M_x \rho M_x^\dagger$, as well as a quantum measurement with outcomes of type ξ.

We define and study various special Kraus families such as zero, identity, application of an operator, sum of two Kraus maps, sequential composition, infinite sum, random sampling, trace, tensor products, and complete measurements.

Furthermore, since working with explicit Kraus families can be cumbersome when the specific family of operators is not relevant, in theory `Kraus_Maps`, we define a Kraus map to be a function between two spaces that is of the form $\rho \mapsto \sum_x M_x \rho M_x^\dagger$ for some Kraus family, and restate our results in terms of such functions.

Contents

*Supported by the ERC consolidator grant CerQuS (819317), and the Estonian Cluster of Excellence "Foundations of the Universe" (TK202).

1 Miscelleanous missing theorems

theory *Misc-Kraus-Maps*
imports
Hilbert-Space-Tensor-Product.Hilbert-Space-Tensor-Product
Hilbert-Space-Tensor-Product.Von-Neumann-Algebras
begin

unbundle *cblinfun-syntax*

lemma *abs-summable-norm*:
assumes ‹*f abs-summable-on A*›
shows ‹(λ*x*. *norm* (*f x*)) *abs-summable-on A*›
using *assms* **by** *simp*

lemma *abs-summable-on-add*:
assumes ‹*f abs-summable-on A*› **and** ‹*g abs-summable-on A*›
shows ‹(λ*x*. *f x* + *g x*) *abs-summable-on A*›
proof −
from *assms* **have** ‹(λ*x*. *norm* (*f x*) + *norm* (*g x*)) *summable-on A*›
using *summable-on-add* **by** *blast*
then show *?thesis*
apply (*rule Infinite-Sum.abs-summable-on-comparison-test'*)
using *norm-triangle-ineq* **by** *blast*
qed

lemma *bdd-above-transform-mono-pos*:
assumes *bdd*: ‹*bdd-above* ((λ*x*. *g x*) ‘ *M*)›
assumes *gpos*: ‹⋀*x*. *x* ∈ *M* ⟹ *g x* ≥ *0*›
assumes *mono*: ‹*mono-on* (*Collect* ((≤) *0*)) *f*›
shows ‹*bdd-above* ((λ*x*. *f* (*g x*)) ‘ *M*)›
proof (*cases* ‹*M* = {}›)
case *True*
then show *?thesis*
by *simp*
next
case *False*
from *bdd* **obtain** *B* **where** *B*: ‹*g x* ≤ *B*› **if** ‹*x* ∈ *M*› **for** *x*
by (*meson bdd-above.unfold imageI*)
with *gpos False* **have** ‹*B* ≥ *0*›
using *dual-order.trans* **by** *blast*
have ‹*f* (*g x*) ≤ *f B*› **if** ‹*x* ∈ *M*› **for** *x*
using *mono - - B*
apply (*rule mono-onD*)
by (*auto intro*!: *gpos that* ‹*B* ≥ *0*›)
then show *?thesis*
by *fast*
qed

lemma *Ex-iffI*:
assumes ‹⋀*x*. *P x* ⟹ *Q* (*f x*)›
assumes ‹⋀*x*. *Q x* ⟹ *P* (*g x*)›
shows ‹*Ex P* ⟷ *Ex Q*›
using *assms*(*1*) *assms*(*2*) **by** *auto*

lemma *has-sum-Sigma'-banach*:
fixes *A* :: *'a set* **and** *B* :: *'a* ⇒ *'b set*
and *f* :: ‹*'a* ⇒ *'b* ⇒ *'c*::*banach*›
assumes ((λ(*x*,*y*). *f x y*) *has-sum S*) (*Sigma A B*)
shows ‹((λ*x*. *infsum* (*f x*) (*B x*)) *has-sum S*) *A*›
by (*metis* (*no-types*, *lifting*) *assms has-sum-cong has-sum-imp-summable has-sum-infsum infsumI infsum-Sigma'-banach summable-on-Sigma-banach*)

lemma *infsum-Sigma-topological-monoid*:
fixes *A* :: *'a set* **and** *B* :: *'a* ⇒ *'b set*
and *f* :: ‹*'a* × *'b* ⇒ *'c*::{*topological-comm-monoid-add*, *t3-space*}›
assumes *summableAB*: *f summable-on* (*Sigma A B*)
assumes *summableB*: ‹⋀*x*. *x*∈*A* ⟹ (λ*y*. *f* (*x*, *y*)) *summable-on* (*B x*)›
shows *infsum f* (*Sigma A B*) = (∑∞*x*∈*A*. ∑∞*y*∈*B x*. *f* (*x*, *y*))
proof −
have *1*: ‹(*f has-sum infsum f* (*Sigma A B*)) (*Sigma A B*)›
by (*simp add*: *assms*)
define *b* **where** ‹*b x* = (∑∞*y*∈*B x*. *f* (*x*, *y*))› **for** *x*
have *2*: ‹((λ*y*. *f* (*x*, *y*)) *has-sum b x*) (*B x*)› **if** ‹*x* ∈ *A*› **for** *x*

```
    using b-def assms(2) that by auto
  have 3: ‹(b has-sum (∑∞x∈A. b x)) A›
    using 1 2 by (metis has-sum-SigmaD infsumI)
  have 4: ‹(f has-sum (∑∞x∈A. b x)) (Sigma A B)›
    using 2 3 apply (rule has-sum-SigmaI)
    using assms by auto
  from 1 4 show ?thesis
    using b-def[abs-def] infsumI by blast
qed

lemma flip-eq-const: ‹(λy. y = x) = ((=) x)›
  by auto

lemma sgn-ket[simp]: ‹sgn (ket x) = ket x›
  by (simp add: sgn-div-norm)

lemma tensor-op-in-tensor-vn:
  assumes ‹a ∈ A› and ‹b ∈ B›
  shows ‹a ⊗o b ∈ A ⊗vN B›
proof −
  have ‹a ⊗o id-cblinfun ∈ A ⊗vN B›
   by (metis (no-types, lifting) Un-iff assms(1) double-commutant-grows' image-iff tensor-vn-def)
  moreover have ‹id-cblinfun ⊗o b ∈ A ⊗vN B›
    by (simp add: assms(2) double-commutant-grows' tensor-vn-def)
  ultimately have ‹(a ⊗o id-cblinfun) oCL (id-cblinfun ⊗o b) ∈ A ⊗vN B›
    using commutant-mult tensor-vn-def by blast
  then show ?thesis
    by (simp add: comp-tensor-op)
qed

lemma commutant-tensor-vn-subset:
  assumes ‹von-neumann-algebra A› and ‹von-neumann-algebra B›
  shows ‹commutant A ⊗vN commutant B ⊆ commutant (A ⊗vN B)›
proof −
  have 1: ‹a ⊗o id-cblinfun ∈ commutant (A ⊗vN B)› if ‹a ∈ commutant A› for a
    apply (simp add: tensor-vn-def)
    using that by (auto intro!: simp: commutant-def comp-tensor-op)
  have 2: ‹id-cblinfun ⊗o b ∈ commutant (A ⊗vN B)› if ‹b ∈ commutant B› for b
    apply (simp add: tensor-vn-def)
    using that by (auto intro!: simp: commutant-def comp-tensor-op)
  show ?thesis
    apply (subst tensor-vn-def)
    apply (rule double-commutant-in-vn-algI)
    using 1 2
    by (auto intro!: von-neumann-algebra-commutant von-neumann-algebra-tensor-vn assms)
qed

lemma commutant-span[simp]: ‹commutant (span X) = commutant X›
proof (rule order-antisym)
```

have ‹*commutant X* ⊆ *commutant* (*cspan X*)›
by (*simp add*: *commutant-cspan*)
also have ‹... ⊆ *commutant* (*span X*)›
by (*simp add*: *commutant-antimono span-subset-cspan*)
finally show ‹*commutant X* ⊆ *commutant* (*span X*)›
by −
show ‹*commutant* (*span X*) ⊆ *commutant X*›
by (*simp add*: *commutant-antimono span-superset*)
qed

lemma *explicit-cblinfun-exists-0*[*simp*]: ‹*explicit-cblinfun-exists* (λ- -. *0*)›
by (*auto intro*!: *explicit-cblinfun-exists-bounded*[**where** *B=0*] *simp*: *explicit-cblinfun-def*)

lemma *explicit-cblinfun-0*[*simp*]: ‹*explicit-cblinfun* (λ- -. *0*) = *0*›
by (*auto intro*!: *equal-ket Rep-ell2-inject*[*THEN iffD1*] *ext simp*: *Rep-ell2-explicit-cblinfun-ket zero-ell2.rep-eq*)

lemma *cnj-of-bool*[*simp*]: ‹*cnj* (*of-bool b*) = *of-bool b*›
by *simp*

lemma *has-sum-single*:
fixes *f* :: ‹- ⇒ -::{*comm-monoid-add*,*t2-space*}›
assumes ⋀*j*. *j* ≠ *i* ⟹ *j*∈*A* ⟹ *f j* = *0*
assumes ‹*s* = (*if i*∈*A then f i else 0*)›
shows *HAS-SUM f A s*
apply (*subst has-sum-cong-neutral*[**where** *T*=‹*A* ∩ {*i*}› **and** *g*=*f*])
using *assms* **by** *auto*

lemma *classical-operator-None*[*simp*]: ‹*classical-operator* (λ-. *None*) = *0*›
by (*auto intro*!: *equal-ket simp*: *classical-operator-ket inj-map-def classical-operator-exists-inj*)

lemma *has-sum-in-in-closedsubspace*:
assumes ‹*has-sum-in T f A l*›
assumes ‹⋀*x*. *x*∈*A* ⟹ *f x* ∈ *S*›
assumes ‹*closedin T S*›
assumes ‹*csubspace S*›
shows ‹*l* ∈ *S*›
proof −
from *assms*
have ‹*limitin T* (*sum f*) *l* (*finite-subsets-at-top A*)›
by (*simp add*: *has-sum-in-def*)
then have ‹*limitin T* (λ*F*. *if F*⊆*A then sum f F else 0*) *l* (*finite-subsets-at-top A*)›
apply (*rule limitin-transform-eventually*[*rotated*])
apply (*rule eventually-finite-subsets-at-top-weakI*)
by *simp*
then show ‹*l* ∈ *S*›
apply (*rule limitin-closedin*)
using *assms* **by** (*auto intro*!: *complex-vector.subspace-0 simp*: *complex-vector.subspace-sum*

subsetD)
qed

lemma *has-sum-coordinatewise*:
‹(*f has-sum s*) *A* ⟷ (∀ *i*. ((λ*x*. *f x i*) *has-sum s i*) *A*)›
proof −
have ‹(*f has-sum s*) *A* ⟷ ((λ*F*. (∑ *x*∈*F*. *f x*)) ⟶ *s*) (*finite-subsets-at-top A*)›
by (*simp add*: *has-sum-def*)
also have ‹... ⟷ (∀ *i*. ((λ*F*. (∑ *x*∈*F*. *f x*) *i*) ⟶ *s i*) (*finite-subsets-at-top A*))›
by (*simp add*: *tendsto-coordinatewise*)
also have ‹... ⟷ (∀ *i*. ((λ*F*. ∑ *x*∈*F*. *f x i*) ⟶ *s i*) (*finite-subsets-at-top A*))›
proof (*rewrite at* ‹∑ *x*∈*F*. *f x i*› *at* ‹λ*F*. ⌑› **in** ‹∀ *i*. ⌑› *to* ‹(∑ *x*∈*F*. *f x*) *i*› *DEADID.rel-mono-strong*)
fix *i*
show ‹(∑ *x*∈*F*. *f x i*) = (∑ *x*∈*F*. *f x*) *i*› **for** *F*
apply (*induction F rule*:*infinite-finite-induct*)
by *auto*
show ‹*P* = *P*› **for** *P* :: *bool*
by *simp*
qed
also have ‹... ⟷ (∀ *i*. ((λ*x*. *f x i*) *has-sum s i*) *A*)›
by (*simp add*: *has-sum-def*)
finally show *?thesis*
by −
qed

lemma *one-dim-butterfly*:
‹*butterfly g h* = (*one-dim-iso g* ∗ *cnj* (*one-dim-iso h*)) ∗$_C$ *1*›
apply (*rule cblinfun-eq-on-canonical-basis*)
apply *simp*
by (*smt* (*verit*, *del-insts*) *Groups.mult-ac*(*2*) *cblinfun.scaleC-left of-complex-def of-complex-inner-1 of-complex-inner-1' one-cblinfun-apply-one one-dim-apply-is-times-def one-dim-iso-def one-dim-scaleC-1*)

lemma *one-dim-tc-butterfly*:
fixes *g* :: ‹'*a* :: *one-dim*› **and** *h* :: ‹'*b* :: *one-dim*›
shows ‹*tc-butterfly g h* = (*one-dim-iso g* ∗ *cnj* (*one-dim-iso h*)) ∗$_C$ *1*›
proof −
have ‹*tc-butterfly g h* = *one-dim-iso* (*butterfly g h*)›
by (*metis* (*mono-tags*, *lifting*) *from-trace-class-one-dim-iso one-dim-iso-inj one-dim-iso-is-of-complex one-dim-iso-of-complex tc-butterfly.rep-eq*)
also have ‹... = (*one-dim-iso g* ∗ *cnj* (*one-dim-iso h*)) ∗$_C$ *1*›
by (*simp add*: *one-dim-butterfly*)
finally show *?thesis*
by −
qed

lemma *one-dim-iso-of-real*[*simp*]: ‹*one-dim-iso* (*of-real x*) = *of-real x*›
apply (*simp add*: *of-real-def*)
by (*simp add*: *scaleR-scaleC del*: *of-complex-of-real-eq*)

lemma *filter-insert-if*:
‹*Set.filter P* (*insert x M*) = (*if P x then insert x* (*Set.filter P M*) *else Set.filter P M*)›
by *auto*

lemma *filter-empty*[*simp*]: ‹*Set.filter P* {} = {}›
by *auto*

lemma *has-sum-in-cong-neutral*:
fixes *f g* :: ‹'*a* ⇒ '*b*::*comm-monoid-add*›
assumes ‹⋀*x*. *x*∈*T*−*S* ⟹ *g x* = *0*›
assumes ‹⋀*x*. *x*∈*S*−*T* ⟹ *f x* = *0*›
assumes ‹⋀*x*. *x*∈*S*∩*T* ⟹ *f x* = *g x*›
shows *has-sum-in X f S x* ⟷ *has-sum-in X g T x*
proof −
have ‹*eventually P* (*filtermap* (*sum f*) (*finite-subsets-at-top S*))
= *eventually P* (*filtermap* (*sum g*) (*finite-subsets-at-top T*))› **for** *P*
proof
assume ‹*eventually P* (*filtermap* (*sum f*) (*finite-subsets-at-top S*))›
then obtain *F0* **where** ‹*finite F0*› **and** ‹*F0* ⊆ *S*› **and** *F0-P*: ‹⋀*F*. *finite F* ⟹ *F* ⊆ *S* ⟹ *F* ⊇ *F0* ⟹ *P* (*sum f F*)›
by (*metis* (*no-types*, *lifting*) *eventually-filtermap eventually-finite-subsets-at-top*)
define *F0'* **where** ‹*F0'* = *F0* ∩ *T*›
have [*simp*]: ‹*finite F0'*› ‹*F0'* ⊆ *T*›
by (*simp-all add*: *F0'-def* ‹*finite F0*›)
have ‹*P* (*sum g F*)› **if** ‹*finite F*› ‹*F* ⊆ *T*› ‹*F* ⊇ *F0'*› **for** *F*
proof −
have ‹*P* (*sum f* ((*F*∩*S*) ∪ (*F0*∩*S*)))›
by (*intro F0-P*) (*use* ‹*F0* ⊆ *S*› ‹*finite F0*› *that* **in** *auto*)
also have ‹*sum f* ((*F*∩*S*) ∪ (*F0*∩*S*)) = *sum g F*›
by (*intro sum.mono-neutral-cong*) (*use that* ‹*finite F0*› *F0'-def assms* **in** *auto*)
finally show *?thesis* **.**
qed
with ‹*F0'* ⊆ *T*› ‹*finite F0'*› **show** ‹*eventually P* (*filtermap* (*sum g*) (*finite-subsets-at-top T*))›
by (*metis* (*no-types*, *lifting*) *eventually-filtermap eventually-finite-subsets-at-top*)
next
assume ‹*eventually P* (*filtermap* (*sum g*) (*finite-subsets-at-top T*))›
then obtain *F0* **where** ‹*finite F0*› **and** ‹*F0* ⊆ *T*› **and** *F0-P*: ‹⋀*F*. *finite F* ⟹ *F* ⊆ *T* ⟹ *F* ⊇ *F0* ⟹ *P* (*sum g F*)›
by (*metis* (*no-types*, *lifting*) *eventually-filtermap eventually-finite-subsets-at-top*)
define *F0'* **where** ‹*F0'* = *F0* ∩ *S*›
have [*simp*]: ‹*finite F0'*› ‹*F0'* ⊆ *S*›
by (*simp-all add*: *F0'-def* ‹*finite F0*›)
have ‹*P* (*sum f F*)› **if** ‹*finite F*› ‹*F* ⊆ *S*› ‹*F* ⊇ *F0'*› **for** *F*

proof −
have ‹*P* (*sum g* ((*F*∩*T*) ∪ (*F0*∩*T*)))›
by (*intro F0-P*) (*use* ‹*F0* ⊆ *T*› ‹*finite F0*› *that* **in** *auto*)
also have ‹*sum g* ((*F*∩*T*) ∪ (*F0*∩*T*)) = *sum f F*›
by (*intro sum.mono-neutral-cong*) (*use that* ‹*finite F0*› *F0'-def assms* **in** *auto*)
finally show *?thesis* **.**
qed
with ‹*F0'* ⊆ *S*› ‹*finite F0'*› **show** ‹*eventually P* (*filtermap* (*sum f*) (*finite-subsets-at-top S*))›
by (*metis* (*no-types*, *lifting*) *eventually-filtermap eventually-finite-subsets-at-top*)
qed

then have *tendsto-x*: *limitin X* (*sum f*) *x* (*finite-subsets-at-top S*) ⟷ *limitin X* (*sum g*) *x* (*finite-subsets-at-top T*) **for** *x*
by (*simp add*: *le-filter-def filterlim-def flip*: *filterlim-nhdsin-iff-limitin*)
then show *?thesis*
by (*simp add*: *has-sum-in-def*)
qed

lemma *infsum-in-cong-neutral*:
fixes *f g* :: ‹'*a* ⇒ '*b*::*comm-monoid-add*›
assumes ‹⋀*x*. *x*∈*T*−*S* ⟹ *g x* = *0*›
assumes ‹⋀*x*. *x*∈*S*−*T* ⟹ *f x* = *0*›
assumes ‹⋀*x*. *x*∈*S*∩*T* ⟹ *f x* = *g x*›
shows ‹*infsum-in X f S* = *infsum-in X g T*›
apply (*rule infsum-in-eqI'*)
apply (*rule has-sum-in-cong-neutral*)
using *assms* **by** *auto*

lemma *filter-image*: ‹*Set.filter P* (*f* ‘ *X*) = *f* ‘ (*Set.filter* (λ*x*. *P* (*f x*)) *X*)›
by *auto*

lemma *Sigma-image-left*: ‹(*SIGMA x*:*f*‘*A*. *B x*) = (λ(*x*,*y*). (*f x*, *y*)) ‘ (*SIGMA x*:*A*. *B* (*f x*))›
by (*auto intro*!: *image-eqI simp*: *split*: *prod.split*)

lemma *finite-subset-filter-image*:
assumes *finite B*
assumes ‹*B* ⊆ *Set.filter P* (*f* ‘ *A*)›
shows ∃ *C*⊆*A*. *finite C* ∧ *B* = *f* ‘ *C*
proof −
from *assms* **have** ‹*B* ⊆ *f* ‘ *A*›
by *auto*
then show *?thesis*
by (*simp add*: *assms*(*1*) *finite-subset-image*)
qed

definition ‹*card-le-1 M* ⟷ (∃ *x*. *M* ⊆ {*x*})›

lemma *card-le-1-empty*[*iff*]: ‹*card-le-1* {}›

by (*simp add*: *card-le-1-def*)

lemma *card-le-1-signleton*[*iff*]: ‹*card-le-1* {*x*}›
using *card-le-1-def* **by** *fastforce*

lemma *sgn-tensor-ell2*: ‹*sgn* (*h* ⊗$_s$ *k*) = *sgn h* ⊗$_s$ *sgn k*›
by (*simp add*: *sgn-div-norm norm-tensor-ell2 scaleR-scaleC tensor-ell2-scaleC1 tensor-ell2-scaleC2*)

lemma *is-ortho-set-tensor*:
assumes ‹*is-ortho-set B*›
assumes ‹*is-ortho-set C*›
shows ‹*is-ortho-set* ((λ(*x*, *y*). *x* ⊗$_s$ *y*) ‘ (*B* × *C*))›
proof (*intro is-ortho-set-def*[*THEN iffD2*] *conjI notI ballI impI*)
fix *bc bc'* :: ‹('*a* × '*b*) *ell2*›
assume ‹*bc* ∈ (λ(*x*, *y*). *x* ⊗$_s$ *y*) ‘ (*B* × *C*)›
then obtain *b c* **where** ‹*b* ∈ *B*› ‹*c* ∈ *C*› **and** *bc*: ‹*bc* = *b* ⊗$_s$ *c*›
by *fast*
assume ‹*bc'* ∈ (λ(*x*, *y*). *x* ⊗$_s$ *y*) ‘ (*B* × *C*)›
then obtain *b' c'* **where** ‹*b'* ∈ *B*› ‹*c'* ∈ *C*› **and** *bc'*: ‹*bc'* = *b'* ⊗$_s$ *c'*›
by *fast*
assume ‹*bc* ≠ *bc'*›
then consider (*neqb*) ‹*b* ≠ *b'*› | (*neqc*) ‹*c* ≠ *c'*›
using *bc bc'* **by** *blast*
then show ‹*is-orthogonal bc bc'*›
proof *cases*
case *neqb*
with ‹*b* ∈ *B*› ‹*b'* ∈ *B*› ‹*is-ortho-set B*›
have ‹*b* ·$_C$ *b'* = *0*›
by (*simp add*: *is-ortho-setD*)
then show *?thesis*
by (*simp add*: *bc bc'*)
next
case *neqc*
with ‹*c* ∈ *C*› ‹*c'* ∈ *C*› ‹*is-ortho-set C*›
have ‹*c* ·$_C$ *c'* = *0*›
by (*simp add*: *is-ortho-setD*)
then show *?thesis*
by (*simp add*: *bc bc'*)
qed
next
assume ‹*0* ∈ (λ(*x*, *y*). *x* ⊗$_s$ *y*) ‘ (*B* × *C*)›
then obtain *b c* **where** ‹*b* ∈ *B*› ‹*c* ∈ *C*› **and** ‹*b* ⊗$_s$ *c* = *0*›
by *auto*
then show *False*
by (*metis assms*(*1*,*2*) *is-ortho-set-def tensor-ell2-nonzero*)
qed

end

2 Kraus families

theory *Kraus-Families*
imports
Wlog.Wlog
Hilbert-Space-Tensor-Product.Partial-Trace

Misc-Kraus-Maps

abbrevs
$=kr = =_{kr}$ **and** $==kr = \equiv_{kr}$ **and** $*kr = *_{kr}$
begin

unbundle *cblinfun-syntax*

2.1 Kraus families

definition ‹*kraus-family* $\mathfrak{E} \longleftrightarrow$ *bdd-above* $((\lambda F.\ \sum (E,x)\in F.\ E* o_{CL}\ E)\ `\ \{F.\ \textit{finite}\ F \wedge F \subseteq \mathfrak{E}\}) \wedge 0 \notin \textit{fst}\ `\ \mathfrak{E}$›
for $\mathfrak{E}$ :: ‹((-::*chilbert-space* $\Rightarrow_{CL}$ -::*chilbert-space*) $\times$ -) *set*›

typedef (**overloaded**) ('*a*::*chilbert-space*, '*b*::*chilbert-space*, '*x*) *kraus-family* =
‹*Collect kraus-family* :: (('*a* $\Rightarrow_{CL}$ '*b*) $\times$ '*x*) *set set*›
by (*rule exI*[*of* - ‹{}›], *auto simp*: *kraus-family-def*)
setup-lifting *type-definition-kraus-family*

lemma *kraus-familyI*:
assumes ‹*bdd-above* $((\lambda F.\ \sum (E,x)\in F.\ E* o_{CL}\ E)\ `\ \{F.\ \textit{finite}\ F \wedge F \subseteq \mathfrak{E}\})$›
assumes ‹$0 \notin \textit{fst}\ `\ \mathfrak{E}$›
shows ‹*kraus-family* $\mathfrak{E}$›
by (*meson assms kraus-family-def*)

lift-definition *kf-apply* :: ‹('*a*::*chilbert-space*, '*b*::*chilbert-space*, '*x*) *kraus-family* $\Rightarrow$ ('*a*,'*a*) *trace-class* $\Rightarrow$ ('*b*,'*b*) *trace-class*› **is**
‹$\lambda\mathfrak{E}\ \varrho.\ (\sum_\infty E\in\mathfrak{E}.\ \textit{sandwich-tc}\ (\textit{fst}\ E)\ \varrho)$› .
notation *kf-apply* (**infixr** ‹$*_{kr}$› *70*)

lemma *kraus-family-if-finite*[*iff*]: ‹*kraus-family* $\mathfrak{E}$› **if** ‹*finite* $\mathfrak{E}$› **and** ‹$0 \notin \textit{fst}\ `\ \mathfrak{E}$›
proof −
define *B* **where** ‹$B = (\sum (E,x)\in\mathfrak{E}.\ E* o_{CL}\ E)$›
have ‹$(\sum (E,x)\in M.\ E* o_{CL}\ E) \leq B$› **if** ‹*finite M*› **and** ‹$M \subseteq \mathfrak{E}$› **for** *M*
unfolding *B-def*
using ‹*finite* $\mathfrak{E}$› ‹$M \subseteq \mathfrak{E}$› **apply** (*rule sum-mono2*)
by (*auto intro*!: *positive-cblinfun-squareI*)
with *that* **show** *?thesis*
by (*auto intro*!: *bdd-aboveI*[*of* - *B*] *simp*: *kraus-family-def*)
qed

lemma *kf-apply-scaleC*:
shows ‹*kf-apply* 𝔈 (*c* ∗$_C$ *x*) = *c* ∗$_C$ *kf-apply* 𝔈 *x*›
by (*simp add*: *kf-apply-def cblinfun.scaleC-right case-prod-unfold sandwich-tc-scaleC-right*
flip: *infsum-scaleC-right*)

lemma *kf-apply-abs-summable*:
assumes ‹*kraus-family* 𝔈›
shows ‹(λ(*E*,*x*). *sandwich-tc E* ϱ) *abs-summable-on* 𝔈›
proof −
wlog *ϱ-pos*: ‹ϱ ≥ *0*› **generalizing** ϱ
proof −
obtain *ϱ1 ϱ2 ϱ3 ϱ4* **where** *ϱ-decomp*: ‹ϱ = *ϱ1* − *ϱ2* + i ∗$_C$ *ϱ3* − i ∗$_C$ *ϱ4*›
and *pos*: ‹*ϱ1* ≥ *0*› ‹*ϱ2* ≥ *0*› ‹*ϱ3* ≥ *0*› ‹*ϱ4* ≥ *0*›
apply *atomize-elim* **using** *trace-class-decomp-4pos'*[*of* ϱ] **by** *blast*
have ‹*norm* (*sandwich-tc x* ϱ)
≤ *norm* (*sandwich-tc x ϱ1*)
+ *norm* (*sandwich-tc x ϱ2*)
+ *norm* (*sandwich-tc x ϱ3*)
+ *norm* (*sandwich-tc x ϱ4*)›
(**is** ‹- ≤ *?S x*›) **for** *x*
by (*auto simp add*: *ϱ-decomp sandwich-tc-plus sandwich-tc-minus sandwich-tc-scaleC-right*
scaleC-add-right scaleC-diff-right norm-mult
intro!: *norm-triangle-le norm-triangle-le-diff*)
then have ∗: ‹*norm* (*sandwich-tc* (*fst x*) ϱ) ≤ *norm* (*?S* (*fst x*))› **for** *x*
by *force*
show *?thesis*
unfolding *case-prod-unfold*
apply (*rule abs-summable-on-comparison-test*[*OF* - ∗])
apply (*intro Misc-Kraus-Maps.abs-summable-on-add abs-summable-norm abs-summable-on-scaleC-right*
pos)
using *hypothesis*
by (*simp-all add*: *case-prod-unfold pos*)
qed

have *aux*: ‹*trace-norm x* = *Re* (*trace x*)› **if** ‹*x* ≥ *0*› **and** ‹*trace-class x*› **for** *x*
by (*metis Re-complex-of-real that*(*1*) *trace-norm-pos*)
have *trace-EEϱ-pos*: ‹*trace* ((*E*∗ o_{CL} *E*) o_{CL} *from-trace-class* ϱ) ≥ *0*› **for** *E* :: ‹*'a* ⇒$_{CL}$ *'b*›
apply (*simp add*: *cblinfun-assoc-right trace-class-comp-right*
flip: *circularity-of-trace*)
by (*auto intro*!: *trace-pos sandwich-pos*
simp: *cblinfun-assoc-left from-trace-class-pos ϱ-pos*
simp flip: *sandwich-apply*)
have *trace-EEϱ-lin*: ‹*linear* (λ*M*. *Re* (*trace* (*M* o_{CL} *from-trace-class* ϱ)))› **for** *M*
apply (*rule linear-compose*[**where** *g*=*Re*, *unfolded o-def*])
by (*auto intro*!: *bounded-linear.linear bounded-clinear.bounded-linear*
bounded-clinear-trace-duality' bounded-linear-Re)
have *trace-EEϱ-mono*: ‹*mono-on* (*Collect* ((≤) *0*)) (λ*A*. *Re* (*trace* (*A* o_{CL} *from-trace-class*
ϱ)))› **for** *M*

```
    apply (intro mono-onI Re-mono)
    apply (subst diff-ge-0-iff-ge[symmetric])
    apply (subst trace-minus[symmetric])
    by (auto intro!: trace-class-comp-right trace-comp-pos
        simp: from-trace-class-pos ϱ-pos
        simp flip: cblinfun-compose-minus-left)

  from assms
  have ‹bdd-above ((λF. (∑ (E,x)∈F. E* o_CL E)) ` {F. finite F ∧ F ⊆ 𝔈})›
    by (simp add: kraus-family-def)
  then have ‹bdd-above ((λF. Re (trace ((∑ (E,x)∈F. E* o_CL E) o_CL from-trace-class ϱ))) `
{F. finite F ∧ F ⊆ 𝔈})›
    apply (rule bdd-above-transform-mono-pos)
    by (auto intro!: sum-nonneg positive-cblinfun-squareI[OF refl] trace-EEϱ-mono
        simp: case-prod-unfold)
  then have ‹bdd-above ((λF. ∑ (E,x)∈F. Re (trace ((E* o_CL E) o_CL from-trace-class ϱ))) `
{F. F ⊆ 𝔈 ∧ finite F})›
   apply (subst (asm) real-vector.linear-sum[where f=‹λM. Re (trace (M o_CL from-trace-class
ϱ))›])
    by (auto intro!: trace-EEϱ-lin simp: case-prod-unfold conj-commute)
  then have ‹(λ(E,-). Re (trace ((E* o_CL E) o_CL from-trace-class ϱ))) summable-on 𝔈›
    apply (rule nonneg-bdd-above-summable-on[rotated])
    using trace-EEϱ-pos
    by (auto simp: less-eq-complex-def)
 then have ‹(λ(E,-). Re (trace (from-trace-class (sandwich-tc E ϱ)))) summable-on 𝔈›
   by (simp add: from-trace-class-sandwich-tc sandwich-apply cblinfun-assoc-right trace-class-comp-right
       flip: circularity-of-trace)
  then have ‹(λ(E,-). trace-norm (from-trace-class (sandwich-tc E ϱ))) summable-on 𝔈›
    by (simp add: aux from-trace-class-pos ϱ-pos  sandwich-tc-pos)
  then show ‹(λ(E,x). sandwich-tc E ϱ) abs-summable-on 𝔈›
    by (simp add: norm-trace-class.rep-eq case-prod-unfold)
qed

lemma kf-apply-summable:
  shows ‹(λ(E,x). sandwich-tc E ϱ) summable-on (Rep-kraus-family 𝔈)›
  apply (rule abs-summable-summable)
  apply (rule kf-apply-abs-summable)
  using Rep-kraus-family by blast


lemma kf-apply-has-sum:
  shows ‹((λ(E,x). sandwich-tc E ϱ) has-sum kf-apply 𝔈 ϱ) (Rep-kraus-family 𝔈)›
  using kf-apply-summable Rep-kraus-family[of 𝔈]
  by (auto intro!: has-sum-infsum simp add: kf-apply-def kf-apply-summable case-prod-unfold)

lemma kf-apply-plus-right:
  shows ‹kf-apply 𝔈 (x + y) = kf-apply 𝔈 x + kf-apply 𝔈 y›
  using kf-apply-summable Rep-kraus-family[of 𝔈]
  by (auto intro!: infsum-add
```

simp add: *kf-apply-def sandwich-tc-plus scaleC-add-right case-prod-unfold*)

lemma *kf-apply-uminus-right*:
shows ‹*kf-apply* $\mathfrak{E}$ $(-x) = -$ *kf-apply* $\mathfrak{E}$ x›
using *kf-apply-summable Rep-kraus-family*[*of* $\mathfrak{E}$]
by (*auto intro*!: *infsum-uminus*
simp add: *kf-apply-def sandwich-tc-uminus-right scaleC-minus-right case-prod-unfold*)

lemma *kf-apply-minus-right*:
shows ‹*kf-apply* $\mathfrak{E}$ $(x - y) =$ *kf-apply* $\mathfrak{E}$ $x -$ *kf-apply* $\mathfrak{E}$ y›
by (*simp only*: *diff-conv-add-uminus kf-apply-plus-right kf-apply-uminus-right*)

lemma *kf-apply-pos*:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-apply* $\mathfrak{E}$ $\varrho \geq 0$›
by (*auto intro*!: *infsum-nonneg-traceclass scaleC-nonneg-nonneg of-nat-0-le-iff*
sandwich-tc-pos assms simp: *kf-apply-def*)

lemma *kf-apply-mono-right*:
assumes ‹$\varrho \geq \tau$›
shows ‹*kf-apply* $\mathfrak{E}$ $\varrho \geq$ *kf-apply* $\mathfrak{E}$ τ›
apply (*subst diff-ge-0-iff-ge*[*symmetric*])
apply (*subst kf-apply-minus-right*[*symmetric*])
apply (*rule kf-apply-pos*)
using *assms* **by** (*subst diff-ge-0-iff-ge*)

lemma *kf-apply-geq-sum*:
assumes ‹$\varrho \geq 0$› **and** ‹$M \subseteq$ *Rep-kraus-family* $\mathfrak{E}$›
shows ‹*kf-apply* $\mathfrak{E}$ $\varrho \geq (\sum (E,\text{-}) \in M.$ *sandwich-tc* E $\varrho)$›
proof (*cases* ‹*finite* M›)
case *True*
have $*$: ‹$(\lambda E.$ *sandwich-tc* (*fst* E) $\varrho)$ *summable-on* X› **if** ‹$X \subseteq$ *Rep-kraus-family* $\mathfrak{E}$› **for** X
apply (*rule summable-on-subset-banach*[**where** A=‹*Rep-kraus-family* $\mathfrak{E}$›])
apply (*rule kf-apply-summable*[*unfolded case-prod-unfold*])
using *assms that* **by** *blast*
show *?thesis*
apply (*subst infsum-finite*[*symmetric*])
using *assms*
by (*auto intro*!: *infsum-mono-neutral-traceclass* $*$ *scaleC-nonneg-nonneg of-nat-0-le-iff*
True sandwich-tc-pos
simp: *kf-apply-def case-prod-unfold*)
next
case *False*
with *assms* **show** *?thesis*
by (*simp add*: *kf-apply-pos*)
qed

lift-definition *kf-domain* :: ‹$('a$::*chilbert-space*,$'b$::*chilbert-space*,$'x)$ *kraus-family* $\Rightarrow$ $'x$ *set*› **is**

‹$\lambda\mathfrak{E}$. *snd* ' $\mathfrak{E}$›.

lemma *kf-apply-clinear*[*iff*]: ‹*clinear* (*kf-apply* $\mathfrak{E}$)›
by (*auto intro*!: *clinearI kf-apply-plus-right kf-apply-scaleC mult.commute*)

lemma *kf-apply-0-right*[*iff*]: ‹*kf-apply* $\mathfrak{E}$ *0* = *0*›
by (*metis ab-left-minus kf-apply-plus-right kf-apply-uminus-right*)

lift-definition *kf-operators* :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*x*) *kraus-family* ⇒ ('*a* ⇒$_{CL}$ '*b*) *set*› **is**
‹*image fst* :: ('*a* ⇒$_{CL}$ '*b* × '*x*) *set* ⇒ ('*a* ⇒$_{CL}$ '*b*) *set*›.

2.2 Bound and norm

lift-definition *kf-bound* :: ‹('*a*::*chilbert-space*, '*b*::*chilbert-space*, '*x*) *kraus-family* ⇒ ('*a* ⇒$_{CL}$ '*a*)› **is**
‹$\lambda\mathfrak{E}$. *infsum-in cweak-operator-topology* (λ(*E*,*x*). *E*∗ *o*$_{CL}$ *E*) $\mathfrak{E}$› .

lemma *kf-bound-def'*:
‹*kf-bound* $\mathfrak{E}$ = *Rep-cblinfun-wot* ($\sum_\infty$(*E*,*x*)∈*Rep-kraus-family* $\mathfrak{E}$. *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*))›
unfolding *kf-bound.rep-eq infsum-euclidean-eq*[*symmetric*]
apply *transfer'*
by *simp*

definition ‹*kf-norm* $\mathfrak{E}$ = *norm* (*kf-bound* $\mathfrak{E}$)›

lemma *kf-norm-sum-bdd*: ‹*bdd-above* ((λ*F*. *norm* ($\sum$(*E*,*x*)∈*F*. *E*∗ *o*$_{CL}$ *E*)) ' {*F*. *F* ⊆ *Rep-kraus-family* $\mathfrak{E}$ ∧ *finite F*})›
proof −
from *Rep-kraus-family*[*of* $\mathfrak{E}$]
obtain *B* **where** *B-bound*: ‹($\sum$(*E*,*x*)∈*F*. *E*∗ *o*$_{CL}$ *E*) ≤ *B*› **if** ‹*F* ⊆ *Rep-kraus-family* $\mathfrak{E}$› **and** ‹*finite F*› **for** *F*
by (*metis* (*mono-tags*, *lifting*) *bdd-above.unfold imageI kraus-family-def mem-Collect-eq*)
have ‹*norm* ($\sum$(*E*,*x*)∈*F*. *E*∗ *o*$_{CL}$ *E*) ≤ *norm B*› **if** ‹*F* ⊆ *Rep-kraus-family* $\mathfrak{E}$› **and** ‹*finite F*› **for** *F*
by (*metis* (*no-types*, *lifting*) *B-bound norm-cblinfun-mono positive-cblinfun-squareI split-def sum-nonneg that*(*1*) *that*(*2*))
then show ‹*bdd-above* ((λ*F*. *norm* ($\sum$(*E*,*x*)∈*F*. *E*∗ *o*$_{CL}$ *E*)) ' {*F*. *F* ⊆ *Rep-kraus-family* $\mathfrak{E}$ ∧ *finite F*})›
by (*metis* (*mono-tags*, *lifting*) *bdd-aboveI2 mem-Collect-eq*)
qed

lemma *kf-norm-geq0*[*iff*]:

shows ‹*kf-norm* $\mathfrak{E} \geq 0$›
proof (*cases* ‹*Rep-kraus-family* $\mathfrak{E} \neq \{\}$›)
case *True*
then obtain E **where** ‹$E \in$ *Rep-kraus-family* $\mathfrak{E}$› **by** *auto*
have ‹$0 \leq (\bigsqcup F{\in}\{F.\ F \subseteq$ *Rep-kraus-family* $\mathfrak{E} \wedge$ *finite* $F\}.$ *norm* $(\sum (E,x){\in}F.\ E{*}\ o_{CL}\ E))$›
apply (*rule cSUP-upper2*[**where** x=‹$\{E\}$›])
using *True* **by** (*simp-all add*: ‹$E \in$ *Rep-kraus-family* $\mathfrak{E}$› *kf-norm-sum-bdd*)
then show *?thesis*
by (*simp add*: *kf-norm-def True*)
next
case *False*
then show *?thesis*
by (*auto simp*: *kf-norm-def*)
qed

lemma *kf-bound-has-sum*:
shows ‹*has-sum-in cweak-operator-topology* $(\lambda(E,x).\ E{*}\ o_{CL}\ E)$ (*Rep-kraus-family* $\mathfrak{E}$) (*kf-bound* $\mathfrak{E}$)›
proof −
obtain B **where** B: ‹*finite* $F \Longrightarrow F \subseteq$ *Rep-kraus-family* $\mathfrak{E} \Longrightarrow (\sum (E,x){\in}F.\ E{*}\ o_{CL}\ E) \leq B$› **for** F
using *Rep-kraus-family*[*of* $\mathfrak{E}$]
by (*auto simp*: *kraus-family-def case-prod-unfold bdd-above-def*)
have ‹*summable-on-in cweak-operator-topology* $(\lambda(E,\ x).\ E{*}\ o_{CL}\ E)$ (*Rep-kraus-family* $\mathfrak{E}$)›
using B **by** (*auto intro*!: *summable-wot-boundedI positive-cblinfun-squareI simp*: *kraus-family-def*)
then show *?thesis*
by (*auto intro*!: *has-sum-in-infsum-in simp*: *kf-bound-def*)
qed

lemma *kraus-family-iff-summable*:
‹*kraus-family* $\mathfrak{E} \longleftrightarrow$ *summable-on-in cweak-operator-topology* $(\lambda(E,x).\ E{*}\ o_{CL}\ E)$ $\mathfrak{E} \wedge 0 \notin$ *fst* ‘ $\mathfrak{E}$›
proof (*intro iffI conjI*)
assume ‹*kraus-family* $\mathfrak{E}$›
have ‹*summable-on-in cweak-operator-topology* $(\lambda(E,x).\ E{*}\ o_{CL}\ E)$ (*Rep-kraus-family* (*Abs-kraus-family* $\mathfrak{E}$))›
using ‹*kraus-family* $\mathfrak{E}$› *kf-bound-has-sum summable-on-in-def* **by** *blast*
with ‹*kraus-family* $\mathfrak{E}$› **show** ‹*summable-on-in cweak-operator-topology* $(\lambda(E,x).\ E{*}\ o_{CL}\ E)$ $\mathfrak{E}$›
by (*simp add*: *Abs-kraus-family-inverse*)
from ‹*kraus-family* $\mathfrak{E}$› **show** ‹$0 \notin$ *fst* ‘ $\mathfrak{E}$›
using *kraus-family-def* **by** *blast*
next
assume ‹*summable-on-in cweak-operator-topology* $(\lambda(E,x).\ E{*}\ o_{CL}\ E)$ $\mathfrak{E} \wedge 0 \notin$ *fst* ‘ $\mathfrak{E}$›
then show ‹*kraus-family* $\mathfrak{E}$›
by (*auto intro*!: *summable-wot-bdd-above*[**where** X=$\mathfrak{E}$] *positive-cblinfun-squareI simp*: *kraus-family-def*)
qed

lemma *kraus-family-iff-summable′*:
‹*kraus-family* $\mathfrak{E} \longleftrightarrow (\lambda(E,x).$ *Abs-cblinfun-wot* $(E{*}\ o_{CL}\ E))$ *summable-on* $\mathfrak{E} \wedge 0 \notin$ *fst* ‘ $\mathfrak{E}$›

apply (*transfer′ fixing*: $\mathfrak{E}$)
by (*simp add*: *kraus-family-iff-summable*)

lemma *kf-bound-summable*:
shows ‹*summable-on-in cweak-operator-topology* (λ(*E*,*x*). *E*∗ o_{CL} *E*) (*Rep-kraus-family* $\mathfrak{E}$)›
using *kf-bound-has-sum summable-on-in-def* **by** *blast*

lemma *kf-bound-has-sum′*:
shows ‹((λ(*E*,*x*). *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*)) *has-sum Abs-cblinfun-wot* (*kf-bound* $\mathfrak{E}$)) (*Rep-kraus-family* $\mathfrak{E}$)›
using *kf-bound-has-sum*[*of* $\mathfrak{E}$]
apply *transfer′*
by *auto*

lemma *kf-bound-summable′*:
‹((λ(*E*,*x*). *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*)) *summable-on Rep-kraus-family* $\mathfrak{E}$)›
using *has-sum-imp-summable kf-bound-has-sum′* **by** *blast*

lemma *kf-bound-is-Sup*:
shows ‹*is-Sup* ((λ*F*. $\sum$(*E*,*x*)∈*F*. *E*∗ o_{CL} *E*) ‘ {*F*. *finite F* ∧ *F* ⊆ *Rep-kraus-family* $\mathfrak{E}$}) (*kf-bound* $\mathfrak{E}$)›
proof −
from *Rep-kraus-family*[*of* $\mathfrak{E}$]
obtain *B* **where** ‹*finite F* ⟹ *F* ⊆ *Rep-kraus-family* $\mathfrak{E}$ ⟹ ($\sum$(*E*,*x*)∈*F*. *E*∗ o_{CL} *E*) ≤ *B*› **for** *F*
by (*metis* (*mono-tags*, *lifting*) *bdd-above.unfold imageI kraus-family-def mem-Collect-eq*)
then have ‹*is-Sup* ((λ*F*. $\sum$(*E*,*x*)∈*F*. *E*∗ o_{CL} *E*) ‘ {*F*. *finite F* ∧ *F* ⊆ *Rep-kraus-family* $\mathfrak{E}$}) (*infsum-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (*Rep-kraus-family* $\mathfrak{E}$))›
apply (*rule infsum-wot-is-Sup*[*OF summable-wot-boundedI*[**where** *B*=*B*]])
by (*auto intro*!: *summable-wot-boundedI positive-cblinfun-squareI simp*: *case-prod-beta*)
then show *?thesis*
by (*auto intro*!: *simp*: *kf-bound-def*)
qed

lemma *kf-bound-leqI*:
assumes ‹⋀*F*. *finite F* ⟹ *F* ⊆ *Rep-kraus-family* $\mathfrak{E}$ ⟹ ($\sum$(*E*,*x*)∈*F*. *E*∗ o_{CL} *E*) ≤ *B*›
shows ‹*kf-bound* $\mathfrak{E}$ ≤ *B*›
using *kf-bound-is-Sup*[*of* $\mathfrak{E}$]
by (*simp add*: *assms is-Sup-def*)

lemma *kf-bound-pos*[*iff*]: ‹*kf-bound* $\mathfrak{E}$ ≥ *0*›
using *kf-bound-is-Sup*[*of* $\mathfrak{E}$]
by (*metis* (*no-types*, *lifting*) *empty-subsetI finite.emptyI image-iff is-Sup-def mem-Collect-eq sum.empty*)

lemma *not-not-singleton-kf-norm-0*:
fixes $\mathfrak{E}$ :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*x*) *kraus-family*›

assumes ‹¬ *class.not-singleton TYPE('a)*›
shows ‹*kf-norm* $\mathfrak{E} = 0$›
by (*simp add*: *not-not-singleton-cblinfun-zero*[*OF assms*] *kf-norm-def*)

lemma *kf-norm-sum-leqI*:
assumes ‹$\bigwedge F$. *finite* $F \Longrightarrow F \subseteq$ *Rep-kraus-family* $\mathfrak{E} \Longrightarrow$ *norm* $(\sum (E,x)\in F.\ E* \ o_{CL}\ E) \leq B$›
shows ‹*kf-norm* $\mathfrak{E} \leq B$›
proof −
have *bpos*: ‹$B \geq 0$›
using *assms*[*of* ‹{}›] **by** *auto*
wlog *not-singleton*: ‹*class.not-singleton TYPE('a)*› **keeping** *bpos*
using *not-not-singleton-kf-norm-0*[*OF negation*, *of* $\mathfrak{E}$]
by (*simp add*: ‹$B \geq 0$›)
have [*simp*]: ‹*norm* (*id-cblinfun* :: $'a \Rightarrow_{CL} 'a$) = 1›
apply (*rule norm-cblinfun-id*[*internalize-sort' 'a*])
apply (*rule complex-normed-vector-axioms*)
by (*rule not-singleton*)
have $*$: ‹*selfadjoint* $(\sum (E,x)\in F.\ E* \ o_{CL}\ E)$› **for** F :: ‹$('a \Rightarrow_{CL} 'b \times 'c)$ *set*›
by (*auto intro*!: *pos-imp-selfadjoint sum-nonneg intro*: *positive-cblinfun-squareI*)
from *assms*
have ‹$\bigwedge F$. *finite* $F \Longrightarrow F \subseteq$ *Rep-kraus-family* $\mathfrak{E} \Longrightarrow (\sum (E,x)\in F.\ E* \ o_{CL}\ E) \leq B *_R$ *id-cblinfun*›
apply (*rule less-eq-scaled-id-norm*)
by (*auto intro*!: $*$)
then have ‹*kf-bound* $\mathfrak{E} \leq B *_R$ *id-cblinfun*›
using *kf-bound-leqI* **by** *blast*
then have ‹*norm* (*kf-bound* $\mathfrak{E}$) $\leq$ *norm* $(B *_R$ (*id-cblinfun* :: $'a \Rightarrow_{CL} 'a)$)›
apply (*rule norm-cblinfun-mono*[*rotated*])
by *simp*
then show *?thesis*
using *bpos* **by** (*simp add*: *kf-norm-def*)
qed

lemma *kf-bound-geq-sum*:
assumes ‹$M \subseteq$ *Rep-kraus-family* $\mathfrak{E}$›
shows ‹$(\sum (E,\text{-})\in M.\ E* \ o_{CL}\ E) \leq$ *kf-bound* $\mathfrak{E}$›
proof (*cases* ‹*finite M*›)
case *True*
then show *?thesis*
using *kf-bound-is-Sup*[*of* $\mathfrak{E}$]
apply (*simp add*: *is-Sup-def case-prod-beta*)
using *assms* **by** *blast*
next
case *False*
then show *?thesis*
by *simp*
qed

lemma *kf-norm-geq-norm-sum*:
assumes ‹$M \subseteq$ *Rep-kraus-family* $\mathfrak{E}$›
shows ‹*norm* ($\sum$ $(E,\text{-})\in M$. $E*$ o_{CL} E) $\leq$ *kf-norm* $\mathfrak{E}$›
using *kf-bound-geq-sum assms*
by (*auto intro*!: *norm-cblinfun-mono sum-nonneg*
intro: *positive-cblinfun-squareI*
simp add: *kf-norm-def case-prod-beta*)

lemma *kf-bound-finite*: ‹*kf-bound* $\mathfrak{E}$ = ($\sum$ $(E,x)\in$*Rep-kraus-family* $\mathfrak{E}$. $E*$ o_{CL} E)› **if** ‹*finite* (*Rep-kraus-family* $\mathfrak{E}$)›
by (*auto intro*!: *kraus-family-if-finite simp*: *kf-bound-def that infsum-in-finite*)

lemma *kf-norm-finite*: ‹*kf-norm* $\mathfrak{E}$ = *norm* ($\sum$ $(E,x)\in$*Rep-kraus-family* $\mathfrak{E}$. $E*$ o_{CL} E)›
if ‹*finite* (*Rep-kraus-family* $\mathfrak{E}$)›
by (*simp add*: *kf-norm-def kf-bound-finite that*)

lemma *kf-apply-bounded-pos*:
assumes ‹$\varrho \geq 0$›
shows ‹*norm* (*kf-apply* $\mathfrak{E}$ ϱ) $\leq$ *kf-norm* $\mathfrak{E}$ $*$ *norm* ϱ›
proof −
have ‹*norm* (*kf-apply* $\mathfrak{E}$ ϱ) = *Re* (*trace-tc* ($\sum_\infty$ $(E,\text{-})\in$*Rep-kraus-family* $\mathfrak{E}$. *sandwich-tc* E ϱ))›
apply (*subst Re-complex-of-real*[*symmetric*])
apply (*subst norm-tc-pos*)
using ‹$\varrho \geq 0$› **apply** (*rule kf-apply-pos*)
by (*simp add*: *kf-apply-def case-prod-unfold*)
also have ‹... = ($\sum_\infty$ $(E,\text{-})\in$*Rep-kraus-family* $\mathfrak{E}$. *Re* (*trace-tc* (*sandwich-tc* E ϱ)))›
using *kf-apply-summable*[*of* - $\mathfrak{E}$]
by (*simp-all flip*: *infsum-bounded-linear*[*of* ‹λx. *Re* (*trace-tc* x)›]
add: *case-prod-unfold bounded-linear-compose*[*of Re trace-tc*] *bounded-linear-Re*
o-def bounded-clinear.bounded-linear)
also have ‹... $\leq$ *kf-norm* $\mathfrak{E}$ $*$ *norm* ϱ›
proof (*rule infsum-le-finite-sums*)
show ‹($\lambda(E,\text{-})$. *Re* (*trace-tc* (*sandwich-tc* E ϱ))) *summable-on* (*Rep-kraus-family* $\mathfrak{E}$)›
unfolding *case-prod-beta*
apply (*rule summable-on-bounded-linear*[*unfolded o-def*, **where** h=‹λx. *Re* (*trace-tc* x)›])
using *kf-apply-summable*[*of* - $\mathfrak{E}$]
by (*simp-all flip*: *infsum-bounded-linear*[*of* ‹λx. *Re* (*trace-tc* x)›]
add: *bounded-linear-compose*[*of Re trace-tc*] *bounded-linear-Re bounded-clinear.bounded-linear*

o-def trace-tc-scaleC assms kf-apply-def case-prod-unfold)
fix M :: ‹(('a ⇒$_{CL}$ 'b) × 'c) *set*› **assume** ‹*finite* M› **and** ‹$M \subseteq$ *Rep-kraus-family* $\mathfrak{E}$›
have ‹($\sum$ $(E,\text{-})\in M$. *Re* (*trace-tc* (*sandwich-tc* E ϱ)))
= ($\sum$ $(E,\text{-})\in M$. *Re* (*trace* (E o_{CL} *from-trace-class* ϱ o_{CL} $E*$)))›
by (*simp add*: *trace-tc.rep-eq from-trace-class-sandwich-tc sandwich-apply scaleC-trace-class.rep-eq trace-scaleC*)
also have ‹... = ($\sum$ $(E,\text{-})\in M$. *Re* (*trace* ($E*$ o_{CL} E o_{CL} *from-trace-class* ϱ)))›
apply (*subst circularity-of-trace*)

```
    by (auto intro!: trace-class-comp-right simp: cblinfun-compose-assoc)
  also have ‹... = Re (trace ((∑(E,-)∈M. E* o_CL E) o_CL from-trace-class ϱ))›
  by (simp only: trace-class-scaleC trace-class-comp-right trace-class-from-trace-class case-prod-unfold
        flip: Re-sum trace-scaleC trace-sum cblinfun.scaleC-left cblinfun-compose-scaleC-left
cblinfun-compose-sum-left)
  also have ‹... ≤ cmod (trace ((∑(E,-)∈M. E* o_CL E) o_CL from-trace-class ϱ))›
    by (rule complex-Re-le-cmod)
  also have ‹... ≤ norm (∑(E,-)∈M. E* o_CL E) * trace-norm (from-trace-class ϱ)›
    apply (rule cmod-trace-times)
    by simp
  also have ‹... ≤ kf-norm 𝔈 * norm ϱ›
    apply (simp add: flip: norm-trace-class.rep-eq)
    apply (rule mult-right-mono)
    apply (rule kf-norm-geq-norm-sum)
    using assms ‹M ⊆ Rep-kraus-family 𝔈› by auto
  finally show ‹(∑(E,-)∈M. Re (trace-tc (sandwich-tc E ϱ))) ≤ kf-norm 𝔈 * norm ϱ›
    by −
 qed
 finally show ?thesis
  by −
qed

lemma kf-apply-bounded:
 — We suspect the factor 4 is not needed here but don't know how to prove that.
 ‹norm (kf-apply 𝔈 ϱ) ≤ 4 * kf-norm 𝔈 * norm ϱ›
proof −
 have aux: ‹4 * x = x + x + x + x› for x :: real
  by auto
 obtain ϱ1 ϱ2 ϱ3 ϱ4 where ϱ-decomp: ‹ϱ = ϱ1 − ϱ2 + i *_C ϱ3 − i *_C ϱ4›
  and pos: ‹ϱ1 ≥ 0› ‹ϱ2 ≥ 0› ‹ϱ3 ≥ 0› ‹ϱ4 ≥ 0›
  and norm: ‹norm ϱ1 ≤ norm ϱ› ‹norm ϱ2 ≤ norm ϱ› ‹norm ϱ3 ≤ norm ϱ› ‹norm ϱ4 ≤
norm ϱ›
  apply atomize-elim using trace-class-decomp-4pos'[of ϱ] by blast
 have ‹norm (kf-apply 𝔈 ϱ) ≤
       norm (kf-apply 𝔈 ϱ1) +
       norm (kf-apply 𝔈 ϱ2) +
       norm (kf-apply 𝔈 ϱ3) +
       norm (kf-apply 𝔈 ϱ4)›
  by (auto intro!: norm-triangle-le norm-triangle-le-diff
      simp add: ϱ-decomp kf-apply-plus-right kf-apply-minus-right
      kf-apply-scaleC)
 also have ‹... ≤
      kf-norm 𝔈 * norm ϱ1
      + kf-norm 𝔈 * norm ϱ2
      + kf-norm 𝔈 * norm ϱ3
      + kf-norm 𝔈 * norm ϱ4›
  by (auto intro!: add-mono simp add: pos kf-apply-bounded-pos)
 also have ‹... = kf-norm 𝔈 * (norm ϱ1 + norm ϱ2 + norm ϱ3 + norm ϱ4)›
  by argo
```

also have ‹... ≤ *kf-norm* 𝔈 ∗ (*norm* ϱ + *norm* ϱ + *norm* ϱ + *norm* ϱ)›
by (*auto intro*!: *mult-left-mono add-mono kf-norm-geq0*
simp only: *aux norm*)
also have ‹... = *4* ∗ *kf-norm* 𝔈 ∗ *norm* ϱ›
by *argo*
finally show *?thesis*
by −
qed

lemma *kf-apply-bounded-clinear*[*bounded-clinear*]:
shows ‹*bounded-clinear* (*kf-apply* 𝔈)›
apply (*rule bounded-clinearI*[**where** *K*=‹*4* ∗ *kf-norm* 𝔈›])
apply (*auto intro*!: *kf-apply-plus-right kf-apply-scaleC mult.commute*)[*2*]
using *kf-apply-bounded*
by (*simp add*: *mult.commute*)

lemma *kf-bound-from-map*: ‹ψ ·$_C$ *kf-bound* 𝔈 φ = *trace-tc* (𝔈 ∗$_{kr}$ *tc-butterfly* φ ψ)›
proof −
have ‹*has-sum-in cweak-operator-topology* (λ(*E*,*x*). *E*∗ *o*$_{CL}$ *E*) (*Rep-kraus-family* 𝔈) (*kf-bound* 𝔈)›
by (*simp add*: *kf-bound-has-sum*)
then have ‹((λ(*E*, *x*). ψ ·$_C$ (*E*∗ *o*$_{CL}$ *E*) φ) *has-sum* ψ ·$_C$ *kf-bound* 𝔈 φ) (*Rep-kraus-family* 𝔈)›
by (*auto intro*!: *simp*: *has-sum-in-cweak-operator-topology-pointwise case-prod-unfold*)
moreover have ‹((λ(*E*, *x*). ψ ·$_C$ (*E*∗ *o*$_{CL}$ *E*) φ) *has-sum trace-tc* (*kf-apply* 𝔈 (*tc-butterfly* φ ψ))) (*Rep-kraus-family* 𝔈)›
proof −
have ∗: ‹*trace-tc* (*sandwich-tc E* (*tc-butterfly* φ ψ)) = ψ ·$_C$ (*E*∗ *o*$_{CL}$ *E*) φ› **for** *E* :: ‹'*a* ⇒$_{CL}$ '*b*›
by (*auto intro*!: *simp*: *trace-tc.rep-eq from-trace-class-sandwich-tc*
sandwich-apply tc-butterfly.rep-eq circularity-of-trace[*symmetric*, *of* - *E*]
trace-class-comp-left cblinfun-compose-assoc trace-butterfly-comp)
from *kf-apply-has-sum Rep-kraus-family*[*of* 𝔈]
have ‹((λ(*E*,*x*). *sandwich-tc E* (*tc-butterfly* φ ψ)) *has-sum kf-apply* 𝔈 (*tc-butterfly* φ ψ)) (*Rep-kraus-family* 𝔈)›
by *blast*
then have ‹((λ(*E*,*x*). *trace-tc* (*sandwich-tc E* (*tc-butterfly* φ ψ))) *has-sum trace-tc* (*kf-apply* 𝔈 (*tc-butterfly* φ ψ))) (*Rep-kraus-family* 𝔈)›
unfolding *case-prod-unfold*
apply (*rule has-sum-bounded-linear*[*rotated*, *unfolded o-def*])
by (*simp add*: *bounded-clinear.bounded-linear*)
then
show *?thesis*
by (*simp add*: ∗)
qed
ultimately show *?thesis*
using *has-sum-unique* **by** *blast*
qed

lemma *trace-from-kf-bound*: ‹*trace-tc* (𝔈 $*_{kr}$ *ϱ*) = *trace-tc* (*compose-tcr* (*kf-bound* 𝔈) *ϱ*)›
proof −
have ‹*separating-set bounded-clinear* (*Collect rank1-tc*)›
apply (*rule separating-set-bounded-clinear-dense*)
by *simp*
moreover have ‹*bounded-clinear* (*λa. trace-tc* (𝔈 $*_{kr}$ *a*))›
by (*intro bounded-linear-intros*)
moreover have ‹*bounded-clinear* (*λa. trace-tc* (*compose-tcr* (*kf-bound* 𝔈) *a*))›
by (*intro bounded-linear-intros*)
moreover have ‹*trace-tc* (𝔈 $*_{kr}$ *t*) = *trace-tc* (*compose-tcr* (*kf-bound* 𝔈) *t*)› **if** ‹*t* ∈ *Collect rank1-tc*› **for** *t*
proof −
from *that* **obtain** *x y* **where** *t*: ‹*t* = *tc-butterfly x y*›
apply (*transfer′ fixing*: *x y*)
by (*auto simp*: *rank1-iff-butterfly*)
then have ‹*trace-tc* (𝔈 $*_{kr}$ *t*) = *y* $\cdot_C$ *kf-bound* 𝔈 *x*›
by (*simp add*: *kf-bound-from-map*)
also have ‹... = *trace-tc* (*compose-tcr* (*kf-bound* 𝔈) (*tc-butterfly x y*))›
apply (*transfer′ fixing*: *x y* 𝔈)
by (*simp add*: *trace-butterfly-comp′*)
also have ‹... = *trace-tc* (*compose-tcr* (*kf-bound* 𝔈) *t*)›
by (*simp add*: *t*)
finally show *?thesis*
by −
qed
ultimately show *?thesis*
by (*rule eq-from-separatingI*[**where** *P*=*bounded-clinear* **and** *S*=‹*Collect rank1-tc*›, *THEN fun-cong*])
qed

lemma *kf-bound-selfadjoint*[*iff*]: ‹*selfadjoint* (*kf-bound* 𝔈)›
by (*simp add*: *positive-selfadjointI*)

lemma *kf-bound-leq-kf-norm-id*:
shows ‹*kf-bound* 𝔈 ≤ *kf-norm* 𝔈 $*_R$ *id-cblinfun*›
by (*auto intro*!: *less-eq-scaled-id-norm simp*: *kf-norm-def*)

2.3 Basic Kraus families

lemma *kf-emptyset*[*iff*]: ‹*kraus-family* {}›
by (*auto simp*: *kraus-family-def*)

instantiation *kraus-family* :: (*chilbert-space*, *chilbert-space*, *type*) ‹*zero*› **begin**
lift-definition *zero-kraus-family* :: ‹('*a*,'*b*,'*x*) *kraus-family*› **is** ‹{}›
by *simp*
instance..
end

lemma *kf-apply-0*[*simp*]: ‹*kf-apply 0 = 0*›
by (*auto simp*: *kf-apply-def zero-kraus-family.rep-eq*)

lemma *kf-bound-0*[*simp*]: ‹*kf-bound 0 = 0*›
by (*metis* (*mono-tags*, *lifting*) *finite.intros*(*1*) *kf-bound.rep-eq kf-bound-finite sum-clauses*(*1*) *zero-kraus-family.rep-eq*)

lemma *kf-norm-0*[*simp*]: ‹*kf-norm 0 = 0*›
by (*simp add*: *kf-norm-def zero-kraus-family.rep-eq*)

lift-definition *kf-of-op* :: ‹('*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space*) ⇒ ('*a*, '*b*, *unit*) *kraus-family*›
is
‹λ*E*::'*a*⇒$_{CL}$'*b*. *if E = 0 then* {} *else* {(*E*, ())}›
by (*auto intro*: *kraus-family-if-finite*)

lemma *kf-of-op0*[*simp*]: ‹*kf-of-op 0 = 0*›
apply *transfer'*
by *simp*

lemma *kf-of-op-norm*[*simp*]: ‹*kf-norm* (*kf-of-op E*) = (*norm E*)^2›
by (*simp add*: *kf-of-op.rep-eq kf-norm-finite*)

lemma *kf-operators-in-kf-of-op*[*simp*]: ‹*kf-operators* (*kf-of-op U*) = (*if U = 0 then* {} *else* {*U*})›
apply (*transfer' fixing*: *U*)
by *simp*

lemma *kf-domain-of-op*[*simp*]: ‹*kf-domain* (*kf-of-op A*) = {()}› **if** ‹*A* ≠ *0*›
apply (*transfer' fixing*: *A*)
using *that* **by** *auto*

definition ‹*kf-id* = *kf-of-op id-cblinfun*›

lemma *kf-domain-id*[*simp*]: ‹*kf-domain* (*kf-id* :: ('*a*::{*chilbert-space*,*not-singleton*},-,-) *kraus-family*) = {()}›
by (*simp add*: *kf-id-def*)

lemma *kf-of-op-id*[*simp*]: ‹*kf-of-op id-cblinfun* = *kf-id*›
by (*simp add*: *kf-id-def*)

lemma *kf-norm-id-leq1*: ‹*kf-norm kf-id* ≤ *1*›
apply (*simp add*: *kf-id-def del*: *kf-of-op-id*)
apply *transfer*
by (*auto intro*!: *power-le-one onorm-pos-le onorm-id-le*)

lemma *kf-norm-id-eq1*[*simp*]: ‹*kf-norm* (*kf-id* :: ('*a* :: {*chilbert-space*, *not-singleton*},'*a*,*unit*) *kraus-family*) = *1*›
by (*auto intro*!: *antisym kf-norm-id-leq1 simp*: *kf-id-def simp del*: *kf-of-op-id*)

lemma *kf-operators-in-kf-id*[*simp*]: ‹*kf-operators kf-id* = (*if* (*id-cblinfun*::'*a*::*chilbert-space*⇒$_{CL}$-)=*0*

```
  then {} else {id-cblinfun::'a⇒CL-})›
  by (simp add: kf-id-def del: kf-of-op-id)


instantiation kraus-family :: (chilbert-space, chilbert-space, type) scaleR begin
lift-definition scaleR-kraus-family :: ‹real ⇒ ('a::chilbert-space,'b::chilbert-space,'x) kraus-family
⇒ ('a,'b,'x) kraus-family› is
  ‹λr 𝔈. if r ≤ 0 then {} else (λ(E,x). (sqrt r *R E, x)) ‘ 𝔈›
proof (rename-tac r 𝔈)
  fix r and 𝔈 :: ‹('a ⇒CL 'b × 'x) set›
  assume asm: ‹𝔈 ∈ Collect kraus-family›
  define scaled where ‹scaled = (λ(E, y). (sqrt r *R E, y)) ‘ 𝔈›
  show ‹(if r ≤ 0 then {} else scaled) ∈ Collect kraus-family›
  proof (cases ‹r > 0›)
    case True
    obtain B where B: ‹(∑∞(E,x)∈M. E* oCL E) ≤ B› if ‹M ⊆ 𝔈› and ‹finite M› for M
      using asm
      by (auto intro!: simp: kraus-family-def bdd-above-def)
    have ‹(∑∞(E,x)∈M. E* oCL E) ≤ r *R B› if Mr𝔈: ‹M ⊆ scaled› and ‹finite M› for M
    proof −
      define M' where ‹M' = (λ(E,x). (E /R sqrt r, x)) ‘ M›
      then have ‹finite M'›
        using that(2) by blast
      moreover have ‹M' ⊆ 𝔈›
        using Mr𝔈 True by (auto intro!: simp: M'-def scaled-def)
      ultimately have 1: ‹(∑∞(E,x)∈M'. E* oCL E) ≤ B›
        using B by auto
      have 2: ‹(∑∞(E,x)∈M. E* oCL E) = r *R (∑∞(E,x)∈M'. E* oCL E)›
        apply (simp add: M'-def case-prod-unfold)
        apply (subst infsum-reindex)
        using True
        by (auto intro!: inj-onI simp: o-def infsum-scaleR-right
            simp flip: inverse-mult-distrib)
      show ?thesis
        using 1 2 True scaleR-le-cancel-left-pos by auto
    qed
    moreover have ‹0 ∉ fst ‘ scaled› if ‹r ≠ 0›
      using asm that
      by (auto intro!: simp: scaled-def kraus-family-def)
    ultimately show ?thesis
      by (auto intro!: simp: kraus-family-def bdd-above-def)
  next
    case False
    then show ?thesis
      by (simp add: scaled-def)
  qed
qed
instance..
end
```

lemma *kf-scale-apply*:
assumes ‹$r \geq 0$›
shows ‹*kf-apply* $(r *_R \mathfrak{E})\ \varrho = r *_R$ *kf-apply* $\mathfrak{E}\ \varrho$›
proof (*cases* ‹$r > 0$›)
case *True*
then show *?thesis*
apply (*simp add*: *scaleR-kraus-family.rep-eq kf-apply-def*)
apply (*subst infsum-reindex*)
by (*auto intro*!: *inj-onI*
simp: *kf-apply-def case-prod-unfold*
o-def sandwich-tc-scaleR-left cblinfun.scaleR-left infsum-scaleR-right)
next
case *False*
with *assms* **show** *?thesis*
by (*simp add*: *kf-apply.rep-eq scaleR-kraus-family.rep-eq*)
qed

lemma *kf-bound-scale*[*simp*]: ‹*kf-bound* $(c *_R \mathfrak{E}) = c *_R$ *kf-bound* $\mathfrak{E}$› **if** ‹$c \geq 0$›
apply (*rule cblinfun-cinner-eqI*)
using *that*
by (*simp add*: *kf-bound-from-map cblinfun.scaleR-left kf-scale-apply scaleR-scaleC trace-tc-scaleC*)

lemma *kf-norm-scale*[*simp*]: ‹*kf-norm* $(c *_R \mathfrak{E}) = c *$ *kf-norm* $\mathfrak{E}$› **if** ‹$c \geq 0$›
by (*simp add*: *kf-norm-def that*)

lemma *kf-of-op-apply*: ‹*kf-apply* (*kf-of-op* E) $\varrho =$ *sandwich-tc* $E\ \varrho$›
by (*simp add*: *kf-apply-def kf-of-op.rep-eq*)

lemma *kf-id-apply*[*simp*]: ‹*kf-apply kf-id* $\varrho = \varrho$›
by (*simp add*: *kf-id-def kf-of-op-apply del*: *kf-of-op-id*)

lemma *kf-scale-apply-neg*:
assumes ‹$r \leq 0$›
shows ‹*kf-apply* $(r *_R \mathfrak{E}) = 0$›
apply (*transfer fixing*: r)
using *assms*
by (*auto intro*!: *ext simp*: *scaleR-kraus-family.rep-eq kf-apply.rep-eq*)

lemma *kf-apply-0-left*[*iff*]: ‹*kf-apply* $0\ \varrho = 0$›
apply (*transfer'* *fixing*: ϱ)
by *simp*

lemma *kf-bound-of-op*[*simp*]: ‹*kf-bound* (*kf-of-op* A) $= A* \ o_{CL}\ A$›
by (*simp add*: *kf-bound-def kf-of-op.rep-eq infsum-in-finite*)

lemma *kf-bound-id*[*simp*]: ‹*kf-bound kf-id* $=$ *id-cblinfun*›
by (*simp add*: *kf-id-def del*: *kf-of-op-id*)

2.4 Filtering

lift-definition *kf-filter* :: ‹('x ⇒ *bool*) ⇒ ('a::*chilbert-space*, 'b::*chilbert-space*, 'x) *kraus-family* ⇒ ('a::*chilbert-space*, 'b::*chilbert-space*, 'x) *kraus-family*› **is**
‹λ(P::'x ⇒ *bool*) 𝔈. *Set.filter* (λ(E,x). P x) 𝔈›
proof (*rename-tac* P 𝔈, *rule CollectI*)
fix P 𝔈
assume ‹𝔈 ∈ *Collect kraus-family*›
then have ‹*kraus-family* 𝔈›
by *simp*
then have ‹*bdd-above* (*sum* (λ(E, x). E* o_{CL} E) ' {F. *finite* F ∧ F ⊆ 𝔈})›
by (*simp add*: *kraus-family-def*)
then have ‹*bdd-above* (*sum* (λ(E, x). E* o_{CL} E) ' {F. *finite* F ∧ F ⊆ *Set.filter* P 𝔈})›
apply (*rule bdd-above-mono2*)
by *auto*
moreover from ‹*kraus-family* 𝔈› **have** ‹0 ∉ *fst* ' *Set.filter* P 𝔈›
by (*auto simp add*: *kraus-family-def*)
ultimately show ‹*kraus-family* (*Set.filter* P 𝔈)›
by (*simp add*: *kraus-family-def*)
qed

lemma *kf-filter-false*[*simp*]: ‹*kf-filter* (λ-. *False*) 𝔈 = 0›
apply *transfer* **by** *auto*

lemma *kf-filter-true*[*simp*]: ‹*kf-filter* (λ-. *True*) 𝔈 = 𝔈›
apply *transfer* **by** *auto*

definition ‹*kf-apply-on* 𝔈 S = *kf-apply* (*kf-filter* (λx. x ∈ S) 𝔈)›
notation *kf-apply-on* (- $*_{kr}$ @-/ -) [71, 1000, 70] 70)

lemma *kf-apply-on-pos*:
assumes ‹ϱ ≥ 0›
shows ‹*kf-apply-on* 𝔈 X ϱ ≥ 0›
by (*simp add*: *kf-apply-on-def kf-apply-pos assms*)

lemma *kf-apply-on-bounded-clinear*[*bounded-clinear*]:
shows ‹*bounded-clinear* (*kf-apply-on* 𝔈 X)›
by (*simp add*: *kf-apply-on-def kf-apply-bounded-clinear*)

lemma *kf-filter-twice*:
‹*kf-filter* P (*kf-filter* Q 𝔈) = *kf-filter* (λx. P x ∧ Q x) 𝔈›
apply (*transfer'* *fixing*: P Q)
by *auto*

lemma *kf-apply-on-union-has-sum*:
assumes ‹⋀X Y. X∈F ⟹ Y∈F ⟹ X≠Y ⟹ *disjnt* X Y›
shows ‹((λX. *kf-apply-on* 𝔈 X ϱ) *has-sum* (*kf-apply-on* 𝔈 (⋃F) ϱ)) F›
proof −
define *EE EEf* **where** ‹*EE* = *Rep-kraus-family* 𝔈› **and** ‹*EEf* X = *Set.filter* (λ(E,x). x∈X)

EE› **for** *X*
have *inj*: ‹*inj-on snd* (*SIGMA X*:*F*. *EEf X*)›
using *assms* **by** (*force intro*!: *simp*: *inj-on-def disjnt-def EEf-def*)
have *snd-Sigma*: ‹*snd* ‘ (*SIGMA X*:*F*. *EEf X*) = *EEf* (⋃ *F*)›
apply (*subgoal-tac* ‹⋀*a b x*. (*a*, *b*) ∈ *EE* ⟹ *x* ∈ *F* ⟹ *b* ∈ *x* ⟹ (*a*, *b*) ∈ *snd* ‘ (*SIGMA X*:*F*. *Set.filter* (λ(*E*, *x*). *x* ∈ *X*) *EE*)›)
apply (*auto simp add*: *EEf-def*)[*1*]
by *force*
have *map′-infsum*: ‹*kf-apply-on* 𝔈 *X* ϱ = (∑ ∞(*E*, *x*)∈*EEf X*. *sandwich-tc E* ϱ)› **for** *X*
by (*simp add*: *kf-apply-on-def kf-apply.rep-eq EEf-def kf-filter.rep-eq EE-def case-prod-unfold*)
have *has-sum*: ‹((λ(*E*,*x*). *sandwich-tc E* ϱ) *has-sum* (*kf-apply-on* 𝔈 *X* ϱ)) (*EEf X*)› **for** *X*
using *kf-apply-has-sum*[*of* ϱ ‹*kf-filter* (λ*x*. *x* ∈ *X*) 𝔈›]
by (*simp add*: *kf-apply-on-def kf-filter.rep-eq EEf-def EE-def*)
then have ‹((λ(*E*,*x*). *sandwich-tc E* ϱ) *has-sum* (*kf-apply-on* 𝔈 (⋃ *F*) ϱ)) (*snd* ‘ (*SIGMA X*:*F*. *EEf X*))›
by (*simp add*: *snd-Sigma*)
then have ‹((λ(*X*,(*E*,*x*)). *sandwich-tc E* ϱ) *has-sum* (*kf-apply-on* 𝔈 (⋃ *F*) ϱ)) (*SIGMA X*:*F*. *EEf X*)›
apply (*subst* (*asm*) *has-sum-reindex*)
apply (*rule inj*)
by (*simp add*: *o-def case-prod-unfold*)
then have ‹((λ*X*. ∑ ∞(*E*, *x*)∈*EEf X*. *sandwich-tc E* ϱ) *has-sum kf-apply-on* 𝔈 (⋃ *F*) ϱ) *F*›
by (*rule has-sum-Sigma′-banach*)
then show ‹((λ*X*. *kf-apply-on* 𝔈 *X* ϱ) *has-sum kf-apply-on* 𝔈 (⋃ *F*) ϱ) *F*›
by (*auto intro*: *has-sum-cong*[*THEN iffD2*, *rotated*] *simp*: *map′-infsum*)
qed

lemma *kf-apply-on-empty*[*simp*]: ‹*kf-apply-on E* {} ϱ = *0*›
by (*simp add*: *kf-apply-on-def*)

lemma *kf-apply-on-union-eqI*:
assumes ‹⋀*X Y*. (*X*,*Y*)∈*F* ⟹ *kf-apply-on* 𝔈 *X* ϱ = *kf-apply-on* 𝔉 *Y* ϱ›
assumes ‹⋀*X Y X′ Y′*. (*X*,*Y*)∈*F* ⟹ (*X′*,*Y′*)∈*F* ⟹ (*X*,*Y*)≠(*X′*,*Y′*) ⟹ *disjnt X X′*›
assumes ‹⋀*X Y X′ Y′*. (*X*,*Y*)∈*F* ⟹ (*X′*,*Y′*)∈*F* ⟹ (*X*,*Y*)≠(*X′*,*Y′*) ⟹ *disjnt Y Y′*›
assumes *XX*: ‹*XX* = ⋃ (*fst* ‘ *F*)› **and** *YY*: ‹*YY* = ⋃ (*snd* ‘ *F*)›
shows ‹*kf-apply-on* 𝔈 *XX* ϱ = *kf-apply-on* 𝔉 *YY* ϱ›
proof −
define *F′* **where** ‹*F′* = *Set.filter* (λ(*X*,*Y*). *X*≠{} ∧ *Y*≠{}) *F*›
have *inj1*: ‹*inj-on fst F′*›
apply (*rule inj-onI*)
using *assms*(*2*)
unfolding *F′-def*
by *fastforce*
have *inj2*: ‹*inj-on snd F′*›
apply (*rule inj-onI*)
using *assms*(*3*)
unfolding *F′-def*
by *fastforce*
have ‹((λ*X*. *kf-apply-on* 𝔈 *X* ϱ) *has-sum kf-apply-on* 𝔈 *XX* ϱ) (*fst* ‘ *F*)›

```
  unfolding XX
  apply (rule kf-apply-on-union-has-sum)
  using assms(2) by force
then have ‹((λX. kf-apply-on 𝔈 X ϱ) has-sum kf-apply-on 𝔈 XX ϱ) (fst ‘ F')›
proof (rule has-sum-cong-neutral[OF - - refl, THEN iffD2, rotated -1])
  show ‹kf-apply-on 𝔈 X ϱ = 0› if ‹X ∈ fst ‘ F' - fst ‘ F› for X
    using that by (auto simp: F'-def)
  show ‹kf-apply-on 𝔈 X ϱ = 0› if ‹X ∈ fst ‘ F - fst ‘ F'› for X
  proof -
    from that obtain Y where ‹(X,Y) ∈ F› and ‹X = {} ∨ Y = {}›
      apply atomize-elim
      by (force intro!: simp: F'-def)
    then consider (X) ‹X = {}› | (Y) ‹Y = {}›
      by auto
    then show ?thesis
    proof cases
      case X
      then show ?thesis
        by simp
    next
      case Y
      then have ‹kf-apply-on 𝔉 Y ϱ = 0›
        by simp
      then show ?thesis
        using ‹(X, Y) ∈ F› assms(1) by presburger
    qed
  qed
qed
then have sum1: ‹((λ(X,Y). kf-apply-on 𝔈 X ϱ) has-sum kf-apply-on 𝔈 XX ϱ) F'›
  apply (subst (asm) has-sum-reindex)
  using inj1 by (auto intro!: simp: o-def case-prod-unfold)
have ‹((λY. kf-apply-on 𝔉 Y ϱ) has-sum kf-apply-on 𝔉 YY ϱ) (snd ‘ F)›
  unfolding YY
  apply (rule kf-apply-on-union-has-sum)
  using assms(3) by force
then have ‹((λY. kf-apply-on 𝔉 Y ϱ) has-sum kf-apply-on 𝔉 YY ϱ) (snd ‘ F')›
proof (rule has-sum-cong-neutral[OF - - refl, THEN iffD2, rotated -1])
  show ‹kf-apply-on 𝔉 Y ϱ = 0› if ‹Y ∈ snd ‘ F' - snd ‘ F› for Y
    using that by (auto simp: F'-def)
  show ‹kf-apply-on 𝔉 Y ϱ = 0› if ‹Y ∈ snd ‘ F - snd ‘ F'› for Y
  proof -
    from that obtain X where ‹(X,Y) ∈ F› and ‹X = {} ∨ Y = {}›
      apply atomize-elim
      by (force intro!: simp: F'-def)
    then consider (X) ‹X = {}› | (Y) ‹Y = {}›
      by auto
    then show ?thesis
    proof cases
      case Y
```

then show *?thesis*
by *simp*
next
case X
then have ‹*kf-apply-on* $\mathfrak{E}$ X $\varrho = 0$›
by *simp*
then show *?thesis*
using ‹$(X,\ Y) \in F$› *assms*(*1*) **by** *simp*
qed
qed
qed
then have ‹((λ(*X*,*Y*). *kf-apply-on* $\mathfrak{F}$ Y ϱ) *has-sum kf-apply-on* $\mathfrak{F}$ YY ϱ) F'›
apply (*subst* (*asm*) *has-sum-reindex*)
using *inj2* **by** (*auto intro*!: *simp*: *o-def case-prod-unfold*)
then have *sum2*: ‹((λ(*X*,*Y*). *kf-apply-on* $\mathfrak{E}$ X ϱ) *has-sum kf-apply-on* $\mathfrak{F}$ YY ϱ) F'›
apply (*rule has-sum-cong*[*THEN iffD1*, *rotated* −*1*])
using *assms*(*1*) **by** (*auto simp*: F'*-def*)
from *sum1 sum2* **show** *?thesis*
using *has-sum-unique* **by** *blast*
qed

lemma *kf-apply-on-UNIV*[*simp*]: ‹*kf-apply-on* $\mathfrak{E}$ *UNIV* = *kf-apply* $\mathfrak{E}$›
by (*auto simp*: *kf-apply-on-def*)

lemma *kf-apply-on-CARD-1*[*simp*]: ‹(λ$\mathfrak{E}$. *kf-apply-on* $\mathfrak{E}$ {*x*::-::*CARD-1*}) = *kf-apply*›
apply (*subst asm-rl*[*of* ‹{*x*} = *UNIV*›])
by *auto*

lemma *kf-apply-on-union-summable-on*:
assumes ‹⋀*X Y*. *X*∈*F* ⟹ *Y*∈*F* ⟹ *X*≠*Y* ⟹ *disjnt X Y*›
shows ‹(λ*X*. *kf-apply-on* $\mathfrak{E}$ X ϱ) *summable-on F*›
using *assms* **by** (*auto intro*!: *has-sum-imp-summable kf-apply-on-union-has-sum*)

lemma *kf-apply-on-union-infsum*:
assumes ‹⋀*X Y*. *X*∈*F* ⟹ *Y*∈*F* ⟹ *X*≠*Y* ⟹ *disjnt X Y*›
shows ‹($\sum_\infty$*X*∈*F*. *kf-apply-on* $\mathfrak{E}$ X ϱ) = *kf-apply-on* $\mathfrak{E}$ ($\bigcup F$) ϱ›
by (*metis assms infsumI kf-apply-on-union-has-sum*)

lemma *kf-bound-filter*:
‹*kf-bound* (*kf-filter P* $\mathfrak{E}$) ≤ *kf-bound* $\mathfrak{E}$›
proof (*unfold kf-bound.rep-eq*, *rule infsum-mono-neutral-wot*)
show ‹*summable-on-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (*Rep-kraus-family* (*kf-filter P* $\mathfrak{E}$))›
using *Rep-kraus-family kraus-family-iff-summable* **by** *blast*
show ‹*summable-on-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (*Rep-kraus-family* $\mathfrak{E}$)›
using *Rep-kraus-family kraus-family-iff-summable* **by** *blast*

fix $Ex :: \langle 'a \Rightarrow_{CL} 'b \times 'c \rangle$
show $\langle (case\ Ex\ of\ (E,\ x) \Rightarrow E*\ o_{CL}\ E) \leq (case\ Ex\ of\ (E,\ x) \Rightarrow E*\ o_{CL}\ E) \rangle$
by *simp*
show $\langle 0 \leq (case\ Ex\ of\ (E,\ x) \Rightarrow E*\ o_{CL}\ E) \rangle$
by (*auto intro*!: *positive-cblinfun-squareI simp*: *case-prod-unfold*)
show $\langle (case\ Ex\ of\ (E,\ x) \Rightarrow E*\ o_{CL}\ E) \leq 0 \rangle$
if $\langle Ex \in Rep\text{-}kraus\text{-}family\ (kf\text{-}filter\ P\ \mathfrak{E}) - Rep\text{-}kraus\text{-}family\ \mathfrak{E} \rangle$
using *that*
by (*auto simp*: *kf-filter.rep-eq*)
qed

lemma *kf-norm-filter*:
$\langle kf\text{-}norm\ (kf\text{-}filter\ P\ \mathfrak{E}) \leq kf\text{-}norm\ \mathfrak{E} \rangle$
unfolding *kf-norm-def*
apply (*rule norm-cblinfun-mono*)
by (*simp-all add*: *kf-bound-filter*)

lemma *kf-domain-filter*[*simp*]:
$\langle kf\text{-}domain\ (kf\text{-}filter\ P\ E) = Set.filter\ P\ (kf\text{-}domain\ E) \rangle$
apply (*transfer' fixing*: *P*)
by *force*

lemma *kf-filter-0-right*[*simp*]: $\langle kf\text{-}filter\ P\ 0 = 0 \rangle$
apply (*transfer' fixing*: *P*)
by *auto*

lemma *kf-apply-on-0-right*[*simp*]: $\langle kf\text{-}apply\text{-}on\ \mathfrak{E}\ X\ 0 = 0 \rangle$
by (*simp add*: *kf-apply-on-def*)

lemma *kf-apply-on-0-left*[*simp*]: $\langle kf\text{-}apply\text{-}on\ 0\ X\ \varrho = 0 \rangle$
by (*simp add*: *kf-apply-on-def*)

lemma *kf-apply-on-mono3*:
assumes $\langle \varrho \leq \sigma \rangle$
shows $\langle \mathfrak{E} *_{kr} @X\ \varrho \leq \mathfrak{E} *_{kr} @X\ \sigma \rangle$
by (*simp add*: *assms kf-apply-mono-right kf-apply-on-def*)

lemma *kf-apply-on-mono2*:
assumes $\langle X \subseteq Y \rangle$ **and** $\langle \varrho \geq 0 \rangle$
shows $\langle \mathfrak{E} *_{kr} @X\ \varrho \leq \mathfrak{E} *_{kr} @Y\ \varrho \rangle$
proof −
wlog $\langle Y \neq \{\} \rangle$
using *assms*(*1*) *negation* **by** *auto*
have [*simp*]: $\langle X \cup Y = Y \rangle$
using *assms*(*1*) **by** *blast*
from *kf-apply-on-union-infsum*[**where** $F=\langle \{X,\ Y-X\} \rangle$ **and** $\mathfrak{E}=\mathfrak{E}$ **and** $\varrho=\varrho$]
have $\langle (\sum X \in \{X,\ Y - X\}.\ \mathfrak{E} *_{kr} @X\ \varrho) = \mathfrak{E} *_{kr} @Y\ \varrho \rangle$

by (*auto simp*: *disjnt-iff sum.insert*)
then have ‹$\mathfrak{E} *_{kr} @X\ \varrho + \mathfrak{E} *_{kr} @(Y-X)\ \varrho = \mathfrak{E} *_{kr} @Y\ \varrho$›
apply (*subst* (*asm*) *sum.insert*)
using ‹$Y \neq \{\}$›
by *auto*
moreover have ‹$\mathfrak{E} *_{kr} @(Y-X)\ \varrho \geq 0$›
by (*simp add*: *assms*(*2*) *kf-apply-on-pos*)
ultimately show ‹$\mathfrak{E} *_{kr} @X\ \varrho \leq \mathfrak{E} *_{kr} @Y\ \varrho$›
by (*metis le-add-same-cancel1*)
qed

lemma *kf-operators-filter*: ‹*kf-operators* (*kf-filter P* $\mathfrak{E}$) $\subseteq$ *kf-operators* $\mathfrak{E}$›
apply (*transfer' fixing*: *P*)
by *auto*

lemma *kf-equal-if-filter-equal*:
assumes ‹$\bigwedge x$. *kf-filter* ((=)*x*) $\mathfrak{E}$ = *kf-filter* ((=)*x*) $\mathfrak{F}$›
shows ‹$\mathfrak{E} = \mathfrak{F}$›
using *assms*
apply *transfer'*
by *fastforce*

2.5 Equivalence

definition ‹*kf-eq-weak* $\mathfrak{E}$ $\mathfrak{F}$ $\longleftrightarrow$ *kf-apply* $\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$›
definition ‹*kf-eq* $\mathfrak{E}$ $\mathfrak{F}$ $\longleftrightarrow$ ($\forall x$. *kf-apply-on* $\mathfrak{E}$ $\{x\}$ = *kf-apply-on* $\mathfrak{F}$ $\{x\}$)›

open-bundle *kraus-map-syntax* **begin**
notation *kf-eq-weak* (**infix** $=_{kr}$ *50*)
notation *kf-eq* (**infix** $\equiv_{kr}$ *50*)
notation *kf-apply* (**infixr** ‹$*_{kr}$› *70*)
notation *kf-apply-on* (- $*_{kr}$ @-/ - [*71*, *1000*, *70*] *70*)
end

lemma *kf-eq-weak-reflI*[*iff*]: ‹$x =_{kr} x$›
by (*simp add*: *kf-eq-weak-def*)

lemma *kf-eq-weak-refl0*[*iff*]: ‹$0 =_{kr} 0$›
by (*simp add*: *kf-eq-weak-def*)

lemma *kf-bound-cong*:
assumes ‹$\mathfrak{E} =_{kr} \mathfrak{F}$›
shows ‹*kf-bound* $\mathfrak{E}$ = *kf-bound* $\mathfrak{F}$›
using *assms* **by** (*auto intro*!: *cblinfun-cinner-eqI simp*: *kf-eq-weak-def kf-bound-from-map*)

lemma *kf-norm-cong*:
assumes ‹$\mathfrak{E} =_{kr} \mathfrak{F}$›
shows ‹*kf-norm* $\mathfrak{E}$ = *kf-norm* $\mathfrak{F}$›
using *assms*

by (*simp add*: *kf-norm-def kf-bound-cong*)

lemma *kf-eq-weakI*:
assumes ‹$\bigwedge\varrho.\ \varrho \geq 0 \Longrightarrow \mathfrak{E} *_{kr} \varrho = \mathfrak{F} *_{kr} \varrho$›
shows ‹$\mathfrak{E} =_{kr} \mathfrak{F}$›
unfolding *kf-eq-weak-def*
apply (*rule eq-from-separatingI*)
apply (*rule separating-density-ops*[**where** *B=1*])
using *assms* **by** *auto*

lemma *kf-eqI*:
assumes ‹$\bigwedge x\ \varrho.\ \varrho \geq 0 \Longrightarrow \mathfrak{E} *_{kr} @\{x\}\ \varrho = \mathfrak{F} *_{kr} @\{x\}\ \varrho$›
shows ‹$\mathfrak{E} \equiv_{kr} \mathfrak{F}$›
using *kf-eq-weakI*
using *assms*
by (*auto simp*: *kf-eq-weak-def kf-eq-def kf-apply-on-def*)

lemma *kf-eq-weak-trans*[*trans*]:
‹$F =_{kr} G \Longrightarrow G =_{kr} H \Longrightarrow F =_{kr} H$›
by (*simp add*: *kf-eq-weak-def*)

lemma *kf-apply-eqI*:
assumes ‹$\mathfrak{E} =_{kr} \mathfrak{F}$›
shows ‹$\mathfrak{E} *_{kr} \varrho = \mathfrak{F} *_{kr} \varrho$›
using *assms* **by** (*simp add*: *kf-eq-weak-def*)

lemma *kf-eq-imp-eq-weak*:
assumes ‹$\mathfrak{E} \equiv_{kr} \mathfrak{F}$›
shows ‹$\mathfrak{E} =_{kr} \mathfrak{F}$›
unfolding *kf-eq-weak-def*
apply (*subst kf-apply-on-UNIV*[*symmetric*])+
apply (*rule ext*)
apply (*rule kf-apply-on-union-eqI*[**where** F=‹*range* ($\lambda x.$ ($\{x\}$,$\{x\}$))› **and** $\mathfrak{E}$=$\mathfrak{E}$ **and** $\mathfrak{F}$=$\mathfrak{F}$])
using *assms* **by** (*auto simp*: *kf-eq-def*)

lemma *kf-filter-cong-eq*[*cong*]:
assumes ‹$\mathfrak{E} = \mathfrak{F}$›
assumes ‹$\bigwedge x.\ x \in$ *kf-domain* $\mathfrak{E} \Longrightarrow P\ x = Q\ x$›
shows ‹*kf-filter* $P\ \mathfrak{E}$ = *kf-filter* $Q\ \mathfrak{F}$›
using *assms*
apply *transfer*
by (*force simp*: *kraus-family-def*)

lemma *kf-filter-cong*:
assumes ‹$\mathfrak{E} \equiv_{kr} \mathfrak{F}$›
assumes ‹$\bigwedge x.\ x \in$ *kf-domain* $\mathfrak{E} \Longrightarrow P\ x = Q\ x$›

```
  shows ‹kf-filter P 𝔈 ≡kr kf-filter Q 𝔉›
proof −
  have ‹kf-apply (kf-filter (λxa. xa = x ∧ P xa) 𝔈)
      = kf-apply (kf-filter (λxa. xa = x ∧ Q xa) 𝔈)› for x
  proof (cases ‹x ∈ kf-domain 𝔈›)
    case True
    with assms have ‹P x = Q x›
      by blast
    then have ‹(λxa. xa = x ∧ P xa) = (λxa. xa = x ∧ Q xa)›
      by auto
    then show ?thesis
      by presburger
  next
    case False
    then have ‹kf-filter (λxa. xa = x ∧ P xa) 𝔈 = 0›
      apply (transfer fixing: P x)
      by force
    have ∗: ‹(∑∞E∈Set.filter (λ(E, xa). xa = x ∧ P xa) 𝔈. sandwich-tc (fst E) ϱ) = 0›
      if ‹x ∉ snd ‘ Set.filter (λ(E, x). E ≠ 0) 𝔈› for 𝔈 :: ‹('a ⇒CL 'b × 'c) set› and ϱ P
      apply (rule infsum-0)
      using that
      by force
    have ‹kf-apply (kf-filter (λxa. xa = x ∧ P xa) 𝔈) = 0› for P
      using False
      apply (transfer fixing: x P)
      using ∗
      by (force intro!: ext simp: kraus-family-def image-iff)
    then show ?thesis
      by simp
  qed
  also have ‹kf-apply (kf-filter (λxa. xa = x ∧ Q xa) 𝔈)
        = kf-apply (kf-filter (λxa. xa = x ∧ Q xa) 𝔉)› for x
  proof (cases ‹Q x›)
    case True
    then have ‹(z = x ∧ Q z) ⟷ (z = x)› for z
      by auto
    with assms show ?thesis
      by (simp add: kf-eq-def kf-apply-on-def)
  next
    case False
    then have [simp]: ‹(z = x ∧ Q z) ⟷ False› for z
      by auto
    show ?thesis
      by (simp add: kf-eq-def kf-apply-on-def)
  qed
  finally show ?thesis
    by (simp add: kf-eq-def kf-apply-on-def kf-filter-twice)
qed
```

lemma *kf-filter-cong-weak*:
assumes ‹$\mathfrak{E} \equiv_{kr} \mathfrak{F}$›
assumes ‹$\bigwedge x.\ x \in$ *kf-domain* $\mathfrak{E} \Longrightarrow P\ x = Q\ x$›
shows ‹*kf-filter* $P\ \mathfrak{E} =_{kr}$ *kf-filter* $Q\ \mathfrak{F}$›
by (*simp add*: *assms kf-eq-imp-eq-weak kf-filter-cong*)

lemma *kf-eq-refl*[*iff*]: ‹$\mathfrak{E} \equiv_{kr} \mathfrak{E}$›
using *kf-eq-def* **by** *blast*

lemma *kf-eq-trans*[*trans*]:
assumes ‹$\mathfrak{E} \equiv_{kr} \mathfrak{F}$›
assumes ‹$\mathfrak{F} \equiv_{kr} \mathfrak{G}$›
shows ‹$\mathfrak{E} \equiv_{kr} \mathfrak{G}$›
by (*metis assms*(*1*) *assms*(*2*) *kf-eq-def*)

lemma *kf-eq-sym*[*sym*]:
assumes ‹$\mathfrak{E} \equiv_{kr} \mathfrak{F}$›
shows ‹$\mathfrak{F} \equiv_{kr} \mathfrak{E}$›
by (*metis assms kf-eq-def*)

lemma *kf-eq-weak-imp-eq-CARD-1*:
fixes $\mathfrak{E}\ \mathfrak{F}$ :: ‹('*a*::*chilbert-space*, '*b*::*chilbert-space*, '*x*::*CARD-1*) *kraus-family*›
assumes ‹$\mathfrak{E} =_{kr} \mathfrak{F}$›
shows ‹$\mathfrak{E} \equiv_{kr} \mathfrak{F}$›
by (*metis CARD-1-UNIV assms kf-eqI kf-eq-weak-def kf-apply-on-UNIV*)

lemma *kf-apply-on-eqI-filter*:
assumes ‹*kf-filter* $(\lambda x.\ x \in X)\ \mathfrak{E} \equiv_{kr}$ *kf-filter* $(\lambda x.\ x \in X)\ \mathfrak{F}$›
shows ‹$\mathfrak{E} *_{kr} @X\ \varrho = \mathfrak{F} *_{kr} @X\ \varrho$›
proof (*rule kf-apply-on-union-eqI*[**where** F=‹$(\lambda x.(\{x\},\{x\}))$ ‘ X›])
show ‹$(A,\ B) \in (\lambda x.\ (\{x\},\ \{x\}))$ ‘ $X \Longrightarrow$
$(A',\ B') \in (\lambda x.\ (\{x\},\ \{x\}))$ ‘ $X \Longrightarrow (A,\ B) \neq (A',\ B') \Longrightarrow$ *disjnt* $A\ A'$›
for $A\ B\ A'\ B'$
by *force*
show ‹$(A,\ B) \in (\lambda x.\ (\{x\},\ \{x\}))$ ‘ $X \Longrightarrow$
$(A',\ B') \in (\lambda x.\ (\{x\},\ \{x\}))$ ‘ $X \Longrightarrow (A,\ B) \neq (A',\ B') \Longrightarrow$ *disjnt* $B\ B'$›
for $A\ B\ A'\ B'$
by *force*
show ‹$X = \bigcup$ (*fst* ‘ $(\lambda x.\ (\{x\},\ \{x\}))$ ‘ X)›
by *simp*
show ‹$X = \bigcup$ (*snd* ‘ $(\lambda x.\ (\{x\},\ \{x\}))$ ‘ X)›
by *simp*
show ‹$\mathfrak{E} *_{kr} @A\ \varrho = \mathfrak{F} *_{kr} @B\ \varrho$› **if** ‹$(A,\ B) \in (\lambda x.\ (\{x\},\ \{x\}))$ ‘ X› **for** $A\ B$
proof −
from *that* **obtain** x **where** ‹$x \in X$› **and** A: ‹$A = \{x\}$› **and** B: ‹$B = \{x\}$›
by *blast*
from ‹$x \in X$› **have** $*$: ‹$(\lambda x'.\ x = x' \wedge x' \in X) = (\lambda x'.\ x' = x)$›
by *blast*

from *assms* **have** ‹*kf-filter* ((=)*x*) (*kf-filter* (λ*x*. *x* ∈ *X*) 𝔈) =$_{kr}$ *kf-filter* ((=)*x*) (*kf-filter* (λ*x*. *x* ∈ *X*) 𝔉)›
by (*simp add*: *kf-filter-cong-weak*)
then have ‹*kf-filter* (λ*x'*. *x'*=*x*) 𝔈 =$_{kr}$ *kf-filter* (λ*x'*. *x'*=*x*) 𝔉›
by (*simp add*: *kf-filter-twice* ∗ *cong del*: *kf-filter-cong-eq*)
then have ‹𝔈 ∗$_{kr}$ @{*x*} ϱ = 𝔉 ∗$_{kr}$ @{*x*} ϱ›
by (*simp add*: *kf-apply-on-def kf-eq-weak-def*)
then show *?thesis*
by (*simp add*: *A B*)
qed
qed

lemma *kf-apply-on-eqI*:
assumes ‹𝔈 ≡$_{kr}$ 𝔉›
shows ‹𝔈 ∗$_{kr}$ @*X* ϱ = 𝔉 ∗$_{kr}$ @*X* ϱ›
apply (*rule kf-apply-on-union-eqI*[**where** *F*=‹(λ*x*.({*x*},{*x*})) ‘ *X*›])
using *assms* **by** (*auto simp*: *kf-eq-def*)

lemma *kf-apply-eq0I*:
assumes ‹𝔈 =$_{kr}$ *0*›
shows ‹𝔈 ∗$_{kr}$ ϱ = *0*›
using *assms kf-eq-weak-def* **by** *force*

lemma *kf-eq-weak0-imp-kf-eq0*:
assumes ‹𝔈 =$_{kr}$ *0*›
shows ‹𝔈 ≡$_{kr}$ *0*›
proof −
have ‹𝔈 ∗$_{kr}$ @{*x*} ϱ = *0*› **if** ‹ϱ ≥ *0*› **for** ϱ *x*
proof −
from *assms* **have** ‹𝔈 ∗$_{kr}$ @*UNIV* ϱ = *0*›
by (*simp add*: *kf-eq-weak-def*)
moreover have ‹𝔈 ∗$_{kr}$ @{*x*} ϱ ≤ 𝔈 ∗$_{kr}$ @*UNIV* ϱ›
apply (*rule kf-apply-on-mono2*)
using *that* **by** *auto*
moreover have ‹𝔈 ∗$_{kr}$ @{*x*} ϱ ≥ *0*›
using *that*
by (*simp add*: *kf-apply-on-pos*)
ultimately show *?thesis*
by (*simp add*: *basic-trans-rules*(*24*))
qed
then show *?thesis*
by (*simp add*: *kf-eqI*)
qed

lemma *kf-apply-on-eq0I*:
assumes ‹𝔈 =$_{kr}$ *0*›
shows ‹𝔈 ∗$_{kr}$ @*X* ϱ = *0*›
proof −
from *assms*

```
  have ‹𝔈 ≡_kr 0›
    by (rule kf-eq-weak0-imp-kf-eq0)
  then have ‹𝔈 *_kr @X ϱ = 0 *_kr @X ϱ›
    by (intro kf-apply-on-eqI-filter kf-filter-cong refl)
  then show ?thesis
    by simp
qed



lemma kf-filter-to-domain[simp]:
  ‹kf-filter (λx. x ∈ kf-domain 𝔈) 𝔈 = 𝔈›
  apply transfer
  by (force simp: kraus-family-def)

lemma kf-eq-0-iff-eq-0: ‹E =_kr 0 ⟷ E = 0›
proof (rule iffI)
  assume asm: ‹E =_kr 0›
  show ‹E = 0›
  proof (insert asm, unfold kf-eq-weak-def, transfer, rename-tac 𝔈)
    fix 𝔈 :: ‹('a ⇒_CL 'b × 'c) set›
    assume ‹𝔈 ∈ Collect kraus-family›
    then have ‹kraus-family 𝔈›
      by simp
    have summable1: ‹(λE. sandwich-tc (fst E) ϱ) summable-on 𝔈› for ϱ
      apply (rule abs-summable-summable)
      using kf-apply-abs-summable[OF ‹kraus-family 𝔈›]
      by (simp add: case-prod-unfold)
    then have summable2: ‹(λE. sandwich-tc (fst E) ϱ) summable-on 𝔈−{E}› for E ϱ
      apply (rule summable-on-subset-banach)
      by simp
    assume ‹(λϱ. ∑_∞E∈𝔈. sandwich-tc (fst E) ϱ) = (λϱ. ∑_∞E∈{}. sandwich-tc (fst E) ϱ)›
    then have sum0: ‹(∑_∞E∈𝔈. sandwich-tc (fst E) ϱ) = 0› for ϱ
      apply simp by meson
    have sand-Eϱ-0:  ‹sandwich-tc (fst E) ϱ = 0› if ‹E ∈ 𝔈› and ‹ϱ ≥ 0› for E ϱ
    proof (rule ccontr)
      assume Eϱ-neq0: ‹sandwich-tc (fst E) ϱ ≠ 0›
      have Eϱ-geq0: ‹sandwich-tc (fst E) ϱ ≥ 0›
        by (simp add: ‹0 ≤ ϱ› sandwich-tc-pos)
      have Eϱ-ge0: ‹sandwich-tc (fst E) ϱ > 0›
        using Eϱ-neq0 Eϱ-geq0 by order
       have ‹(∑_∞E∈𝔈. sandwich-tc (fst E) ϱ) = (∑_∞E∈𝔈−{E}. sandwich-tc (fst E) ϱ) +
sandwich-tc (fst E) ϱ›
        apply (subst asm-rl[of ‹𝔈 = insert E (𝔈−{E})›])
        using that apply blast
        apply (subst infsum-insert)
        by (auto intro!: summable2)
      also have ‹... ≥ 0 + sandwich-tc (fst E) ϱ› (is ‹- ≥ ...›)
        by (simp add: ‹0 ≤ ϱ› infsum-nonneg-traceclass sandwich-tc-pos)
```

```
    also have ‹... > 0› (is ‹- > ...›)
      using Eϱ-ge0
      by simp
    ultimately have ‹(∑∞E∈𝔈. sandwich-tc (fst E) ϱ) > 0›
      by simp
    then show False
      using sum0 by simp
  qed
  then have ‹fst E = 0› if ‹E ∈ 𝔈› for E
    apply (rule sandwich-tc-eq0-D[where B=1])
    using that by auto
  then show ‹𝔈 = {}›
    using ‹kraus-family 𝔈›
    by (auto simp: kraus-family-def)
 qed
next
 assume ‹E = 0›
 then show ‹E =kr 0›
   by simp
qed

lemma in-kf-domain-iff-apply-nonzero:
 ‹x ∈ kf-domain 𝔈 ⟷ kf-apply-on 𝔈 {x} ≠ 0›
proof −
 define 𝔈' where ‹𝔈' = Rep-kraus-family 𝔈›
 have ‹x ∉ kf-domain 𝔈 ⟷ (∀(E,x')∈Rep-kraus-family 𝔈. x'≠x)›
   by (force simp: kf-domain.rep-eq)
 also have ‹... ⟷ (∀(E,x')∈Rep-kraus-family (kf-filter (λy. y=x) 𝔈). False)›
   by (auto simp: kf-filter.rep-eq)
 also have ‹... ⟷ Rep-kraus-family (kf-filter (λy. y=x) 𝔈) = {}›
   by (auto simp:)
 also have ‹... ⟷ (kf-filter (λy. y=x) 𝔈) = 0›
   using Rep-kraus-family-inject zero-kraus-family.rep-eq by auto
 also have ‹... ⟷ kf-apply (kf-filter (λy. y=x) 𝔈) = 0›
   apply (subst kf-eq-0-iff-eq-0[symmetric])
   by (simp add: kf-eq-weak-def)
 also have ‹... ⟷ kf-apply-on 𝔈 {x} = 0›
   by (simp add: kf-apply-on-def)
 finally show ?thesis
   by auto
qed


lemma kf-domain-cong:
 assumes ‹𝔈 ≡kr 𝔉›
 shows ‹kf-domain 𝔈 = kf-domain 𝔉›
 apply (rule Set.set-eqI)
 using assms
 by (simp add: kf-eq-def in-kf-domain-iff-apply-nonzero)
```

lemma *kf-eq-weak-sym*[*sym*]:
assumes ‹$\mathfrak{E} =_{kr} \mathfrak{F}$›
shows ‹$\mathfrak{F} =_{kr} \mathfrak{E}$›
by (*metis assms kf-eq-weak-def*)

lemma *kf-eqI-from-filter-eq-weak*:
assumes ‹$\bigwedge x.$ *kf-filter* ((=) *x*) *E* $=_{kr}$ *kf-filter* ((=) *x*) *F*›
shows ‹*E* $\equiv_{kr}$ *F*›
using *assms*
apply (*simp add*: *kf-eq-weak-def kf-eq-def kf-apply-on-def*)
apply (*subst flip-eq-const*)
apply (*subst flip-eq-const*)
by *simp*

lemma *kf-eq-weak-from-separatingI*:
fixes *E* :: ‹('*q*::*chilbert-space*,'*r*::*chilbert-space*,'*x*) *kraus-family*›
and *F* :: ‹('*q*,'*r*,'*y*) *kraus-family*›
assumes ‹*separating-set* (*bounded-clinear* :: (('*q*,'*q*) *trace-class* ⇒ ('*r*,'*r*) *trace-class*) ⇒ *bool*) *S*›
assumes ‹$\bigwedge \varrho.$ $\varrho \in S \Longrightarrow E *_{kr} \varrho = F *_{kr} \varrho$›
shows ‹*E* $=_{kr}$ *F*›
proof −
have ‹*kf-apply E* = *kf-apply F*›
by (*metis assms*(*1*) *assms*(*2*) *kf-apply-bounded-clinear separating-set-def*)
then show *?thesis*
by (*simp add*: *kf-eq-weakI*)
qed

lemma *kf-eq-weak-eq-trans*[*trans*]: ‹$a =_{kr} b \Longrightarrow b \equiv_{kr} c \Longrightarrow a =_{kr} c$›
by (*metis kf-eq-imp-eq-weak kf-eq-weak-def*)

lemma *kf-eq-eq-weak-trans*[*trans*]: ‹$a \equiv_{kr} b \Longrightarrow b =_{kr} c \Longrightarrow a =_{kr} c$›
by (*metis kf-eq-imp-eq-weak kf-eq-weak-def*)

instantiation *kraus-family* :: (*chilbert-space*,*chilbert-space*,*type*) *preorder* **begin**
definition *less-eq-kraus-family* **where** ‹$\mathfrak{E} \leq \mathfrak{F} \longleftrightarrow (\forall x.\ \forall \varrho{\geq}0.$ *kf-apply-on* $\mathfrak{E}$ $\{x\}$ $\varrho \leq$ *kf-apply-on* $\mathfrak{F}$ $\{x\}$ $\varrho)$›
definition *less-kraus-family* **where** ‹$\mathfrak{E} < \mathfrak{F} \longleftrightarrow \mathfrak{E} \leq \mathfrak{F} \wedge \neg\ \mathfrak{E} \equiv_{kr} \mathfrak{F}$›
lemma *kf-antisym*: ‹$\mathfrak{E} \equiv_{kr} \mathfrak{F} \longleftrightarrow \mathfrak{E} \leq \mathfrak{F} \wedge \mathfrak{F} \leq \mathfrak{E}$›
for $\mathfrak{E}$ $\mathfrak{F}$ :: ‹('*a*, '*b*, '*c*) *kraus-family*›
by (*smt* (*verit*, *ccfv-SIG*) *kf-apply-on-eqI kf-eqI less-eq-kraus-family-def order.refl order-antisym-conv*)
instance
proof (*intro-classes*)
fix $\mathfrak{E}$ $\mathfrak{F}$ $\mathfrak{G}$ :: ‹('*a*, '*b*, '*c*) *kraus-family*›
show ‹($\mathfrak{E} < \mathfrak{F}$) = ($\mathfrak{E} \leq \mathfrak{F} \wedge \neg\ \mathfrak{F} \leq \mathfrak{E}$)›
using *kf-antisym less-kraus-family-def* **by** *auto*

show ‹$\mathfrak{E} \leq \mathfrak{E}$›
using *less-eq-kraus-family-def* **by** *auto*
show ‹$\mathfrak{E} \leq \mathfrak{F} \Longrightarrow \mathfrak{F} \leq \mathfrak{G} \Longrightarrow \mathfrak{E} \leq \mathfrak{G}$›
by (*meson basic-trans-rules*(*23*) *less-eq-kraus-family-def*)
qed
end

lemma *kf-apply-on-mono1*:
assumes ‹$\mathfrak{E} \leq \mathfrak{F}$› **and** ‹$\varrho \geq 0$›
shows ‹$\mathfrak{E} *_{kr} @X\ \varrho \leq \mathfrak{F} *_{kr} @X\ \varrho$›
proof −
have [*simp*]: ‹$\bigcup((\lambda x.\ \{x\})\ `\ X) = X$› **for** X :: ‹$'c\ set$›
by *auto*
have ‹$((\lambda X.\ \mathfrak{E} *_{kr} @X\ \varrho)\ has\text{-}sum\ \mathfrak{E} *_{kr} @(\bigcup((\lambda x.\ \{x\})\ `\ X))\ \varrho)\ ((\lambda x.\ \{x\})\ `\ X)$›
for $\mathfrak{E}$ :: ‹$('a,\ 'b,\ 'c)\ kraus\text{-}family$› **and** X
apply (*rule kf-apply-on-union-has-sum*)
by *auto*
then have *sum*: ‹$((\lambda X.\ \mathfrak{E} *_{kr} @X\ \varrho)\ has\text{-}sum\ \mathfrak{E} *_{kr} @X\ \varrho)\ ((\lambda x.\ \{x\})\ `\ X)$›
for $\mathfrak{E}$ :: ‹$('a,\ 'b,\ 'c)\ kraus\text{-}family$› **and** X
by *simp*
from *assms*
have *leq*: ‹$\mathfrak{E} *_{kr} @\{x\}\ \varrho \leq \mathfrak{F} *_{kr} @\{x\}\ \varrho$› **for** x
by (*simp add*: *less-eq-kraus-family-def*)
show *?thesis*
using *sum sum* **apply** (*rule has-sum-mono-traceclass*)
using *leq* **by** *fast*
qed

lemma *kf-apply-mono-left*: ‹$\mathfrak{E} \leq \mathfrak{F} \Longrightarrow \varrho \geq 0 \Longrightarrow \mathfrak{E} *_{kr} \varrho \leq \mathfrak{F} *_{kr} \varrho$›
by (*metis kf-apply-on-UNIV kf-apply-on-mono1*)

lemma *kf-apply-mono*:
assumes ‹$\varrho \geq 0$›
assumes ‹$\mathfrak{E} \leq \mathfrak{F}$› **and** ‹$\varrho \leq \sigma$›
shows ‹$\mathfrak{E} *_{kr} \varrho \leq \mathfrak{F} *_{kr} \sigma$›
by (*meson assms*(*1*,*2*,*3*) *basic-trans-rules*(*23*) *kf-apply-mono-left kf-apply-mono-right*)

lemma *kf-apply-on-mono*:
assumes ‹$\varrho \geq 0$›
assumes ‹$\mathfrak{E} \leq \mathfrak{F}$› **and** ‹$X \subseteq Y$› **and** ‹$\varrho \leq \sigma$›
shows ‹$\mathfrak{E} *_{kr} @X\ \varrho \leq \mathfrak{F} *_{kr} @Y\ \sigma$›
apply (*rule order.trans*)
using *assms*(*2*,*1*) **apply** (*rule kf-apply-on-mono1*)
apply (*rule order.trans*)
using *assms*(*3*,*1*) **apply** (*rule kf-apply-on-mono2*)
using *assms*(*4*) **by** (*rule kf-apply-on-mono3*)

lemma *kf-one-dim-is-id*[*simp*]:
fixes $\mathfrak{E}$ :: ‹$('a$::*one-dim*, $'a$::*one-dim*, $'x)\ kraus\text{-}family$›

```
  shows ‹𝔈 =_kr kf-norm 𝔈 *_R kf-id›
proof (rule kf-eq-weakI)
  fix t :: ‹('a, 'a) trace-class›
  have 𝔈1pos[iff]: ‹𝔈 *_kr 1 ≥ 0›
    apply (rule kf-apply-pos)
   by (metis one-cinner-one one-dim-iso-of-one one-dim-scaleC-1 tc-butterfly-pos trace-tc-butterfly
        trace-tc-one-dim-iso)

  have 𝔈t: ‹𝔈 *_kr t = trace-tc t *_C (𝔈 *_kr 1)› if ‹NO-MATCH 1 t›for t
    by (metis kf-apply-scaleC one-dim-scaleC-1 trace-tc-one-dim-iso)
  have ‹kf-bound 𝔈 = norm (𝔈 *_kr 1) *_R id-cblinfun›
  proof (rule cblinfun-cinner-eqI)
    fix h :: 'a
    assume ‹norm h = 1›
    have ‹h •_C kf-bound 𝔈 h = one-dim-iso h * cnj (one-dim-iso h) * one-dim-iso (𝔈 *_kr 1)›
      apply (subst kf-bound-from-map)
     by (simp add: 𝔈t cinner-scaleR-right cblinfun.scaleR-left cdot-square-norm one-dim-tc-butterfly)
    also have 1: ‹... = one-dim-iso (𝔈 *_kr 1)›
     by (smt (verit, best) ‹norm h = 1› cinner-simps(5) cnorm-eq-1 id-apply more-arith-simps(6)
mult.commute
          one-dim-iso-def one-dim-iso-id one-dim-iso-is-of-complex one-dim-scaleC-1)
    also have ‹... = trace-tc (𝔈 *_kr 1)›
      by simp
    also have ‹... = norm (𝔈 *_kr 1)›
      apply (subst norm-tc-pos)
      by (simp-all add: 𝔈1pos)
    also have ‹... = h •_C (norm (𝔈 *_kr 1) *_R id-cblinfun *_V h)›
     by (metis ‹norm h = 1› cblinfun.scaleR-left cinner-commute cinner-scaleR-left cnorm-eq-1
          complex-cnj-complex-of-real id-cblinfun.rep-eq mult.commute mult-cancel-right1)
    finally show ‹h •_C (kf-bound 𝔈 *_V h) = h •_C (norm (𝔈 *_kr 1) *_R id-cblinfun *_V h)›
      by −
  qed
  then have ‹kf-norm 𝔈 = cmod (one-dim-iso (𝔈 *_kr 1))›
    by (simp add: kf-norm-def)
  then have norm: ‹complex-of-real (kf-norm 𝔈) = one-dim-iso (𝔈 *_kr 1)›
  using norm-tc-pos by fastforce

  have ‹(one-dim-iso (𝔈 *_kr t) :: complex) = one-dim-iso t * one-dim-iso (𝔈 *_kr 1)›
 by (metis (mono-tags, lifting) kf-apply-scaleC of-complex-one-dim-iso one-dim-iso-is-of-complex
      one-dim-iso-scaleC one-dim-scaleC-1 scaleC-one-dim-is-times)
  also have ‹... = one-dim-iso t * complex-of-real (kf-norm 𝔈)›
    by (simp add: norm)
  also have ‹... = one-dim-iso (kf-norm 𝔈 *_R t)›
    by (simp add: scaleR-scaleC)
  also have ‹... = one-dim-iso (kf-norm 𝔈 *_R kf-id *_kr t)›
    by (simp add: kf-scale-apply)
  finally show ‹𝔈 *_kr t = kf-norm 𝔈 *_R kf-id *_kr t›
    using one-dim-iso-inj by blast
qed
```

2.6 Mapping and flattening

definition *kf-similar-elements* :: ‹('a::*chilbert-space*, 'b::*chilbert-space*, 'x) *kraus-family* ⇒ ('a ⇒$_{CL}$ 'b) ⇒ ('a ⇒$_{CL}$ 'b × 'x) *set*› **where**
— All elements of the Kraus family that are equal up to rescaling (and belong to the same output)
‹*kf-similar-elements* 𝔈 *E* = {(*F*,*x*) ∈ *Rep-kraus-family* 𝔈. (∃ *r*>0. *E* = *r* ∗$_R$ *F*)}›

definition *kf-element-weight* **where**
— The total weight (norm of the square) of all similar elements
‹*kf-element-weight* 𝔈 *E* = (∑∞(*F*,-)∈*kf-similar-elements* 𝔈 *E*. *norm* (*F*∗ *o*$_{CL}$ *F*))›

lemma *kf-element-weight-geq0*[*simp*]: ‹*kf-element-weight* 𝔈 *E* ≥ 0›
by (*auto intro*!: *infsum-nonneg simp*: *kf-element-weight-def*)

lemma *kf-similar-elements-abs-summable*:
fixes 𝔈 :: ‹('a::*chilbert-space*,'b::*chilbert-space*,'x) *kraus-family*›
shows ‹(λ(*F*,-). *F*∗ *o*$_{CL}$ *F*) *abs-summable-on* (*kf-similar-elements* 𝔈 *E*)›
proof (*cases* ‹*E* = 0›)
case *True*
show *?thesis*
apply (*rule summable-on-cong*[**where** *g*=‹λ-. 0›, *THEN iffD2*])
by (*auto simp*: *kf-similar-elements-def True*)
next
case *False*
then obtain ψ **where** *E*ψ: ‹*E* ψ ≠ 0›
by (*metis cblinfun.zero-left cblinfun-eqI*)
define φ **where** ‹φ = ((*norm E*)^2 / (ψ ∙$_C$ (*E*∗ ∗$_V$ *E* ∗$_V$ ψ))) ∗$_C$ ψ›
have *normFF*: ‹*norm* (*fst Fx*∗ *o*$_{CL}$ *fst Fx*) = ψ ∙$_C$ (*fst Fx*∗ ∗$_V$ *fst Fx* ∗$_V$ φ)›
if ‹*Fx* ∈ *kf-similar-elements* 𝔈 *E*› **for** *Fx*
proof −
define *F* **where** ‹*F* = *fst Fx*›
have ‹∃ *ra*. *x* = (*ra* ∗ *r*) ∗$_R$ *x*› **if** ‹*r* > 0› **for** *r* **and** *x* :: ‹'a ⇒$_{CL}$ 'b›
apply (*rule exI*[*of* - ‹*inverse r*›])
using *that*
by *auto*
with *that* **obtain** *r* **where** *FE*: ‹*F* = *r* ∗$_R$ *E*›
apply *atomize-elim*
by (*auto simp*: *kf-similar-elements-def F-def*)
show ‹*norm* (*F*∗ *o*$_{CL}$ *F*) = ψ ∙$_C$ (*F*∗ ∗$_V$ *F* ∗$_V$ φ)›
by (*simp add*: *False* φ-*def FE cblinfun.scaleR-left cblinfun.scaleR-right*
*cblinfun.scaleC-left cblinfun.scaleC-right cinner-adj-right E*ψ)
qed

have ψφ-*mono*: ‹*mono* (λ*A*. ψ ∙$_C$ (*A* ∗$_V$ φ))›
proof (*rule monoI*)
fix *A B* :: ‹'a ⇒$_{CL}$ 'a›
assume ‹*A* ≤ *B*›
then have ‹*B* − *A* ≥ 0›

by *auto*
then have ‹ψ ∙$_C$ ((B − A) ∗$_V$ ψ) ≥ 0›
by (*simp add*: *cinner-pos-if-pos*)
then have ‹ψ ∙$_C$ ((B − A) ∗$_V$ φ) ≥ 0›
by (*auto intro*!: *mult-nonneg-nonneg*
simp: φ*-def cblinfun.scaleC-right divide-inverse cinner-adj-right power2-eq-square*)
then show ‹ψ ∙$_C$ (A ∗$_V$ φ) ≤ ψ ∙$_C$ (B ∗$_V$ φ)›
by (*simp add*: *cblinfun.diff-left cinner-diff-right*)
qed

have $\psi\varphi$*-linear*: ‹*clinear* (λA. ψ ∙$_C$ (A ∗$_V$ φ))›
by (*auto intro*!: *clinearI simp*: *cblinfun.add-left cinner-add-right*)

from *Rep-kraus-family*[*of* $\mathfrak{E}$]
have ‹*bdd-above* ((λM. ∑(*E*, *x*)∈M. E∗ o_{CL} E) ‘ {M. *finite* M ∧ M ⊆ *Rep-kraus-family* $\mathfrak{E}$})›
by (*simp add*: *kraus-family-def*)
then have ‹*bdd-above* ((λM. ∑(*F*, *x*)∈M. F∗ o_{CL} F) ‘ {M. M ⊆ *kf-similar-elements* $\mathfrak{E}$ E ∧ *finite* M})›
apply (*rule bdd-above-mono2*[*OF - - order.refl*])
by (*auto simp*: *kf-similar-elements-def*)
then have ‹*bdd-above* ((λA. ψ ∙$_C$ (A ∗$_V$ φ)) ‘ (λM. ∑(*F*, *x*)∈M. F∗ o_{CL} F) ‘ {M. M ⊆ *kf-similar-elements* $\mathfrak{E}$ E ∧ *finite* M})›
by (*rule bdd-above-image-mono*[*OF* $\psi\varphi$*-mono*])
then have ‹*bdd-above* ((λM. ψ ∙$_C$ ((∑(*F*, *x*)∈M. F∗ o_{CL} F) ∗$_V$ φ)) ‘ {M. M ⊆ *kf-similar-elements* $\mathfrak{E}$ E ∧ *finite* M})›
by (*simp add*: *image-image*)
then have ‹*bdd-above* ((λM. ∑(*F*, *x*)∈M. ψ ∙$_C$ ((F∗ o_{CL} F) ∗$_V$ φ)) ‘ {M. M ⊆ *kf-similar-elements* $\mathfrak{E}$ E ∧ *finite* M})›
unfolding *case-prod-beta*
by (*subst complex-vector.linear-sum*[*OF* $\psi\varphi$*-linear*, *symmetric*])
then have ‹*bdd-above* ((λM. ∑(*F*, *x*)∈M. *complex-of-real* (*norm* (F∗ o_{CL} F))) ‘ {M. M ⊆ *kf-similar-elements* $\mathfrak{E}$ E ∧ *finite* M})›
apply (*rule bdd-above-mono2*[*OF - subset-refl*])
unfolding *case-prod-unfold*
apply (*subst sum.cong*[*OF refl normFF*])
by *auto*
then have ‹*bdd-above* ((λM. ∑(*F*, *x*)∈M. *norm* (F∗ o_{CL} F)) ‘ {M. M ⊆ *kf-similar-elements* $\mathfrak{E}$ E ∧ *finite* M})›
by (*auto simp add*: *bdd-above-def case-prod-unfold less-eq-complex-def*)
then have ‹(λ(*F*,-). *norm* (F∗ o_{CL} F)) *summable-on* (*kf-similar-elements* $\mathfrak{E}$ E)›
apply (*rule nonneg-bdd-above-summable-on*[*rotated*])
by *auto*
then show ‹(λ(*F*,-). F∗ o_{CL} F) *abs-summable-on kf-similar-elements* $\mathfrak{E}$ E›
by (*simp add*: *case-prod-unfold*)
qed

lemma *kf-similar-elements-kf-operators*:
assumes ‹(*F*,*x*) ∈ *kf-similar-elements* $\mathfrak{E}$ E›

shows ‹$F \in$ *span* (*kf-operators* $\mathfrak{E}$)›
using *assms*
unfolding *kf-similar-elements-def*
apply (*transfer' fixing*: E F x)
by (*metis* (*no-types*, *lifting*) *Product-Type.Collect-case-prodD fst-conv image-eqI span-base*)

lemma *kf-element-weight-neq0*: ‹*kf-element-weight* $\mathfrak{E}$ $E \neq 0$›
if ‹$(E,x) \in$ *Rep-kraus-family* $\mathfrak{E}$› **and** ‹$E \neq 0$›
proof −
have *1*: ‹$(E,\ x) \in$ *kf-similar-elements* $\mathfrak{E}$ E›
by (*auto intro*!: *exI*[**where** *x=1*] *simp*: *kf-similar-elements-def that*)

have ‹*kf-element-weight* $\mathfrak{E}$ $E = (\sum_\infty (F,\ x) \in$ *kf-similar-elements* $\mathfrak{E}$ $E.\ (norm\ F)^2)$›
by (*simp add*: *kf-element-weight-def*)
moreover have ‹$\ldots \geq (\sum_\infty (F,\ x) \in \{(E,x)\}.\ (norm\ F)^2)$› (**is** ‹- ≥ ...›)
apply (*rule infsum-mono-neutral*)
using *kf-similar-elements-abs-summable*
by (*auto intro*!: *1 simp*: *that case-prod-unfold*)
moreover have ‹$\ldots > 0$›
using *that* **by** *simp*
ultimately show *?thesis*
by *auto*
qed

lemma *kf-element-weight-0-left*[*simp*]: ‹*kf-element-weight* 0 $E = 0$›
by (*simp add*: *kf-element-weight-def kf-similar-elements-def zero-kraus-family.rep-eq*)

lemma *kf-element-weight-0-right*[*simp*]: ‹*kf-element-weight* E $0 = 0$›
by (*auto intro*!: *infsum-0 simp add*: *kf-element-weight-def kf-similar-elements-def*)

lemma *kf-element-weight-scale*:
assumes ‹$r > 0$›
shows ‹*kf-element-weight* $\mathfrak{E}$ $(r *_R E) =$ *kf-element-weight* $\mathfrak{E}$ E›
proof −
have [*simp*]: ‹$(\exists r'>0.\ r *_R E = r' *_R F) \longleftrightarrow (\exists r'>0.\ E = r' *_R F)$› **for** F
apply (*rule Ex-iffI*[**where** f=‹$\lambda r'.\ r' /_R r$› **and** g=‹$\lambda r'.\ r *_R r'$›])
apply (*smt* (*verit*, *best*) *assms divideR-right real-scaleR-def scaleR-scaleR scaleR-simps*(*7*) *zero-le-scaleR-iff*)
using *assms* **by** *force*
show *?thesis*
using *assms*
by (*simp add*: *kf-similar-elements-def kf-element-weight-def*)
qed

lemma *kf-element-weight-kf-operators*:
assumes ‹*kf-element-weight* $\mathfrak{E}$ $E \neq 0$›
shows ‹$E \in$ *span* (*kf-operators* $\mathfrak{E}$)›
proof −

from *assms*
have ‹$(\sum_\infty(F,\text{-})\in\{(F,x).\ (F,x)\in \textit{Rep-kraus-family}\ \mathfrak{E}\wedge(\exists r{>}0.\ E=r *_R F)\}.\ \textit{norm}\ (F* o_{CL}\ F))\neq 0$›
by (*simp add*: *kf-element-weight-def kf-similar-elements-def*)
then obtain *F x r* **where** ‹$(F,x)\in \textit{Rep-kraus-family}\ \mathfrak{E}$› **and** ‹$E=r *_R F$›
by (*smt* (*verit*, *ccfv-SIG*) *Product-Type.Collect-case-prodD infsum-0*)
then have ‹$F\in \textit{kf-operators}\ \mathfrak{E}$›
by (*metis fst-conv image-eqI kf-operators.rep-eq*)
with ‹$E=r *_R F$› **show** *?thesis*
by (*simp add*: *span-clauses*)
qed

lemma *kf-map-aux*:
fixes *f* :: ‹$'x\Rightarrow 'y$› **and** $\mathfrak{E}$ :: ‹$('a{::}\textit{chilbert-space}, 'b{::}\textit{chilbert-space}, 'x)\ \textit{kraus-family}$›
defines ‹$B\equiv \textit{kf-bound}\ \mathfrak{E}$›
defines ‹$\textit{filtered}\ y\equiv \textit{kf-filter}\ (\lambda x.\ f\ x=y)\ \mathfrak{E}$›
defines ‹$\textit{flattened}\equiv\{(E,y).\ \textit{norm}\ (E* o_{CL}\ E)=\textit{kf-element-weight}\ (\textit{filtered}\ y)\ E\wedge E\neq 0\}$›
defines ‹$\textit{good}\equiv(\lambda(E,y).\ (\textit{norm}\ E)^2=\textit{kf-element-weight}\ (\textit{filtered}\ y)\ E\wedge E\neq 0)$›
shows ‹$\textit{has-sum-in cweak-operator-topology}\ (\lambda(E,\text{-}).\ E* o_{CL}\ E)\ \textit{flattened}\ B$› (**is** *?has-sum*)
and ‹$\textit{snd}\ `\ (\textit{SIGMA}\ (E,y){:}\textit{Collect good.}\ \textit{kf-similar-elements}\ (\textit{filtered}\ y)\ E)$
$=\{(F,x).\ (F,x)\in \textit{Rep-kraus-family}\ \mathfrak{E}\wedge F\neq 0\}$› (**is** *?snd-sigma*)
and ‹$\textit{inj-on snd}\ (\textit{SIGMA}\ p{:}\textit{Collect good.}\ \textit{kf-similar-elements}\ (\textit{filtered}\ (\textit{snd}\ p))\ (\textit{fst}\ p))$› (**is** *?inj-snd*)
proof −
have *E-inv*: ‹$\textit{kf-element-weight}\ (\textit{filtered}\ y)\ E\neq 0$› **if** ‹$\textit{good}\ (E,y)$› **for** *E y*
using *that* **by** (*auto simp*: *good-def*)

show *snd-sigma*: *?snd-sigma*
proof (*intro Set.set-eqI iffI*)
fix *Fx*
assume *asm*: ‹$Fx\in \textit{snd}\ `\ (\textit{SIGMA}\ (E,y){:}\textit{Collect good.}\ \textit{kf-similar-elements}\ (\textit{filtered}\ y)\ E)$›
obtain *F x* **where** *Fx-def*: ‹$Fx=(F,x)$› **by** *fastforce*
with *asm* **obtain** *E y* **where** *Fx-rel-E*: ‹$(F,x)\in \textit{kf-similar-elements}\ (\textit{filtered}\ y)\ E$› **and** ‹$\textit{good}\ (E,y)$›
by *auto*
then have ‹$(F,x)\in \textit{Rep-kraus-family}\ \mathfrak{E}$›
by (*simp add*: *kf-similar-elements-def filtered-def kf-filter.rep-eq*)
from *Fx-rel-E* **obtain** *r* **where** ‹$E=r *_R F$›
by (*smt* (*verit*) *kf-similar-elements-def mem-Collect-eq prod.sel*(*1*) *split-def*)
with ‹$\textit{good}\ (E,y)$› **have** ‹$F\neq 0$›
by (*simp add*: *good-def*)
with ‹$(F,x)\in \textit{Rep-kraus-family}\ \mathfrak{E}$› **show** ‹$Fx\in\{(F,x).\ (F,x)\in \textit{Rep-kraus-family}\ \mathfrak{E}\wedge F\neq 0\}$›
by (*simp add*: *Fx-def*)
next
fix *Fx*
assume *asm*: ‹$Fx\in\{(F,x).\ (F,x)\in \textit{Rep-kraus-family}\ \mathfrak{E}\wedge F\neq 0\}$›
obtain *F x* **where** *Fx-def*: ‹$Fx=(F,x)$› **by** *fastforce*

```
  from asm have Fx-𝔈: ‹(F, x) ∈ Rep-kraus-family 𝔈› and [simp]: ‹F ≠ 0›
    by (auto simp: Fx-def)
  have weight-fx-F-not0: ‹kf-element-weight (filtered (f x)) F ≠ 0›
    using Fx-𝔈 by (simp-all add: filtered-def kf-filter.rep-eq kf-element-weight-neq0)
  then have weight-fx-F-pos: ‹kf-element-weight (filtered (f x)) F > 0›
    using kf-element-weight-geq0
    by (metis less-eq-real-def)
  define E where ‹E = (sqrt (kf-element-weight (filtered (f x)) F) / norm F) *R F›
  have [simp]: ‹E ≠ 0›
    by (auto intro!: weight-fx-F-not0 simp: E-def)
  have E-F-same: ‹kf-element-weight (filtered (f x)) E = kf-element-weight (filtered (f x)) F›
    by (simp add: E-def kf-element-weight-scale weight-fx-F-pos)
  have ‹good (E, f x)›
    apply (simp add: good-def E-F-same)
    by (simp add: E-def)
  have 1: ‹sqrt (kf-element-weight (filtered (f x)) F) / norm F > 0›
    by (auto intro!: divide-pos-pos weight-fx-F-pos)
  then have ‹(F, x) ∈ kf-similar-elements (filtered (f x)) E›
   by (auto intro!: ‹(F, x) ∈ Rep-kraus-family 𝔈› simp: kf-similar-elements-def E-def ‹F ≠ 0›
       filtered-def kf-filter.rep-eq)
  with ‹good (E,f x)›
  show ‹Fx ∈ snd ' (SIGMA (E, y):Collect good. kf-similar-elements (filtered y) E)›
    by (force intro!: image-eqI[where x=‹((E,()),F,x)›] simp: Fx-def filtered-def)
qed

show inj-snd: ?inj-snd
proof (rule inj-onI)
  fix EFx EFx' :: ‹('a ⇒CL 'b × 'y) × 'a ⇒CL 'b × 'x›
  assume EFx-in: ‹EFx ∈ (SIGMA p:Collect good. kf-similar-elements (filtered (snd p)) (fst
p))›
    and EFx'-in: ‹EFx' ∈ (SIGMA p:Collect good. kf-similar-elements (filtered (snd p)) (fst
p))›
  assume snd-eq: ‹snd EFx = snd EFx'›
  obtain E F x y where [simp]: ‹EFx = ((E,y),F,x)›
    by (metis (full-types) old.unit.exhaust surj-pair)
  obtain E' F' x' y' where [simp]: ‹EFx' = ((E',y'),(F',x'))›
    by (metis (full-types) old.unit.exhaust surj-pair)
  from snd-eq have [simp]: ‹F' = F› and [simp]: ‹x' = x›
    by auto
  from EFx-in have ‹good (E,y)› and F-rel-E: ‹(F, x) ∈ kf-similar-elements (filtered y) E›
    by auto
  from EFx'-in have ‹good (E',y')› and F-rel-E': ‹(F, x) ∈ kf-similar-elements (filtered y')
E'›
    by auto
  from ‹good (E,y)› have ‹E ≠ 0›
    by (simp add: good-def)
  from ‹good (E',y')› have ‹E' ≠ 0›
    by (simp add: good-def)
  from F-rel-E obtain r where ErF: ‹E = r *R F› and ‹r > 0›
```

```
    by (auto intro!: simp: kf-similar-elements-def)
  from F-rel-E' obtain r' where E'rF: ‹E' = r' *R F› and ‹r' > 0›
    by (auto intro!: simp: kf-similar-elements-def)

  from EFx-in have ‹y = f x›
    by (auto intro!: simp: filtered-def kf-similar-elements-def kf-filter.rep-eq)
  moreover from EFx'-in have ‹y' = f x'›
    by (auto intro!: simp: filtered-def kf-similar-elements-def kf-filter.rep-eq)
  ultimately have [simp]: ‹y = y'›
    by simp

  define r'' where ‹r'' = r' / r›
  with E'rF ErF ‹E ≠ 0›
  have E'-E: ‹E' = r'' *R E›
    by auto
  with ‹r' > 0› ‹r > 0› ‹E' ≠ 0›
  have [simp]: ‹r'' > 0›
    by (fastforce simp: r''-def)
  from E'-E have ‹kf-element-weight (filtered y') E' = kf-element-weight (filtered y) E›
    by (simp add: kf-element-weight-scale)
  with ‹good (E,y)› ‹good (E',y')› have ‹(norm E')² = (norm E)²›
    by (auto intro!: simp: good-def)
  with ‹E' = r'' *R E›
  have ‹E' = E›
    using ‹0 < r''› by force
  then show ‹EFx = EFx'›
    by simp
 qed

 show ?has-sum
 proof (unfold has-sum-in-cweak-operator-topology-pointwise, intro allI)
  fix ψ φ :: 'a
  define B' where ‹B' = ψ ·C B φ›
  define normal where ‹normal E y = E /R sqrt (kf-element-weight (filtered y) E)› for E y
  have ‹has-sum-in cweak-operator-topology (λ(F,x). F* oCL F) (Rep-kraus-family 𝔈) B›
    using B-def kf-bound-has-sum by blast
  then have ‹((λ(F,x). ψ ·C (F* oCL F) φ) has-sum B') (Rep-kraus-family 𝔈)›
    by (simp add: B'-def has-sum-in-cweak-operator-topology-pointwise case-prod-unfold)
  then have ‹((λ(F,x). ψ ·C (F* oCL F) φ) has-sum B') {(F,x)∈Rep-kraus-family 𝔈. F ≠
0}›
    apply (rule has-sum-cong-neutral[THEN iffD2, rotated -1])
    by (auto simp: zero-cblinfun-wot-def)
  then have ‹((λ(F,x). ψ ·C (F* oCL F) φ) has-sum B')
        (snd ‘ (SIGMA (E,y):Collect good. kf-similar-elements (filtered y) E))›
    by (simp add: snd-sigma)
  then have ‹((λ((E,x), (F,y)). ψ ·C (F* oCL F) φ) has-sum B')
     (SIGMA (E,y):Collect good. kf-similar-elements (filtered y) E)›
    apply (subst (asm) has-sum-reindex)
    by (auto intro!: inj-on-def inj-snd simp: o-def case-prod-unfold)
```

then have *sum1*: ‹((λ((E,y), (F,x)). (*norm F*)^2 $*_R$ (ψ $\cdot_C$ (*normal E y*$*$ o_{CL} *normal E y*) φ)) *has-sum B'*)
(*SIGMA* (E,y):*Collect good*. *kf-similar-elements* (*filtered y*) *E*)›
apply (*rule has-sum-cong*[*THEN iffD2*, *rotated*])
apply (*subgoal-tac* ‹⋀*b aa ba r*.
(*r* $*$ *norm aa*)^2 = *kf-element-weight* (*filtered b*) (*complex-of-real r* $*_C$ *aa*) ⟹
aa ≠ *0* ⟹
0 < *r* ⟹
(*complex-of-real* (*norm aa*))^2 $*$
(*inverse* (*complex-of-real* (*sqrt* (*kf-element-weight* (*filtered b*) (*complex-of-real r* $*_C$ *aa*))))
$*$
(*inverse* (*complex-of-real* (*sqrt* (*kf-element-weight* (*filtered b*) (*complex-of-real r* $*_C$ *aa*))))
$*$
(*complex-of-real r* $*$ *complex-of-real r*))) ≠
1 ⟹
is-orthogonal ψ (*aa*$*$ $*_V$ *aa* $*_V$ φ)›)
apply (*auto intro*!: *simp*: *good-def normal-def sandwich-tc-scaleR-left power-inverse real-sqrt-pow2 E-inv*
kf-similar-elements-def kf-element-weight-scale
cblinfun.scaleC-left cblinfun.scaleC-right cinner-scaleC-right scaleR-scaleC)[1]
by (*smt* (*verit*) *Extra-Ordered-Fields.mult-sign-intros*(*5*) *Extra-Ordered-Fields.sign-simps*(*5*) *inverse-eq-iff-eq left-inverse more-arith-simps*(*11*) *of-real-eq-0-iff of-real-mult power2-eq-square power-inverse real-inv-sqrt-pow2 zero-less-norm-iff*)
then have ‹((λ(E,y). $\sum_\infty$(F, x)∈*kf-similar-elements* (*filtered y*) *E*.
(*norm F*)^2 $*_R$ (ψ $\cdot_C$ (*normal E y*$*$ o_{CL} *normal E y*) φ)) *has-sum B'*) (*Collect good*)›
by (*auto intro*!: *has-sum-Sigma'-banach simp add*: *case-prod-unfold*)
then have ‹((λ(E,y). ($\sum_\infty$(F, x)∈*kf-similar-elements* (*filtered y*) *E*.
(*norm F*)^2) $*_R$ (ψ $\cdot_C$ (*normal E y*$*$ o_{CL} *normal E y*) φ))
has-sum B') (*Collect good*)›
apply (*rule has-sum-cong*[*THEN iffD1*, *rotated*])
apply *simp*
by (*smt* (*verit*) *Complex-Vector-Spaces.infsum-scaleR-left cblinfun.scaleR-left infsum-cong split-def*)
then have ‹((λ(E,y). *kf-element-weight* (*filtered y*) *E* $*_R$
(ψ $\cdot_C$ (*normal E y*$*$ o_{CL} *normal E y*) φ)) *has-sum B'*) (*Collect good*)›
by (*simp add*: *kf-element-weight-def*)
then have ‹((λ(E,-). ψ $\cdot_C$ (*E*$*$ o_{CL} *E*) φ) *has-sum B'*) (*Collect good*)›
apply (*rule has-sum-cong*[*THEN iffD1*, *rotated*])
by (*auto intro*!: *field-class.field-inverse*
simp add: *normal-def sandwich-tc-scaleR-left power-inverse real-sqrt-pow2 E-inv*
cblinfun.scaleR-left scaleR-scaleC
simp flip: *inverse-mult-distrib semigroup-mult.mult-assoc of-real-mult*
split!: *prod.split*)
then have ‹((λ(E,-). ψ $\cdot_C$ (*E*$*$ o_{CL} *E*) φ) *has-sum B'*) *flattened*›
by (*simp add*: *flattened-def good-def*)
then show ‹((λ*x*. ψ $\cdot_C$ ((*case x of* (*E*, *uu*-) ⇒ *E*$*$ o_{CL} *E*) $*_V$ φ)) *has-sum B'*) *flattened*›
by (*simp add*: *case-prod-unfold*)
qed
qed

lift-definition *kf-map* :: ‹('*x* ⇒ '*y*) ⇒ ('*a*::*chilbert-space*, '*b*::*chilbert-space*, '*x*) *kraus-family* ⇒ ('*a*, '*b*, '*y*) *kraus-family*› **is**
‹λ*f* 𝔈. {(*E*, *y*). *norm* (*E*∗ *o*$_{CL}$ *E*) = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) 𝔈) *E* ∧ *E* ≠ *0*}›
proof (*rename-tac f* 𝔈)
fix *f* :: ‹'*x* ⇒ '*y*› **and** 𝔈 :: ‹('*a*, '*b*, '*x*) *kraus-family*›
define *filtered flattened B*
where ‹*filtered y* = *kf-filter* (λ*x*. *f x* = *y*) 𝔈›
and ‹*flattened* = {(*E*, *y*). *norm* (*E*∗ *o*$_{CL}$ *E*) = *kf-element-weight* (*filtered y*) *E* ∧ *E* ≠ *0*}›
and ‹*B* = *kf-bound* 𝔈›
for *y*

from *kf-map-aux*[*of f* 𝔈]
have *bound-has-sum*: ‹*has-sum-in cweak-operator-topology* (λ(*E*,-). *E*∗ *o*$_{CL}$ *E*) *flattened B*›
by (*simp-all add*: *filtered-def flattened-def B-def*)

have *nonzero*: ‹*0* ∉ *fst* ‘ *flattened*›
by (*auto intro*!: *simp*: *flattened-def*)

from *bound-has-sum nonzero* **show** ‹*flattened* ∈ *Collect kraus-family*›
by (*auto simp*: *kraus-family-iff-summable summable-on-in-def*)
qed

lemma
fixes 𝔈 :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*x*) *kraus-family*›
shows *kf-apply-map*[*simp*]: ‹*kf-apply* (*kf-map f* 𝔈) = *kf-apply* 𝔈›
and *kf-map-bound*: ‹*kf-bound* (*kf-map f* 𝔈) = *kf-bound* 𝔈›
and *kf-map-norm*[*simp*]: ‹*kf-norm* (*kf-map f* 𝔈) = *kf-norm* 𝔈›
proof (*rule ext*)
fix ϱ :: ‹('*a*, '*a*) *trace-class*›
define *filtered good flattened B normal* σ
where ‹*filtered y* = *kf-filter* (λ*x*. *f x* = *y*) 𝔈›
and ‹*good* = (λ(*E*,*y*). (*norm E*)^2 = *kf-element-weight* (*filtered y*) *E* ∧ *E* ≠ *0*)›
and ‹*flattened* = {(*E*, *y*). *norm* (*E*∗ *o*$_{CL}$ *E*) = *kf-element-weight* (*filtered y*) *E* ∧ *E* ≠ *0*}›
and ‹*B* = *kf-bound* 𝔈›
and ‹*normal E y* = *E* /$_R$ *sqrt* (*kf-element-weight* (*filtered y*) *E*)›
and ‹σ = 𝔈 ∗$_{kr}$ ϱ›
for *E y*
have *E-inv*: ‹*kf-element-weight* (*filtered y*) *E* ≠ *0*› **if** ‹*good* (*E*,*y*)› **for** *E y*
using *that* **by** (*auto simp*: *good-def*)

from *kf-map-aux*[*of f* 𝔈]
have *snd-sigma*: ‹*snd* ‘ (*SIGMA* (*E*, *y*):*Collect good*. *kf-similar-elements* (*filtered y*) *E*)
= {(*F*, *x*). (*F*, *x*) ∈ *Rep-kraus-family* 𝔈 ∧ *F* ≠ *0*}›
and *inj-snd*: ‹*inj-on snd* (*SIGMA p*:*Collect good*. *kf-similar-elements* (*filtered* (*snd p*)) (*fst p*))›
and *bound-has-sum*: ‹*has-sum-in cweak-operator-topology* (λ(*E*,-). *E*∗ *o*$_{CL}$ *E*) *flattened B*›

by (*simp-all add*: *good-def filtered-def flattened-def B-def*)

show ‹*kf-apply* (*kf-map f* $\mathfrak{E}$) $\varrho = \sigma$›
proof −
have ‹(λ(*F*,*x*). *sandwich-tc F* ϱ) *summable-on Rep-kraus-family* $\mathfrak{E}$›
using *kf-apply-summable* **by** (*simp add*: *case-prod-unfold*)
then have ‹((λ(*F*,*x*). *sandwich-tc F* ϱ) *has-sum* σ) (*Rep-kraus-family* $\mathfrak{E}$)›
by (*simp add*: σ*-def kf-apply-def split-def*)
then have ‹((λ(*F*,*x*). *sandwich-tc F* ϱ) *has-sum* σ) {(*F*,*x*)∈*Rep-kraus-family* $\mathfrak{E}$. *F* ≠ *0*}›
apply (*rule has-sum-cong-neutral*[*THEN iffD2*, *rotated* −*1*])
by *auto*
then have ‹((λ(*F*,*x*). *sandwich-tc F* ϱ) *has-sum* σ)
(*snd* ‘ (*SIGMA* (*E*,*y*):*Collect good*. *kf-similar-elements* (*filtered y*) *E*))›
by (*simp add*: *snd-sigma*)
then have ‹((λ((*E*,*x*), (*F*,*y*)). *sandwich-tc F* ϱ) *has-sum* σ)
(*SIGMA* (*E*,*y*):*Collect good*. *kf-similar-elements* (*filtered y*) *E*)›
apply (*subst* (*asm*) *has-sum-reindex*)
by (*auto intro*!: *inj-on-def inj-snd simp*: *o-def case-prod-unfold*)
then have *sum1*: ‹((λ((*E*,*y*), (*F*,-)). (*norm F*)^2 $*_R$ *sandwich-tc* (*normal E y*) ϱ) *has-sum* σ)
(*SIGMA* (*E*,*y*):*Collect good*. *kf-similar-elements* (*filtered y*) *E*)›
apply (*rule has-sum-cong*[*THEN iffD2*, *rotated*])
apply (*subgoal-tac* ‹⋀*b a r*. (*r* ∗ *norm a*)^2 = *kf-element-weight* (*filtered b*) *a* ⟹
a ≠ *0* ⟹ *0* < *r* ⟹ ((*norm a*)^2 ∗ *inverse* (*kf-element-weight* (*filtered b*) *a*) ∗ *r*^{2}) $*_R$ *sandwich-tc a* ϱ = *sandwich-tc a* ϱ ›)
apply (*auto intro*!: *real-vector.scale-one simp*: *good-def normal-def sandwich-tc-scaleR-left power-inverse real-sqrt-pow2 E-inv*
kf-similar-elements-def kf-element-weight-scale)[*1*]
by (*metis* (*no-types*, *opaque-lifting*) *Extra-Ordered-Fields.sign-simps*(*5*) *linorder-not-less more-arith-simps*(*11*) *mult-eq-0-iff norm-le-zero-iff order.refl power2-eq-square right-inverse scale-one*)
then have ‹((λ(*E*,*y*). ∑$_\infty$(*F*, *x*)∈*kf-similar-elements* (*filtered y*) *E*.
(*norm F*)^2 $*_R$ *sandwich-tc* (*normal E y*) ϱ) *has-sum* σ) (*Collect good*)›
by (*auto intro*!: *has-sum-Sigma'-banach simp add*: *case-prod-unfold*)
then have ‹((λ(*E*,*y*). (∑$_\infty$(*F*, *x*)∈*kf-similar-elements* (*filtered y*) *E*. (*norm F*)^2) $*_R$ *sandwich-tc* (*normal E y*) ϱ)
has-sum σ) (*Collect good*)›
apply (*rule has-sum-cong*[*THEN iffD1*, *rotated*])
apply *simp*
by (*smt* (*verit*) *Complex-Vector-Spaces.infsum-scaleR-left cblinfun.scaleR-left infsum-cong split-def*)
then have ‹((λ(*E*,*y*). *kf-element-weight* (*filtered y*) *E* $*_R$ *sandwich-tc* (*normal E y*) ϱ) *has-sum* σ) (*Collect good*)›
by (*simp add*: *kf-element-weight-def*)
then have ‹((λ(*E*,-). *sandwich-tc E* ϱ) *has-sum* σ) (*Collect good*)›
apply (*rule has-sum-cong*[*THEN iffD1*, *rotated*])
by (*auto intro*!: *simp*: *normal-def sandwich-tc-scaleR-left power-inverse real-sqrt-pow2 E-inv*)
then have ‹((λ(*E*,-). *sandwich-tc E* ϱ) *has-sum* σ) *flattened*›
by (*simp add*: *flattened-def good-def*)
then show ‹*kf-map f* $\mathfrak{E}$ $*_{kr}$ ϱ = σ›
by (*simp add*: *kf-apply.rep-eq kf-map.rep-eq flattened-def*

case-prod-unfold infsumI filtered-def)
qed

from *bound-has-sum* **show** *bound*: ‹*kf-bound* (*kf-map f* 𝔈) = *B*›
apply (*simp add*: *kf-bound-def flattened-def kf-map.rep-eq B-def filtered-def*)
using *has-sum-in-infsum-in has-sum-in-unique hausdorff-cweak-operator-topology summable-on-in-def*
by *blast*

from *bound* **show** ‹*kf-norm* (*kf-map f* 𝔈) = *kf-norm* 𝔈›
by (*simp add*: *kf-norm-def B-def*)
qed

abbreviation ‹*kf-flatten* ≡ *kf-map* (λ-. ())›

Like *kf-map*, but with a much simpler definition. However, only makes sense for injective functions.

lift-definition *kf-map-inj* :: ‹('*x* ⇒ '*y*) ⇒ ('*a*::*chilbert-space*, '*b*::*chilbert-space*, '*x*) *kraus-family* ⇒ ('*a*, '*b*, '*y*) *kraus-family*› **is**
‹λ*f* 𝔈. (λ(*E*,*x*). (*E*, *f x*)) ‘ 𝔈›
proof (*rule CollectI*, *rule kraus-familyI*, *rename-tac f* 𝔈)
fix *f* :: ‹'*x* ⇒ '*y*› **and** 𝔈 :: ‹('*a* ⇒$_{CL}$ '*b* × '*x*) *set*›
assume ‹𝔈 ∈ *Collect kraus-family*›
then obtain *B* **where** *B*: ‹(∑ (*E*, *x*)∈*F*. *E*∗ *o*$_{CL}$ *E*) ≤ *B*› **if** ‹*F* ∈ {*F*. *finite F* ∧ *F* ⊆ 𝔈}› **for** *F*
by (*auto simp*: *kraus-family-def bdd-above-def*)
show ‹*bdd-above* ((λ*F*. ∑ (*E*, *x*)∈*F*. *E*∗ *o*$_{CL}$ *E*) ‘ {*F*. *finite F* ∧ *F* ⊆ (λ(*E*, *x*). (*E*, *f x*)) ‘ 𝔈})›
proof (*rule bdd-aboveI2*)
fix *F* **assume** ‹*F* ∈ {*F*. *finite F* ∧ *F* ⊆ (λ(*E*, *x*). (*E*, *f x*)) ‘ 𝔈}›
then obtain *F*′ **where** ‹*finite F*′› **and** ‹*F*′ ⊆ 𝔈› **and** *F-F*′: ‹*F* = (λ(*E*, *x*). (*E*, *f x*)) ‘ *F*′›
and *inj*: ‹*inj-on* (λ(*E*, *x*). (*E*, *f x*)) *F*′›
by (*metis* (*no-types*, *lifting*) *finite-imageD mem-Collect-eq subset-image-inj*)
have ‹(∑ (*E*, *x*)∈*F*′. *E*∗ *o*$_{CL}$ *E*) ≤ *B*›
by (*auto intro*!: *B* ‹*finite F*′› ‹*F*′ ⊆ 𝔈›)
moreover have ‹(∑ (*E*, *x*)∈*F*. *E*∗ *o*$_{CL}$ *E*) ≤ (∑ (*E*, *x*)∈*F*′. *E*∗ *o*$_{CL}$ *E*)›
apply (*simp add*: *F-F*′ *inj sum.reindex*)
by (*simp add*: *case-prod-beta*)
ultimately show ‹(∑ (*E*, *x*)∈*F*. *E*∗ *o*$_{CL}$ *E*) ≤ *B*›
by *simp*
qed
from ‹𝔈 ∈ *Collect kraus-family*›
show ‹0 ∉ *fst* ‘ (λ(*E*,*x*). (*E*, *f x*)) ‘ 𝔈›
by (*force simp*: *kraus-family-def*)
qed

lemma *kf-element-weight-map-inj*:
assumes ‹*inj-on f* (*kf-domain* 𝔈)›
shows ‹*kf-element-weight* (*kf-map-inj f* 𝔈) *E* = *kf-element-weight* 𝔈 *E*›
proof −

wlog ‹$E \neq 0$›
using *negation* **by** *simp*
have *inj2*: ‹*inj-on* ($\lambda(E, x).\ (E, f\ x)$) {$(F, x).\ (F, x) \in$ *Rep-kraus-family* $\mathfrak{E} \wedge (\exists\, r{>}0.\ E = r *_R F)$}›
proof (*rule inj-onI*)
fix *Fy Gx* :: ‹$'c \Rightarrow_{CL} 'd \times 'a$›
obtain *F y G x* **where** [*simp*]: ‹$Fy = (F,y)$› ‹$Gx = (G,x)$›
by (*auto simp*: *prod-eq-iff*)
assume ‹$Fy \in$ {$(F,\ y).\ (F,\ y) \in$ *Rep-kraus-family* $\mathfrak{E} \wedge (\exists\, r{>}0.\ E = r *_R F)$}› **and** ‹$Gx \in$ {$(G,\ x).\ (G,\ x) \in$ *Rep-kraus-family* $\mathfrak{E} \wedge (\exists\, r{>}0.\ E = r *_R G)$}›
then have *Fy*-$\mathfrak{E}$: ‹$(F,\ y) \in$ *Rep-kraus-family* $\mathfrak{E}$› **and** *ErF*: ‹$\exists\, r{>}0.\ E = r *_R F$› **and** *Gx*-$\mathfrak{E}$: ‹$(G,\ x) \in$ *Rep-kraus-family* $\mathfrak{E}$› **and** *ErG*: ‹$\exists\, r{>}0.\ E = r *_R G$›
by *auto*
from *ErF* ‹$E \neq 0$› **have** ‹$F \neq 0$›
by *auto*
with *Fy*-$\mathfrak{E}$ **have** ‹$y \in$ *kf-domain* $\mathfrak{E}$›
by (*force simp add*: *kf-domain.rep-eq*)
from *ErG* ‹$E \neq 0$› **have** ‹$G \neq 0$›
by *auto*
with *Gx*-$\mathfrak{E}$ **have** ‹$x \in$ *kf-domain* $\mathfrak{E}$›
by (*force simp add*: *kf-domain.rep-eq*)
assume ‹(*case Fy of* $(F,\ y) \Rightarrow (F, f\ y)$) = (*case Gx of* $(G,\ x) \Rightarrow (G, f\ x)$)›
then have [*simp*]: ‹$F = G$› **and** ‹$f\ y = f\ x$›
by *auto*
with *assms* ‹$x \in$ *kf-domain* $\mathfrak{E}$› ‹$y \in$ *kf-domain* $\mathfrak{E}$›
have ‹$y = x$›
by (*simp add*: *inj-onD*)
then show ‹$Fy = Gx$›
by *simp*
qed

have ‹*kf-element-weight* (*kf-map-inj f* $\mathfrak{E}$) *E*
$= (\sum_\infty (F,\ \text{-}) \in \{(F,\ x).\ (F,\ x) \in (\lambda(E,x).\ (E,\ f\ x))\ `\ \textit{Rep-kraus-family}\ \mathfrak{E} \wedge (\exists\, r{>}0.\ E = r *_R F)\}.\ (\textit{norm}\ F)^2)$›
by (*simp add*: *kf-element-weight-def assms kf-similar-elements-def kf-map-inj.rep-eq*)
also have ‹$\ldots = (\sum_\infty (F,\ \text{-}) \in (\lambda(E,x).\ (E,\ f\ x))\ `\ \{(F,\ x).\ (F,\ x) \in$ *Rep-kraus-family* $\mathfrak{E} \wedge (\exists\, r{>}0.\ E = r *_R F)\}.\ (\textit{norm}\ F)^2)$›
apply (*rule arg-cong2*[**where** *f*=*infsum*])
by *auto*
also have ‹$\ldots = (\sum_\infty (F,\ \text{-}) \in \{(F,\ x).\ (F,\ x) \in$ *Rep-kraus-family* $\mathfrak{E} \wedge (\exists\, r{>}0.\ E = r *_R F)\}.\ (\textit{norm}\ F)^2)$›
apply (*subst infsum-reindex*)
apply (*rule inj2*)
using *assms* **by** (*simp add*: *o-def case-prod-unfold*)
also have ‹$\ldots$ = *kf-element-weight* $\mathfrak{E}$ *E*›
by (*simp add*: *kf-element-weight-def assms kf-similar-elements-def*)
finally show *?thesis*
by −
qed

lemma *kf-eq-weak-kf-map-left*: ‹*kf-map f F* $=_{kr}$ *G*› **if** ‹*F* $=_{kr}$ *G*›
using *that* **by** (*simp add*: *kf-eq-weak-def kf-apply-map*)

lemma *kf-eq-weak-kf-map-right*: ‹*F* $=_{kr}$ *kf-map f G*› **if** ‹*F* $=_{kr}$ *G*›
using *that* **by** (*simp add*: *kf-eq-weak-def kf-apply-map*)

lemma *kf-filter-map*:
fixes 𝔈 :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*x*) *kraus-family*›
shows ‹*kf-filter P* (*kf-map f* 𝔈) = *kf-map f* (*kf-filter* (λ*x*. *P* (*f x*)) 𝔈)›
proof −
have ‹(*E*,*x*) ∈ *Set.filter* (λ(*E*, *y*). *P y*) {(*E*, *y*). *norm* (*E*∗ o_{CL} *E*) = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) 𝔈) *E* ∧ *E* ≠ *0*}
⟷ (*E*,*x*) ∈ {(*E*, *y*). *norm* (*E*∗ o_{CL} *E*) = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) (*kf-filter* (λ*x*. *P* (*f x*)) 𝔈)) *E* ∧ *E* ≠ *0*}›
for *E x* **and** 𝔈 :: ‹('*a*, '*b*, '*x*) *kraus-family*›
proof (*cases* ‹*P x*›)
case *True*
then show *?thesis*
apply (*simp add*: *kf-filter-twice*)
by (*metis* (*mono-tags*, *lifting*) *kf-filter-cong-eq*)
next
case *False*
then have [*simp*]: ‹(λ*z*. *f z* = *x* ∧ *P* (*f z*)) = (λ-. *False*)›
by *auto*
from *False* **show** *?thesis*
apply (*simp add*: *kf-filter-twice*)
by (*smt* (*verit*, *ccfv-SIG*) *kf-element-weight-0-left kf-filter-cong-eq kf-filter-false norm-eq-zero zero-eq-power2*)
qed
then show *?thesis*
apply (*transfer' fixing*: *P f*)
by *blast*
qed

lemma *kf-filter-map-inj*:
fixes 𝔈 :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*x*) *kraus-family*›
shows ‹*kf-filter P* (*kf-map-inj f* 𝔈) = *kf-map-inj f* (*kf-filter* (λ*x*. *P* (*f x*)) 𝔈)›
apply (*transfer' fixing*: *P f*)
by (*force simp*: *case-prod-beta image-iff*)

lemma *kf-map-kf-map-inj-comp*:
assumes ‹*inj-on f* (*kf-domain* 𝔈)›
shows ‹*kf-map g* (*kf-map-inj f* 𝔈) = *kf-map* (*g o f*) 𝔈›
proof (*transfer' fixing*: *f g* 𝔈)
from *assms*
have ‹*inj-on f* (*kf-domain* (*kf-filter* (λ*x*. *g* (*f x*) = *y*) 𝔈))› **for** *y*
apply (*rule inj-on-subset*) **by** *force*
then show ‹{(*E*, *y*). *norm* (*E*∗ o_{CL} *E*) = *kf-element-weight* (*kf-filter* (λ*x*. *g x* = *y*) (*kf-map-inj*

$f\ \mathfrak{E})\ E \wedge E \neq 0\} =$
$\{(E,\ y).\ norm\ (E*\ o_{CL}\ E) = kf\text{-}element\text{-}weight\ (kf\text{-}filter\ (\lambda x.\ (g \circ f)\ x = y)\ \mathfrak{E})\ E \wedge E \neq 0\}$›
by (*simp add*: *kf-filter-map-inj kf-element-weight-map-inj assms*)
qed

lemma *kf-element-weight-eqweak0*:
assumes ‹$\mathfrak{E} =_{kr}\ 0$›
shows ‹*kf-element-weight* $\mathfrak{E}\ E = 0$›
apply (*subst kf-eq-0-iff-eq-0*[*THEN iffD1*])
using *assms* **by** *auto*

lemma *kf-map-inj-kf-map-comp*:
assumes ‹*inj-on g* (*f* ‘ *kf-domain* $\mathfrak{E}$)›
shows ‹*kf-map-inj g* (*kf-map f* $\mathfrak{E}$) = *kf-map* (*g o f*) $\mathfrak{E}$›
proof (*transfer′ fixing*: *f g* $\mathfrak{E}$, *rule Set.set-eqI*)
fix *Ex* :: ‹$'d \Rightarrow_{CL}\ 'e \times\ 'b$›
obtain *E x* **where** [*simp*]: ‹*Ex* = (*E*,*x*)›
by (*auto simp*: *prod-eq-iff*)
have ‹*Ex* ∈ (λ(*E*, *x*). (*E*, *g x*)) ‘ {(*E*, *y*). *norm* (*E** o_{CL} *E*) = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) $\mathfrak{E}$) *E* ∧ *E* ≠ *0*} ⟷
(∃ *y*. *x* = *g y* ∧ (*norm E*)^2 = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) $\mathfrak{E}$) *E*) ∧ *E* ≠ *0*›
by *auto*
also have ‹... ⟷ (*norm E*)^2 = *kf-element-weight* (*kf-filter* (λ*z*. *g* (*f z*) = *x*) $\mathfrak{E}$) *E* ∧ *E* ≠ *0*›
proof (*rule iffI*)
assume *asm*: ‹(∃ *y*. *x* = *g y* ∧ (*norm E*)^2 = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) $\mathfrak{E}$) *E*) ∧ *E* ≠ *0*›
then obtain *y* **where** *xy*: ‹*x* = *g y*› **and** *weight*: ‹(*norm E*)^2 = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) $\mathfrak{E}$) *E*›
by *auto*
from *asm* **have** ‹*E* ≠ *0*›
by *simp*
with *weight* **have** ‹¬ *kf-filter* (λ*x*. *f x* = *y*) $\mathfrak{E} =_{kr}\ 0$›
using *kf-element-weight-eqweak0* **by** *fastforce*
then have ‹¬ *kf-filter* (λ*x*. *f x* = *y* ∧ *x* ∈ *kf-domain* $\mathfrak{E}$) $\mathfrak{E} =_{kr}\ 0$›
by (*smt* (*verit*, *del-insts*) *kf-eq-0-iff-eq-0 kf-filter-cong-eq*)
then have ‹(λ*x*. *f x* = *y* ∧ *x* ∈ *kf-domain* $\mathfrak{E}$) ≠ (λ*z*. *False*)›
using *kf-filter-false*[*of* $\mathfrak{E}$]
by (*metis kf-eq-weak-refl0*)
then have *yf*$\mathfrak{E}$: ‹*y* ∈ *f* ‘ *kf-domain* $\mathfrak{E}$›
by *fast*
have ‹*kf-element-weight* ((*kf-filter* (λ*x*. *f x* = *y*) $\mathfrak{E}$)) *E* = *kf-element-weight* ((*kf-filter* (λ*z*. *g* (*f z*) = *x*) $\mathfrak{E}$)) *E*›
apply (*rule arg-cong2*[**where** *f*=*kf-element-weight*, *OF - refl*])
apply (*rule kf-filter-cong-eq*[*OF refl*])
using *yf*$\mathfrak{E}$ *assms xy*
by (*meson image-eqI inj-onD*)

then
have ‹*kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) 𝔈) *E* = *kf-element-weight* (*kf-filter* (λ*z*. *g* (*f z*) = *x*) 𝔈) *E*›
by *simp*
with *weight* ‹*E* ≠ *0*›
show ‹(*norm E*)² = *kf-element-weight* (*kf-filter* (λ*z*. *g* (*f z*) = *x*) 𝔈) *E* ∧ *E* ≠ *0*›
by *simp*
next
assume *asm*: ‹(*norm E*)² = *kf-element-weight* (*kf-filter* (λ*z*. *g* (*f z*) = *x*) 𝔈) *E* ∧ *E* ≠ *0*›
then have ‹*E* ≠ *0*›
by *simp*
from *asm* **have** ‹¬ *kf-filter* (λ*z*. *g* (*f z*) = *x*) 𝔈 =$_{kr}$ *0*›
using *kf-element-weight-eqweak0* **by** *fastforce*
then have ‹¬ *kf-filter* (λ*z*. *g* (*f z*) = *x* ∧ *z* ∈ *kf-domain* 𝔈) 𝔈 =$_{kr}$ *0*›
by (*smt* (*verit*, *ccfv-threshold*) *kf-eq-0-iff-eq-0 kf-filter-cong-eq*)
then have ‹(λ*z*. *g* (*f z*) = *x* ∧ *z* ∈ *kf-domain* 𝔈) ≠ (λ*z*. *False*)›
using *kf-filter-false*[*of* 𝔈]
by (*metis kf-eq-weak-refl0*)
then obtain *z* **where** ‹*z* ∈ *kf-domain* 𝔈› **and** *gfz*: ‹*g* (*f z*) = *x*›
using *kf-filter-false*[*of* 𝔈]
by *auto*
have ‹*kf-element-weight* ((*kf-filter* (λ*x*. *f x* = *f z*) 𝔈)) *E* = *kf-element-weight* ((*kf-filter* (λ*z*. *g* (*f z*) = *x*) 𝔈)) *E*›
apply (*rule arg-cong2*[**where** *f*=*kf-element-weight*, *OF - refl*])
apply (*rule kf-filter-cong-eq*[*OF refl*])
using *assms gfz* ‹*z* ∈ *kf-domain* 𝔈›
by (*metis image-eqI inj-onD*)
with *asm* ‹*E* ≠ *0*›
show ‹(∃ *y*. *x* = *g y* ∧ (*norm E*)² = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) 𝔈) *E*) ∧ *E* ≠ *0*›
by (*auto intro*!: *exI*[*of* - ‹*f z*›] *simp flip*: *gfz*)
qed
also have ‹... ⟷ *Ex* ∈ {(*E*, *y*). *norm* (*E*∗ o_{CL} *E*) = *kf-element-weight* (*kf-filter* (λ*x*. (*g* ∘ *f*) *x* = *y*) 𝔈) *E* ∧ *E* ≠ *0*}›
by *simp*
finally show ‹*Ex* ∈ (λ(*E*, *x*). (*E*, *g x*)) ‘ {(*E*, *y*). *norm* (*E*∗ o_{CL} *E*) = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *y*) 𝔈) *E* ∧ *E* ≠ *0*} ⟷
Ex ∈ {(*E*, *y*). *norm* (*E*∗ o_{CL} *E*) = *kf-element-weight* (*kf-filter* (λ*x*. (*g* ∘ *f*) *x* = *y*) 𝔈) *E* ∧ *E* ≠ *0*}›
by −
qed

lemma *kf-apply-map-inj*[*simp*]:
assumes ‹*inj-on f* (*kf-domain* 𝔈)›
shows ‹*kf-map-inj f* 𝔈 ∗$_{kr}$ ϱ = 𝔈 ∗$_{kr}$ ϱ›
proof −
define *EE* **where** ‹*EE* = *Set.filter* (λ(*E*,*x*). *E*≠*0*) (*Rep-kraus-family* 𝔈)›
have ‹*kf-map-inj f* 𝔈 ∗$_{kr}$ ϱ = (∑$_\infty$*E*∈(λ(*E*, *x*). (*E*, *f x*)) ‘ *Rep-kraus-family* 𝔈. *sandwich-tc* (*fst E*) ϱ)›

by (*simp add*: *kf-apply.rep-eq kf-map-inj.rep-eq*)
also have ‹... = $(\sum_\infty E{\in}(\lambda(E,\ x).\ (E,\ f\ x))$ ‘ *EE*. *sandwich-tc* (*fst E*) ϱ)›
apply (*rule infsum-cong-neutral*)
by (*force simp*: *EE-def*)+
also have ‹... = *infsum* ((λ*E*. *sandwich-tc* (*fst E*) ϱ) ∘ (λ(*E*, *x*). (*E*, *f x*))) *EE*›
apply (*rule infsum-reindex*)
apply (*subgoal-tac* ‹$\bigwedge$*aa ba b*. ∀ *x*∈*Rep-kraus-family* $\mathfrak{E}$. ∀ *y*∈*Rep-kraus-family* $\mathfrak{E}$. *f* (*snd x*) = *f* (*snd y*) ⟶ *snd x* = *snd y* ⟹
(*aa*, *ba*) ∈ *Rep-kraus-family* $\mathfrak{E}$ ⟹ *f b* = *f ba* ⟹ (*aa*, *b*) ∈ *Rep-kraus-family* $\mathfrak{E}$ ⟹ *aa* ≠ 0 ⟹ *b* = *ba*›)
using *assms*
by (*auto intro*!: *simp*: *inj-on-def kf-domain.rep-eq EE-def*)
also have ‹... = $(\sum_\infty (E,x){\in}EE$. *sandwich-tc E* ϱ)›
by (*simp add*: *o-def case-prod-unfold*)
also have ‹... = $(\sum_\infty (E,x){\in}$*Rep-kraus-family* $\mathfrak{E}$. *sandwich-tc E* ϱ)›
apply (*rule infsum-cong-neutral*)
by (*auto simp*: *EE-def*)
also have ‹... = $\mathfrak{E}$ $*_{kr}$ ϱ›
by (*metis* (*no-types*, *lifting*) *infsum-cong kf-apply.rep-eq prod.case-eq-if*)
finally show ‹*kf-map-inj f* $\mathfrak{E}$ $*_{kr}$ ϱ = $\mathfrak{E}$ $*_{kr}$ ϱ›
by −
qed

lemma *kf-map-inj-eq-kf-map*:
assumes ‹*inj-on f* (*kf-domain* $\mathfrak{E}$)›
shows ‹*kf-map-inj f* $\mathfrak{E}$ ≡$_{kr}$ *kf-map f* $\mathfrak{E}$›
proof (*rule kf-eqI*)
fix *x* ϱ
define $\mathfrak{E}fx$ **where** ‹$\mathfrak{E}fx$ = *kf-filter* (λ*z*. *f z* = *x*) $\mathfrak{E}$›
from *assms* **have** *inj-*$\mathfrak{E}fx$: ‹*inj-on f* (*kf-domain* $\mathfrak{E}fx$)›
by (*auto simp add*: *inj-on-def kf-domain.rep-eq* $\mathfrak{E}fx$*-def kf-filter.rep-eq*)
have ‹*kf-map-inj f* $\mathfrak{E}$ $*_{kr}$ @{*x*} ϱ = *kf-filter* (λ*z*. *z*=*x*) (*kf-map-inj f* $\mathfrak{E}$) $*_{kr}$ ϱ›
by (*simp add*: *kf-apply-on-def*)
also have ‹... = *kf-map-inj f* $\mathfrak{E}fx$ $*_{kr}$ ϱ›
apply (*rule arg-cong*[**where** *f*=‹λ*t*. *kf-apply t* ϱ›])
unfolding $\mathfrak{E}fx$*-def*
apply (*transfer*′ *fixing*: *f*)
by *force*
also have ‹... = $\mathfrak{E}fx$ $*_{kr}$ ϱ›
using *inj-*$\mathfrak{E}fx$ **by** (*rule kf-apply-map-inj*)
also have ‹... = *kf-map f* (*kf-filter* (λ*z*. *f z* = *x*) $\mathfrak{E}$) $*_{kr}$ ϱ›
by (*simp add*: $\mathfrak{E}fx$*-def*)
also have ‹... = *kf-apply* (*kf-filter* (λ*xa*. *xa* = *x*) (*kf-map f* $\mathfrak{E}$)) ϱ›
apply (*subst kf-filter-map*)
by *simp*
also have ‹... = *kf-map f* $\mathfrak{E}$ $*_{kr}$ @{*x*} ϱ›
by (*simp add*: *kf-apply-on-def*)
finally show ‹*kf-map-inj f* $\mathfrak{E}$ $*_{kr}$ @{*x*} ϱ = *kf-map f* $\mathfrak{E}$ $*_{kr}$ @{*x*} ϱ›
by −

qed

lemma *kf-map-inj-id*[*simp*]: ‹*kf-map-inj id* $\mathfrak{E}$ = $\mathfrak{E}$›
apply *transfer'* **by** *simp*

lemma *kf-map-id*: ‹*kf-map id* $\mathfrak{E}$ $\equiv_{kr}$ $\mathfrak{E}$›
by (*metis inj-on-id kf-eq-sym kf-map-inj-eq-kf-map kf-map-inj-id*)

lemma *kf-map-inj-bound*[*simp*]:
fixes $\mathfrak{E}$ :: ‹('a::*chilbert-space*,'b::*chilbert-space*,'x) *kraus-family*›
assumes ‹*inj-on f* (*kf-domain* $\mathfrak{E}$)›
shows ‹*kf-bound* (*kf-map-inj f* $\mathfrak{E}$) = *kf-bound* $\mathfrak{E}$›
by (*metis assms kf-eq-imp-eq-weak kf-map-inj-eq-kf-map kf-bound-cong kf-map-bound*)

lemma *kf-map-inj-norm*[*simp*]:
fixes $\mathfrak{E}$ :: ‹('a::*chilbert-space*,'b::*chilbert-space*,'x) *kraus-family*›
assumes ‹*inj-on f* (*kf-domain* $\mathfrak{E}$)›
shows ‹*kf-norm* (*kf-map-inj f* $\mathfrak{E}$) = *kf-norm* $\mathfrak{E}$›
using *assms kf-eq-imp-eq-weak kf-map-inj-eq-kf-map kf-norm-cong* **by** *fastforce*

lemma *kf-map-cong-weak*:
assumes ‹$\mathfrak{E}$ $=_{kr}$ $\mathfrak{F}$›
shows ‹*kf-map f* $\mathfrak{E}$ $=_{kr}$ *kf-map g* $\mathfrak{F}$›
by (*metis assms kf-eq-weak-def kf-apply-map*)

lemma *kf-flatten-cong-weak*:
assumes ‹$\mathfrak{E}$ $=_{kr}$ $\mathfrak{F}$›
shows ‹*kf-flatten* $\mathfrak{E}$ $=_{kr}$ *kf-flatten* $\mathfrak{F}$›
using *assms* **by** (*rule kf-map-cong-weak*)

lemma *kf-flatten-cong*:
assumes ‹$\mathfrak{E}$ $=_{kr}$ $\mathfrak{F}$›
shows ‹*kf-flatten* $\mathfrak{E}$ $\equiv_{kr}$ *kf-flatten* $\mathfrak{F}$›
by (*simp add*: *assms kf-eq-weak-imp-eq-CARD-1 kf-flatten-cong-weak*)

lemma *kf-map-twice*:
‹*kf-map f* (*kf-map g* $\mathfrak{E}$) $\equiv_{kr}$ *kf-map* (*f* ∘ *g*) $\mathfrak{E}$›
apply (*rule kf-eqI*)
by (*simp add*: *kf-filter-map kf-apply-on-def*)

lemma *kf-map-cong*:
assumes ‹$\bigwedge x.\ x \in$ *kf-domain* $\mathfrak{E}$ $\Longrightarrow$ *f x* = *g x*›
assumes ‹$\mathfrak{E}$ $\equiv_{kr}$ $\mathfrak{F}$›
shows ‹*kf-map f* $\mathfrak{E}$ $\equiv_{kr}$ *kf-map g* $\mathfrak{F}$›
proof −
have ‹*kf-filter* (λ*y*. *f y* = *x*) $\mathfrak{E}$ $=_{kr}$ *kf-filter* (λ*y*. *g y* = *x*) $\mathfrak{F}$› **for** *x*
apply (*rule kf-filter-cong-weak*)
using *assms* **by** *auto*
then have ‹$\mathfrak{E}$ $*_{kr}$ @(*f*−‘{*x*}) ϱ = $\mathfrak{F}$ $*_{kr}$ @(*g*−‘{*x*}) ϱ› **for** *x* ϱ

by (*auto intro*!: *kf-apply-eqI simp add*: *kf-apply-on-def*)
then show *?thesis*
apply (*rule-tac kf-eqI*)
by (*simp add*: *kf-apply-on-def kf-filter-map*)
qed

lemma *kf-map-inj-cong-eq*:
assumes ‹$\bigwedge x.\ x \in$ *kf-domain* $\mathfrak{E} \Longrightarrow f\ x = g\ x$›
assumes ‹$\mathfrak{E} = \mathfrak{F}$›
shows ‹*kf-map-inj f* $\mathfrak{E}$ = *kf-map-inj g* $\mathfrak{F}$›
using *assms*
apply *transfer'*
by *force*

lemma *kf-domain-map*[*simp*]:
‹*kf-domain* (*kf-map f* $\mathfrak{E}$) = *f* ‘ *kf-domain* $\mathfrak{E}$›
proof (*rule Set.set-eqI*, *rule iffI*)
fix *x* **assume** ‹$x \in$ *kf-domain* (*kf-map f* $\mathfrak{E}$)›
then obtain *a* **where** ‹$(norm\ a)^2$ = *kf-element-weight* (*kf-filter* ($\lambda xa.\ f\ xa = x$) $\mathfrak{E}$) *a*› **and** ‹*a* ≠ *0*›
by (*auto intro*!: *simp*: *kf-domain.rep-eq kf-map.rep-eq*)
then have ‹*kf-element-weight* (*kf-filter* ($\lambda xa.\ f\ xa = x$) $\mathfrak{E}$) *a* ≠ *0*›
by *force*
then have ‹($\sum_\infty$(*E*, -)∈*kf-similar-elements* (*kf-filter* ($\lambda xa.\ f\ xa = x$) $\mathfrak{E}$) *a*. $(norm\ E)^2$) ≠ *0*›
by (*simp add*: *kf-element-weight-def*)
from *this*[*unfolded not-def*, *rule-format*, *OF infsum-0*]
obtain *E x'* **where** *rel-ops*: ‹(*E*,*x'*) ∈ *kf-similar-elements* (*kf-filter* ($\lambda xa.\ f\ xa = x$) $\mathfrak{E}$) *a*›
and ‹$(norm\ E)^2$ ≠ *0*›
by *fast*
then have ‹*E* ≠ *0*›
by *force*
with *rel-ops* **obtain** *E'* **where** ‹*E'* ≠ *0*› **and** ‹(*E'*,*x'*) ∈ *Rep-kraus-family* (*kf-filter* ($\lambda xa.\ f\ xa = x$) $\mathfrak{E}$)›
apply *atomize-elim*
by (*auto simp*: *kf-similar-elements-def*)
then have ‹(*E'*,*x'*) ∈ *Rep-kraus-family* $\mathfrak{E}$› **and** ‹*f x'* = *x*›
by (*auto simp*: *kf-filter.rep-eq*)
with ‹*E'* ≠ *0*› **have** ‹*x'* ∈ *kf-domain* $\mathfrak{E}$›
by (*force simp*: *kf-domain.rep-eq*)
with ‹*f x'* = *x*›
show ‹*x* ∈ *f* ‘ *kf-domain* $\mathfrak{E}$›
by *fast*
next
fix *x* **assume** ‹*x* ∈ *f* ‘ *kf-domain* $\mathfrak{E}$›
then obtain *y* **where** ‹*x* = *f y*› **and** ‹*y* ∈ *kf-domain* $\mathfrak{E}$›

```
    by blast
  then obtain E where ‹E ≠ 0› and ‹(E,y) ∈ Rep-kraus-family 𝔈›
    using Rep-kraus-family by (force simp: kf-domain.rep-eq kraus-family-def)
  then have Ey: ‹(E,y) ∈ Rep-kraus-family (kf-filter (λz. f z=x) 𝔈)›
    by (simp add: kf-filter.rep-eq ‹x = f y›)
  then have ‹kf-bound (kf-filter (λz. f z=x) 𝔈) ≠ 0›
  proof −
    define B where ‹B = kf-bound (kf-filter (λz. f z=x) 𝔈)›
    have ‹has-sum-in cweak-operator-topology (λ(E, x). E* o_CL E) {(E,y)} (E* o_CL E)›
      apply (subst asm-rl[of ‹E* o_CL E = (∑(E,x)∈{(E,y)}. E* o_CL E)›], simp)
      apply (rule has-sum-in-finite)
      by auto
   moreover have ‹has-sum-in cweak-operator-topology (λ(E, x). E* o_CL E) (Rep-kraus-family
(kf-filter (λz. f z = x) 𝔈)) B›
      using kf-bound-has-sum B-def by blast
    ultimately have ‹B ≥ E* o_CL E›
      apply (rule has-sum-mono-neutral-wot)
      using Ey positive-cblinfun-squareI by auto
    then show ‹B ≠ 0›
     by (meson ‹E ≠ 0› basic-trans-rules(24) op-square-nondegenerate positive-cblinfun-squareI)
  qed
  then have ‹kf-bound (kf-map f (kf-filter (λz. f z=x) 𝔈)) ≠ 0›
    by (simp add: kf-map-bound)
  then have ‹kf-bound (kf-filter (λz. z=x) (kf-map f 𝔈)) ≠ 0›
    by (simp add: kf-filter-map)
  from this[unfolded not-def kf-bound.rep-eq, rule-format, OF infsum-in-0]
  obtain E' x' where ‹(E',x') ∈ Rep-kraus-family (kf-filter (λz. z=x) (kf-map f 𝔈))›
    and ‹E' ≠ 0›
    by fastforce
  then have ‹(E',x') ∈ Rep-kraus-family (kf-map f 𝔈)› and ‹x' = x›
    by (auto simp: kf-filter.rep-eq)
  with ‹E' ≠ 0› show ‹x ∈ kf-domain (kf-map f 𝔈)›
    by (auto simp: kf-domain.rep-eq image-iff Bex-def)
qed


lemma kf-apply-on-map[simp]:
  ‹(kf-map f E) *_kr @X ϱ = E *_kr @(f −‘ X) ϱ›
  by (auto intro!: simp: kf-apply-on-def kf-filter-map)

lemma kf-apply-on-map-inj[simp]:
  assumes ‹inj-on f ((f −‘ X) ∩ kf-domain E)›
  shows  ‹kf-map-inj f E *_kr @X ϱ = E *_kr @(f −‘ X) ϱ›
proof −
  from assms
  have ‹inj-on f (Set.filter (λx. f x ∈ X) (kf-domain E))›
    by (simp add: Collect-conj-eq Int-ac(3) vimage-def)
  then show ?thesis
    by (auto intro!: simp: kf-apply-on-def kf-filter-map-inj)
```

qed

lemma *kf-map0*[*simp*]: ‹*kf-map f 0 = 0*›
apply *transfer'*
by *auto*

lemma *kf-map-inj-kr-eq-weak*:
assumes ‹*inj-on f* (*kf-domain* $\mathfrak{E}$)›
shows ‹*kf-map-inj f* $\mathfrak{E}$ =$_{kr}$ $\mathfrak{E}$›
by (*simp add*: *assms kf-eq-weakI*)

lemma *kf-map-inj-0*[*simp*]: ‹*kf-map-inj f 0 = 0*›
apply (*transfer' fixing*: *f*)
by *simp*

lemma *kf-domain-map-inj*[*simp*]: ‹*kf-domain* (*kf-map-inj f* $\mathfrak{E}$) = *f* ‘ *kf-domain* $\mathfrak{E}$›
apply *transfer'*
by *force*

lemma *kf-operators-kf-map*:
‹*kf-operators* (*kf-map f* $\mathfrak{E}$) ⊆ *span* (*kf-operators* $\mathfrak{E}$)›
proof (*rule subsetI*)
fix *E*
assume ‹*E* ∈ *kf-operators* (*kf-map f* $\mathfrak{E}$)›
then obtain *b* **where** ‹(*norm E*)^2 = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *b*) $\mathfrak{E}$) *E* ∧ *E* ≠ *0*›
by (*auto simp add*: *kf-operators.rep-eq kf-map.rep-eq*)
then have ‹*kf-element-weight* (*kf-filter* (λ*x*. *f x* = *b*) $\mathfrak{E}$) *E* ≠ *0*›
by *force*
then have ‹*E* ∈ *span* (*kf-operators* (*kf-filter* (λ*x*. *f x* = *b*) $\mathfrak{E}$))›
by (*rule kf-element-weight-kf-operators*)
then show ‹*E* ∈ *span* (*kf-operators* $\mathfrak{E}$)›
using *kf-operators-filter*[*of* ‹(λ*x*. *f x* = *b*)› $\mathfrak{E}$]
by (*meson basic-trans-rules*(*31*) *span-mono*)
qed

lemma *kf-operators-kf-map-inj*[*simp*]: ‹*kf-operators* (*kf-map-inj f* $\mathfrak{E}$) = *kf-operators* $\mathfrak{E}$›
apply *transfer'* **by** *force*

2.7 Addition

lift-definition *kf-plus* :: ‹('*a*::*chilbert-space*, '*b*::*chilbert-space*, '*x*) *kraus-family* ⇒ ('*a*,'*b*,'*y*) *kraus-family* ⇒ ('*a*,'*b*,'*x*+'*y*) *kraus-family*› **is**
‹λ$\mathfrak{E}$ $\mathfrak{F}$. (λ(*E*,*x*). (*E*, *Inl x*)) ‘ $\mathfrak{E}$ ∪ (λ(*F*,*y*). (*F*, *Inr y*)) ‘ $\mathfrak{F}$›
proof (*rename-tac* $\mathfrak{E}$ $\mathfrak{F}$)
fix $\mathfrak{E}$ :: ‹('*a* ⇒$_{CL}$ '*b* × '*x*) *set*› **and** $\mathfrak{F}$ :: ‹('*a* ⇒$_{CL}$ '*b* × '*y*) *set*›
assume ‹$\mathfrak{E}$ ∈ *Collect kraus-family*› **and** ‹$\mathfrak{F}$ ∈ *Collect kraus-family*›
then have ‹*kraus-family* $\mathfrak{E}$› **and** ‹*kraus-family* $\mathfrak{F}$›
by *auto*

then have ‹*kraus-family* ((λ(*E*, *x*). (*E*, *Inl x*)) ‘ 𝔈 ∪ (λ(*F*, *y*). (*F*, *Inr y*)) ‘ 𝔉)›
by (*force intro*!: *summable-on-Un-disjoint*
summable-on-reindex[*THEN iffD2*] *inj-onI*
simp: *kraus-family-iff-summable′ o-def case-prod-unfold conj-commute*)
then show ‹(λ(*E*, *x*). (*E*, *Inl x*)) ‘ 𝔈 ∪ (λ(*F*, *y*). (*F*, *Inr y*)) ‘ 𝔉 ∈ *Collect kraus-family*›
by *simp*
qed

instantiation *kraus-family* :: (*chilbert-space*, *chilbert-space*, *type*) *plus* **begin**
definition *plus-kraus-family* **where** ‹𝔈 + 𝔉 = *kf-map* (λ*xy*. *case xy of Inl x* ⇒ *x* | *Inr y* ⇒ *y*) (*kf-plus* 𝔈 𝔉)›
instance..
end

lemma *kf-plus-apply*:
fixes 𝔈 :: ‹(′*a*::*chilbert-space*, ′*b*::*chilbert-space*, ′*x*) *kraus-family*›
and 𝔉 :: ‹(′*a*, ′*b*, ′*y*) *kraus-family*›
shows ‹*kf-apply* (*kf-plus* 𝔈 𝔉) *ϱ* = *kf-apply* 𝔈 *ϱ* + *kf-apply* 𝔉 *ϱ*›
proof −
have ‹*kf-apply* (*kf-plus* 𝔈 𝔉) *ϱ* =
(∑∞*EF*∈(λ(*E*,*x*). (*E*, *Inl x*)) ‘ *Rep-kraus-family* 𝔈 ∪ (λ(*F*,*y*). (*F*, *Inr y*)) ‘ *Rep-kraus-family* 𝔉. *sandwich-tc* (*fst EF*) *ϱ*)›
by (*simp add*: *kf-plus.rep-eq kf-apply-def case-prod-unfold*)
also have ‹. . . = (∑∞*EF*∈(λ(*E*,*x*). (*E*, *Inl x* :: ′*x*+′*y*)) ‘ *Rep-kraus-family* 𝔈. *sandwich-tc* (*fst EF*) *ϱ*)
+ (∑∞*EF*∈(λ(*F*,*y*). (*F*, *Inr y* :: ′*x*+′*y*)) ‘ *Rep-kraus-family* 𝔉. *sandwich-tc* (*fst EF*) *ϱ*)›
apply (*subst infsum-Un-disjoint*)
using *kf-apply-summable*
by (*auto intro*!: *summable-on-reindex*[*THEN iffD2*] *inj-onI*
simp: *o-def case-prod-unfold kf-apply-summable*)
also have ‹. . . = (∑∞*E*∈*Rep-kraus-family* 𝔈. *sandwich-tc* (*fst E*) *ϱ*) + (∑∞*F*∈*Rep-kraus-family* 𝔉. *sandwich-tc* (*fst F*) *ϱ*)›
apply (*subst infsum-reindex*)
apply (*auto intro*!: *inj-onI*)[*1*]
apply (*subst infsum-reindex*)
apply (*auto intro*!: *inj-onI*)[*1*]
by (*simp add*: *o-def case-prod-unfold*)
also have ‹. . . = *kf-apply* 𝔈 *ϱ* + *kf-apply* 𝔉 *ϱ*›
by (*simp add*: *kf-apply-def*)
finally show *?thesis*
by −
qed

lemma *kf-plus-apply′*: ‹(𝔈 + 𝔉) ∗*kr* *ϱ* = 𝔈 ∗*kr* *ϱ* + 𝔉 ∗*kr* *ϱ*›
by (*simp add*: *kf-plus-apply plus-kraus-family-def*)

lemma *kf-plus-0-left*[*simp*]: ‹*kf-plus 0* 𝔈 = *kf-map-inj Inr* 𝔈›
apply *transfer′* **by** *auto*

lemma *kf-plus-0-right*[*simp*]: ‹*kf-plus* 𝔈 *0* = *kf-map-inj Inl* 𝔈›
apply *transfer'* **by** *auto*

lemma *kf-plus-0-left'*[*simp*]: ‹*0* + 𝔈 ≡$_{kr}$ 𝔈›
proof −
define *merge* **where** ‹*merge xy* = (*case xy of Inl x* ⇒ *x* | *Inr y* ⇒ *y*)› **for** *xy* :: ‹'*c* + '*c*›
have ‹*0* + 𝔈 = *kf-map merge* (*kf-map-inj Inr* 𝔈)›
by (*simp add*: *plus-kraus-family-def merge-def*[*abs-def*])
also have ‹... ≡$_{kr}$ *kf-map merge* (*kf-map Inr* 𝔈)›
by (*auto intro*!: *kf-map-cong kf-map-inj-eq-kf-map*)
also have ‹... ≡$_{kr}$ *kf-map* (*merge o Inr*) 𝔈›
by (*simp add*: *kf-map-twice*)
also have ‹... = *kf-map id* 𝔈›
apply (*rule arg-cong2*[**where** *f*=*kf-map*])
by (*auto simp*: *merge-def*)
also have ‹... ≡$_{kr}$ 𝔈›
by (*simp add*: *kf-map-id*)
finally show *?thesis*
by −
qed

lemma *kf-plus-0-right'*: ‹𝔈 + *0* ≡$_{kr}$ 𝔈›
proof −
define *merge* **where** ‹*merge xy* = (*case xy of Inl x* ⇒ *x* | *Inr y* ⇒ *y*)› **for** *xy* :: ‹'*c* + '*c*›
have ‹𝔈 + *0* = *kf-map merge* (*kf-map-inj Inl* 𝔈)›
by (*simp add*: *plus-kraus-family-def merge-def*[*abs-def*])
also have ‹... ≡$_{kr}$ *kf-map merge* (*kf-map Inl* 𝔈)›
by (*auto intro*!: *kf-map-cong kf-map-inj-eq-kf-map*)
also have ‹... ≡$_{kr}$ *kf-map* (*merge o Inl*) 𝔈›
by (*simp add*: *kf-map-twice*)
also have ‹... = *kf-map id* 𝔈›
apply (*rule arg-cong2*[**where** *f*=*kf-map*])
by (*auto simp*: *merge-def*)
also have ‹... ≡$_{kr}$ 𝔈›
by (*simp add*: *kf-map-id*)
finally show *?thesis*
by −
qed

lemma *kf-plus-bound*: ‹*kf-bound* (*kf-plus* 𝔈 𝔉) = *kf-bound* 𝔈 + *kf-bound* 𝔉›
proof −
define *l r* **where** ‹*l* = (λ(*E*::'*a*⇒$_{CL}$'*b*, *x*) ⇒ (*E*, *Inl x* :: '*c*+'*d*))›
and ‹*r* = (λ(*F*::'*a*⇒$_{CL}$'*b*, *y*) ⇒ (*F*, *Inr y* :: '*c*+'*d*))›
have ‹*Abs-cblinfun-wot* (*kf-bound* (*kf-plus* 𝔈 𝔉))
= (∑$_\infty$(*E*, *x*)∈*l* ‘ *Rep-kraus-family* 𝔈 ∪ *r* ‘ *Rep-kraus-family* 𝔉. *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*))›
by (*simp add*: *kf-bound-def' kf-plus.rep-eq Rep-cblinfun-wot-inverse flip*: *l-def r-def*)
also have ‹... = (∑$_\infty$(*E*, *x*)∈*l* ‘ *Rep-kraus-family* 𝔈. *compose-wot* (*adj-wot* (*Abs-cblinfun-wot*

E)) (*Abs-cblinfun-wot E*))
+ (∑∞(*E*, *x*)∈*r* ‘ *Rep-kraus-family* 𝔉. *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*))
(*Abs-cblinfun-wot E*))›
apply (*rule infsum-Un-disjoint*)
apply (*metis* (*no-types*, *lifting*) *ext Un-empty-right l-def image-empty kf-bound-summable′ kf-plus.rep-eq*
zero-kraus-family.rep-eq)
apply (*metis* (*no-types*, *lifting*) *ext r-def empty-subsetI image-empty kf-bound-summable′ kf-plus.rep-eq*
sup.absorb-iff2 zero-kraus-family.rep-eq)
by (*auto intro*!: *simp*: *l-def r-def*)
also have ‹... = *Abs-cblinfun-wot* (*kf-bound* (*kf-map-inj Inl* 𝔈 :: (-,-,′*c*+′*d*) *kraus-family*)) +
Abs-cblinfun-wot (*kf-bound* (*kf-map-inj Inr* 𝔉 :: (-,-,′*c*+′*d*) *kraus-family*))›
by (*simp add*: *kf-bound-def′ Rep-cblinfun-wot-inverse l-def r-def kf-map-inj.rep-eq case-prod-unfold*)
also have ‹... = *Abs-cblinfun-wot* (*kf-bound* 𝔈 + *kf-bound* 𝔉)›
by (*simp add*: *kf-map-inj-bound plus-cblinfun-wot.abs-eq*)
finally show *?thesis*
by (*metis* (*no-types*, *lifting*) *Rep-cblinfun-wot-inverse kf-bound-def′ plus-cblinfun-wot.rep-eq*)
qed

lemma *kf-plus-bound′*: ‹*kf-bound* (𝔈 + 𝔉) = *kf-bound* 𝔈 + *kf-bound* 𝔉›
by (*simp add*: *kf-map-bound kf-plus-bound plus-kraus-family-def*)

lemma *kf-norm-triangle*: ‹*kf-norm* (*kf-plus* 𝔈 𝔉) ≤ *kf-norm* 𝔈 + *kf-norm* 𝔉›
by (*simp add*: *kf-norm-def kf-plus-bound norm-triangle-ineq*)

lemma *kf-norm-triangle′*: ‹*kf-norm* (𝔈 + 𝔉) ≤ *kf-norm* 𝔈 + *kf-norm* 𝔉›
by (*simp add*: *kf-norm-def kf-plus-bound′ norm-triangle-ineq*)

lemma *kf-plus-map-both*:
‹*kf-plus* (*kf-map f* 𝔈) (*kf-map g* 𝔉) = *kf-map* (*map-sum f g*) (*kf-plus* 𝔈 𝔉)›
proof −
have *1*: ‹*kf-filter* (λ*x*. *map-sum f g x* = *Inl y*) (*kf-plus* 𝔈 𝔉) =
kf-map-inj Inl (*kf-filter* (λ*x*. *f x* = *y*) 𝔈)› **for** *y*
apply (*transfer′ fixing*: *f g y*)
by *force*
have *2*: ‹*kf-filter* (λ*x*. *map-sum f g x* = *Inr y*) (*kf-plus* 𝔈 𝔉) =
kf-map-inj Inr (*kf-filter* (λ*x*. *g x* = *y*) 𝔉)› **for** *y*
apply (*transfer′ fixing*: *f g y*)
by *force*
show *?thesis*
apply (*transfer′ fixing*: *f g* 𝔈 𝔉)
apply (*rule Set.set-eqI*)
subgoal for *x*
apply (*cases* ‹*snd x*›)
by (*auto intro*!: *simp*: *1 2 kf-element-weight-map-inj split*!: *sum.split*)
by −
qed

2.8 Composition

lemma *kf-comp-dependent-raw-norm-aux*:
fixes $\mathfrak{E}$ :: ‹'a ⇒ ('e::chilbert-space, 'f::chilbert-space, 'g) kraus-family›
and $\mathfrak{F}$ :: ‹('b::chilbert-space, 'e, 'a) kraus-family›
assumes *B*: ‹$\bigwedge x.\ x \in$ kf-domain $\mathfrak{F} \Longrightarrow$ kf-norm ($\mathfrak{E}$ x) ≤ B›
assumes [*simp*]: ‹B ≥ 0›
assumes ‹finite C›
assumes *C-subset*: ‹C ⊆ (λ((F,y), (E,x)). (E o_{CL} F, (F,E,y,x))) ‘ (SIGMA (F,y):Rep-kraus-family $\mathfrak{F}$. Rep-kraus-family ($\mathfrak{E}$ y))›
shows ‹($\sum$ (E,x)∈C. E* o_{CL} E) ≤ (B * kf-norm $\mathfrak{F}$) $*_R$ id-cblinfun›
proof −
define *BF* :: ‹'b ⇒$_{CL}$ 'b› **where** ‹BF = kf-norm $\mathfrak{F}$ $*_R$ id-cblinfun›
then have ‹kf-bound $\mathfrak{F}$ ≤ BF›
by (*simp add*: *kf-bound-leq-kf-norm-id*)
then have *BF*: ‹($\sum$ (F, y)∈M. (F* o_{CL} F)) ≤ BF› **if** ‹M ⊆ Rep-kraus-family $\mathfrak{F}$› **and** ‹finite M› **for** *M*
using *dual-order.trans kf-bound-geq-sum that*(1) **by** *blast*
define *BE* :: ‹'e ⇒$_{CL}$ 'e› **where** ‹BE = B $*_R$ id-cblinfun›
define $\mathfrak{F}x\mathfrak{E}$ **where** ‹$\mathfrak{F}x\mathfrak{E}$ = (SIGMA (F,y):Rep-kraus-family $\mathfrak{F}$. Rep-kraus-family ($\mathfrak{E}$ y))›
have *BE*: ‹($\sum$ (E, x)∈M. (E* o_{CL} E)) ≤ BE› **if** ‹y ∈ kf-domain $\mathfrak{F}$› **and** ‹M ⊆ Rep-kraus-family ($\mathfrak{E}$ y)› **and** ‹finite M› **for** *M y*
proof −
from *B that*(1,2)
have ‹norm ($\sum$ (E, x)∈M. E* o_{CL} E) ≤ B›
by (*smt* (*verit*) *kf-norm-geq-norm-sum that*)
then show *?thesis*
by (*auto intro*!: *less-eq-scaled-id-norm pos-selfadjoint sum-nonneg intro*: *positive-cblinfun-squareI simp*: *BE-def*)
qed

define *A* **where** ‹A = ($\sum$ (E,x)∈C. E* o_{CL} E)›
define *CE CF* **where** ‹CE y = (λ(-,(F,E,y,x)). (E,x)) ‘ Set.filter (λ(-,(F,E,y',x)). y'=y) C›
and ‹CF = (λ(-,(F,E,y,x)). (F,y)) ‘ C› **for** *y*
with ‹finite C› **have** [*simp*]: ‹finite (CE y)› ‹finite CF› **for** *y*
by *auto*
have *C-C1C2*: ‹C ⊆ (λ((F,y), (E,x)). (E o_{CL} F, (F,E,y,x))) ‘ (SIGMA (F,y):CF. CE y)›
proof (*rule subsetI*)
fix *c* **assume** ‹c ∈ C›
then obtain *EF E F x y* **where** *c-def*: ‹c = (EF,(F,E,y,x))›
by (*metis surj-pair*)
from ‹c ∈ C› **have** *EF-def*: ‹EF = E o_{CL} F›
using *C-subset* **by** (*auto intro*!: *simp*: *c-def*)
from ‹c ∈ C› **have** *1*: ‹(F,y) ∈ CF›
apply (*simp add*: *CF-def c-def*)
by *force*
from ‹c ∈ C› **have** *2*: ‹(E,x) ∈ CE y›
apply (*simp add*: *CE-def c-def*)
by *force*

from *1 2* **show** ‹$c \in (\lambda((F,\ y),\ E,\ x).\ (E\ o_{CL}\ F,\ (F,E,y,x)))$ ‘ $(SIGMA\ (F,\ y){:}CF.\ CE\ y)$›
apply (*simp add*: *c-def EF-def*)
by *force*
qed

have *CE-sub-*$\mathfrak{E}$: ‹$CE\ y \subseteq$ *Rep-kraus-family* $(\mathfrak{E}\ y)$› **and** ‹$CF \subseteq$ *Rep-kraus-family* $\mathfrak{F}$› **for** y
using *C-subset* **by** (*auto simp add*: *CE-def CF-def* $\mathfrak{F}x\mathfrak{E}$*-def case-prod-unfold*)
have *CE-BE*: ‹$(\sum (E,\ x){\in}CE\ y.\ (E{*}\ o_{CL}\ E)) \leq BE$› **if** ‹$y \in$ *kf-domain* $\mathfrak{F}$› **for** y
using *BE*[**where** $y{=}y$] *CE-sub-*$\mathfrak{E}$[*of* y] *that*
by *auto*

have ‹$A \leq (\sum (E,x) \in (\lambda((F,y),\ (E,x)).\ (E\ o_{CL}\ F,\ (F,E,y,x)))$ ‘ $(SIGMA\ (F,y){:}CF.\ CE\ y).\ E{*}\ o_{CL}\ E)$›
using *C-C1C2* **by** (*auto intro*!: *finite-imageI sum-mono2 positive-cblinfun-squareI simp*: *A-def simp flip*: *adj-cblinfun-compose*)[*1*]
also have ‹$\ldots = (\sum ((F,y),\ (E,x)){\in}(SIGMA\ (F,y){:}CF.\ CE\ y).\ (F{*}\ o_{CL}\ (E{*}\ o_{CL}\ E)\ o_{CL}\ F))$›
apply (*subst sum.reindex*)
by (*auto intro*!: *inj-onI simp*: *case-prod-unfold cblinfun-compose-assoc*)
also have ‹$\ldots = (\sum (F,\ y){\in}CF.\ sandwich\ (F{*})\ (\sum (E,\ x){\in}CE\ y.\ (E{*}\ o_{CL}\ E)))$›
apply (*subst sum.Sigma*[*symmetric*])
by (*auto intro*!: *simp*: *case-prod-unfold sandwich-apply cblinfun-compose-sum-right cblinfun-compose-sum-left simp flip*:)
also have ‹$\ldots \leq (\sum (F,\ y){\in}CF.\ sandwich\ (F{*})\ BE)$›
proof (*rule sum-mono*)
fix $i :: {'b \Rightarrow_{CL} 'e \times 'a}$ **assume** ‹$i \in CF$›
obtain $F\ y$ **where** i: ‹$i = (F,y)$›
by *force*
have *1*: ‹$sandwich\ (F{*})\ {*_V}\ (\sum (E,x){\in}CE\ y.\ E{*}\ o_{CL}\ E) \leq sandwich\ (F{*})\ {*_V}\ BE$› **if** ‹$y \in$ *kf-domain* $\mathfrak{F}$›
apply (*rule sandwich-mono*)
using *that CE-BE* **by** *simp*
have ‹$F = 0$› **if** ‹$y \notin$ *kf-domain* $\mathfrak{F}$›
using *C-subset CF-def* ‹$CF \subseteq$ *Rep-kraus-family* $\mathfrak{F}$› ‹$i \in CF$› *that i*
by (*smt* (*verit*, *ccfv-SIG*) *Set.basic-monos*(*7*) *Set.filter-eq case-prodI image-iff kf-domain.rep-eq prod.sel*(*2*))
then have *2*: ‹$sandwich\ (F{*})\ {*_V}\ (\sum (E,x){\in}CE\ y.\ E{*}\ o_{CL}\ E) \leq sandwich\ (F{*})\ {*_V}\ BE$› **if** ‹$y \notin$ *kf-domain* $\mathfrak{F}$›
using *that* **by** *simp*
from *1 2* **show** ‹$(case\ i\ of\ (F,\ y) \Rightarrow sandwich\ (F{*})\ {*_V}\ (\sum (E,\ x){\in}CE\ y.\ E{*}\ o_{CL}\ E))$
$\leq (case\ i\ of\ (F,\ y) \Rightarrow sandwich\ (F{*})\ {*_V}\ BE)$›
by (*auto simp*: *case-prod-unfold i*)
qed
also have ‹$\ldots = B\ {*_R}\ (\sum (F,\ y){\in}CF.\ F{*}\ o_{CL}\ F)$›
by (*simp add*: *scaleR-sum-right case-prod-unfold sandwich-apply BE-def*)
also have ‹$\ldots \leq B\ {*_R}\ BF$›
using *BF* **by** (*simp add*: ‹$CF \subseteq$ *Rep-kraus-family* $\mathfrak{F}$› *scaleR-left-mono case-prod-unfold*)
also have ‹$B\ {*_R}\ BF = (B * kf\text{-}norm\ \mathfrak{F})\ {*_R}\ id\text{-}cblinfun$›
by (*simp add*: *BF-def*)

finally show ‹$A \leq (B * \mathit{kf\text{-}norm}\ \mathfrak{F}) *_R \mathit{id\text{-}cblinfun}$›
by −
qed

lift-definition *kf-comp-dependent-raw* :: ‹('x ⇒ ('b::*chilbert-space*,'c::*chilbert-space*,'y) *kraus-family*) ⇒ ('a::*chilbert-space*,'b,'x) *kraus-family* ⇒ ('a, 'c, ('a ⇒$_{CL}$ 'b) × ('b ⇒$_{CL}$ 'c) × 'x × 'y) *kraus-family*› **is**
‹$\lambda \mathfrak{E}\ \mathfrak{F}$. *if bdd-above* ((*kf-norm o* $\mathfrak{E}$) ‘ *kf-domain* $\mathfrak{F}$) *then Set.filter* (λ(*EF*,-). *EF*≠*0*) ((λ((*F*,*y*), (*E*::'b⇒$_{CL}$'c,*x*::'y)). (*E* o_{CL} *F*, (*F*,*E*,*y*,*x*))) ‘ (*SIGMA* (*F*::'a⇒$_{CL}$'b,*y*::'x):*Rep-kraus-family* $\mathfrak{F}$. (*Rep-kraus-family* ($\mathfrak{E}$ *y*)))) *else* {}›
proof (*rename-tac* $\mathfrak{E}$ $\mathfrak{F}$)
fix $\mathfrak{E}$ :: ‹'x ⇒ ('b, 'c, 'y) *kraus-family*› **and** $\mathfrak{F}$ :: ‹('a, 'b, 'x) *kraus-family*›
show ‹(*if bdd-above* ((*kf-norm o* $\mathfrak{E}$) ‘ *kf-domain* $\mathfrak{F}$) *then Set.filter* (λ(*EF*,-). *EF*≠*0*) ((λ((*F*, *y*), *E*, *x*). (*E* o_{CL} *F*, (*F*, *E*, *y*, *x*))) ‘ (*SIGMA* (*F*, *y*):*Rep-kraus-family* $\mathfrak{F}$. *Rep-kraus-family* ($\mathfrak{E}$ *y*))) *else* {}) ∈ *Collect kraus-family*›
proof (*cases* ‹*bdd-above* ((*kf-norm o* $\mathfrak{E}$) ‘ *kf-domain* $\mathfrak{F}$)›)
case *True*
obtain *B* **where** $\mathfrak{E}$-*uniform*: ‹*y* ∈ *kf-domain* $\mathfrak{F}$ ⟹ *kf-norm* ($\mathfrak{E}$ *y*) ≤ *B*› **and** ‹*B* ≥ *0*› **for** *y*
proof *atomize-elim*
from *True*
obtain *B0* **where** ‹*y* ∈ *kf-domain* $\mathfrak{F}$ ⟹ *kf-norm* ($\mathfrak{E}$ *y*) ≤ *B0*› **for** *y*
by (*auto simp*: *bdd-above-def*)
then show ‹∃ *B*. (∀ *y*. *y* ∈ *kf-domain* $\mathfrak{F}$ ⟶ *kf-norm* ($\mathfrak{E}$ *y*) ≤ *B*) ∧ *0* ≤ *B*›
apply (*rule-tac exI*[*of* - ‹*max 0 B0*›])
by *force*
qed
define $\mathfrak{F}x\mathfrak{E}$ **where** ‹$\mathfrak{F}x\mathfrak{E}$ = (*SIGMA* (*F*,*y*):*Rep-kraus-family* $\mathfrak{F}$. *Rep-kraus-family* ($\mathfrak{E}$ *y*))›
have ‹*bdd-above* ((λ*M*. $\sum$(*E*,*x*)∈*M*. *E*∗ o_{CL} *E*) ‘ {*M*. *M* ⊆ *Set.filter* (λ(*EF*, -). *EF* ≠ *0*) ((λ((*F*,*y*), (*E*,*x*)). (*E* o_{CL} *F*, (*F*,*E*,*y*,*x*))) ‘ $\mathfrak{F}x\mathfrak{E}$) ∧ *finite M*})›
proof (*rule bdd-aboveI*, *rename-tac A*)
fix *A* :: ‹'a ⇒$_{CL}$ 'a›
assume ‹*A* ∈ (λ*M*. $\sum$(*E*, *x*)∈*M*. *E*∗ o_{CL} *E*) ‘ {*M*. *M* ⊆ *Set.filter* (λ(*EF*, -). *EF* ≠ *0*) ((λ((*F*,*y*), (*E*,*x*)). (*E* o_{CL} *F*, (*F*,*E*,*y*,*x*))) ‘ $\mathfrak{F}x\mathfrak{E}$) ∧ *finite M*}›
then obtain *C* **where** *A-def*: ‹*A* = ($\sum$(*E*,*x*)∈*C*. *E*∗ o_{CL} *E*)›
and *C*$\mathfrak{F}\mathfrak{E}$: ‹*C* ⊆ *Set.filter* (λ(*EF*,-). *EF* ≠ *0*) ((λ((*F*,*y*), (*E*,*x*)). (*E* o_{CL} *F*, (*F*,*E*,*y*,*x*))) ‘ $\mathfrak{F}x\mathfrak{E}$)›
and [*simp*]: ‹*finite C*›
by *auto*
from *kf-comp-dependent-raw-norm-aux*[*OF* $\mathfrak{E}$-*uniform* ‹*B* ≥ *0*› ‹*finite C*›]
show ‹*A* ≤ (*B* ∗ *kf-norm* $\mathfrak{F}$) $*_R$ *id-cblinfun*›
using *C*$\mathfrak{F}\mathfrak{E}$
by (*force intro*!: *simp*: *A-def* $\mathfrak{F}x\mathfrak{E}$-*def*)
qed
then have ‹*kraus-family* (*Set.filter* (λ(*EF*,-). *EF*≠*0*) ((λ((*F*, *y*), *E*, *x*). (*E* o_{CL} *F*, (*F*,*E*,*y*,*x*))) ‘ (*SIGMA* (*F*, *y*):*Rep-kraus-family* $\mathfrak{F}$. *Rep-kraus-family* ($\mathfrak{E}$ *y*))))›

```
    by (auto intro!: kraus-familyI simp: conj-commute 𝔉x𝔈-def)
  then show ?thesis
    using True by simp
 next
  case False
  then show ?thesis
    by (auto simp: kraus-family-def)
 qed
qed

lemma kf-comp-dependent-raw-norm-leq:
 fixes 𝔈 :: ‹'a ⇒ ('b::chilbert-space, 'c::chilbert-space, 'd) kraus-family›
  and 𝔉 :: ‹('e::chilbert-space, 'b, 'a) kraus-family›
 assumes ‹⋀x. x ∈ kf-domain 𝔉 ⟹ kf-norm (𝔈 x) ≤ B›
 assumes ‹B ≥ 0›
 shows ‹kf-norm (kf-comp-dependent-raw 𝔈 𝔉) ≤ B * kf-norm 𝔉›
proof −
 wlog not-singleton: ‹class.not-singleton TYPE('e)›
  using not-not-singleton-kf-norm-0[OF negation, of 𝔉]
  using not-not-singleton-kf-norm-0[OF negation, of ‹kf-comp-dependent-raw 𝔈 𝔉›]
  by simp
 show ?thesis
 proof (rule kf-norm-sum-leqI)
  fix F assume ‹finite F› and F-subset: ‹F ⊆ Rep-kraus-family (kf-comp-dependent-raw 𝔈
𝔉)›
  have [simp]: ‹norm (id-cblinfun :: 'e ⇒CL 'e) = 1›
   apply (rule norm-cblinfun-id[internalize-sort' 'a])
   apply (rule complex-normed-vector-axioms)
   by (rule not-singleton)
  from assms have bdd: ‹bdd-above ((λx. kf-norm (𝔈 x)) ‘ kf-domain 𝔉)›
   by fast
  have ‹(∑(E, x)∈F. E* o_CL E) ≤ (B * kf-norm 𝔉) *R id-cblinfun›
   using assms ‹finite F› apply (rule kf-comp-dependent-raw-norm-aux)
   using F-subset by (auto simp: kf-comp-dependent-raw.rep-eq bdd)
  then have ‹norm (∑(E, x)∈F. E* o_CL E) ≤ norm ((B * kf-norm 𝔉) *R (id-cblinfun :: 'e
⇒CL 'e))›
   apply (rule norm-cblinfun-mono[rotated])
   using positive-cblinfun-squareI
   by (auto intro!: sum-nonneg)
  then show ‹norm (∑(E, x)∈F. E* o_CL E) ≤ B * kf-norm 𝔉›
   using ‹B ≥ 0› by auto
 qed
qed

hide-fact kf-comp-dependent-raw-norm-aux

definition ‹kf-comp-dependent 𝔈 𝔉 = kf-map (λ(F,E,y,x). (y,x)) (kf-comp-dependent-raw 𝔈 𝔉)›

definition ‹kf-comp 𝔈 𝔉 = kf-comp-dependent (λ-. 𝔈) 𝔉›
```

lemma *kf-comp-dependent-norm-leq*:
assumes ‹$\bigwedge x.\ x \in$ *kf-domain* $\mathfrak{F} \Longrightarrow$ *kf-norm* $(\mathfrak{E}\ x) \leq B$›
assumes ‹$B \geq 0$›
shows ‹*kf-norm* (*kf-comp-dependent* $\mathfrak{E}\ \mathfrak{F}$) $\leq B *$ *kf-norm* $\mathfrak{F}$›
using *assms* **by** (*auto intro*!: *kf-comp-dependent-raw-norm-leq simp*: *kf-comp-dependent-def*)

lemma *kf-comp-norm-leq*:
shows ‹*kf-norm* (*kf-comp* $\mathfrak{E}\ \mathfrak{F}$) $\leq$ *kf-norm* $\mathfrak{E} *$ *kf-norm* $\mathfrak{F}$›
by (*auto intro*!: *kf-comp-dependent-norm-leq simp*: *kf-comp-def*)

lemma *kf-comp-dependent-raw-apply*:
fixes $\mathfrak{E}$ :: ‹'*y* ⇒ ('*a*::*chilbert-space*, '*b*::*chilbert-space*, '*x*) *kraus-family*›
and $\mathfrak{F}$ :: ‹('*c*::*chilbert-space*, '*a*, '*y*) *kraus-family*›
assumes ‹*bdd-above* ((*kf-norm o* $\mathfrak{E}$) ‘ *kf-domain* $\mathfrak{F}$)›
shows ‹*kf-comp-dependent-raw* $\mathfrak{E}\ \mathfrak{F} *_{kr} \varrho$
$= (\sum_\infty (F,y)\in$*Rep-kraus-family* $\mathfrak{F}$. $\mathfrak{E}\ y *_{kr}$ *sandwich-tc* $F\ \varrho)$›
proof −
have *sum2*: ‹(λ(*F*, *x*). *sandwich-tc F* ϱ) *summable-on* (*Rep-kraus-family* $\mathfrak{F}$)›
using *kf-apply-summable*[*of* ϱ $\mathfrak{F}$] **by** (*simp add*: *case-prod-unfold*)
have ‹(λ*E*. *sandwich-tc* (*fst E*) ϱ) *summable-on*
(*Set.filter* (λ(*E*,*x*). *E* ≠ *0*) ((λ((*F*,*y*), (*E*,*x*)). (*E* o_{CL} *F*, (*F*,*E*,*y*,*x*))) ‘ (*SIGMA* (*F*,*y*):*Rep-kraus-family* $\mathfrak{F}$. *Rep-kraus-family* ($\mathfrak{E}$ *y*))))›
using *kf-apply-summable*[*of* - ‹*kf-comp-dependent-raw* $\mathfrak{E}\ \mathfrak{F}$›] *assms*
by (*simp add*: *kf-comp-dependent-raw.rep-eq case-prod-unfold*)
then have ‹(λ*E*. *sandwich-tc* (*fst E*) ϱ) *summable-on*
(λ((*F*,*y*), (*E*,*x*)). (*E* o_{CL} *F*, (*F*,*E*,*y*,*x*))) ‘ (*SIGMA* (*F*,*y*):*Rep-kraus-family* $\mathfrak{F}$. *Rep-kraus-family* ($\mathfrak{E}$ *y*))›
apply (*rule summable-on-cong-neutral*[*THEN iffD1*, *rotated* −*1*])
by *force*+
then have *sum1*: ‹(λ((*F*,*y*), (*E*,*x*)). *sandwich-tc* (*E* o_{CL} *F*) ϱ) *summable-on* (*SIGMA* (*F*,*y*):*Rep-kraus-family* $\mathfrak{F}$. *Rep-kraus-family* ($\mathfrak{E}$ *y*))›
apply (*subst* (*asm*) *summable-on-reindex*)
by (*auto intro*!: *inj-onI simp*: *o-def case-prod-unfold*)
have ‹*kf-comp-dependent-raw* $\mathfrak{E}\ \mathfrak{F} *_{kr} \varrho$
$= (\sum_\infty E\in$(*Set.filter* (λ(*E*,*x*). *E* ≠ *0*) ((λ((*F*,*y*), (*E*,*x*)). (*E* o_{CL} *F*, (*F*,*E*,*y*,*x*))) ‘ (*SIGMA* (*F*,*y*):*Rep-kraus-family* $\mathfrak{F}$. *Rep-kraus-family* ($\mathfrak{E}$ *y*)))). *sandwich-tc* (*fst E*) ϱ)›
using *assms* **by** (*simp add*: *kf-apply-def kf-comp-dependent-raw.rep-eq case-prod-unfold*)
also have ‹... = $(\sum_\infty E\in$(λ((*F*,*y*), (*E*,*x*)). (*E* o_{CL} *F*, (*F*,*E*,*y*,*x*))) ‘ (*SIGMA* (*F*,*y*):*Rep-kraus-family* $\mathfrak{F}$. *Rep-kraus-family* ($\mathfrak{E}$ *y*)). *sandwich-tc* (*fst E*) ϱ)›
apply (*rule infsum-cong-neutral*)
by *force*+
also have ‹... = $(\sum_\infty$((*F*,*y*), (*E*,*x*))∈(*SIGMA* (*F*,*y*):*Rep-kraus-family* $\mathfrak{F}$. *Rep-kraus-family* ($\mathfrak{E}$ *y*)). *sandwich-tc* (*E* o_{CL} *F*) ϱ)›
apply (*subst infsum-reindex*)
by (*auto intro*!: *inj-onI simp*: *o-def case-prod-unfold*)
also have ‹... = $(\sum_\infty (F,y)\in$*Rep-kraus-family* $\mathfrak{F}$. $\sum_\infty (E,x)\in$*Rep-kraus-family* ($\mathfrak{E}$ *y*). *sandwich-tc* (*E* o_{CL} *F*) ϱ)›
apply (*subst infsum-Sigma'-banach*[*symmetric*])

using *sum1* **by** (*auto simp*: *case-prod-unfold*)
also have ‹... = ($\sum_\infty$(F,y)∈*Rep-kraus-family* $\mathfrak{F}$. $\sum_\infty$(E,x)∈*Rep-kraus-family* ($\mathfrak{E}$ y). *sandwich-tc E* (*sandwich-tc F* ϱ))›
by (*simp add*: *sandwich-tc-compose*)
also have ‹... = ($\sum_\infty$(F,y)∈*Rep-kraus-family* $\mathfrak{F}$. *kf-apply* ($\mathfrak{E}$ y) (*sandwich-tc F* ϱ))›
by (*simp add*: *kf-apply-def case-prod-unfold*)
finally show *?thesis*
by −
qed

lemma *kf-comp-dependent-apply*:
fixes $\mathfrak{E}$ :: ‹'y ⇒ ('a::*chilbert-space*, 'b::*chilbert-space*, 'x) *kraus-family*›
and $\mathfrak{F}$ :: ‹('c::*chilbert-space*, 'a, 'y) *kraus-family*›
assumes ‹*bdd-above* ((*kf-norm o* $\mathfrak{E}$) ' *kf-domain* $\mathfrak{F}$)›
shows ‹*kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ ϱ
= ($\sum_\infty$(F,y)∈*Rep-kraus-family* $\mathfrak{F}$. $\mathfrak{E}$ y $*_{kr}$ *sandwich-tc F* ϱ)›
using *assms* **by** (*simp add*: *kf-comp-dependent-def kf-apply-map*
kf-comp-dependent-raw-apply)

lemma *kf-comp-apply*:
shows ‹*kf-apply* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$) = *kf-apply* $\mathfrak{E}$ ∘ *kf-apply* $\mathfrak{F}$›
proof (*rule ext*, *rename-tac* ϱ)
fix ϱ :: ‹('a, 'a) *trace-class*›

have *sumF*: ‹(λ(F, y). *sandwich-tc F* ϱ) *summable-on Rep-kraus-family* $\mathfrak{F}$›
by (*rule kf-apply-summable*)
have ‹*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ ϱ = ($\sum_\infty$(F,y)∈*Rep-kraus-family* $\mathfrak{F}$. $\mathfrak{E}$ $*_{kr}$ *sandwich-tc F* ϱ)›
by (*auto intro*!: *kf-comp-dependent-apply simp*: *kf-comp-def*)
also have ‹... = *kf-apply* $\mathfrak{E}$ ($\sum_\infty$(F,y)∈*Rep-kraus-family* $\mathfrak{F}$. *sandwich-tc F* ϱ)›
apply (*subst infsum-bounded-linear*[*symmetric*, **where** h=‹*kf-apply* $\mathfrak{E}$›])
using *sumF* **by** (*auto intro*!: *bounded-clinear.bounded-linear kf-apply-bounded-clinear*
simp: *o-def case-prod-unfold*)
also have ‹... = (*kf-apply* $\mathfrak{E}$ ∘ *kf-apply* $\mathfrak{F}$) ϱ›
by (*simp add*: *o-def kf-apply-def case-prod-unfold*)
finally show ‹*kf-apply* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$) ϱ = (*kf-apply* $\mathfrak{E}$ ∘ *kf-apply* $\mathfrak{F}$) ϱ›
by −
qed

lemma *kf-comp-cong-weak*: ‹*kf-comp F G* $=_{kr}$ *kf-comp F' G'*›
if ‹*F* $=_{kr}$ *F'*› **and** ‹*G* $=_{kr}$ *G'*›
by (*metis kf-eq-weak-def kf-comp-apply that*)

lemma *kf-comp-dependent-raw-assoc*:
fixes $\mathfrak{E}$ :: ‹'f ⇒ ('c::*chilbert-space*,'d::*chilbert-space*,'e) *kraus-family*›
and $\mathfrak{F}$ :: ‹'g ⇒ ('b::*chilbert-space*,'c::*chilbert-space*,'f) *kraus-family*›
and $\mathfrak{G}$ :: ‹('a::*chilbert-space*,'b::*chilbert-space*,'g) *kraus-family*›
defines ‹*reorder* :: 'a $\Rightarrow_{CL}$ 'c × 'c $\Rightarrow_{CL}$ 'd × ('a $\Rightarrow_{CL}$ 'b × 'b $\Rightarrow_{CL}$ 'c × 'g × 'f) × 'e
⇒ 'a $\Rightarrow_{CL}$ 'b × 'b $\Rightarrow_{CL}$ 'd × 'g × 'b $\Rightarrow_{CL}$ 'c × 'c $\Rightarrow_{CL}$ 'd × 'f × 'e ≡
λ(FG::'a $\Rightarrow_{CL}$ 'c, E::'c $\Rightarrow_{CL}$ 'd, (G::'a $\Rightarrow_{CL}$ 'b, F::'b $\Rightarrow_{CL}$ 'c, g::'g, f::'f), e::'e).

```
            (G, E o_CL F, g, F, E, f, e)›
  assumes ‹bdd-above (range (kf-norm o 𝔈))›
  assumes ‹bdd-above (range (kf-norm o 𝔉))›
  shows ‹kf-comp-dependent-raw (λg::'g. kf-comp-dependent-raw 𝔈 (𝔉 g)) 𝔊
        = kf-map-inj reorder (kf-comp-dependent-raw (λ(-,-,-,f). 𝔈 f) (kf-comp-dependent-raw 𝔉
𝔊))›
    (is ‹?lhs = ?rhs›)
proof (rule Rep-kraus-family-inject[THEN iffD1])
  from assms have bdd-E: ‹bdd-above ((kf-norm o 𝔈) ` X)› for X
    by (simp add: bdd-above-mono2)
  from assms have bdd-F: ‹bdd-above ((kf-norm o 𝔉) ` X)› for X
    by (simp add: bdd-above-mono2)
  have bdd1: ‹bdd-above ((λx. kf-norm (kf-comp-dependent-raw 𝔈 (𝔉 x))) ` X)› for X
  proof −
    from bdd-F[where X=UNIV] obtain BF where BF: ‹kf-norm (𝔉 x) ≤ BF› for x
      by (auto simp: bdd-above-def)
    moreover from bdd-E[where X=UNIV] obtain BE where BE: ‹kf-norm (𝔈 x) ≤ BE›
for x
      by (auto simp: bdd-above-def)
    ultimately have ‹kf-norm (kf-comp-dependent-raw (λx. 𝔈 x) (𝔉 x)) ≤ BE * BF› for x
    by (smt (verit, best) kf-comp-dependent-raw-norm-leq kf-norm-geq0 landau-omega.R-mult-left-mono)
    then show ?thesis
      by (auto intro!: bdd-aboveI)
  qed
  have bdd2: ‹bdd-above ((kf-norm ∘ (λ(-::'a ⇒_CL 'b, -::'b ⇒_CL 'c, -::'g, y::'f). 𝔈 y)) ` X)› for
X
    using assms(2) by (auto simp: bdd-above-def)
  define EE FF GG where ‹EE f = Rep-kraus-family (𝔈 f)› and ‹FF g = Rep-kraus-family
(𝔉 g)› and ‹GG = Rep-kraus-family 𝔊› for f g
  have ‹Rep-kraus-family ?lhs
        = (Set.filter (λ(E,x). E ≠ 0) ((λ((F,y), (E, x)). (E o_CL F, F, E, y, x)) `
          (SIGMA (F, y):GG. Rep-kraus-family (kf-comp-dependent-raw 𝔈 (𝔉 y)))))›
    apply (subst kf-comp-dependent-raw.rep-eq)
    using bdd1 by (simp add: GG-def)
  also have ‹... = (Set.filter (λ(E,x). E ≠ 0) ((λ((G, g), (EF, x)). (EF o_CL G, G, EF, g, x))
`
    (SIGMA (G, g):GG. Set.filter (λ(E,x). E ≠ 0) ((λ((F, f), (E, e)). (E o_CL F, F, E, f, e))
` (SIGMA (F, f):FF g. EE f)))))›
    unfolding EE-def FF-def
    apply (subst kf-comp-dependent-raw.rep-eq)
    using assms bdd-E by (simp add: case-prod-beta)
  also have ‹... = (Set.filter (λ(E,x). E ≠ 0) ((λ((G, g), (EF, x)). (EF o_CL G, G, EF, g, x))
`
    (SIGMA (G, g):GG. (λ((F, f), (E, e)). (E o_CL F, F, E, f, e)) ` (SIGMA (F, f):FF g. EE
f))))›
    by force
  also have ‹... = (Set.filter (λ(E,x). E ≠ 0) ((λ((F, y), (E, x)). (E o_CL F, reorder (F, E,
y, x))) `
        (SIGMA (FG, -, -, -, y):(λ((G, g), (F, f)). (F o_CL G, G, F, g, f)) ` (SIGMA (G, g):GG.
```

FF g). EE y)))›
by (*force simp*: *reorder-def image-iff case-prod-unfold cblinfun-compose-assoc*)
also have ‹... = (*Set.filter* (λ(*E*,*x*). *E* ≠ *0*) ((λ((*F*, *y*), (*E*, *x*)). (*E* o_{CL} *F*, *reorder* (*F*, *E*, *y*, *x*))) ‘
(*SIGMA* (*FG*, -, -, -, *y*):*Set.filter* (λ(*E*,*x*). *E* ≠ *0*) ((λ((*G*, *g*), (*F*, *f*)). (*F* o_{CL} *G*, *G*, *F*, *g*, *f*)) ‘ (*SIGMA* (*G*, *g*):*GG*. *FF g*)). *EE y*)))›
by *force*
also have ‹... = (*Set.filter* (λ(*E*,*x*). *E* ≠ *0*) ((λ((*F*, *y*), (*E*, *x*)). (*E* o_{CL} *F*, *reorder* (*F*, *E*, *y*, *x*))) ‘
(*SIGMA* (*FG*, (-, -, -, *f*)):*Rep-kraus-family* (*kf-comp-dependent-raw* 𝔉 𝔊). *EE f*)))›
apply (*rule arg-cong*[**where** *f*=‹*Set.filter* -›])
apply (*subst kf-comp-dependent-raw.rep-eq*)
using *assms bdd-F*
by (*simp add*: *flip*: *FF-def GG-def*)
also have ‹... = (*Set.filter* (λ(*E*,*x*). *E* ≠ *0*) ((λ(*E*,*z*). (*E*, *reorder z*)) ‘ (λ((*F*, *y*), (*E*, *x*)). (*E* o_{CL} *F*, *F*, *E*, *y*, *x*)) ‘
(*SIGMA* (*FG*, (-, -, -, *f*)):*Rep-kraus-family* (*kf-comp-dependent-raw* 𝔉 𝔊). *EE f*)))›
by (*simp add*: *image-image case-prod-beta*)
also have ‹... = (λ(*E*,*z*). (*E*, *reorder z*)) ‘ (*Set.filter* (λ(*E*,*x*). *E* ≠ *0*) ((λ((*F*, *y*), (*E*, *x*)). (*E* o_{CL} *F*, *F*, *E*, *y*, *x*)) ‘
(*SIGMA* (*FG*, (-, -, -, *f*)):*Rep-kraus-family* (*kf-comp-dependent-raw* 𝔉 𝔊). *EE f*)))›
by *force+*
also have ‹... = (λ(*E*,*x*). (*E*,*reorder x*)) ‘ *Rep-kraus-family*
(*kf-comp-dependent-raw* (λ(-, -, -, *y*). 𝔈 *y*) (*kf-comp-dependent-raw* 𝔉 𝔊))›
apply (*subst* (*2*) *kf-comp-dependent-raw.rep-eq*)
using *bdd2* **by** (*simp add*: *case-prod-unfold EE-def*)
also have ‹... = *Rep-kraus-family ?rhs*›
by (*simp add*: *kf-map-inj.rep-eq case-prod-beta*)
finally show ‹*Rep-kraus-family ?lhs* = *Rep-kraus-family ?rhs*›
by −
qed

lemma *kf-filter-comp-dependent*:
fixes 𝔉 :: ‹'*e* ⇒ ('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*f*) *kraus-family*›
and 𝔈 :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*e*) *kraus-family*›
assumes ‹*bdd-above* ((*kf-norm o* 𝔉) ‘ *kf-domain* 𝔈)›
shows ‹*kf-filter* (λ(*e*,*f*). *F e f* ∧ *E e*) (*kf-comp-dependent* 𝔉 𝔈)
= *kf-comp-dependent* (λ*e*. *kf-filter* (*F e*) (𝔉 *e*)) (*kf-filter E* 𝔈)›
proof −
from *assms*
have *bdd2*: ‹*bdd-above* ((λ*e*. *kf-norm* (*kf-filter* (*F e*) (𝔉 *e*))) ‘ *kf-domain* 𝔈)›
apply (*rule bdd-above-mono2*)
by (*auto intro*!: *kf-norm-filter*)
then have *bdd3*: ‹*bdd-above* ((λ*x*. *kf-norm* (*kf-filter* (*F x*) (𝔉 *x*))) ‘
kf-domain (*kf-filter E* 𝔈))›
apply (*rule bdd-above-mono2*)
by *auto*
show *?thesis*
unfolding *kf-comp-dependent-def kf-filter-map*

apply (*rule arg-cong*[**where** *f*=‹*kf-map* -›])
using *assms bdd2 bdd3* **apply** (*transfer′ fixing*: *F E*)
by (*auto intro*!: *simp*: *kf-filter.rep-eq case-prod-unfold image-iff Bex-def*)
qed

lemma *kf-comp-assoc-weak*:
fixes $\mathfrak{E}$ :: ‹('*c*::*chilbert-space*,'*d*::*chilbert-space*,'*e*) *kraus-family*›
and $\mathfrak{F}$ :: ‹('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*f*) *kraus-family*›
and $\mathfrak{G}$ :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*g*) *kraus-family*›
shows ‹*kf-comp* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$) $\mathfrak{G}$ =$_{kr}$ *kf-comp* $\mathfrak{E}$ (*kf-comp* $\mathfrak{F}$ $\mathfrak{G}$)›
apply (*rule kf-eq-weakI*)
by (*simp add*: *kf-comp-apply*)

lemma *kf-comp-dependent-raw-cong-left*:
assumes ‹*bdd-above* ((*kf-norm o* $\mathfrak{E}$) ‘ *kf-domain* $\mathfrak{F}$)›
assumes ‹*bdd-above* ((*kf-norm o* $\mathfrak{E}'$) ‘ *kf-domain* $\mathfrak{F}$)›
assumes ‹$\bigwedge x.$ *x* ∈ *snd* ‘ *Rep-kraus-family* $\mathfrak{F}$ ⟹ $\mathfrak{E}$ *x* = $\mathfrak{E}'$ *x*›
shows ‹*kf-comp-dependent-raw* $\mathfrak{E}$ $\mathfrak{F}$ = *kf-comp-dependent-raw* $\mathfrak{E}'$ $\mathfrak{F}$›
proof −
show *?thesis*
apply (*rule Rep-kraus-family-inject*[*THEN iffD1*])
using *assms*
by (*force simp*: *kf-comp-dependent-def kf-comp-dependent-raw.rep-eq*
image-iff case-prod-beta Bex-def)
qed

lemma *kf-comp-dependent-cong-left*:
assumes ‹*bdd-above* ((*kf-norm o* $\mathfrak{E}$) ‘ *kf-domain* $\mathfrak{F}$)›
assumes ‹*bdd-above* ((*kf-norm o* $\mathfrak{E}'$) ‘ *kf-domain* $\mathfrak{F}$)›
assumes ‹$\bigwedge x.$ *x* ∈ *kf-domain* $\mathfrak{F}$ ⟹ $\mathfrak{E}$ *x* = $\mathfrak{E}'$ *x*›
shows ‹*kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$ = *kf-comp-dependent* $\mathfrak{E}'$ $\mathfrak{F}$›
proof −
have ‹*kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$ = *kf-map* (λ(*F*, *E*, *y*). *y*) (*kf-comp-dependent-raw* $\mathfrak{E}$ $\mathfrak{F}$)›
by (*simp add*: *kf-comp-dependent-def id-def*)
also have ‹... = *kf-map* (λ(*F*, *E*, *y*). *y*) ((*kf-comp-dependent-raw* (λ*x*. ($\mathfrak{E}'$ *x*)) ($\mathfrak{F}$)))›
apply (*rule arg-cong*[**where** *f*=‹λ*t*. *kf-map* - *t*›])
apply (*rule kf-comp-dependent-raw-cong-left*[*OF assms*])
using *assms* **by** (*auto intro*!: *simp*: *kf-domain.rep-eq*)
also have ‹... = *kf-comp-dependent* $\mathfrak{E}'$ $\mathfrak{F}$›
by (*simp add*: *kf-comp-dependent-def id-def*)
finally show *?thesis*
by −
qed

lemma *kf-domain-comp-dependent-raw-subset*:
‹*kf-domain* (*kf-comp-dependent-raw* $\mathfrak{E}$ $\mathfrak{F}$) ⊆ *UNIV* × *UNIV* × (*SIGMA x*:*kf-domain* $\mathfrak{F}$. *kf-domain* ($\mathfrak{E}$ *x*))›

by (*auto intro*!: *simp*: *kf-comp-dependent-raw.rep-eq kf-domain.rep-eq image-iff Bex-def*)

lemma *kf-domain-comp-dependent-subset*:
‹*kf-domain* (*kf-comp-dependent* 𝔈 𝔉) ⊆ (*SIGMA* *x*:*kf-domain* 𝔉. *kf-domain* (𝔈 *x*))›
apply (*simp add*: *kf-comp-dependent-def kf-domain-map id-def*)
by (*auto intro*!: *simp*: *kf-comp-dependent-raw.rep-eq kf-domain.rep-eq image-iff Bex-def*)

lemma *kf-domain-comp-subset*: ‹*kf-domain* (*kf-comp* 𝔈 𝔉) ⊆ *kf-domain* 𝔉 × *kf-domain* 𝔈›
by (*metis Sigma-cong kf-comp-def kf-domain-comp-dependent-subset*)

lemma *kf-apply-comp-dependent-cong*:
fixes 𝔈 :: ‹'*f* ⇒ ('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*e1*) *kraus-family*›
and 𝔈' :: ‹'*f* ⇒ ('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*e2*) *kraus-family*›
and 𝔉 𝔉' :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*f*) *kraus-family*›
assumes *bdd*: ‹*bdd-above* ((*kf-norm* *o* 𝔈) ‘ *kf-domain* 𝔉)›
assumes *bdd'*: ‹*bdd-above* ((*kf-norm* *o* 𝔈') ‘ *kf-domain* 𝔉')›
assumes ‹*f* ∈ *kf-domain* 𝔉 ⟹ *kf-apply-on* (𝔈 *f*) *E* = *kf-apply-on* (𝔈' *f*) *E'*›
assumes ‹*kf-apply-on* 𝔉 {*f*} = *kf-apply-on* 𝔉' {*f*}›
shows ‹*kf-apply-on* (*kf-comp-dependent* 𝔈 𝔉) ({*f*}×*E*) = *kf-apply-on* (*kf-comp-dependent* 𝔈' 𝔉') ({*f*}×*E'*)›
proof (*rule ext*)
fix ϱ :: ‹('*a*, '*a*) *trace-class*›

have *rewrite-comp*: ‹*kf-apply-on* (*kf-comp-dependent* 𝔈 𝔉) ({*f*}×*E*) =
kf-apply (*kf-comp* (*kf-filter* (λ*x*. *x*∈*E*) (𝔈 *f*))
(*kf-filter* (λ*x*. *x*=*f*) 𝔉))›
if ‹*bdd-above* ((*kf-norm* *o* 𝔈) ‘ *kf-domain* 𝔉)›
for *E* **and** 𝔈 :: ‹'*f* ⇒ ('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*e*) *kraus-family*›
and 𝔉 :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*f*) *kraus-family*›
proof −
have *bdd-filter*: ‹*bdd-above* ((*kf-norm* ∘ (λ*f*. *kf-filter* (λ*x*. *x*∈*E*) (𝔈 *f*))) ‘ *kf-domain* 𝔉)›
apply (*rule bdd-above-mono2*[*OF* - *subset-refl*, *rotated*])
using *kf-norm-filter* **apply** *blast*
using *that*
by (*metis* (*mono-tags*, *lifting*) *bdd-above-mono2 comp-apply kf-norm-filter order.refl*)
have *aux*: ‹(λ(*x*,*y*). *y*∈*E* ∧ *x*=*f*) = (λ*x*. *x*∈{*f*}×*E*)›
by *auto*
have ∗: ‹*kf-filter* (λ*x*. *x*∈{*f*}×*E*) (*kf-comp-dependent* 𝔈 𝔉)
= *kf-comp-dependent* (λ*f*. *kf-filter* (λ*x*. *x*∈*E*) (𝔈 *f*))
(*kf-filter* (λ*x*. *x*=*f*) 𝔉)›
using *kf-filter-comp-dependent*[**where** 𝔈=𝔉 **and** 𝔉=𝔈 **and** *F*=‹λ- *x*. *x*∈*E*› **and** *E*=‹λ*x*. *x*=*f*›,
OF that]
unfolding *aux* **by** *auto*
have ‹*kf-apply-on* (*kf-comp-dependent* 𝔈 𝔉) ({*f*}×*E*)
= *kf-apply* (*kf-comp-dependent* (λ*f*. *kf-filter* (λ*x*. *x*∈*E*) (𝔈 *f*))
(*kf-filter* (λ*x*. *x*=*f*) 𝔉))›
by (*auto intro*!: *simp*: *kf-apply-on-def* ∗)
also have ‹... = *kf-apply*

```
    (kf-comp-dependent (λ-. kf-filter (λx. x∈E) (𝔈 f))
      (kf-filter (λx. x=f) 𝔉))›
     apply (rule arg-cong[where f=λz. kf-apply z])
     apply (rule kf-comp-dependent-cong-left)
     using bdd-filter by auto
   also have ‹... = kf-apply (kf-comp (kf-filter (λx. x∈E) (𝔈 f))
     (kf-filter (λx. x=f) 𝔉))›
     by (simp add: kf-comp-def)
   finally show ?thesis
     by −
 qed

 have rew-𝔈: ‹kf-apply (kf-filter (λx. x∈E) (𝔈 f))
    = kf-apply (kf-filter (λx. x∈E') (𝔈' f))›
   if ‹f ∈ kf-domain 𝔉›
   using assms(3)[OF that]
   by (simp add: kf-apply-on-def)
 have rew-𝔉: ‹kf-apply (kf-filter (λf'. f' = f) 𝔉')
   = kf-apply (kf-filter (λf'. f' = f) 𝔉)›
   using assms(4)
   by (simp add: kf-apply-on-def)
 have 𝔉-0: ‹kf-apply (kf-filter (λf'. f' = f) 𝔉) = 0›
   if ‹f ∉ kf-domain 𝔉›
 proof −


   have ‹kf-filter (λf'. f' = f) 𝔉 ≡kr
         kf-filter (λf'. f' = f) (kf-filter (λx. x ∈ kf-domain 𝔉) 𝔉)›
     by (auto intro!: kf-filter-cong intro: kf-eq-sym simp: kf-filter-to-domain)
   also have ‹... ≡kr (kf-filter (λ-. False) 𝔉)›
     using that apply (simp add: kf-filter-twice del: kf-filter-false)
     by (smt (verit) kf-eq-def kf-filter-cong-eq)
   also have ‹... ≡kr 0›
     by (simp add: kf-filter-false)
   finally show ?thesis
     by (metis kf-apply-0 kf-eq-imp-eq-weak kf-eq-weak-def)
 qed

 show ‹kf-comp-dependent 𝔈 𝔉 *kr @({f} × E) ϱ =
   kf-comp-dependent 𝔈' 𝔉' *kr @({f} × E') ϱ›
   apply (cases ‹f ∈ kf-domain 𝔉›)
   by (auto intro!: ext simp add: rewrite-comp[OF bdd] rewrite-comp[OF bdd']
       kf-comp-apply rew-𝔈 rew-𝔉 𝔉-0)
qed



lemma kf-comp-dependent-cong-weak:
 fixes 𝔈 :: ‹'f ⇒ ('b::chilbert-space,'c::chilbert-space,'e1) kraus-family›
```

and $\mathfrak{E}'$ :: ‹'f ⇒ ('b::chilbert-space,'c::chilbert-space,'e2) kraus-family›
and $\mathfrak{F}$ $\mathfrak{F}'$ :: ‹('a::chilbert-space,'b::chilbert-space,'f) kraus-family›
assumes bdd: ‹bdd-above ((kf-norm o $\mathfrak{E}$) ' kf-domain $\mathfrak{F}$)›
assumes eq: ‹⋀x. x ∈ kf-domain $\mathfrak{F}$ ⟹ $\mathfrak{E}$ x =$_{kr}$ $\mathfrak{E}'$ x›
assumes ‹$\mathfrak{F}$ ≡$_{kr}$ $\mathfrak{F}'$›
shows ‹kf-comp-dependent $\mathfrak{E}$ $\mathfrak{F}$ =$_{kr}$ kf-comp-dependent $\mathfrak{E}'$ $\mathfrak{F}'$›
proof −
have ‹kf-apply-on (kf-comp-dependent $\mathfrak{E}$ $\mathfrak{F}$) ({f}×UNIV) = kf-apply-on (kf-comp-dependent $\mathfrak{E}'$ $\mathfrak{F}'$) ({f}×UNIV)› **for** f
proof −
note bdd
moreover have ‹bdd-above ((kf-norm ∘ $\mathfrak{E}'$) ' kf-domain $\mathfrak{F}'$)›
by (metis (no-types, lifting) assms(1) assms(2) assms(3) comp-apply image-cong kf-norm-cong kf-domain-cong)
moreover have ‹kf-apply-on ($\mathfrak{E}$ x) UNIV = kf-apply-on ($\mathfrak{E}'$ x) UNIV› **if** ‹x ∈ kf-domain $\mathfrak{F}$› **for** x
using assms(2) kf-eq-weak-def that
by (metis kf-apply-on-UNIV)
moreover have ‹kf-apply-on $\mathfrak{F}$ {f} = kf-apply-on $\mathfrak{F}'$ {f}›
by (meson assms(3) kf-eq-def)
ultimately show ?thesis
by (rule kf-apply-comp-dependent-cong)
qed
then have ‹kf-apply-on (kf-comp-dependent $\mathfrak{E}$ $\mathfrak{F}$) (⋃f. {f}×UNIV) = kf-apply-on (kf-comp-dependent $\mathfrak{E}'$ $\mathfrak{F}'$) (⋃f. {f}×UNIV)›
apply (rule-tac ext)
apply (rule kf-apply-on-union-eqI[**where** F=‹range (λf. ({f}×UNIV,{f}×UNIV))›])
by auto
moreover have ‹(⋃f. {f} × UNIV) = UNIV›
by fast
ultimately show ?thesis
by (metis kf-eq-weak-def kf-apply-on-UNIV)
qed

lemma kf-comp-dependent-assoc:
fixes $\mathfrak{E}$ :: ‹'g ⇒ 'f ⇒ ('c::chilbert-space,'d::chilbert-space,'e) kraus-family›
and $\mathfrak{F}$ :: ‹'g ⇒ ('b::chilbert-space,'c::chilbert-space,'f) kraus-family›
and $\mathfrak{G}$ :: ‹('a::chilbert-space,'b::chilbert-space,'g) kraus-family›
assumes bdd-E: ‹bdd-above ((kf-norm o case-prod $\mathfrak{E}$) ' (SIGMA x:kf-domain $\mathfrak{G}$. kf-domain ($\mathfrak{F}$ x)))›
assumes bdd-F: ‹bdd-above ((kf-norm o $\mathfrak{F}$) ' kf-domain $\mathfrak{G}$)›
shows ‹(kf-comp-dependent (λg. kf-comp-dependent ($\mathfrak{E}$ g) ($\mathfrak{F}$ g)) $\mathfrak{G}$) ≡$_{kr}$
kf-map (λ((g,f),e). (g,f,e)) (kf-comp-dependent (λ(g,f). $\mathfrak{E}$ g f) (kf-comp-dependent $\mathfrak{F}$ $\mathfrak{G}$))›
(**is** ‹?lhs ≡$_{kr}$?rhs›)
proof (rule kf-eqI)
fix gfe :: ‹'g × 'f × 'e› **and** ϱ
obtain g f e **where** gfe-def: ‹gfe = (g,f,e)›
apply atomize-elim

apply (*rule exI*[*of* - ‹*fst gfe*›])
apply (*rule exI*[*of* - ‹*fst* (*snd gfe*)›])
apply (*rule exI*[*of* - ‹*snd* (*snd gfe*)›])
by *simp*
have *aux*: ‹(λ*x*. (*fst* (*fst x*), *snd* (*fst x*), *snd x*) = *gfe*) = (λ*x*. *x*=((*g*,*f*),*e*))›
by (*auto simp*: *gfe-def*)
have *bdd1*: ‹*bdd-above* ((λ*x*. *kf-norm* (*kf-filter* (λ*x*. *x* = *f*) (𝔉 *x*))) ‘ *kf-domain* 𝔊)›
using *kf-norm-filter bdd-F*
by (*metis* (*mono-tags*, *lifting*) *bdd-above-mono2 o-apply order.refl*)
from *bdd-E* **have** *bdd2*: ‹*bdd-above* ((*kf-norm* ∘ 𝔈 *g*) ‘ *kf-domain* (𝔉 *g*))› **if** ‹*g* ∈ *kf-domain* 𝔊› **for** *g*
apply (*rule bdd-above-mono*)
using *that*
by (*force simp*: *image-iff*)
from *bdd-E* **have** *bdd3*: ‹*bdd-above* ((λ*x*. *kf-norm* (*kf-filter* (λ*x*. *x* = *e*) (*case-prod* 𝔈 *x*))) ‘ (*SIGMA x*:*kf-domain* 𝔊. *kf-domain* (𝔉 *x*)))›
apply (*rule bdd-above-mono2*)
by (*auto simp*: *kf-norm-filter*)
then have *bdd4*: ‹*bdd-above* ((λ*x*. *kf-norm* (*kf-filter* (λ*x*. *x* = *e*) (𝔈 (*fst x*) (*snd x*)))) ‘ *X*)›
if ‹*X* ⊆ (*SIGMA x*:*kf-domain* 𝔊. *kf-domain* (𝔉 *x*))› **for** *X*
apply (*rule bdd-above-mono2*)
using *that* **by** (*auto simp*: *bdd-above-def*)
have *bdd5*: ‹*bdd-above* ((*kf-norm* ∘ (λ*g*. *kf-comp-dependent* (𝔈 *g*) (𝔉 *g*))) ‘ *X*)›
if *X*𝔊: ‹*X* ⊆ *kf-domain* 𝔊› **for** *X*
proof −
from *bdd-E*
obtain *BE′* **where** *BE′*: ‹*g* ∈ *kf-domain* 𝔊 ⟹ *x* ∈ *kf-domain* (𝔉 *g*) ⟹ *kf-norm* (𝔈 *g x*) ≤ *BE′*› **for** *g x*
by (*auto simp*: *bdd-above-def*)
define *BE* **where** ‹*BE* = *max BE′ 0›
from *BE′* **have** *BE*: ‹*g* ∈ *kf-domain* 𝔊 ⟹ *x* ∈ *kf-domain* (𝔉 *g*) ⟹ *kf-norm* (𝔈 *g x*) ≤ *BE*› **and** ‹*BE* ≥ *0*› **for** *g x*
by (*force simp*: *BE-def*)+
then have ‹*BE* ≥ *0*›
by (*smt* (*z3*) *kf-norm-geq0*)
from *bdd-F* **obtain** *BF* **where** *BF*: ‹*kf-norm* (𝔉 *x*) ≤ *BF*› **if** ‹*x* ∈ *kf-domain* 𝔊› **for** *x*
by (*auto simp*: *bdd-above-def*)
have ‹*kf-norm* (*kf-comp-dependent* (𝔈 *g*) (𝔉 *g*))
≤ *BE* ∗ *kf-norm* (𝔉 *g*)› **if** ‹*g* ∈ *kf-domain* 𝔊› **for** *g*
apply (*rule kf-comp-dependent-norm-leq*[*OF BE* ‹*BE* ≥ *0*›])
using *that* **by** *auto*
then have ‹*kf-norm* (*kf-comp-dependent* (𝔈 *g*) (𝔉 *g*)) ≤ *BE* ∗ *BF*› **if** ‹*g* ∈ *kf-domain* 𝔊› **for** *g*
apply (*rule order-trans*)
using *BF* ‹*BE* ≥ *0*› *that*
by (*auto intro*!: *mult-left-mono*)
with *X*𝔊 **show** *?thesis*
by (*auto intro*!: *bdd-aboveI*)
qed

have ‹*?lhs* $*_{kr}$ @{*gfe*} ϱ
  = *kf-comp-dependent*
    (λ*g*. *kf-filter* (λ*x*. *x*=(*f*,*e*)) (*kf-comp-dependent* ($\mathfrak{E}$ *g*) ($\mathfrak{F}$ *g*)))
    (*kf-filter* (λ*x*. *x*=*g*) $\mathfrak{G}$) $*_{kr}$ ϱ›
 unfolding *kf-apply-on-def*
 apply (*subst kf-filter-comp-dependent*[*symmetric*])
  apply (*rule bdd5*, *rule order-refl*)
 apply (*subst asm-rl*[*of* ‹(λ(*x*, *y*). *y* = (*f*, *e*) ∧ *x* = *g*) = (λ*x*. *x* ∈ {*gfe*})›])
 by (*auto simp*: *gfe-def*)
also have ‹... = *kf-comp-dependent*
    (λ*g*. *kf-comp-dependent*
        (λ*f*. *kf-filter* (λ*x*. *x*=*e*) ($\mathfrak{E}$ *g f*)) (*kf-filter* (λ*x*. *x*=*f*) ($\mathfrak{F}$ *g*)))
    (*kf-filter* (λ*x*. *x*=*g*) $\mathfrak{G}$) $*_{kr}$ ϱ›
 apply (*rule kf-apply-eqI*)
 apply (*rule kf-comp-dependent-cong-weak*[*OF* - - *kf-eq-refl*])
  apply *fastforce*
 apply (*subst kf-filter-comp-dependent*[*symmetric*])
 using *bdd2* **apply** *fastforce*
 apply (*subst asm-rl*[*of* ‹(λ(*ea*, *fa*). *fa* = *e* ∧ *ea* = *f*) = (λ*x*. *x* = (*f*, *e*))›])
 by *auto*
also have ‹... = *kf-comp-dependent*
    (λ-. *kf-comp-dependent*
        (λ*f*. *kf-filter* (λ*x*. *x*=*e*) ($\mathfrak{E}$ *g f*)) (*kf-filter* (λ*x*. *x*=*f*) ($\mathfrak{F}$ *g*)))
    (*kf-filter* (λ*x*. *x*=*g*) $\mathfrak{G}$) $*_{kr}$ ϱ›
 apply (*rule arg-cong*[**where** *f*=‹λ*t*. *kf-apply t* ϱ›])
 apply (*rule kf-comp-dependent-cong-left*)
 by (*auto intro*!: *simp*: *kf-domain.rep-eq kf-filter.rep-eq*)
also have ‹... = *kf-comp-dependent*
    (λ-. *kf-comp-dependent*
        (λ-. *kf-filter* (λ*x*. *x*=*e*) ($\mathfrak{E}$ *g f*)) (*kf-filter* (λ*x*. *x*=*f*) ($\mathfrak{F}$ *g*)))
    (*kf-filter* (λ*x*. *x*=*g*) $\mathfrak{G}$) $*_{kr}$ ϱ›
 apply (*rule arg-cong*[**where** *f*=‹λ*t*. *kf-comp-dependent t* - $*_{kr}$ ϱ›])
 apply (*rule ext*)
 apply (*rule kf-comp-dependent-cong-left*)
 by (*auto intro*!: *simp*: *kf-domain.rep-eq kf-filter.rep-eq*)
also have ‹... = *kf-comp*
    (*kf-comp*
        (*kf-filter* (λ*x*. *x*=*e*) ($\mathfrak{E}$ *g f*)) (*kf-filter* (λ*x*. *x*=*f*) ($\mathfrak{F}$ *g*)))
    (*kf-filter* (λ*x*. *x*=*g*) $\mathfrak{G}$) $*_{kr}$ ϱ›
 by (*simp add*: *kf-comp-def*)
also have ‹... = *kf-comp* (*kf-filter* (λ*x*. *x*=*e*) ($\mathfrak{E}$ *g f*))
      (*kf-comp* (*kf-filter* (λ*x*. *x*=*f*) ($\mathfrak{F}$ *g*)) (*kf-filter* (λ*x*. *x*=*g*) $\mathfrak{G}$)) $*_{kr}$ ϱ›
 by (*simp add*: *kf-comp-assoc-weak*[*unfolded kf-eq-weak-def*])
also have ‹... = *kf-comp-dependent* (λ-. *kf-filter* (λ*x*. *x*=*e*) ($\mathfrak{E}$ *g f*))
    (*kf-comp-dependent* (λ-. *kf-filter* (λ*x*. *x*=*f*) ($\mathfrak{F}$ *g*))
        (*kf-filter* (λ*x*. *x*=*g*) $\mathfrak{G}$)) $*_{kr}$ ϱ›
 by (*simp add*: *kf-comp-def*)
also have ‹... = *kf-comp-dependent* (λ(*g*,*f*). *kf-filter* (λ*x*. *x*=*e*) ($\mathfrak{E}$ *g f*))

```
          (kf-comp-dependent (λ-. kf-filter (λx. x=f) (𝔉 g))
                                    (kf-filter (λx. x=g) 𝔊)) *kr ϱ›
    apply (rule arg-cong[where f=‹λt. t *kr ϱ›])
    apply (rule kf-comp-dependent-cong-left)
    apply force
   using kf-domain-comp-dependent-subset apply (fastforce intro!: bdd4 simp: o-def case-prod-unfold)
    using kf-domain-comp-dependent-subset[of ‹(λ-. kf-filter (λx. x = f) (𝔉 g))› ‹kf-filter (λx. x
= g) 𝔊›]
    by (auto intro!: simp: kf-filter.rep-eq case-prod-unfold)
  also have ‹... = kf-comp-dependent (λ(g,f). kf-filter (λx. x=e) (𝔈 g f))
          (kf-comp-dependent (λg. kf-filter (λx. x=f) (𝔉 g))
                                    (kf-filter (λx. x=g) 𝔊)) *kr ϱ›
    apply (rule arg-cong[where f=‹λt. kf-comp-dependent - t *kr -›])
    apply (rule kf-comp-dependent-cong-left)
    by (auto intro!: simp: kf-domain.rep-eq kf-filter.rep-eq)
  also have ‹... = kf-comp-dependent (λ(g,f). kf-filter (λx. x=e) (𝔈 g f))
          (kf-filter (λx. x=(g,f)) (kf-comp-dependent 𝔉 𝔊)) *kr ϱ›
    apply (subst kf-filter-comp-dependent[symmetric])
    using bdd-F apply (simp add: bdd-above-mono2)
    apply (subst asm-rl[of ‹(λ(e, fa). fa = f ∧ e = g) = (λx. x = (g, f))›])
    by auto
  also have ‹... = kf-filter (λx. x=((g,f),e))
      (kf-comp-dependent (λ(g,f). 𝔈 g f) (kf-comp-dependent 𝔉 𝔊)) *kr ϱ›
    unfolding case-prod-beta
    apply (subst kf-filter-comp-dependent[symmetric])
   using bdd-E kf-domain-comp-dependent-subset[of 𝔉 𝔊] apply (fastforce simp: bdd-above-def)
    apply (subst asm-rl[of ‹(λ(ea, fa). fa = e ∧ ea = (g, f)) = (λx. x = ((g, f), e))›])
    by auto
  also have ‹... = kf-filter (λx. x=gfe)
       (kf-map (λ((g, f), e). (g, f, e))
      (kf-comp-dependent (λ(g,f). 𝔈 g f) (kf-comp-dependent 𝔉 𝔊))) *kr ϱ›
    apply (simp add: kf-filter-map case-prod-beta aux)
    apply (subst aux)
    by simp
  also have ‹... = ?rhs *kr @{gfe} ϱ›
    by (simp add: kf-apply-on-def)
  finally show ‹?lhs *kr @{gfe} ϱ = ?rhs *kr @{gfe} ϱ›
    by −
qed

lemma kf-comp-dependent-assoc-weak:
  fixes 𝔈 :: ‹'g ⇒ 'f ⇒ ('c::chilbert-space,'d::chilbert-space,'e) kraus-family›
    and 𝔉 :: ‹'g ⇒ ('b::chilbert-space,'c::chilbert-space,'f) kraus-family›
    and 𝔊 :: ‹('a::chilbert-space,'b::chilbert-space,'g) kraus-family›
  assumes bdd-E: ‹bdd-above ((kf-norm o case-prod 𝔈) ‘ (SIGMA x:kf-domain 𝔊. kf-domain (𝔉
x)))›
  assumes bdd-F: ‹bdd-above ((kf-norm o 𝔉) ‘ kf-domain 𝔊)›
  shows ‹kf-comp-dependent (λg. kf-comp-dependent (𝔈 g) (𝔉 g)) 𝔊 =kr
        kf-comp-dependent (λ(g,f). 𝔈 g f) (kf-comp-dependent 𝔉 𝔊)›
```

using *kf-comp-dependent-assoc*[*OF assms*, *THEN kf-eq-imp-eq-weak*]
by (*metis* (*no-types*, *lifting*) *kf-apply-map kf-eq-weak-def*)

lemma *kf-comp-dependent-comp-assoc-weak*:
fixes $\mathfrak{E}$ :: ‹('*c*::*chilbert-space*,'*d*::*chilbert-space*,'*e*) *kraus-family*›
and $\mathfrak{F}$ :: ‹'*g* ⇒ ('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*f*) *kraus-family*›
and $\mathfrak{G}$ :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*g*) *kraus-family*›
assumes ‹*bdd-above* ((*kf-norm o* $\mathfrak{F}$) ' *kf-domain* $\mathfrak{G}$)›
shows ‹*kf-comp-dependent* (λ*g*. *kf-comp* $\mathfrak{E}$ ($\mathfrak{F}$ *g*)) $\mathfrak{G}$ =$_{kr}$
kf-comp $\mathfrak{E}$ (*kf-comp-dependent* $\mathfrak{F}$ $\mathfrak{G}$)›
using *kf-comp-dependent-assoc-weak*[**where** $\mathfrak{E}$=‹λ- -. $\mathfrak{E}$› **and** $\mathfrak{G}$=$\mathfrak{G}$, *OF* - *assms*]
by (*fastforce simp*: *case-prod-unfold kf-comp-def*)

lemma *kf-comp-comp-dependent-assoc-weak*:
fixes $\mathfrak{E}$ :: ‹'*f* ⇒ ('*c*::*chilbert-space*,'*d*::*chilbert-space*,'*e*) *kraus-family*›
and $\mathfrak{F}$ :: ‹('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*f*) *kraus-family*›
and $\mathfrak{G}$ :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*g*) *kraus-family*›
assumes *bdd-E*: ‹*bdd-above* ((*kf-norm o* $\mathfrak{E}$) ' *kf-domain* $\mathfrak{F}$)›
shows ‹*kf-comp* (*kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$) $\mathfrak{G}$ =$_{kr}$
kf-comp-dependent (λ(-,*f*). $\mathfrak{E}$ *f*) (*kf-comp* $\mathfrak{F}$ $\mathfrak{G}$)›
proof −
from *bdd-E* **have** ‹*bdd-above* ((*kf-norm* ∘ (λ(*g*, *y*). $\mathfrak{E}$ *y*)) ' (*kf-domain* $\mathfrak{G}$ × *kf-domain* $\mathfrak{F}$))›
by (*auto intro*!: *simp*: *bdd-above-def*)
then have ‹*kf-comp-dependent* (λ-. *kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$) $\mathfrak{G}$ =$_{kr}$ *kf-comp-dependent* (λ(-, *f*). $\mathfrak{E}$ *f*) (*kf-comp-dependent* (λ-. $\mathfrak{F}$) $\mathfrak{G}$)›
apply (*rule kf-comp-dependent-assoc-weak*)
by *auto*
then show *?thesis*
by (*simp add*: *case-prod-unfold kf-comp-def*)
qed

lemma *kf-comp-assoc*:
fixes $\mathfrak{E}$ :: ‹('*c*::*chilbert-space*,'*d*::*chilbert-space*,'*e*) *kraus-family*›
and $\mathfrak{F}$ :: ‹('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*f*) *kraus-family*›
and $\mathfrak{G}$ :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*g*) *kraus-family*›
shows ‹*kf-comp* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$) $\mathfrak{G}$ ≡$_{kr}$
kf-map (λ((*g*,*f*),*e*). (*g*,*f*,*e*)) (*kf-comp* $\mathfrak{E}$ (*kf-comp* $\mathfrak{F}$ $\mathfrak{G}$))›
apply (*simp add*: *kf-comp-def*)
apply (*rule kf-eq-trans*)
apply (*rule kf-comp-dependent-assoc*)
by (*auto simp*: *case-prod-unfold*)

lemma *kf-comp-dependent-cong*:
fixes $\mathfrak{E}$ $\mathfrak{E}'$ :: ‹'*f* ⇒ ('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*e*) *kraus-family*›
and $\mathfrak{F}$ $\mathfrak{F}'$ :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*f*) *kraus-family*›
assumes *bdd*: ‹*bdd-above* ((*kf-norm o* $\mathfrak{E}$) ' *kf-domain* $\mathfrak{F}$)›
assumes ‹⋀*x*. *x* ∈ *kf-domain* $\mathfrak{F}$ ⟹ $\mathfrak{E}$ *x* ≡$_{kr}$ $\mathfrak{E}'$ *x*›
assumes ‹$\mathfrak{F}$ ≡$_{kr}$ $\mathfrak{F}'$›
shows ‹*kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$ ≡$_{kr}$ *kf-comp-dependent* $\mathfrak{E}'$ $\mathfrak{F}'$›

proof (*rule kf-eqI*)
fix ϱ :: ‹('a, 'a) *trace-class*›
fix x :: ‹'f × 'e›

note *bdd*
moreover have *bdd'*: ‹*bdd-above* ((*kf-norm* ∘ $\mathfrak{E}'$) ' *kf-domain* $\mathfrak{F}'$)›
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *image-cong kf-eq-imp-eq-weak kf-norm-cong kf-domain-cong o-apply*)
moreover have ‹*kf-apply-on* ($\mathfrak{E}$ *xa*) {*snd x*} = *kf-apply-on* ($\mathfrak{E}'$ *xa*) {*snd x*}› **if** ‹*xa* ∈ *kf-domain* $\mathfrak{F}$› **for** *xa*
by (*meson assms*(*2*) *kf-eq-def that*)
moreover have ‹*kf-apply-on* $\mathfrak{F}$ {*fst x*} = *kf-apply-on* $\mathfrak{F}'$ {*fst x*}›
by (*meson assms*(*3*) *kf-eq-def*)
ultimately have ‹*kf-apply-on* (*kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$) ({*fst x*} × {*snd x*}) = *kf-apply-on* (*kf-comp-dependent* $\mathfrak{E}'$ $\mathfrak{F}'$) ({*fst x*} × {*snd x*})›
by (*rule kf-apply-comp-dependent-cong*)
then show ‹*kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ @{*x*} ϱ = *kf-comp-dependent* $\mathfrak{E}'$ $\mathfrak{F}'$ $*_{kr}$ @{*x*} ϱ›
by *simp*
qed

lemma *kf-comp-cong*:
fixes $\mathfrak{E}$ $\mathfrak{E}'$ :: ‹('b::*chilbert-space*,'c::*chilbert-space*,'e) *kraus-family*›
and $\mathfrak{F}$ $\mathfrak{F}'$ :: ‹('a::*chilbert-space*,'b::*chilbert-space*,'f) *kraus-family*›
assumes ‹$\mathfrak{E}$ $\equiv_{kr}$ $\mathfrak{E}'$›
assumes ‹$\mathfrak{F}$ $\equiv_{kr}$ $\mathfrak{F}'$›
shows ‹*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$ $\equiv_{kr}$ *kf-comp* $\mathfrak{E}'$ $\mathfrak{F}'$›
by (*auto intro*!: *kf-comp-dependent-cong assms*
simp add: *kf-comp-def*)

lemma *kf-bound-comp-dependent-raw-of-op*:
shows ‹*kf-bound* (*kf-comp-dependent-raw* $\mathfrak{E}$ (*kf-of-op U*))
= *sandwich* (*U*∗) (*kf-bound* ($\mathfrak{E}$ ()))›
proof −
write *compose-wot* (**infixl** o_W *55*)
define *EE* **where** ‹*EE* = *Rep-kraus-family* ($\mathfrak{E}$ ())›

have *sum1*: ‹*summable-on-in cweak-operator-topology* (λ(*E*,*x*). *E*∗ o_{CL} *E*) *EE*›
by (*simp add*: *EE-def kf-bound-summable*)
then have *sum2*: ‹*summable-on-in cweak-operator-topology* (λ(*E*,*x*). *U*∗ o_{CL} (*E*∗ o_{CL} *E*)) *EE*›
by (*simp add*: *case-prod-unfold summable-on-in-wot-compose-left*)

have ‹*kf-bound* (*kf-comp-dependent-raw* $\mathfrak{E}$ (*kf-of-op U*)) =
infsum-in cweak-operator-topology (λ(*E*, *x*). *E*∗ o_{CL} *E*)
(*Set.filter* (λ(*EF*,-). *EF* ≠ *0*) (((λ((*F*, *y*), (*E*, *x*)). ((*E* o_{CL} *F*, *F*, *E*, *y*, *x*))) ' (*SIGMA* (*F*, *y*):*if U=0 then* {} *else* {(*U*, ())}. *EE*)))›
by (*simp add*: *kf-bound.rep-eq kf-comp-dependent-raw.rep-eq kf-of-op.rep-eq EE-def*)
also have ‹... = *infsum-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*)

```
        ((λ((F, y), (E, x)). ((E o_CL F, F, E, y, x))) ‘ (SIGMA (F, y):{(U, ())}. EE))›
    apply (cases ‹U=0›; rule infsum-in-cong-neutral)
    by force+
  also have ‹... = infsum-in cweak-operator-topology (λ(E, x). E* o_CL E)
                ((λ(E,x). (E o_CL U, U, E, (), x)) ‘ EE)›
    apply (rule arg-cong[where f=‹infsum-in - -›])
    by force
  also have ‹... = infsum-in cweak-operator-topology (λ(E, x). (E o_CL U)* o_CL (E o_CL U))
EE›
    apply (subst infsum-in-reindex)
    by (auto intro!: inj-onI simp: o-def case-prod-unfold infsum-in-reindex)
  also have ‹... = infsum-in cweak-operator-topology (λ(E, x). U* o_CL (E* o_CL E) o_CL U)
EE›
    by (metis (no-types, lifting) adj-cblinfun-compose cblinfun-assoc-left(1))
  also have ‹... = U* o_CL infsum-in cweak-operator-topology (λ(E,x). E* o_CL E) EE o_CL U›
   using sum1 sum2 by (simp add: case-prod-unfold infsum-in-wot-compose-right infsum-in-wot-compose-left)
  also have ‹... = sandwich (U*) (kf-bound (𝔈 ()))›
    by (simp add: EE-def kf-bound.rep-eq sandwich-apply)
  finally show ?thesis
    by −
qed


lemma kf-bound-comp-dependent-of-op:
  shows ‹kf-bound (kf-comp-dependent 𝔈 (kf-of-op U)) = sandwich (U*) (kf-bound (𝔈 ()))›
  by (simp add: kf-comp-dependent-def kf-map-bound kf-bound-comp-dependent-raw-of-op)

lemma kf-bound-comp-of-op:
  shows ‹kf-bound (kf-comp 𝔈 (kf-of-op U)) = sandwich (U*) (kf-bound 𝔈)›
  by (simp add: kf-bound-comp-dependent-of-op kf-comp-def)

lemma kf-norm-comp-dependent-of-op-coiso:
  assumes ‹isometry (U*)›
  shows ‹kf-norm (kf-comp-dependent 𝔈 (kf-of-op U)) = kf-norm (𝔈 ())›
  using assms
  by (simp add: kf-bound-comp-dependent-of-op kf-norm-def sandwich-apply
      norm-isometry-compose norm-isometry-compose')


lemma kf-norm-comp-of-op-coiso:
  assumes ‹isometry (U*)›
  shows ‹kf-norm (kf-comp 𝔈 (kf-of-op U)) = kf-norm 𝔈›
  using assms
  by (simp add: kf-bound-comp-of-op kf-norm-def sandwich-apply
      norm-isometry-compose norm-isometry-compose')

lemma kf-bound-comp-dependend-raw-iso:
  assumes ‹isometry U›
  shows ‹kf-bound (kf-comp-dependent-raw (λ-. kf-of-op U) 𝔈)
```

```
      = kf-bound 𝔈›
proof −
  write compose-wot (infixl o_W 55)
  define EE where ‹EE = Rep-kraus-family 𝔈›

  have ‹bdd-above ((λx. (norm U)²) ` kf-domain 𝔈)›
    by auto
  then have ‹kf-bound (kf-comp-dependent-raw (λ-. kf-of-op U) 𝔈) =
        infsum-in cweak-operator-topology (λ(E, x). E* o_CL E)
          (Set.filter (λ(EF, -). EF ≠ 0) ((λ((F, y), (E, x)). (E o_CL F, F, E, y, ())) ` (SIGMA
(F, y):EE. if U=0 then {} else {(U, ())})))›
    by (simp add: kf-bound.rep-eq kf-comp-dependent-raw.rep-eq kf-of-op.rep-eq EE-def)
  also have ‹... = infsum-in cweak-operator-topology (λ(E, x). E* o_CL E)
          ((λ((F, y), (E, x)). (E o_CL F, F, E, y, ())) ` (SIGMA (F, y):EE. {(U, ())}))›
    apply (cases ‹U = 0›; rule infsum-in-cong-neutral)
    by force+
  also have ‹...  = infsum-in cweak-operator-topology (λ(E, x). E* o_CL E)
                ((λ(E,x). (U o_CL E, E, U, x, ())) ` EE)›
    apply (rule arg-cong[where f=‹infsum-in - -›])
    by force
  also have ‹... = infsum-in cweak-operator-topology (λ(E, x). (U o_CL E)* o_CL (U o_CL E))
EE›
    apply (subst infsum-in-reindex)
    by (auto intro!: inj-onI simp: o-def case-prod-unfold infsum-in-reindex)
  also have ‹... = infsum-in cweak-operator-topology (λ(E, x). E* o_CL (U* o_CL U) o_CL E)
EE›
    by (metis (no-types, lifting) adj-cblinfun-compose cblinfun-assoc-left(1))
  also have ‹... = infsum-in cweak-operator-topology (λ(E,x). E* o_CL E) EE›
    using assms by simp
  also have ‹... = kf-bound 𝔈›
    by (simp add: EE-def kf-bound.rep-eq sandwich-apply)
  finally show ?thesis
    by −
qed

lemma kf-bound-comp-dependent-iso:
  assumes ‹isometry U›
  shows ‹kf-bound (kf-comp-dependent (λ-. kf-of-op U) 𝔈) = kf-bound 𝔈›
 using assms by (simp add: kf-comp-dependent-def kf-map-bound kf-bound-comp-dependend-raw-iso)

lemma kf-bound-comp-iso:
  assumes ‹isometry U›
  shows ‹kf-bound (kf-comp (kf-of-op U) 𝔈) = kf-bound 𝔈›
  using assms by (simp add: kf-bound-comp-dependent-iso kf-comp-def)

lemma kf-norm-comp-dependent-iso:
  assumes ‹isometry U›
  shows ‹kf-norm (kf-comp-dependent (λ-. kf-of-op U) 𝔈) = kf-norm 𝔈›
  using assms
```

by (*simp add*: *kf-bound-comp-dependent-iso kf-norm-def sandwich-apply norm-isometry-compose norm-isometry-compose'*)

lemma *kf-norm-comp-iso*:
assumes ‹*isometry U*›
shows ‹*kf-norm* (*kf-comp* (*kf-of-op U*) $\mathfrak{E}$) = *kf-norm* $\mathfrak{E}$›
using *assms*
by (*simp add*: *kf-bound-comp-iso kf-norm-def sandwich-apply norm-isometry-compose norm-isometry-compose'*)

lemma *kf-comp-dependent-raw-0-right*[*simp*]: ‹*kf-comp-dependent-raw* $\mathfrak{E}$ *0* = *0*›
apply *transfer'*
by (*auto intro*!: *simp*: *zero-kraus-family.rep-eq*)

lemma *kf-comp-dependent-raw-0-left*[*simp*]: ‹*kf-comp-dependent-raw 0* $\mathfrak{E}$ = *0*›
apply *transfer'*
by (*auto intro*!: *simp*: *zero-kraus-family.rep-eq*)

lemma *kf-comp-dependent-0-left*[*simp*]: ‹*kf-comp-dependent* (λ-. *0*) *E* = *0*›
proof −
have ‹*bdd-above* ((*kf-norm* ∘ *0*) ‘ *kf-domain E*)›
by *auto*
then have ‹*kf-comp-dependent 0 E* =$_{kr}$ *0*›
by (*auto intro*!: *ext simp*: *kf-eq-weak-def kf-comp-dependent-apply split-def*)
then have ‹(*kf-comp-dependent 0 E*) = *0*›
using *kf-eq-0-iff-eq-0* **by** *auto*
then show ‹*kf-comp-dependent* (λ-. *0*) *E* = *0*›
by (*simp add*: *kf-comp-dependent-def zero-fun-def*)
qed

lemma *kf-comp-dependent-0-right*[*simp*]: ‹*kf-comp-dependent E 0* = *0*›
by (*auto intro*!: *ext simp add*: *kf-eq-weak-def kf-comp-dependent-def*)

lemma *kf-comp-0-left*[*simp*]: ‹*kf-comp 0 E* = *0*›
using *kf-comp-dependent-0-left*[*of E*]
by (*simp add*: *kf-comp-def zero-fun-def*)

lemma *kf-comp-0-right*[*simp*]: ‹*kf-comp E 0* = *0*›
using *kf-comp-dependent-0-right*[*of* ‹λ-. *E*›]
by (*simp add*: *kf-comp-def*)

lemma *kf-filter-comp*:
fixes $\mathfrak{F}$ :: ‹('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*f*) *kraus-family*›
and $\mathfrak{E}$ :: ‹('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*e*) *kraus-family*›
shows ‹*kf-filter* (λ(*e*,*f*). *F f* ∧ *E e*) (*kf-comp* $\mathfrak{F}$ $\mathfrak{E}$)
= *kf-comp* (*kf-filter F* $\mathfrak{F}$) (*kf-filter E* $\mathfrak{E}$)›
unfolding *kf-comp-def*
apply (*rule kf-filter-comp-dependent*)

by *auto*

lemma *kf-comp-dependent-invalid*:
assumes ‹¬ *bdd-above* ((*kf-norm o* 𝔈) ‘ *kf-domain* 𝔉)›
shows ‹*kf-comp-dependent* 𝔈 𝔉 = *0*›
by (*metis* (*no-types*, *lifting*) *Rep-kraus-family-inject assms kf-comp-dependent-def kf-comp-dependent-raw.rep-eq kf-map0 zero-kraus-family.rep-eq*)

lemma *kf-comp-dependent-map-left*:
‹*kf-comp-dependent* (λ*x*. *kf-map* (*f x*) (*E x*)) *F*
≡$_{kr}$ *kf-map* (λ(*x*,*y*). (*x*, *f x y*)) (*kf-comp-dependent E F*)›
proof (*cases* ‹*bdd-above* ((*kf-norm* ∘ *E*) ‘ *kf-domain F*)›)
case *True*
show *?thesis*
proof (*rule kf-eqI*)
fix *xy* :: ‹*'c* × *'d*› **and** ϱ
obtain *x y* **where** *xy*: ‹*xy* = (*x*, *y*)›
by *force*
define *F'* **where** ‹*F' x* = *kf-filter* (λ*x'*. *x'* = *x*) *F*› **for** *x*
define *E'f* **where** ‹*E'f y e* = *kf-filter* (λ*x*. *f e x* = *y*) (*E e*)› **for** *e y*
have *bdd2*: ‹*bdd-above* ((λ*x*. *kf-norm* (*E'f y x*)) ‘ *kf-domain* (*F' x*))›
by (*simp add*: *E'f-def F'-def*)
have ‹*kf-comp-dependent* (λ*x*. *kf-map* (*f x*) (*E x*)) *F* ∗$_{kr}$ @{*xy*} ϱ
= *kf-filter* (λ(*x'*,*y'*). *y'*=*y* ∧ *x'*=*x*) (*kf-comp-dependent* (λ*x*. *kf-map* (*f x*) (*E x*)) *F*) ∗$_{kr}$ ϱ›
(**is** ‹*?lhs* = -›)
apply (*simp add*: *kf-apply-on-def xy case-prod-unfold*)
by (*metis fst-conv prod.collapse snd-conv*)
also have ‹... = *kf-apply* (*kf-comp-dependent* (λ*e*. *kf-filter* (λ*y'*. *y'* = *y*)
(*kf-map* (*f e*) (*E e*))) (*F' x*)) ϱ›
using *True* **by** (*simp add*: *kf-filter-comp-dependent F'-def*)
also have ‹... = *kf-apply* (*kf-comp-dependent*
(λ*e*. *kf-map* (*f e*) (*E'f y e*)) (*F' x*)) ϱ›
by (*simp add*: *kf-filter-map E'f-def*)
also have ‹... = *kf-apply* (*kf-comp-dependent* (*E'f y*) (*F' x*)) ϱ›
apply (*rule kf-apply-eqI*)
apply (*rule kf-comp-dependent-cong-weak*)
by (*simp-all add*: *bdd2 kf-eq-weak-def*)
also have ‹... = *kf-apply*
(*kf-filter* (λ(*x'*,*y'*). *f x' y'* = *y* ∧ *x'* = *x*) (*kf-comp-dependent E F*)) ϱ›
apply (*subst kf-filter-comp-dependent*)
using *True* **by** (*simp-all add*: *o-def F'-def E'f-def*[*abs-def*])
also have ‹... = *kf-apply* (*kf-map* (λ(*x*,*y*). (*x*, *f x y*))
(*kf-filter* (λ(*x'*,*y'*). *f x' y'* = *y* ∧ *x'* = *x*) (*kf-comp-dependent E F*))) ϱ›
by *simp*
also have ‹...
= *kf-apply* (*kf-filter* (λ(*x'*,*y'*). *y'*=*y* ∧ *x'*=*x*)
(*kf-map* (λ(*x*,*y*). (*x*, *f x y*)) (*kf-comp-dependent E F*))) ϱ›
by (*simp add*: *kf-filter-map case-prod-unfold*)
also have ‹... = *kf-map* (λ(*x*,*y*). (*x*, *f x y*)) (*kf-comp-dependent E F*) ∗$_{kr}$ @{*xy*} ϱ›

```
    apply (simp add: kf-apply-on-def xy case-prod-unfold)
    by (metis fst-conv prod.collapse snd-conv)
  finally show ‹?lhs = ...›
    by −
 qed
next
 case False
 then show ?thesis
  by (simp add: kf-comp-dependent-invalid)
qed



lemma kf-comp-dependent-map-right:
 ‹kf-comp-dependent E (kf-map f F)
   ≡kr kf-map (λ(x,y). (f x, y)) (kf-comp-dependent (λx. E (f x)) F)›
proof (cases ‹bdd-above ((kf-norm ∘ E) ‘ kf-domain (kf-map f F))›)
 case True
 show ?thesis
 proof (rule kf-eqI)
  fix xy :: ‹'c × 'd› and ϱ
  obtain x y where xy: ‹xy = (x, y)›
   by force
  define F'f where ‹F'f x = kf-filter (λxa. f xa = x) F› for x
  define E' where ‹E' y e = kf-filter (λy'. y' = y) (E e)› for e y
  have bdd2: ‹bdd-above ((kf-norm ∘ E' y) ‘ kf-domain (kf-map f (F'f x)))›
   apply (simp add: E'-def F'f-def)
   by (metis (no-types, lifting) bdd-above.I2 image-iff order.refl)
  have bdd3: ‹bdd-above ((kf-norm ∘ (λx. E (f x))) ‘ kf-domain F)›
   by (metis (no-types, lifting) ext True comp-apply image-comp kf-domain-map)
  have bdd4: ‹bdd-above ((kf-norm ∘ (λ-. E' y x)) ‘ kf-domain (F'f x))›
   by fastforce
  have ‹kf-comp-dependent E (kf-map f F) ∗kr @{xy} ϱ
     = kf-filter (λ(x',y'). y'=y ∧ x'=x) (kf-comp-dependent E (kf-map f F)) ∗kr ϱ›
   (is ‹?lhs = -›)
   apply (simp add: kf-apply-on-def xy case-prod-unfold)
   by (metis fst-conv prod.collapse snd-conv)
  also have ‹... = kf-apply (kf-comp-dependent (E' y) (kf-filter (λx'. x' = x) (kf-map f F)))
ϱ›
   using True by (simp add: kf-filter-comp-dependent F'f-def E'-def[abs-def])
  also have ‹... = kf-apply (kf-comp-dependent
            (E' y) (kf-map f (F'f x))) ϱ›
   by (simp add: kf-filter-map F'f-def)
  also have ‹... = kf-apply (kf-comp (E' y x) (kf-map f (F'f x))) ϱ›
   unfolding kf-comp-def
   apply (rule kf-apply-eqI)
   using bdd2 apply (rule kf-comp-dependent-cong-weak)
   by (auto simp: F'f-def)
  also have ‹... = kf-apply (E' y x) (kf-apply (F'f x) ϱ)›
```

```
    by (simp add: kf-comp-apply)
  also have ‹... = kf-apply (kf-comp (E' y x) (F'f x)) ϱ›
    by (simp add: kf-comp-apply)
  also have ‹... = kf-apply (kf-comp-dependent (λe. kf-filter (λy'. y' = y) (E (f e))) (F'f x))
ϱ›
    unfolding kf-comp-def
    apply (rule kf-apply-eqI)
    using bdd4 apply (rule kf-comp-dependent-cong-weak)
    by (auto intro!: simp: F'f-def E'-def)
  also have ‹... = kf-apply (kf-filter (λ(x',y'). y' = y ∧ f x' = x) (kf-comp-dependent (λx. E
(f x)) F)) ϱ›
    using bdd3 by (simp add: kf-filter-comp-dependent F'f-def[abs-def] E'-def[abs-def])
  also have ‹... = kf-apply (kf-filter (λ(x',y'). y'=y ∧ x'=x)
        (kf-map (λ(x, y). (f x, y)) (kf-comp-dependent (λx. E (f x)) F))) ϱ›
    by (simp add: kf-filter-map case-prod-unfold)
  also have ‹... = kf-map (λ(x, y). (f x, y)) (kf-comp-dependent (λx. E (f x)) F) *kr @{xy}
ϱ›
    apply (simp add: kf-apply-on-def xy case-prod-unfold)
    by (metis fst-conv prod.collapse snd-conv)

  finally show ‹?lhs = ...›
    by -
 qed
next
 case False
 have not-bdd2: ‹¬ bdd-above ((kf-norm ∘ (λx. E (f x))) ‘ kf-domain F)›
   by (metis (no-types, lifting) False comp-apply image-comp image-cong kf-domain-map)
 show ?thesis
   using False not-bdd2
   by (simp add: kf-comp-dependent-invalid)
qed

lemma kf-comp-dependent-raw-map-inj-right:
 ‹kf-comp-dependent-raw E (kf-map-inj f F)
   = kf-map-inj (λ(E,F,x,y). (E, F, f x, y)) (kf-comp-dependent-raw (λx. E (f x)) F)›
proof -
 have ‹(λ((F, y), (E, x)). (E o_CL F, F, E, y, x)) ‘ (SIGMA (F, y):Rep-kraus-family (kf-map-inj
f F). Rep-kraus-family (E y)) =
        (λ((F, y),(E,x)). (E o_CL F, F, E, f y, x)) ‘ (SIGMA (F, y):Rep-kraus-family F.
Rep-kraus-family (E (f y)))›
   by (auto intro!: image-eqI simp: Sigma-image-left kf-map-inj.rep-eq)
 then show ?thesis
   apply (transfer' fixing: f)
   by (simp add: case-prod-unfold filter-image image-image del: Set.filter-eq)
qed

lemma kf-comp-dependent-map-inj-right:
 assumes ‹inj-on f (kf-domain F)›
 shows ‹kf-comp-dependent E (kf-map-inj f F)
```

```
    = kf-map-inj (λ(x,y). (f x, y)) (kf-comp-dependent (λx. E (f x)) F)›
proof −
 have dom: ‹kf-domain (kf-comp-dependent-raw (λx. E (f x)) F) ⊆ UNIV × UNIV × kf-domain
F × UNIV›
 proof −
   have ‹kf-domain (kf-comp-dependent-raw (λx. E (f x)) F) ⊆ UNIV × UNIV × (SIGMA
x:kf-domain F. kf-domain (E (f x)))›
     by (rule kf-domain-comp-dependent-raw-subset)
   also have ‹... ⊆ UNIV × UNIV × kf-domain F × UNIV›
     by auto
   finally show ?thesis
     by −
 qed
 have inj2: ‹inj-on (λ(E, F, x, y). (E, F, f x, y)) (kf-domain (kf-comp-dependent-raw (λx. E
(f x)) F))›
 proof −
   have ‹inj-on (λ(E, F, x, y). (E, F, f x, y)) (UNIV × UNIV × kf-domain F × UNIV)›
     using assms by (auto simp: inj-on-def)
   with dom show ?thesis
     by (rule inj-on-subset[rotated])
 qed
 have inj3: ‹inj-on (λ(x, y). (f x, y)) ((λ(F, E, x). x) ‘ kf-domain (kf-comp-dependent-raw (λx.
E (f x)) F))›
 proof −
   from dom have ‹((λ(F, E, x). x) ‘ kf-domain (kf-comp-dependent-raw (λx. E (f x)) F)) ⊆
kf-domain F × UNIV›
     by auto
   moreover have ‹inj-on (λ(x, y). (f x, y)) (kf-domain F × UNIV)›
     using assms by (auto simp: o-def inj-on-def)
   ultimately show ?thesis
     by (rule inj-on-subset[rotated])
 qed
 have ‹kf-comp-dependent E (kf-map-inj f F) = kf-map (λ(F, E, y). y) (kf-comp-dependent-raw
E (kf-map-inj f F))›
   by (simp add: kf-comp-dependent-def id-def)
 also have ‹... = kf-map (λ(F,E,y). y) (kf-map-inj (λ(E,F,x,y). (E, F, f x, y)) (kf-comp-dependent-raw
(λx. E (f x)) F))›
   by (simp add: kf-comp-dependent-raw-map-inj-right)
 also have ‹... = kf-map (λ(E,F,x,y). (f x, y)) (kf-comp-dependent-raw (λx. E (f x)) F)›
   apply (subst kf-map-kf-map-inj-comp)
    apply (rule inj2)
   using assms by (simp add: o-def case-prod-unfold)
 also have ‹... = kf-map-inj (λ(x, y). (f x, y)) (kf-map (λ(F, E, x). x) (kf-comp-dependent-raw
(λx. E (f x)) F))›
   apply (subst kf-map-inj-kf-map-comp)
    apply (rule inj3)
   by (simp add: o-def case-prod-unfold)
 also have ‹... = kf-map-inj (λ(x,y). (f x, y)) (kf-comp-dependent (λx. E (f x)) F)›
   by (simp add: kf-comp-dependent-def id-def)
```

finally show *?thesis*
by −
qed

lemma *kf-comp-dependent-map-right-weak*:
‹*kf-comp-dependent E* (*kf-map f F*)
$=_{kr}$ *kf-comp-dependent* (λ*x*. *E* (*f x*)) *F*›
by (*smt* (*verit*) *kf-apply-eqI kf-apply-map kf-comp-dependent-cong kf-comp-dependent-invalid kf-comp-dependent-map-right kf-eq-def kf-eq-imp-eq-weak kf-eq-weakI*)

lemma *kf-comp-map-left*:
‹*kf-comp* (*kf-map f E*) *F* $\equiv_{kr}$ *kf-map* (λ(*x*,*y*). (*x*, *f y*)) (*kf-comp E F*)›
by (*simp add*: *kf-comp-def kf-comp-dependent-map-left*)

lemma *kf-comp-map-right*:
‹*kf-comp E* (*kf-map f F*) $\equiv_{kr}$ *kf-map* (λ(*x*,*y*). (*f x*, *y*)) (*kf-comp E F*)›
using *kf-comp-dependent-map-right*[**where** *E*=‹λ-. *E*› **and** *f*=*f* **and** *F*=*F*]
by (*simp add*: *kf-comp-def*)

lemma *kf-comp-map-both*:
‹*kf-comp* (*kf-map e E*) (*kf-map f F*) $\equiv_{kr}$ *kf-map* (λ(*x*,*y*). (*f x*, *e y*)) (*kf-comp E F*)›
apply (*rule kf-comp-map-left*[*THEN kf-eq-trans*])
apply (*rule kf-map-cong*[*THEN kf-eq-trans*, *OF refl*])
apply (*rule kf-comp-map-right*)
apply (*rule kf-map-twice*[*THEN kf-eq-trans*])
by (*simp add*: *o-def case-prod-unfold*)

lemma *kf-apply-commute*:
assumes ‹*kf-operators* $\mathfrak{F}$ ⊆ *commutant* (*kf-operators* $\mathfrak{E}$)›
shows ‹*kf-apply* $\mathfrak{F}$ *o kf-apply* $\mathfrak{E}$ = *kf-apply* $\mathfrak{E}$ *o kf-apply* $\mathfrak{F}$›
proof (*rule eq-from-separatingI*[*OF separating-density-ops*[**where** *B*=*1*], *rotated 3*])
show ‹*0* < (*1* :: *real*)›
by *simp*
show ‹*clinear* ((∗$_{kr}$) $\mathfrak{F}$ ∘ (∗$_{kr}$) $\mathfrak{E}$)›
by (*simp add*: *clinear-compose*)
show ‹*clinear* ((∗$_{kr}$) $\mathfrak{E}$ ∘ (∗$_{kr}$) $\mathfrak{F}$)›
using *clinear-compose* **by** *blast*
fix *t* :: ‹('*a*, '*a*) *trace-class*›
assume ‹*t* ∈ {*t*. *0* ≤ *t* ∧ *norm t* ≤ *1*}›
then have ‹*t* ≥ *0*›
by *simp*
from *assms*
have ‹∀ *E* ∈ *kf-operators* $\mathfrak{E}$. ∀ *F* ∈ *kf-operators* $\mathfrak{F}$. *E* o_{CL} *F* = *F* o_{CL} *E*›
unfolding *commutant-def* **by** *auto*
then show ‹((∗$_{kr}$) $\mathfrak{F}$ ∘ (∗$_{kr}$) $\mathfrak{E}$) *t* = ((∗$_{kr}$) $\mathfrak{E}$ ∘ (∗$_{kr}$) $\mathfrak{F}$) *t*›
proof (*transfer fixing*: *t*, *tactic* ‹*FILTER* (*fn st => Thm.nprems-of st = 1*) *all-tac*›)

fix $\mathfrak{E}$ :: ‹('*a* ⇒$_{CL}$ '*a* × '*c*) *set*› **and** $\mathfrak{F}$ :: ‹('*a* ⇒$_{CL}$ '*a* × '*b*) *set*›

assume ‹$\mathfrak{F} \in$ *Collect kraus-family*› **and** ‹$\mathfrak{E} \in$ *Collect kraus-family*›
then have [*iff*]: ‹*kraus-family* $\mathfrak{F}$› ‹*kraus-family* $\mathfrak{E}$›
by *auto*
assume *comm*: ‹$\forall E{\in}$*fst* ‘ $\mathfrak{E}$. $\forall F{\in}$*fst* ‘ $\mathfrak{F}$. $E\ o_{CL}\ F = F\ o_{CL}\ E$›
have *sum1*: ‹$(\lambda E.$ *sandwich-tc* (*fst* E) $t)$ *summable-on* $\mathfrak{E}$›
apply (*rule abs-summable-summable*)
apply (*rule kf-apply-abs-summable*[*unfolded case-prod-unfold*])
by *simp*
have *sum2*: ‹$(\lambda y.$ *sandwich-tc a* (*sandwich-tc* (*fst* y) $t)$) *summable-on* $\mathfrak{E}$› **for** a
apply (*rule summable-on-bounded-linear*[**where** *h*=‹*sandwich-tc -*›])
by (*simp-all add*: *bounded-clinear.bounded-linear bounded-clinear-sandwich-tc sum1*)
have *sum3*: ‹$(\lambda F.$ *sandwich-tc* (*fst* F) $t)$ *summable-on* $\mathfrak{F}$› **for** t
apply (*rule abs-summable-summable*)
apply (*rule kf-apply-abs-summable*[*unfolded case-prod-unfold*])
by *simp*

have ‹$((\lambda\varrho.\ \sum_\infty F{\in}\mathfrak{F}.$ *sandwich-tc* (*fst* F) $\varrho) \circ (\lambda\varrho.\ \sum_\infty E{\in}\mathfrak{E}.$ *sandwich-tc* (*fst* E) $\varrho))\ t$
$= (\sum_\infty F{\in}\mathfrak{F}.\ \sum_\infty E{\in}\mathfrak{E}.$ *sandwich-tc* (*fst* F) (*sandwich-tc* (*fst* E) t))› (**is** ‹*?lhs* = -›)
apply (*subst infsum-bounded-linear*[**where** *h*=‹*sandwich-tc -*›])
by (*simp-all add*: *bounded-clinear.bounded-linear bounded-clinear-sandwich-tc sum1*)
also have ‹$\ldots = (\sum_\infty E{\in}\mathfrak{E}.\ \sum_\infty F{\in}\mathfrak{F}.$ *sandwich-tc* (*fst* F) (*sandwich-tc* (*fst* E) t))›
apply (*rule infsum-swap-positive-tc*)
using ‹$t \geq 0$› **by** (*simp-all add*: *sum2 sum3 sandwich-tc-pos*)
also have ‹$\ldots = (\sum_\infty E{\in}\mathfrak{E}.\ \sum_\infty F{\in}\mathfrak{F}.$ *sandwich-tc* (*fst* E) (*sandwich-tc* (*fst* F) t))›
apply (*intro infsum-cong*)
apply (*subst sandwich-tc-compose*[*THEN fun-cong*, *unfolded o-def*, *symmetric*])+
using *comm*
by *simp*
also have ‹$\ldots = ((\lambda\varrho.\ \sum_\infty F{\in}\mathfrak{E}.$ *sandwich-tc* (*fst* F) $\varrho) \circ (\lambda\varrho.\ \sum_\infty E{\in}\mathfrak{F}.$ *sandwich-tc* (*fst* E) $\varrho))\ t$›
apply (*subst infsum-bounded-linear*[**where** *h*=‹*sandwich-tc -*›])
by (*simp-all add*: *bounded-clinear.bounded-linear bounded-clinear-sandwich-tc sum3*)
finally show ‹*?lhs* = $\ldots$›
by −
qed
qed

lemma *kf-comp-commute-weak*:
assumes ‹*kf-operators* $\mathfrak{F} \subseteq$ *commutant* (*kf-operators* $\mathfrak{E}$)›
shows ‹*kf-comp* $\mathfrak{F}$ $\mathfrak{E}$ $=_{kr}$ *kf-comp* $\mathfrak{E}$ $\mathfrak{F}$›
apply (*rule kf-eq-weakI*)
apply (*simp add*: *kf-comp-apply*)
using *kf-apply-commute*[*OF assms*, *unfolded o-def*]
by *meson*

lemma *kf-comp-commute*:
assumes ‹*kf-operators* $\mathfrak{F} \subseteq$ *commutant* (*kf-operators* $\mathfrak{E}$)›
shows ‹*kf-comp* $\mathfrak{F}$ $\mathfrak{E}$ $\equiv_{kr}$ *kf-map prod.swap* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$)›
proof (*rule kf-eqI-from-filter-eq-weak*)

fix *xy* :: ‹$'c \times 'b$›
obtain *x y* **where** *xy*: ‹$xy = (x,y)$›
by (*simp add*: *prod-eq-iff*)
have $*$: ‹*kf-operators* (*kf-filter* ((=) *y*) $\mathfrak{F}$) $\subseteq$ *commutant* (*kf-operators* (*kf-filter* ((=) *x*) $\mathfrak{E}$))›
using *kf-operators-filter commutant-antimono*[*OF kf-operators-filter*] *assms*
by *fastforce*
have ‹*kf-filter* ((=) *xy*) (*kf-comp* $\mathfrak{F}$ $\mathfrak{E}$) = *kf-comp* (*kf-filter* ((=) *y*) $\mathfrak{F}$) (*kf-filter* ((=) *x*) $\mathfrak{E}$)›
apply (*simp add*: *xy prod-eq-iff*[*abs-def*] *case-prod-unfold flip*: *kf-filter-comp*)
by *meson*
also have ‹... $=_{kr}$ *kf-comp* (*kf-filter* ((=) *x*) $\mathfrak{E}$) (*kf-filter* ((=) *y*) $\mathfrak{F}$)›
apply (*rule kf-comp-commute-weak*)
by (*simp add*: $*$)
also have ‹... = *kf-filter* ((=) (*prod.swap xy*)) (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$)›
apply (*simp add*: *xy prod-eq-iff*[*abs-def*] *case-prod-unfold flip*: *kf-filter-comp*)
by *meson*
also have ‹... $=_{kr}$ *kf-filter* ((=) *xy*) (*kf-map prod.swap* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$))›
apply (*simp add*: *kf-filter-map*)
by (*metis* (*mono-tags*, *lifting*) *kf-eq-refl kf-eq-weak-kf-map-right kf-filter-cong-weak swap-swap*)
finally show ‹*kf-filter* ((=) *xy*) (*kf-comp* $\mathfrak{F}$ $\mathfrak{E}$) $=_{kr}$ *kf-filter* ((=) *xy*) (*kf-map prod.swap* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$))›
by −
qed

lemma *kf-comp-apply-on-singleton*:
‹*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ @{*x*} ϱ = $\mathfrak{E}$ $*_{kr}$ @{*snd x*} ($\mathfrak{F}$ $*_{kr}$ @{*fst x*} ϱ)›
proof −
have ‹*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ @{*x*} ϱ = *kf-filter* (λ(*x1*,*x2*). *x2* = *snd x* $\wedge$ *x1* = *fst x*) (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$) $*_{kr}$ ϱ›
by (*simp add*: *kf-apply-on-def prod-eq-iff case-prod-unfold conj-commute*)
also have ‹... = *kf-comp* (*kf-filter* (λ*x2*. *x2* = *snd x*) $\mathfrak{E}$) (*kf-filter* (λ*x1*. *x1* = *fst x*) $\mathfrak{F}$) $*_{kr}$ ϱ›
by (*simp add*: *kf-filter-comp*)
also have ‹... = $\mathfrak{E}$ $*_{kr}$ @{*snd x*} ($\mathfrak{F}$ $*_{kr}$ @{*fst x*} ϱ)›
by (*simp add*: *kf-apply-on-def kf-comp-apply*)
finally show *?thesis*
by −
qed

lemma *kf-comp-dependent-apply-on-singleton*:
assumes ‹*bdd-above* ((*kf-norm* $\circ$ $\mathfrak{E}$) ‘ *kf-domain* $\mathfrak{F}$)›
shows ‹*kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ @{*x*} ϱ = $\mathfrak{E}$ (*fst x*) $*_{kr}$ @{*snd x*} ($\mathfrak{F}$ $*_{kr}$ @{*fst x*} ϱ)›
proof −
have ‹*kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ @{*x*} ϱ = *kf-comp-dependent* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ @({*fst x*}×{*snd x*}) ϱ›
by *fastforce*
also have ‹... = *kf-comp-dependent* (λ-. $\mathfrak{E}$ (*fst x*)) $\mathfrak{F}$ $*_{kr}$ @({*fst x*}×{*snd x*}) ϱ›
apply (*rule kf-apply-comp-dependent-cong*[*THEN fun-cong*])
using *assms* **by** *auto*
also have ‹... = *kf-comp* ($\mathfrak{E}$ (*fst x*)) $\mathfrak{F}$ $*_{kr}$ @({*x*}) ϱ›
by (*simp add*: *kf-comp-def*)

also have ‹... = 𝔈 (*fst x*) $*_{kr}$ @{*snd x*} (𝔉 $*_{kr}$ @{*fst x*} *ϱ*)›
by (*simp add*: *kf-comp-apply-on-singleton*)
finally show *?thesis*
by −
qed

lemma *kf-comp-id-left*: ‹*kf-comp kf-id* 𝔈 ≡$_{kr}$ *kf-map* (λ*x*. (*x*,())) 𝔈›
proof (*rule kf-eqI-from-filter-eq-weak*)
fix *xy* :: ‹'*c* × *unit*›
define *x* **where** ‹*x* = *fst xy*›
then have *xy*: ‹*xy* = (*x*,())›
by (*auto intro*!: *prod.expand*)
have ‹*kf-filter* ((=) *xy*) (*kf-comp kf-id* 𝔈) = *kf-comp* (*kf-filter* (λ-. *True*) *kf-id*) (*kf-filter* ((=)*x*) 𝔈)›
by (*auto intro*!: *arg-cong2*[**where** *f*=*kf-filter*] *simp add*: *xy simp flip*: *kf-filter-comp simp del*: *kf-filter-true*)
also have ‹... =$_{kr}$ *kf-filter* ((=)*x*) 𝔈›
by (*simp add*: *kf-comp-apply kf-eq-weakI*)
also have ‹... =$_{kr}$ *kf-map* (λ*x*. (*x*,())) (*kf-filter* ((=)*x*) 𝔈)›
by (*simp add*: *kf-eq-weak-def*)
also have ‹... = *kf-filter* ((=) *xy*) (*kf-map* (λ*x*. (*x*,())) 𝔈)›
by (*simp add*: *kf-filter-map xy*)
finally show ‹*kf-filter* ((=) *xy*) (*kf-comp kf-id* 𝔈) =$_{kr}$ ...›
by −
qed

lemma *kf-comp-id-right*: ‹*kf-comp* 𝔈 *kf-id* ≡$_{kr}$ *kf-map* (λ*x*. ((),*x*)) 𝔈›
proof (*rule kf-eqI-from-filter-eq-weak*)
fix *xy* :: ‹*unit* × '*c*›
define *y* **where** ‹*y* = *snd xy*›
then have *xy*: ‹*xy* = ((),*y*)›
by (*auto intro*!: *prod.expand simp*: *y-def*)
have ‹*kf-filter* ((=) *xy*) (*kf-comp* 𝔈 *kf-id*) = *kf-comp* (*kf-filter* ((=)*y*) 𝔈) (*kf-filter* (λ-. *True*) *kf-id*)›
by (*auto intro*!: *arg-cong2*[**where** *f*=*kf-filter*] *simp add*: *xy simp flip*: *kf-filter-comp simp del*: *kf-filter-true*)
also have ‹... =$_{kr}$ *kf-filter* ((=)*y*) 𝔈›
by (*simp add*: *kf-comp-apply kf-eq-weakI*)
also have ‹... =$_{kr}$ *kf-map* (*Pair* ()) (*kf-filter* ((=)*y*) 𝔈)›
by (*simp add*: *kf-eq-weak-def*)
also have ‹... = *kf-filter* ((=) *xy*) (*kf-map* (λ*x*. ((),*x*)) 𝔈)›
by (*simp add*: *kf-filter-map xy*)
finally show ‹*kf-filter* ((=) *xy*) (*kf-comp* 𝔈 *kf-id*) =$_{kr}$ ...›
by −
qed

lemma *kf-comp-dependent-raw-kf-plus-left*:
fixes 𝔇 :: ‹'*f* ⇒ ('*b*::*chilbert-space*, '*c*::*chilbert-space*, '*d*) *kraus-family*›
fixes 𝔈 :: ‹'*f* ⇒ ('*b*::*chilbert-space*, '*c*::*chilbert-space*, '*e*) *kraus-family*›

fixes $\mathfrak{F}$ :: ‹('a::*chilbert-space*, 'b, 'f) *kraus-family*›
assumes ‹*bdd-above* ((λx. *kf-norm* ($\mathfrak{D}$ x)) ‘ *kf-domain* $\mathfrak{F}$)›
assumes ‹*bdd-above* ((λx. *kf-norm* ($\mathfrak{E}$ x)) ‘ *kf-domain* $\mathfrak{F}$)›
shows ‹*kf-comp-dependent-raw* (λx. *kf-plus* ($\mathfrak{D}$ x) ($\mathfrak{E}$ x)) $\mathfrak{F}$ =
kf-map-injx (λx. *case* x *of Inl* (F,D,f,d) ⇒ (F, D, f, *Inl* d) | *Inr* (F,E,f,e) ⇒ (F, E, f, *Inr* e))
(*kf-plus* (*kf-comp-dependent-raw* $\mathfrak{D}$ $\mathfrak{F}$) (*kf-comp-dependent-raw* $\mathfrak{E}$ $\mathfrak{F}$))›
proof (*rule Rep-kraus-family-inject*[*THEN iffD1*], *rule Set.set-eqI*, *rename-tac tuple*)
fix *tuple* :: ‹('a ⇒$_{CL}$ 'c) × ('a ⇒$_{CL}$ 'b) × ('b ⇒$_{CL}$ 'c) × 'f × ('d + 'e)›
have [*simp*]: ‹*bdd-above* ((λx. *kf-norm* (*kf-plus* ($\mathfrak{D}$ x) ($\mathfrak{E}$ x))) ‘ *kf-domain* $\mathfrak{F}$)›
apply (*rule bdd-above-mono2*[*rotated*])
apply (*rule order-refl*)
apply (*rule kf-norm-triangle*)
using *assms* **by** (*rule bdd-above-plus*)
obtain *EF F E f y* **where** *tuple*: ‹*tuple* = (*EF*,*F*,*E*,*f*,*y*)›
by (*auto simp*: *prod-eq-iff*)
have ‹*tuple* ∈ *Rep-kraus-family* (*kf-comp-dependent-raw* (λx. *kf-plus* ($\mathfrak{D}$ x) ($\mathfrak{E}$ x)) $\mathfrak{F}$)
⟷ *EF* = *E* o$_{CL}$ *F* ∧ *EF* ≠ 0 ∧ (*F*,*f*) ∈ *Rep-kraus-family* $\mathfrak{F}$ ∧ (*E*,*y*) ∈ *Rep-kraus-family* (*kf-plus* ($\mathfrak{D}$ *f*) ($\mathfrak{E}$ *f*))›
by (*auto intro*!: *image-eqI simp add*: *tuple kf-comp-dependent-raw.rep-eq*)
also have ‹... ⟷ *EF* = *E* o$_{CL}$ *F* ∧ *EF* ≠ 0 ∧ (*F*,*f*) ∈ *Rep-kraus-family* $\mathfrak{F}$ ∧
(*case y of Inl d* ⇒ (*E*,*d*) ∈ *Rep-kraus-family* ($\mathfrak{D}$ *f*)
| *Inr e* ⇒ (*E*,*e*) ∈ *Rep-kraus-family* ($\mathfrak{E}$ *f*))›
apply (*cases y*)
by (*auto simp*: *kf-plus.rep-eq*)
also have ‹... ⟷ (*case y of Inl d* ⇒ (*EF*, *F*, *E*, *f*, *d*) ∈ *Rep-kraus-family* (*kf-comp-dependent-raw* $\mathfrak{D}$ $\mathfrak{F}$)
| *Inr e* ⇒ (*EF*, *F*, *E*, *f*, *e*) ∈ *Rep-kraus-family* (*kf-comp-dependent-raw* $\mathfrak{E}$ $\mathfrak{F}$))›
apply (*cases y*)
by (*force intro*!: *simp*: *kf-comp-dependent-raw.rep-eq assms*)+
also have ‹... ⟷ (*case y of Inl d* ⇒ (*EF*, *Inl* (*F*, *E*, *f*, *d*)) ∈ *Rep-kraus-family* (*kf-plus* (*kf-comp-dependent-raw* $\mathfrak{D}$ $\mathfrak{F}$) (*kf-comp-dependent-raw* $\mathfrak{E}$ $\mathfrak{F}$))
| *Inr e* ⇒ (*EF*, *Inr* (*F*, *E*, *f*, *e*)) ∈ *Rep-kraus-family* (*kf-plus* (*kf-comp-dependent-raw* $\mathfrak{D}$ $\mathfrak{F}$) (*kf-comp-dependent-raw* $\mathfrak{E}$ $\mathfrak{F}$)))›
apply (*cases y*)
by (*auto intro*!: *simp*: *kf-plus.rep-eq*)
also have ‹... ⟷
(*tuple* ∈ *Rep-kraus-family* (*kf-map-inj*
(λx. *case x of Inl* (*F*, *D*, *f*, *d*) ⇒ (*F*, *D*, *f*, *Inl d*) | *Inr* (*F*, *E*, *f*, *e*) ⇒ (*F*, *E*, *f*, *Inr* e))
(*kf-plus* (*kf-comp-dependent-raw* $\mathfrak{D}$ $\mathfrak{F}$) (*kf-comp-dependent-raw* $\mathfrak{E}$ $\mathfrak{F}$))))›
apply (*cases y*)
by (*force simp*: *tuple kf-map-inj.rep-eq split*!: *sum.split-asm prod.split*)+
finally
show ‹(*tuple* ∈ *Rep-kraus-family* (*kf-comp-dependent-raw* (λx. *kf-plus* ($\mathfrak{D}$ x) ($\mathfrak{E}$ x)) $\mathfrak{F}$)) ⟷
(*tuple* ∈ *Rep-kraus-family* (*kf-map-inj*
(λx. *case x of Inl* (*F*, *D*, *f*, *d*) ⇒ (*F*, *D*, *f*, *Inl d*) | *Inr* (*F*, *E*, *f*, *e*) ⇒ (*F*, *E*, *f*, *Inr* e))

```
        (kf-plus (kf-comp-dependent-raw 𝔇 𝔉) (kf-comp-dependent-raw 𝔈 𝔉)))›
  by −
qed


lemma kf-comp-dependent-kf-plus-left:
  assumes ‹bdd-above ((λx. kf-norm (𝔇 x)) ‘ kf-domain 𝔉)›
  assumes ‹bdd-above ((λx. kf-norm (𝔈 x)) ‘ kf-domain 𝔉)›
  shows ‹kf-comp-dependent (λx. kf-plus (𝔇 x) (𝔈 x)) 𝔉 =
   kf-map-inj (λx. case x of Inl (f,d) ⇒ (f, Inl d) | Inr (f,e) ⇒ (f, Inr e)) (kf-plus (kf-comp-dependent
𝔇 𝔉) (kf-comp-dependent 𝔈 𝔉))›
 apply (simp add: kf-comp-dependent-def kf-comp-dependent-raw-kf-plus-left[OF assms] kf-plus-map-both)
  apply (subst kf-map-kf-map-inj-comp)
   apply (auto simp add: inj-on-def split!: sum.split-asm)[1]
  apply (subst kf-map-inj-kf-map-comp)
   apply (auto simp add: inj-on-def split!: sum.split-asm)[1]
  apply (rule arg-cong2[where f=kf-map])
  by (auto intro!: ext simp add: o-def split!: sum.split)

lemma kf-map-inj-twice:
  shows ‹kf-map-inj f (kf-map-inj g 𝔈) = kf-map-inj (f o g) 𝔈›
  apply (transfer′ fixing: f g)
  by (simp add: image-image case-prod-unfold)


lemma kf-comp-dependent-kf-plus-left′:
  assumes ‹bdd-above ((λx. kf-norm (𝔇 x)) ‘ kf-domain 𝔉)›
  assumes ‹bdd-above ((λx. kf-norm (𝔈 x)) ‘ kf-domain 𝔉)›
  shows ‹kf-plus (kf-comp-dependent 𝔇 𝔉) (kf-comp-dependent 𝔈 𝔉) =
    kf-map-inj (λ(f,de). case de of Inl d ⇒ Inl (f,d) | Inr e ⇒ Inr (f,e)) (kf-comp-dependent
(λx. kf-plus (𝔇 x) (𝔈 x)) 𝔉)›
  apply (subst kf-comp-dependent-kf-plus-left[OF assms])
  apply (subst kf-map-inj-twice)
  apply (rewrite at ‹kf-map-inj ¤ -› to id DEADID.rel-mono-strong)
  by (auto intro!: ext simp: split!: sum.split)


lemma kf-comp-dependent-plus-left:
  assumes ‹bdd-above ((λx. kf-norm (𝔇 x)) ‘ kf-domain 𝔉)›
  assumes ‹bdd-above ((λx. kf-norm (𝔈 x)) ‘ kf-domain 𝔉)›
 shows ‹kf-comp-dependent (λx. 𝔇 x + 𝔈 x) 𝔉 ≡kr kf-comp-dependent 𝔇 𝔉 + kf-comp-dependent
𝔈 𝔉›
proof −
  have ‹kf-comp-dependent (λx. 𝔇 x + 𝔈 x) 𝔉 ≡kr
     kf-map (λ(x, y). (x, case y of Inl x ⇒ x | Inr y ⇒ y)) (kf-comp-dependent (λx. kf-plus (𝔇
x) (𝔈 x)) 𝔉)›
    unfolding plus-kraus-family-def
    by (rule kf-comp-dependent-map-left)
  also have ‹... = kf-map (λ(x, y). (x, case y of Inl x ⇒ x | Inr y ⇒ y))
```

(*kf-map-inj* (*case-sum* (λ(*f*, *d*). (*f*, *Inl d*)) (λ(*f*, *e*). (*f*, *Inr e*)))
(*kf-plus* (*kf-comp-dependent* 𝔇 𝔉) (*kf-comp-dependent* 𝔈 𝔉)))›
by (*simp add*: *kf-comp-dependent-kf-plus-left*[*OF assms*])
also have ‹... = *kf-map* (λ*y*. *case y of Inl x* ⇒ *x* | *Inr y* ⇒ *y*)
(*kf-plus* (*kf-comp-dependent* 𝔇 𝔉) (*kf-comp-dependent* 𝔈 𝔉))›
apply (*subst kf-map-kf-map-inj-comp*)
apply (*auto intro*!: *inj-onI split*!: *sum.split-asm*)[*1*]
apply (*rule arg-cong2*[**where** *f*=*kf-map*, *OF* - *refl*])
by (*auto intro*!: *ext simp add*: *o-def split*!: *sum.split*)
also have ‹... = *kf-comp-dependent* 𝔇 𝔉 + *kf-comp-dependent* 𝔈 𝔉›
by (*simp add*: *plus-kraus-family-def*)
finally show *?thesis*
by −
qed

2.9 Infinite sums

lift-definition *kf-infsum* :: ‹('*a* ⇒ ('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*x*) *kraus-family*) ⇒ '*a set* ⇒ ('*b*,'*c*,'*a*×'*x*) *kraus-family*› **is**
‹λ𝔈 *A*. *if summable-on-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) *A*
then (λ(*a*,(*E*,*x*)). (*E*,(*a*,*x*))) ‘ *Sigma A* (λ*a*. *Rep-kraus-family* (𝔈 *a*)) *else* {}›
proof (*rule CollectI*, *rename-tac* 𝔈 *A*)
fix 𝔈 :: ‹'*a* ⇒ ('*b*, '*c*, '*x*) *kraus-family*› **and** *A*
define 𝔈' **where** ‹𝔈' *a* = *Rep-kraus-family* (𝔈 *a*)› **for** *a*
show ‹*kraus-family* (*if summable-on-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) *A*
then (λ(*a*,(*E*,*x*)). (*E*,(*a*,*x*))) ‘ *Sigma A* 𝔈'
else {})›
proof (*cases* ‹*summable-on-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) *A*›)
case *True*
have ‹*kraus-family* ((λ(*a*, *E*, *x*). (*E*, *a*, *x*)) ‘ *Sigma A* 𝔈')›
proof (*intro kraus-familyI bdd-aboveI*)
fix *C* **assume** ‹*C* ∈ (λ*F*. ∑(*E*, *x*)∈*F*. *E*∗ *o*$_{CL}$ *E*) ‘ {*F*. *finite F* ∧ *F* ⊆ (λ(*a*, *E*, *x*). (*E*, *a*, *x*)) ‘ *Sigma A* 𝔈'}›
then obtain *F* **where** ‹*finite F*› **and** *F-subset*: ‹*F* ⊆ (λ(*a*, *E*, *x*). (*E*, *a*, *x*)) ‘ *Sigma A* 𝔈'›
and *C-def*: ‹*C* = (∑(*E*, *x*)∈*F*. *E*∗ *o*$_{CL}$ *E*)›
by *blast*
define *B F'* **where** ‹*B* = *infsum-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) *A*›
and ‹*F'* = (λ(*E*,*a*,*x*). (*a*,*E*,*x*)) ‘ *F*›

have [*iff*]: ‹*finite F'*›
using ‹*finite F*› **by** (*auto intro*!: *simp*: *F'-def*)
have *F'-subset*: ‹*F'* ⊆ *Sigma A* 𝔈'›
using *F-subset* **by** (*auto simp*: *F'-def*)
have *Sigma-decomp*: ‹(*SIGMA a*:(λ*x*. *fst x*) ‘ *F'*. *snd* ‘ *Set.filter* (λ(*a'*,*Ex*). *a'*=*a*) *F'*) = *F'*›
by *force*

have ‹*C* = (∑(*a*, (*E*, *x*))∈*F'*. *E*∗ *o*$_{CL}$ *E*)›
unfolding *F'-def*

apply (*subst sum.reindex*)
by (*auto intro*!: *inj-onI simp*: *C-def case-prod-unfold*)
also have ‹... = ($\sum$ *a*∈*fst* ‘ *F'*. $\sum$ (*E*, *x*)∈*snd* ‘ *Set.filter* (λ(*a'*,*Ex*). *a'*=*a*) *F'*. *E*∗ o_{CL} *E*)›
apply (*subst sum.Sigma*)
by (*auto intro*!: *finite-imageI simp*: *Sigma-decomp simp del*: *Set.filter-eq*)
also have ‹... = ($\sum$ *a*∈*fst* ‘ *F'*. *infsum-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (*snd* ‘ *Set.filter* (λ(*a'*,*Ex*). *a'*=*a*) *F'*))›
apply (*rule sum.cong*[*OF refl*])
apply (*rule infsum-in-finite*[*symmetric*])
by *auto*
also have ‹... ≤ ($\sum$ *a*∈*fst* ‘ *F'*. *infsum-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (𝔈' *a*))›
proof (*rule sum-mono*, *rule infsum-mono-neutral-wot*)
fix *a* **assume** ‹*a* ∈ *fst* ‘ *F'*›
show ‹*summable-on-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (*snd* ‘ *Set.filter* (λ(*a'*, *Ex*). *a'* = *a*) *F'*)›
by (*auto intro*!: *summable-on-in-finite*)
show ‹*summable-on-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (𝔈' *a*)›
using 𝔈'-*def*[*abs-def*] *kf-bound-has-sum summable-on-in-def* **by** *blast*
show ‹(*case Ex of* (*E*, *x*) ⇒ *E*∗ o_{CL} *E*) ≤ (*case Ex of* (*E*, *x*) ⇒ *E*∗ o_{CL} *E*)› **for** *Ex*
by *blast*
have ‹*snd* ‘ *Set.filter* (λ(*a'*, *Ex*). *a'* = *a*) *F'* ≤ 𝔈' *a*›
using *F'-subset* **by** *auto*
then show ‹(*case Ex of* (*E*, *x*) ⇒ *E*∗ o_{CL} *E*) ≤ *0*›
if ‹*Ex* ∈ *snd* ‘ *Set.filter* (λ(*a'*, *Ex*). *a'* = *a*) *F'* − 𝔈' *a*› **for** *Ex*
using *that* **by** *blast*
show ‹*0* ≤ (*case Ex of* (*E*, *x*) ⇒ *E*∗ o_{CL} *E*)› **for** *Ex*
by (*auto intro*!: *positive-cblinfun-squareI simp*: *case-prod-unfold*)
qed
also have ‹... = ($\sum$ *a*∈*fst* ‘ *F'*. *kf-bound* (𝔈 *a*))›
unfolding 𝔈'-*def*
apply (*transfer' fixing*: *F'*)
by *simp*
also have ‹... = *infsum-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) (*fst* ‘ *F'*)›
apply (*rule infsum-in-finite*[*symmetric*])
by *auto*
also have ‹... ≤ *infsum-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) *A*›
proof (*rule infsum-mono-neutral-wot*)
show ‹*summable-on-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) (*fst* ‘ *F'*)›
by (*auto intro*!: *summable-on-in-finite*)
show ‹*summable-on-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) *A*›
using *True* **by** *blast*
show ‹*kf-bound* (𝔈 *a*) ≤ *kf-bound* (𝔈 *a*)› **for** *a*
by *blast*
show ‹*kf-bound* (𝔈 *a*) ≤ *0*› **if** ‹*a* ∈ *fst* ‘ *F'* − *A*› **for** *a*
using *F'-subset that* **by** *auto*
show ‹*0* ≤ *kf-bound* (𝔈 *a*)› **for** *a*
by *simp*

```
    qed
    also have ‹... = B›
      using B-def by blast
    finally show ‹C ≤ B›
      by −
  next
    show ‹0 ∉ fst ‘ (λ(a, E, x). (E, a, x)) ‘ Sigma A 𝔈'›
      using Rep-kraus-family by (force simp: 𝔈'-def kraus-family-def)
  qed
  with True show ?thesis
    by simp
 next
  case False
  then show ?thesis
    by (auto intro!: bdd-aboveI simp add: kraus-family-def)
 qed
qed

definition kf-summable :: ‹('a ⇒ ('b::chilbert-space,'c::chilbert-space,'x) kraus-family) ⇒ 'a set
⇒ bool› where
 ‹kf-summable 𝔈 A ⟷ summable-on-in cweak-operator-topology (λa. kf-bound (𝔈 a)) A›

lemma kf-summable-from-abs-summable:
 assumes sum: ‹(λa. kf-norm (𝔈 a)) summable-on A›
 shows ‹kf-summable 𝔈 A›
 using assms
 by (simp add: summable-imp-wot-summable abs-summable-summable kf-summable-def kf-norm-def)

lemma kf-infsum-apply:
 assumes ‹kf-summable 𝔈 A›
 shows ‹kf-infsum 𝔈 A *kr ϱ = (∑∞a∈A. 𝔈 a *kr ϱ)›
proof −
 from kf-apply-summable[of ϱ ‹kf-infsum 𝔈 A›]
 have summable: ‹(λ(a, E, x). sandwich-tc E ϱ) summable-on (SIGMA a:A. Rep-kraus-family
(𝔈 a))›
   using assms
  by (simp add: kf-summable-def kf-infsum.rep-eq case-prod-unfold summable-on-reindex inj-on-def
prod.expand o-def)
 have ‹kf-infsum 𝔈 A *kr ϱ = (∑∞(E,ax)∈(λ(a, E, x) ⇒ (E, a, x)) ‘ (SIGMA a:A. Rep-kraus-family
(𝔈 a)). sandwich-tc E ϱ)›
   using assms unfolding kf-summable-def
   apply (transfer' fixing: 𝔈)
   by (simp add: case-prod-unfold)
 also have ‹... = (∑∞(a, E, x) ∈ (SIGMA a:A. Rep-kraus-family (𝔈 a)). sandwich-tc E ϱ)›
   apply (subst infsum-reindex)
   by (auto intro!: inj-onI simp: o-def case-prod-unfold)
 also have ‹... = (∑∞a∈A. ∑∞(E, b)∈Rep-kraus-family (𝔈 a). sandwich-tc E ϱ)›
   apply (subst infsum-Sigma'-banach[symmetric])
   using summable by auto
```

also have ‹... = ($\sum_\infty a{\in}A$. $\mathfrak{E}$ a $*_{kr}$ ϱ)›
by (*metis* (*no-types*, *lifting*) *infsum-cong kf-apply.rep-eq split-def*)
finally show *?thesis*
by −
qed

lemma *kf-infsum-apply-summable*:
assumes ‹*kf-summable* $\mathfrak{E}$ *A*›
shows ‹(λ*a*. $\mathfrak{E}$ *a* $*_{kr}$ ϱ) *summable-on A*›
proof −
from *kf-apply-summable*[*of* ϱ ‹*kf-infsum* $\mathfrak{E}$ *A*›]
have *summable*: ‹(λ(*a*, *E*, *x*). *sandwich-tc E* ϱ) *summable-on* (*SIGMA a*:*A*. *Rep-kraus-family* ($\mathfrak{E}$ *a*))›
using *assms*
by (*simp add*: *kf-summable-def kf-infsum.rep-eq case-prod-unfold summable-on-reindex inj-on-def prod.expand o-def*)
from *summable-on-Sigma-banach*[*OF this*]
have ‹(λ*a*. $\sum_\infty$(*E*, *x*)∈*Rep-kraus-family* ($\mathfrak{E}$ *a*). *sandwich-tc E* ϱ) *summable-on A*›
by *simp*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *infsum-cong kf-apply.rep-eq split-def summable-on-cong*)
qed

lemma *kf-bound-infsum*:
fixes *f* :: ‹'*a* ⇒ ('*b*::*chilbert-space*,'*c*::*chilbert-space*,'*x*) *kraus-family*›
assumes ‹*kf-summable f A*›
shows ‹*kf-bound* (*kf-infsum f A*) = *infsum-in cweak-operator-topology* (λ*a*. *kf-bound* (*f a*)) *A*›
proof −
have *pos*: ‹0 ≤ *compose-wot* (*adj-wot a*) *a*› **for** *a* :: ‹('*b*,'*c*) *cblinfun-wot*›
apply *transfer'*
using *positive-cblinfun-squareI* **by** *blast*
have *sum3*: ‹(λ*x*. $\sum_\infty$(*E*,-)∈*Rep-kraus-family* (*f x*). *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*)) *summable-on A*›
proof −
define *b* **where** ‹*b x* = *kf-bound* (*f x*)› **for** *x*
have ‹(λ*x*. *Abs-cblinfun-wot* (*b x*)) *summable-on A*›
using *assms* **unfolding** *kf-summable-def*
apply (*subst* (*asm*) *b-def*[*symmetric*])
apply (*transfer' fixing*: *b A*)
by *simp*
then show *?thesis*
by (*simp add*: *Rep-cblinfun-wot-inverse kf-bound-def' b-def*)
qed
have *sum2*: ‹(λ(*E*, -). *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*)) *summable-on*
Rep-kraus-family (*f x*)› **if** ‹*x* ∈ *A*› **for** *x*
by (*rule kf-bound-summable'*)
have *sum1*: ‹(λ(-, *E*, -). *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*)) *summable-on*

(*SIGMA a:A. Rep-kraus-family (f a)*)›
apply (*rule summable-on-Sigma-wotI*)
using *sum3 sum2*
by (*auto intro*!: *pos simp*: *case-prod-unfold*)

have ‹*Abs-cblinfun-wot* (*kf-bound* (*kf-infsum f A*)) =
($\sum_\infty$(*E*, *x*)∈*Rep-kraus-family* (*kf-infsum f A*).
compose-wot (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*))›
by (*simp add*: *kf-bound-def' Rep-cblinfun-wot-inverse*)
also have ‹... = ($\sum_\infty$(*E*, *x*)∈(λ(*a*, *E*, *xa*). (*E*, *a*, *xa*)) ' (*SIGMA a:A. Rep-kraus-family* (*f a*)).
compose-wot (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*))›
using *assms* **by** (*simp add*: *kf-infsum.rep-eq kf-summable-def*)
also have ‹... = ($\sum_\infty$(*a*, *E*, *x*)∈(*SIGMA a:A. Rep-kraus-family* (*f a*)). *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*))›
apply (*subst infsum-reindex*)
by (*auto intro*!: *inj-onI simp*: *o-def case-prod-unfold*)
also have ‹... = ($\sum_\infty$*a*∈*A*. $\sum_\infty$(*E*, *x*)∈*Rep-kraus-family* (*f a*). *compose-wot* (*adj-wot* (*Abs-cblinfun-wot E*)) (*Abs-cblinfun-wot E*))›
apply (*subst infsum-Sigma-topological-monoid*)
using *sum1 sum2* **by** *auto*
also have ‹... = ($\sum_\infty$*a*∈*A*. *Abs-cblinfun-wot* (*kf-bound* (*f a*)))›
by (*simp add*: *kf-bound-def' Rep-cblinfun-wot-inverse*)
also have ‹... = *Abs-cblinfun-wot* (*infsum-in cweak-operator-topology* (λ*a*. *kf-bound* (*f a*)) *A*)›
apply (*simp only*: *infsum-euclidean-eq*[*symmetric*])
apply (*transfer' fixing*: *f A*)
by *simp*
finally show *?thesis*
apply (*rule Abs-cblinfun-wot-inject*[*THEN iffD1*, *rotated* −*1*])
by *simp-all*
qed

lemma *kf-norm-infsum*:
assumes *sum*: ‹(λ*a*. *kf-norm* (𝔈 *a*)) *summable-on A*›
shows ‹*kf-norm* (*kf-infsum* 𝔈 *A*) ≤ ($\sum_\infty$*a*∈*A*. *kf-norm* (𝔈 *a*))›
proof −
have ‹*kf-norm* (*kf-infsum* 𝔈 *A*) = *norm* (*infsum-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) *A*)›
unfolding *kf-norm-def*
apply (*subst kf-bound-infsum*)
by (*simp-all add*: *kf-summable-from-abs-summable assms*)
also have ‹... ≤ ($\sum_\infty$*a*∈*A*. *norm* (*kf-bound* (𝔈 *a*)))›
apply (*rule triangle-ineq-wot*)
using *sum* **by** (*simp add*: *kf-norm-def*)
also have ‹... ≤ ($\sum_\infty$*a*∈*A*. *kf-norm* (𝔈 *a*))›
by (*smt* (*verit*, *del-insts*) *infsum-cong kf-norm-def*)
finally show *?thesis*
by −
qed

lemma *kf-filter-infsum*:
assumes ‹*kf-summable* 𝔈 *A*›
shows ‹*kf-filter P* (*kf-infsum* 𝔈 *A*)
= *kf-infsum* (λ*a*. *kf-filter* (λ*x*. *P* (*a*,*x*)) (𝔈 *a*)) {*a*∈*A*. ∃ *x*. *P* (*a*,*x*)}›
(**is** ‹*?lhs* = *?rhs*›)
proof −
have *summ*: ‹*summable-on-in cweak-operator-topology* (λ*a*. *kf-bound* (*kf-filter* (λ*x*. *P* (*a*, *x*)) (𝔈 *a*)))
{*a* ∈ *A*. ∃ *x*. *P* (*a*, *x*)}›
proof (*rule summable-wot-boundedI*)
fix *F* **assume** ‹*finite F*› **and** *F-subset*: ‹*F* ⊆ {*a* ∈ *A*. ∃ *x*. *P* (*a*, *x*)}›
have ‹(∑ *a*∈*F*. *kf-bound* (*kf-filter* (λ*x*. *P* (*a*, *x*)) (𝔈 *a*)))
≤ (∑ *a*∈*F*. *kf-bound* (𝔈 *a*))›
by (*meson kf-bound-filter sum-mono*)
also have ‹... = *infsum-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) *F*›
apply (*rule infsum-in-finite*[*symmetric*])
by (*auto intro*!: ‹*finite F*›)
also have ‹... ≤ *infsum-in cweak-operator-topology* (λ*a*. *kf-bound* (𝔈 *a*)) *A*›
apply (*rule infsum-mono-neutral-wot*)
using *F-subset assms*
by (*auto intro*!: ‹*finite F*› *intro*: *summable-on-in-finite simp*: *kf-summable-def*)
finally show ‹(∑ *a*∈*F*. *kf-bound* (*kf-filter* (λ*x*. *P* (*a*, *x*)) (𝔈 *a*))) ≤ ...›
by −
next
show ‹*0* ≤ *kf-bound* (*kf-filter* (λ*y*. *P* (*x*, *y*)) (𝔈 *x*))› **for** *x*
by *simp*
qed
have ‹*Rep-kraus-family ?lhs* = *Rep-kraus-family ?rhs*›
using *assms* **by** (*force simp add*: *kf-filter.rep-eq kf-infsum.rep-eq assms summ kf-summable-def*)
then show ‹*?lhs* = *?rhs*›
by (*simp add*: *Rep-kraus-family-inject*)
qed

lemma *kf-infsum-empty*[*simp*]: ‹*kf-infsum* 𝔈 {} = *0*›
apply *transfer′* **by** *simp*

lemma *kf-infsum-singleton*[*simp*]: ‹*kf-infsum* 𝔈 {*a*} = *kf-map-inj* (λ*x*. (*a*,*x*)) (𝔈 *a*)›
apply (*rule Rep-kraus-family-inject*[*THEN iffD1*])
by (*force simp add*: *kf-infsum.rep-eq summable-on-in-finite kf-map-inj.rep-eq*)

lemma *kf-infsum-invalid*:
assumes ‹¬ *kf-summable* 𝔈 *A*›
shows ‹*kf-infsum* 𝔈 *A* = *0*›
using *assms*
apply *transfer′*
unfolding *kf-summable-def*
by *simp*

lemma *kf-infsum-cong*:
fixes $\mathfrak{E}$ $\mathfrak{F}$ :: ‹'*a* ⇒ ('*b*::*chilbert-space*, '*c*::*chilbert-space*, '*x*) *kraus-family*›
assumes ‹⋀*a*. *a* ∈ *A* ⟹ $\mathfrak{E}$ *a* ≡$_{kr}$ $\mathfrak{F}$ *a*›
shows ‹*kf-infsum* $\mathfrak{E}$ *A* ≡$_{kr}$ *kf-infsum* $\mathfrak{F}$ *A*›
proof (*cases* ‹*kf-summable* $\mathfrak{E}$ *A*›)
case *True*
then have *True'*: ‹*kf-summable* $\mathfrak{F}$ *A*›
unfolding *kf-summable-def*
apply (*rule summable-on-in-cong*[*THEN iffD1*, *rotated*])
by (*simp add*: *kf-bound-cong assms kf-eq-imp-eq-weak*)
show *?thesis*
proof (*rule kf-eqI*)
fix *ax* :: ‹'*a* × '*x*› **and** *ϱ*
obtain *a x* **where** *ax-def*: ‹*ax* = (*a*,*x*)›
by *fastforce*
have ∗: ‹{*a'*. *a'* = *a* ∧ *a'* ∈ *A*} = (*if a*∈*A then* {*a*} *else* {})›
by *auto*
have ‹*kf-infsum* $\mathfrak{E}$ *A* ∗$_{kr}$ @{*ax*} *ϱ* = (*if a*∈*A then* $\mathfrak{E}$ *a* ∗$_{kr}$ @{*x*} *ϱ else 0*)›
by (*simp add*: *ax-def kf-apply-on-def True kf-filter-infsum* ∗
kf-apply-map-inj inj-on-def)
also from *assms* **have** ‹... = (*if a*∈*A then* $\mathfrak{F}$ *a* ∗$_{kr}$ @{*x*} *ϱ else 0*)›
by (*auto intro*!: *kf-apply-on-eqI*)
also have ‹... = *kf-infsum* $\mathfrak{F}$ *A* ∗$_{kr}$ @{*ax*} *ϱ*›
by (*simp add*: *ax-def kf-apply-on-def True' kf-filter-infsum* ∗
kf-apply-map-inj inj-on-def)
finally show ‹*kf-infsum* $\mathfrak{E}$ *A* ∗$_{kr}$ @{*ax*} *ϱ* = *kf-infsum* $\mathfrak{F}$ *A* ∗$_{kr}$ @{*ax*} *ϱ*›
by −
qed
next
case *False*
then have *False'*: ‹¬ *kf-summable* $\mathfrak{F}$ *A*›
unfolding *kf-summable-def*
apply (*subst* (*asm*) *assms*[*THEN kf-eq-imp-eq-weak*,
THEN kf-bound-cong, *THEN summable-on-in-cong*])
by *auto*
show *?thesis*
by (*simp add*: *kf-infsum-invalid False False'*)
qed

2.10 Trace-preserving maps

definition ‹*kf-trace-preserving* $\mathfrak{E}$ ⟷ (∀ *ϱ*. *trace-tc* ($\mathfrak{E}$ ∗$_{kr}$ *ϱ*) = *trace-tc ϱ*)›

definition ‹*kf-trace-reducing* $\mathfrak{E}$ ⟷ (∀ *ϱ*≥*0*. *trace-tc* ($\mathfrak{E}$ ∗$_{kr}$ *ϱ*) ≤ *trace-tc ϱ*)›

lemma *kf-trace-reducing-iff-norm-leq1*: ‹*kf-trace-reducing* $\mathfrak{E}$ ⟷ *kf-norm* $\mathfrak{E}$ ≤ *1*›
proof (*unfold kf-trace-reducing-def*, *intro iffI allI impI*)
assume *assm*: ‹*kf-norm* $\mathfrak{E}$ ≤ *1*›

```
  fix ϱ :: ‹('a, 'a) trace-class›
  assume ‹ϱ ≥ 0›
  have ‹trace-tc (𝔈 *kr ϱ) = norm (𝔈 *kr ϱ)›
    by (simp add: ‹0 ≤ ϱ› kf-apply-pos norm-tc-pos)
  also have ‹... ≤ kf-norm 𝔈 * norm ϱ›
    using ‹0 ≤ ϱ› complex-of-real-mono kf-apply-bounded-pos by blast
  also have ‹... ≤ norm ϱ›
    by (metis assm complex-of-real-mono kf-norm-geq0 mult-left-le-one-le norm-ge-zero)
  also have ‹... = trace-tc ϱ›
    by (simp add: ‹0 ≤ ϱ› norm-tc-pos)
  finally show ‹trace-tc (𝔈 *kr ϱ) ≤ trace-tc ϱ›
    by -
next
  assume assm[rule-format]: ‹∀ϱ≥0. trace-tc (𝔈 *kr ϱ) ≤ trace-tc ϱ›
  have ‹kf-bound 𝔈 ≤ id-cblinfun›
  proof (rule cblinfun-leI)
    fix x
    have ‹x ·C kf-bound 𝔈 x = trace-tc (𝔈 *kr tc-butterfly x x)›
      by (simp add: kf-bound-from-map)
    also have ‹... ≤ trace-tc (tc-butterfly x x)›
      apply (rule assm)
      by simp
    also have ‹... = x ·C id-cblinfun x›
      by (simp add: tc-butterfly.rep-eq trace-butterfly trace-tc.rep-eq)
  finally show ‹x ·C kf-bound 𝔈 x ≤ x ·C id-cblinfun x›
      by -
  qed
  then show ‹kf-norm 𝔈 ≤ 1›
    by (smt (verit, best) kf-norm-def kf-bound-pos norm-cblinfun-id-le norm-cblinfun-mono)
qed

lemma kf-trace-preserving-iff-bound-id: ‹kf-trace-preserving 𝔈 ⟷ kf-bound 𝔈 = id-cblinfun›
proof (unfold kf-trace-preserving-def, intro iffI allI)
  assume asm[rule-format]: ‹∀ϱ. trace-tc (𝔈 *kr ϱ) = trace-tc ϱ›
  have ‹ψ ·C kf-bound 𝔈 ψ = ψ ·C id-cblinfun ψ› for ψ
  proof -
    have ‹ψ ·C kf-bound 𝔈 ψ = trace-tc (𝔈 *kr tc-butterfly ψ ψ)›
      by (simp add: kf-bound-from-map)
    also have ‹... = trace-tc (tc-butterfly ψ ψ)›
      by (simp add: asm)
    also have ‹... = ψ ·C id-cblinfun ψ›
      by (simp add: tc-butterfly.rep-eq trace-butterfly trace-tc.rep-eq)
    finally show ?thesis
      by -
  qed
  then show ‹kf-bound 𝔈 = id-cblinfun›
    using cblinfun-cinner-eq0I[where a=‹kf-bound 𝔈 - id-cblinfun›]
    by (simp add: cblinfun.real.diff-left cinner-diff-right)
next
```

assume *asm*: ‹*kf-bound* $\mathfrak{E}$ = *id-cblinfun*›
fix ϱ
have ‹*trace-tc* ($\mathfrak{E}$ $*_{kr}$ ϱ) = *trace-tc* (*compose-tcr* (*kf-bound* $\mathfrak{E}$) ϱ)›
by (*rule trace-from-kf-bound*)
also have ‹... = *trace-tc* ϱ›
using *asm* **by** *fastforce*
finally show ‹*trace-tc* ($\mathfrak{E}$ $*_{kr}$ ϱ) = *trace-tc* ϱ›
by −
qed

lemma *kf-trace-norm-preserving*: ‹*kf-norm* $\mathfrak{E}$ ≤ *1*› **if** ‹*kf-trace-preserving* $\mathfrak{E}$›
apply (*rule kf-trace-reducing-iff-norm-leq1*[*THEN iffD1*])
using *that*
by (*simp add*: *kf-trace-preserving-def kf-trace-reducing-def*)

lemma *kf-trace-norm-preserving-eq*:
fixes $\mathfrak{E}$ :: ‹('*a*::{*chilbert-space*,*not-singleton*}, '*b*::*chilbert-space*, '*c*) *kraus-family*›
assumes ‹*kf-trace-preserving* $\mathfrak{E}$›
shows ‹*kf-norm* $\mathfrak{E}$ = *1*›
using *assms* **by** (*simp add*: *kf-trace-preserving-iff-bound-id kf-norm-def*)

lemma *kf-trace-preserving-map*[*simp*]: ‹*kf-trace-preserving* (*kf-map f* $\mathfrak{E}$) ⟷ *kf-trace-preserving* $\mathfrak{E}$›
by (*simp add*: *kf-map-bound kf-trace-preserving-iff-bound-id*)

lemma *kf-trace-reducing-map*[*simp*]: ‹*kf-trace-reducing* (*kf-map f* $\mathfrak{E}$) ⟷ *kf-trace-reducing* $\mathfrak{E}$›
by (*simp add*: *kf-trace-reducing-iff-norm-leq1*)

lemma *kf-trace-preserving-id*[*iff*]: ‹*kf-trace-preserving kf-id*›
by (*simp add*: *kf-trace-preserving-iff-bound-id*)

lemma *kf-trace-reducing-id*[*iff*]: ‹*kf-trace-reducing kf-id*›
by (*simp add*: *kf-norm-id-leq1 kf-trace-reducing-iff-norm-leq1*)

2.11 Sampling

lift-definition *kf-sample* :: ‹('*x* ⇒ *real*) ⇒ ('*a*::*chilbert-space*, '*a*, '*x*) *kraus-family*› **is**
‹λ*p*. *if* (∀ *x*. *p x* ≥ *0*) ∧ *p summable-on UNIV then Set.filter* (λ(*E*,-). *E*≠*0*) (*range* (λ*x*. (*sqrt* (*p x*) $*_R$ *id-cblinfun*, *x*))) *else* {}›
proof −
fix *p* :: ‹'*x* ⇒ *real*›
show ‹*?thesis p*›
proof (*cases* ‹(∀ *x*. *p x* ≥ *0*) ∧ *p summable-on UNIV*›)
case *True*
then have ‹*p abs-summable-on UNIV*›
by *simp*
from *abs-summable-iff-bdd-above*[*THEN iffD1*, *OF this*]
obtain *B* **where** *F-B*: ‹*finite F* ⟹ (∑ *x*∈*F*. *p x*) ≤ *B*› **for** *F*
apply *atomize-elim*

```
    using True by (auto simp: bdd-above-def)
  have ‹(∑ (E,x)∈F. E∗ o_CL E) ≤ B ∗_R id-cblinfun›
   if ‹finite F› and ‹F ⊆ Set.filter (λ(E,-). E≠0) (range (λx. (sqrt (p x) ∗_R id-cblinfun, x)))›
    for F :: ‹('a⇒_CL'a × 'x) set›
  proof −
    from that
    obtain F' where ‹finite F'› and F-def: ‹F = (λx. (sqrt (p x) ∗_R id-cblinfun, x)) ‘ F'›
      using finite-subset-filter-image
      by meson
    then have ‹(∑ (E,x)∈F. E∗ o_CL E) = (∑ x∈F'. (sqrt (p x) ∗_R id-cblinfun)∗ o_CL (sqrt
(p x) ∗_R id-cblinfun))›
      by (simp add: sum.reindex inj-on-def)
    also have ‹... = (∑ x∈F'. p x ∗_R id-cblinfun)›
      using True by simp
    also have ‹... = (∑ x∈F'. p x) ∗_R id-cblinfun›
      by (metis scaleR-left.sum)
    also have ‹... ≤ B ∗_R id-cblinfun›
    using ‹⋀F. finite F ⟹ (∑ x∈F. p x) ≤ B› ‹finite F'› positive-id-cblinfun scaleR-right-mono
by blast
    finally show ?thesis
      by −
  qed
  then have ‹kraus-family (Set.filter (λ(E,-). E≠0) (range (λx. (sqrt (p x) ∗_R (id-cblinfun
::'a⇒_CL-), x))))›
    apply (simp add: kraus-family-def case-prod-unfold del: Set.filter-eq)
    apply (safe intro!: bdd-aboveI[where M=‹B ∗_R id-cblinfun›])
    by simp-all
  then show ?thesis
    using True by simp
 next
  case False
  then show ?thesis
    by auto
 qed
qed

lemma kf-sample-norm:
 fixes p :: ‹'x ⇒ real›
 assumes ‹⋀x. p x ≥ 0›
 assumes ‹p summable-on UNIV›
 shows ‹kf-norm (kf-sample p :: ('a::{chilbert-space,not-singleton},'a,'x) kraus-family)
          = (∑ ∞x. p x)›
proof −
 define B :: ‹'a⇒_CL'a› where ‹B = kf-bound (kf-sample p)›
 obtain ψ :: 'a where ‹norm ψ = 1›
  using ex-norm1-not-singleton by blast

 have ‹ψ •_C (B ∗_V ψ) = ψ •_C ((∑ ∞x. p x) ∗_R id-cblinfun ∗_V ψ)›
  if ‹norm ψ = 1› for ψ
```

proof −
have ‹*has-sum-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (*Rep-kraus-family* (*kf-sample p*)) *B*›
using *B-def kf-bound-has-sum* **by** *blast*
then have ‹*has-sum-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (*Set.filter* (λ(*E*,-). *E*≠*0*) (*range* (λ*x*. (*sqrt* (*p x*) $*_R$ *id-cblinfun*, *x*)))) *B*›
by (*simp add*: *kf-sample.rep-eq assms*)
then have ‹*has-sum-in cweak-operator-topology* (λ(*E*, *x*). *E*∗ o_{CL} *E*) (*range* (λ*x*. (*sqrt* (*p x*) $*_R$ *id-cblinfun*, *x*))) *B*›
apply (*rule has-sum-in-cong-neutral*[*THEN iffD1*, *rotated* −*1*])
by *auto*
then have ‹*has-sum-in cweak-operator-topology* (λ*x*. *norm* (*p x*) $*_R$ *id-cblinfun*) *UNIV B*›
by (*simp add*: *has-sum-in-reindex inj-on-def o-def*)
then have ‹*has-sum-in cweak-operator-topology* (λ*x*. *p x* $*_R$ *id-cblinfun*) *UNIV B*›
apply (*rule has-sum-in-cong*[*THEN iffD1*, *rotated*])
by (*simp add*: *assms*(*1*))
then have ‹*has-sum-in euclidean* (λ*x*. *ψ* $\cdot_C$ (*p x* $*_R$ *id-cblinfun*) *ψ*) *UNIV* (*ψ* $\cdot_C$ *B ψ*)›
apply (*rule has-sum-in-comm-additive*[*rotated 3*, *OF cweak-operator-topology-cinner-continuous*, *unfolded o-def*])
by (*simp-all add*: *Modules.additive-def cblinfun.add-left cinner-simps*)
then have ‹((λ*x*. *of-real* (*p x*)) *has-sum* (*ψ* $\cdot_C$ *B ψ*)) *UNIV*›
apply (*simp add*: *scaleR-scaleC has-sum-euclidean-iff*)
using ‹*norm ψ* = *1*› *cnorm-eq-1* **by** *force*
then have ‹*ψ* $\cdot_C$ *B ψ* = (∑$_\infty$*x*. *of-real* (*p x*))›
by (*simp add*: *infsumI*)
also have ‹... = *of-real* (∑$_\infty$*x*. *p x*)›
by (*metis infsum-of-real*)
also have ‹... = *ψ* $\cdot_C$ ((∑$_\infty$*x*. *p x*) $*_R$ *id-cblinfun*) *ψ*›
using ‹*norm ψ* = *1*› **by** (*simp add*: *scaleR-scaleC cnorm-eq-1*)
finally show *?thesis*
by −
qed
then have ‹*B* = (∑$_\infty$*x*. *p x*) $*_R$ *id-cblinfun*›
by (*rule cblinfun-cinner-eqI*)
then have ‹*norm B* = *norm* (∑$_\infty$*x*. *p x*)›
by *simp*
also have ‹... = (∑$_\infty$*x*. *p x*)›
by (*simp add*: *abs-of-nonneg assms infsum-nonneg*)
finally show *?thesis*
by (*simp add*: *kf-norm-def B-def*)
qed

2.12 Trace

lift-definition *kf-trace* :: ‹'*a set* ⇒ ('*a*::*chilbert-space*, '*b*::*one-dim*, '*a*) *kraus-family*› **is**
‹λ*B*. *if is-onb B then* (λ*x*. (*vector-to-cblinfun x*∗, *x*)) ‘ *B else* {}›
proof (*rename-tac B*)
fix *B* :: ‹'*a set*›
define *family* :: ‹('*a* ⇒$_{CL}$ '*b* × '*a*) *set*› **where** ‹*family* = (λ*x*. (*vector-to-cblinfun x*∗, *x*)) ‘ *B*›

```
  have ‹kraus-family family› if ‹is-onb B›
  proof −
    have ‹(∑ (E, x)∈F. E* o_CL E) ≤ id-cblinfun› if ‹finite F› and FB: ‹F ⊆ family› for F ::
‹('a ⇒_CL 'b × 'a) set›
    proof −
      obtain G where ‹finite G› and ‹G ⊆ B› and FG: ‹F = (λx. (vector-to-cblinfun x*, x))
‘ G›
        apply atomize-elim
        using ‹finite F› and FB
        apply (simp add: family-def)
        by (meson finite-subset-image)
      from ‹G ⊆ B› have [simp]: ‹is-ortho-set G›
        by (meson ‹is-onb B› is-onb-def is-ortho-set-antimono)
      from ‹G ⊆ B› have [simp]: ‹e ∈ G ⟹ norm e = 1› for e
        by (meson Set.basic-monos(7) ‹is-onb B› is-onb-def)
      have [simp]: ‹inj-on (λx. (vector-to-cblinfun x*, x)) G›
        by (meson inj-onI prod.inject)
      have ‹(∑ (E, x)∈F. E* o_CL E) = (∑ x∈G. selfbutter x)›
        by (simp add: FG sum.reindex flip: butterfly-def-one-dim)
      also have ‹(∑ x∈G. selfbutter x) ≤ id-cblinfun›
        apply (rule sum-butterfly-leq-id)
        by auto
      finally show ?thesis
        by −
    qed
    moreover have ‹0 ∉ fst ‘ family›
      apply (simp add: family-def image-image)
      using ‹is-onb B› apply (simp add: is-onb-def)
      by (smt (verit) adj-0 double-adj imageE norm-vector-to-cblinfun norm-zero)
    ultimately show ?thesis
      by (auto intro!: bdd-aboveI[where M=id-cblinfun] kraus-familyI)
  qed
  then
  show ‹(if is-onb B then family else {}) ∈ Collect kraus-family›
    by auto
qed

lemma kf-trace-is-trace:
  assumes ‹is-onb B›
  shows ‹kf-trace B *_kr ϱ = one-dim-iso (trace-tc ϱ)›
proof −
  define ϱ' where ‹ϱ' = from-trace-class ϱ›
  have ‹kf-apply (kf-trace B) ϱ = (∑_∞x∈B. sandwich-tc (vector-to-cblinfun x*) ϱ)›
    apply (simp add: kf-apply.rep-eq kf-trace.rep-eq assms)
    apply (subst infsum-reindex)
     apply (meson inj-onI prod.simps(1))
    by (simp add: o-def)
  also have ‹... = (∑_∞x∈B. one-dim-iso (x •_C (ϱ' x)))›
    apply (intro infsum-cong from-trace-class-inject[THEN iffD1])
```

apply (*subst from-trace-class-sandwich-tc*)
by (*simp add*: *sandwich-apply flip*: *ϱ′-def*)
also have ‹... = *one-dim-iso* (∑∞x∈B. (*x* •*C* (*ϱ′ x*)))›
by (*metis* (*mono-tags*, *lifting*) *ϱ′-def infsum-cblinfun-apply infsum-cong assms one-cblinfun.rep-eq trace-class-from-trace-class trace-exists*)
also have ‹... = *one-dim-iso* (*trace ϱ′*)›
by (*metis ϱ′-def trace-class-from-trace-class trace-alt-def*[*OF assms*])
also have ‹... = *one-dim-iso* (*trace-tc ϱ*)›
by (*simp add*: *ϱ′-def trace-tc.rep-eq*)
finally show *?thesis*
by −
qed

lemma *kf-eq-weak-kf-trace*:
assumes ‹*is-onb A*› **and** ‹*is-onb B*›
shows ‹*kf-trace A* =*kr kf-trace B*›
by (*auto simp*: *kf-eq-weak-def kf-trace-is-trace assms*)

lemma *trace-is-kf-trace*:
assumes ‹*is-onb B*›
shows ‹*trace-tc t* = *one-dim-iso* (*kf-trace B* ∗*kr t*)›
by (*simp add*: *kf-trace-is-trace assms*)

lemma *kf-trace-bound*[*simp*]:
assumes ‹*is-onb B*›
shows ‹*kf-bound* (*kf-trace B*) = *id-cblinfun*›
using *assms*
by (*auto intro*!: *cblinfun-cinner-eqI simp*: *kf-bound-from-map kf-trace-is-trace trace-tc-butterfly*)

lemma *kf-trace-norm-eq1*[*simp*]:
fixes *B* :: ‹*′a*::{*chilbert-space*, *not-singleton*} *set*›
assumes ‹*is-onb B*›
shows ‹*kf-norm* (*kf-trace B*) = *1*›
using *assms* **by** (*simp add*: *kf-trace-bound kf-norm-def*)

lemma *kf-trace-norm-leq1*[*simp*]:
fixes *B* :: ‹*′a*::*chilbert-space set*›
assumes ‹*is-onb B*›
shows ‹*kf-norm* (*kf-trace B*) ≤ *1*›
by (*simp add*: *assms kf-norm-def norm-cblinfun-id-le*)

2.13 Constant maps

lift-definition *kf-constant-onedim* :: ‹(*′b*,*′b*) *trace-class* ⇒ (*′a*::*one-dim*, *′b*::*chilbert-space*, *unit*) *kraus-family*› **is**
‹λ*t*::(*′b*,*′b*) *trace-class*. *if t* ≥ *0 then*
Set.filter (λ(*E*,-). *E*≠*0*) ((λ*v*. (*vector-to-cblinfun v*,())) ‘ *spectral-dec-vecs-tc t*)
else {}›
proof (*rule CollectI*, *rename-tac t*)

fix t :: ‹('b,'b) *trace-class*›
show ‹*kraus-family* (*if* $t \geq 0$ *then Set.filter* (λ(E,-). E≠0) ((λv. (*vector-to-cblinfun* v :: 'a⇒$_{CL}$'b,())) ‘ *spectral-dec-vecs-tc* t) *else* {})›
proof (*cases* ‹$t \geq 0$›)
case *True*
have ‹*kraus-family* (*Set.filter* (λ(E,-). E≠0) ((λv. (*vector-to-cblinfun* v :: 'a⇒$_{CL}$'b,())) ‘ *spectral-dec-vecs-tc* t))›
proof (*intro kraus-familyI bdd-aboveI*, *rename-tac* E)
fix E :: ‹'a ⇒$_{CL}$ 'a›
assume ‹E ∈ (λF. ∑ (E, x)∈F. E∗ o_{CL} E) ‘ {F. *finite* F ∧ F ⊆ *Set.filter* (λ(E,-). E≠0) ((λv. (*vector-to-cblinfun* v, ())) ‘ *spectral-dec-vecs-tc* t)}›
then obtain F **where** *E-def*: ‹E = (∑ (E, x)∈F. E∗ o_{CL} E)› **and** ‹*finite* F› **and** ‹F ⊆ *Set.filter* (λ(E,-). E≠0) ((λv. (*vector-to-cblinfun* v, ())) ‘ *spectral-dec-vecs-tc* t)›
by *blast*
then obtain F' **where** *F-def*: ‹F = (λv. (*vector-to-cblinfun* v, ())) ‘ F'› **and** ‹*finite* F'› **and** *F'-subset*: ‹F' ⊆ *spectral-dec-vecs-tc* t›
by (*meson finite-subset-filter-image*)
have *inj*: ‹*inj-on* (λv. (*vector-to-cblinfun* v :: 'a⇒$_{CL}$'b, ())) F'›
proof (*rule inj-onI*, *rule ccontr*)
fix x y
assume ‹x ∈ F'› **and** ‹y ∈ F'›
assume *eq*: ‹(*vector-to-cblinfun* x :: 'a⇒$_{CL}$'b, ()) = (*vector-to-cblinfun* y, ())›
assume ‹x ≠ y›
have *ortho*: ‹*is-ortho-set* (*spectral-dec-vecs* (*from-trace-class* t))›
using *True*
by (*auto intro*!: *spectral-dec-vecs-ortho trace-class-compact pos-selfadjoint simp*: *selfadjoint-tc.rep-eq from-trace-class-pos*)
with ‹x ≠ y› *F'-subset* ‹x ∈ F'› ‹y ∈ F'›
have ‹x •$_C$ y = 0›
by (*auto simp*: *spectral-dec-vecs-tc.rep-eq is-ortho-set-def*)
then have ‹(*vector-to-cblinfun* x :: 'a⇒$_{CL}$'b)∗ o_{CL} (*vector-to-cblinfun* y :: 'a⇒$_{CL}$'b) = 0›
by *simp*
with *eq* **have** ‹(*vector-to-cblinfun* x :: 'a⇒$_{CL}$'b)= 0›
by *force*
then have ‹*norm* x = 0›
by (*smt* (*verit*, *del-insts*) *norm-vector-to-cblinfun norm-zero*)
with *ortho F'-subset* ‹x ∈ F'› **show** *False*
by (*auto simp*: *spectral-dec-vecs-tc.rep-eq is-ortho-set-def*)
qed
have ‹E = (∑ (E, x)∈F. E∗ o_{CL} E)›
by (*simp add*: *E-def*)
also have ‹... = (∑ v∈F'. (*vector-to-cblinfun* v :: 'a⇒$_{CL}$'b)∗ o_{CL} *vector-to-cblinfun* v)›
unfolding *F-def*
apply (*subst sum.reindex*)
by (*auto intro*!: *inj*)
also have ‹... = (∑ v∈F'. ((*norm* (*vector-to-cblinfun* v :: 'a⇒$_{CL}$'b))^2) ∗$_R$ 1)›
by (*auto intro*!: *sum.cong simp*: *power2-norm-eq-cinner scaleR-scaleC*)
also have ‹... = (∑ v∈F'. (*norm* (*vector-to-cblinfun* v :: 'a⇒$_{CL}$'b))^2) ∗$_R$ 1›

by (*metis scaleR-left.sum*)
also have ‹... = ($\sum_\infty$v∈F'. (*norm* (*vector-to-cblinfun v* :: 'a⇒$_{CL}$'b))^2) $*_R$ 1›
using ‹*finite F'*› **by** *force*
also have ‹... ≤ ($\sum_\infty$v∈*spectral-dec-vecs-tc t*. (*norm* (*vector-to-cblinfun v* :: 'a⇒$_{CL}$'b))^2) $*_R$ 1›
apply (*intro scaleR-right-mono infsum-mono-neutral*)
using *F'-subset*
by (*auto intro*!: *one-dim-cblinfun-one-pos spectral-dec-vec-tc-norm-summable True simp*: ‹*finite F'*›)
finally show ‹*E* ≤ ($\sum_\infty$v∈*spectral-dec-vecs-tc t*. (*norm* (*vector-to-cblinfun v* :: 'a⇒$_{CL}$'b))^2) $*_R$ 1›
by −
next
show ‹*0* ∉ *fst* ‘ *Set.filter* (λ(*E*, -). *E* ≠ *0*) ((λv. (*vector-to-cblinfun v*, ())) ‘ *spectral-dec-vecs-tc t*)›
by *auto*
qed
with *True* **show** *?thesis*
by *simp*
next
case *False*
then show *?thesis*
by *simp*
qed
qed

definition *kf-constant* :: ‹('b,'b) *trace-class* ⇒ ('a::*chilbert-space*, 'b::*chilbert-space*, *unit*) *kraus-family*›
where
‹*kf-constant* ϱ = *kf-flatten* (*kf-comp* (*kf-constant-onedim* ϱ :: (*complex*,-,-) *kraus-family*) (*kf-trace some-chilbert-basis*))›

lemma *kf-constant-onedim-invalid*: ‹¬ ϱ ≥ *0* ⟹ *kf-constant-onedim* ϱ = *0*›
apply *transfer'*
by *simp*

lemma *kf-constant-invalid*: ‹¬ ϱ ≥ *0* ⟹ *kf-constant* ϱ = *0*›
by (*simp add*: *kf-constant-def kf-constant-onedim-invalid*)

lemma *kf-constant-onedim-apply*:
assumes ‹ϱ ≥ *0*›
shows ‹*kf-apply* (*kf-constant-onedim* ϱ) σ = *one-dim-iso* σ $*_C$ ϱ›
proof −
have ‹*kf-apply* (*kf-constant-onedim* ϱ) σ
= ($\sum_\infty$(E,x)∈*Set.filter* (λ(E,-). E≠*0*) ((λv. (*vector-to-cblinfun v*, ())) ‘ *spectral-dec-vecs-tc* ϱ). *sandwich-tc E* σ)›
by (*simp add*: *assms kf-apply.rep-eq kf-constant-onedim.rep-eq case-prod-unfold*)
also have ‹... = ($\sum_\infty$(E,x)∈(λv. (*vector-to-cblinfun v*, ())) ‘ *spectral-dec-vecs-tc* ϱ. *sandwich-tc E* σ)›
apply (*rule infsum-cong-neutral*)

by *auto*
also have ‹... = ($\sum_\infty v \in$ *spectral-dec-vecs-tc* ϱ. *sandwich-tc* (*vector-to-cblinfun* v) σ)›
apply (*subst infsum-reindex*)
using *vector-to-cblinfun-inj*[*of UNIV*]
by (*auto simp*: *o-def inj-on-def*)
also have ‹... = ($\sum_\infty v \in$ *spectral-dec-vecs-tc* ϱ. *one-dim-iso* σ $*_C$ *tc-butterfly* v v)›
apply (*rule infsum-cong*)
apply (*subst one-dim-scaleC-1*[*symmetric*])
apply (*rule from-trace-class-inject*[*THEN iffD1*])
apply (*simp only*: *sandwich-tc-def compose-tcl.rep-eq compose-tcr.rep-eq scaleC-trace-class.rep-eq tc-butterfly.rep-eq cblinfun-compose-scaleC-right cblinfun-compose-scaleC-left*)
by (*simp flip*: *butterfly-def-one-dim*)
also have ‹... = *one-dim-iso* σ $*_C$ ($\sum_\infty v \in$ *spectral-dec-vecs-tc* ϱ. *tc-butterfly* v v)›
using *infsum-scaleC-right* **by** *fastforce*
also have ‹... = *one-dim-iso* σ $*_C$ ϱ›
by (*simp add*: *assms butterfly-spectral-dec-vec-tc-has-sum infsumI*)
finally show *?thesis*
by −
qed

lemma *kf-constant-apply*:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-apply* (*kf-constant* ϱ) σ = *trace-tc* σ $*_C$ ϱ›
using *assms* **by** (*simp add*: *kf-constant-def kf-comp-apply kf-trace-is-trace kf-constant-onedim-apply*)

lemma *kf-bound-constant-onedim*[*simp*]:
fixes ϱ :: ‹('*a*::*chilbert-space*, '*a*) *trace-class*›
assumes ‹$\varrho \geq 0$›
shows ‹*kf-bound* (*kf-constant-onedim* ϱ) = *norm* ϱ $*_R$ *id-cblinfun*›
proof (*rule cblinfun-cinner-eqI*)
fix ψ :: '*b* **assume** ‹*norm* ψ = 1›
have ‹ψ $\cdot_C$ *kf-bound* (*kf-constant-onedim* ϱ) ψ = *trace-tc* (*kf-apply* (*kf-constant-onedim* ϱ) (*tc-butterfly* ψ ψ))›
by (*rule kf-bound-from-map*)
also have ‹... = *trace-tc* (*trace-tc* (*tc-butterfly* ψ ψ) $*_C$ ϱ)›
by (*simp add*: *kf-constant-onedim-apply assms*)
also have ‹... = *trace-tc* ϱ›
by (*metis* ‹*norm* ψ = 1› *cinner-complex-def complex-cnj-one complex-vector.vector-space-assms*(4) *norm-mult norm-one norm-tc-butterfly norm-tc-pos of-real-hom.hom-one one-cinner-one tc-butterfly-pos*)
also have ‹... = ψ $\cdot_C$ (*trace-tc* ϱ $*_C$ *id-cblinfun*) ψ›
by (*metis* ‹*norm* ψ = 1› *cblinfun.scaleC-left cinner-scaleC-right cnorm-eq-1 id-apply id-cblinfun.rep-eq of-complex-def one-dim-iso-id one-dim-iso-is-of-complex scaleC-conv-of-complex*)
also have ‹... = ψ $\cdot_C$ (*norm* ϱ $*_R$ *id-cblinfun*) ψ›
by (*simp add*: *assms norm-tc-pos scaleR-scaleC*)
finally show ‹ψ $\cdot_C$ *kf-bound* (*kf-constant-onedim* ϱ) ψ = ψ $\cdot_C$ (*norm* ϱ $*_R$ *id-cblinfun*) ψ›
by −
qed

lemma *kf-bound-constant*[*simp*]:

fixes ϱ :: ‹('a::*chilbert-space*, 'a) *trace-class*›
assumes ‹$\varrho \geq 0$›
shows ‹*kf-bound* (*kf-constant* ϱ) = *norm* ϱ $*_R$ *id-cblinfun*›
apply (*rule cblinfun-cinner-eqI*)
using *assms*
by (*simp add*: *kf-bound-from-map kf-constant-apply trace-tc-butterfly norm-tc-pos scaleR-scaleC trace-tc-scaleC*)

lemma *kf-norm-constant-onedim*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-norm* (*kf-constant-onedim* ϱ) = *norm* ϱ›
using *assms*
by (*simp add*: *kf-bound-constant kf-norm-def*)

lemma *kf-norm-constant*:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-norm* (*kf-constant* ϱ :: ('a::{*chilbert-space*,*not-singleton*},'b::*chilbert-space*,-) *kraus-family*) = *norm* ϱ›
using *assms* **by** (*simp add*: *kf-norm-def norm-cblinfun-id*)

lemma *kf-norm-constant-leq*:
shows ‹*kf-norm* (*kf-constant* ϱ) $\leq$ *norm* ϱ›
apply (*cases* ‹$\varrho \geq 0$›)
apply (*simp add*: *kf-norm-def*)
apply (*metis Groups.mult-ac*(2) *mult-cancel-right1 mult-left-mono norm-cblinfun-id-le norm-ge-zero*)
by (*simp add*: *kf-constant-invalid*)

lemma *kf-comp-constant-right*:
assumes [*iff*]: ‹$t \geq 0$›
shows ‹*kf-map fst* (*kf-comp* E (*kf-constant* t)) $\equiv_{kr}$ *kf-constant* (E $*_{kr}$ t)›
proof (*rule kf-eqI*)
fix ϱ :: ‹('b, 'b) *trace-class*› **assume** [*iff*]: ‹$\varrho \geq 0$›
have [*simp*]: ‹*fst* −‘ {()} = *UNIV*›
by *auto*
have [*simp*]: ‹{()} = *UNIV*›
by *auto*
fix x
have ‹*kf-map fst* (*kf-comp* E (*kf-constant* t)) $*_{kr}$ @{x} ϱ = *kf-comp* E (*kf-constant* t) $*_{kr}$ ϱ›
by (*simp add*: *kf-apply-on-map*)
also have ‹... = E $*_{kr}$ *trace-tc* ϱ $*_C$ t›
by (*simp add*: *kf-comp-apply kf-constant-apply*)
also have ‹... = *kf-constant* (E $*_{kr}$ t) $*_{kr}$ @{x} ϱ›
by (*simp add*: *kf-constant-apply kf-apply-pos kf-apply-scaleC*)
finally show ‹*kf-map fst* (*kf-comp* E (*kf-constant* t)) $*_{kr}$ @{x} ϱ = *kf-constant* (E $*_{kr}$ t) $*_{kr}$ @{x} ϱ›
by −
qed

lemma *kf-comp-constant-right-weak*:

assumes [*iff*]: ‹$t \geq 0$›
shows ‹*kf-comp E* (*kf-constant t*) $=_{kr}$ *kf-constant* (*E* $*_{kr}$ *t*)›
by (*metis assms kf-apply-map kf-comp-constant-right kf-eq-imp-eq-weak kf-eq-weak-def*)

2.14 Tensor products

lemma *kf-tensor-raw-bound-aux*:
fixes $\mathfrak{E}$ :: ‹(*'a ell2* $\Rightarrow_{CL}$ *'b ell2* $\times$ *'x*) *set*› **and** $\mathfrak{F}$ :: ‹(*'c ell2* $\Rightarrow_{CL}$ *'d ell2* $\times$ *'y*) *set*›
assumes ‹$\bigwedge S$. *finite S* $\Longrightarrow$ $S \subseteq \mathfrak{E} \Longrightarrow (\sum (E,\, x) \in S.\ E* \ o_{CL}\ E) \leq B$›
assumes ‹$\bigwedge S$. *finite S* $\Longrightarrow$ $S \subseteq \mathfrak{F} \Longrightarrow (\sum (E,\, x) \in S.\ E* \ o_{CL}\ E) \leq C$›
assumes ‹*finite U*›
assumes ‹$U \subseteq ((\lambda((E,\, x),\, F,\, y).\ (E \otimes_o F,\, E,\, F,\, x,\, y))\ `\ (\mathfrak{E} \times \mathfrak{F}))$›
shows ‹$(\sum (G,\, z) \in U.\ G* \ o_{CL}\ G) \leq B \otimes_o C$›
proof −
from *assms*(*1*)[**where** *S*=‹{}›] **have** [*simp*]: ‹$B \geq 0$›
by *simp*
define *f* :: ‹((*'a ell2* $\Rightarrow_{CL}$ *'b ell2* $\times$ *'x*) $\times$ (*'c ell2* $\Rightarrow_{CL}$ *'d ell2* $\times$ *'y*)) $\Rightarrow$ -›
where ‹$f = (\lambda((E,x),\, (F,y)).\ (E \otimes_o F,\, (E,F,x,y)))$›
from *assms*
obtain *V* **where** *V-subset*: ‹$V \subseteq \mathfrak{E} \times \mathfrak{F}$› **and** [*simp*]: ‹*finite V*› **and** ‹$U = f\ `\ V$›
apply (*simp flip*: *f-def*)
by (*meson finite-subset-image*)
define *W* **where** ‹$W = fst\ `\ V \times snd\ `\ V$›
have ‹*inj-on f W*›
by (*auto intro*!: *inj-onI simp*: *f-def*)
from ‹*finite V*› **have** [*simp*]: ‹*finite W*›
using *W-def* **by** *blast*
have ‹$W \supseteq V$›
by (*auto intro*!: *image-eqI simp*: *W-def*)
have ‹$(\sum (G,\, z) \in U.\ G* \ o_{CL}\ G) \leq (\sum (G,\, z) \in f\ `\ W.\ G* \ o_{CL}\ G)$›
using ‹$U = f\ `\ V$› ‹$V \subseteq W$›
by (*auto intro*!: *sum-mono2 positive-cblinfun-squareI*)
also have ‹$\ldots = (\sum ((E,x),(F,y)) \in W.\ (E \otimes_o F)* \ o_{CL}\ (E \otimes_o F))$›
apply (*subst sum.reindex*)
using ‹*inj-on f W*›
by (*auto simp*: *case-prod-unfold f-def*)
also have ‹$\ldots = (\sum ((E,x),(F,y)) \in W.\ (E* \ o_{CL}\ E) \otimes_o (F* \ o_{CL}\ F))$›
by (*simp add*: *comp-tensor-op tensor-op-adjoint*)
also have ‹$\ldots = (\sum (E,x) \in fst`V.\ E* \ o_{CL}\ E) \otimes_o (\sum (F,y) \in snd`V.\ F* \ o_{CL}\ F)$›
unfolding *W-def*
apply (*subst sum.Sigma*[*symmetric*])
apply *simp*
apply *simp*
apply (*simp add*: *case-prod-beta tensor-op-cbilinear.sum-left*)
by (*simp add*: *tensor-op-cbilinear.sum-right*)
also have ‹$\ldots \leq B \otimes_o C$›
using *V-subset*
by (*auto intro*!: *tensor-op-mono assms sum-nonneg intro*: *positive-cblinfun-squareI*)
finally show *?thesis*

```
  by−
qed


lift-definition kf-tensor-raw :: ‹('a ell2, 'b ell2, 'x) kraus-family ⇒ ('c ell2, 'd ell2, 'y) kraus-family
⇒
      (('a×'c) ell2, ('b×'d) ell2, (('a ell2⇒CL'b ell2)×('c ell2⇒CL'd ell2)×'x×'y)) kraus-family›
is
  ‹λ𝔈 𝔉. (λ((E,x), (F,y)). (E ⊗o F, (E,F,x,y))) ‘ (𝔈×𝔉)›
proof (rename-tac 𝔈 𝔉, intro CollectI)
  fix 𝔈 :: ‹('a ell2 ⇒CL 'b ell2 × 'x) set› and 𝔉 :: ‹('c ell2 ⇒CL 'd ell2 × 'y) set›
  assume ‹𝔈 ∈ Collect kraus-family› and ‹𝔉 ∈ Collect kraus-family›
  then have ‹kraus-family 𝔈› and ‹kraus-family 𝔉›
    by auto
  define tensor where ‹tensor = ((λ((E, x), F, y). (E ⊗o F, E, F, x, y)) ‘ (𝔈 × 𝔉))›
  show ‹kraus-family tensor›
  proof (intro kraus-familyI)
    from ‹kraus-family 𝔈›
    obtain B where B: ‹(∑(E, x)∈S. E* oCL E) ≤ B› if ‹finite S› and ‹S ⊆ 𝔈› for S
      apply atomize-elim
      by (auto simp: kraus-family-def bdd-above-def)
    from B[where S=‹{}›] have [simp]: ‹B ≥ 0›
      by simp
    from ‹kraus-family 𝔉›
    obtain C where C: ‹(∑(F, x)∈T. F* oCL F) ≤ C› if ‹finite T› and ‹T ⊆ 𝔉› for T
      apply atomize-elim
      by (auto simp: kraus-family-def bdd-above-def)
    have ‹(∑(G, z)∈U. G* oCL G) ≤ B ⊗o C› if ‹finite U› and ‹U ⊆ tensor› for U
      using that by (auto intro!: kf-tensor-raw-bound-aux B C simp: tensor-def)
    then show ‹bdd-above ((λF. ∑(E, x)∈F. E* oCL E) ‘ {F. finite F ∧ F ⊆ tensor})›
      by fast
    show ‹0 ∉ fst ‘ tensor›
    proof (rule notI)
      assume ‹0 ∈ fst ‘ tensor›
      then obtain E F x y where EF0: ‹E ⊗o F = 0› and Ex: ‹(E,x) ∈ 𝔈› and Fy: ‹(F,y) ∈
𝔉›
        by (auto intro!: simp: tensor-def)
      from ‹kraus-family 𝔈› Ex
      have ‹E ≠ 0›
        by (force simp: kraus-family-def)
      from ‹kraus-family 𝔉› Fy
      have ‹F ≠ 0›
        by (force simp: kraus-family-def)
      from ‹E ≠ 0› ‹F ≠ 0› have ‹E ⊗o F ≠ 0›
        using tensor-op-nonzero by blast
      with EF0 show False
        by simp
    qed
  qed
```

qed

lemma *kf-apply-tensor-raw-as-infsum*:
‹*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ ϱ = ($\sum_\infty$((*E*,*x*),(*F*,*y*))∈*Rep-kraus-family* $\mathfrak{E}$ × *Rep-kraus-family* $\mathfrak{F}$. *sandwich-tc* (*E* $\otimes_o$ *F*) ϱ)›
proof −
have *inj*: ‹*inj-on* (λ((*E*, *x*), *F*, *y*). (*E* $\otimes_o$ *F*, *E*, *F*, *x*, *y*)) (*Rep-kraus-family* $\mathfrak{E}$ × *Rep-kraus-family* $\mathfrak{F}$)›
by (*auto intro*!: *inj-onI*)
show ‹*kf-apply* (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$) ϱ
= ($\sum_\infty$((*E*,*x*),(*F*,*y*))∈*Rep-kraus-family* $\mathfrak{E}$ × *Rep-kraus-family* $\mathfrak{F}$. *sandwich-tc* (*E* $\otimes_o$ *F*) ϱ)›
apply (*simp add*: *kf-apply.rep-eq kf-tensor-raw.rep-eq infsum-reindex inj o-def*)
by (*simp add*: *case-prod-unfold*)
qed

lemma *kf-apply-tensor-raw*:
shows ‹*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ *tc-tensor* ϱ σ = *tc-tensor* ($\mathfrak{E}$ $*_{kr}$ ϱ) ($\mathfrak{F}$ $*_{kr}$ σ)›
proof −
have *inj*: ‹*inj-on* (λ((*E*, *x*), *F*, *y*). (*E* $\otimes_o$ *F*, *E*, *F*, *x*, *y*)) (*Rep-kraus-family* $\mathfrak{E}$ × *Rep-kraus-family* $\mathfrak{F}$)›
by (*auto intro*!: *inj-onI*)
have [*simp*]: ‹*bounded-linear* (λ*x*. *tc-tensor x* (*kf-apply* $\mathfrak{F}$ σ))›
by (*intro bounded-linear-intros*)
have [*simp*]: ‹*bounded-linear* (*tc-tensor* (*sandwich-tc E* ϱ))› **for** *E*
by (*intro bounded-linear-intros*)
have *sum2*: ‹(λ(*E*, *x*). *sandwich-tc E* ϱ) *summable-on Rep-kraus-family* $\mathfrak{E}$›
using *kf-apply-summable* **by** *blast*
have *sum3*: ‹(λ(*F*, *y*). *sandwich-tc F* σ) *summable-on Rep-kraus-family* $\mathfrak{F}$›
using *kf-apply-summable* **by** *blast*

from *kf-apply-summable*[*of* - ‹*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$›]
have *sum1*: ‹(λ((*E*, *x*), *F*, *y*). *sandwich-tc* (*E* $\otimes_o$ *F*) (*tc-tensor* ϱ σ)) *summable-on Rep-kraus-family* $\mathfrak{E}$ × *Rep-kraus-family* $\mathfrak{F}$›
apply (*simp add*: *kf-apply.rep-eq kf-tensor-raw.rep-eq summable-on-reindex inj o-def*)
by (*simp add*: *case-prod-unfold*)

have ‹*kf-apply* (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$) (*tc-tensor* ϱ σ)
= ($\sum_\infty$((*E*,*x*),(*F*,*y*))∈*Rep-kraus-family* $\mathfrak{E}$ × *Rep-kraus-family* $\mathfrak{F}$. *sandwich-tc* (*E* $\otimes_o$ *F*) (*tc-tensor* ϱ σ))›
by (*rule kf-apply-tensor-raw-as-infsum*)
also have ‹... = ($\sum_\infty$(*E*,*x*)∈*Rep-kraus-family* $\mathfrak{E}$. $\sum_\infty$(*F*,*y*)∈*Rep-kraus-family* $\mathfrak{F}$. *sandwich-tc* (*E* $\otimes_o$ *F*) (*tc-tensor* ϱ σ))›
apply (*subst infsum-Sigma-banach*[*symmetric*])
using *sum1* **by** (*auto intro*!: *simp*: *case-prod-unfold*)
also have ‹... = ($\sum_\infty$(*E*,*x*)∈*Rep-kraus-family* $\mathfrak{E}$. $\sum_\infty$(*F*,*y*)∈*Rep-kraus-family* $\mathfrak{F}$. *tc-tensor* (*sandwich-tc E* ϱ) (*sandwich-tc F* σ))›
by (*simp add*: *sandwich-tc-tensor*)
also have ‹... = ($\sum_\infty$(*E*,*x*)∈*Rep-kraus-family* $\mathfrak{E}$. *tc-tensor* (*sandwich-tc E* ϱ) ($\sum_\infty$(*F*,*y*)∈*Rep-kraus-family* $\mathfrak{F}$. (*sandwich-tc F* σ)))›

apply (*subst infsum-bounded-linear*[**where** *h*=‹*tc-tensor* (*sandwich-tc* - *ϱ*)›, *symmetric*])
apply (*auto intro*!: *sum3*)[*2*]
by (*simp add*: *o-def case-prod-unfold*)
also have ‹... = (∑∞(*E*,*x*)∈*Rep-kraus-family* 𝔈. *tc-tensor* (*sandwich-tc E ϱ*) (*kf-apply* 𝔉 *σ*))›
by (*simp add*: *kf-apply-def case-prod-unfold*)
also have ‹... = *tc-tensor* (∑∞(*E*,*x*)∈*Rep-kraus-family* 𝔈. *sandwich-tc E ϱ*) (*kf-apply* 𝔉 *σ*)›
apply (*subst infsum-bounded-linear*[**where** *h*=‹λ*x*. *tc-tensor x* (*kf-apply* 𝔉 *σ*)›, *symmetric*])
apply (*auto intro*!: *sum2*)[*2*]
by (*simp add*: *o-def case-prod-unfold*)
also have ‹... = *tc-tensor* (*kf-apply* 𝔈 *ϱ*) (*kf-apply* 𝔉 *σ*)›
by (*simp add*: *kf-apply-def case-prod-unfold*)
finally show *?thesis*
by −
qed

hide-fact *kf-tensor-raw-bound-aux*

definition ‹*kf-tensor* 𝔈 𝔉 = *kf-map* (λ(*E*, *F*, *x*, *y*). (*x*,*y*)) (*kf-tensor-raw* 𝔈 𝔉)›

lemma *kf-apply-tensor*:
‹*kf-tensor* 𝔈 𝔉 ∗$_{kr}$ *tc-tensor ϱ σ* = *tc-tensor* (𝔈 ∗$_{kr}$ *ϱ*) (𝔉 ∗$_{kr}$ *σ*)›
by (*auto intro*!: *simp*: *kf-tensor-def kf-apply-map kf-apply-tensor-raw*)

lemma *kf-apply-tensor-as-infsum*:
‹*kf-tensor* 𝔈 𝔉 ∗$_{kr}$ *ϱ* = (∑∞((*E*,*x*),(*F*,*y*))∈*Rep-kraus-family* 𝔈 × *Rep-kraus-family* 𝔉. *sandwich-tc* (*E* ⊗$_o$ *F*) *ϱ*)›
by (*simp add*: *kf-tensor-def kf-apply-tensor-raw-as-infsum*)

lemma *kf-bound-tensor-raw*:
‹*kf-bound* (*kf-tensor-raw* 𝔈 𝔉) = *kf-bound* 𝔈 ⊗$_o$ *kf-bound* 𝔉›
proof (*rule cblinfun-cinner-tensor-eqI*)
fix *ψ φ*

from *kf-bound-summable*[*of* ‹*kf-tensor-raw* 𝔈 𝔉›]
have *sum1*: ‹*summable-on-in cweak-operator-topology* (λ((*E*,*x*),(*F*,*y*)). (*E* ⊗$_o$ *F*)∗ *o*$_{CL}$ (*E* ⊗$_o$ *F*))
(*Rep-kraus-family* 𝔈 × *Rep-kraus-family* 𝔉)›
unfolding *kf-tensor-raw.rep-eq*
apply (*subst* (*asm*) *summable-on-in-reindex*)
by (*auto simp add*: *kf-tensor-raw.rep-eq case-prod-unfold inj-on-def o-def*)
have *sum4*: ‹*summable-on-in cweak-operator-topology* (λ(*E*,*x*). *E*∗ *o*$_{CL}$ *E*) (*Rep-kraus-family* 𝔈)›
using *kf-bound-summable* **by** *blast*
have *sum5*: ‹*summable-on-in cweak-operator-topology* (λ(*F*,*y*). *F*∗ *o*$_{CL}$ *F*) (*Rep-kraus-family* 𝔉)›
using *kf-bound-summable* **by** *blast*
have *sum2*: ‹(λ(*E*, *x*). *ψ* ∙$_C$ ((*E*∗ *o*$_{CL}$ *E*) ∗$_V$ *ψ*)) *abs-summable-on Rep-kraus-family* 𝔈›

using *kf-bound-summable* **by** (*auto intro*!: *summable-on-in-cweak-operator-topology-pointwise*

simp add: *case-prod-unfold simp flip*: *summable-on-iff-abs-summable-on-complex*
simp del: *cblinfun-apply-cblinfun-compose*)
have *sum3*: ‹(λ(*F*,*y*). *φ* ·$_C$ ((*F*∗ o_{CL} *F*) ∗$_V$ *φ*)) *abs-summable-on Rep-kraus-family* 𝔉›
using *kf-bound-summable* **by** (*auto intro*!: *summable-on-in-cweak-operator-topology-pointwise*
simp add: *case-prod-unfold simp flip*: *summable-on-iff-abs-summable-on-complex*
simp del: *cblinfun-apply-cblinfun-compose*)

have ‹(*ψ* ⊗$_s$ *φ*) ·$_C$ (*kf-bound* (*kf-tensor-raw* 𝔈 𝔉) ∗$_V$ *ψ* ⊗$_s$ *φ*)
= (*ψ* ⊗$_s$ *φ*) ·$_C$
(*infsum-in cweak-operator-topology* ((λ(*E*, *x*). *E*∗ o_{CL} *E*) ∘ (λ((*E*, *x*), *F*, *y*). (*E* ⊗$_o$ *F*,
E, *F*, *x*, *y*)))
(*Rep-kraus-family* 𝔈 × *Rep-kraus-family* 𝔉) ∗$_V$
ψ ⊗$_s$ *φ*)›
unfolding *kf-bound.rep-eq kf-tensor-raw.rep-eq*
apply (*subst infsum-in-reindex*)
by (*auto simp add*: *inj-on-def case-prod-unfold*)
also have ‹... = (*ψ* ⊗$_s$ *φ*) ·$_C$
(*infsum-in cweak-operator-topology* (λ((*E*,*x*),(*F*,*y*)). (*E* ⊗$_o$ *F*)∗ o_{CL} (*E* ⊗$_o$ *F*))
(*Rep-kraus-family* 𝔈 × *Rep-kraus-family* 𝔉) ∗$_V$
ψ ⊗$_s$ *φ*)›
by (*simp add*: *o-def case-prod-unfold*)
also have ‹... = (∑$_\infty$((*E*,*x*),(*F*,*y*)) ∈ *Rep-kraus-family* 𝔈 × *Rep-kraus-family* 𝔉.
(*ψ* ⊗$_s$ *φ*) ·$_C$ ((*E* ⊗$_o$ *F*)∗ o_{CL} (*E* ⊗$_o$ *F*)) (*ψ* ⊗$_s$ *φ*))›
apply (*subst infsum-in-cweak-operator-topology-pointwise*)
using *sum1* **by** (*auto intro*!: *simp*: *case-prod-unfold*)
also have ‹... = (∑$_\infty$((*E*,*x*),(*F*,*y*)) ∈ *Rep-kraus-family* 𝔈 × *Rep-kraus-family* 𝔉.
(*ψ* ·$_C$ (*E*∗ o_{CL} *E*) *ψ*) ∗ (*φ* ·$_C$ (*F*∗ o_{CL} *F*) *φ*))›
apply (*rule infsum-cong*)
by (*simp-all add*: *tensor-op-adjoint tensor-op-ell2*)
also have ‹... = (∑$_\infty$(*E*,*x*) ∈ *Rep-kraus-family* 𝔈. *ψ* ·$_C$ (*E*∗ o_{CL} *E*) *ψ*)
∗ (∑$_\infty$(*F*,*y*) ∈ *Rep-kraus-family* 𝔉. *φ* ·$_C$ (*F*∗ o_{CL} *F*) *φ*)›
apply (*subst infsum-product'*)
using *sum2 sum3* **by** (*simp-all add*: *case-prod-unfold*)
also have ‹... = (*ψ* ·$_C$ *kf-bound* 𝔈 *ψ*) ∗ (*φ* ·$_C$ *kf-bound* 𝔉 *φ*)›
unfolding *kf-bound.rep-eq case-prod-unfold*
apply (*subst infsum-in-cweak-operator-topology-pointwise*[*symmetric*])
using *sum4* **apply** (*simp add*: *case-prod-unfold*)
apply (*subst infsum-in-cweak-operator-topology-pointwise*[*symmetric*])
using *sum5* **apply** (*simp add*: *case-prod-unfold*)
by (*rule refl*)
also have ‹... = (*ψ* ⊗$_s$ *φ*) ·$_C$ ((*kf-bound* 𝔈 ⊗$_o$ *kf-bound* 𝔉) ∗$_V$ *ψ* ⊗$_s$ *φ*)›
by (*simp add*: *tensor-op-ell2*)
finally show ‹(*ψ* ⊗$_s$ *φ*) ·$_C$ (*kf-bound* (*kf-tensor-raw* 𝔈 𝔉) ∗$_V$ *ψ* ⊗$_s$ *φ*) =
(*ψ* ⊗$_s$ *φ*) ·$_C$ ((*kf-bound* 𝔈 ⊗$_o$ *kf-bound* 𝔉) ∗$_V$ *ψ* ⊗$_s$ *φ*)›
by −
qed

lemma *kf-bound-tensor*:
‹*kf-bound* (*kf-tensor* 𝔈 𝔉) = *kf-bound* 𝔈 ⊗ₒ *kf-bound* 𝔉›
by (*simp add*: *kf-tensor-def kf-map-bound kf-bound-tensor-raw*)

lemma *kf-norm-tensor*:
‹*kf-norm* (*kf-tensor* 𝔈 𝔉) = *kf-norm* 𝔈 * *kf-norm* 𝔉›
by (*auto intro*!: *norm-cblinfun-mono*
simp add: *kf-norm-def kf-bound-tensor*
simp flip: *tensor-op-norm*)

lemma *kf-tensor-cong-weak*:
assumes ‹𝔈 =$_{kr}$ 𝔈'›
assumes ‹𝔉 =$_{kr}$ 𝔉'›
shows ‹*kf-tensor* 𝔈 𝔉 =$_{kr}$ *kf-tensor* 𝔈' 𝔉'›
proof (*rule kf-eq-weakI*)
show ‹*kf-apply* (*kf-tensor* 𝔈 𝔉) ϱ = *kf-apply* (*kf-tensor* 𝔈' 𝔉') ϱ› **for** ϱ
proof (*rule eq-from-separatingI2x*[**where** *x*=ϱ, *OF separating-set-bounded-clinear-tc-tensor*])
show ‹*bounded-clinear* (*kf-apply* (*kf-tensor* 𝔈 𝔉))›
by (*simp add*: *kf-apply-bounded-clinear*)
show ‹*bounded-clinear* (*kf-apply* (*kf-tensor* 𝔈' 𝔉'))›
by (*simp add*: *kf-apply-bounded-clinear*)
have 𝔈𝔈': ‹*kf-apply* 𝔈 ϱ = *kf-apply* 𝔈' ϱ› **for** ϱ
by (*metis assms*(*1*) *kf-eq-weak-def*)
have 𝔉𝔉': ‹*kf-apply* 𝔉 ϱ = *kf-apply* 𝔉' ϱ› **for** ϱ
by (*metis assms*(*2*) *kf-eq-weak-def*)
fix ϱ :: ‹('*a ell2*, '*a ell2*) *trace-class*› **and** σ :: ‹('*e ell2*, '*e ell2*) *trace-class*›
show ‹*kf-apply* (*kf-tensor* 𝔈 𝔉) (*tc-tensor* ϱ σ) = *kf-apply* (*kf-tensor* 𝔈' 𝔉') (*tc-tensor* ϱ σ)›
by (*auto intro*!: *simp*: *kf-apply-tensor* 𝔈𝔈' 𝔉𝔉'
simp flip: *tensor-ell2-ket tensor-tc-butterfly*)
qed
qed

lemma *kf-filter-tensor*:
‹*kf-filter* (λ(*x*,*y*). *P x* ∧ *Q y*) (*kf-tensor* 𝔈 𝔉) = *kf-tensor* (*kf-filter P* 𝔈) (*kf-filter Q* 𝔉)›
apply (*simp add*: *kf-tensor-def kf-filter-map*)
apply (*rule arg-cong*[**where** *f*=‹*kf-map* -›])
apply *transfer*
by (*force simp add*: *image-iff case-prod-unfold*)

lemma *kf-filter-tensor-singleton*:
‹*kf-filter* ((=) *x*) (*kf-tensor* 𝔈 𝔉) = *kf-tensor* (*kf-filter* ((=) (*fst x*)) 𝔈) (*kf-filter* ((=) (*snd x*)) 𝔉)›
by (*simp add*: *kf-filter-tensor*[*symmetric*] *case-prod-unfold prod-eq-iff*)

lemma *kf-tensor-cong*:
fixes 𝔈 𝔈' :: ‹('*a ell2*, '*b ell2*, '*x*) *kraus-family*›
and 𝔉 𝔉' :: ‹('*c ell2*, '*d ell2*, '*y*) *kraus-family*›
assumes ‹𝔈 ≡$_{kr}$ 𝔈'›

assumes ‹$\mathfrak{F} \equiv_{kr} \mathfrak{F}'$›
shows ‹*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$ $\equiv_{kr}$ *kf-tensor* $\mathfrak{E}'$ $\mathfrak{F}'$›
proof (*rule kf-eqI*)
fix *xy* :: ‹$'x \times 'y$› **and** ϱ
obtain *x y* **where** [*simp*]: ‹*xy* = (*x*,*y*)›
by *fastforce*
have *aux1*: ‹$(\lambda xy'.\ xy' = (x,\ y)) = (\lambda(x',\ y').\ x' = x \wedge y' = y)$›
by *auto*
have ‹*kf-apply-on* (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$) {*xy*}
= *kf-apply* (*kf-tensor* (*kf-filter* ($\lambda z.\ z = x$) $\mathfrak{E}$) (*kf-filter* ($\lambda z.\ z = y$) $\mathfrak{F}$))›
apply (*simp add*: *kf-apply-on-def aux1 kf-filter-tensor*)
apply (*subst aux1*)
by (*simp add*: *kf-apply-on-def aux1 kf-filter-tensor*)
also have ‹... = *kf-apply* (*kf-tensor* (*kf-filter* ($\lambda z.\ z = x$) $\mathfrak{E}'$) (*kf-filter* ($\lambda z.\ z = y$) $\mathfrak{F}'$))›
apply (*rule ext*)
apply (*rule kf-apply-eqI*)
apply (*rule kf-tensor-cong-weak*)
by (*auto intro*!: *kf-filter-cong-weak assms*)
also have ‹... = *kf-apply-on* (*kf-tensor* $\mathfrak{E}'$ $\mathfrak{F}'$) {*xy*}›
apply (*simp add*: *kf-apply-on-def aux1 kf-filter-tensor*)
apply (*subst aux1*)
by (*simp add*: *kf-apply-on-def aux1 kf-filter-tensor*)
finally show ‹*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$ $*_{kr}$ @{*xy*} ϱ = *kf-tensor* $\mathfrak{E}'$ $\mathfrak{F}'$ $*_{kr}$ @{*xy*} ϱ›
by *simp*
qed

lemma *kf-tensor-compose-distrib-weak*:
shows ‹*kf-tensor* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$) (*kf-comp* $\mathfrak{G}$ $\mathfrak{H}$)
$=_{kr}$ *kf-comp* (*kf-tensor* $\mathfrak{E}$ $\mathfrak{G}$) (*kf-tensor* $\mathfrak{F}$ $\mathfrak{H}$)›
by (*auto intro*!: *eq-from-separatingI2*[*OF separating-set-bounded-clinear-tc-tensor*]
kf-apply-bounded-clinear comp-bounded-clinear
simp: *kf-eq-weak-def kf-apply-tensor kf-comp-apply*)

lemma *kf-tensor-compose-distrib*:
shows ‹*kf-tensor* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$) (*kf-comp* $\mathfrak{G}$ $\mathfrak{H}$)
$\equiv_{kr}$ *kf-map* ($\lambda((e,g),(f,h))$. ((*e*,*f*),(*g*,*h*))) (*kf-comp* (*kf-tensor* $\mathfrak{E}$ $\mathfrak{G}$) (*kf-tensor* $\mathfrak{F}$ $\mathfrak{H}$))›
proof (*rule kf-eqI-from-filter-eq-weak*)
fix *efgh* :: ‹$('e \times 'f) \times ('g \times 'h)$›
obtain *e f g h* **where** *efgh*: ‹*efgh* = ((*e*,*f*),(*g*,*h*))›
apply *atomize-elim*
apply (*cases efgh*)
by *auto*
have ‹*kf-filter* ((=) *efgh*) (*kf-tensor* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$) (*kf-comp* $\mathfrak{G}$ $\mathfrak{H}$))
$=_{kr}$ *kf-filter* ($\lambda(x,y)$. (*e*,*f*)=*x* $\wedge$ (*g*,*h*)=*y*) (*kf-tensor* (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$) (*kf-comp* $\mathfrak{G}$ $\mathfrak{H}$))›
by (*smt* (*verit*, *best*) *case-prod-unfold kf-eq-def kf-filter-cong-weak split-pairs2 efgh*)
also have ‹... = *kf-tensor* (*kf-filter* ((=) (*e*,*f*)) (*kf-comp* $\mathfrak{E}$ $\mathfrak{F}$)) (*kf-filter* ((=) (*g*,*h*)) (*kf-comp* $\mathfrak{G}$ $\mathfrak{H}$))›
by (*simp add*: *kf-filter-tensor*)

also have ‹... = *kf-tensor* (*kf-filter* (λ(*e'*,*f'*). *f*=*f'* ∧ *e*=*e'*) (*kf-comp* 𝔈 𝔉)) (*kf-filter* (λ(*g'*,*h'*). *h*=*h'* ∧ *g*=*g'*) (*kf-comp* 𝔊 ℌ))›
apply (*intro arg-cong2*[**where** *f*=*kf-tensor*] *arg-cong2*[**where** *f*=*kf-filter*])
by *auto*
also have ‹... = *kf-tensor* (*kf-comp* (*kf-filter* ((=) *f*) 𝔈) (*kf-filter* ((=) *e*) 𝔉)) (*kf-comp* (*kf-filter* ((=) *h*) 𝔊) (*kf-filter* ((=) *g*) ℌ))›
by (*simp add*: *kf-filter-comp*)
also have ‹... =$_{kr}$ *kf-comp* (*kf-tensor* (*kf-filter* ((=) *f*) 𝔈) (*kf-filter* ((=) *h*) 𝔊)) (*kf-tensor* (*kf-filter* ((=) *e*) 𝔉) (*kf-filter* ((=) *g*) ℌ))›
by (*rule kf-tensor-compose-distrib-weak*)
also have ‹... =$_{kr}$ *kf-filter* (λ((*e'*,*g'*), (*f'*,*h'*)). (*f* = *f'* ∧ *h* = *h'*) ∧ (*e* = *e'* ∧ *g* = *g'*)) (*kf-comp* (*kf-tensor* 𝔈 𝔊) (*kf-tensor* 𝔉 ℌ))›
by (*simp add*: *case-prod-unfold flip*: *kf-filter-tensor kf-filter-comp*)
also have ‹... =$_{kr}$ *kf-map* (λ((*e*,*g*),(*f*,*h*)). ((*e*,*f*),(*g*,*h*))) (*kf-filter* (λ((*e'*,*g'*), (*f'*,*h'*)). (*f* = *f'* ∧ *h* = *h'*) ∧ (*e* = *e'* ∧ *g* = *g'*)) (*kf-comp* (*kf-tensor* 𝔈 𝔊) (*kf-tensor* 𝔉 ℌ)))›
by (*simp add*: *kf-eq-weak-def*)
also have ‹... =$_{kr}$ *kf-filter* (λ((*e'*,*f'*), (*g'*,*h'*)). (*f* = *f'* ∧ *h* = *h'*) ∧ (*e* = *e'* ∧ *g* = *g'*)) (*kf-map* (λ((*e*,*g*),(*f*,*h*)). ((*e*,*f*),(*g*,*h*))) (*kf-comp* (*kf-tensor* 𝔈 𝔊) (*kf-tensor* 𝔉 ℌ)))›
by (*simp add*: *kf-filter-map case-prod-unfold*)
also have ‹... = *kf-filter* ((=) *efgh*) (*kf-map* (λ((*e*,*g*),(*f*,*h*)). ((*e*,*f*),(*g*,*h*))) (*kf-comp* (*kf-tensor* 𝔈 𝔊) (*kf-tensor* 𝔉 ℌ)))›
apply (*rule arg-cong2*[**where** *f*=‹*kf-filter*›])
by (*auto simp*: *efgh*)
finally show ‹*kf-filter* ((=) *efgh*) (*kf-tensor* (*kf-comp* 𝔈 𝔉) (*kf-comp* 𝔊 ℌ)) =$_{kr}$ ...›
by −
qed

lemma *kf-tensor-compose-distrib'*:
shows ‹*kf-comp* (*kf-tensor* 𝔈 𝔊) (*kf-tensor* 𝔉 ℌ)
≡$_{kr}$ *kf-map* (λ((*e*,*f*),(*g*,*h*)). ((*e*,*g*),(*f*,*h*))) (*kf-tensor* (*kf-comp* 𝔈 𝔉) (*kf-comp* 𝔊 ℌ))›
proof −
have *aux*: ‹((λ((*e*,*f*),(*g*,*h*)). ((*e*,*g*),(*f*,*h*))) *o* (λ((*e*,*g*),(*f*,*h*)). ((*e*,*f*),(*g*,*h*)))) = *id*›
by *auto*
have ‹*kf-comp* (*kf-tensor* 𝔈 𝔊) (*kf-tensor* 𝔉 ℌ) ≡$_{kr}$ *kf-map* ((λ((*e*,*f*),(*g*,*h*)). ((*e*,*g*),(*f*,*h*))) *o* (λ((*e*,*g*),(*f*,*h*)). ((*e*,*f*),(*g*,*h*)))) (*kf-comp* (*kf-tensor* 𝔈 𝔊) (*kf-tensor* 𝔉 ℌ))›
by (*simp add*: *aux kf-eq-sym kf-map-id*)
also have ‹... ≡$_{kr}$ *kf-map* (λ((*e*,*f*),(*g*,*h*)). ((*e*,*g*),(*f*,*h*))) (*kf-map* (λ((*e*,*g*),(*f*,*h*)). ((*e*,*f*),(*g*,*h*))) (*kf-comp* (*kf-tensor* 𝔈 𝔊) (*kf-tensor* 𝔉 ℌ)))›
using *kf-eq-sym kf-map-twice* **by** *blast*
also have ‹... ≡$_{kr}$ *kf-map* (λ((*e*,*f*),(*g*,*h*)). ((*e*,*g*),(*f*,*h*))) (*kf-tensor* (*kf-comp* 𝔈 𝔉) (*kf-comp* 𝔊 ℌ))›
using *kf-eq-sym kf-map-cong*[*OF refl*] *kf-tensor-compose-distrib* **by** *blast*
finally show *?thesis*
by −
qed

definition *kf-tensor-right* :: ‹('*extra ell2*, '*extra ell2*) *trace-class* ⇒ ('*qu ell2*, ('*qu*×'*extra*) *ell2*, *unit*) *kraus-family*› **where**

— *kf-tensor-right* ϱ maps σ to $\sigma \otimes_o \varrho$
‹*kf-tensor-right* ϱ = *kf-map-inj* (λ-. ()) (*kf-comp* (*kf-tensor kf-id* (*kf-constant-onedim* ϱ)) (*kf-of-op* (*tensor-ell2-right* (*ket* ())))))›
definition *kf-tensor-left* :: ‹('*extra ell2*, '*extra ell2*) *trace-class* ⇒ ('*qu ell2*, ('*extra*×'*qu*) *ell2*, *unit*) *kraus-family*› **where**
— *kf-tensor-right* ϱ maps σ to $\varrho \otimes_o \sigma$
‹*kf-tensor-left* ϱ = *kf-map-inj* (λ-. ()) (*kf-comp* (*kf-tensor* (*kf-constant-onedim* ϱ) *kf-id*) (*kf-of-op* (*tensor-ell2-left* (*ket* ())))))›

lemma *kf-apply-tensor-right*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-tensor-right* ϱ $*_{kr}$ σ = *tc-tensor* σ ϱ›
proof −
have $*$: ‹*sandwich-tc* (*tensor-ell2-right* (*ket* ())) σ = *tc-tensor* σ (*tc-butterfly* (*ket*()) (*ket*()))›
apply *transfer'*
using *sandwich-tensor-ell2-right'* **by** *blast*
show *?thesis*
by (*simp add*: *kf-tensor-right-def kf-apply-map-inj inj-on-def kf-comp-apply*
kf-of-op-apply $*$ *kf-apply-tensor kf-constant-onedim-apply assms trace-tc-butterfly*
flip: *trace-tc-one-dim-iso*)
qed
lemma *kf-apply-tensor-left*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-tensor-left* ϱ $*_{kr}$ σ = *tc-tensor* ϱ σ›
proof −
have $*$: ‹*sandwich-tc* (*tensor-ell2-left* (*ket* ())) σ = *tc-tensor* (*tc-butterfly* (*ket*()) (*ket*())) σ›
apply *transfer'*
using *sandwich-tensor-ell2-left'* **by** *blast*
show *?thesis*
by (*simp add*: *kf-tensor-left-def kf-apply-map-inj inj-on-def kf-comp-apply*
kf-of-op-apply $*$ *kf-apply-tensor kf-constant-onedim-apply assms trace-tc-butterfly*
flip: *trace-tc-one-dim-iso*)
qed

lemma *kf-bound-tensor-right*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-bound* (*kf-tensor-right* ϱ) = *norm* ϱ $*_C$ *id-cblinfun*›
proof (*rule cblinfun-cinner-eqI*)
fix ψ :: ‹'*b ell2*›
have ‹ψ $\cdot_C$ *kf-bound* (*kf-tensor-right* ϱ) ψ = *trace-tc* (*kf-tensor-right* ϱ $*_{kr}$ *tc-butterfly* ψ ψ)›
by (*simp add*: *kf-bound-from-map*)
also have ‹... = *trace-tc* (*tc-tensor* (*tc-butterfly* ψ ψ) ϱ)›
using *assms* **by** (*simp add*: *kf-apply-tensor-right*)
also have ‹... = *trace-tc* (*tc-butterfly* ψ ψ) $*$ *trace-tc* ϱ›
by (*metis* (*no-types*, *lifting*) *assms norm-tc-pos norm-tc-tensor of-real-mult tc-butterfly-pos tc-tensor-pos*)
also have ‹... = *norm* ϱ $*$ *trace-tc* (*tc-butterfly* ψ ψ)›
by (*simp add*: *assms norm-tc-pos*)
also have ‹... = ψ $\cdot_C$ (*norm* ϱ $*_C$ *id-cblinfun*) ψ›

by (*simp add*: *trace-tc-butterfly*)
finally show ‹$\psi \cdot_C$ (*kf-bound* (*kf-tensor-right* ϱ)) $\psi = \psi \cdot_C$ (*norm* $\varrho *_C$ *id-cblinfun*) ψ›
by −
qed
lemma *kf-bound-tensor-left*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-bound* (*kf-tensor-left* ϱ) = *norm* $\varrho *_C$ *id-cblinfun*›
proof (*rule cblinfun-cinner-eqI*)
fix ψ :: ‹'*b ell2*›
have ‹$\psi \cdot_C$ *kf-bound* (*kf-tensor-left* ϱ) ψ = *trace-tc* (*kf-tensor-left* $\varrho *_{kr}$ *tc-butterfly* ψ ψ)›
by (*simp add*: *kf-bound-from-map*)
also have ‹... = *trace-tc* (*tc-tensor* ϱ (*tc-butterfly* ψ ψ))›
using *assms* **by** (*simp add*: *kf-apply-tensor-left*)
also have ‹... = *trace-tc* $\varrho *$ *trace-tc* (*tc-butterfly* ψ ψ)›
by (*metis* (*no-types*, *lifting*) *assms norm-tc-pos norm-tc-tensor of-real-mult tc-butterfly-pos tc-tensor-pos*)
also have ‹... = *norm* $\varrho *$ *trace-tc* (*tc-butterfly* ψ ψ)›
by (*simp add*: *assms norm-tc-pos*)
also have ‹... = $\psi \cdot_C$ (*norm* $\varrho *_C$ *id-cblinfun*) ψ›
by (*simp add*: *trace-tc-butterfly*)
finally show ‹$\psi \cdot_C$ (*kf-bound* (*kf-tensor-left* ϱ)) $\psi = \psi \cdot_C$ (*norm* $\varrho *_C$ *id-cblinfun*) ψ›
by −
qed

lemma *kf-norm-tensor-right*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-norm* (*kf-tensor-right* ϱ) = *norm* ϱ›
using *assms* **by** (*simp add*: *kf-norm-def*)
lemma *kf-norm-tensor-left*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*kf-norm* (*kf-tensor-left* ϱ) = *norm* ϱ›
using *assms* **by** (*simp add*: *kf-norm-def*)

lemma *kf-trace-preserving-tensor*:
assumes ‹*kf-trace-preserving* $\mathfrak{E}$› **and** ‹*kf-trace-preserving* $\mathfrak{F}$›
shows ‹*kf-trace-preserving* (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$)›
by (*metis assms*(*1*,*2*) *kf-bound-tensor kf-trace-preserving-iff-bound-id tensor-id*)

lemma *kf-trace-reducing-tensor*:
assumes ‹*kf-trace-reducing* $\mathfrak{E}$› **and** ‹*kf-trace-reducing* $\mathfrak{F}$›
shows ‹*kf-trace-reducing* (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$)›
using *assms*
by (*auto intro*!: *mult-le-one simp*: *kf-trace-reducing-iff-norm-leq1 kf-norm-tensor kf-norm-geq0*)

lemma *kf-tensor-map-left*:
‹*kf-tensor* (*kf-map* f $\mathfrak{E}$) $\mathfrak{F} \equiv_{kr}$ *kf-map* (*apfst* f) (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$)›
proof (*rule kf-eqI-from-filter-eq-weak*)

fix *xy* :: ‹*'e* × *'f*›
obtain *x y* **where** *xy*: ‹*xy* = (*x*,*y*)›
apply *atomize-elim*
by (*auto intro*!: *prod.exhaust*)
have ‹*kf-filter* ((=) *xy*) (*kf-tensor* (*kf-map f* 𝔈) 𝔉) = *kf-tensor* (*kf-filter* ((=)*x*) (*kf-map f* 𝔈)) (*kf-filter* ((=)*y*) 𝔉)›
by (*auto intro*!: *arg-cong2*[**where** *f*=*kf-filter*] *simp add*: *xy simp flip*: *kf-filter-tensor*)
also have ‹... = *kf-tensor* (*kf-map f* (*kf-filter* (λ*x'*. *x* = *f x'*) 𝔈)) (*kf-filter* ((=) *y*) 𝔉)›
by (*simp add*: *kf-filter-map*)
also have ‹... =$_{kr}$ *kf-tensor* (*kf-filter* (λ*x'*. *x* = *f x'*) 𝔈) (*kf-filter* ((=) *y*) 𝔉)›
apply (*rule kf-tensor-cong-weak*)
by (*simp-all add*: *kf-eq-weak-def*)
also have ‹... =$_{kr}$ *kf-filter* (λ(*x'*,*y'*). *x* = *f x'* ∧ *y* = *y'*) (*kf-tensor* 𝔈 𝔉)›
by (*auto intro*!: *arg-cong2*[**where** *f*=*kf-filter*] *simp add*: *xy simp flip*: *kf-filter-tensor*)
also have ‹... =$_{kr}$ *kf-map* (*apfst f*) (*kf-filter* (λ(*x'*,*y'*). *x* = *f x'* ∧ *y* = *y'*) (*kf-tensor* 𝔈 𝔉))›
by (*simp add*: *kf-eq-weak-def*)
also have ‹... = *kf-filter* ((=) *xy*) (*kf-map* (*apfst f*) (*kf-tensor* 𝔈 𝔉))›
by (*auto intro*!: *arg-cong2*[**where** *f*=*kf-map*] *arg-cong2*[**where** *f*=*kf-filter*] *simp add*: *xy kf-filter-map simp flip*:)
finally show ‹*kf-filter* ((=) *xy*) (*kf-tensor* (*kf-map f* 𝔈) 𝔉) =$_{kr}$ ...›
by −
qed

lemma *kf-tensor-map-right*:
‹*kf-tensor* 𝔈 (*kf-map f* 𝔉) ≡$_{kr}$ *kf-map* (*apsnd f*) (*kf-tensor* 𝔈 𝔉)›
proof (*rule kf-eqI-from-filter-eq-weak*)
fix *xy* :: ‹*'e* × *'f*›
obtain *x y* **where** *xy*: ‹*xy* = (*x*,*y*)›
apply *atomize-elim*
by (*auto intro*!: *prod.exhaust*)
have ‹*kf-filter* ((=) *xy*) (*kf-tensor* 𝔈 (*kf-map f* 𝔉)) = *kf-tensor* (*kf-filter* ((=)*x*) 𝔈) (*kf-filter* ((=)*y*) (*kf-map f* 𝔉))›
by (*auto intro*!: *arg-cong2*[**where** *f*=*kf-filter*] *simp add*: *xy simp flip*: *kf-filter-tensor*)
also have ‹... = *kf-tensor* (*kf-filter* ((=)*x*) 𝔈) (*kf-map f* (*kf-filter* (λ*y'*. *y* = *f y'*) 𝔉))›
by (*simp add*: *kf-filter-map*)
also have ‹... =$_{kr}$ *kf-tensor* (*kf-filter* ((=)*x*) 𝔈) (*kf-filter* (λ*y'*. *y* = *f y'*) 𝔉)›
apply (*rule kf-tensor-cong-weak*)
by (*simp-all add*: *kf-eq-weak-def*)
also have ‹... =$_{kr}$ *kf-filter* (λ(*x'*,*y'*). *x* = *x'* ∧ *y* = *f y'*) (*kf-tensor* 𝔈 𝔉)›
by (*auto intro*!: *arg-cong2*[**where** *f*=*kf-filter*] *simp add*: *xy simp flip*: *kf-filter-tensor*)
also have ‹... =$_{kr}$ *kf-map* (*apsnd f*) (*kf-filter* (λ(*x'*,*y'*). *x* = *x'* ∧ *y* = *f y'*) (*kf-tensor* 𝔈 𝔉))›
by (*simp add*: *kf-eq-weak-def*)
also have ‹... = *kf-filter* ((=) *xy*) (*kf-map* (*apsnd f*) (*kf-tensor* 𝔈 𝔉))›
by (*auto intro*!: *arg-cong2*[**where** *f*=*kf-map*] *arg-cong2*[**where** *f*=*kf-filter*] *simp add*: *xy kf-filter-map simp flip*:)
finally show ‹*kf-filter* ((=) *xy*) (*kf-tensor* 𝔈 (*kf-map f* 𝔉)) =$_{kr}$ ...›
by −
qed

lemma *kf-tensor-map-both*:
‹*kf-tensor* (*kf-map f* $\mathfrak{E}$) (*kf-map g* $\mathfrak{F}$) $\equiv_{kr}$ *kf-map* (*map-prod f g*) (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$)›
apply (*rule kf-tensor-map-left*[*THEN kf-eq-trans*])
apply (*rule kf-map-cong*[*THEN kf-eq-trans*, *OF refl*])
apply (*rule kf-tensor-map-right*)
apply (*rule kf-map-twice*[*THEN kf-eq-trans*])
by (*simp add*: *o-def map-prod-def case-prod-unfold*)

lemma *kf-tensor-raw-map-inj-both*:
‹*kf-tensor-raw* (*kf-map-inj f* $\mathfrak{E}$) (*kf-map-inj g* $\mathfrak{F}$) = *kf-map-inj* (λ(*E*,*F*,*x*,*y*). (*E*,*F*,*f x*,*g y*)) (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$)›
apply (*transfer′ fixing*: *f g*)
by *force*

lemma *kf-domain-tensor-raw-subset*:
‹*kf-domain* (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$) ⊆ *kf-operators* $\mathfrak{E}$ × *kf-operators* $\mathfrak{F}$ × *kf-domain* $\mathfrak{E}$ × *kf-domain* $\mathfrak{F}$›
apply *transfer′*
by *force*

lemma *kf-tensor-map-inj-both*:
assumes ‹*inj-on f* (*kf-domain* $\mathfrak{E}$)›
assumes ‹*inj-on g* (*kf-domain* $\mathfrak{F}$)›
shows ‹*kf-tensor* (*kf-map-inj f* $\mathfrak{E}$) (*kf-map-inj g* $\mathfrak{F}$) = *kf-map-inj* (*map-prod f g*) (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$)›
proof −
from *assms*
have *inj1*: ‹*inj-on* (λ(*E*, *F*, *x*, *y*). (*E*, *F*, *f x*, *g y*)) (*kf-domain* (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$))›
using *kf-domain-tensor-raw-subset*[*of* $\mathfrak{E}$ $\mathfrak{F}$]
apply (*simp add*: *inj-on-def ball-def split*!: *prod.split*)
by *blast*+
from *assms*
have *inj2*: ‹*inj-on* (*map-prod f g*) ((λ(*E*, *F*, *x*). *x*) ‘ *kf-domain* (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$))›
using *kf-domain-tensor-raw-subset*[*of* $\mathfrak{E}$ $\mathfrak{F}$]
apply (*simp add*: *inj-on-def ball-def split*!: *prod.split*)
by *blast*+
have ‹*kf-tensor* (*kf-map-inj f* $\mathfrak{E}$) (*kf-map-inj g* $\mathfrak{F}$) = *kf-map* (λ(*E*, *F*, *y*). *y*) (*kf-map-inj* (λ(*E*, *F*, *x*, *y*). (*E*, *F*, *f x*, *g y*)) (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$))›
by (*simp add*: *kf-tensor-def kf-tensor-raw-map-inj-both id-def*)
also have ‹. . . = *kf-map* (λ(*E*, *F*, *x*, *y*). (*f x*, *g y*)) (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$)›
apply (*subst kf-map-kf-map-inj-comp*)
apply (*rule inj1*)
by (*simp add*: *case-prod-unfold o-def*)
also have ‹. . . = *kf-map-inj* (*map-prod f g*) (*kf-map* (λ(*E*, *F*, *x*). *x*) (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$))›
apply (*subst kf-map-inj-kf-map-comp*)
apply (*rule inj2*)
by (*simp add*: *o-def case-prod-unfold map-prod-def*)
also have ‹. . . = *kf-map-inj* (*map-prod f g*) (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$)›
by (*simp add*: *kf-tensor-def kf-tensor-raw-map-inj-both id-def*)

finally show *?thesis*
by −
qed

lemma *kf-operators-tensor-raw*:
shows ‹*kf-operators* (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$) = {$E \otimes_o F$ | E F. $E \in$ *kf-operators* $\mathfrak{E} \wedge F \in$ *kf-operators* $\mathfrak{F}$}›
apply (*simp add*: *kf-operators.rep-eq kf-tensor-raw.rep-eq*)
by (*force simp*: *case-prod-unfold*)

lemma *kf-operators-tensor*:
shows ‹*kf-operators* (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$) ⊆ *span* {$E \otimes_o F$ | E F. $E \in$ *kf-operators* $\mathfrak{E} \wedge F \in$ *kf-operators* $\mathfrak{F}$}›
proof −
have ‹*kf-operators* (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$) ⊆ *span* (*kf-operators* (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$))›
by (*simp add*: *kf-operators-kf-map kf-tensor-def*)
also have ‹... = *span* {$E \otimes_o F$ | E F. $E \in$ *kf-operators* $\mathfrak{E} \wedge F \in$ *kf-operators* $\mathfrak{F}$}›
by (*metis kf-operators-tensor-raw*)
finally show *?thesis*
by −
qed

lemma *kf-domain-tensor*: ‹*kf-domain* (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$) = *kf-domain* $\mathfrak{E}$ × *kf-domain* $\mathfrak{F}$›
proof (*intro Set.set-eqI iffI*)
fix *xy* **assume** *xy-dom*: ‹*xy* ∈ *kf-domain* (*kf-tensor* $\mathfrak{E}$ $\mathfrak{F}$)›
obtain *x y* **where** *xy*: ‹*xy* = (*x*,*y*)›
by (*auto simp*: *prod-eq-iff*)
from *xy-dom* **obtain** *E F* **where** ‹(*E*,*F*,*x*,*y*) ∈ *kf-domain* (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$)›
by (*force simp add*: *kf-tensor-def xy*)
then obtain *EF* **where** *EFEFxy*: ‹(*EF*,*E*,*F*,*x*,*y*) ∈ *Rep-kraus-family* (*kf-tensor-raw* $\mathfrak{E}$ $\mathfrak{F}$)›
by (*auto simp*: *kf-domain-def*)
then have ‹*EF* = *E* ⊗$_o$ *F*›
by (*force simp*: *kf-tensor-raw.rep-eq case-prod-unfold*)
from *EFEFxy* **have** ‹*EF* ≠ *0*›
using *Rep-kraus-family*
by (*force simp*: *kraus-family-def*)
with ‹*EF* = *E* ⊗$_o$ *F*› **have** ‹*E* ≠ *0*› **and** ‹*F* ≠ *0*›
by *fastforce+*
from *EFEFxy* **have** ‹(*E*,*x*) ∈ *Rep-kraus-family* $\mathfrak{E}$›
apply (*transfer′ fixing*: *E x*)
by *auto*
with ‹*E* ≠ *0*› **have** ‹*x* ∈ *kf-domain* $\mathfrak{E}$›
apply (*transfer′ fixing*: *E x*)
by *force*
from *EFEFxy* **have** ‹(*F*,*y*) ∈ *Rep-kraus-family* $\mathfrak{F}$›
apply (*transfer′ fixing*: *F y*)
by *auto*
with ‹*F* ≠ *0*› **have** ‹*y* ∈ *kf-domain* $\mathfrak{F}$›
apply (*transfer′ fixing*: *F y*)

```
    by force
  from ‹x ∈ kf-domain 𝔈› ‹y ∈ kf-domain 𝔉›
  show ‹xy ∈ kf-domain 𝔈 × kf-domain 𝔉›
    by (simp add: xy)
next
  fix xy assume xy-dom: ‹xy ∈ kf-domain 𝔈 × kf-domain 𝔉›
  then obtain x y where xy: ‹xy = (x,y)› and xE: ‹x ∈ kf-domain 𝔈› and yF: ‹y ∈ kf-domain
𝔉›
    by (auto simp: prod-eq-iff)
  from xE obtain E where Ex: ‹(E,x) ∈ Rep-kraus-family 𝔈›
    by (auto simp: kf-domain-def)
  from yF obtain F where Fy: ‹(F,y) ∈ Rep-kraus-family 𝔉›
    by (auto simp: kf-domain-def)
  from Ex Fy have ‹(E ⊗o F, E, F, x, y) ∈ Rep-kraus-family (kf-tensor-raw 𝔈 𝔉)›
    by (force simp: kf-tensor-raw.rep-eq case-prod-unfold)
  moreover
  have ‹E ≠ 0› and ‹F ≠ 0›
    using Ex Fy Rep-kraus-family
    by (force simp: kraus-family-def)+
  then have ‹E ⊗o F ≠ 0›
    by (simp add: tensor-op-nonzero)
  ultimately have ‹(E, F, x, y) ∈ kf-domain (kf-tensor-raw 𝔈 𝔉)›
    by (force simp: kf-domain-def)
  then show ‹xy ∈ kf-domain (kf-tensor 𝔈 𝔉)›
    by (force simp: kf-tensor-def xy case-prod-unfold)
qed

lemma kf-tensor-raw-0-left[simp]: ‹kf-tensor-raw 0 𝔈 = 0›
  apply transfer'
  by simp

lemma kf-tensor-raw-0-right[simp]: ‹kf-tensor-raw 𝔈 0 = 0›
  apply transfer'
  by simp

lemma kf-tensor-0-left[simp]: ‹kf-tensor 0 𝔈 = 0›
  by (simp add: kf-tensor-def)

lemma kf-tensor-0-right[simp]: ‹kf-tensor 𝔈 0 = 0›
  by (simp add: kf-tensor-def)

lemma kf-tensor-of-op:
  ‹kf-tensor (kf-of-op A) (kf-of-op B) = kf-map (λ(). ((),())) (kf-of-op (A ⊗o B))›
proof −
  wlog Aneq0: ‹A ≠ 0›
    using negation
    by simp
  wlog Bneq0: ‹B ≠ 0› keeping Aneq0
    using negation
```

by *simp*
have [*simp*]: ‹(λ(). ((), ())) = *Pair* ()›
by *auto*
have ‹*kf-tensor-raw* (*kf-of-op A*) (*kf-of-op B*) = *kf-map-inj* (λ(). (*A*, *B*, (), ())) (*kf-of-op* (*A* ⊗$_o$ *B*))›
apply (*transfer′ fixing*: *A B*)
by (*simp add*: *case-unit-Unity tensor-op-nonzero*)
then show *?thesis*
by (*simp add*: *kf-tensor-def kf-map-kf-map-inj-comp inj-on-def o-def case-unit-Unity*)
qed

2.15 Partial trace

definition *kf-partial-trace-right* :: ‹(('*a*×'*b*) *ell2*, '*a ell2*, '*b*) *kraus-family*› **where**
‹*kf-partial-trace-right* = *kf-map* (λ((-,*b*),-). *inv ket b*)
(*kf-comp* (*kf-of-op* (*tensor-ell2-right* (*ket* ())*))
(*kf-tensor kf-id* (*kf-trace* (*range ket*))))›

definition *kf-partial-trace-left* :: ‹(('*a*×'*b*) *ell2*, '*b ell2*, '*a*) *kraus-family*› **where**
‹*kf-partial-trace-left* = *kf-map-inj snd* (*kf-comp kf-partial-trace-right* (*kf-of-op swap-ell2*))›

lemma *partial-trace-is-kf-partial-trace*:
fixes *t* :: ‹(('*a* × '*b*) *ell2*, ('*a* × '*b*) *ell2*) *trace-class*›
shows ‹*partial-trace t* = *kf-partial-trace-right* ∗$_{kr}$ *t*›
proof −
have ‹*partial-trace t* = *kf-apply* (*kf-of-op* (*tensor-ell2-right* (*ket* ())*))
(*kf-apply* (*kf-tensor kf-id* (*kf-trace* (*range ket*))) *t*)›
proof (*rule eq-from-separatingI2x*[**where** *x*=*t*, *OF separating-set-bounded-clinear-tc-tensor*])
show ‹*bounded-clinear partial-trace*›
by *simp*
show ‹*bounded-clinear*
(λ*t*. *kf-apply* (*kf-of-op* (*tensor-ell2-right* (*ket* ())*))
(*kf-apply* (*kf-tensor kf-id* (*kf-trace* (*range ket*))) *t*))›
by (*intro bounded-linear-intros*)
fix ϱ :: ‹('*a ell2*, '*a ell2*) *trace-class*› **and** σ :: ‹('*b ell2*, '*b ell2*) *trace-class*›
have ‹*trace* (*from-trace-class* σ) ∗$_C$ *from-trace-class* ϱ =
tensor-ell2-right (*ket* ())* o_{CL} *from-trace-class* ϱ ⊗$_o$ *of-complex* (*trace* (*from-trace-class* σ)) o_{CL} *tensor-ell2-right* (*ket* ())›
by (*auto intro*!: *cblinfun-eqI simp*: *tensor-op-ell2 ket-CARD-1-is-1*)
then show ‹*partial-trace* (*tc-tensor* ϱ σ) =
kf-apply (*kf-of-op* (*tensor-ell2-right* (*ket* ())*))
(*kf-apply* (*kf-tensor kf-id* (*kf-trace* (*range ket*))) (*tc-tensor* ϱ σ))›
by (*auto intro*!: *from-trace-class-inject*[*THEN iffD1*]
simp: *partial-trace-tensor kf-apply-tensor kf-trace-is-trace kf-of-op-apply*
from-trace-class-sandwich-tc sandwich-apply trace-tc.rep-eq tc-tensor.rep-eq scaleC-trace-class.rep-eq)
qed
then show *?thesis*
by (*simp add*: *kf-partial-trace-right-def kf-comp-apply*)
qed

lemma *partial-trace-ignores-kraus-family*:
assumes ‹*kf-trace-preserving* $\mathfrak{E}$›
shows ‹*partial-trace* (*kf-tensor* $\mathfrak{F}$ $\mathfrak{E}$ $*_{kr}$ ϱ) = $\mathfrak{F}$ $*_{kr}$ *partial-trace* ϱ›
proof (*rule eq-from-separatingI2x*[**where** *x*=ϱ, *OF separating-set-bounded-clinear-tc-tensor*])
show ‹*bounded-clinear* (λ*a*. *partial-trace* (*kf-tensor* $\mathfrak{F}$ $\mathfrak{E}$ $*_{kr}$ *a*))›
by (*intro bounded-linear-intros*)
show ‹*bounded-clinear* (λ*a*. $\mathfrak{F}$ $*_{kr}$ *partial-trace a*)›
by (*intro bounded-linear-intros*)
fix ϱ :: ‹('*e ell2*, '*e ell2*) *trace-class*› **and** σ :: ‹('*a ell2*, '*a ell2*) *trace-class*›
from *assms*
show ‹*partial-trace* (*kf-tensor* $\mathfrak{F}$ $\mathfrak{E}$ $*_{kr}$ *tc-tensor* ϱ σ) =
$\mathfrak{F}$ $*_{kr}$ *partial-trace* (*tc-tensor* ϱ σ)›
by (*simp add*: *kf-apply-tensor partial-trace-tensor kf-trace-preserving-def kf-apply-scaleC*)
qed

lemma *kf-partial-trace-bound*[*simp*]:
shows ‹*kf-bound kf-partial-trace-right* = *id-cblinfun*›
by (*simp add*: *kf-partial-trace-right-def kf-map-bound*
unitary-tensor-ell2-right-CARD-1 kf-bound-comp-iso kf-bound-tensor
kf-trace-bound)

lemma *kf-partial-trace-norm*[*simp*]:
shows ‹*kf-norm kf-partial-trace-right* = *1*›
by (*simp add*: *kf-norm-def*)

lemma *kf-partial-trace-right-apply-singleton*:
‹*kf-partial-trace-right* $*_{kr}$ @{*x*} ϱ = *sandwich-tc* (*tensor-ell2-right* (*ket x*)*) ϱ›
proof −
have ‹*kf-partial-trace-right* $*_{kr}$ @{*x*} (*tc-tensor* (*tc-butterfly* (*ket a*) (*ket b*)) (*tc-butterfly* (*ket c*) (*ket d*))) =
sandwich-tc (*tensor-ell2-right* (*ket x*)*) (*tc-tensor* (*tc-butterfly* (*ket a*) (*ket b*)) (*tc-butterfly* (*ket c*) (*ket d*)))› **for** *a b* :: '*a* **and** *c d* :: '*b*
proof −
have *aux1*: ‹(λ*xa*. (*case xa of* (*x*, *xa*) ⇒ (*case x of* (*uu*-, *b*) ⇒ λ-. *inv ket b*) *xa*) ∈ {*x*}) = (λ(*e*,*f*). *True* ∧ *inv ket* (*snd e*) = *x*)›
by *auto*
have *aux2*: ‹(λ*e*. *inv ket* (*snd e*) = *x*) = (λ(*a*,*b*). *True* ∧ *inv ket b* = *x*)›
by *auto*
have ‹*kf-partial-trace-right* $*_{kr}$ @{*x*} (*tc-tensor* (*tc-butterfly* (*ket a*) (*ket b*)) (*tc-butterfly* (*ket c*) (*ket d*))) =
sandwich-tc (*tensor-ell2-right* (*ket* ())*)
(*tc-tensor* (*tc-butterfly* (*ket a*) (*ket b*)) (*kf-apply* (*kf-filter* (λ*b*. *inv ket b* = *x*) (*kf-trace* (*range ket*))) (*tc-butterfly* (*ket c*) (*ket d*))))›
by (*auto simp only*: *kf-apply-on-def kf-partial-trace-right-def*
kf-filter-map aux1 kf-filter-comp kf-of-op-apply
kf-filter-true kf-filter-tensor aux2 kf-apply-map
kf-comp-apply o-def kf-apply-tensor kf-id-apply)
also have ‹... = *sandwich-tc* (*tensor-ell2-right* (*ket* ())*) (*tc-tensor* (*tc-butterfly* (*ket a*) (*ket*

```
b)) (of-bool (x=c ∧ x=d) *R tc-butterfly (ket ()) (ket ()))›
    proof (rule arg-cong[where f=‹λx. sandwich-tc - (tc-tensor - x)›])
      have ‹kf-apply (kf-filter (λb. inv ket b = x) (kf-trace (range ket))) (tc-butterfly (ket c) (ket
d))
          = sandwich-tc (vector-to-cblinfun (ket x)*) (tc-butterfly (ket c) (ket d))›
        apply (transfer' fixing: x)
        apply (subst infsum-single[where i=‹((vector-to-cblinfun (ket x))*, ket x)›])
        by auto
      also have ‹... = of-bool (x=c ∧ x=d) *R tc-butterfly (ket ()) (ket ())›
        by (auto simp add: sandwich-tc-butterfly ket-CARD-1-is-1 cinner-ket)
      finally show ‹kf-apply (kf-filter (λb. inv ket b = x) (kf-trace (range ket))) (tc-butterfly (ket
c) (ket d))
          = of-bool (x=c ∧ x=d) *R tc-butterfly (ket ()) (ket ())›
        by −
    qed
    also have ‹... = sandwich-tc (tensor-ell2-right (ket x)*) (tc-tensor (tc-butterfly (ket a) (ket
b)) (tc-butterfly (ket c) (ket d)))›
      by (auto simp: tensor-tc-butterfly sandwich-tc-butterfly)
    finally show ?thesis
      by −
  qed
  then show ?thesis
    apply (rule-tac eq-from-separatingI2x[where x=ϱ])
      apply (rule separating-set-bounded-clinear-tc-tensor-nested)
       apply (rule separating-set-tc-butterfly-nested)
        apply (rule separating-set-ket)
       apply (rule separating-set-ket)
      apply (rule separating-set-tc-butterfly-nested)
       apply (rule separating-set-ket)
      apply (rule separating-set-ket)
   by (auto intro!: kf-apply-on-bounded-clinear bounded-clinear-sandwich-tc separating-set-tc-butterfly-nested
simp: )
qed

lemma kf-partial-trace-left-apply-singleton:
  ‹kf-partial-trace-left *kr @{x} ϱ = sandwich-tc (tensor-ell2-left (ket x)*) ϱ›
proof −
  have aux1: ‹(λxa. snd xa = x) = (λ(e,f). f=x ∧ True)›
    by auto
  have aux2: ‹(λxa. xa ∈ {x}) = (λxa. xa = x)›
    by auto
  have inj-snd: ‹inj-on (snd :: unit×'b ⇒ 'b) X› for X
    by (auto intro!: inj-onI)
  have aux3: ‹tensor-ell2-right (ket x)* *V ket (b, a) = of-bool (x=a) *R ket b› for x a :: 'x
and b :: 'y
    by (smt (verit) cinner-ket-same of-bool-eq(1) of-bool-eq(2) of-real-1 of-real-hom.hom-0-iff
orthogonal-ket scaleR-scaleC tensor-ell2-ket tensor-ell2-right-adj-apply)
  have aux4: ‹tensor-ell2-left (ket x)* *V ket (a, b) = of-bool (x=a) *R ket b› for x a :: 'x and
b :: 'y
```

by (*smt* (*verit*, *del-insts*) *cinner-ket-same of-bool-eq*(*1*) *of-bool-eq*(*2*) *of-real-1 of-real-hom.hom-0-iff orthogonal-ket scaleR-scaleC tensor-ell2-ket tensor-ell2-left-adj-apply*)
have *aux5*: ‹*tensor-ell2-right* (*ket x*)∗ o_{CL} *swap-ell2* = *tensor-ell2-left* (*ket x*)∗›
apply (*rule equal-ket*)
by (*auto intro*!: *simp*: *aux3 aux4*)
have ‹*kf-partial-trace-left* $*_{kr}$ @{*x*} *ϱ*
= *kf-partial-trace-right* $*_{kr}$ @{*x*} (*sandwich-tc swap-ell2 ϱ*)›
by (*simp only*: *kf-partial-trace-left-def kf-apply-on-def kf-filter-map-inj*
aux1 kf-filter-comp kf-apply-map-inj inj-snd kf-filter-true
kf-comp-apply o-def kf-of-op-apply aux2)
also have ‹... = *sandwich-tc* (*tensor-ell2-left* (*ket x*)∗) *ϱ*›
by (*auto intro*!: *arg-cong*[**where** *f*=‹λ*x*. *sandwich-tc x* -›]
simp: *kf-partial-trace-right-apply-singleton aux5*
simp flip: *sandwich-tc-compose*[*unfolded o-def*, *THEN fun-cong*])
finally show *?thesis*
by −
qed

lemma *kf-domain-partial-trace-right*[*simp*]: ‹*kf-domain kf-partial-trace-right* = *UNIV*›
proof (*intro Set.set-eqI iffI UNIV-I*)
fix *x* :: *'a* **and** *y* :: *'b*

have ‹*kf-partial-trace-right* $*_{kr}$ @{*x*} (*tc-tensor* (*tc-butterfly* (*ket y*) (*ket y*)) (*tc-butterfly* (*ket x*) (*ket x*)))
= *tc-butterfly* (*ket y*) (*ket y*)›
by (*simp add*: *kf-partial-trace-right-apply-singleton tensor-tc-butterfly sandwich-tc-butterfly*)
also have ‹... ≠ *0*›
proof −
have ‹*norm* (*tc-butterfly* (*ket y*) (*ket y*)) = *1*›
by (*simp add*: *norm-tc-butterfly*)
then show *?thesis*
by *auto*
qed
finally have ‹*kf-apply-on* (*kf-partial-trace-right* :: ((*'b*×*'a*) *ell2*, *'b ell2*, *'a*) *kraus-family*) {*x*} ≠ *0*›
by *auto*
then show ‹*x* ∈ *kf-domain* (*kf-partial-trace-right* :: ((*'b*×*'a*) *ell2*, *'b ell2*, *'a*) *kraus-family*)›
by (*rule in-kf-domain-iff-apply-nonzero*[*THEN iffD2*])
qed

lemma *kf-domain-partial-trace-left*[*simp*]: ‹*kf-domain kf-partial-trace-left* = *UNIV*›
proof (*intro Set.set-eqI iffI UNIV-I*)
fix *x* :: *'a* **and** *y* :: *'b*

have ‹*kf-partial-trace-left* $*_{kr}$ @{*x*} (*tc-tensor* (*tc-butterfly* (*ket x*) (*ket x*)) (*tc-butterfly* (*ket y*) (*ket y*)))
= *tc-butterfly* (*ket y*) (*ket y*)›
by (*simp add*: *kf-partial-trace-left-apply-singleton tensor-tc-butterfly sandwich-tc-butterfly*)

also have ‹... ≠ 0›
proof −
have ‹*norm* (*tc-butterfly* (*ket y*) (*ket y*)) = 1›
by (*simp add*: *norm-tc-butterfly*)
then show *?thesis*
by *auto*
qed
finally have ‹*kf-apply-on* (*kf-partial-trace-left* :: (('*a*×'*b*) *ell2*, '*b ell2*, '*a*) *kraus-family*) {*x*} ≠ 0›
by *auto*
then show ‹*x* ∈ *kf-domain* (*kf-partial-trace-left* :: (('*a*×'*b*) *ell2*, '*b ell2*, '*a*) *kraus-family*)›
by (*rule in-kf-domain-iff-apply-nonzero*[*THEN iffD2*])
qed

2.16 Complete measurement

lemma *complete-measurement-aux*:
fixes *B* **and** *F* :: ‹('*a*::*chilbert-space* ⇒$_{CL}$ '*a* × '*a*) *set*›
defines ‹*family* ≡ (λ*x*. (*selfbutter* (*sgn x*), *x*)) ‘ *B*›
assumes ‹*finite F*› **and** *FB*: ‹*F* ⊆ *family*› **and** ‹*is-ortho-set B*›
shows ‹(∑ (*E*, *x*)∈*F*. *E*∗ *o*$_{CL}$ *E*) ≤ *id-cblinfun*›
proof −
obtain *G* **where** ‹*finite G*› **and** ‹*G* ⊆ *B*› **and** *FG*: ‹*F* = (λ*x*. (*selfbutter* (*sgn x*), *x*)) ‘ *G*›
apply *atomize-elim*
using ‹*finite F*› **and** *FB*
apply (*simp add*: *family-def*)
by (*meson finite-subset-image*)
from ‹*G* ⊆ *B*› **have** [*simp*]: ‹*is-ortho-set G*›
by (*simp add*: ‹*is-ortho-set B*› *is-ortho-set-antimono*)
then have [*simp*]: ‹*e* ∈ *G* ⟹ *norm* (*sgn e*) = 1› **for** *e*
apply (*simp add*: *is-ortho-set-def*)
by (*metis norm-sgn*)
have [*simp*]: ‹*inj-on* (λ*x*. (*selfbutter* (*sgn x*), *x*)) *G*›
by (*meson inj-onI prod.inject*)
have [*simp*]: ‹*inj-on sgn G*›
proof (*rule inj-onI*, *rule ccontr*)
fix *x y* **assume** ‹*x* ∈ *G*› **and** ‹*y* ∈ *G*› **and** *sgn-eq*: ‹*sgn x* = *sgn y*› **and** ‹*x* ≠ *y*›
with ‹*is-ortho-set G*› **have** ‹*is-orthogonal x y*›
by (*meson is-ortho-set-def*)
then have ‹*is-orthogonal* (*sgn x*) (*sgn y*)›
by *fastforce*
with *sgn-eq* **have** ‹*sgn x* = 0›
by *force*
with ‹*x* ∈ *G*› ‹*is-ortho-set G*› **show** *False*
by (*metis* ‹*x* ≠ *y*› *local.sgn-eq sgn-zero-iff*)
qed
have ‹(∑ (*E*, *x*)∈*F*. *E*∗ *o*$_{CL}$ *E*) = (∑ *x*∈*G*. *selfbutter* (*sgn x*))›
by (*simp add*: *FG sum.reindex cdot-square-norm*)
also

```
  have ‹(∑ x∈G. selfbutter (sgn x)) ≤ id-cblinfun›
    apply (subst sum.reindex[where h=sgn, unfolded o-def, symmetric])
     apply simp
    apply (subgoal-tac ‹⋀x y. ∀x∈G. ∀y∈G. x ≠ y ⟶ is-orthogonal x y ⟹
          0 ∉ G ⟹ x ∈ G ⟹ y ∈ G ⟹ sgn x ≠ sgn y ⟹ is-orthogonal x y›)
    using ‹is-ortho-set G›
     apply (auto intro!: sum-butterfly-leq-id simp: is-ortho-set-def sgn-zero-iff)[1]
    by fast
  finally show ?thesis
    by −
qed

lemma complete-measurement-is-kraus-family:
  assumes ‹is-ortho-set B›
  shows ‹kraus-family ((λx. (selfbutter (sgn x), x)) ‘ B)›
proof (rule kraus-familyI)
  show ‹bdd-above (sum (λ(E, x). E* o_CL E) ‘ {F. finite F ∧ F ⊆ (λx. (selfbutter (sgn x), x))
‘ B})›
    using complete-measurement-aux[OF - - assms]
    by (auto intro!: bdd-aboveI[where M=id-cblinfun] kraus-familyI)
  have ‹selfbutter (sgn x) = 0 ⟹ x = 0› for x
    by (smt (verit, best) mult-cancel-right1 norm-butterfly norm-sgn norm-zero)
  then show ‹0 ∉ fst ‘ (λx. (selfbutter (sgn x), x)) ‘ B›
    using assms by (force simp: is-ortho-set-def)
qed

lift-definition kf-complete-measurement :: ‹'a set ⇒ ('a::chilbert-space, 'a, 'a) kraus-family› is
  ‹λB. if is-ortho-set B then (λx. (selfbutter (sgn x), x)) ‘ B else {}›
  by (auto intro!: complete-measurement-is-kraus-family)

definition kf-complete-measurement-ket :: ‹('a ell2, 'a ell2, 'a) kraus-family› where
  ‹kf-complete-measurement-ket = kf-map-inj (inv ket) (kf-complete-measurement (range ket))›

lemma kf-complete-measurement-domain[simp]:
  assumes ‹is-ortho-set B›
  shows ‹kf-domain (kf-complete-measurement B) = B›
  apply (transfer fixing: B)
  using assms by (auto simp: image-image)

lemma kf-complete-measurement-ket-domain[simp]:
  ‹kf-domain kf-complete-measurement-ket = UNIV›
  by (simp add: kf-complete-measurement-ket-def)

lemma kf-complete-measurement-ket-kf-map:
  ‹kf-complete-measurement-ket ≡_kr kf-map (inv ket) (kf-complete-measurement (range ket))›
  unfolding kf-complete-measurement-ket-def
  apply (rule kf-map-inj-eq-kf-map)
  using inj-on-inv-into by fastforce+
```

lemma *kf-bound-complete-measurement*:
assumes ‹*is-ortho-set B*›
shows ‹*kf-bound* (*kf-complete-measurement B*) ≤ *id-cblinfun*›
apply (*rule kf-bound-leqI*)
by (*simp add*: *assms complete-measurement-aux kf-complete-measurement.rep-eq*)

lemma *kf-norm-complete-measurement*:
assumes ‹*is-ortho-set B*›
shows ‹*kf-norm* (*kf-complete-measurement B*) ≤ *1*›
by (*smt* (*verit*, *ccfv-SIG*) *assms kf-norm-def kf-bound-complete-measurement kf-bound-pos norm-cblinfun-id-le norm-cblinfun-mono*)

lemma *kf-complete-measurement-invalid*:
assumes ‹¬ *is-ortho-set B*›
shows ‹*kf-complete-measurement B* = *0*›
apply (*transfer′ fixing*: *B*)
using *assms* **by** *simp*

lemma *kf-complete-measurement-idem*:
‹*kf-comp* (*kf-complete-measurement B*) (*kf-complete-measurement B*)
≡$_{kr}$ *kf-map* (λ*b*. (*b*,*b*)) (*kf-complete-measurement B*)›
proof −
wlog [*iff*]: ‹*is-ortho-set B*›
using *negation*
by (*simp add*: *kf-complete-measurement-invalid*)
define *f* **where** ‹*f b* = (*selfbutter* (*sgn b*), *selfbutter* (*sgn b*), *b*, *b*)› **for** *b* :: *′a*
have *b2*: ‹*sgn b* ·$_C$ *sgn b* = *1*› **if** ‹*b* ∈ *B*› **for** *b*
by (*metis* ‹*b* ∈ *B*› ‹*is-ortho-set B*› *cnorm-eq-1 is-ortho-set-def norm-sgn*)
have *1*: ‹(*E* *o*$_{CL}$ *F*, *F*, *E*, *b*, *c*) ∈ (λ*x*. (*selfbutter* (*sgn x*), *f x*)) ‘ *B*›
if ‹*E* *o*$_{CL}$ *F* ≠ *0*› **and** ‹(*F*, *b*) ∈ *Rep-kraus-family* (*kf-complete-measurement B*)› **and** ‹(*E*, *c*) ∈ *Rep-kraus-family* (*kf-complete-measurement B*)›
for *E F b c*
proof −
from *that* **have** ‹*b* ∈ *B*› **and** ‹*c* ∈ *B*› **and** *E-def*: ‹*E* = *selfbutter* (*sgn c*)› **and** *F-def*: ‹*F* = *selfbutter* (*sgn b*)›
by (*auto simp add*: *kf-complete-measurement.rep-eq*)
have ‹*E* *o*$_{CL}$ *F* = (*sgn c* ·$_C$ *sgn b*) ∗$_C$ *butterfly* (*sgn c*) (*sgn b*)›
by (*simp add*: *E-def F-def*)
with *that* **have** ‹*c* ·$_C$ *b* ≠ *0*›
by *fastforce*
with ‹*b* ∈ *B*› ‹*c* ∈ *B*› ‹*is-ortho-set B*›
have ‹*c* = *b*›
using *is-ortho-setD* **by** *blast*
then have ‹(*E* *o*$_{CL}$ *F*, *F*, *E*, *b*, *c*) = (λ*x*. (*selfbutter* (*sgn x*), *f x*)) *c*›
by (*simp add*: *b2* ‹*b* ∈ *B*› *f-def E-def F-def*)
with ‹*c* ∈ *B*› **show** *?thesis*
by *blast*

qed
have *2*: ‹$x \in B \Longrightarrow$ *selfbutter* (*sgn x*) $\neq$ *0*› **for** *x*
by (*smt* (*verit*) ‹*is-ortho-set B*› *inverse-1 is-ortho-set-def norm-butterfly norm-sgn norm-zero right-inverse*)
have *3*: ‹(*selfbutter* (*sgn x*), *f x*) ∈ (λ((*F*,*y*),(*E*, *x*)). (*E* o_{CL} *F*, *F*, *E*, *y*, *x*)) ‘
(*Rep-kraus-family* (*kf-complete-measurement B*) × *Rep-kraus-family* (*kf-complete-measurement B*))›
if ‹*x* ∈ *B*› **and** ‹(*E*, *F*, *c*, *b*) = *f x*›
for *E F c b x*
apply (*rule image-eqI*[**where** *x*=‹((*selfbutter* (*sgn x*),*x*),(*selfbutter* (*sgn x*),*x*))›])
by (*auto intro*!: *simp*: *b2 that f-def kf-complete-measurement.rep-eq*)
have *raw*: ‹(*kf-comp-dependent-raw* (λ-. *kf-complete-measurement B*) (*kf-complete-measurement B*))
= *kf-map-inj f* (*kf-complete-measurement B*)›
apply (*transfer' fixing*: *f B*)
using *1 2 3*
by (*auto simp*: *image-image case-prod-unfold*)
have ‹*kf-comp* (*kf-complete-measurement B*) (*kf-complete-measurement B*)
≡$_{kr}$ *kf-map* (λ(*F*, *E*, *y*). *y*) ((*kf-comp-dependent-raw* (λ-. *kf-complete-measurement B*) (*kf-complete-measurement B*)))›
by (*simp add*: *kf-comp-def kf-comp-dependent-def id-def*)
also have ‹... ≡$_{kr}$ *kf-map* (λ(*F*, *E*, *y*). *y*) (*kf-map-inj f* (*kf-complete-measurement B*))›
by (*simp add*: *raw*)
also have ‹... ≡$_{kr}$ *kf-map* (λ(*F*, *E*, *y*). *y*) (*kf-map f* (*kf-complete-measurement B*))›
by (*auto intro*!: *kf-map-cong kf-map-inj-eq-kf-map inj-onI simp*: *f-def*)
also have ‹... ≡$_{kr}$ *kf-map* (λ*b*. (*b*, *b*)) (*kf-complete-measurement B*)›
apply (*rule kf-map-twice*[*THEN kf-eq-trans*])
by (*simp add*: *f-def o-def*)
finally show *?thesis*
by −
qed

lemma *kf-complete-measurement-idem-weak*:
‹*kf-comp* (*kf-complete-measurement B*) (*kf-complete-measurement B*)
=$_{kr}$ *kf-complete-measurement B*›
by (*metis* (*no-types*, *lifting*) *kf-apply-map kf-complete-measurement-idem kf-eq-imp-eq-weak kf-eq-weak-def*)

lemma *kf-complete-measurement-ket-idem*:
‹*kf-comp kf-complete-measurement-ket kf-complete-measurement-ket*
≡$_{kr}$ *kf-map* (λ*b*. (*b*,*b*)) *kf-complete-measurement-ket*›
proof −
have ‹*kf-comp kf-complete-measurement-ket kf-complete-measurement-ket*
≡$_{kr}$ *kf-comp* (*kf-map* (*inv ket*) (*kf-complete-measurement* (*range ket*))) (*kf-map* (*inv ket*) (*kf-complete-measurement* (*range ket*)))›
by (*intro kf-comp-cong kf-complete-measurement-ket-kf-map*)
also have ‹... ≡$_{kr}$ *kf-map* (λ(*x*, *y*). (*x*, *inv ket y*))
(*kf-map* (λ(*x*, *y*). (*inv ket x*, *y*)) (*kf-comp* (*kf-complete-measurement* (*range ket*)) (*kf-complete-measurement* (*range ket*))))›

by (*intro kf-comp-map-left*[*THEN kf-eq-trans*] *kf-map-cong kf-comp-map-right refl*)
also have ‹... $\equiv_{kr}$ *kf-map* (λ(*x*, *y*). (*x*, *inv ket y*))
(*kf-map* (λ(*x*, *y*). (*inv ket x*, *y*)) (*kf-map* (λ*b*. (*b*, *b*)) (*kf-complete-measurement* (*range ket*))))›
by (*intro kf-map-cong kf-complete-measurement-idem refl*)
also have ‹... $\equiv_{kr}$ *kf-map* (λ*x*. (*inv ket x*, *inv ket x*)) (*kf-complete-measurement* (*range ket*))›
apply (*intro kf-map-twice*[*THEN kf-eq-trans*])
by (*simp add*: *o-def*)
also have ‹... $\equiv_{kr}$ *kf-map* (λ*b*. (*b*, *b*)) (*kf-map* (*inv ket*) (*kf-complete-measurement* (*range ket*)))›
apply (*rule kf-eq-sym*)
apply (*rule kf-map-twice*[*THEN kf-eq-trans*])
by (*simp add*: *o-def*)
also have ‹... $\equiv_{kr}$ *kf-map* (λ*b*. (*b*, *b*)) *kf-complete-measurement-ket*›
using *kf-complete-measurement-ket-kf-map kf-eq-sym kf-map-cong* **by** *fastforce*
finally show *?thesis*
by −
qed

lemma *kf-complete-measurement-ket-idem-weak*:
‹*kf-comp kf-complete-measurement-ket kf-complete-measurement-ket*
$=_{kr}$ *kf-complete-measurement-ket*›
by (*metis* (*no-types*, *lifting*) *kf-apply-map kf-complete-measurement-ket-idem kf-eq-imp-eq-weak kf-eq-weak-def*)

lemma *kf-complete-measurement-apply*:
assumes [*simp*]: ‹*is-ortho-set B*›
shows ‹*kf-complete-measurement B* $*_{kr}$ *t* = ($\sum_\infty$*x*∈*B*. *sandwich-tc* (*selfbutter* (*sgn x*)) *t*)›
proof −
have ‹*kf-complete-measurement B* $*_{kr}$ *t* =
($\sum_\infty$*E*∈(λ*x*. (*selfbutter* (*sgn x*), *x*)) ‘ *B*. *sandwich-tc* (*fst E*) *t*)›
apply (*transfer' fixing*: *B t*)
by *simp*
also have ‹... = ($\sum_\infty$*x*∈*B*. *sandwich-tc* (*selfbutter* (*sgn x*)) *t*)›
apply (*subst infsum-reindex*)
by (*auto intro*!: *inj-onI simp*: *o-def*)
finally show *?thesis*
by −
qed

lemma *kf-complete-measurement-has-sum*:
assumes ‹*is-ortho-set B*›
shows ‹((λ*x*. *sandwich-tc* (*selfbutter* (*sgn x*)) *ϱ*) *has-sum kf-complete-measurement B* $*_{kr}$ *ϱ*) *B*›
using *kf-apply-has-sum*[*of* - ‹*kf-complete-measurement B*›] *assms*
by (*simp add*: *kf-complete-measurement-apply kf-complete-measurement.rep-eq*
has-sum-reindex inj-on-def o-def)

lemma *kf-complete-measurement-has-sum-onb*:
assumes ‹*is-onb B*›
shows ‹((λ*x*. *sandwich-tc* (*selfbutter x*) ϱ) *has-sum kf-complete-measurement B* $*_{kr}$ ϱ) *B*›
proof −
have ‹*is-ortho-set B*›
using *assms* **by** (*simp add*: *is-onb-def*)
have *sgnx*: ‹*sgn x* = *x*› **if** ‹*x* ∈ *B*› **for** *x*
using *assms that*
by (*simp add*: *is-onb-def sgn-div-norm*)
from *kf-complete-measurement-has-sum*[*OF* ‹*is-ortho-set B*›]
show *?thesis*
apply (*rule has-sum-cong*[*THEN iffD1*, *rotated*])
by (*simp add*: *sgnx*)
qed

lemma *kf-complete-measurement-ket-has-sum*:
‹((λ*x*. *sandwich-tc* (*selfbutter* (*ket x*)) ϱ) *has-sum kf-complete-measurement-ket* $*_{kr}$ ϱ) *UNIV*›
proof −
from *kf-complete-measurement-has-sum-onb*
have ‹((λ*x*. *sandwich-tc* (*selfbutter x*) ϱ) *has-sum kf-complete-measurement* (*range ket*) $*_{kr}$ ϱ) (*range ket*)›
by *force*
then have ‹((λ*x*. *sandwich-tc* (*selfbutter* (*ket x*)) ϱ) *has-sum kf-complete-measurement* (*range ket*) $*_{kr}$ ϱ) *UNIV*›
apply (*subst* (*asm*) *has-sum-reindex*)
by (*simp-all add*: *o-def*)
then have ‹((λ*x*. *sandwich-tc* (*selfbutter* (*ket x*)) ϱ) *has-sum kf-map* (*inv ket*) (*kf-complete-measurement* (*range ket*)) $*_{kr}$ ϱ) *UNIV*›
by *simp*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *kf-apply-on-UNIV kf-apply-on-eqI kf-complete-measurement-ket-kf-map*)
qed

lemma *kf-complete-measurement-apply-onb*:
assumes ‹*is-onb B*›
shows ‹*kf-complete-measurement B* $*_{kr}$ *t* = ($\sum_\infty$*x*∈*B*. *sandwich-tc* (*selfbutter x*) *t*)›
using *kf-complete-measurement-has-sum-onb*[*OF assms*]
by (*metis* (*lifting*) *infsumI*)

lemma *kf-complete-measurement-ket-apply*: ‹*kf-complete-measurement-ket* $*_{kr}$ *t* = ($\sum_\infty$*x*. *sandwich-tc* (*selfbutter* (*ket x*)) *t*)›
proof −
have ‹*kf-complete-measurement-ket* $*_{kr}$ *t* = *kf-complete-measurement* (*range ket*) $*_{kr}$ *t*›
by (*metis kf-apply-map kf-apply-on-UNIV kf-apply-on-eqI kf-complete-measurement-ket-kf-map*)
also have ‹... = ($\sum_\infty$*x*∈*range ket*. *sandwich-tc* (*selfbutter x*) *t*)›
by (*simp add*: *kf-complete-measurement-apply-onb*)
also have ‹... = ($\sum_\infty$*x*. *sandwich-tc* (*selfbutter* (*ket x*)) *t*)›
by (*simp add*: *infsum-reindex o-def*)
finally show *?thesis*

by −
qed

lemma *kf-bound-complete-measurement-onb*[*simp*]:
assumes ‹*is-onb B*›
shows ‹*kf-bound* (*kf-complete-measurement B*) = *id-cblinfun*›
proof (*rule cblinfun-eq-gen-eqI*[**where** *G*=*B*], *rule cinner-extensionality-basis*[**where** *B*=*B*])
show ‹*ccspan B* = ⊤›
using *assms is-onb-def* **by** *blast*
then show ‹*ccspan B* = ⊤›
by −
fix *x y* **assume** ‹*x* ∈ *B*› **and** ‹*y* ∈ *B*›
have *aux1*: ‹*j* ≠ *x* ⟹ *j* ∈ *B* ⟹ *sandwich-tc* (*selfbutter j*) (*tc-butterfly y x*) = *0*› **for** *j*
apply (*transfer′ fixing*: *x B j y*)
by (*smt* (*z3*) ‹*x* ∈ *B*› *apply-id-cblinfun assms butterfly-0-right butterfly-adjoint butterfly-def cblinfun-apply-cblinfun-compose*
cblinfun-comp-butterfly is-onb-def is-ortho-setD of-complex-eq-id sandwich-apply vector-to-cblinfun-adj-apply)
have *aux2*: ‹*trace-tc* (*sandwich-tc* (*selfbutter y*) (*tc-butterfly y y*)) = *1*›
apply (*transfer′ fixing*: *y*)
by (*metis* (*no-types*, *lifting*) *ext* ‹*y* ∈ *B*› *assms butterfly-adjoint butterfly-comp-butterfly cblinfun-comp-butterfly cinner-simps*(*6*)
is-onb-def norm-one one-cinner-a-scaleC-one one-cinner-one sandwich-apply selfbutter-pos trace-butterfly trace-norm-butterfly
trace-norm-pos trace-scaleC)
have *aux3*: ‹*x* ≠ *y* ⟹ *trace-tc* (*sandwich-tc* (*selfbutter x*) (*tc-butterfly y x*)) = *0*›
apply (*transfer′ fixing*: *x y*)
by (*metis* (*no-types*, *lifting*) *ext Trace-Class.trace-0* ‹*x* ∈ *B*› ‹*y* ∈ *B*› *apply-id-cblinfun assms butterfly-0-left butterfly-def*
cblinfun.zero-right cblinfun-apply-cblinfun-compose cblinfun-comp-butterfly is-onb-def is-ortho-setD of-complex-eq-id
sandwich-apply vector-to-cblinfun-adj-apply)

have ‹*x* •$_C$ (*kf-bound* (*kf-complete-measurement B*) *∗$_V$ y*) = *trace-tc* (*kf-complete-measurement B ∗$_{kr}$ tc-butterfly y x*)›
by (*simp add*: *kf-bound-from-map*)
also have ‹... = *trace-tc* (∑$_\infty$*xa*∈*B*. *sandwich-tc* (*selfbutter xa*) (*tc-butterfly y x*))›
by (*simp add*: *kf-complete-measurement-apply-onb assms*)
also have ‹... = *trace-tc* (*if x* ∈ *B then sandwich-tc* (*selfbutter x*) (*tc-butterfly y x*) *else 0*)›
apply (*subst infsum-single*[**where** *i*=*x*])
using *aux1* **by** *auto*
also have ‹... = *of-bool* (*x* = *y*)›
using ‹*y* ∈ *B*› *aux2 aux3* **by** (*auto intro*!: *simp*:)
also have ‹... = *x* •$_C$ (*id-cblinfun ∗$_V$ y*)›
using ‹*x* ∈ *B*› ‹*y* ∈ *B*› *assms cnorm-eq-1 is-onb-def is-ortho-setD* **by** *fastforce*
finally show ‹*x* •$_C$ (*kf-bound* (*kf-complete-measurement B*) *∗$_V$ y*) = *x* •$_C$ (*id-cblinfun ∗$_V$ y*)›
by −
qed

lemma *kf-bound-complete-measurement-ket*[*simp*]:
‹*kf-bound kf-complete-measurement-ket = id-cblinfun*›
by (*metis is-onb-ket kf-bound-complete-measurement-onb kf-bound-cong kf-complete-measurement-ket-kf-map kf-eq-imp-eq-weak*
kf-map-bound)

lemma *kf-norm-complete-measurement-onb*[*simp*]:
fixes *B* :: ‹'*a*::{*not-singleton*, *chilbert-space*} *set*›
assumes ‹*is-onb B*›
shows ‹*kf-norm* (*kf-complete-measurement B*) = *1*›
by (*simp add*: *kf-norm-def assms*)

lemma *kf-norm-complete-measurement-ket*[*simp*]:
‹*kf-norm kf-complete-measurement-ket = 1*›
by (*simp add*: *kf-norm-def*)

lemma *kf-complete-measurement-ket-diagonal-operator*[*simp*]:
‹*kf-complete-measurement-ket* $*_{kr}$ *diagonal-operator-tc f = diagonal-operator-tc f*›
proof (*cases* ‹*f abs-summable-on UNIV*›)
case *True*
have ‹*kf-complete-measurement-ket* $*_{kr}$ *diagonal-operator-tc f* = ($\sum_\infty$*x*. *sandwich-tc* (*selfbutter* (*ket x*)) (*diagonal-operator-tc f*))›
by (*simp add*: *kf-complete-measurement-ket-apply*)
also have ‹... = ($\sum_\infty$*x*. *sandwich-tc* (*selfbutter* (*ket x*)) ($\sum_\infty$*y*. *f y* $*_C$ *tc-butterfly* (*ket y*) (*ket y*)))›
by (*simp add*: *flip*: *tc-butterfly-scaleC-infsum*)
also have ‹... = ($\sum_\infty$*x*. $\sum_\infty$*y*. *sandwich-tc* (*selfbutter* (*ket x*)) (*f y* $*_C$ *tc-butterfly* (*ket y*) (*ket y*)))›
apply (*rule infsum-cong*)
apply (*rule infsum-bounded-linear*[*unfolded o-def*, *symmetric*])
by (*auto intro*!: *bounded-clinear.bounded-linear bounded-clinear-sandwich-tc tc-butterfly-scaleC-summable True*)
also have ‹... = ($\sum_\infty$*x*. $\sum_\infty$*y*. *of-bool* (*y*=*x*) $*_C$ *f x* $*_C$ *tc-butterfly* (*ket x*) (*ket x*))›
apply (*rule infsum-cong*)+
apply (*transfer*' *fixing*: *f*)
by (*simp add*: *sandwich-apply*)
also have ‹... = ($\sum_\infty$*x*. *f x* $*_C$ *tc-butterfly* (*ket x*) (*ket x*))›
apply (*subst infsum-of-bool-scaleC*)
by *simp*
also have ‹... = *diagonal-operator-tc f*›
by (*simp add*: *flip*: *tc-butterfly-scaleC-infsum*)
finally show *?thesis*
by −
next
case *False*
then have ‹*diagonal-operator-tc f = 0*›
by (*rule diagonal-operator-tc-invalid*)

then show *?thesis*
by *simp*
qed

lemma *kf-operators-complete-measurement*:
‹*kf-operators* (*kf-complete-measurement B*) = (*selfbutter o sgn*) ‘ *B*› **if** ‹*is-ortho-set B*›
apply (*transfer′ fixing*: *B*)
using *that* **by** *force*

lemma *kf-operators-complete-measurement-invalid*:
‹*kf-operators* (*kf-complete-measurement B*) = {}› **if** ‹¬ *is-ortho-set B*›
apply (*transfer′ fixing*: *B*)
using *that* **by** *force*

lemma *kf-operators-complete-measurement-ket*:
‹*kf-operators kf-complete-measurement-ket* = *range* (λ*c*. *butterfly* (*ket c*) (*ket c*))›
by (*simp add*: *kf-complete-measurement-ket-def kf-operators-complete-measurement image-image*)

lemma *kf-complete-measurement-apply-butterfly*:
assumes ‹*is-ortho-set B*› **and** ‹*b* ∈ *B*›
shows ‹*kf-complete-measurement B* ∗$_{kr}$ *tc-butterfly b b* = *tc-butterfly b b*›
proof −
have ‹*kf-complete-measurement B* ∗$_{kr}$ *tc-butterfly b b* = (∑$_\infty$*x*∈*B*. *sandwich-tc* (*selfbutter* (*sgn x*)) (*tc-butterfly b b*))›
by (*simp add*: *kf-complete-measurement-apply assms*)
also have ‹... = (*if b* ∈ *B then sandwich-tc* (*selfbutter* (*sgn b*)) (*tc-butterfly b b*) *else 0*)›
proof (*rule infsum-single*[**where** *i*=*b*])
fix *c* **assume** ‹*c* ∈ *B*› **and** ‹*c* ≠ *b*›
then have [*iff*]: ‹*c* •$_C$ *b* = *0*›
using *assms*(*1*,*2*) *is-ortho-setD* **by** *blast*
then have [*iff*]: ‹*sgn c* •$_C$ *b* = *0*›
by *auto*
then
show ‹*sandwich-tc* (*selfbutter* (*sgn c*)) (*tc-butterfly b b*) = *0*›
by (*auto intro*!: *simp*: *sandwich-tc-butterfly*)
qed
also have ‹... = *tc-butterfly b b*›
by (*simp add*: ‹*b* ∈ *B*› *sandwich-tc-butterfly*)
finally show *?thesis*
by −
qed

lemma *kf-complete-measurement-ket-apply-butterfly*:
‹*kf-complete-measurement-ket* ∗$_{kr}$ *tc-butterfly* (*ket x*) (*ket x*) = *tc-butterfly* (*ket x*) (*ket x*)›
by (*simp add*: *kf-complete-measurement-ket-def kf-apply-map-inj inj-on-def kf-complete-measurement-apply-butterfly*

lemma *kf-map-eq-kf-map-inj-singleton*:
assumes ‹*card-le-1* (*Rep-kraus-family* $\mathfrak{E}$)›
shows ‹*kf-map f* $\mathfrak{E}$ = *kf-map-inj f* $\mathfrak{E}$›
proof (*cases* ‹$\mathfrak{E}$ = *0*›)
case *True*
then show *?thesis*
by *simp*
next
case *False*
then have ‹*Rep-kraus-family* $\mathfrak{E}$ ≠ {}›
using *Rep-kraus-family-inject zero-kraus-family.rep-eq* **by** *auto*
with *assms* **obtain** *x* **where** *Rep*$\mathfrak{E}$: ‹*Rep-kraus-family* $\mathfrak{E}$ = {*x*}›
by (*meson card-le-1-def subset-singleton-iff*)
obtain *E y* **where** *Ey*: ‹*x* = (*E*,*y*)›
by *force*
have ‹(*E*,*y*) ∈ *Rep-kraus-family* $\mathfrak{E}$›
using *Ey Rep*$\mathfrak{E}$ **by** *force*
then have [*iff*]: ‹*E* ≠ *0*›
using *Rep-kraus-family*[*of* $\mathfrak{E}$]
by (*force simp*: *kraus-family-def*)
have ∗: ‹{(*F*, *xa*). (*F*, *xa*) = *x* ∧ *f xa* = *f y* ∧ (∃ *r*>*0*. *E* = *r* ∗$_R$ *F*)} = {(*E*,*y*)}›
apply (*subgoal-tac* ‹∃ *r*>*0*. *E* = *r* ∗$_R$ *E*›)
apply (*auto intro*!: *simp*: *Ey*)[*1*]
by (*metis scaleR-simps*(*12*) *verit-comp-simplify*(*28*))
have *1*: ‹(*norm E*)^2 = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *f y*) $\mathfrak{E}$) *E*›
by (*auto simp add*: *kf-element-weight-def kf-similar-elements-def kf-filter.rep-eq* ∗ *Rep*$\mathfrak{E}$)
have *2*: ‹*z* = *f y*› **if** ‹(*norm F*)^2 = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *z*) $\mathfrak{E}$) *F*› **and** ‹*F* ≠ *0*› **for** *F z*
proof (*rule ccontr*)
assume ‹*z* ≠ *f y*›
with *Rep*$\mathfrak{E}$ **have** ‹*kf-filter* (λ*x*. *f x* = *z*) $\mathfrak{E}$ = *0*›
apply (*transfer*′ *fixing*: *f z y x*)
by (*auto simp*: *Ey*)
then have ‹*kf-element-weight* (*kf-filter* (λ*x*. *f x* = *z*) $\mathfrak{E}$) *F* = *0*›
by *simp*
with *that* **show** *False*
by *fastforce*
qed
have *3*: ‹*F* = *E*› **if** ‹(*norm F*)^2 = *kf-element-weight* (*kf-filter* (λ*x*. *f x* = *z*) $\mathfrak{E}$) *F*› **and** ‹*F* ≠ *0*› **and** ‹*z* = *f y*› **for** *F z*
proof −
from *that Rep*$\mathfrak{E}$ **have** *f*$\mathfrak{E}$: ‹*kf-filter* (λ*x*. *f x* = *z*) $\mathfrak{E}$ = $\mathfrak{E}$›
apply (*transfer*′ *fixing*: *F f z y x*)
using *Ey Rep-kraus-family-inject kf-filter.rep-eq* **by** *fastforce*
have ‹∃ *r*>*0*. *F* = *r* ∗$_R$ *E*›
proof (*rule ccontr*)
assume ‹¬ (∃ *r*>*0*. *F* = *r* ∗$_R$ *E*)›

```
    with Rep𝔈 have ‹kf-similar-elements 𝔈 F = {}›
      by (simp add: kf-similar-elements-def Ey)
    with that f𝔈 show False
      by (simp add: kf-element-weight-def)
  qed
  then obtain r where FrE: ‹F = r *R E› and rpos: ‹r > 0›
    by auto
  with Rep𝔈 have ‹kf-similar-elements 𝔈 F = {(E,y)}›
    by (force simp: kf-similar-elements-def Ey)
  then have ‹kf-element-weight 𝔈 F = (norm E)²›
    by (simp add: kf-element-weight-def)
  with that have ‹norm E = norm F›
    using f𝔈 by force
  with FrE rpos have ‹r = 1›
    by simp
  with FrE show ‹F = E›
    by simp
 qed
 show ?thesis
  apply (rule Rep-kraus-family-inject[THEN iffD1])
  by (auto intro!: 1 2 3 simp: kf-map.rep-eq kf-map-inj.rep-eq Rep𝔈 Ey)
qed

lemma kf-map-eq-kf-map-inj-singleton':
 assumes ‹⋀y. card-le-1 (Rep-kraus-family (kf-filter ((=)y) 𝔈))›
 assumes ‹inj-on f (kf-domain 𝔈)›
 shows ‹kf-map f 𝔈 = kf-map-inj f 𝔈›
proof −
 have 1: ‹card-le-1 (Rep-kraus-family (kf-filter (λz. x = f z) 𝔈))› for x
 proof (cases ‹x ∈ f ‘ kf-domain 𝔈›)
  case True
  then have ‹kf-filter (λz. x = f z) 𝔈 = kf-filter ((=) (inv-into (kf-domain 𝔈) f x)) 𝔈›
    apply (rule-tac kf-filter-cong-eq)
    using assms(2)
    by auto
  with assms(1) show ?thesis
    by presburger
 next
  case False
  then have ‹kf-filter (λz. x = f z) 𝔈 = 0›
    by (simp add: image-iff)
  then show ?thesis
    by (simp add: zero-kraus-family.rep-eq)
 qed
 show ?thesis
  apply (rule kf-equal-if-filter-equal)
  unfolding kf-filter-map kf-filter-map-inj
  apply (rule kf-map-eq-kf-map-inj-singleton)
  by (rule 1)
```

qed

lemma *kf-filter-singleton-kf-complete-measurement*:
assumes ‹$x \in B$› **and** ‹*is-ortho-set* B›
shows ‹*kf-filter* ((=)x) (*kf-complete-measurement* B) = *kf-map-inj* (λ-. x) (*kf-of-op* (*selfbutter* (*sgn* x)))›
proof −
from *assms* **have** [*iff*]: ‹*selfbutter* (*sgn* x) ≠ 0›
by (*smt* (*verit*, *best*) *inverse-1 is-ortho-set-def norm-butterfly norm-sgn norm-zero right-inverse*)
show *?thesis*
apply (*transfer′ fixing*: x B)
by (*auto intro*!: *simp*: *assms*)
qed

lemma *kf-filter-singleton-kf-complete-measurement′*:
assumes ‹$x \in B$› **and** ‹*is-ortho-set* B›
shows ‹*kf-filter* ((=)x) (*kf-complete-measurement* B) = *kf-map* (λ-. x) (*kf-of-op* (*selfbutter* (*sgn* x)))›
using *kf-filter-singleton-kf-complete-measurement*[*OF assms*]
apply (*subst kf-map-eq-kf-map-inj-singleton*)
by (*auto simp*: *kf-of-op.rep-eq*)

lemma *kf-filter-disjoint*:
assumes ‹$\bigwedge x.\ x \in$ *kf-domain* $\mathfrak{E} \Longrightarrow P\ x =$ *False*›
shows ‹*kf-filter* P $\mathfrak{E} = 0$›
using *assms*
apply (*transfer′ fixing*: P)
by *fastforce*

lemma *kf-complete-measurement-tensor*:
assumes ‹*is-ortho-set* B› **and** ‹*is-ortho-set* C›
shows ‹*kf-map* (λ(*b*,*c*). $b \otimes_s c$) (*kf-tensor* (*kf-complete-measurement* B) (*kf-complete-measurement* C))
= *kf-complete-measurement* ((λ(*b*,*c*). $b \otimes_s c$) ‘ ($B \times C$))›
proof (*rule kf-equal-if-filter-equal*, *rename-tac bc*)
fix *bc* :: ‹('*a* × '*b*) *ell2*›
define *BC* **where** ‹BC = ((λ(x, y). $x \otimes_s y$) ‘ ($B \times C$))›
consider (*bc-tensor*) b c **where** ‹$b \in B$› ‹$c \in C$› ‹$bc = b \otimes_s c$›
| (*not-bc-tensor*) ‹¬ (∃ *b*∈B. ∃ *c*∈C. $bc = b \otimes_s c$)›
by *fastforce*
then show ‹*kf-filter* ((=) *bc*) (*kf-map* (λ(*b*, *c*). $b \otimes_s c$) (*kf-tensor* (*kf-complete-measurement* B) (*kf-complete-measurement* C)))
= *kf-filter* ((=) *bc*) (*kf-complete-measurement* ((λ(*b*, *c*). $b \otimes_s c$) ‘ ($B \times C$)))›
proof *cases*
case *bc-tensor*
have *uniq*: ‹$b = b'$› ‹$c = c'$› **if** ‹$b' \in B$› **and** ‹$c' \in C$› **and** ‹$b \otimes_s c = b' \otimes_s c'$› **for** b' c'
proof −

from *bc-tensor* **have** ‹$b \neq 0$›
using *assms*(1) *is-ortho-set-def* **by** *blast*
with *that*(3) **obtain** γ **where** *upto*: ‹$c = \gamma *_C c'$›
apply *atomize-elim*
by (*rule tensor-ell2-almost-injective*)
from *bc-tensor* **have** ‹$c \neq 0$›
using *assms*(2) *is-ortho-set-def* **by** *blast*
with *upto* **have** ‹$\gamma \neq 0$›
by *auto*
with ‹$c \neq 0$› *upto* **have** ‹$c \cdot_C c' \neq 0$›
by *simp*
with ‹$c \in C$› ‹$c' \in C$› ‹*is-ortho-set C*›
show ‹$c = c'$›
using *is-ortho-setD* **by** *blast*
with *that* **have** ‹$b \otimes_s c = b' \otimes_s c$›
by *simp*
with ‹$c \neq 0$› ‹$b \in B$› ‹$b' \in B$› ‹*is-ortho-set B*› **show** ‹$b = b'$›
by (*metis cinner-eq-zero-iff is-ortho-setD nonzero-mult-div-cancel-right tensor-ell2-inner-prod*)
qed

have [*iff*]: ‹*selfbutter* (*sgn b*) ≠ *0*›
by (*smt* (*verit*, *ccfv-SIG*) *bc-tensor assms inverse-1 is-ortho-set-def norm-butterfly norm-sgn norm-zero right-inverse*)
have [*iff*]: ‹*selfbutter* (*sgn c*) ≠ *0*›
by (*smt* (*verit*) *bc-tensor assms inverse-1 is-ortho-set-def norm-butterfly norm-sgn norm-zero right-inverse*)
have [*iff*]: ‹*selfbutter* (*sgn bc*) ≠ *0*›
by (*smt* (*verit*, *del-insts*) ‹*selfbutter* (*sgn b*) ≠ *0*› ‹*selfbutter* (*sgn c*) ≠ *0*› *bc-tensor*(3) *butterfly-0-right mult-cancel-right1 norm-butterfly norm-sgn sgn-zero*
tensor-ell2-nonzero)
have ‹*kf-filter* ((=) *bc*) (*kf-map* (λ(*b*, *c*). *b* ⊗*s* *c*) (*kf-tensor* (*kf-complete-measurement B*) (*kf-complete-measurement C*)))
= *kf-map* (λ(*x*, *y*). *x* ⊗*s* *y*) (*kf-filter* (λ*x*. *bc* = (*case x of* (*b*, *c*) ⇒ *b* ⊗*s* *c*))
(*kf-tensor* (*kf-complete-measurement B*) (*kf-complete-measurement C*)))›
by (*simp add*: *kf-filter-map*)
also have ‹... = *kf-map* (λ(*x*, *y*). *x* ⊗*s* *y*) (*kf-filter* ((=)(*b*,*c*))
(*kf-tensor* (*kf-complete-measurement B*) (*kf-complete-measurement C*)))›
apply (*rule arg-cong*[**where** *f*=‹*kf-map* -›])
apply (*rule kf-filter-cong-eq*[*OF refl*])
by (*auto intro*!: *uniq simp add*: *kf-domain-tensor kf-complete-measurement-domain assms case-prod-beta bc-tensor split*!: *prod.split*)
also have ‹... = *kf-map* (λ(*x*, *y*). *x* ⊗*s* *y*) (*kf-tensor* (*kf-filter* ((=) *b*) (*kf-complete-measurement B*))
(*kf-filter* ((=) *c*) (*kf-complete-measurement C*)))›
by (*simp add*: *kf-filter-tensor-singleton*)
also have ‹... = *kf-map* (λ(*x*, *y*). *x* ⊗*s* *y*) (*kf-map* (λ(*E*, *F*, *y*). *y*) (*kf-tensor-raw* (*kf-filter* ((=) *b*) (*kf-complete-measurement B*))

```
            (kf-filter ((=) c) (kf-complete-measurement C))))›
    by (simp add: kf-tensor-def id-def)
  also have ‹... = kf-map (λ(x, y). x ⊗s y) (kf-map (λ(E, F, y). y) (kf-tensor-raw (kf-map
(λ-. b) (kf-of-op (selfbutter (sgn b))))
            (kf-map (λ-. c) (kf-of-op (selfbutter (sgn c))))))›
    by (simp add: kf-filter-singleton-kf-complete-measurement' bc-tensor assms)
  also have ‹... = kf-map-inj (λ(x, y). x ⊗s y) (kf-map-inj (λ(E, F, y). y) (kf-tensor-raw
(kf-map-inj (λ-. b) (kf-of-op (selfbutter (sgn b))))
            (kf-map-inj (λ-. c) (kf-of-op (selfbutter (sgn c))))))›
    apply (subst (4) kf-map-eq-kf-map-inj-singleton)
     apply (simp add: kf-of-op.rep-eq)
    apply (subst (3) kf-map-eq-kf-map-inj-singleton)
     apply (simp add: kf-of-op.rep-eq)
    apply (subst (2) kf-map-eq-kf-map-inj-singleton)
     apply (simp add: kf-of-op.rep-eq kf-map-inj.rep-eq kf-tensor-raw.rep-eq)
    apply (subst (1) kf-map-eq-kf-map-inj-singleton)
     apply (simp add: kf-of-op.rep-eq kf-map-inj.rep-eq kf-tensor-raw.rep-eq)
    by blast
  also have ‹... = kf-map-inj (λ-. b ⊗s c) (kf-of-op (selfbutter (sgn bc)))›
    apply (transfer' fixing: b c bc)
    by (auto intro!: bc-tensor simp: tensor-butterfly simp flip: sgn-tensor-ell2 bc-tensor)
  also have ‹... = kf-map (λ-. bc) (kf-of-op (selfbutter (sgn bc)))›
    apply (subst kf-map-eq-kf-map-inj-singleton)
    by (auto intro!: bc-tensor simp: kf-of-op.rep-eq kf-map-inj.rep-eq kf-tensor-raw.rep-eq simp
flip: bc-tensor)
  also have ‹... = kf-filter ((=) bc) (kf-complete-measurement ((λ(b, c). b ⊗s c) ' (B × C)))›
    apply (subst kf-filter-singleton-kf-complete-measurement')
    by (auto simp: is-ortho-set-tensor bc-tensor assms)
  finally show ?thesis
    by −
 next
  case not-bc-tensor
  have ortho: ‹is-ortho-set ((λx. case x of (x, xa) ⇒ x ⊗s xa) ' (B × C))›
    by (metis is-ortho-set-tensor not-bc-tensor assms)
  show ?thesis
    apply (subst kf-filter-disjoint)
    using not-bc-tensor
     apply (simp add: kf-domain-tensor kf-complete-measurement-domain assms)
     apply fastforce
    apply (subst kf-filter-disjoint)
    using not-bc-tensor
     apply (simp add: kf-domain-tensor kf-complete-measurement-domain ortho)
     apply fastforce
    by simp
 qed
qed

lemma card-le-1-kf-filter: ‹card-le-1 (Rep-kraus-family (kf-filter P 𝔈))› if ‹card-le-1 (Rep-kraus-family
𝔈)›
```

by (*metis* (*no-types*, *lifting*) *card-le-1-def filter-insert-if kf-filter.rep-eq kf-filter-0-right subset-singleton-iff that zero-kraus-family.rep-eq*)

lemma *card-le-1-kf-map-inj*[*iff*]: ‹*card-le-1* (*Rep-kraus-family* (*kf-map-inj f* 𝔈))› **if** ‹*card-le-1* (*Rep-kraus-family* 𝔈)›
using *that*
apply *transfer'*
by (*auto simp*: *card-le-1-def case-prod-unfold*)

lemma *card-le-1-kf-map*[*iff*]: ‹*card-le-1* (*Rep-kraus-family* (*kf-map f* 𝔈))› **if** ‹*card-le-1* (*Rep-kraus-family* 𝔈)›
using *kf-map-eq-kf-map-inj-singleton*[*OF that*] *card-le-1-kf-map-inj*[*OF that*]
by *metis*

lemma *card-le-1-kf-tensor-raw*[*iff*]: ‹*card-le-1* (*Rep-kraus-family* (*kf-tensor-raw* 𝔈 𝔉))› **if** ‹*card-le-1* (*Rep-kraus-family* 𝔈)› **and** ‹*card-le-1* (*Rep-kraus-family* 𝔉)›
using *that*
apply *transfer'*
apply (*simp add*: *card-le-1-def*)
by *fast*

lemma *card-le-1-kf-tensor*[*iff*]: ‹*card-le-1* (*Rep-kraus-family* (*kf-tensor* 𝔈 𝔉))› **if** ‹*card-le-1* (*Rep-kraus-family* 𝔈)› **and** ‹*card-le-1* (*Rep-kraus-family* 𝔉)›
by (*auto intro*!: *card-le-1-kf-map card-le-1-kf-tensor-raw that simp add*: *kf-tensor-def*)

lemma *card-le-1-kf-filter-complete-measurement*: ‹*card-le-1* (*Rep-kraus-family* (*kf-filter* ((=)*x*) (*kf-complete-measurement B*)))›
proof −
consider (*inB*) ‹*is-ortho-set B* ∧ *x* ∈ *B*› | (*not-ortho*) ‹¬ *is-ortho-set B*› | (*notinB*) ‹*x* ∉ *B*› ‹*is-ortho-set B*›
by *auto*
then show *?thesis*
proof *cases*
case *inB*
then have ‹*Rep-kraus-family* (*kf-filter* ((=)*x*) (*kf-complete-measurement B*)) = {(*selfbutter* (*sgn x*),*x*)}›
by (*auto simp*: *kf-filter.rep-eq kf-complete-measurement.rep-eq*)
then show *?thesis*
by (*simp add*: *card-le-1-def*)
next
case *not-ortho*
then show *?thesis*
by (*simp add*: *kf-complete-measurement-invalid zero-kraus-family.rep-eq*)
next
case *notinB*
then have ‹*kf-filter* ((=) *x*) (*kf-complete-measurement B*) = *0*›
by (*metis kf-complete-measurement-domain kf-filter-disjoint*)

```
    then show ?thesis
      by (simp add: zero-kraus-family.rep-eq)
  qed
qed

lemma kf-complete-measurement-ket-tensor:
  shows ‹kf-tensor (kf-complete-measurement-ket :: (-,-,'a) kraus-family) (kf-complete-measurement-ket
:: (-,-,'b) kraus-family)
            = kf-complete-measurement-ket›
proof −
  have 1: ‹(λ(b, c). b ⊗s c) ‘ (range ket × range ket) = range ket›
    by (auto intro!: image-eqI simp: tensor-ell2-ket case-prod-unfold)
  have 2: ‹inj-on (inv ket ∘ (λ(x, y). x ⊗s y)) (range ket × range ket)›
    by (auto intro!: inj-onI simp: tensor-ell2-ket)
  have ‹(kf-complete-measurement-ket :: (-,-,'a×'b) kraus-family)
      = kf-map-inj (inv ket) (kf-complete-measurement ((λ(b,c). b ⊗s c) ‘ (range ket × range
ket)))›
    by (simp add: 1 kf-complete-measurement-ket-def)
  also have ‹... = kf-map-inj (inv ket)
     (kf-map (λ(x, y). x ⊗s y)
      (kf-tensor (kf-complete-measurement (range ket)) (kf-complete-measurement (range ket))))›
    by (simp flip: kf-complete-measurement-tensor)
  also have ‹... = kf-map (inv ket ∘ (λ(x, y). x ⊗s y))
     (kf-tensor (kf-complete-measurement (range ket)) (kf-complete-measurement (range ket)))›
    apply (subst kf-map-inj-kf-map-comp)
    by (auto intro!: simp: inj-on-def kf-domain-tensor tensor-ell2-ket)
  also have ‹... = kf-map-inj (inv ket ∘ (λ(x, y). x ⊗s y))
     (kf-tensor (kf-complete-measurement (range ket)) (kf-complete-measurement (range ket)))›
    apply (rule kf-map-eq-kf-map-inj-singleton')
   apply (auto intro!: card-le-1-kf-filter-complete-measurement card-le-1-kf-tensor simp: kf-filter-tensor-singleton)[1]
    by (simp add: kf-domain-tensor 2)
  also have ‹... = kf-map-inj (map-prod (inv ket) (inv ket))
      (kf-tensor (kf-complete-measurement (range ket)) (kf-complete-measurement (range ket)))›
    apply (rule kf-map-inj-cong-eq)
    by (auto simp: kf-domain-tensor tensor-ell2-ket)
  also have ‹... = kf-tensor (kf-map-inj (inv ket) (kf-complete-measurement (range ket)))
                           (kf-map-inj (inv ket) (kf-complete-measurement (range ket)))›
    by (simp add: kf-tensor-map-inj-both inj-on-inv-into)
  also have ‹... = kf-tensor kf-complete-measurement-ket kf-complete-measurement-ket›
    by (simp add: kf-complete-measurement-ket-def)
  finally show ?thesis
    by simp
qed
```

2.17 Reconstruction

```
lemma kf-reconstruction-is-bounded-clinear:
  assumes ‹⋀ϱ. ((λa. sandwich-tc (f a) ϱ) has-sum 𝔈 ϱ) A›
  shows ‹bounded-clinear 𝔈›
```

```
proof −
  have linear: ‹clinear 𝔈›
  proof (rule clinearI)
    fix ϱ σ c
    have ‹((λa. sandwich-tc (f a) ϱ + sandwich-tc (f a) σ) has-sum (𝔈 ϱ + 𝔈 σ)) A›
      by (intro has-sum-add assms)
    then have ‹((λa. sandwich-tc (f a) (ϱ + σ)) has-sum (𝔈 ϱ + 𝔈 σ)) A›
      by (meson has-sum-cong sandwich-tc-plus)
    with assms[of ‹ϱ + σ›]
    show ‹𝔈 (ϱ + σ) = 𝔈 ϱ + 𝔈 σ›
      by (rule has-sum-unique)
    from assms[of ϱ]
    have ‹((λa. sandwich-tc (f a) (c *C ϱ)) has-sum c *C 𝔈 ϱ) A›
      using has-sum-scaleC-right[where A=A and s=‹𝔈 ϱ›]
      by (auto intro!: has-sum-scaleC-right simp: sandwich-tc-scaleC-right)
    with assms[of ‹c *C ϱ›]
    show ‹𝔈 (c *C ϱ) = c *C 𝔈 ϱ›
      by (rule has-sum-unique)
  qed
  have pos: ‹𝔈 ϱ ≥ 0› if ‹ϱ ≥ 0› for ϱ
    apply (rule has-sum-mono-traceclass[where f=‹λ-. 0› and g=‹(λa. sandwich-tc (f a) ϱ)›])
    using assms
    by (auto intro!: sandwich-tc-pos simp: that)
  have mono: ‹𝔈 ϱ ≤ 𝔈 σ› if ‹ϱ ≤ σ› for ϱ σ
  proof −
    have ‹𝔈 (σ − ϱ) ≥ 0›
      apply (rule pos)
      using that
      by auto
    then show ?thesis
      by (simp add: linear complex-vector.linear-diff)
  qed
  have bounded-pos: ‹∃ B≥0. ∀ ϱ≥0. norm (𝔈 ϱ) ≤ B * norm ϱ›
  proof (rule ccontr)
    assume asm: ‹¬ (∃ B≥0. ∀ ϱ≥0. norm (𝔈 ϱ) ≤ B * norm ϱ)›
    obtain ϱ0 where 𝔈-big0: ‹norm (𝔈 (ϱ0 i)) > 2^i * norm (ϱ0 i)› and ϱ0-pos: ‹ϱ0 i ≥ 0›
for i :: nat
      proof (atomize-elim, rule choice2, rule allI, rule ccontr)
        fix i
        define B :: real where ‹B = 2^i›
        have ‹B ≥ 0›
          by (simp add: B-def)
        assume ‹∄ ϱ0. B * norm ϱ0 < norm (𝔈 ϱ0) ∧ 0 ≤ ϱ0›
        then have ‹∀ ϱ≥0. norm (𝔈 ϱ) ≤ B * norm ϱ›
          by force
        with asm ‹B ≥ 0› show False
          by blast
      qed
      have ϱ0-neq0: ‹ϱ0 i ≠ 0› for i
```

using $\mathfrak{E}$-*big0*[*of i*] *linear complex-vector.linear-0* **by** *force*
define ϱ **where** ‹$\varrho\ i = \varrho 0\ i\ /_R\ norm\ (\varrho 0\ i)$› **for** i
have *ϱ-pos*: ‹$\varrho\ i \geq 0$› **for** i
by (*simp add*: *ϱ-def ϱ0-pos scaleR-nonneg-nonneg*)
have *norm-ϱ*: ‹$norm\ (\varrho\ i) = 1$› **for** i
by (*simp add*: *ϱ0-neq0 ϱ-def*)
from $\mathfrak{E}$-*big0* **have** $\mathfrak{E}$-*big*: ‹$trace\text{-}tc\ (\mathfrak{E}\ (\varrho\ i)) \geq 2\hat{}i$› **for** $i :: nat$
proof −
have ‹$trace\text{-}tc\ (\mathfrak{E}\ (\varrho\ i)) = trace\text{-}tc\ (\mathfrak{E}\ (\varrho 0\ i)\ /_R\ norm\ (\varrho 0\ i))$›
by (*simp add*: *ϱ-def linear scaleR-scaleC clinear.scaleC*
bounded-clinear-trace-tc[*THEN bounded-clinear.clinear*])
also have ‹$\ldots = norm\ (\mathfrak{E}\ (\varrho 0\ i)\ /_R\ norm\ (\varrho 0\ i))$›
using *ϱ0-pos pos*
by (*metis linordered-field-class.inverse-nonnegative-iff-nonnegative norm-ge-zero norm-tc-pos scaleR-nonneg-nonneg*)
also have ‹$\ldots = norm\ (\mathfrak{E}\ (\varrho 0\ i))\ /\ norm\ (\varrho 0\ i)$›
by (*simp add*: *divide-inverse-commute*)
also have ‹$\ldots > (2\hat{}i * norm\ (\varrho 0\ i))\ /\ norm\ (\varrho 0\ i)$› (**is** ‹- > ...›)
using $\mathfrak{E}$-*big0 ϱ0-neq0*
by (*smt* (*verit*, *best*) *complex-of-real-strict-mono-iff divide-le-eq norm-le-zero-iff*)
also have ‹$\ldots = 2\hat{}i$›
using *ϱ0-neq0* **by** *force*
finally show *?thesis*
by *simp*
qed
define $\sigma\ \tau$ **where** ‹$\sigma\ n = (\sum i{<}n.\ \varrho\ i\ /_R\ 2\hat{}i)$› **and** ‹$\tau = (\sum_\infty i.\ \varrho\ i\ /_R\ 2\hat{}i)$› **for** $n :: nat$
have ‹$(\lambda i.\ \varrho\ i\ /_R\ 2\hat{}i)\ abs\text{-}summable\text{-}on\ UNIV$›
proof (*rule infsum-tc-norm-bounded-abs-summable*)
from *ϱ-pos* **show** ‹$\varrho\ i\ /_R\ 2\hat{}i \geq 0$› **for** i
by (*simp add*: *scaleR-nonneg-nonneg*)
show ‹$norm\ (\sum i{\in}F.\ \varrho\ i\ /_R\ 2\hat{}i) \leq 2$› **if** ‹*finite F*› **for** F
proof −
from *finite-nat-bounded*[*OF that*]
obtain n **where** *i-leq-n*: ‹$i \leq n$› **if** ‹$i \in F$› **for** i
apply *atomize-elim*
by (*auto intro*!: *order.strict-implies-order simp*: *lessThan-def Ball-def simp flip*: *Ball-Collect*)
have ‹$norm\ (\sum i{\in}F.\ \varrho\ i\ /_R\ 2\hat{}i) \leq (\sum i{\in}F.\ norm\ (\varrho\ i\ /_R\ 2\hat{}i))$›
by (*simp add*: *sum-norm-le*)
also have ‹$\ldots = (\sum i{\in}F.\ (1/2)\hat{}i)$›
using *norm-ϱ*
by (*smt* (*verit*, *del-insts*) *Extra-Ordered-Fields.sign-simps*(*23*) *divide-inverse-commute linordered-field-class.inverse-nonnegative-iff-nonnegative*
norm-scaleR power-inverse power-one sum.cong zero-le-power)
also have ‹$\ldots \leq (\sum i{\leq}n.\ (1/2)\hat{}i)$›
apply (*rule sum-mono2*)
using *i-leq-n*
by *auto*
also have ‹$\ldots \leq (\sum i.\ (1/2)\hat{}i)$›

```
      apply (rule sum-le-suminf)
      by auto
    also have ‹... = 2›
      using suminf-geometric[of ‹1/2 :: real›]
      by simp
    finally show ?thesis
      by −
  qed
 qed
 then have summable: ‹(λi. ϱ i /R 2^i) summable-on UNIV›
  by (simp add: abs-summable-summable)
 have ‹trace-tc (𝔈 τ) ≥ n› for n :: nat
 proof −
  have ‹trace-tc (𝔈 τ) ≥ trace-tc (𝔈 (σ n))› (is ‹- ≥ ...›)
   by (auto intro!: trace-tc-mono mono infsum-mono-neutral-traceclass
       simp: τ-def σ-def summable ϱ-pos scaleR-nonneg-nonneg simp flip: infsum-finite)
  moreover have ‹... = (∑i<n. trace-tc (𝔈 (ϱ i)) / 2^i)›
   by (simp add: σ-def complex-vector.linear-sum linear scaleR-scaleC trace-scaleC
       bounded-clinear-trace-tc[THEN bounded-clinear.clinear] clinear.scaleC
       add.commute mult.commute divide-inverse)
  moreover have ‹... ≥ (∑i<n. 2^i / 2^i)› (is ‹- ≥ ...›)
   apply (intro sum-mono divide-right-mono)
   using 𝔈-big
   by (simp-all add: less-eq-complex-def)
  moreover have ‹... = (∑i<n. 1)›
   by fastforce
  moreover have ‹... = n›
   by simp
  ultimately show ?thesis
   by order
 qed
 then have Re: ‹Re (trace-tc (𝔈 τ)) ≥ n› for n :: nat
  using Re-mono by fastforce
 obtain n :: nat where ‹n > Re (trace-tc (𝔈 τ))›
  apply atomize-elim
  by (rule reals-Archimedean2)
 with Re show False
  by (smt (verit, ccfv-threshold))
qed
then obtain B where bounded-B: ‹norm (𝔈 ϱ) ≤ B * norm ϱ› and B-pos: ‹B ≥ 0› if ‹ϱ ≥
0› for ϱ
 by auto
have bounded: ‹norm (𝔈 ϱ) ≤ (4*B) * norm ϱ› for ϱ
proof −
 obtain ϱ1 ϱ2 ϱ3 ϱ4 where ϱ-decomp: ‹ϱ = ϱ1 − ϱ2 + i *C ϱ3 − i *C ϱ4›
  and pos: ‹ϱ1 ≥ 0› ‹ϱ2 ≥ 0› ‹ϱ3 ≥ 0› ‹ϱ4 ≥ 0›
  and norm: ‹norm ϱ1 ≤ norm ϱ› ‹norm ϱ2 ≤ norm ϱ› ‹norm ϱ3 ≤ norm ϱ› ‹norm ϱ4 ≤
norm ϱ›
  apply atomize-elim using trace-class-decomp-4pos'[of ϱ] by blast
```

```
    have ‹norm (𝔈 ϱ) ≤ norm (𝔈 ϱ1) + norm (𝔈 ϱ2) + norm (𝔈 ϱ3) + norm (𝔈 ϱ4)›
      using linear
      by (auto intro!: norm-triangle-le norm-triangle-le-diff
          simp add: ϱ-decomp kf-apply-plus-right kf-apply-minus-right
          kf-apply-scaleC complex-vector.linear-diff complex-vector.linear-add clinear.scaleC)
    also have ‹... ≤ B * norm ϱ1 + B * norm ϱ2 + B * norm ϱ3 + B * norm ϱ4›
      using pos by (auto intro!: add-mono simp add: pos bounded-B)
    also have ‹... = B * (norm ϱ1 + norm ϱ2 + norm ϱ3 + norm ϱ4)›
      by argo
    also have ‹... ≤ B * (norm ϱ + norm ϱ + norm ϱ + norm ϱ)›
      by (auto intro!: mult-left-mono add-mono pos B-pos
          simp only: norm)
    also have ‹... = (4 * B) * norm ϱ›
      by argo
    finally show ?thesis
      by −
  qed
  show ?thesis
    apply (rule bounded-clinearI[where K=‹4*B›])
     apply (simp add: complex-vector.linear-add linear)
    apply (simp add: complex-vector.linear-scale linear)
    using bounded by (metis Groups.mult-ac)
qed

lemma kf-reconstruction-is-kraus-family:
  assumes sum: ‹⋀ϱ. ((λa. sandwich-tc (f a) ϱ) has-sum 𝔈 ϱ) A›
  defines ‹F ≡ Set.filter (λ(E,-). E≠0) ((λa. (f a, a)) ‘ A)›
  shows ‹kraus-family F›
proof −
  from sum have ‹bounded-clinear 𝔈›
    by (rule kf-reconstruction-is-bounded-clinear)
  then obtain B where B: ‹norm (𝔈 ϱ) ≤ B * norm ϱ› for ϱ
    apply atomize-elim
    by (simp add: bounded-clinear-axioms-def bounded-clinear-def mult.commute)
  show ?thesis
  proof (intro kraus-familyI bdd-aboveI2)
    fix S assume ‹S ∈ {S. finite S ∧ S ⊆ F}›
    then have ‹S ⊆ F› and ‹finite S›
      by auto
    then obtain A' where ‹finite A'› and ‹A' ⊆ A› and S-A': ‹S = (λa. (f a,a)) ‘ A'›
      by (metis (no-types, lifting) F-def finite-subset-filter-image)
    show ‹(∑(E, x)∈S. E* o_CL E) ≤ B *_C id-cblinfun›
    proof (rule cblinfun-leI)
      fix h :: 'a assume ‹norm h = 1›
      have ‹h •_C ((∑(E, x)∈S. E* o_CL E) h) = h •_C (∑a∈A'. (f a)* o_CL f a) h›
        by (simp add: S-A' sum.reindex inj-on-def)
      also have ‹... = (∑a∈A'. h •_C ((f a)* o_CL f a) h)›
        apply (rule complex-vector.linear-sum)
        by (simp add: bounded-clinear.clinear bounded-clinear-cinner-right-comp)
```

```
    also have ‹... = (∑ a∈A'. trace-tc (sandwich-tc (f a) (tc-butterfly h h)))›
      by (auto intro!: sum.cong[OF refl]
          simp: trace-tc.rep-eq from-trace-class-sandwich-tc
          tc-butterfly.rep-eq cblinfun-comp-butterfly sandwich-apply trace-butterfly-comp)
    also have ‹... = trace-tc (∑ a∈A'. sandwich-tc (f a) (tc-butterfly h h))›
      apply (rule complex-vector.linear-sum[symmetric])
      using clinearI trace-tc-plus trace-tc-scaleC by blast
    also have ‹... = trace-tc (∑∞ a∈A'. sandwich-tc (f a) (tc-butterfly h h))›
      by (simp add: ‹finite A'›)
    also have ‹... ≤ trace-tc (∑∞ a∈A. (sandwich-tc (f a) (tc-butterfly h h)))›
      apply (intro trace-tc-mono infsum-mono-neutral-traceclass)
      using ‹A' ⊆ A› sum[of ‹tc-butterfly h h›]
      by (auto intro!: sandwich-tc-pos has-sum-imp-summable simp: ‹finite A'›)
    also have ‹... = trace-tc (𝔈 (tc-butterfly h h))›
      by (metis sum infsumI)
    also have ‹... = norm (𝔈 (tc-butterfly h h))›
     by (metis (no-types, lifting) infsumI infsum-nonneg-traceclass norm-tc-pos sandwich-tc-pos
sum tc-butterfly-pos)
    also from B have ‹... ≤ B * norm (tc-butterfly h h)›
      using complex-of-real-mono by blast
    also have ‹... = B›
      by (simp add: ‹norm h = 1› norm-tc-butterfly)
    also have ‹... = h •C (complex-of-real B *C id-cblinfun *V h)›
      using ‹norm h = 1› cnorm-eq-1 by auto
   finally show ‹h •C ((∑ (E, x)∈S. E* oCL E) *V h) ≤ h •C (complex-of-real B *C id-cblinfun
*V h)›
      by -
  qed
 next
  show ‹0 ∉ fst ` F›
    by (force simp: F-def)
 qed
qed

lemma kf-reconstruction:
 assumes sum: ‹⋀ϱ. ((λa. sandwich-tc (f a) ϱ) has-sum 𝔈 ϱ) A›
 defines ‹F ≡ Abs-kraus-family (Set.filter (λ(E,-). E≠0) ((λa. (f a, a)) ` A))›
 shows ‹kf-apply F = 𝔈›
proof (rule ext)
 fix ϱ :: ‹('a, 'a) trace-class›
 have Rep-F: ‹Rep-kraus-family F = (Set.filter (λ(E,-). E≠0) ((λa. (f a,a)) ` A))›
  unfolding F-def
  apply (rule Abs-kraus-family-inverse)
  by (auto intro!: kf-reconstruction-is-kraus-family[of - - 𝔈] assms simp: F-def
      simp del: Set.filter-eq)
 have ‹((λ(E,x). sandwich-tc E ϱ) has-sum kf-apply F ϱ) (Rep-kraus-family F)›
  by (auto intro!: kf-apply-has-sum)
 then have ‹((λ(E,x). sandwich-tc E ϱ) has-sum kf-apply F ϱ) ((λa. (f a,a)) ` A)›
  apply (rule has-sum-cong-neutral[THEN iffD2, rotated -1])
```

by (*auto simp*: *Rep-F*)
then have ‹((λ*a*. *sandwich-tc* (*f a*) ϱ) *has-sum kf-apply F* ϱ) *A*›
apply (*subst* (*asm*) *has-sum-reindex*)
by (*auto simp*: *inj-on-def o-def*)
with *sum* **show** ‹*kf-apply F* ϱ = 𝔈 ϱ›
by (*metis* (*no-types*, *lifting*) *infsumI*)
qed

2.18 Cleanup

unbundle *no cblinfun-syntax*
unbundle *no kraus-map-syntax*

end

3 Kraus maps

theory *Kraus-Maps*
imports *Kraus-Families*
begin

3.1 Kraus maps

unbundle *kraus-map-syntax*
unbundle *cblinfun-syntax*

definition *kraus-map* :: ‹(('*a*::*chilbert-space*,'*a*) *trace-class* ⇒ ('*b*::*chilbert-space*,'*b*) *trace-class*)
⇒ *bool*› **where**
kraus-map-def-raw: ‹*kraus-map* 𝔈 ⟷ (∃ *EE* :: ('*a*,'*b*,*unit*) *kraus-family*. 𝔈 = *kf-apply EE*)›

lemma *kraus-map-def*: ‹*kraus-map* 𝔈 ⟷ (∃ *EE* :: ('*a*::*chilbert-space*,'*b*::*chilbert-space*,'*x*) *kraus-family*.
𝔈 = *kf-apply EE*)›
— Has a more general type than the original definition
proof (*rule iffI*)
assume ‹*kraus-map* 𝔈›
then obtain *EE* :: ‹('*a*,'*b*,*unit*) *kraus-family*› **where** *EE*: ‹𝔈 = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
define *EE'* :: ‹('*a*,'*b*,'*x*) *kraus-family*› **where** ‹*EE'* = *kf-map* (λ-. *undefined*) *EE*›
have ‹*kf-apply EE'* = *kf-apply EE*›
by (*simp add*: *EE'-def kf-apply-map*)
with *EE* **show** ‹∃ *EE* :: ('*a*,'*b*,'*x*) *kraus-family*. 𝔈 = *kf-apply EE*›
by *metis*
next
assume ‹∃ *EE* :: ('*a*,'*b*,'*x*) *kraus-family*. 𝔈 = *kf-apply EE*›
then obtain *EE* :: ‹('*a*,'*b*,'*x*) *kraus-family*› **where** *EE*: ‹𝔈 = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
define *EE'* :: ‹('*a*,'*b*,*unit*) *kraus-family*› **where** ‹*EE'* = *kf-map* (λ-. ()) *EE*›
have ‹*kf-apply EE'* = *kf-apply EE*›
by (*simp add*: *EE'-def kf-apply-map*)

with *EE* **show** ‹*kraus-map* $\mathfrak{E}$›
apply (*simp add*: *kraus-map-def-raw*)
by *metis*
qed

lemma *kraus-mapI*:
assumes ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$›
shows ‹*kraus-map* $\mathfrak{E}$›
using *assms kraus-map-def* **by** *blast*

lemma *kraus-map-bounded-clinear*:
‹*bounded-clinear* $\mathfrak{E}$› **if** ‹*kraus-map* $\mathfrak{E}$›
by (*metis kf-apply-bounded-clinear kraus-map-def that*)

lemma *kraus-map-pos*:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹$\varrho \geq 0$›
shows ‹$\mathfrak{E}$ $\varrho \geq 0$›
proof −
from *assms* **obtain** $\mathfrak{E}'$ :: ‹('*a*, '*b*, *unit*) *kraus-family*› **where** $\mathfrak{E}'$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$›
using *kraus-map-def* **by** *blast*
show *?thesis*
by (*simp add*: $\mathfrak{E}'$ *assms*(*2*) *kf-apply-pos*)
qed

lemma *kraus-map-mono*:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹$\varrho \geq \tau$›
shows ‹$\mathfrak{E}$ $\varrho \geq \mathfrak{E}$ τ›
by (*metis assms kf-apply-mono-right kraus-map-def-raw*)

lemma *kraus-map-kf-apply*[*iff*]: ‹*kraus-map* (*kf-apply* $\mathfrak{E}$)›
using *kraus-map-def* **by** *blast*

definition *km-some-kraus-family* :: ‹(('*a*::*chilbert-space*, '*a*) *trace-class* ⇒ ('*b*::*chilbert-space*, '*b*) *trace-class*) ⇒ ('*a*, '*b*, *unit*) *kraus-family*› **where**
‹*km-some-kraus-family* $\mathfrak{E}$ = (*if kraus-map* $\mathfrak{E}$ *then SOME* $\mathfrak{F}$. $\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$ *else 0*)›

lemma *kf-apply-km-some-kraus-family*[*simp*]:
assumes ‹*kraus-map* $\mathfrak{E}$›
shows ‹*kf-apply* (*km-some-kraus-family* $\mathfrak{E}$) = $\mathfrak{E}$›
unfolding *km-some-kraus-family-def*
apply (*rule someI2-ex*)
using *assms kraus-map-def* **by** *auto*

lemma *km-some-kraus-family-invalid*:
assumes ‹¬ *kraus-map* $\mathfrak{E}$›
shows ‹*km-some-kraus-family* $\mathfrak{E}$ = *0*›
by (*simp add*: *assms km-some-kraus-family-def*)

definition *km-operators-in* :: ‹(('*a*::*chilbert-space*,'*a*) *trace-class* ⇒ ('*b*::*chilbert-space*,'*b*) *trace-class*)

$\Rightarrow$ ('a $\Rightarrow_{CL}$ 'b) *set* $\Rightarrow$ *bool*› **where**
‹*km-operators-in* $\mathfrak{E}$ S $\longleftrightarrow$ ($\exists \mathfrak{F}$ :: ('a,'b,*unit*) *kraus-family*. *kf-apply* $\mathfrak{F}$ = $\mathfrak{E}$ $\wedge$ *kf-operators* $\mathfrak{F}$ $\subseteq$ S)›

lemma *km-operators-in-mono*: ‹$S \subseteq T \Longrightarrow$ *km-operators-in* $\mathfrak{E}$ $S \Longrightarrow$ *km-operators-in* $\mathfrak{E}$ T›
by (*metis basic-trans-rules*(*23*) *km-operators-in-def*)

lemma *km-operators-in-kf-apply*:
assumes ‹*span* (*kf-operators* $\mathfrak{E}$) $\subseteq$ S›
shows ‹*km-operators-in* (*kf-apply* $\mathfrak{E}$) S›
proof (*unfold km-operators-in-def*, *intro conjI exI*[**where** x=‹*kf-flatten* $\mathfrak{E}$›])
show ‹($*_{kr}$) (*kf-flatten* $\mathfrak{E}$) = ($*_{kr}$) $\mathfrak{E}$›
by *simp*
from *assms* **show** ‹*kf-operators* (*kf-flatten* $\mathfrak{E}$) $\subseteq$ S›
using *kf-operators-kf-map* **by** *fastforce*
qed

lemma *km-operators-in-kf-apply-flattened*:
fixes $\mathfrak{E}$ :: ‹('a::*chilbert-space*,'b::*chilbert-space*,'x::*CARD-1*) *kraus-family*›
assumes ‹*kf-operators* $\mathfrak{E}$ $\subseteq$ S›
shows ‹*km-operators-in* (*kf-apply* $\mathfrak{E}$) S›
proof (*unfold km-operators-in-def*, *intro conjI exI*[**where** x=‹*kf-map-inj* ($\lambda x.()$) $\mathfrak{E}$›])
show ‹($*_{kr}$) (*kf-map-inj* ($\lambda \mathfrak{F}$. ()) $\mathfrak{E}$) = ($*_{kr}$) $\mathfrak{E}$›
by (*auto intro*!: *ext kf-apply-map-inj inj-onI*)
have ‹*kf-operators* (*kf-map-inj* ($\lambda \mathfrak{F}$. ()) $\mathfrak{E}$) = *kf-operators* $\mathfrak{E}$›
by *simp*
with *assms* **show** ‹*kf-operators* (*kf-map-inj* ($\lambda \mathfrak{F}$. ()) $\mathfrak{E}$) $\subseteq$ S›
by *blast*
qed

lemma *km-commute*:
assumes ‹*km-operators-in* $\mathfrak{E}$ S›
assumes ‹*km-operators-in* $\mathfrak{F}$ T›
assumes ‹$S \subseteq$ *commutant* T›
shows ‹$\mathfrak{F}$ *o* $\mathfrak{E}$ = $\mathfrak{E}$ *o* $\mathfrak{F}$›
proof −
from *assms* **obtain** $\mathfrak{E}'$ :: ‹(-,-,*unit*) *kraus-family*› **where** $\mathfrak{E}'$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$› **and** $\mathfrak{E}'S$: ‹*kf-operators* $\mathfrak{E}'$ $\subseteq$ S›
by (*metis km-operators-in-def*)
from *assms* **obtain** $\mathfrak{F}'$ :: ‹(-,-,*unit*) *kraus-family*› **where** $\mathfrak{F}'$: ‹$\mathfrak{F}$ = *kf-apply* $\mathfrak{F}'$› **and** $\mathfrak{F}'T$: ‹*kf-operators* $\mathfrak{F}'$ $\subseteq$ T›
by (*metis km-operators-in-def*)

have ‹*kf-operators* $\mathfrak{E}'$ $\subseteq$ S›
by (*rule* $\mathfrak{E}'S$)
also have ‹$S \subseteq$ *commutant* T›
by (*rule assms*)
also have ‹*commutant* $T \subseteq$ *commutant* (*kf-operators* $\mathfrak{F}'$)›
apply (*rule commutant-antimono*)

by (*rule* $\mathfrak{F}'T$)
finally show *?thesis*
unfolding $\mathfrak{E}'$ $\mathfrak{F}'$
by (*rule kf-apply-commute*[*symmetric*])
qed

lemma *km-operators-in-UNIV*:
assumes ‹*kraus-map* $\mathfrak{E}$›
shows ‹*km-operators-in* $\mathfrak{E}$ *UNIV*›
by (*metis assms kf-apply-km-some-kraus-family km-operators-in-def top.extremum*)

lemma *separating-kraus-map-bounded-clinear*:
fixes *S* :: ‹('*a*::*chilbert-space*,'*a*) *trace-class set*›
assumes ‹*separating-set* (*bounded-clinear* :: (- ⇒ ('*b*::*chilbert-space*,'*b*) *trace-class*) ⇒ -) *S*›
shows ‹*separating-set* (*kraus-map* :: (- ⇒ ('*b*::*chilbert-space*,'*b*) *trace-class*) ⇒ -) *S*›
by (*metis* (*mono-tags*, *lifting*) *assms kraus-map-bounded-clinear separating-set-def*)

3.2 Bound and norm

definition *km-bound* :: ‹(('*a*::*chilbert-space*, '*a*) *trace-class* ⇒ ('*b*::*chilbert-space*, '*b*) *trace-class*) ⇒ ('*a*, '*a*) *cblinfun*› **where**
‹*km-bound* $\mathfrak{E}$ = (*if* ∃ $\mathfrak{E}'$ :: (-, -, *unit*) *kraus-family*. $\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$ *then kf-bound* (*SOME* $\mathfrak{E}'$ :: (-, -, *unit*) *kraus-family*. $\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$) *else 0*)›

lemma *km-bound-kf-bound*:
assumes ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$›
shows ‹*km-bound* $\mathfrak{E}$ = *kf-bound* $\mathfrak{F}$›
proof −
wlog *ex*: ‹∃ $\mathfrak{E}'$ :: (-, -, *unit*) *kraus-family*. $\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$›
using *assms kraus-map-def-raw negation* **by** *blast*
define $\mathfrak{E}'$ **where** ‹$\mathfrak{E}'$ = (*SOME* $\mathfrak{E}'$ :: (-, -, *unit*) *kraus-family*. $\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$)›
have ‹$\mathfrak{E}$ = *kf-apply* (*kf-flatten* $\mathfrak{F}$)›
by (*simp add*: *assms*)
then have ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$›
by (*metis* (*mono-tags*, *lifting*) $\mathfrak{E}'$*-def someI-ex*)
then have ‹$\mathfrak{E}'$ =$_{kr}$ $\mathfrak{F}$›
using *assms kf-eq-weak-def* **by** *force*
then have ‹*kf-bound* $\mathfrak{E}'$ = *kf-bound* $\mathfrak{F}$›
using *kf-bound-cong* **by** *blast*
then show *?thesis*
by (*metis* $\mathfrak{E}'$*-def km-bound-def ex*)
qed

definition *km-norm* :: ‹(('*a*::*chilbert-space*, '*a*) *trace-class* ⇒ ('*b*::*chilbert-space*, '*b*) *trace-class*) ⇒ *real*› **where**
‹*km-norm* $\mathfrak{E}$ = *norm* (*km-bound* $\mathfrak{E}$)›

lemma *km-norm-kf-norm*:

assumes ‹$\mathfrak{E} = kf\text{-}apply\ \mathfrak{F}$›
shows ‹$km\text{-}norm\ \mathfrak{E} = kf\text{-}norm\ \mathfrak{F}$›
by (*simp add*: *assms kf-norm-def km-bound-kf-bound km-norm-def*)

lemma *km-bound-invalid*:
assumes ‹$\neg$ *kraus-map* $\mathfrak{E}$›
shows ‹$km\text{-}bound\ \mathfrak{E} = 0$›
by (*metis assms km-bound-def kraus-map-def-raw*)

lemma *km-norm-invalid*:
assumes ‹$\neg$ *kraus-map* $\mathfrak{E}$›
shows ‹$km\text{-}norm\ \mathfrak{E} = 0$›
by (*simp add*: *assms km-bound-invalid km-norm-def*)

lemma *km-norm-geq0*[*iff*]: ‹$km\text{-}norm\ \mathfrak{E} \geq 0$›
by (*simp add*: *km-norm-def*)

lemma *kf-bound-pos*[*iff*]: ‹$km\text{-}bound\ \mathfrak{E} \geq 0$›
apply (*cases* ‹*kraus-map* $\mathfrak{E}$›)
apply (*simp add*: *km-bound-def*)
by (*simp add*: *km-bound-invalid*)

lemma *km-bounded-pos*:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹$\varrho \geq 0$›
shows ‹$norm\ (\mathfrak{E}\ \varrho) \leq km\text{-}norm\ \mathfrak{E} * norm\ \varrho$›
proof −
from *assms* **obtain** $\mathfrak{E}'$ :: ‹('*a*, '*b*, *unit*) *kraus-family*› **where** $\mathfrak{E}'$: ‹$\mathfrak{E} = kf\text{-}apply\ \mathfrak{E}'$›
using *kraus-map-def* **by** *blast*
show *?thesis*
by (*simp add*: $\mathfrak{E}'$ *assms*(*2*) *kf-apply-bounded-pos km-norm-kf-norm*)
qed

lemma *km-bounded*:
assumes ‹*kraus-map* $\mathfrak{E}$›
shows ‹$norm\ (\mathfrak{E}\ \varrho) \leq 4 * km\text{-}norm\ \mathfrak{E} * norm\ \varrho$›
proof −
from *assms* **obtain** $\mathfrak{E}'$ :: ‹('*a*, '*b*, *unit*) *kraus-family*› **where** $\mathfrak{E}'$: ‹$\mathfrak{E} = kf\text{-}apply\ \mathfrak{E}'$›
using *kraus-map-def* **by** *blast*
show *?thesis*
by (*simp add*: $\mathfrak{E}'$ *kf-apply-bounded km-norm-kf-norm*)
qed

lemma *km-bound-from-map*:
assumes ‹*kraus-map* $\mathfrak{E}$›
shows ‹$\psi \cdot_C km\text{-}bound\ \mathfrak{E}\ \varphi = trace\text{-}tc\ (\mathfrak{E}\ (tc\text{-}butterfly\ \varphi\ \psi))$›
by (*metis assms kf-bound-from-map km-bound-kf-bound kraus-map-def-raw*)

lemma *trace-from-km-bound*:
assumes ‹*kraus-map* $\mathfrak{E}$›
shows ‹*trace-tc* ($\mathfrak{E}$ ϱ) = *trace-tc* (*compose-tcr* (*km-bound* $\mathfrak{E}$) ϱ)›
by (*metis assms km-bound-kf-bound kraus-map-def-raw trace-from-kf-bound*)

lemma *km-bound-selfadjoint*[*iff*]: ‹*selfadjoint* (*km-bound* $\mathfrak{E}$)›
by (*simp add*: *pos-selfadjoint*)

lemma *km-bound-leq-km-norm-id*: ‹*km-bound* $\mathfrak{E}$ ≤ *km-norm* $\mathfrak{E}$ $*_R$ *id-cblinfun*›
by (*simp add*: *km-norm-def less-eq-scaled-id-norm*)

lemma *kf-norm-km-some-kraus-family*[*simp*]: ‹*kf-norm* (*km-some-kraus-family* $\mathfrak{E}$) = *km-norm* $\mathfrak{E}$›
apply (*cases* ‹*kraus-map* $\mathfrak{E}$›)
by (*auto intro*!: *km-norm-kf-norm*[*symmetric*] *simp*: *km-some-kraus-family-invalid km-norm-invalid*)

3.3 Basic Kraus maps

Zero map and constant maps. Addition and rescaling and composition of maps.

lemma *kraus-map-0*[*iff*]: ‹*kraus-map* 0›
by (*metis kf-apply-0 kraus-mapI*)

lemma *kraus-map-0'*[*iff*]: ‹*kraus-map* (λ-. 0)›
using *kraus-map-0* **unfolding** *func-zero* **by** *simp*
lemma *km-bound-0*[*simp*]: ‹*km-bound* 0 = 0›
using *km-bound-kf-bound*[*of* 0 0]
by *simp*

lemma *km-norm-0*[*simp*]: ‹*km-norm* 0 = 0›
by (*simp add*: *km-norm-def*)

lemma *km-some-kraus-family-0*[*simp*]: ‹*km-some-kraus-family* 0 = 0›
apply (*rule kf-eq-0-iff-eq-0*[*THEN iffD1*])
by (*simp add*: *kf-eq-weak-def*)

lemma *kraus-map-id*[*iff*]: ‹*kraus-map id*›
by (*auto intro*!: *ext kraus-mapI*[*of* - *kf-id*])

lemma *km-bound-id*[*simp*]: ‹*km-bound id* = *id-cblinfun*›
using *km-bound-kf-bound*[*of id kf-id*]
by (*simp add*: *kf-id-apply*[*abs-def*] *id-def*)

lemma *km-norm-id-leq1*[*iff*]: ‹*km-norm id* ≤ 1›
by (*simp add*: *km-norm-def norm-cblinfun-id-le*)

lemma *km-norm-id-eq1*[*simp*]: ‹*km-norm* (*id* :: ('*a* :: {*chilbert-space*, *not-singleton*}, '*a*) *trace-class* ⇒ -) = 1›
by (*simp add*: *km-norm-def*)

lemma *km-operators-in-id*[*iff*]: ‹*km-operators-in id* {*id-cblinfun*}›
apply (*subst asm-rl*[*of* ‹*id* = *kf-apply kf-id*›])
by (*auto simp*: *km-operators-in-kf-apply-flattened*)

lemma *kraus-map-add*[*iff*]:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹*kraus-map* $\mathfrak{F}$›
shows ‹*kraus-map* ($\lambda\varrho$. $\mathfrak{E}$ ϱ + $\mathfrak{F}$ ϱ)›
proof −
from *assms* **obtain** $\mathfrak{E}'$ :: ‹('*a*, '*b*, *unit*) *kraus-family*› **where** $\mathfrak{E}'$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$›
using *kraus-map-def* **by** *blast*
from *assms* **obtain** $\mathfrak{F}'$ :: ‹('*a*, '*b*, *unit*) *kraus-family*› **where** $\mathfrak{F}'$: ‹$\mathfrak{F}$ = *kf-apply* $\mathfrak{F}'$›
using *kraus-map-def* **by** *blast*
show *?thesis*
apply (*rule kraus-mapI*[*of* - ‹$\mathfrak{E}'$ + $\mathfrak{F}'$›])
by (*auto intro*!: *ext simp*: $\mathfrak{E}'$ $\mathfrak{F}'$ *kf-plus-apply'*)
qed

lemma *kraus-map-plus'*[*iff*]:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹*kraus-map* $\mathfrak{F}$›
shows ‹*kraus-map* ($\mathfrak{E}$ + $\mathfrak{F}$)›
using *assms* **by** (*simp add*: *plus-fun-def*)

lemma *km-bound-plus*:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹*kraus-map* $\mathfrak{F}$›
shows ‹*km-bound* ($\mathfrak{E}$ + $\mathfrak{F}$) = *km-bound* $\mathfrak{E}$ + *km-bound* $\mathfrak{F}$›
proof −
from *assms* **obtain** *EE* :: ‹('*a*,'*b*,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
from *assms* **obtain** *FF* :: ‹('*a*,'*b*,*unit*) *kraus-family*› **where** *FF*: ‹$\mathfrak{F}$ = *kf-apply FF*›
using *kraus-map-def-raw* **by** *blast*
show *?thesis*
apply (*rule km-bound-kf-bound*[**where** $\mathfrak{F}$=‹*EE* + *FF*›, *THEN trans*])
by (*auto intro*!: *ext simp*: *EE FF kf-plus-apply' kf-plus-bound' km-bound-kf-bound*)
qed

lemma *km-norm-triangle*:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹*kraus-map* $\mathfrak{F}$›
shows ‹*km-norm* ($\mathfrak{E}$ + $\mathfrak{F}$) ≤ *km-norm* $\mathfrak{E}$ + *km-norm* $\mathfrak{F}$›
by (*simp add*: *km-norm-def km-bound-plus assms norm-triangle-ineq*)

lemma *kraus-map-constant*[*iff*]: ‹*kraus-map* ($\lambda\sigma$. *trace-tc* σ *$_C$ ϱ)› **if** ‹$\varrho \geq 0$›
apply (*rule kraus-mapI*[**where** $\mathfrak{E}'$=‹*kf-constant* ϱ›])
by (*simp add*: *kf-constant-apply*[*OF that*, *abs-def*])

lemma *kraus-map-constant-invalid*:
‹¬ *kraus-map* ($\lambda\sigma$ :: ('*a*::{*chilbert-space*,*not-singleton*},'*a*) *trace-class*. *trace-tc* σ *$_C$ ϱ)› **if** ‹$\sim$ $\varrho \geq 0$›

proof (*rule ccontr*)
assume ‹¬ ¬ *kraus-map* (λσ :: ('a,'a) *trace-class*. *trace-tc* σ ∗$_C$ ϱ)›
then have *km*: ‹*kraus-map* (λσ :: ('a,'a) *trace-class*. *trace-tc* σ ∗$_C$ ϱ)›
by *simp*
obtain *h* :: 'a **where** ‹*norm h* = *1*›
using *ex-norm1-not-singleton* **by** *blast*
from *km* **have** ‹*trace-tc* (*tc-butterfly h h*) ∗$_C$ ϱ ≥ *0*›
using *kraus-map-pos* **by** *fastforce*
with ‹*norm h* = *1*›
have ‹ϱ ≥ *0*›
by (*metis mult-cancel-left2 norm-tc-butterfly norm-tc-pos of-real-1 scaleC-one tc-butterfly-pos*)
with *that* **show** *False*
by *simp*
qed

lemma *kraus-map-scale*:
assumes ‹*kraus-map* 𝔈› **and** ‹*c* ≥ *0*›
shows ‹*kraus-map* (λϱ. *c* ∗$_R$ 𝔈 ϱ)›
proof −
from *assms* **obtain** 𝔈' :: ‹('a, 'b, *unit*) *kraus-family*› **where** 𝔈': ‹𝔈 = *kf-apply* 𝔈'›
using *kraus-map-def* **by** *blast*
then show *?thesis*
apply (*rule-tac kraus-mapI*[**where** 𝔈'=‹*c* ∗$_R$ 𝔈'›])
by (*auto intro*!: *ext simp add*: 𝔈' *kf-scale-apply assms*)
qed

lemma *km-bound-scale*[*simp*]: ‹*km-bound* (λϱ. *c* ∗$_R$ 𝔈 ϱ) = *c* ∗$_R$ *km-bound* 𝔈› **if** ‹*c* ≥ *0*›
proof −
consider (*km*) ‹*kraus-map* 𝔈› | (*c0*) ‹*c* = *0*› | (*not-km*) ‹¬ *kraus-map* 𝔈› ‹*c* > *0*›
using ‹*0* ≤ *c*› **by** *argo*
then show *?thesis*
proof *cases*
case *km*
then obtain *EE* :: ‹('a,'b,*unit*) *kraus-family*› **where** *EE*: ‹𝔈 = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
with ‹*c* ≥ *0* ›**have** ‹*kf-bound* (*c* ∗$_R$ *EE*) = *c* ∗$_R$ *kf-bound EE*›
by *simp*
then show *?thesis*
using *km-bound-kf-bound*[*of* ‹λϱ. *c* ∗$_R$ 𝔈 ϱ› ‹*c* ∗$_R$ *EE*›, *symmetric*]
using *kf-scale-apply*[*OF* ‹*c* ≥ *0*›, *of EE*, *abs-def*]
by (*simp add*: *EE km-bound-kf-bound*)
next
case *c0*
then show *?thesis*
by (*simp flip*: *func-zero*)
next
case *not-km*
have ‹¬ *kraus-map* (λϱ. *c* ∗$_R$ 𝔈 ϱ)›
proof (*rule ccontr*)

assume ‹¬ ¬ *kraus-map* (λϱ. *c* *$_R$ 𝔈 ϱ)›
then have ‹*kraus-map* (λϱ. *inverse c* *$_R$ (*c* *$_R$ 𝔈 ϱ))›
apply (*rule-tac kraus-map-scale*)
using *not-km* **by** *auto*
then have ‹*kraus-map* 𝔈›
using *not-km* **by** (*simp add*: *scaleR-scaleR field.field-inverse*)
with *not-km* **show** *False*
by *simp*
qed
with *not-km* **show** *?thesis*
by (*simp add*: *km-bound-invalid*)
qed
qed

lemma *km-norm-scale*[*simp*]: ‹*km-norm* (λϱ. *c* *$_R$ 𝔈 ϱ) = *c* * *km-norm* 𝔈› **if** ‹*c* ≥ *0*›
using *that* **by** (*simp add*: *km-norm-def*)

lemma *kraus-map-sandwich*[*iff*]: ‹*kraus-map* (*sandwich-tc A*)›
apply (*rule kraus-mapI*[*of* - ‹*kf-of-op A*›])
using *kf-of-op-apply*[*of A*, *abs-def*]
by *simp*

lemma *km-bound-sandwich*[*simp*]: ‹*km-bound* (*sandwich-tc A*) = *A** *o$_{CL}$ A*›
using *km-bound-kf-bound*[*of* ‹*sandwich-tc A*› ‹*kf-of-op A*›, *symmetric*]
using *kf-bound-of-op*[*of A*]
using *kf-of-op-apply*[*of A*]
by *fastforce*

lemma *km-norm-sandwich*[*simp*]: ‹*km-norm* (*sandwich-tc A*) = (*norm A*)^2›
by (*simp add*: *km-norm-def*)

lemma *km-operators-in-sandwich*: ‹*km-operators-in* (*sandwich-tc U*) {*U*}›
apply (*subst kf-of-op-apply*[*abs-def*, *symmetric*])
apply (*rule km-operators-in-kf-apply-flattened*)
by *simp*

lemma *km-constant-bound*[*simp*]: ‹*km-bound* (λσ. *trace-tc* σ *$_C$ ϱ) = *norm* ϱ *$_R$ id-cblinfun›* **if** ‹ϱ ≥ *0*›
apply (*rule km-bound-kf-bound*[*THEN trans*])
using *that* **apply** (*rule kf-constant-apply*[*symmetric*, *THEN ext*])
using *that* **by** (*rule kf-bound-constant*)

lemma *km-constant-norm*[*simp*]: ‹*km-norm* (λσ::('*a*::{*chilbert-space*,*not-singleton*},'*a*) *trace-class*. *trace-tc* σ *$_C$ ϱ) = *norm* ϱ› **if** ‹ϱ ≥ *0*›
apply (*subst km-norm-kf-norm*[*of* ‹(λσ::('*a*,'*a*) *trace-class*. *trace-tc* σ *$_C$ ϱ)› ‹*kf-constant* ϱ›])
apply (*subst kf-constant-apply*[*OF that*, *abs-def*], *rule refl*)

apply (*rule kf-norm-constant*)
by (*fact that*)

lemma *km-constant-norm-leq*[*simp*]: ‹*km-norm* (λσ::('*a*::*chilbert-space*,'*a*) *trace-class*. *trace-tc* σ *$_C$* ϱ) ≤ *norm* ϱ›
proof −
consider (*pos*) ‹ϱ ≥ 0› | (*singleton*) ‹¬ *class.not-singleton TYPE*('*a*)› | (*nonpos*) ‹¬ ϱ ≥ 0› ‹*class.not-singleton TYPE*('*a*)›
by *blast*
then show *?thesis*
proof *cases*
case *pos*
show *?thesis*
apply (*subst km-norm-kf-norm*[*of* ‹(λσ::('*a*,'*a*) *trace-class*. *trace-tc* σ *$_C$* ϱ)› ‹*kf-constant* ϱ›])
apply (*subst kf-constant-apply*[*OF pos*, *abs-def*], *rule refl*)
by (*rule kf-norm-constant-leq*)
next
case *nonpos*
have ‹¬ *kraus-map* (λσ::('*a*,'*a*) *trace-class*. *trace-tc* σ *$_C$* ϱ)›
apply (*rule kraus-map-constant-invalid*[*internalize-sort'* '*a*])
apply (*rule chilbert-space-axioms*)
using *nonpos* **by** −
then have ‹*km-norm* (λσ::('*a*,'*a*) *trace-class*. *trace-tc* σ *$_C$* ϱ) = 0›
using *km-norm-invalid* **by** *blast*
then show *?thesis*
by (*metis norm-ge-zero*)
next
case *singleton*
then have ‹(λσ::('*a*,'*a*) *trace-class*. *trace-tc* σ *$_C$* ϱ) = 0›
apply (*subst not-not-singleton-tc-zero*)
by *auto*
then show *?thesis*
by *simp*
qed
qed

lemma *kraus-map-comp*:
assumes ‹*kraus-map* 𝔈› **and** ‹*kraus-map* 𝔉›
shows ‹*kraus-map* (𝔈 *o* 𝔉)›
proof −
from *assms* **obtain** *EE* :: ‹('*a*,'*b*,*unit*) *kraus-family*› **where** *EE*: ‹𝔈 = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
from *assms* **obtain** *FF* :: ‹('*c*,'*a*,*unit*) *kraus-family*› **where** *FF*: ‹𝔉 = *kf-apply FF*›
using *kraus-map-def-raw* **by** *blast*
show *?thesis*
apply (*rule kraus-mapI*[**where** 𝔈'=‹*kf-comp EE FF*›])
by (*simp add*: *EE FF kf-comp-apply*)
qed

lemma *km-comp-norm-leq*:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹*kraus-map* $\mathfrak{F}$›
shows ‹*km-norm* ($\mathfrak{E}$ *o* $\mathfrak{F}$) ≤ *km-norm* $\mathfrak{E}$ ∗ *km-norm* $\mathfrak{F}$›
proof −
from *assms* **obtain** *EE* :: ‹('*a*,'*b*,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
from *assms* **obtain** *FF* :: ‹('*c*,'*a*,*unit*) *kraus-family*› **where** *FF*: ‹$\mathfrak{F}$ = *kf-apply FF*›
using *kraus-map-def-raw* **by** *blast*
have ‹*km-norm* ($\mathfrak{E}$ *o* $\mathfrak{F}$) = *kf-norm* (*kf-comp EE FF*)›
by (*simp add*: *EE FF kf-comp-apply km-norm-kf-norm*)
also have ‹... ≤ *kf-norm EE* ∗ *kf-norm FF*›
by (*simp add*: *kf-comp-norm-leq*)
also have ‹... = *km-norm* $\mathfrak{E}$ ∗ *km-norm* $\mathfrak{F}$›
by (*simp add*: *EE FF km-norm-kf-norm*)
finally show *?thesis*
by −
qed

lemma *km-bound-comp-sandwich*:
assumes ‹*kraus-map* $\mathfrak{E}$›
shows ‹*km-bound* (λϱ. $\mathfrak{E}$ (*sandwich-tc U* ϱ)) = *sandwich* (*U*∗) (*km-bound* $\mathfrak{E}$)›
proof −
from *assms* **obtain** *EE* :: ‹('*a*,'*b*,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
have ‹*km-bound* (λϱ. $\mathfrak{E}$ (*sandwich-tc U* ϱ)) = *kf-bound* (*kf-comp EE* (*kf-of-op U*))›
apply (*rule km-bound-kf-bound*)
by (*simp add*: *kf-comp-apply o-def EE kf-of-op-apply*)
also have ‹... = *sandwich* (*U*∗) ∗$_V$ *kf-bound EE*›
by (*simp add*: *kf-bound-comp-of-op*)
also have ‹... = *sandwich* (*U*∗) (*km-bound* $\mathfrak{E}$)›
by (*simp add*: *EE km-bound-kf-bound*)
finally show *?thesis*
by −
qed

lemma *km-norm-comp-sandwich-coiso*:
assumes ‹*isometry* (*U*∗)›
shows ‹*km-norm* (λϱ. $\mathfrak{E}$ (*sandwich-tc U* ϱ)) = *km-norm* $\mathfrak{E}$›
proof (*cases* ‹*kraus-map* $\mathfrak{E}$›)
case *True*
then obtain *EE* :: ‹(-,-,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
have ‹*km-norm* (λϱ. $\mathfrak{E}$ (*sandwich-tc U* ϱ)) = *kf-norm* (*kf-comp EE* (*kf-of-op U*))›
apply (*rule km-norm-kf-norm*)

```
    by (auto intro!: simp: kf-comp-apply o-def EE kf-of-op-apply)
  also have ‹... = kf-norm EE›
    by (simp add: assms kf-norm-comp-of-op-coiso)
  also have ‹... = km-norm 𝔈›
    by (simp add: EE km-norm-kf-norm)
  finally show ?thesis
    by −
next
  case False
  have ‹¬ kraus-map (λϱ. 𝔈 (sandwich-tc U ϱ))›
  proof (rule ccontr)
    assume ‹¬ ¬ kraus-map (λϱ. 𝔈 (sandwich-tc U ϱ))›
    then have ‹kraus-map (λϱ. 𝔈 (sandwich-tc U ϱ))›
      by blast
    then have ‹kraus-map (λϱ. 𝔈 (sandwich-tc U (sandwich-tc (U*) ϱ)))›
      apply (rule kraus-map-comp[unfolded o-def])
      by fastforce
    then have ‹kraus-map 𝔈›
      using isometryD[OF assms]
      by (simp flip: sandwich-tc-compose[unfolded o-def, THEN fun-cong])
    then show False
      using False by blast
  qed
  with False show ?thesis
    by (simp add: km-norm-invalid)
qed

lemma km-bound-comp-sandwich-iso:
  assumes ‹isometry U›
  shows ‹km-bound (λϱ. sandwich-tc U (𝔈 ϱ)) = km-bound 𝔈›
proof (cases ‹kraus-map 𝔈›)
  case True
  then obtain EE :: ‹(-,-,unit) kraus-family› where EE: ‹𝔈 = kf-apply EE›
    using kraus-map-def-raw by blast
  have ‹km-bound (λϱ. sandwich-tc U (𝔈 ϱ)) = kf-bound (kf-comp (kf-of-op U) EE)›
    apply (rule km-bound-kf-bound)
    by (auto intro!: simp: kf-comp-apply o-def EE kf-of-op-apply)
  also have ‹... = kf-bound EE›
    by (simp add: assms kf-bound-comp-iso)
  also have ‹... = km-bound 𝔈›
    by (simp add: EE km-bound-kf-bound)
  finally show ?thesis
    by −
next
  case False
  have ‹¬ kraus-map (λϱ. sandwich-tc U (𝔈 ϱ))›
  proof (rule ccontr)
    assume ‹¬ ¬ kraus-map (λϱ. sandwich-tc U (𝔈 ϱ))›
    then have ‹kraus-map (λϱ. sandwich-tc U (𝔈 ϱ))›
```

```
    by blast
  then have ‹kraus-map (λϱ. sandwich-tc (U*) (sandwich-tc U (𝔈 ϱ)))›
    apply (rule kraus-map-comp[unfolded o-def, rotated])
    by fastforce
  then have ‹kraus-map 𝔈›
    using isometryD[OF assms]
    by (simp flip: sandwich-tc-compose[unfolded o-def, THEN fun-cong])
  then show False
    using False by blast
 qed
 with False show ?thesis
  by (simp add: km-bound-invalid)
qed

lemma km-norm-comp-sandwich-iso:
 assumes ‹isometry U›
 shows ‹km-norm (λϱ. sandwich-tc U (𝔈 ϱ)) = km-norm 𝔈›
proof (cases ‹kraus-map 𝔈›)
 case True
 then obtain EE :: ‹(-,-,unit) kraus-family› where EE: ‹𝔈 = kf-apply EE›
  using kraus-map-def-raw by blast
 have ‹km-norm (λϱ. sandwich-tc U (𝔈 ϱ)) = kf-norm (kf-comp (kf-of-op U) EE)›
  apply (rule km-norm-kf-norm)
  by (auto intro!: simp: kf-comp-apply o-def EE kf-of-op-apply)
 also have ‹... = kf-norm EE›
  by (simp add: assms kf-norm-comp-iso)
 also have ‹... = km-norm 𝔈›
  by (simp add: EE km-norm-kf-norm)
 finally show ?thesis
  by −
next
 case False
 have ‹¬ kraus-map (λϱ. sandwich-tc U (𝔈 ϱ))›
 proof (rule ccontr)
  assume ‹¬ ¬ kraus-map (λϱ. sandwich-tc U (𝔈 ϱ))›
  then have ‹kraus-map (λϱ. sandwich-tc U (𝔈 ϱ))›
    by blast
  then have ‹kraus-map (λϱ. sandwich-tc (U*) (sandwich-tc U (𝔈 ϱ)))›
    apply (rule kraus-map-comp[unfolded o-def, rotated])
    by fastforce
  then have ‹kraus-map 𝔈›
    using isometryD[OF assms]
    by (simp flip: sandwich-tc-compose[unfolded o-def, THEN fun-cong])
  then show False
    using False by blast
 qed
 with False show ?thesis
  by (simp add: km-norm-invalid)
qed
```

lemma *kraus-map-sum*:
assumes ‹$\bigwedge x.\ x{\in}A \Longrightarrow$ *kraus-map* ($\mathfrak{E}$ x)›
shows ‹*kraus-map* ($\sum x{\in}A.\ \mathfrak{E}\ x$)›
apply (*insert assms*, *induction A rule*:*infinite-finite-induct*)
by *auto*

lemma *km-bound-sum*:
assumes ‹$\bigwedge x.\ x{\in}A \Longrightarrow$ *kraus-map* ($\mathfrak{E}$ x)›
shows ‹*km-bound* ($\sum x{\in}A.\ \mathfrak{E}\ x$) = ($\sum x{\in}A.$ *km-bound* ($\mathfrak{E}$ x))›
proof (*insert assms*, *induction A rule*:*infinite-finite-induct*)
case (*infinite A*)
then show *?case*
by (*metis km-bound-0 sum.infinite*)
next
case *empty*
then show *?case*
by (*metis km-bound-0 sum.empty*)
next
case (*insert x F*)
have ‹*km-bound* ($\sum x{\in}$*insert x F*. $\mathfrak{E}\ x$) = *km-bound* ($\mathfrak{E}\ x$ + ($\sum x{\in}F.\ \mathfrak{E}\ x$))›
by (*simp add*: *insert.hyps*)
also have ‹$\ldots$ = *km-bound* ($\mathfrak{E}\ x$) + *km-bound* ($\sum x{\in}F.\ \mathfrak{E}\ x$)›
by (*simp add*: *km-bound-plus kraus-map-sum insert.prems*)
also have ‹$\ldots$ = *km-bound* ($\mathfrak{E}\ x$) + ($\sum x{\in}F.$ *km-bound* ($\mathfrak{E}\ x$))›
by (*simp add*: *insert*)
also have ‹$\ldots$ = ($\sum x{\in}$*insert x F*. *km-bound* ($\mathfrak{E}\ x$))›
using *insert.hyps* **by** *fastforce*
finally show *?case*
by −
qed

3.4 Infinite sums

lemma
assumes ‹$\bigwedge\varrho.$ (($\lambda a.$ *sandwich-tc* ($f\ a$) ϱ) *has-sum* $\mathfrak{E}$ ϱ) A›
defines ‹$EE \equiv$ *Set.filter* ($\lambda(E,\text{-}).\ E{\neq}0$) (($\lambda a.$ ($f\ a,\ a$)) ‘ A)›
shows *kraus-mapI-sum*: ‹*kraus-map* $\mathfrak{E}$›
and *kraus-map-sum-kraus-family*: ‹*kraus-family EE*›
and *kraus-map-sum-kf-apply*: ‹$\mathfrak{E}$ = *kf-apply* (*Abs-kraus-family EE*)›
proof −
show ‹*kraus-family EE*›
unfolding *EE-def*
apply (*rule kf-reconstruction-is-kraus-family*)
by (*fact assms*(*1*))
show ‹$\mathfrak{E}$ = *kf-apply* (*Abs-kraus-family EE*)›
unfolding *EE-def*
apply (*rule kf-reconstruction*[*symmetric*])

by (*fact assms*(*1*))
then show ‹*kraus-map* $\mathfrak{E}$›
by (*rule kraus-mapI*)
qed

lemma *kraus-map-infsum-sandwich*:
assumes ‹$\bigwedge\varrho$. (λa. *sandwich-tc* (*f a*) ϱ) *summable-on A*›
shows ‹*kraus-map* ($\lambda\varrho$. $\sum_\infty a{\in}A$. *sandwich-tc* (*f a*) ϱ)›
apply (*rule kraus-mapI-sum*)
using *assms* **by** (*rule has-sum-infsum*)

lemma *kraus-map-sum-sandwich*: ‹*kraus-map* ($\lambda\varrho$. $\sum a{\in}A$. *sandwich-tc* (*f a*) ϱ)›
apply (*cases* ‹*finite A*›)
apply (*simp add*: *kraus-map-infsum-sandwich flip*: *infsum-finite*)
by *simp*

lemma *kraus-map-as-infsum*:
assumes ‹*kraus-map* $\mathfrak{E}$›
shows ‹$\exists M.\ \forall\varrho.$ (($\lambda E.$ *sandwich-tc E* ϱ) *has-sum* $\mathfrak{E}$ ϱ) *M*›
proof −
from *assms* **obtain** $\mathfrak{E}'$ :: ‹('*a*, '*b*, *unit*) *kraus-family*› **where** $\mathfrak{E}'$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{E}'$›
using *kraus-map-def* **by** *blast*
define *M* **where** ‹*M* = *fst* ‘ *Rep-kraus-family* $\mathfrak{E}'$›
have ‹(($\lambda E.$ *sandwich-tc E* ϱ) *has-sum* $\mathfrak{E}$ ϱ) *M*› **for** ϱ
proof −
have [*simp*]: ‹*inj-on fst* (*Rep-kraus-family* $\mathfrak{E}'$)›
by (*auto intro*!: *inj-onI*)
from *kf-apply-has-sum*[*of* ϱ $\mathfrak{E}'$]
have ‹(($\lambda(E,\ x).$ *sandwich-tc E* ϱ) *has-sum kf-apply* $\mathfrak{E}'$ ϱ) (*Rep-kraus-family* $\mathfrak{E}'$)›
by −
then show *?thesis*
by (*simp add*: *M-def has-sum-reindex o-def case-prod-unfold* $\mathfrak{E}'$)
qed
then show *?thesis*
by *auto*
qed

definition *km-summable* :: ‹('*a* ⇒ ('*b*::*chilbert-space*,'*b*) *trace-class* ⇒ ('*c*::*chilbert-space*,'*c*) *trace-class*) ⇒ '*a set* ⇒ *bool*› **where**
‹*km-summable* $\mathfrak{E}$ *A* ⟷ *summable-on-in cweak-operator-topology* ($\lambda a.$ *km-bound* ($\mathfrak{E}$ *a*)) *A*›

lemma *km-summable-kf-summable*:
assumes ‹$\bigwedge a\ \varrho.\ a \in A \Longrightarrow \mathfrak{E}\ a\ \varrho = \mathfrak{F}\ a *_{kr} \varrho$›
shows ‹*km-summable* $\mathfrak{E}$ *A* ⟷ *kf-summable* $\mathfrak{F}$ *A*›
proof −

have ‹*km-summable* $\mathfrak{E}$ *A* $\longleftrightarrow$ *summable-on-in cweak-operator-topology* (*λa. km-bound* ($\mathfrak{E}$ *a*)) *A*›
using *km-summable-def* **by** *blast*
also have ‹... $\longleftrightarrow$ *summable-on-in cweak-operator-topology* (*λa. kf-bound* ($\mathfrak{F}$ *a*)) *A*›
apply (*rule summable-on-in-cong*)
apply (*rule km-bound-kf-bound*)
using *assms* **by** *fastforce*
also have ‹... $\longleftrightarrow$ *kf-summable* $\mathfrak{F}$ *A*›
using *kf-summable-def* **by** *blast*
finally show *?thesis*
by −
qed

lemma *km-summable-summable*:
assumes *km*: ‹$\bigwedge$*a*. *a*∈*A* ⟹ *kraus-map* ($\mathfrak{E}$ *a*)›
assumes *sum*: ‹*km-summable* $\mathfrak{E}$ *A*›
shows ‹(*λa*. $\mathfrak{E}$ *a* *ϱ*) *summable-on A*›
proof −
from *km*
obtain *EE* :: ‹- ⇒ (-,-,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ *a* = *kf-apply* (*EE a*)› **if** ‹*a* ∈ *A*› **for** *a*
apply *atomize-elim*
apply (*rule-tac choice*)
by (*simp add*: *kraus-map-def-raw*)
from *sum*
have ‹*kf-summable EE A*›
by (*simp add*: *EE km-summable-kf-summable*)
then have ‹(*λa. EE a* $*_{kr}$ *ϱ*) *summable-on A*›
by (*rule kf-infsum-apply-summable*)
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *EE summable-on-cong*)
qed

lemma *kraus-map-infsum*:
assumes *km*: ‹$\bigwedge$*a*. *a*∈*A* ⟹ *kraus-map* ($\mathfrak{E}$ *a*)›
assumes *sum*: ‹*km-summable* $\mathfrak{E}$ *A*›
shows ‹*kraus-map* (*λϱ*. $\sum_\infty$*a*∈*A*. $\mathfrak{E}$ *a* *ϱ*)›
proof −
from *km*
obtain *EE* :: ‹- ⇒ (-,-,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ *a* = *kf-apply* (*EE a*)› **if** ‹*a* ∈ *A*› **for** *a*
apply *atomize-elim*
apply (*rule-tac choice*)
by (*simp add*: *kraus-map-def-raw*)
from *sum*
have ‹*kf-summable EE A*›
by (*simp add*: *EE km-summable-kf-summable*)
then have ‹*kf-infsum EE A* $*_{kr}$ *ϱ* = ($\sum_\infty$*a*∈*A*. *EE a* $*_{kr}$ *ϱ*)› **for** *ϱ*

by (*metis kf-infsum-apply*)
also have ‹... $\varrho = (\sum_\infty a{\in}A.\ \mathfrak{E}\ a\ \varrho)$› **for** ϱ
by (*metis* (*mono-tags*, *lifting*) *EE infsum-cong*)
finally show *?thesis*
apply (*rule-tac kraus-mapI*[*of* - ‹*kf-infsum EE A*›])
by *auto*
qed

lemma *km-bound-infsum*:
assumes *km*: ‹$\bigwedge a.\ a{\in}A \Longrightarrow$ *kraus-map* ($\mathfrak{E}\ a$)›
assumes *sum*: ‹*km-summable* $\mathfrak{E}\ A$›
shows ‹*km-bound* ($\lambda\varrho.\ \sum_\infty a{\in}A.\ \mathfrak{E}\ a\ \varrho$) = *infsum-in cweak-operator-topology* ($\lambda a.$ *km-bound* ($\mathfrak{E}\ a$)) *A*›
proof −
from *km*
obtain *EE* :: ‹- ⇒ (-,-,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}\ a$ = *kf-apply* (*EE a*)› **if** ‹$a \in A$› **for** *a*
apply *atomize-elim*
apply (*rule-tac choice*)
by (*simp add*: *kraus-map-def-raw*)
from *sum* **have** ‹*kf-summable EE A*›
by (*simp add*: *EE km-summable-kf-summable*)
have ‹*km-bound* ($\lambda\varrho.\ \sum_\infty a{\in}A.\ \mathfrak{E}\ a\ \varrho$) = *km-bound* ($\lambda\varrho.\ \sum_\infty a{\in}A.\ EE\ a\ *_{kr}\ \varrho$)›
apply (*rule arg-cong*[**where** *f*=*km-bound*])
by (*auto intro*!: *infsum-cong simp add*: *EE*)
also have ‹... = *kf-bound* (*kf-infsum EE A*)›
apply (*rule km-bound-kf-bound*)
using *kf-infsum-apply*[*OF* ‹*kf-summable EE A*›]
by *auto*
also have ‹... = *infsum-in cweak-operator-topology* ($\lambda a.$ *kf-bound* (*EE a*)) *A*›
using ‹*kf-summable EE A*› *kf-bound-infsum* **by** *fastforce*
also have ‹... = *infsum-in cweak-operator-topology* ($\lambda a.$ *km-bound* ($\mathfrak{E}\ a$)) *A*›
by (*metis* (*mono-tags*, *lifting*) *EE infsum-in-cong km-bound-kf-bound*)
finally show *?thesis*
by −
qed

lemma *km-norm-infsum*:
assumes *km*: ‹$\bigwedge a.\ a{\in}A \Longrightarrow$ *kraus-map* ($\mathfrak{E}\ a$)›
assumes *sum*: ‹($\lambda a.$ *km-norm* ($\mathfrak{E}\ a$)) *summable-on A*›
shows ‹*km-norm* ($\lambda\varrho.\ \sum_\infty a{\in}A.\ \mathfrak{E}\ a\ \varrho$) ≤ ($\sum_\infty a{\in}A.$ *km-norm* ($\mathfrak{E}\ a$))›
proof −
from *km*
obtain *EE* :: ‹- ⇒ (-,-,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}\ a$ = *kf-apply* (*EE a*)› **if** ‹$a \in A$› **for** *a*
apply *atomize-elim*
apply (*rule-tac choice*)
by (*simp add*: *kraus-map-def-raw*)

from *sum* **have** *sum1*: ‹$(\lambda a.\ kf\text{-}norm\ (EE\ a))\ summable\text{-}on\ A$›
by (*metis* (*mono-tags*, *lifting*) *EE km-norm-kf-norm summable-on-cong*)
then have *sum2*: ‹*kf-summable EE A*›
using *kf-summable-from-abs-summable* **by** *blast*
have ‹$km\text{-}norm\ (\lambda\varrho.\ \sum_\infty a{\in}A.\ \mathfrak{E}\ a\ \varrho) = km\text{-}norm\ (\lambda\varrho.\ \sum_\infty a{\in}A.\ EE\ a *_{kr} \varrho)$›
apply (*rule arg-cong*[**where** *f*=*km-norm*])
apply (*rule ext*)
apply (*rule infsum-cong*)
by (*simp add*: *EE*)
also have ‹$\ldots = kf\text{-}norm\ (kf\text{-}infsum\ EE\ A)$›
apply (*subst kf-infsum-apply*[*OF sum2*, *abs-def*, *symmetric*])
using *km-norm-kf-norm* **by** *blast*
also have ‹$\ldots \leq (\sum_\infty a{\in}A.\ kf\text{-}norm\ (EE\ a))$›
by (*metis kf-norm-infsum sum1*)
also have ‹$\ldots = (\sum_\infty a{\in}A.\ km\text{-}norm\ (\mathfrak{E}\ a))$›
by (*metis* (*mono-tags*, *lifting*) *EE infsum-cong km-norm-kf-norm*)
finally show *?thesis*
by −
qed

lemma *kraus-map-has-sum*:
assumes ‹$\bigwedge x.\ x \in A \Longrightarrow kraus\text{-}map\ (\mathfrak{E}\ x)$›
assumes ‹*km-summable* $\mathfrak{E}$ *A*›
assumes ‹$(\mathfrak{E}\ has\text{-}sum\ \mathfrak{F})\ A$›
shows ‹*kraus-map* $\mathfrak{F}$›
proof −
from ‹$(\mathfrak{E}\ has\text{-}sum\ \mathfrak{F})\ A$›
have ‹$((\lambda x.\ \mathfrak{E}\ x\ t)\ has\text{-}sum\ \mathfrak{F}\ t)\ A$› **for** *t*
by (*simp add*: *has-sum-coordinatewise*)
then have ‹$\mathfrak{F}\ t = (\sum_\infty a{\in}A.\ \mathfrak{E}\ a\ t)$› **for** *t*
by (*metis infsumI*)
with *kraus-map-infsum*[*OF assms*(*1*,*2*)]
show ‹*kraus-map* $\mathfrak{F}$›
by *presburger*
qed

lemma *km-summable-iff-sums-to-kraus-map*:
assumes ‹$\bigwedge a.\ a{\in}A \Longrightarrow kraus\text{-}map\ (\mathfrak{E}\ a)$›
shows ‹$km\text{-}summable\ \mathfrak{E}\ A \longleftrightarrow (\exists \mathfrak{F}.\ (\forall t.\ ((\lambda x.\ \mathfrak{E}\ x\ t)\ has\text{-}sum\ \mathfrak{F}\ t)\ A) \wedge kraus\text{-}map\ \mathfrak{F})$›
proof (*rule iffI*)
assume *asm*: ‹*km-summable* $\mathfrak{E}$ *A*›
define $\mathfrak{F}$ **where** ‹$\mathfrak{F}\ t = (\sum_\infty a{\in}A.\ \mathfrak{E}\ a\ t)$› **for** *t*
from *km-summable-summable*[*OF assms asm*]
have ‹$((\lambda x.\ \mathfrak{E}\ x\ t)\ has\text{-}sum\ \mathfrak{F}\ t)\ A$› **for** *t*
using $\mathfrak{F}$*-def* **by** *fastforce*
moreover from *kraus-map-infsum*[*OF assms asm*]
have ‹*kraus-map* $\mathfrak{F}$›
by (*simp add*: $\mathfrak{F}$*-def*[*abs-def*])
ultimately show ‹$(\exists \mathfrak{F}.\ (\forall t.\ ((\lambda x.\ \mathfrak{E}\ x\ t)\ has\text{-}sum\ \mathfrak{F}\ t)\ A) \wedge kraus\text{-}map\ \mathfrak{F})$›

by *auto*
next
assume ‹$\exists \mathfrak{F}.\ (\forall t.\ ((\lambda x.\ \mathfrak{E}\ x\ t)\ \textit{has-sum}\ \mathfrak{F}\ t)\ A) \wedge \textit{kraus-map}\ \mathfrak{F}$›
then obtain $\mathfrak{F}$ **where** ‹*kraus-map* $\mathfrak{F}$› **and** ‹$((\lambda x.\ \mathfrak{E}\ x\ t)\ \textit{has-sum}\ \mathfrak{F}\ t)\ A$› **for** t
by *auto*
then have ‹$((\lambda x.\ \textit{trace-tc}\ (\mathfrak{E}\ x\ (\textit{tc-butterfly}\ k\ h)))\ \textit{has-sum}\ \textit{trace-tc}\ (\mathfrak{F}\ (\textit{tc-butterfly}\ k\ h)))\ A$› **for** $h\ k$
using *bounded-clinear.bounded-linear bounded-clinear-trace-tc has-sum-bounded-linear* **by** *blast*
then have ‹$((\lambda x.\ \textit{trace-tc}\ (\mathfrak{E}\ x\ (\textit{tc-butterfly}\ k\ h)))\ \textit{has-sum}\ h\ \cdot_C\ (\textit{km-bound}\ \mathfrak{F}\ *_V\ k))\ A$› **for** $h\ k$
by (*simp add*: *km-bound-from-map* ‹*kraus-map* $\mathfrak{F}$›)
then have ‹$((\lambda x.\ h\ \cdot_C\ (\textit{km-bound}\ (\mathfrak{E}\ x)\ *_V\ k))\ \textit{has-sum}\ h\ \cdot_C\ (\textit{km-bound}\ \mathfrak{F}\ *_V\ k))\ A$› **for** $h\ k$
apply (*rule has-sum-cong*[*THEN iffD1*, *rotated 1*])
by (*simp add*: *km-bound-from-map assms*)
then have ‹*has-sum-in cweak-operator-topology* $(\lambda a.\ \textit{km-bound}\ (\mathfrak{E}\ a))\ A\ (\textit{km-bound}\ \mathfrak{F})$›
by (*simp add*: *has-sum-in-cweak-operator-topology-pointwise*)
then show ‹*km-summable* $\mathfrak{E}\ A$›
using *summable-on-in-def km-summable-def* **by** *blast*
qed

3.5 Tensor products

definition *km-tensor-exists* :: ‹$(('a\ \textit{ell2},\ 'b\ \textit{ell2})\ \textit{trace-class} \Rightarrow ('c\ \textit{ell2},\ 'd\ \textit{ell2})\ \textit{trace-class})$
$\Rightarrow (('e\ \textit{ell2},\ 'f\ \textit{ell2})\ \textit{trace-class} \Rightarrow ('g\ \textit{ell2},\ 'h\ \textit{ell2})\ \textit{trace-class}) \Rightarrow \textit{bool}$›
where
‹*km-tensor-exists* $\mathfrak{E}\ \mathfrak{F} \longleftrightarrow (\exists \mathfrak{EF}.\ \textit{bounded-clinear}\ \mathfrak{EF} \wedge (\forall \varrho\ \sigma.\ \mathfrak{EF}\ (\textit{tc-tensor}\ \varrho\ \sigma) = \textit{tc-tensor}\ (\mathfrak{E}\ \varrho)\ (\mathfrak{F}\ \sigma)))$›

definition *km-tensor* :: ‹$(('a\ \textit{ell2},\ 'c\ \textit{ell2})\ \textit{trace-class} \Rightarrow ('e\ \textit{ell2},\ 'g\ \textit{ell2})\ \textit{trace-class})$
$\Rightarrow (('b\ \textit{ell2},\ 'd\ \textit{ell2})\ \textit{trace-class} \Rightarrow ('f\ \textit{ell2},\ 'h\ \textit{ell2})\ \textit{trace-class})$
$\Rightarrow (('a \times 'b)\ \textit{ell2},\ ('c \times 'd)\ \textit{ell2})\ \textit{trace-class} \Rightarrow (('e \times 'f)\ \textit{ell2},\ ('g \times 'h)\ \textit{ell2})$
trace-class› **where**
‹*km-tensor* $\mathfrak{E}\ \mathfrak{F} = (\textit{if}\ \textit{km-tensor-exists}\ \mathfrak{E}\ \mathfrak{F}$
then SOME $\mathfrak{EF}.\ \textit{bounded-clinear}\ \mathfrak{EF} \wedge (\forall \varrho\ \sigma.\ \mathfrak{EF}\ (\textit{tc-tensor}\ \varrho\ \sigma) = \textit{tc-tensor}\ (\mathfrak{E}\ \varrho)\ (\mathfrak{F}\ \sigma))$
else $0)$›

lemma *km-tensor-invalid*:
assumes ‹$\neg$ *km-tensor-exists* $\mathfrak{E}\ \mathfrak{F}$›
shows ‹*km-tensor* $\mathfrak{E}\ \mathfrak{F} = 0$›
by (*simp add*: *assms km-tensor-def*)

lemma *km-tensor-exists-bounded-clinear*[*iff*]:
assumes ‹*km-tensor-exists* $\mathfrak{E}\ \mathfrak{F}$›
shows ‹*bounded-clinear* (*km-tensor* $\mathfrak{E}\ \mathfrak{F}$)›
unfolding *km-tensor-def*

apply (*rule someI2-ex*[**where** *P*=‹λ𝔈𝔉. *bounded-clinear* 𝔈𝔉 ∧ (∀ϱ σ. 𝔈𝔉 (*tc-tensor* ϱ σ) = *tc-tensor* (𝔈 ϱ) (𝔉 σ))›])
using *assms*
by (*simp-all add*: *km-tensor-exists-def*)

lemma *km-tensor-apply*[*simp*]:
assumes ‹*km-tensor-exists* 𝔈 𝔉›
shows ‹*km-tensor* 𝔈 𝔉 (*tc-tensor* ϱ σ) = *tc-tensor* (𝔈 ϱ) (𝔉 σ)›
unfolding *km-tensor-def*
apply (*rule someI2-ex*[**where** *P*=‹λ𝔈𝔉. *bounded-clinear* 𝔈𝔉 ∧ (∀ϱ σ. 𝔈𝔉 (*tc-tensor* ϱ σ) = *tc-tensor* (𝔈 ϱ) (𝔉 σ))›])
using *assms*
by (*simp-all add*: *km-tensor-exists-def*)

lemma *km-tensor-unique*:
assumes ‹*bounded-clinear* 𝔈𝔉›
assumes ‹⋀ϱ σ. 𝔈𝔉 (*tc-tensor* ϱ σ) = *tc-tensor* (𝔈 ϱ) (𝔉 σ)›
shows ‹𝔈𝔉 = *km-tensor* 𝔈 𝔉›
proof −
define *P* **where** ‹*P* 𝔈𝔉 ⟷ *bounded-clinear* 𝔈𝔉 ∧ (∀ϱ σ. 𝔈𝔉 (*tc-tensor* ϱ σ) = *tc-tensor* (𝔈 ϱ) (𝔉 σ))› **for** 𝔈𝔉
have ‹*P* 𝔈𝔉›
using *P-def assms* **by** *presburger*
then have ‹*km-tensor-exists* 𝔈 𝔉›
using *P-def km-tensor-exists-def* **by** *blast*
with ‹*P* 𝔈𝔉› **have** *Ptensor*: ‹*P* (*km-tensor* 𝔈 𝔉)›
by (*simp add*: *P-def*)
show *?thesis*
apply (*rule eq-from-separatingI2*)
apply (*rule separating-set-bounded-clinear-tc-tensor*)
using *assms Ptensor* **by** (*simp-all add*: *P-def*)
qed

lemma *km-tensor-kf-tensor*: ‹*km-tensor* (*kf-apply* 𝔈) (*kf-apply* 𝔉) = *kf-apply* (*kf-tensor* 𝔈 𝔉)›
by (*metis kf-apply-bounded-clinear kf-apply-tensor km-tensor-unique*)

lemma *km-tensor-kraus-map*:
assumes ‹*kraus-map* 𝔈› **and** ‹*kraus-map* 𝔉›
shows ‹*kraus-map* (*km-tensor* 𝔈 𝔉)›
proof −
from *assms* **obtain** *EE* :: ‹(-,-,*unit*) *kraus-family*› **where** *EE*: ‹𝔈 = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
from *assms* **obtain** *FF* :: ‹(-,-,*unit*) *kraus-family*› **where** *FF*: ‹𝔉 = *kf-apply FF*›
using *kraus-map-def-raw* **by** *blast*
show *?thesis*
by (*simp add*: *EE FF km-tensor-kf-tensor*)
qed

lemma *km-tensor-kraus-map-exists*:

assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹*kraus-map* $\mathfrak{F}$›
shows ‹*km-tensor-exists* $\mathfrak{E}$ $\mathfrak{F}$›
proof −
from *assms* **obtain** *EE* :: ‹(-,-,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
from *assms* **obtain** *FF* :: ‹(-,-,*unit*) *kraus-family*› **where** *FF*: ‹$\mathfrak{F}$ = *kf-apply FF*›
using *kraus-map-def-raw* **by** *blast*
show *?thesis*
using *EE FF kf-apply-bounded-clinear kf-apply-tensor km-tensor-exists-def* **by** *blast*
qed

lemma *km-tensor-as-infsum*:
assumes ‹$\bigwedge\varrho$. ((λ*i*. *sandwich-tc* (*E i*) ϱ) *has-sum* $\mathfrak{E}$ ϱ) *I*›
assumes ‹$\bigwedge\varrho$. ((λ*j*. *sandwich-tc* (*F j*) ϱ) *has-sum* $\mathfrak{F}$ ϱ) *J*›
shows ‹*km-tensor* $\mathfrak{E}$ $\mathfrak{F}$ ϱ = ($\sum_\infty$(*i*,*j*)∈*I*×*J*. *sandwich-tc* (*E i* ⊗$_o$ *F j*) ϱ)›
proof −
define *EE FF* **where** ‹*EE* = *Set.filter* (λ(*E*,-). *E*≠*0*) ((λ*a*. (*E a*, *a*)) ‘ *I*)› **and** ‹*FF* = *Set.filter* (λ(*E*,-). *E*≠*0*) ((λ*a*. (*F a*, *a*)) ‘ *J*)›
then have [*simp*]: ‹*kraus-family EE*› ‹*kraus-family FF*›
using *assms kraus-map-sum-kraus-family*
by *blast*+
have ‹$\mathfrak{E}$ = *kf-apply* (*Abs-kraus-family EE*)› **and** ‹$\mathfrak{F}$ = *kf-apply* (*Abs-kraus-family FF*)›
using *assms kraus-map-sum-kf-apply EE-def FF-def*
by *blast*+
then have ‹*km-tensor* $\mathfrak{E}$ $\mathfrak{F}$ ϱ = *kf-apply* (*kf-tensor* (*Abs-kraus-family EE*) (*Abs-kraus-family FF*)) ϱ›
by (*simp add*: *km-tensor-kf-tensor*)
also have ‹... = ($\sum_\infty$((*E*, *x*), (*F*, *y*))∈*EE* × *FF*. *sandwich-tc* (*E* ⊗$_o$ *F*) ϱ)›
by (*simp add*: *kf-apply-tensor-as-infsum Abs-kraus-family-inverse*)
also have ‹... = ($\sum_\infty$((*E*, *x*), (*F*, *y*))∈(λ(*i*,*j*). ((*E i*, *i*), (*F j*, *j*)))‘(*I*×*J*). *sandwich-tc* (*E* ⊗$_o$ *F*) ϱ)›
apply (*rule infsum-cong-neutral*)
by (*auto simp*: *EE-def FF-def*)
also have ‹... = ($\sum_\infty$(*i*,*j*)∈*I*×*J*. *sandwich-tc* (*E i* ⊗$_o$ *F j*) ϱ)›
by (*simp add*: *infsum-reindex inj-on-def o-def case-prod-unfold*)
finally show *?thesis*
by −
qed

lemma *km-bound-tensor*:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹*kraus-map* $\mathfrak{F}$›
shows ‹*km-bound* (*km-tensor* $\mathfrak{E}$ $\mathfrak{F}$) = *km-bound* $\mathfrak{E}$ ⊗$_o$ *km-bound* $\mathfrak{F}$›
proof −
from *assms* **obtain** *EE* :: ‹(-,-,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
from *assms* **obtain** *FF* :: ‹(-,-,*unit*) *kraus-family*› **where** *FF*: ‹$\mathfrak{F}$ = *kf-apply FF*›
using *kraus-map-def-raw* **by** *blast*
show *?thesis*
by (*simp add*: *EE FF km-tensor-kf-tensor kf-bound-tensor km-bound-kf-bound*)

qed

lemma *km-norm-tensor*:
assumes ‹*kraus-map* $\mathfrak{E}$› **and** ‹*kraus-map* $\mathfrak{F}$›
shows ‹*km-norm* (*km-tensor* $\mathfrak{E}$ $\mathfrak{F}$) = *km-norm* $\mathfrak{E}$ ∗ *km-norm* $\mathfrak{F}$›
proof −
from *assms* **obtain** *EE* :: ‹(-,-,*unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
from *assms* **obtain** *FF* :: ‹(-,-,*unit*) *kraus-family*› **where** *FF*: ‹$\mathfrak{F}$ = *kf-apply FF*›
using *kraus-map-def-raw* **by** *blast*
show *?thesis*
by (*simp add*: *EE FF km-tensor-kf-tensor kf-norm-tensor km-norm-kf-norm*)
qed

lemma *km-tensor-compose-distrib*:
assumes ‹*km-tensor-exists* $\mathfrak{E}$ $\mathfrak{G}$› **and** ‹*km-tensor-exists* $\mathfrak{F}$ $\mathfrak{H}$›
shows ‹*km-tensor* ($\mathfrak{E}$ *o* $\mathfrak{F}$) ($\mathfrak{G}$ *o* $\mathfrak{H}$) = *km-tensor* $\mathfrak{E}$ $\mathfrak{G}$ *o km-tensor* $\mathfrak{F}$ $\mathfrak{H}$›
by (*smt* (*verit*, *del-insts*) *assms*(*1*,*2*) *comp-bounded-clinear km-tensor-exists-def km-tensor-unique o-apply*)

lemma *kraus-map-tensor-right*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*kraus-map* ($\lambda\sigma$. *tc-tensor* σ ϱ)›
apply (*rule kraus-mapI*[*of* - ‹*kf-tensor-right* ϱ›])
by (*auto intro*!: *ext simp*: *kf-apply-tensor-right assms*)
lemma *kraus-map-tensor-left*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*kraus-map* ($\lambda\sigma$. *tc-tensor* ϱ σ)›
apply (*rule kraus-mapI*[*of* - ‹*kf-tensor-left* ϱ›])
by (*auto intro*!: *ext simp*: *kf-apply-tensor-left assms*)

lemma *km-bound-tensor-right*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*km-bound* ($\lambda\sigma$. *tc-tensor* σ ϱ) = *norm* ϱ $*_C$ *id-cblinfun*›
apply (*subst km-bound-kf-bound*)
apply (*rule ext*)
apply (*subst kf-apply-tensor-right*[*OF assms*])
by (*auto intro*!: *simp*: *kf-bound-tensor-right assms*)
lemma *km-bound-tensor-left*[*simp*]:
assumes ‹$\varrho \geq 0$›
shows ‹*km-bound* ($\lambda\sigma$. *tc-tensor* ϱ σ) = *norm* ϱ $*_C$ *id-cblinfun*›
apply (*subst km-bound-kf-bound*)
apply (*rule ext*)
apply (*subst kf-apply-tensor-left*[*OF assms*])
by (*auto intro*!: *simp*: *kf-bound-tensor-left assms*)

lemma *kf-norm-tensor-right*[*simp*]:
assumes ‹$\varrho \geq 0$›

shows ‹*km-norm* (λσ. *tc-tensor* σ ϱ) = *norm* ϱ›
by (*simp add*: *km-norm-def km-bound-tensor-right assms*)

lemma *kf-norm-tensor-left*[*simp*]:
assumes ‹ϱ ≥ 0›
shows ‹*km-norm* (λσ. *tc-tensor* ϱ σ) = *norm* ϱ›
by (*simp add*: *km-norm-def km-bound-tensor-left assms*)

lemma *km-operators-in-tensor*:
assumes ‹*km-operators-in* 𝔈 *S*›
assumes ‹*km-operators-in* 𝔉 *T*›
shows ‹*km-operators-in* (*km-tensor* 𝔈 𝔉) (*span* {*s* ⊗ₒ *t* | *s t*. *s*∈*S* ∧ *t*∈*T*})›
proof −
have [*iff*]: ‹*inj-on* (λ((),()). ()) *X*› **for** *X*
by (*simp add*: *inj-on-def*)
from *assms* **obtain** 𝔈′ :: ‹(-,-,*unit*) *kraus-family*› **where** 𝔈-*def*: ‹𝔈 = *kf-apply* 𝔈′› **and** 𝔈′*S*: ‹*kf-operators* 𝔈′ ⊆ *S*›
by (*metis km-operators-in-def*)
from *assms* **obtain** 𝔉′ :: ‹(-,-,*unit*) *kraus-family*› **where** 𝔉-*def*: ‹𝔉 = *kf-apply* 𝔉′› **and** 𝔉′*T*: ‹*kf-operators* 𝔉′ ⊆ *T*›
by (*metis km-operators-in-def*)
define 𝔈𝔉 **where** ‹𝔈𝔉 = *kf-map-inj* (λ((),()). ()) (*kf-tensor* 𝔈′ 𝔉′)›
then have ‹*kf-operators* 𝔈𝔉 = *kf-operators* (*kf-tensor* 𝔈′ 𝔉′)›
by (*simp add*: 𝔈𝔉-*def*)
also have ‹*kf-operators* (*kf-tensor* 𝔈′ 𝔉′) ⊆ *span* {*E* ⊗ₒ *F* | *E F*. *E* ∈ *kf-operators* 𝔈′ ∧ *F* ∈ *kf-operators* 𝔉′}›
using *kf-operators-tensor* **by** *force*
also have ‹... ⊆ *span* {*E* ⊗ₒ *F* | *E F*. *E* ∈ *S* ∧ *F* ∈ *T*}›
by (*smt* (*verit*) *Collect-mono-iff* 𝔈′*S* 𝔉′*T span-mono subset-iff*)
finally have ‹*kf-operators* 𝔈𝔉 ⊆ *span* {*s* ⊗ₒ *t* |*s t*. *s* ∈ *S* ∧ *t* ∈ *T*}›
using 𝔈𝔉-*def* **by** *blast*
moreover have ‹*kf-apply* 𝔈𝔉 = *km-tensor* 𝔈 𝔉›
by (*simp add*: 𝔈𝔉-*def* 𝔈-*def* 𝔉-*def kf-apply-bounded-clinear kf-apply-tensor km-tensor-unique*)
ultimately show *?thesis*
by (*auto intro*!: *exI*[*of* - 𝔈𝔉] *simp add*: *km-operators-in-def*)
qed

lemma *km-tensor-sandwich-tc*:
‹*km-tensor* (*sandwich-tc A*) (*sandwich-tc B*) = *sandwich-tc* (*A* ⊗ₒ *B*)›
by (*metis bounded-clinear-sandwich-tc km-tensor-unique sandwich-tc-tensor*)

3.6 Trace and partial trace

definition ‹*km-trace-preserving* 𝔈 ⟷ (∃𝔉::(-,-,*unit*) *kraus-family*. 𝔈 = *kf-apply* 𝔉 ∧ *kf-trace-preserving* 𝔉)›
lemma *km-trace-preserving-def'*: ‹*km-trace-preserving* 𝔈 ⟷ (∃𝔉::(-, -, *'c*) *kraus-family*. 𝔈 = *kf-apply* 𝔉 ∧ *kf-trace-preserving* 𝔉)›
— Has a more general type than *km-trace-preserving-def*
proof (*rule iffI*)

assume ‹*km-trace-preserving* $\mathfrak{E}$›
then obtain $\mathfrak{F}$ :: ‹(-,-,*unit*) *kraus-family*› **where** $\mathfrak{E}\mathfrak{F}$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$› **and** *tp*$\mathfrak{F}$: ‹*kf-trace-preserving* $\mathfrak{F}$›
using *km-trace-preserving-def* **by** *blast*
from $\mathfrak{E}\mathfrak{F}$ **have** ‹$\mathfrak{E}$ = *kf-apply* (*kf-map* (λ-. *undefined* :: 'c) $\mathfrak{F}$)›
by *simp*
moreover from *tp*$\mathfrak{F}$ **have** ‹*kf-trace-preserving* (*kf-map* (λ-. *undefined* :: 'c) $\mathfrak{F}$)›
by (*metis* $\mathfrak{E}\mathfrak{F}$ *calculation kf-trace-preserving-iff-bound-id km-bound-kf-bound*)
ultimately show ‹∃$\mathfrak{F}$::(-, -, 'c) *kraus-family*. $\mathfrak{E}$ = ($*_{kr}$) $\mathfrak{F}$ ∧ *kf-trace-preserving* $\mathfrak{F}$›
by *blast*
next
assume ‹∃$\mathfrak{F}$::(-, -, 'c) *kraus-family*. $\mathfrak{E}$ = ($*_{kr}$) $\mathfrak{F}$ ∧ *kf-trace-preserving* $\mathfrak{F}$›
then obtain $\mathfrak{F}$ :: ‹(-,-,'c) *kraus-family*› **where** $\mathfrak{E}\mathfrak{F}$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$› **and** *tp*$\mathfrak{F}$: ‹*kf-trace-preserving* $\mathfrak{F}$›
by *blast*
from $\mathfrak{E}\mathfrak{F}$ **have** ‹$\mathfrak{E}$ = *kf-apply* (*kf-flatten* $\mathfrak{F}$)›
by *simp*
moreover from *tp*$\mathfrak{F}$ **have** ‹*kf-trace-preserving* (*kf-flatten* $\mathfrak{F}$)›
by (*metis* $\mathfrak{E}\mathfrak{F}$ *calculation kf-trace-preserving-iff-bound-id km-bound-kf-bound*)
ultimately show ‹*km-trace-preserving* $\mathfrak{E}$›
using *km-trace-preserving-def* **by** *blast*
qed

definition *km-trace-reducing-def*: ‹*km-trace-reducing* $\mathfrak{E}$ ⟷ (∃$\mathfrak{F}$::(-,-,*unit*) *kraus-family*. $\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$ ∧ *kf-trace-reducing* $\mathfrak{F}$)›
lemma *km-trace-reducing-def'*: ‹*km-trace-reducing* $\mathfrak{E}$ ⟷ (∃$\mathfrak{F}$::(-, -, 'c) *kraus-family*. $\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$ ∧ *kf-trace-reducing* $\mathfrak{F}$)›
proof (*rule iffI*)
assume ‹*km-trace-reducing* $\mathfrak{E}$›
then obtain $\mathfrak{F}$ :: ‹(-,-,*unit*) *kraus-family*› **where** $\mathfrak{E}\mathfrak{F}$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$› **and** *tp*$\mathfrak{F}$: ‹*kf-trace-reducing* $\mathfrak{F}$›
using *km-trace-reducing-def* **by** *blast*
from $\mathfrak{E}\mathfrak{F}$ **have** ‹$\mathfrak{E}$ = *kf-apply* (*kf-map* (λ-. *undefined* :: 'c) $\mathfrak{F}$)›
by *simp*
moreover from *tp*$\mathfrak{F}$ **have** ‹*kf-trace-reducing* (*kf-map* (λ-. *undefined* :: 'c) $\mathfrak{F}$)›
by (*simp add*: *kf-trace-reducing-iff-norm-leq1*)
ultimately show ‹∃$\mathfrak{F}$::(-, -, 'c) *kraus-family*. $\mathfrak{E}$ = ($*_{kr}$) $\mathfrak{F}$ ∧ *kf-trace-reducing* $\mathfrak{F}$›
by *blast*
next
assume ‹∃$\mathfrak{F}$::(-, -, 'c) *kraus-family*. $\mathfrak{E}$ = ($*_{kr}$) $\mathfrak{F}$ ∧ *kf-trace-reducing* $\mathfrak{F}$›
then obtain $\mathfrak{F}$ :: ‹(-,-,'c) *kraus-family*› **where** $\mathfrak{E}\mathfrak{F}$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$› **and** *tp*$\mathfrak{F}$: ‹*kf-trace-reducing* $\mathfrak{F}$›
by *blast*
from $\mathfrak{E}\mathfrak{F}$ **have** ‹$\mathfrak{E}$ = *kf-apply* (*kf-flatten* $\mathfrak{F}$)›
by *simp*
moreover from *tp*$\mathfrak{F}$ **have** ‹*kf-trace-reducing* (*kf-flatten* $\mathfrak{F}$)›
by (*simp add*: *kf-trace-reducing-iff-norm-leq1*)
ultimately show ‹*km-trace-reducing* $\mathfrak{E}$›
using *km-trace-reducing-def* **by** *blast*

qed

lemma *km-trace-preserving-apply*[*simp*]: ‹*km-trace-preserving* (*kf-apply* $\mathfrak{E}$) = *kf-trace-preserving* $\mathfrak{E}$›
using *kf-trace-preserving-def km-trace-preserving-def'* **by** *auto*

lemma *km-trace-reducing-apply*[*simp*]: ‹*km-trace-reducing* (*kf-apply* $\mathfrak{E}$) = *kf-trace-reducing* $\mathfrak{E}$›
by (*metis kf-trace-reducing-iff-norm-leq1 km-norm-kf-norm km-trace-reducing-def'*)

lemma *km-trace-preserving-iff*: ‹*km-trace-preserving* $\mathfrak{E}$ ⟷ *kraus-map* $\mathfrak{E}$ ∧ (∀ ϱ. *trace-tc* ($\mathfrak{E}$ ϱ) = *trace-tc* ϱ)›
proof (*intro iffI conjI allI*)
assume *tp*: ‹*km-trace-preserving* $\mathfrak{E}$›
then obtain $\mathfrak{F}$ :: ‹(-,-,*unit*) *kraus-family*› **where** $\mathfrak{EF}$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$› **and** *tp*$\mathfrak{F}$: ‹*kf-trace-preserving* $\mathfrak{F}$›
by (*metis kf-trace-preserving-def kf-trace-reducing-def km-trace-preserving-def order.refl*)
then show ‹*kraus-map* $\mathfrak{E}$›
by *blast*
from *tp*$\mathfrak{F}$ **show** ‹*trace-tc* ($\mathfrak{E}$ ϱ) = *trace-tc* ϱ› **for** ϱ
by (*simp add*: $\mathfrak{EF}$ *kf-trace-preserving-def*)
next
assume *asm*: ‹*kraus-map* $\mathfrak{E}$ ∧ (∀ ϱ. *trace-tc* ($\mathfrak{E}$ ϱ) = *trace-tc* ϱ)›
then obtain $\mathfrak{F}$ :: ‹(-,-,*unit*) *kraus-family*› **where** $\mathfrak{EF}$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$›
using *kraus-map-def-raw* **by** *blast*
from *asm* $\mathfrak{EF}$ **have** ‹*trace-tc* (*kf-apply* $\mathfrak{F}$ ϱ) = *trace-tc* ϱ› **for** ϱ
by *blast*
then have ‹*kf-trace-preserving* $\mathfrak{F}$›
using *kf-trace-preserving-def* **by** *blast*
with $\mathfrak{EF}$ **show** ‹*km-trace-preserving* $\mathfrak{E}$›
using *km-trace-preserving-def* **by** *blast*
qed

lemma *km-trace-reducing-iff*: ‹*km-trace-reducing* $\mathfrak{E}$ ⟷ *kraus-map* $\mathfrak{E}$ ∧ (∀ $\varrho{\geq}0$. *trace-tc* ($\mathfrak{E}$ ϱ) ≤ *trace-tc* ϱ)›
proof (*intro iffI conjI allI impI*)
assume *tp*: ‹*km-trace-reducing* $\mathfrak{E}$›
then obtain $\mathfrak{F}$ :: ‹(-,-,*unit*) *kraus-family*› **where** $\mathfrak{EF}$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$› **and** *tp*$\mathfrak{F}$: ‹*kf-trace-reducing* $\mathfrak{F}$›
by (*metis kf-trace-reducing-def kf-trace-reducing-def km-trace-reducing-def order.refl*)
then show ‹*kraus-map* $\mathfrak{E}$›
by *blast*
from *tp*$\mathfrak{F}$ $\mathfrak{EF}$ **show** ‹*trace-tc* ($\mathfrak{E}$ ϱ) ≤ *trace-tc* ϱ› **if** ‹$\varrho \geq 0$› **for** ϱ
using *kf-trace-reducing-def that* **by** *blast*
next
assume *asm*: ‹*kraus-map* $\mathfrak{E}$ ∧ (∀ $\varrho{\geq}0$. *trace-tc* ($\mathfrak{E}$ ϱ) ≤ *trace-tc* ϱ)›
then obtain $\mathfrak{F}$ :: ‹(-,-,*unit*) *kraus-family*› **where** $\mathfrak{EF}$: ‹$\mathfrak{E}$ = *kf-apply* $\mathfrak{F}$›
using *kraus-map-def-raw* **by** *blast*
from *asm* $\mathfrak{EF}$ **have** ‹*trace-tc* (*kf-apply* $\mathfrak{F}$ ϱ) ≤ *trace-tc* ϱ› **if** ‹$\varrho \geq 0$› **for** ϱ

```
    using that by blast
  then have ‹kf-trace-reducing 𝔉›
    using kf-trace-reducing-def by blast
  with 𝔈𝔉 show ‹km-trace-reducing 𝔈›
    using km-trace-reducing-def by blast
qed

lemma km-trace-preserving-imp-reducing:
  assumes ‹km-trace-preserving 𝔈›
  shows ‹km-trace-reducing 𝔈›
  using assms km-trace-preserving-iff km-trace-reducing-iff by fastforce

lemma km-trace-preserving-id[iff]: ‹km-trace-preserving id›
  by (simp add: km-trace-preserving-iff)

lemma km-trace-reducing-iff-norm-leq1: ‹km-trace-reducing 𝔈 ⟷ kraus-map 𝔈 ∧ km-norm 𝔈
≤ 1›
proof (intro iffI conjI)
  assume ‹km-trace-reducing 𝔈›
  then show ‹kraus-map 𝔈›
    using km-trace-reducing-iff by blast
  then obtain EE :: ‹('a,'b,unit) kraus-family› where EE: ‹𝔈 = kf-apply EE›
    using kraus-map-def-raw by blast
  with ‹km-trace-reducing 𝔈›
  have ‹kf-trace-reducing EE›
    using kf-trace-reducing-def km-trace-reducing-iff by blast
  then have ‹kf-norm EE ≤ 1›
    using kf-trace-reducing-iff-norm-leq1 by blast
  then show ‹km-norm 𝔈 ≤ 1›
    by (simp add: EE km-norm-kf-norm)
next
  assume asm: ‹kraus-map 𝔈 ∧ km-norm 𝔈 ≤ 1›
  then obtain EE :: ‹('a,'b,unit) kraus-family› where EE: ‹𝔈 = kf-apply EE›
    using kraus-map-def-raw by blast
  from asm have ‹km-norm 𝔈 ≤ 1›
    by blast
  then have ‹kf-norm EE ≤ 1›
    by (simp add: EE km-norm-kf-norm)
  then have ‹kf-trace-reducing EE›
    by (simp add: kf-trace-reducing-iff-norm-leq1)
  then show ‹km-trace-reducing 𝔈›
    using EE km-trace-reducing-def by blast
qed


lemma km-trace-preserving-iff-bound-id: ‹km-trace-preserving 𝔈 ⟷ kraus-map 𝔈 ∧ km-bound
𝔈 = id-cblinfun›
proof (intro iffI conjI)
  assume ‹km-trace-preserving 𝔈›
```

then show ‹*kraus-map* 𝔈›
using *km-trace-preserving-iff* **by** *blast*
then obtain *EE* :: ‹('*a*,'*b*,*unit*) *kraus-family*› **where** *EE*: ‹𝔈 = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
with ‹*km-trace-preserving* 𝔈›
have ‹*kf-trace-preserving EE*›
using *kf-trace-preserving-def km-trace-preserving-iff* **by** *blast*
then have ‹*kf-bound EE* = *id-cblinfun*›
by (*simp add*: *kf-trace-preserving-iff-bound-id*)
then show ‹*km-bound* 𝔈 = *id-cblinfun*›
by (*simp add*: *EE km-bound-kf-bound*)
next
assume *asm*: ‹*kraus-map* 𝔈 ∧ *km-bound* 𝔈 = *id-cblinfun*›
then have ‹*kraus-map* 𝔈›
by *simp*
then obtain *EE* :: ‹('*a*,'*b*,*unit*) *kraus-family*› **where** *EE*: ‹𝔈 = *kf-apply EE*›
using *kraus-map-def-raw* **by** *blast*
from *asm* **have** ‹*kf-bound EE* = *id-cblinfun*›
by (*simp add*: *EE km-bound-kf-bound*)
then have ‹*kf-trace-preserving EE*›
by (*simp add*: *kf-trace-preserving-iff-bound-id*)
then show ‹*km-trace-preserving* 𝔈›
using *EE km-trace-preserving-def* **by** *blast*
qed

lemma *km-trace-preserving-iff-bound-id'*:
fixes 𝔈 :: ‹('*a*::{*chilbert-space*, *not-singleton*}, '*a*) *trace-class* ⇒ -›
shows ‹*km-trace-preserving* 𝔈 ⟷ *km-bound* 𝔈 = *id-cblinfun*›
using *km-bound-invalid km-trace-preserving-iff-bound-id* **by** *fastforce*

lemma *km-trace-norm-preserving*: ‹*km-norm* 𝔈 ≤ *1*› **if** ‹*km-trace-preserving* 𝔈›
using *km-trace-preserving-imp-reducing km-trace-reducing-iff-norm-leq1 that* **by** *blast*

lemma *km-trace-norm-preserving-eq*:
fixes 𝔈 :: ‹('*a*::{*chilbert-space*,*not-singleton*},'*a*) *trace-class* ⇒ ('*b*::*chilbert-space*,'*b*) *trace-class*›
assumes ‹*km-trace-preserving* 𝔈›
shows ‹*km-norm* 𝔈 = *1*›
using *assms*
by (*simp add*: *km-trace-preserving-iff-bound-id' km-norm-def*)

lemma *kraus-map-trace*: ‹*kraus-map* (*one-dim-iso o trace-tc*)›
by (*auto intro*!: *ext kraus-mapI*[*of* - ‹*kf-trace some-chilbert-basis*›]
simp: *kf-trace-is-trace*)

lemma *trace-preserving-trace-kraus-map*[*iff*]: ‹*km-trace-preserving* (*one-dim-iso o trace-tc*)›
using *km-trace-preserving-iff kraus-map-trace* **by** *fastforce*

lemma *km-trace-bound*[*simp*]: ‹*km-bound* (*one-dim-iso o trace-tc*) = *id-cblinfun*›

using *km-trace-preserving-iff-bound-id* **by** *blast*

lemma *km-trace-norm-eq1*[*simp*]: ‹*km-norm* (*one-dim-iso o trace-tc* :: (*'a*::{*chilbert-space*,*not-singleton*},*'a*) *trace-class* ⇒ -) = *1*›
using *km-trace-norm-preserving-eq* **by** *blast*

lemma *km-trace-norm-leq1*[*simp*]: ‹*km-norm* (*one-dim-iso o trace-tc*) ≤ *1*›
using *km-trace-norm-preserving* **by** *blast*

lemma *kraus-map-partial-trace*[*iff*]: ‹*kraus-map partial-trace*›
by (*auto intro*!: *ext kraus-mapI*[*of* - ‹*kf-partial-trace-right*›] *simp flip*: *partial-trace-is-kf-partial-trace*)

lemma *partial-trace-ignores-kraus-map*:
assumes ‹*km-trace-preserving* 𝔈›
assumes ‹*kraus-map* 𝔉›
shows ‹*partial-trace* (*km-tensor* 𝔉 𝔈 *ϱ*) = 𝔉 (*partial-trace ϱ*)›
proof −
from *assms*
obtain *EE* :: ‹(-,-,*unit*) *kraus-family*› **where** *EE-def*: ‹𝔈 = *kf-apply EE*› **and** *tpEE*: ‹*kf-trace-preserving EE*›
using *km-trace-preserving-def* **by** *blast*
obtain *FF* :: ‹(-,-,*unit*) *kraus-family*› **where** *FF-def*: ‹𝔉 = *kf-apply FF*›
using *assms*(*2*) *kraus-map-def-raw* **by** *blast*
have ‹*partial-trace* (*km-tensor* 𝔉 𝔈 *ϱ*) = *partial-trace* (*kf-tensor FF EE* ∗$_{kr}$ *ϱ*)›
using *assms*
by (*simp add*: *km-trace-preserving-def partial-trace-ignores-kraus-family*
km-tensor-kf-tensor EE-def kf-id-apply[*abs-def*] *id-def FF-def*
flip: *km-tensor-kf-tensor*)
also have ‹... = *FF* ∗$_{kr}$ *partial-trace ϱ*›
by (*simp add*: *partial-trace-ignores-kraus-family tpEE*)
also have ‹... = 𝔉 (*partial-trace ϱ*)›
using *FF-def* **by** *presburger*
finally show *?thesis*
by −
qed

lemma *km-partial-trace-bound*[*simp*]: ‹*km-bound partial-trace* = *id-cblinfun*›
apply (*subst km-bound-kf-bound*[*of* - *kf-partial-trace-right*])
using *partial-trace-is-kf-partial-trace* **by** *auto*

lemma *km-partial-trace-norm*[*simp*]:
shows ‹*km-norm partial-trace* = *1*›
by (*simp add*: *km-norm-def*)

lemma *km-trace-preserving-tensor*:
assumes ‹*km-trace-preserving* 𝔈› **and** ‹*km-trace-preserving* 𝔉›
shows ‹*km-trace-preserving* (*km-tensor* 𝔈 𝔉)›

proof −
from *assms* **obtain** *EE* :: ‹('*a ell2*, '*b ell2*, *unit*) *kraus-family*› **where** *EE*: ‹$\mathfrak{E}$ = *kf-apply EE*›
and *tE*: ‹*kf-trace-preserving EE*›
using *km-trace-preserving-def* **by** *blast*
from *assms* **obtain** *FF* :: ‹('*c ell2*, '*d ell2*, *unit*) *kraus-family*› **where** *FF*: ‹$\mathfrak{F}$ = *kf-apply FF*›
and *tF*: ‹*kf-trace-preserving FF*›
using *km-trace-preserving-def* **by** *blast*
show *?thesis*
by (*auto intro*!: *kf-trace-preserving-tensor simp*: *EE FF km-tensor-kf-tensor tE tF*)
qed

lemma *km-trace-reducing-tensor*:
assumes ‹*km-trace-reducing* $\mathfrak{E}$› **and** ‹*km-trace-reducing* $\mathfrak{F}$›
shows ‹*km-trace-reducing* (*km-tensor* $\mathfrak{E}$ $\mathfrak{F}$)›
by (*smt* (*z3*) *assms*(*1*,*2*) *km-norm-geq0 km-norm-tensor km-tensor-kraus-map km-trace-reducing-iff-norm-leq1*
mult-left-le-one-le)

3.7 Complete measurements

definition ‹*km-complete-measurement B* ϱ = ($\sum_\infty$*x*∈*B*. *sandwich-tc* (*selfbutter* (*sgn x*)) ϱ)›
abbreviation ‹*km-complete-measurement-ket* ≡ *km-complete-measurement* (*range ket*)›

lemma *km-complete-measurement-kf-complete-measurement*: ‹*km-complete-measurement B* =
kf-apply (*kf-complete-measurement B*)› **if** ‹*is-ortho-set B*›
by (*simp add*: *kf-complete-measurement-apply*[*OF that*, *abs-def*] *km-complete-measurement-def*[*abs-def*])

lemma *km-complete-measurement-ket-kf-complete-measurement-ket*: ‹*km-complete-measurement-ket*
= *kf-apply kf-complete-measurement-ket*›
by (*metis Complex-L2.is-ortho-set-ket kf-apply-map kf-complete-measurement-ket-kf-map kf-eq-imp-eq-weak*
kf-eq-weak-def
km-complete-measurement-kf-complete-measurement)

lemma *km-complete-measurement-has-sum*:
assumes ‹*is-ortho-set B*›
shows ‹((λ*x*. *sandwich-tc* (*selfbutter* (*sgn x*)) ϱ) *has-sum km-complete-measurement B* ϱ) *B*›
using *kf-complete-measurement-has-sum*[*OF assms*] **and** *assms*
by (*simp add*: *kf-complete-measurement-apply km-complete-measurement-def*)

lemma *km-complete-measurement-ket-has-sum*:
‹((λ*x*. *sandwich-tc* (*selfbutter* (*ket x*)) ϱ) *has-sum km-complete-measurement-ket* ϱ) *UNIV*›
by (*smt* (*verit*) *has-sum-cong has-sum-reindex inj-ket is-onb-ket is-ortho-set-ket kf-complete-measurement-apply*
kf-complete-measurement-has-sum-onb km-complete-measurement-def o-def)

lemma *km-bound-complete-measurement*:
assumes ‹*is-ortho-set B*›
shows ‹*km-bound* (*km-complete-measurement B*) ≤ *id-cblinfun*›
apply (*subst km-bound-kf-bound*[*of* - ‹*kf-complete-measurement B*›])

using *assms kf-complete-measurement-apply km-complete-measurement-def* **apply** *fastforce*
by (*simp add*: *assms kf-bound-complete-measurement*)

lemma *km-norm-complete-measurement*:
assumes ‹*is-ortho-set B*›
shows ‹*km-norm* (*km-complete-measurement B*) ≤ *1*›
apply (*subst km-norm-kf-norm*[*of* - ‹*kf-complete-measurement B*›])
apply (*simp add*: *assms km-complete-measurement-kf-complete-measurement*)
by (*simp-all add*: *assms kf-norm-complete-measurement*)

lemma *km-bound-complete-measurement-onb*[*simp*]:
assumes ‹*is-onb B*›
shows ‹*km-bound* (*km-complete-measurement B*) = *id-cblinfun*›
apply (*subst km-bound-kf-bound*[*of* - ‹*kf-complete-measurement B*›])
using *assms*
by (*auto intro*!: *ext simp*: *kf-complete-measurement-apply is-onb-def km-complete-measurement-def*)

lemma *km-bound-complete-measurement-ket*[*simp*]: ‹*km-bound km-complete-measurement-ket* = *id-cblinfun*›
by *fastforce*

lemma *km-norm-complete-measurement-onb*[*simp*]:
fixes *B* :: ‹'*a*::{*not-singleton*, *chilbert-space*} *set*›
assumes ‹*is-onb B*›
shows ‹*km-norm* (*km-complete-measurement B*) = *1*›
apply (*subst km-norm-kf-norm*[*of* - ‹*kf-complete-measurement B*›])
using *assms*
by (*auto intro*!: *ext simp*: *kf-complete-measurement-apply is-onb-def km-complete-measurement-def*)

lemma *km-norm-complete-measurement-ket*[*simp*]:
shows ‹*km-norm km-complete-measurement-ket* = *1*›
by *fastforce*

lemma *kraus-map-complete-measurement*:
assumes ‹*is-ortho-set B*›
shows ‹*kraus-map* (*km-complete-measurement B*)›
apply (*rule kraus-mapI*[*of* - ‹*kf-complete-measurement B*›])
by (*auto intro*!: *ext simp add*: *assms kf-complete-measurement-apply km-complete-measurement-def*)

lemma *kraus-map-complete-measurement-ket*[*iff*]:
shows ‹*kraus-map km-complete-measurement-ket*›
by (*simp add*: *kraus-map-complete-measurement*)

lemma *km-complete-measurement-idem*[*simp*]:
assumes ‹*is-ortho-set B*›
shows ‹*km-complete-measurement B* (*km-complete-measurement B* ϱ) = *km-complete-measurement B* ϱ›
using *kf-complete-measurement-idem*[*of B*]
kf-complete-measurement-apply[*OF assms*] *km-complete-measurement-def*

by (*metis* (*no-types*, *lifting*) *ext kf-comp-apply kf-complete-measurement-idem-weak kf-eq-weak-def o-def*)

lemma *km-complete-measurement-ket-idem*[*simp*]:
‹*km-complete-measurement-ket* (*km-complete-measurement-ket* ϱ) = *km-complete-measurement-ket* ϱ›
by *fastforce*

lemma *km-complete-measurement-has-sum-onb*:
assumes ‹*is-onb B*›
shows ‹((λ*x*. *sandwich-tc* (*selfbutter x*) ϱ) *has-sum km-complete-measurement B* ϱ) *B*›
using *kf-complete-measurement-has-sum-onb*[*OF assms*] **and** *assms*
by (*simp add*: *kf-complete-measurement-apply km-complete-measurement-def is-onb-def*)

lemma *km-complete-measurement-ket-diagonal-operator*[*simp*]:
‹*km-complete-measurement-ket* (*diagonal-operator-tc f*) = *diagonal-operator-tc f*›
using *kf-complete-measurement-ket-diagonal-operator*[*of f*]
by (*metis* (*no-types*, *lifting*) *is-ortho-set-ket kf-apply-map kf-apply-on-UNIV kf-apply-on-eqI kf-complete-measurement-apply*
kf-complete-measurement-ket-kf-map km-complete-measurement-def)

lemma *km-operators-complete-measurement*:
assumes ‹*is-ortho-set B*›
shows ‹*km-operators-in* (*km-complete-measurement B*) (*span* (*selfbutter* ‘ *B*))›
proof −
have ‹*span* ((*selfbutter* ∘ *sgn*) ‘ *B*) ⊆ *span* (*selfbutter* ‘ *B*)›
proof (*intro real-vector.span-minimal*[*OF - real-vector.subspace-span*] *subsetI*)
fix *a*
assume ‹*a* ∈ (*selfbutter* ∘ *sgn*) ‘ *B*›
then obtain *h* **where** ‹*a* = *selfbutter* (*sgn h*)› **and** ‹*h* ∈ *B*›
by *force*
then have ‹*a* = (*inverse* (*norm h*))^2 $*_R$ *selfbutter h*›
by (*simp add*: *sgn-div-norm scaleR-scaleC power2-eq-square*)
with ‹*h* ∈ *B*›
show ‹*a* ∈ *span* (*selfbutter* ‘ *B*)›
by (*simp add*: *span-clauses*(*1*) *span-mul*)
qed
then show *?thesis*
by (*simp add*: *km-complete-measurement-kf-complete-measurement assms km-operators-in-kf-apply*
kf-operators-complete-measurement)
qed

lemma *km-operators-complete-measurement-ket*:
shows ‹*km-operators-in km-complete-measurement-ket* (*span* (*range* (λ*c*. (*selfbutter* (*ket c*)))))›
by (*metis* (*no-types*, *lifting*) *image-cong is-ortho-set-ket km-operators-complete-measurement range-composition*)

lemma *km-complete-measurement-ket-butterket*[*simp*]:

‹*km-complete-measurement-ket* (*tc-butterfly* (*ket c*) (*ket c*)) = *tc-butterfly* (*ket c*) (*ket c*)›
by (*simp add*: *km-complete-measurement-ket-kf-complete-measurement-ket kf-complete-measurement-ket-apply-butte*

lemma *km-complete-measurement-tensor*:
assumes ‹*is-ortho-set B*› **and** ‹*is-ortho-set C*›
shows ‹*km-tensor* (*km-complete-measurement B*) (*km-complete-measurement C*)
= *km-complete-measurement* ((λ(*b*,*c*). *b* ⊗$_s$ *c*) ‘ (*B* × *C*))›
by (*simp add*: *km-complete-measurement-kf-complete-measurement assms is-ortho-set-tensor km-tensor-kf-tensor*
flip: *kf-complete-measurement-tensor*)

lemma *km-complete-measurement-ket-tensor*:
shows ‹*km-tensor* (*km-complete-measurement-ket* :: ('*a ell2*, -) *trace-class* ⇒ -) (*km-complete-measurement-ket* :: ('*b ell2*, -) *trace-class* ⇒ -)
= *km-complete-measurement-ket*›
by (*simp add*: *km-complete-measurement-ket-kf-complete-measurement-ket km-tensor-kf-tensor kf-complete-measurement-ket-tensor*)

lemma *km-tensor-0-left*[*simp*]: ‹*km-tensor* (*0* :: ('*a ell2*, '*b ell2*) *trace-class* ⇒ ('*c ell2*, '*d ell2*) *trace-class*) 𝔈 = *0*›
proof (*cases* ‹*km-tensor-exists* (*0* :: ('*a ell2*, '*b ell2*) *trace-class* ⇒ ('*c ell2*, '*d ell2*) *trace-class*) 𝔈›)
case *True*
then show *?thesis*
apply (*rule-tac eq-from-separatingI2*[*OF separating-set-bounded-clinear-tc-tensor*])
by (*simp-all add*: *km-tensor-apply*)
next
case *False*
then show *?thesis*
using *km-tensor-invalid* **by** *blast*
qed

lemma *km-tensor-0-right*[*simp*]: ‹*km-tensor* 𝔈 (*0* :: ('*a ell2*, '*b ell2*) *trace-class* ⇒ ('*c ell2*, '*d ell2*) *trace-class*) = *0*›
proof (*cases* ‹*km-tensor-exists* 𝔈 (*0* :: ('*a ell2*, '*b ell2*) *trace-class* ⇒ ('*c ell2*, '*d ell2*) *trace-class*)›)
case *True*
then show *?thesis*
apply (*rule-tac eq-from-separatingI2*[*OF separating-set-bounded-clinear-tc-tensor*])
by (*simp-all add*: *km-tensor-apply*)
next
case *False*
then show *?thesis*
using *km-tensor-invalid* **by** *blast*
qed

unbundle *no kraus-map-syntax*

unbundle *no cblinfun-syntax*

end

References

[1] K. Kraus. *States, effects, and operations: fundamental notions of quantum theory*, volume 190 of *LNP*. Springer, 1983. Defines Kraus maps in Section 3, Theorem 1. Only considers separable Hilbert spaces.