The Oneway to Hiding Theorem*

Katharina Heidler   Dominique Unruh

October 13, 2025

Abstract

As the standardization process for post-quantum cryptography progresses, the need for computer-verified security proofs against classical and quantum attackers increases. Even though some tools are already tackling this issue, none are foundational. We take a first step in this direction and present a complete formalization of the One-way to Hiding (O2H) Theorem, a central theorem for security proofs against quantum attackers. With this new formalization, we build more secure foundations for proof-checking tools in the quantum setting. Using the theorem prover Isabelle, we verify the semi-classical O2H Theorem by Ambainis, Hamburg and Unruh (Crypto 2019) in different variations. We also give a novel (and for the formalization simpler) proof to the O2H Theorem for mixed states and extend the theorem to non-terminating adversaries. This work provides a theoretical and foundational background for several verification tools and for security proofs in the quantum setting.

A paper describing this work in more detail appeared at [2].

Contents

*Supported by the research training group ConVeY of the German Research Foundation under grant GRK 2428 and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – NI 491/18-1, by the ERC consolidator grant CerQuS (Certified Quantum Security, 819317), by the Estonian Centre of Excellence in IT (EXCITE, TK148), by the Estonian Centre of Excellence "Foundations of the Universe" (TK202), and by the Estonian Research Council PRG grant "Secure Quantum Technology" (PRG946).

theory *O2H-Additional-Lemmas*
imports *Registers.Pure-States*
begin

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

This theory contains additional lemmas on summability, trace, tensor product, sandwiching operator, arithmetic-quadratic mean inequality, matrices with norm less or equal one, projections and more.

An additional lemma

lemma *abs-summable-on-reindex*:
assumes ‹*inj-on h A*›
shows ‹g *abs-summable-on* $(h$ ‘ $A) \longleftrightarrow (g \circ h)$ *abs-summable-on* A›
proof –
have $(norm \circ g)$ *summable-on* $(h$ ‘ $A) \longleftrightarrow ((norm \circ g) \circ h)$ *summable-on* A
by (*rule summable-on-reindex*[*OF assms*])
then show *?thesis* **unfolding** *comp-def* **by** *auto*
qed

lemma *abs-summable-norm*:
  assumes ‹*f abs-summable-on A*›
  shows ‹(λx. *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 ‹(λx. *f x* + *g x*) *abs-summable-on A*›
proof −
  from *assms* **have** ‹(λx. *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 *sandwich-tc-has-sum*:
  assumes (*f has-sum x*) *A*
  shows ((*sandwich-tc* ϱ *o f*) *has-sum sandwich-tc* ϱ *x*) *A*
  unfolding *o-def* **by** (*intro has-sum-bounded-linear*[*OF - assms*])
    (*auto simp add*: *bounded-clinear.bounded-linear bounded-clinear-sandwich-tc*)

lemma *sandwich-tc-abs-summable-on*:
  assumes *f abs-summable-on A*
  shows (*sandwich-tc* ϱ *o f*) *abs-summable-on A*
  by (*intro abs-summable-on-bounded-linear*)
    (*auto simp add*: *assms bounded-clinear.bounded-linear bounded-clinear-sandwich-tc*)

lemma *trace-tc-abs-summable-on*:
  assumes *f abs-summable-on A*
  shows (*trace-tc o f*) *abs-summable-on A*
 by (*intro abs-summable-on-bounded-linear*) (*auto simp add*: *assms bounded-clinear.bounded-linear*)

Defining a self butterfly on trace class.

lemma *trace-selfbutter-norm*:
  trace (*selfbutter A*) = *norm A* ^*2*
  by (*simp add*: *power2-norm-eq-cinner trace-butterfly*)

definition *tc-selfbutter* **where** *tc-selfbutter a* = *tc-butterfly a a*

lemma *norm-tc-selfbutter*[*simp*]:
  norm (*tc-selfbutter a*) = (*norm a*)^*2*
  unfolding *tc-selfbutter-def* **using** *norm-tc-butterfly* **by** (*metis power2-eq-square*)

lemma *trace-tc-sandwich-tc-isometry*:
  assumes *isometry U*
  shows *trace-tc* (*sandwich-tc U A*) = *trace-tc A*

using *assms* **by** *transfer auto*

lemma *norm-sandwich-tc-unitary*:
assumes *isometry U* $\varrho \geq 0$
shows *norm* (*sandwich-tc U* ϱ) = *norm* ϱ
using *trace-tc-sandwich-tc-isometry*[*OF assms*(*1*)] *assms*
by (*simp add*: *norm-tc-pos-Re sandwich-tc-pos*)

Lemmas on *trace-tc*

lemma *trace-tc-minus*:
trace-tc (*a*−*b*) = *trace-tc a* − *trace-tc b*
by (*metis add-implies-diff diff-add-cancel trace-tc-plus*)

lemma *trace-tc-sum*:
trace-tc (*sum a I*) = ($\sum$ *i*∈*I*. *trace-tc* (*a i*))
by (*simp add*: *from-trace-class-sum trace-sum trace-tc.rep-eq*)

lemma *selfbutter-sandwich*:
fixes *A* :: *'a ell2* $\Rightarrow_{CL}$ *'a ell2* **and** *B* :: *'a ell2*
shows *selfbutter* (*A* $*_V$ *B*) = *sandwich A* $*_V$ *selfbutter B*
by (*simp add*: *butterfly-comp-cblinfun cblinfun-comp-butterfly sandwich-apply*)

lemma *tc-tensor-sum-left*:
tc-tensor (*sum g S*) *x* = ($\sum$ *i*∈*S*. *tc-tensor* (*g i*) *x*)
by *transfer* (*auto simp add*: *tensor-op-cbilinear.sum-left*)

lemma *tc-tensor-sum-right*:
tc-tensor x (*sum g S*) = ($\sum$ *i*∈*S*. *tc-tensor x* (*g i*))
by *transfer* (*auto simp add*: *tensor-op-cbilinear.sum-right*)

lemma *complex-of-real-abs*: *complex-of-real* |*f*| = |*complex-of-real f*|
by (*simp add*: *abs-complex-def*)

Additional Lemmas on Tensors

lemma *tensor-op-padding*:
(*A* $\otimes_o$ *B*) $*_V$ *v* = (*A* $\otimes_o$ *id-cblinfun*) $*_V$ (*id-cblinfun* $\otimes_o$ *B*) $*_V$ *v*
by (*metis cblinfun-apply-cblinfun-compose cblinfun-compose-id-left cblinfun-compose-id-right comp-tensor-op*)

lemma *tensor-op-padding'*:
(*A* $\otimes_o$ *B*) $*_V$ *v* = (*id-cblinfun* $\otimes_o$ *B*) $*_V$ (*A* $\otimes_o$ *id-cblinfun*) $*_V$ *v*
by (*subst cblinfun-apply-cblinfun-compose*[*symmetric*], *subst comp-tensor-op*) *auto*

lemma *tensor-op-left-minus*: (*x* − *y*) $\otimes_o$ *b* = *x* $\otimes_o$ *b* − *y* $\otimes_o$ *b*
by (*metis ordered-field-class.sign-simps*(*10*) *tensor-op-left-add*)

lemma *tensor-op-right-minus*: *b* $\otimes_o$ (*x* − *y*) = *b* $\otimes_o$ *x* − *b* $\otimes_o$ *y*
by (*metis ordered-field-class.sign-simps*(*10*) *tensor-op-right-add*)

lemma *id-cblinfun-selfbutter-tensor-ell2-right*:
id-cblinfun $\otimes_o$ *selfbutter* (*ket i*) = (*tensor-ell2-right* (*ket i*)) o_{CL} (*tensor-ell2-right* (*ket i*)$*$)
by (*intro equal-ket*) (*auto simp add*: *tensor-ell2-ket*[*symmetric*] *tensor-op-ell2*
tensor-ell2-scaleC2 tensor-ell2-scaleC1)

A lot lof lemmas on limit processes with several functions.

lemma *tendsto-Re*:
assumes ($f \longrightarrow x$) F
shows ((λx. *Re* (f x)) $\longrightarrow$ *Re* x) F
using *assms* **by** (*simp add*: *tendsto-Re*)

lemma *tendsto-tc-tensor*:
assumes ($f \longrightarrow x$) F
shows ((λx. *tc-tensor* (f x) a) $\longrightarrow$ *tc-tensor* x a) F
using *bounded-linear.tendsto*[*OF bounded-clinear.bounded-linear*[*OF bounded-clinear-tc-tensor-left*] *assms*]
by *auto*

lemma *tc-tensor-has-sum*:
assumes (f *has-sum* x) A
shows ((λy. *tc-tensor* y a) o f *has-sum* *tc-tensor* x a) A
unfolding *o-def* **using** *has-sum-bounded-linear bounded-clinear-tc-tensor-left*
assms bounded-clinear.bounded-linear **by** *blast*

lemma *Re-has-sum*:
assumes (f *has-sum* x) A
shows ((λn. *Re* n) o f *has-sum* *Re* x) A
by (*metis assms*(*1*) *has-sum-Re has-sum-cong o-def*)

Relationship norm and condition

lemma *norm-1-to-cond*:
fixes A :: *'a ell2* $\Rightarrow_{CL}$ *'a ell2*
assumes *norm* $A \leq 1$
shows $A*$ o_{CL} $A \leq$ *id-cblinfun*
proof $-$
have *norm* ($A *_V \Psi$) $\leq$ *norm* Ψ **for** Ψ :: *'a ell2*
by (*meson assms basic-trans-rules*(*23*) *mult-left-le-one-le norm-cblinfun norm-ge-zero*)
then have *ineq*: *norm* ($A *_V \Psi$)^*2* $\leq$ *norm* Ψ^*2* **for** Ψ :: *'a ell2* **by** *simp*
have *left*: *norm* ($A *_V \Psi$)^*2* = $\Psi \bullet_C$ (($A*$ o_{CL} A) $*_V$ Ψ) **for** Ψ:: *'a ell2*
by (*simp add*: *cdot-square-norm cinner-adj-right*)
have *right*: *norm* Ψ^*2* = $\Psi \bullet_C$ (*id-cblinfun* $*_V$ Ψ) **for** Ψ:: *'a ell2*
by (*simp add*: *cdot-square-norm*)
have $\Psi \bullet_C$ (($A*$ o_{CL} A) $*_V$ Ψ) $\leq \Psi \bullet_C$ (*id-cblinfun* $*_V$ Ψ) **for** Ψ:: *'a ell2*
using *ineq left right* **by** (*simp add*: *cnorm-le*)
then show $A*$ o_{CL} $A \leq$ *id-cblinfun* **using** *cblinfun-leI* **by** *blast*
qed

lemma *cond-to-norm-1*:
  fixes A :: $'a\ ell2 \Rightarrow_{CL} 'a\ ell2$
  assumes $A*\ o_{CL}\ A \le$ *id-cblinfun*
  shows *norm* $A \le 1$
proof –
  have *left*: *norm* $(A *_V \Psi)$^2 $= \Psi \bullet_C ((A*\ o_{CL}\ A) *_V \Psi)$ **for** Ψ:: $'a\ ell2$
    by (*simp add*: *cdot-square-norm cinner-adj-right*)
  have *right*: *norm* Ψ^2 $= \Psi \bullet_C$ (*id-cblinfun* $*_V \Psi$) **for** Ψ:: $'a\ ell2$
    by (*simp add*: *cdot-square-norm*)
  have $\Psi \bullet_C ((A*\ o_{CL}\ A) *_V \Psi) \le \Psi \bullet_C$ (*id-cblinfun* $*_V \Psi$) **for** Ψ:: $'a\ ell2$
    using *assms less-eq-cblinfun-def* **by** *blast*
  then have *ineq*: *norm* $(A *_V \Psi)$^2 $\le$ *norm* Ψ^2 **for** Ψ :: $'a\ ell2$
    by (*metis complex-of-real-mono-iff left right*)
  then have *norm* $(A *_V \Psi) \le$ *norm* Ψ **for** Ψ :: $'a\ ell2$ **by** *simp*
  then show *norm* $A \le 1$ **by** (*simp add*: *norm-cblinfun-bound*)
qed

Arithmetic mean - quadratic mean inequality (AM-QM)

lemma *arith-quad-mean-ineq*:
  fixes n::*nat* **and** x :: *nat* $\Rightarrow$ *real*
  assumes $\bigwedge i.\ i \in I \Longrightarrow x\ i \ge 0$
  shows $(\sum i \in I.\ x\ i)$^2 $\le$ (*card* I) $*$ $(\sum i \in I.\ (x\ i)$ ^2)
  using *Cauchy-Schwarz-ineq-sum*[*of* (λ-. *1*) x I] **by** *auto*

lemma *sqrt-binom*:
  assumes $a \ge 0$ $b \ge 0$
  shows $|a - b| = |$*sqrt* a $-$ *sqrt* $b|$ $*$ $|$*sqrt* a $+$ *sqrt* $b|$
  by (*metis abs-mult abs-pos assms*(*1*) *assms*(*2*) *cross3-simps*(*11*) *real-sqrt-abs2 real-sqrt-mult square-diff-square-factored*)

Lemmas on *sandwich-tc*

lemma *sandwich-tc-compose'*:
  sandwich-tc (A o_{CL} B) ϱ = *sandwich-tc* A (*sandwich-tc* B ϱ)
  using *sandwich-tc-compose* **unfolding** *o-def* **by** *metis*

lemma *sandwich-tc-sum*:
  sandwich-tc E (*sum* f A) = *sum* (*sandwich-tc* E *o* f) A
  by (*metis sandwich-tc-0-right sandwich-tc-plus sum-comp-morphism*)

lemma *isCont-sandwich-tc*:
  isCont (*sandwich-tc* A) x
  by (*intro linear-continuous-at*)
    (*simp add*: *bounded-clinear.bounded-linear bounded-clinear-sandwich-tc*)

lemma *bounded-linear-trace-norm-sandwich-tc*:
  bounded-linear (λy. *trace-tc* (*sandwich-tc* E y))

unfolding *bounded-linear-def* **by** (*metis bounded-clinear.bounded-linear bounded-clinear-sandwich-tc bounded-clinear-trace-tc bounded-linear-compose bounded-linear-def*)

lemma *sandwich-add1*:
sandwich $(A+B)$ C = *sandwich* A C + *sandwich* B C + A o_{CL} C o_{CL} $B*$ + B o_{CL} C o_{CL} $A*$
by (*simp add*: *adj-plus cblinfun-compose-add-left cblinfun-compose-add-right sandwich.rep-eq*)

lemma *sandwich-tc-add1*:
sandwich-tc $(A+B)$ C = *sandwich-tc* A C + *sandwich-tc* B C +
compose-tcl (*compose-tcr* B C) $(A*)$ + *compose-tcl* (*compose-tcr* A C) $(B*)$
unfolding *sandwich-tc-def*
by (*auto simp add*: *compose-tcr.add-left compose-tcl.add-left compose-tcl.add-right adj-plus*)

lemma *sandwich-add2*:
sandwich A $(B+C)$ = *sandwich* A B + *sandwich* A C
by (*simp add*: *cblinfun.add-right*)

On the spaces of projections with and/or or events.

lemma *splitting-Proj-and*:
assumes *is-Proj P is-Proj Q*
shows *Proj* $(((P \otimes_o$ *id-cblinfun*$) *_S \top) \sqcap (($*id-cblinfun* $\otimes_o Q) *_S \top)) = P \otimes_o Q$
proof −
have *tensor1*: $(P \otimes_o$ (*id-cblinfun*::'*b ell2* $\Rightarrow_{CL}$ '*b ell2*)$) *_S \top = (P *_S \top) \otimes_S \top$
and *tensor2*: ((*id-cblinfun*::'*a ell2* $\Rightarrow_{CL}$ '*a ell2*) $\otimes_o Q) *_S \top = \top \otimes_S (Q *_S \top)$
by (*auto simp add*: *Proj-on-own-range assms tensor-ccsubspace-via-Proj*)

have $((P \otimes_o$ *id-cblinfun*$) *_S \top) \sqcap (($*id-cblinfun* $\otimes_o Q) *_S \top) =$
$((P *_S \top) \otimes_S \top) \sqcap (\top \otimes_S (Q *_S \top))$
using *tensor1 tensor2* **by** *auto*
also have ... $= (P *_S \top) \otimes_S (Q *_S \top)$
by (*smt* (*z3*) *Proj-image-leq Proj-o-Proj-subspace-right Proj-on-own-range Proj-range assms boolean-algebra-cancel.inf1 cblinfun-assoc-left*(*2*) *cblinfun-image-id inf.orderE inf-commute inf-le2 is-Proj-id is-Proj-tensor-op isometry-cblinfun-image-inf-distrib' tensor-ccsubspace-image tensor-ccsubspace-top*)
finally have *rew*: $((P \otimes_o$ *id-cblinfun*$) *_S \top) \sqcap (($*id-cblinfun* $\otimes_o Q) *_S \top) =$
$((P \otimes_o Q) *_S \top)$
by (*simp add*: *tensor-ccsubspace-image*)
show *?thesis* **unfolding** *rew*
by (*simp add*: *Proj-on-own-range assms*(*1*) *assms*(*2*) *is-Proj-tensor-op*)
qed

lemma *splitting-Proj-or*:
assumes *is-Proj P is-Proj Q*
shows *Proj* $(((P \otimes_o$ *id-cblinfun*$) *_S \top) \sqcup (($*id-cblinfun* $\otimes_o Q) *_S \top)) =$
$P \otimes_o$ (*id-cblinfun* $- Q$) + *id-cblinfun* $\otimes_o Q$
proof −

have *tensor1*: $(P \otimes_o (\textit{id-cblinfun}::'b\ ell2 \Rightarrow_{CL} 'b\ ell2)) *_S \top = (P *_S \top) \otimes_S \top$
and *tensor2*: $((\textit{id-cblinfun}::'a\ ell2 \Rightarrow_{CL} 'a\ ell2) \otimes_o Q) *_S \top = \top \otimes_S (Q *_S \top)$
by (*auto simp add*: *Proj-on-own-range assms tensor-ccsubspace-via-Proj*)
have $((P \otimes_o \textit{id-cblinfun}) *_S \top) \sqcup ((\textit{id-cblinfun} \otimes_o Q) *_S \top) =$
$((P *_S \top) \otimes_S \top) \sqcup (\top \otimes_S (Q *_S \top))$ **using** *tensor1 tensor2* **by** *auto*
also have $\ldots = ((P *_S \top) \otimes_S (Q *_S \top)) \sqcup ((P *_S \top) \otimes_S - (Q *_S \top)) \sqcup$
$((P *_S \top) \otimes_S (Q *_S \top)) \sqcup (-(P *_S \top) \otimes_S (Q *_S \top))$
by (*smt* (*z3*) *cblinfun-image-id cblinfun-image-sup complemented-lattice-class.sup-compl-top ortho-tensor-ccsubspace-left ortho-tensor-ccsubspace-right sup-aci*(*2*) *tensor1 tensor2 tensor-ccsubspace-image*)
also have $\ldots = ((P *_S \top) \otimes_S - (Q *_S \top)) \sqcup ((P *_S \top) \otimes_S (Q *_S \top)) \sqcup$
$(-(P *_S \top) \otimes_S (Q *_S \top))$ **by** (*simp add*: *inf-sup-aci*(*5*))
also have $\ldots = ((P *_S \top) \otimes_S - (Q *_S \top)) \sqcup (\top \otimes_S (Q *_S \top))$
by (*smt* (*verit*, *del-insts*) *cblinfun-image-id cblinfun-image-sup complemented-lattice-class.compl-sup-top*

inf-sup-aci(*5*) *ortho-tensor-ccsubspace-left sup-aci*(*2*) *tensor2 tensor-ccsubspace-image*)
finally have *rew*: $((P \otimes_o \textit{id-cblinfun}) *_S \top) \sqcup ((\textit{id-cblinfun} \otimes_o Q) *_S \top) =$
$((P \otimes_o (\textit{id-cblinfun} - Q)) *_S \top) \sqcup ((\textit{id-cblinfun} \otimes_o Q) *_S \top)$
by (*simp add*: *Proj-on-own-range assms*(*2*) *range-cblinfun-code-def tensor2 tensor-ccsubspace-image uminus-Span-code*)
have $\textit{Proj}\ (((P \otimes_o \textit{id-cblinfun}) *_S \top) \sqcup ((\textit{id-cblinfun} \otimes_o Q) *_S \top)) =$
$\textit{Proj}\ ((P \otimes_o (\textit{id-cblinfun} - Q)) *_S \top) + \textit{Proj}\ ((\textit{id-cblinfun} \otimes_o Q) *_S \top)$
unfolding *rew* **by** (*intro Proj-sup*) (*smt* (*verit*, *del-insts*) *Proj-image-leq Proj-on-own-range Proj-ortho-compl assms*(*1*) *assms*(*2*) *ortho-tensor-ccsubspace-right orthogonal-spaces-leq-compl*

range-cblinfun-code-def tensor2 tensor-ccsubspace-mono tensor-ccsubspace-via-Proj top-greatest

uminus-Span-code)
also have $\ldots = P \otimes_o (\textit{id-cblinfun} - Q) + \textit{id-cblinfun} \otimes_o Q$
by (*simp add*: *Proj-on-own-range assms*(*1*) *assms*(*2*) *is-Proj-tensor-op*)
finally show *?thesis* **by** *auto*
qed

Additional stuff

lemma *Union-cong*:
assumes $\bigwedge i.\ i{\in}A \Longrightarrow f\ i = g\ i$
shows $(\bigcup i{\in}A.\ f\ i) = (\bigcup\ i{\in}A.\ g\ i)$
using *assms* **by** *auto*

lemma *proj-ket-apply*:
$\textit{proj}\ (\textit{ket}\ i) *_V \textit{ket}\ j = (\textit{if}\ i{=}j\ \textit{then ket}\ i\ \textit{else}\ 0)$
by (*metis Proj-fixes-image ccspan-superset′ insertI1 kernel-Proj kernel-memberD mem-ortho-ccspanI orthogonal-ket singletonD*)

Missing lemmas for Kraus maps

lemma *infsum-in-finite*:
assumes *finite F*
assumes ‹*Hausdorff-space T*›

assumes ‹*sum f F* ∈ *topspace T*›
shows *infsum-in T f F* = *sum f F*
using *has-sum-in-finite*[*OF assms*(*1*,*3*)]
using *assms*(*2*) *has-sum-in-infsum-in has-sum-in-unique summable-on-in-def* **by** *blast*

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

lemma *inj-on-CARD-1*[*iff*]: ‹*inj-on f X*› **for** *X* :: ‹′*a*::*CARD-1 set*›
by (*auto intro*!: *inj-onI*)

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*

end
theory *Definition-O2H*

imports *Registers.Pure-States*
O2H-Additional-Lemmas

begin

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

1 Definitions for the one-way to Hiding (O2H) Lemma

Here, we first define the context of the O2H Lemma and foundations.

First of all, we need a notion of a query to the oracle. This is defined in the unitary *Uquery*, where the input H is the (classical) oracle.

definition *Uquery* :: ‹('x ⇒ ('y::plus)) ⇒ (('x × 'y) ell2 $\Rightarrow_{CL}$ ('x × 'y) ell2)› **where**
Uquery H = classical-operator (Some o (λ(x,y). (x, y + (H x))))

1.1 Locale for the general O2H setting

Locale for O2H assumptions and setting.

locale *o2h-setting* =
— Fix types for instantiations of locales
fixes *type-x* ::'x *itself*
fixes *type-y* :: ('y::group-add) *itself*
fixes *type-mem* :: 'mem *itself*
fixes *type-l* :: 'l *itself*

— X and Y are the embeddings of the (classical) oracle domain types. *'mem* is the type of the quantum memory we work on.
fixes X :: *'x update ⇒ 'mem update*
fixes Y :: *('y::group-add) update ⇒ 'mem update*

— The embeddings X and Y must be compatible with the registers.
assumes *compat[register]*: *mutually compatible* (X,Y)

— We fix the query depth d of A. We ensure that we have queries at least once.
fixes d :: *nat*
assumes *d-gr-0*: $d > 0$
— The initial quantum state *init* of the registers. For this version of the O2H, we work with a pure initial state.
fixes *init* :: ‹'mem ell2›
assumes *norm-init*: *norm init = 1* — *init* is a pure state

— The type *'l* represents the quantum register for the query log. We also need three functions depending on the type *'l*, namely *flip*, *bit* and *valid*. *flip* is a bit-flipping operation that changes bits on the valid set and may behave like an identity function on the rest. *bit* is a function returning the i-th bit of a valid element in *'l*. *valid* is a functional representation of the valid set of the query log. Since *'l* may be (theoretically) infinitely large, we need to restrict on the valid set in many lemmas.

fixes *flip*:: ‹nat ⇒ 'l ⇒ 'l›
fixes *bit*:: ‹'l ⇒ nat ⇒ bool›
fixes *valid*:: ‹'l ⇒ bool›
fixes *empty* :: ‹'l›

— Empty is the initial state on $'l$ (equalling the zero state).
assumes *valid-empty*: *valid empty*

— Assumptions on *flip*, *bit* and *valid*: *flip* is a function that takes an index i and an element $l{::}'l$ and "flips the i-th bit". However, to remain in the valid range, this flip is only performed for indices smaller than d, otherwise we may assume *flip* to be the identity.
assumes *valid-flip*: $i<d \Longrightarrow valid\ l \Longrightarrow valid\ (flip\ i\ l)$
— The *flip* operation must be idempotent.
assumes *inj-flip*: $inj\ (flip\ i)$
assumes *valid-flip-flip*: $i<d \Longrightarrow valid\ l \Longrightarrow flip\ i\ (flip\ i\ l) = l$
— The *flip* operation must be commutative with itself.
assumes *valid-flip-comm*: $i<d \Longrightarrow j<d \Longrightarrow valid\ l \Longrightarrow flip\ i\ (flip\ j\ l) = flip\ j\ (flip\ i\ l)$

— For valid elements, the bits in the range up to d behave as in a normal bit-flipping operation.
assumes *valid-bit-flip-same*: $i<d \Longrightarrow valid\ l \Longrightarrow bit\ (flip\ i\ l)\ i = (\neg\ (bit\ l\ i))$
assumes *valid-bit-flip-diff*: $i<d \Longrightarrow valid\ l \Longrightarrow i\neq j \Longrightarrow bit\ (flip\ i\ l)\ j = bit\ l\ j$

begin

We introduce a set of 2^d valid elements for counting. Since we need a finite set for easier proofs while counting the adversarial queries, we embed the set of 2^d elements into the valid set. The elements from *blog* can all be derived by flipping bits from the initial empty state. We then only look at the elements with bits in the first d entries.

inductive *blog* :: $'l \Rightarrow bool$ **where**
blog empty
| $blog\ l \Longrightarrow i<d \Longrightarrow blog\ (flip\ i\ l)$

lemma *blog-empty*: *blog empty*
by (*rule blog.intros*)

lemma *blog-flip*: $i<d \Longrightarrow blog\ l \Longrightarrow blog\ (flip\ i\ l)$
by (*rule blog.intros*)

lemma *blog-valid*:
$blog\ l \Longrightarrow valid\ l$
by (*induction rule*: *blog.induct*) (*auto simp add*: *valid-empty valid-flip*)

lemma *flip-flip*: $i<d \Longrightarrow blog\ l \Longrightarrow flip\ i\ (flip\ i\ l) = l$
using *blog-valid valid-flip-flip* **by** *auto*

lemma *bit-flip-same*: $i<d \Longrightarrow blog\ l \Longrightarrow bit\ (flip\ i\ l)\ i = (\neg\ (bit\ l\ i))$
using *blog-valid valid-bit-flip-same* **by** *auto*

lemma *bit-flip-diff*: $i<d \Longrightarrow blog\ l \Longrightarrow i\neq j \Longrightarrow bit\ (flip\ i\ l)\ j = bit\ l\ j$

using *blog-valid valid-bit-flip-diff* **by** *auto*

The embedding of a boolean list (of length d) into the *blog* set.

fun *list-to-l* :: *bool list* ⇒ *'l* **where**
list-to-l [] = *empty* |
list-to-l (*False* # *list*) = *list-to-l list* |
list-to-l (*True* # *list*) = *flip* (*length list*) (*list-to-l list*)

definition *len-d-lists* :: *bool list set* **where**
len-d-lists = {*xs*. *length xs* = *d*}

lemma *card-len-d-lists*:
card (*len-d-lists*) = (*2*::*nat*)^*d*
proof −
have *l*: *len-d-lists* = {*xs*. *set xs* ⊆ {*True*, *False*} ∧ *length xs* = *d*}
unfolding *len-d-lists-def* **by** *auto*
show *?thesis* **unfolding** *l*
by (*subst card-lists-length-eq*[*of* {*True*, *False*}])(*auto simp add*: *numeral-2-eq-2*)
qed

lemma *finite-len-d-lists*[*simp*]:
finite len-d-lists
using *card-len-d-lists card.infinite* **by** *force*

lemma *blog-list-to-l*:
assumes *length ls* ≤ *d*
shows *blog* (*list-to-l ls*)
using *assms* **by** (*induction rule*: *list-to-l.induct*) (*auto simp add*: *blog.intros*)

lemma *flip-commute*:
assumes *i*≠*j* *i*<*d* *j*<*d* *length ls* ≤ *d*
shows *flip i* (*flip j* (*list-to-l ls*)) = *flip j* (*flip i* (*list-to-l ls*))
by (*simp add*: *assms*(*2*) *assms*(*3*) *assms*(*4*) *blog-list-to-l blog-valid valid-flip-comm*)

lemma *flip-list-to-l*:
assumes *i* < *length ls length ls* ≤ *d*
shows *flip i* (*list-to-l ls*) = *list-to-l* (*ls*[*length ls* − *i* − *1* := ¬ *ls* ! (*length ls* − *i* − *1*)])
using *assms* **proof** (*induction ls arbitrary*: *i rule*: *list-to-l.induct*)
case (*2 l*)
have *i*<*d* **using** *2* **by** *auto*
show *?case* **proof** (*cases i*=*length l*)
case *True*
have *flip i* (*list-to-l* (*False* # *l*)) = *flip i* (*list-to-l l*) **by** *auto*
also have ... = *list-to-l* (*True* # *l*) **by** (*simp add*: *True*)
also have ... = *list-to-l* ((*False* # *l*)[*length* (*False* # *l*) − *i* − *1* :=

¬ (*False* # *l*) ! (*length* (*False* # *l*) − *i* − *1*)]) **unfolding** ‹*i*=*length l*› **by** *auto*
finally show *?thesis* **by** *auto*
next
case *False*
then have *i*<*length l* **using** *2*(*2*) **by** *auto*
let *?l* = *False* # *l*
have *flip i* (*list-to-l* (*False* # *l*)) = *flip i* (*list-to-l l*) **by** *auto*
also have … = *list-to-l* (*l*[*length l*−*i*−*1* := ¬*l*!(*length l*−*i*−*1*)]) **using** *2*
by (*simp add*: ‹*i* < *length l*›)
also have … = *list-to-l* (*?l*[*length ?l*−*i*−*1* := ¬*?l*!(*length ?l*−*i*−*1*)])
by (*smt* (*verit*, *ccfv-threshold*) *2*(*2*) *Nat.diff-add-assoc2 Suc-diff-Suc Suc-eq-plus1*
‹*i* < *length l*› *diff-Suc-1 diff-is-0-eq leD le-simps*(*3*) *list.size*(*4*) *list-update-code*(*3*)
nth-Cons′ numeral-nat(*7*) *list-to-l.simps*(*2*))
finally show *?thesis* **by** *auto*
qed
next
case (*3 l*)
have *i*<*d* **using** *3* **by** *auto*
show *?case* **proof** (*cases i*=*length l*)
case *True*
have *blog*: *blog* (*list-to-l l*) **using** *blog-list-to-l 3*(*3*) **by** *auto*
have *flip i* (*list-to-l* (*True* # *l*)) = *flip i* (*flip i* (*list-to-l l*)) **using** *True* **by** *auto*
also have … = *list-to-l l* **using** *flip-flip*[*OF* ‹*i*<*d*›] *blog* **by** *auto*
finally show *?thesis* **using** *True* **by** *auto*
next
case *False*
then have *i*<*length l* **using** *3*(*2*) **by** *auto*
let *?l* = *True* # *l*
have *flip i* (*list-to-l ?l*) = *flip i* (*flip* (*length l*) (*list-to-l l*)) **by** *auto*
also have … = *flip* (*length l*) (*flip i* (*list-to-l l*))
by (*intro flip-commute*) (*use 3* **in** ‹*auto simp add*: *False* ‹*i*<*d*› ›)
also have … = *flip* (*length l*) (*list-to-l* (*l*[*length l*−*i*−*1* := ¬*l*!(*length l*−*i*−*1*)]))
using *3*(*3*) *3.IH* ‹*i* < *length l*› **by** *auto*
also have … = *list-to-l* (*True* # (*l*[*length l*−*i*−*1* := ¬*l*!(*length l*−*i*−*1*)])) **by** *simp*
also have … = *list-to-l* (*?l*[*length ?l*−*i*−*1* := ¬*?l*!(*length ?l*−*i*−*1*)])
by (*smt* (*verit*, *ccfv-threshold*) *Suc-diff-Suc* ‹*i* < *length l*›
cancel-ab-semigroup-add-class.diff-right-commute diff-Suc-eq-diff-pred length-tl list.sel(*3*)

list-update-code(*3*) *nth-Cons-Suc*)
finally show *?thesis* **by** *auto*
qed
qed *auto*

The initial list corresponding to the initial value *empty* is the list containing only *False*.

definition *empty-list* **where**
empty-list = *replicate d False*

lemma *empty-list-to-l-replicate*:
list-to-l (*replicate n False*) = *empty*

```
  by (induct n, auto)

lemma empty-list-to-l [simp]:
  list-to-l empty-list = empty
  by (auto simp add: empty-list-to-l-replicate empty-list-def)

lemma empty-list-len-d[simp]:
  empty-list ∈ len-d-lists
  unfolding empty-list-def len-d-lists-def by auto

lemma empty-list-to-l-elem [simp]:
  empty ∈ list-to-l ‘ len-d-lists
  by (metis empty-list-len-d empty-list-to-l imageI)
```

Lemmas on how *list-to-l* works with *flip* and *bit*.

```
lemma list-to-l-flip:
  assumes i < length ls length ls ≤ d
  shows list-to-l (ls[i := ¬ ls ! i]) = flip (length ls − 1 − i) (list-to-l ls)
  using assms proof (induction ls arbitrary: i rule: list-to-l.induct)
  case (2 list)
  then show ?case proof (cases i=0)
    case False
    then obtain j where j: i = Suc j using not0-implies-Suc by presburger
    have list-to-l ((False # list)[i := ¬ (False # list) ! i]) =
      list-to-l (False # list[i−1 := ¬ list ! (i−1)]) unfolding j by auto
    also have ... = list-to-l (list[i−1 := ¬ list ! (i−1)]) by auto
    also have ... = flip (length list − 1 − (i−1)) (list-to-l list) using 2 2.IH j by auto
    also have ... = flip (length (False # list) − 1 − i) (list-to-l (False # list))
      by (simp add: j)
    finally show ?thesis by auto
  qed auto
next
  case (3 list)
  then show ?case proof (cases i=0)
    case True
    have False: (True # list)[i := ¬ (True # list) ! i] = False # list using True by auto
    have len: length (True # list) − 1 − i = length list using True by auto
    have len′: length list < d using 3 by auto
    have len′′: length list ≤ d using 3 by auto
    have list-to-l list = flip (length list) (flip (length list) (list-to-l list))
      using flip-flip[OF len′] by (simp add: blog-list-to-l len′′)
    then show ?thesis unfolding False len by (subst list-to-l.simps)+ auto
  next
    case False
    then obtain j where j: i = Suc j using not0-implies-Suc by presburger
    have len: length list < d using 3 by auto
    have list-to-l ((True # list)[i := ¬ (True # list) ! i]) =
      list-to-l (True # list[i−1 := ¬ list ! (i−1)]) unfolding j by auto
    also have ... = flip (length list) (list-to-l (list[i−1 := ¬ list ! (i−1)])) by auto
```

```
    also have ... = flip (length list) (flip (length list − 1 − (i−1)) (list-to-l list))
      using 3 3.IH j by auto
    also have ... = flip (length list) (flip (length (True # list) − 1 − i) (list-to-l list))
      by (simp add: j)
    also have ... = flip (length (True # list) − 1 − i) (flip (length list) (list-to-l list))
      by (intro flip-commute) (use 3 j in ‹auto simp add: len›)
    finally show ?thesis by auto
  qed
qed auto

lemma surj-list-to-l: list-to-l ‘ len-d-lists = Collect blog
proof (safe, goal-cases)
  case (1 - xa)
  then have length xa ≤ d unfolding len-d-lists-def by auto
  then show ?case proof (induct xa rule: list-to-l.induct)
    case 1
    show ?case by (auto simp add: blog.intros)
  next
    case (2 list)
    then show ?case by (subst list-to-l.simps(2), simp)
  next
    case (3 list)
    then show ?case by (subst list-to-l.simps(3), intro blog.intros, auto)
  qed
next
  case (2 x)
  then show ?case proof (induct rule: blog.induct)
    case 1
    show ?case by (subst empty-list-to-l[symmetric]) auto
  next
    case (2 l i)
    then obtain ls where ls: list-to-l ls = l length ls = d using len-d-lists-def by force
    have blog (flip i l) using 2 by (intro blog.intros, auto)
    define flip-ls where flip-ls = ls [d−i−1:= ¬ ls!(d−i−1)]
    then have length flip-ls = d using ‹length ls = d› by auto
    moreover have list-to-l flip-ls = flip i l unfolding flip-ls-def ls(2)[symmetric]
      by (subst flip-list-to-l[symmetric]) (auto simp add:2 ls)
    ultimately show ?case unfolding len-d-lists-def by (simp add: rev-image-eqI)
  qed
qed

lemma bit-list-to-l-over:
  assumes length l ≤ d i<d length l ≤ i
  shows bit (list-to-l l) i = bit empty i
  using assms proof (induct rule: list-to-l.induct)
  case (2 list)
  then show ?case using bit-flip-same[OF ‹i<d›] by auto
next
  case (3 list)
```

```
  then show ?case by (auto simp add: bit-flip-diff blog-list-to-l)
qed auto


lemma bit-list-to-l:
  assumes length l ≤ d i<length l
  shows bit (list-to-l l) i = (if l!(length l−i−1) then ¬ bit empty i else bit empty i)
  using assms proof (induct rule: list-to-l.induct)
  case (2 list)
  let ?l = False # list
  have length list ≤ d using 2 by auto
  have i<d using 2 by auto
  have rew: bit (list-to-l ?l) i = bit (list-to-l list) i by auto
  have c1: bit (list-to-l list) i = (if ?l!(length ?l−i−1) then ¬ bit empty i else bit empty i)
    if i≠length list using 2(1) 2(3) ‹length list ≤ d› that by auto
  have c2: bit (list-to-l list) i = (if ?l!(length ?l−i−1) then ¬ bit empty i else bit empty i)
    if i=length list using that by (subst bit-list-to-l-over[OF ‹length list ≤ d› ‹i<d›]) auto
  show ?case by (subst rew, cases i = length list)(use c1 c2 in ‹auto›)
next
  case (3 list)
  let ?l = True # list
  have length list ≤ d using 3 by auto
  have i<d using 3 by auto
  have rew: bit (list-to-l ?l) i = bit (flip (length list) (list-to-l list)) i (is - = ?right)
    by auto
  have c1: ?right = (if ?l!(length ?l−i−1) then ¬ bit empty i else bit empty i)
    if i≠length list
  proof −
    have ?right = bit (list-to-l list) i using that 3(2) blog-list-to-l
        blog-valid valid-bit-flip-diff by force
    also have ... = (if ?l!(length ?l−i−1) then ¬ bit empty i else bit empty i)
      using 3 ‹length list ≤ d› that by auto
    finally show ?thesis by auto
  qed
  have c2: ?right = (if ?l!(length ?l−i−1) then ¬ bit empty i else bit empty i)
    if i=length list
  proof −
    have ?right = (¬ bit (list-to-l list) i)
      using ‹i < d› ‹length list ≤ d› bit-flip-same blog-list-to-l that by blast
    also have ... = (¬ bit empty i)
      using ‹i < d› bit-list-to-l-over less-or-eq-imp-le that by blast
    finally show ?thesis using that by auto
  qed
  show ?case by (subst rew, cases i = length list)(use c1 c2 in ‹auto›)
qed auto

lemma list-to-l-eq:
  assumes list-to-l xs = list-to-l ys length xs = d length ys = d
  shows xs = ys
```

using *le0[of d] assms* **proof** (*induct d arbitrary*: *xs ys rule*: *Nat.dec-induct*)
case (*step n*)
obtain *x xs′* **where** *xs*: *xs* = *x* # *xs′ length xs′* = *n* **using** *step* **by** (*meson length-Suc-conv*)
obtain *y ys′* **where** *ys*: *ys* = *y* # *ys′ length ys′* = *n***using** *step* **by** (*meson length-Suc-conv*)
consider (*same*) *x*=*y* | (*neq*) *x*≠*y* **by** *blast*
then have *list-to-l xs′* = *list-to-l ys′* ∧ *x*=*y*
proof (*cases*)
case *same*
then have *list-to-l xs′* = *list-to-l ys′*
by (*metis* (*full-types*) *blog-list-to-l flip-flip list-to-l.simps*(*2*,*3*) *step*(*2*,*4*)
not-le order.asym xs ys)
then show *?thesis* **using** *same* **by** *blast*
next
case *neq*
have *False* **by** (*metis One-nat-def Suc-leI bit-list-to-l diff-Suc-1 diff-add-inverse2 lessI*
step(*2*,*4*,*5*,*6*) *neq nth-Cons-0 plus-1-eq-Suc xs*(*1*) *ys*(*1*))
then show *?thesis* **by** *auto*
qed
then show *?case* **unfolding** *xs ys* **by** (*simp add*: *local.step*(*3*) *xs*(*2*) *ys*(*2*))
qed *auto*

lemma *inj-list-to-l*: *inj-on list-to-l* (*len-d-lists*)
unfolding *inj-on-def* **proof** (*safe*, *goal-cases*)
case (*1 xs ys*)
have *len*: *length xs* = *d length ys* = *d* **using** *1* **unfolding** *len-d-lists-def* **by** *auto*
show *?case* **using** *len 1 list-to-l-eq* **by** *auto*
qed

lemma *bij-betw-list-to-l*: *bij-betw list-to-l len-d-lists* (*Collect blog*)
using *bij-betw-def inj-list-to-l surj-list-to-l* **by** *blast*

lemma *card-blog*: *card* (*Collect blog*) = *2^d*
by (*metis card-image card-len-d-lists inj-list-to-l surj-list-to-l*)

We split the 2^d elements into elements that have bits only in a certain set. This is later used to argue that an adversary running up to some n can only generate a count up to the n-th bit.

definition *has-bits* :: *nat set* ⇒ *bool list set* **where**
has-bits A = {*l*. *l*∈*len-d-lists* ∧ *True* ∈ (λ*i*. *l*!(*d*−*i*−*1*)) ‘ *A*}

lemma *has-bits-empty*[*simp*]:
has-bits {} = {}
unfolding *has-bits-def* **by** *auto*

lemma *has-bits-not-empty*:
assumes *y* ∈ *has-bits A A*≠{} *y*∈*len-d-lists*

shows $\mathit{list\text{-}to\text{-}l}\ y \neq \mathit{empty}$
proof −
obtain x **where** $x{\in}A\ y!(d{-}x{-}1)$ **using** *assms* **unfolding** *has-bits-def* **by** *auto*
then show *?thesis*
by (*smt* (*verit, best*) *assms*(*3*) *bit-list-to-l d-gr-0 diff-Suc-1 diff-diff-cancel diff-is-0-eq′ diff-le-self le-simps*(*2*) *len-d-lists-def mem-Collect-eq not-le*)
qed

lemma *has-bits-empty-list*:
$\mathit{empty\text{-}list} \notin \mathit{has\text{-}bits}\ \{0..{<}d\}$
using *has-bits-not-empty* **by** *fastforce*

lemma *has-bits-incl*:
assumes $A{\subseteq}B$
shows $\mathit{has\text{-}bits}\ A \subseteq \mathit{has\text{-}bits}\ B$
using *assms has-bits-def* **by** *auto*

lemma *has-bits-in-len-d-lists*[*simp*]:
$\mathit{has\text{-}bits}\ A \subseteq \mathit{len\text{-}d\text{-}lists}$
unfolding *has-bits-def* **by** *auto*

lemma *finite-has-bits*[*simp*]:
finite (*has-bits A*)
by (*meson finite-len-d-lists has-bits-in-len-d-lists rev-finite-subset*)

lemma *has-bits-not-elem*:
assumes $y{\in}\mathit{has\text{-}bits}\ A\ \ A{\neq}\{\}\ \ A{\subseteq}\{0..{<}d\}\ \ y{\in}\mathit{len\text{-}d\text{-}lists}\ \ n \notin A\ \ n{<}d$
shows $y[d{-}n{-}1{:=}\neg y!(d{-}n{-}1)] \in \mathit{has\text{-}bits}\ A$
proof −
obtain i **where** i: $y\ !\ (d - i - 1)\ \ i{\in}A$ **using** *assms has-bits-def* **by** *auto*
then have $n{\neq}i$ **using** *assms* **by** *auto*
then have $y[d{-}n{-}1 := \neg\ y!(d{-}n{-}1)]!(d{-}i{-}1)$ **using** i *assms* **by** (*subst nth-list-update-neq*) *auto*
moreover have $\mathit{length}\ (y[d - n - 1 := \neg\ y\ !\ (d - n - 1)]) = d$ **using** *assms len-d-lists-def* **by** *auto*
ultimately show *?thesis* **using** ‹$i{\in}A$› **unfolding** *has-bits-def len-d-lists-def* **by** *auto*
qed

lemma *has-bits-split-Suc*:
assumes $n{<}d$
shows $\mathit{has\text{-}bits}\ \{n..{<}d\} = \mathit{has\text{-}bits}\ \{n\} \cup \mathit{has\text{-}bits}\ \{\mathit{Suc}\ n..{<}d\}$
proof −
have $x \in \mathit{len\text{-}d\text{-}lists} \Longrightarrow x\ !\ (d - \mathit{Suc}\ xa) \Longrightarrow \forall xa{\in}\{\mathit{Suc}\ n..{<}d\}.\ \neg\ x\ !\ (d - \mathit{Suc}\ xa) \Longrightarrow n \le xa \Longrightarrow xa < d \Longrightarrow x\ !\ (d - \mathit{Suc}\ n)$ **for** $x\ xa$
by (*metis atLeastLessThan-iff le-eq-less-or-eq le-simps*(*3*))
moreover have $x \in \mathit{len\text{-}d\text{-}lists} \Longrightarrow x\ !\ (d - \mathit{Suc}\ n) \Longrightarrow \exists xa{\in}\{n..{<}d\}.\ x\ !\ (d - \mathit{Suc}\ xa)$ **for** x
using *assms atLeastLessThan-iff* **by** *blast*
ultimately show *?thesis* **unfolding** *has-bits-def* **by** *auto*
qed

The function *has-bits-upto* looks only at elements with bits lower than some n.

definition *has-bits-upto* **where**
  has-bits-upto n = len-d-lists − has-bits $\{n..<d\}$

lemma *finite-has-bits-upto* [*simp*]:
  finite (*has-bits-upto n*)
  unfolding *has-bits-upto-def* **by** *auto*

lemma *has-bits-elem*:
  assumes $x \in$ *len-d-lists − has-bits A* $a \in A$
  shows $\neg x!(d-a-1)$
  using *assms*(*1*) *assms*(*2*) *has-bits-def* **by** *force*

lemma *has-bits-upto-elem*:
  assumes $x \in$ *has-bits-upto n* $n<d$
  shows $\neg x!(d-n-1)$
  using *assms has-bits-upto-def* **by** (*intro has-bits-elem*[*of x* $\{n..<d\}$ *n*]) *auto*

lemma *has-bits-upto-incl*:
  assumes $n \leq m$
  shows *has-bits-upto n* $\subseteq$ *has-bits-upto m*
  using *assms* **unfolding** *has-bits-upto-def* **by** (*simp add*: *Diff-mono has-bits-incl*)

lemma *has-bits-upto-d*:
  has-bits-upto d = len-d-lists
  unfolding *has-bits-upto-def* **by** *auto*

lemma *empty-list-has-bits-upto*:
  empty-list $\in$ *has-bits-upto n*
  using *empty-list-to-l has-bits-not-empty has-bits-upto-def* **by** *fastforce*

lemma *empty-list-to-l-has-bits-upto*:
  empty $\in$ *list-to-l* ‘ *has-bits-upto n*
  using *empty-list-has-bits-upto empty-list-to-l* **by** (*metis image-eqI*)

lemma *len-d-empty-has-bits*:
  len-d-lists − {*empty-list*} *= has-bits* $\{0..<d\}$
proof (*safe*, *goal-cases*)
  case (*1 x*)
  then have $\neg\ x!(d-i-1)$ **if** $i<d$ **for** i **using** *has-bits-elem that* **by** *auto*
  then have $\neg\ x!i$ **if** $i<d$ **for** i
    by (*metis Suc-leI d-gr-0 diff-Suc-less diff-add-inverse diff-diff-cancel plus-1-eq-Suc that*)
  then have *x = empty-list* **unfolding** *empty-list-def*
    by (*smt* (*verit*, *best*) *1*(*1*) *in-set-conv-nth len-d-lists-def mem-Collect-eq replicate-eqI*)
  then show *?case* **by** *auto*
qed (*auto simp add*: *has-bits-def has-bits-empty-list empty-list-def*)

Properties of d

lemma *two-d-gr-1*:

$2\hat{}d > (1::nat)$
by (*meson d-gr-0 one-less-power rel-simps*(*49*) *semiring-norm*(*76*))

Lemmas on *flip*, *bit* and *valid*.

lemma *valid-inv*: $-$ *Collect valid* $=$ *valid* $-$‘ {*False*} **by** *auto*

lemma *blog-inv*: $-$ *Collect blog* $=$ *blog* $-$‘ {*False*} **by** *auto*

lemma *not-blog-flip*: $i<d \Longrightarrow (\neg$ *blog l*$) \Longrightarrow (\neg$ *blog* (*flip i l*))
by (*metis blog.intros*(*2*) *blog-valid inj-def inj-flip valid-flip-flip*)

Lemmas on X and Y. X and Y are embeddings of the classical memory parts of input and output registers to the oracle function into the quantum register $'mem$.

lemma *register-X*:
register X
by *auto*

lemma *register-Y*:
register Y
by *auto*

lemma *X-0*:
X 0 $=$ *0* **using** *clinear-register complex-vector.linear-0 register-X* **by** *blast*

How to check that no qubit in $'x$ is in the set S in a quantum setting. This is more complicated, since we cannot just ask if $x \in S$. We need to ask for the embedding of the projection of the classical set S in the register X.

definition *proj-classical-set M* $=$ *Proj* (*ccspan* (*ket* ‘ *M*))

definition *S-embed* S' $=$ *X* (*proj-classical-set* (*Collect* S'))
definition *not-S-embed* S' $=$ *X* (*proj-classical-set* ($-$ (*Collect* S')))

lemma *is-Proj-proj-classical-set*:
is-Proj (*proj-classical-set M*)
unfolding *proj-classical-set-def* **by** *auto*

lemma *proj-classical-set-split-id*:
id-cblinfun $=$ *proj-classical-set M* $+$ *proj-classical-set* ($-M$)
unfolding *proj-classical-set-def*
by (*smt* (*verit*) *Compl-iff Proj-orthog-ccspan-union Proj-top boolean-algebra-class.sup-compl-top*

ccspan-range-ket imageE image-Un orthogonal-ket)

lemma *proj-classical-set-sum-ket-finite*:
assumes *finite A*
shows *proj-classical-set A* $=$ ($\sum i{\in}A.$ *selfbutter* (*ket i*))
using *assms* **proof** (*induction A rule*: *Finite-Set.finite.induct*)

case (*insertI A a*)
show *?case* **proof** (*cases a*$\in$*A*)
case *False*
have *insert*: *ket* ‘ (*insert a A*) = *insert* (*ket a*) (*ket* ‘ *A*) **by** *auto*
have *Proj*: *Proj* (*ccspan* (*ket* ‘ *A*)) = ($\sum$ *i*$\in$*A*. *proj* (*ket i*))
using *insertI* **unfolding** *proj-classical-set-def* **by** (*auto simp add*: *butterfly-eq-proj*)
show *?thesis* **unfolding** *proj-classical-set-def insert*
proof (*subst Proj-orthog-ccspan-insert*, *goal-cases*)
case *2*
then show *?case* **unfolding** *Proj*
by (*simp add*: *False butterfly-eq-proj local.insertI*(*1*))
qed (*auto simp add*: *False*)
qed (*simp add*: *insertI insert-absorb*)
qed (*auto simp add*: *proj-classical-set-def*)

lemma *proj-classical-set-not-elem*:
assumes *i*$\notin$*A*
shows *proj-classical-set A* $*_V$ *ket i* = *0*
by (*metis Compl-iff Proj-fixes-image add-cancel-right-left assms cblinfun.add-left ccspan-superset'*

id-cblinfun-apply proj-classical-set-def proj-classical-set-split-id rev-image-eqI)

lemma *proj-classical-set-elem*:
assumes *i*$\in$*A*
shows *proj-classical-set A* $*_V$ *ket i* = *ket i*
using *assms* **by** (*simp add*: *Proj-fixes-image ccspan-superset' proj-classical-set-def*)

lemma *proj-classical-set-upto*:
assumes *i*<*j*
shows *proj-classical-set* {*j*..} $*_V$ *ket* (*i*::*nat*) = *0*
by (*intro proj-classical-set-not-elem*) (*use assms* **in** ‹*auto*›)

lemma *proj-classical-set-apply*:
assumes *finite A*
shows *proj-classical-set A* $*_V$ *y* = ($\sum$ *i*$\in$*A*. *Rep-ell2 y i* $*_C$ *ket i*)
unfolding *proj-classical-set-def trunc-ell2-as-Proj*[*symmetric*]
by (*intro trunc-ell2-finite-sum*, *simp add*: *assms*)

lemma *proj-classical-set-split-Suc*:
proj-classical-set {*n*..} = *proj* (*ket n*) + *proj-classical-set* {*Suc n*..}
proof −
have *ket* ‘ {*n*..} = *insert* (*ket n*) (*ket* ‘ {*Suc n*..}) **by** *fastforce*
then show *?thesis* **unfolding** *proj-classical-set-def*
by (*subst Proj-orthog-ccspan-insert*[*symmetric*]) *auto*
qed

lemma *proj-classical-set-union*:

assumes $\bigwedge x\ y.\ x \in ket\ `\ A \Longrightarrow y \in ket\ `\ B \Longrightarrow is\text{-}orthogonal\ x\ y$
shows *proj-classical-set* $(A \cup B)$ = *proj-classical-set* A + *proj-classical-set* B
unfolding *proj-classical-set-def*
by (*subst image-Un*, *intro Proj-orthog-ccspan-union*)(*auto simp add*: *assms*)

Later, we need to project only on the second part of the register (the counting part).

definition *Proj-ket-set* :: $'a$ *set* $\Rightarrow$ $('mem \times 'a)$ *update* **where**
Proj-ket-set A = *id-cblinfun* $\otimes_o$ *proj-classical-set* A

lemma *Proj-ket-set-vec*:
assumes $y \in A$
shows *Proj-ket-set* $A *_V (v \otimes_s ket\ y) = v \otimes_s ket\ y$
unfolding *Proj-ket-set-def* **using** *proj-classical-set-elem*[*OF assms*]
by (*auto simp add*: *tensor-op-ell2*)

definition *Proj-ket-upto* :: *bool list set* $\Rightarrow$ $('mem \times 'l)$ *update* **where**
Proj-ket-upto A = *Proj-ket-set* (*list-to-l* ` A)

lemma *Proj-ket-upto-vec*:
assumes $y \in A$
shows *Proj-ket-upto* $A *_V (v \otimes_s ket\ (list\text{-}to\text{-}l\ y)) = v \otimes_s ket\ (list\text{-}to\text{-}l\ y)$
unfolding *Proj-ket-upto-def* **using** *assms* **by** (*auto intro*!: *Proj-ket-set-vec*)

We can split a state into two parts: the part in S and the one not in S.

lemma *S-embed-not-S-embed-id* [*simp*]:
S-embed S' + *not-S-embed* S' = *id-cblinfun*
proof −
have *proj-classical-set* (*Collect* S') + *proj-classical-set* (− (*Collect* S')) = *id-cblinfun*
unfolding *proj-classical-set-def*
by (*subst Proj-orthog-ccspan-union*[*symmetric*]) (*auto simp add*: *image-Un*[*symmetric*])
then have ∗: X (*proj-classical-set* (*Collect* S') + *proj-classical-set* (− (*Collect* S'))) =
X *id-cblinfun* **by** *auto*
have X (*proj-classical-set* (*Collect* S')) + X (*proj-classical-set* (− (*Collect* S'))) =
X *id-cblinfun* **unfolding** ∗[*symmetric*]
using *clinear-register*[*OF register-X*] **by** (*simp add*: *clinear-iff*)
then show *?thesis* **unfolding** *S-embed-def not-S-embed-def* **by** *auto*
qed

lemma *S-embed-not-S-embed-add*:
S-embed S' (*ket* a) + *not-S-embed* S' (*ket* a) = *ket* a
using *S-embed-not-S-embed-id*
by (*metis cblinfun-id-cblinfun-apply plus-cblinfun.rep-eq*)

lemma *S-embed-idem* [*simp*]:
S-embed S' o_{CL} *S-embed* S' = *S-embed* S'
unfolding *S-embed-def Axioms-Quantum.register-mult*[*OF register-X*] *proj-classical-set-def* **by** *auto*

lemma *S-embed-adj*:
$(S\text{-}embed\ S')* = S\text{-}embed\ S'$
unfolding *S-embed-def register-adj*[*OF register-X*, *symmetric*] *proj-classical-set-def adj-Proj*
by *auto*

lemma *not-S-embed-idem*:
$not\text{-}S\text{-}embed\ S'\ o_{CL}\ not\text{-}S\text{-}embed\ S' = not\text{-}S\text{-}embed\ S'$
unfolding *not-S-embed-def Axioms-Quantum.register-mult*[*OF register-X*] *proj-classical-set-def*
by *auto*

lemma *not-S-embed-adj*:
$(not\text{-}S\text{-}embed\ S')* = not\text{-}S\text{-}embed\ S'$
unfolding *not-S-embed-def register-adj*[*OF register-X*, *symmetric*] *proj-classical-set-def adj-Proj*
by *auto*

lemma *not-S-embed-S-embed* [*simp*]:
$not\text{-}S\text{-}embed\ S'\ o_{CL}\ S\text{-}embed\ S' = 0$
proof −
have *orthogonal-spaces* (*ccspan* (*ket* ‘ (− *Collect* S'))) (*ccspan* (*ket* ‘ *Collect* S'))
using *orthogonal-spaces-ccspan* **by** *fastforce*
then have *proj-classical-set* (− *Collect* S') o_{CL} *proj-classical-set* (*Collect* S') = *0*
unfolding *proj-classical-set-def* **using** *orthogonal-projectors-orthogonal-spaces* **by** *auto*
then show *?thesis* **unfolding** *not-S-embed-def S-embed-def Axioms-Quantum.register-mult*[*OF register-X*]
using *X-0* **by** *auto*
qed

lemma *S-embed-not-S-embed* [*simp*]:
$S\text{-}embed\ S'\ o_{CL}\ not\text{-}S\text{-}embed\ S' = 0$
by (*metis S-embed-adj adj-0 adj-cblinfun-compose not-S-embed-S-embed not-S-embed-adj*)

lemma *not-S-embed-Proj*:
$not\text{-}S\text{-}embed\ S = Proj\ (not\text{-}S\text{-}embed\ S *_S \top)$
unfolding *not-S-embed-def* **using** *register-projector*[*OF register-X is-Proj-proj-classical-set*]
by (*simp add*: *Proj-on-own-range*)

lemma *not-S-embed-in-X-image*:
assumes $a \in space\text{-}as\text{-}set\ (-\ (not\text{-}S\text{-}embed\ S *_S \top))$
shows $(not\text{-}S\text{-}embed\ S) *_V\ a = 0$
using *register-projector*[*OF register-X is-Proj-proj-classical-set*]
Proj-0-compl[*OF assms*] **unfolding** *not-S-embed-def* **by** (*simp add*: *Proj-on-own-range*)

In the register for the adversary runs *run-B* and *run-B-count*, we want to look at the *'mem* part only. Ψs lets us look at the *'mem* part that is tensored with the *i*-th ket state.

definition Ψs **where** $\Psi s\ i\ v = (tensor\text{-}ell2\text{-}right\ (ket\ i)*) *_V\ v$

lemma *tensor-ell2-right-compose-id-cblinfun*:
$tensor\text{-}ell2\text{-}right\ (ket\ a)*\ o_{CL}\ A \otimes_o id\text{-}cblinfun = A\ o_{CL}\ tensor\text{-}ell2\text{-}right\ (ket\ a)*$

by (*intro equal-ket*)(*auto simp add*: *tensor-ell2-ket*[*symmetric*] *tensor-op-ell2 cblinfun.scaleC-right*)

lemma *Ψs-id-cblinfun*:
$\Psi s\ a\ ((A \otimes_o \textit{id-cblinfun}) *_V\ v) = A *_V (\Psi s\ a\ v)$
unfolding *Ψs-def*
by (*auto simp add*: *cblinfun-apply-cblinfun-compose*[*symmetric*] *tensor-ell2-right-compose-id-cblinfun simp del*: *cblinfun-apply-cblinfun-compose*)

Additional Lemmas

lemma *id-cblinfun-tensor-split-finite*:
assumes *finite A*
shows $(\textit{id-cblinfun}:: ('mem \times 'a)\ ell2 \Rightarrow_{CL} ('mem \times 'a)\ ell2) =$
$(\sum i{\in}A.\ (\textit{tensor-ell2-right}\ (ket\ i))\ o_{CL}\ (\textit{tensor-ell2-right}\ (ket\ i)*)) +$
Proj-ket-set $(-A)$
proof −
have $(\textit{id-cblinfun}:: ('mem \times 'a)\ update) = \textit{id-cblinfun} \otimes_o \textit{id-cblinfun}$ **by** *auto*
also have $\ldots = \textit{id-cblinfun} \otimes_o (\textit{proj-classical-set}\ A + \textit{proj-classical-set}\ (-A))$
by (*subst proj-classical-set-split-id*[*of A*]) (*auto*)
also have $\ldots = \textit{id-cblinfun} \otimes_o (\sum i{\in}A.\ \textit{selfbutter}\ (ket\ i)) +$
Proj-ket-set $(-A)$ **unfolding** *Proj-ket-set-def*
by (*subst proj-classical-set-sum-ket-finite*[*OF assms*])(*auto simp add*: *tensor-op-right-add*)
also have $\ldots = (\sum i{\in}A.\ \textit{id-cblinfun} \otimes_o \textit{selfbutter}\ (ket\ i)) +$
Proj-ket-set $(-A)$
using *clinear-tensor-right complex-vector.linear-sum* **by** (*smt* (*verit*, *best*) *sum.cong*)
also have $\ldots = (\sum i{\in}A.\ (\textit{tensor-ell2-right}\ (ket\ i))\ o_{CL}\ (\textit{tensor-ell2-right}\ (ket\ i)*)) +$
Proj-ket-set $(-A)$
by (*simp add*: *tensor-ell2-right-butterfly*)
finally show *?thesis* **by** *auto*
qed

Lemmas on sums of butterflys

lemma *sum-butterfly-ket0*:
assumes $(y::nat)<d+1$
shows $(\sum i<d+1.\ \textit{butterfly}\ (ket\ 0)\ (ket\ i)) *_V (ket\ y) = ket\ 0$
proof −
have $(\sum i<d+1.\ \textit{butterfly}\ (ket\ 0)\ (ket\ i)) *_V ket\ y = (\sum i<d+1.\ \textit{if}\ i{=}y\ \textit{then}\ ket\ 0\ \textit{else}\ 0)$
by (*subst cblinfun.sum-left*, *intro sum.cong*, *auto*)
also have $\ldots = ket\ 0$ **by** (*subst sum.delta*, *use assms* **in** ‹*auto*›)
finally show *?thesis* **by** *auto*
qed

lemma *sum-butterfly-ket0'*:
$(\sum i<d+1.\ \textit{butterfly}\ (ket\ 0)\ (ket\ i)) *_V \textit{proj-classical-set}\ \{..<d+1\} *_V\ y =$
$(\sum i<d+1.\ \textit{Rep-ell2}\ y\ i) *_C ket\ 0$
for *y*::*nat ell2*
proof −
have $(\sum i<d+1.\ \textit{butterfly}\ (ket\ 0)\ (ket\ i)) *_V \textit{proj-classical-set}\ \{..<d+1\} *_V\ y =$
$(\sum i<d+1.\ \textit{Rep-ell2}\ y\ i *_C (\sum i<d+1.\ \textit{butterfly}\ (ket\ 0)\ (ket\ i)) *_V ket\ i)$
by (*subst proj-classical-set-apply*, *simp*)

(*subst cblinfun.sum-right*, *intro sum.cong*, *auto simp add*: *cblinfun.scaleC-right*)
also have ... $= (\sum i<d+1.\ Rep\text{-}ell2\ y\ i *_C ket\ 0)$
by (*intro sum.cong*) (*use sum-butterfly-ket0* **in** ‹*auto*›)
also have ... $= (\sum i<d+1.\ Rep\text{-}ell2\ y\ i) *_C ket\ 0$ **by** (*rule scaleC-sum-left*[*symmetric*])
finally show *?thesis* **by** *auto*
qed

The oracle query is a unitary.

lemma *inj-Uquery-map*:
inj (λ(*x*, (*y*::'*y*)). (*x*, *y* + *H x*))
unfolding *inj-def* **by** *auto*

lemma *classical-operator-exists-Uquery*:
classical-operator-exists (*Some o* (λ(*x*,(*y*::'*y*)). (*x*, *y* + (*H x*))))
by (*intro classical-operator-exists-inj*, *subst inj-map-total*)
(*auto simp add*: *inj-Uquery-map*)

lemma *Uquery-ket*:
$Uquery\ F *_V ket\ (a::'x) \otimes_s ket\ (b::'y) = ket\ a \otimes_s ket\ (b + F\ a)$
unfolding *Uquery-def tensor-ell2-ket*
by (*subst classical-operator-ket*[*OF classical-operator-exists-Uquery*]) *auto*

lemma *unitary-H*: *unitary* (*Uquery* (*H*::'*x* ⇒ '*y*))
proof −
have *inj*: *inj* (λ(*x*, *y*). (*x*, *y* + *H x*)) **by** (*auto simp add*: *inj-on-def*)
have *surj*: *surj* (λ(*x*, *y*). (*x*, *y* + *H x*))
by (*metis* (*mono-tags*, *lifting*) *case-prod-Pair-iden diff-add-cancel split-conv split-def surj-def*)
show *?thesis* **unfolding** *Uquery-def*
by (*intro unitary-classical-operator*) (*auto simp add*: *inj surj bij-def*)
qed

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*

end
theory *More-Kraus-Maps*

imports *Kraus-Maps.Kraus-Maps*
O2H-Additional-Lemmas
begin

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

Fst on kraus families.

```
lemma inj-Fst-alt:
  assumes c≠0
  shows a ⊗o c = b ⊗o c ⟹ a = b
  using inj-tensor-left[OF assms] unfolding inj-def by auto

lift-definition kf-Fst :: ('a ell2, 'c ell2, unit) kraus-family ⇒
  (('a × 'b) ell2, ('c × 'b) ell2, unit) kraus-family is
  λE. (λ(x,-). (x ⊗o id-cblinfun, ())) ` E
proof (rename-tac 𝔈, intro CollectI)
  fix 𝔈 :: ‹('a ell2 ⇒CL 'c ell2 × unit) set›
  assume ‹𝔈 ∈ Collect kraus-family›
  then have ‹kraus-family 𝔈› by auto
  define f :: ‹('a ell2 ⇒CL 'c ell2 × unit) ⇒ -› where ‹f = (λ(E,x). (E ⊗o id-cblinfun, ()))›
  define Fst where ‹Fst = f ` 𝔈›
  show ‹kraus-family Fst›
  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[of ‹{}›] have [simp]: ‹B ≥ 0› by simp
    have ‹(∑(G, z)∈U. G* oCL G) ≤ B ⊗o id-cblinfun› if ‹finite U› and ‹U ⊆ Fst› for U
    proof −
      from that
      obtain V where V-subset: ‹V ⊆ 𝔈› and [simp]: ‹finite V› and ‹U = f ` V›
        apply (simp only: Fst-def flip: f-def)
        by (meson finite-subset-image)
      have ‹inj-on f V› by (auto intro!: inj-onI simp: f-def inj-Fst-alt[OF id-cblinfun-not-0])
      have ‹(∑(G, z)∈U. G* oCL G) = (∑(G, z)∈f ` V. G* oCL G)›
        using ‹U = f ` V› by (auto)
      also have ‹... = (∑(E,x)∈V. (E ⊗o id-cblinfun)* oCL (E ⊗o id-cblinfun))›
        by (subst sum.reindex) (use ‹inj-on f V› in ‹auto simp: case-prod-unfold f-def›)
      also have ‹... = (∑(E,x)∈V. (E* oCL E)) ⊗o id-cblinfun›
        by (subst tensor-op-cbilinear.sum-left)
          (simp add: case-prod-unfold comp-tensor-op tensor-op-adjoint)
      also have ‹... ≤ B ⊗o id-cblinfun›
        using V-subset by (auto intro!: tensor-op-mono B)
      finally show ?thesis by−
    qed
    then show ‹bdd-above ((λF. ∑(E, x)∈F. E* oCL E) ` {F. finite F ∧ F ⊆ Fst})›
      by fast
    show ‹0 ∉ fst ` Fst› (is ‹?zero ∉ -›)
    proof (rule notI)
      assume ‹?zero ∈ fst ` Fst›
      then have ‹?zero ∈ (λx. fst x ⊗o id-cblinfun) ` 𝔈›
        by (simp add: Fst-def f-def image-image case-prod-unfold)
      then obtain E where ‹E ∈ 𝔈› and ‹?zero = fst E ⊗o id-cblinfun›
        by blast
      then have ‹fst E = 0›
```

by (*metis id-cblinfun-not-0 tensor-op-nonzero*)
with ‹$E \in \mathfrak{E}$›
show *False*
using ‹*kraus-family* $\mathfrak{E}$›
by (*simp add*: *kraus-family-def*)
qed
qed
qed

lemma *summable-on-in-kf-Fst*:
fixes f :: $'c \Rightarrow 'a\ ell2 \Rightarrow_{CL} 'a\ ell2$
and b :: $'b\ ell2 \Rightarrow_{CL} 'b\ ell2$
shows *summable-on-in cweak-operator-topology* ($\lambda x.$ (*fst* $x* \ o_{CL}$ *fst* x) $\otimes_o$ *id-cblinfun*) (*Rep-kraus-family* G)
proof −
have *bdd-above* (*sum* ($\lambda(E, x).\ E* \ o_{CL}\ E$) ‘ $\{F.$ *finite* $F \wedge F \subseteq$ *Rep-kraus-family* $G\}$)
by (*intro summable-wot-bdd-above*[*OF kf-bound-summable positive-cblinfun-squareI*])
(*auto simp add*: *case-prod-beta*)
then obtain B **where** B: ‹($\sum x{\in}S.$ *fst* $x* \ o_{CL}$ *fst* x) $\leq B$› **if** ‹*finite* S› **and** ‹$S \subseteq$ (*Rep-kraus-family* G)› **for** S
apply *atomize-elim* **unfolding** *bdd-above-def* **by** (*auto simp*: *case-prod-beta*)
have *bound*: ($\sum x{\in}F.$ (*fst* $x* \ o_{CL}$ *fst* x) $\otimes_o$ *id-cblinfun*) $\leq B \otimes_o$ *id-cblinfun*
if *finite* F $F \subseteq$ (*Rep-kraus-family* G) **for** F
by (*subst tensor-op-cbilinear.sum-left*[*symmetric*], *intro tensor-op-mono-left*)
(*auto simp add*: *B that*)
have *pos*: $x \in$ (*Rep-kraus-family* G) $\Longrightarrow 0 \leq$ (*fst* $x* \ o_{CL}$ *fst* x) $\otimes_o$ *id-cblinfun* **for** x
using *positive-cblinfun-squareI positive-id-cblinfun tensor-op-pos* **by** *blast*
show *?thesis* **by** (*auto intro*!: *summable-wot-boundedI*[*OF bound pos*])
qed

lemma *infsum-in-kf-Fst*:
fixes f :: $'c \Rightarrow 'a\ ell2 \Rightarrow_{CL} 'a\ ell2$
and b :: $'b\ ell2 \Rightarrow_{CL} 'b\ ell2$
shows *infsum-in cweak-operator-topology* ($\lambda x.$ (*fst* $x* \ o_{CL}$ *fst* x) $\otimes_o$ *id-cblinfun*) (*Rep-kraus-family* G) $\leq$
(*infsum-in cweak-operator-topology* ($\lambda x.$ *fst* $x* \ o_{CL}$ *fst* x) (*Rep-kraus-family* G)) $\otimes_o$ *id-cblinfun*
proof −
have *sum*: *summable-on-in cweak-operator-topology* ($\lambda x.$ *fst* $x* \ o_{CL}$ *fst* x) (*Rep-kraus-family* G)
using *kf-bound-summable*
by (*metis* (*mono-tags*, *lifting*) *cond-case-prod-eta fst-conv*)
have *pos-f*: $x \in$ *Rep-kraus-family* $G \Longrightarrow 0 \leq$ *fst* $x* \ o_{CL}$ *fst* x **for** x
using *positive-cblinfun-squareI* **by** *blast*
have *pos*: $x \in$ *Rep-kraus-family* $G \Longrightarrow 0 \leq$ (*fst* $x* \ o_{CL}$ *fst* x) $\otimes_o$ *id-cblinfun* **for** x
by (*simp add*: *positive-cblinfun-squareI tensor-op-pos*)

```
 define s where s = infsum-in cweak-operator-topology (λx. fst x* o_CL fst x) (Rep-kraus-family
G)
  have sum (λx. fst x* o_CL fst x) F ≤ s if finite F and F ⊆ (Rep-kraus-family G) for F
    unfolding s-def
    using that infsum-wot-is-Sup[OF sum pos-f] unfolding is-Sup-def by auto
  then have sum (λx. (fst x* o_CL fst x) ⊗o id-cblinfun) F ≤ s ⊗o id-cblinfun
    if finite F and F ⊆ Rep-kraus-family G for F
    apply (subst tensor-op-cbilinear.sum-left[symmetric])
    apply (intro tensor-op-mono-left[OF - positive-id-cblinfun])
    by (use that in ‹auto›)
  moreover have is-Sup (sum (λx. (fst x* o_CL fst x) ⊗o id-cblinfun) `
    {F. finite F ∧ F ⊆ (Rep-kraus-family G)})
    (infsum-in cweak-operator-topology (λx. (fst x* o_CL fst x) ⊗o id-cblinfun) (Rep-kraus-family
G))
    by (intro infsum-wot-is-Sup[OF summable-on-in-kf-Fst pos], auto)
  ultimately have infsum-in cweak-operator-topology (λx. (fst x* o_CL fst x) ⊗o id-cblinfun)
    (Rep-kraus-family G) ≤ s ⊗o id-cblinfun
    by (smt (verit, ccfv-threshold) image-iff is-Sup-def mem-Collect-eq)
  then show ?thesis unfolding s-def by auto
qed

lemma kf-bound-kf-Fst:
  kf-bound (kf-Fst F:: (('a × 'b) ell2, ('c × 'b) ell2, unit) kraus-family) ≤
  kf-bound F ⊗o id-cblinfun
proof −
  have inj: inj-on (λ(x, -). (x ⊗o id-cblinfun, ())) (Rep-kraus-family F)
    unfolding inj-on-def by (auto simp add: inj-Fst-alt[OF id-cblinfun-not-0])
 have infsum-in cweak-operator-topology (λx. (fst x* o_CL fst x) ⊗o id-cblinfun) (Rep-kraus-family
F) ≤
   infsum-in cweak-operator-topology (λx. (fst x* o_CL fst x)) (Rep-kraus-family F) ⊗o id-cblinfun
    by (rule infsum-in-kf-Fst)
 then have infsum-in cweak-operator-topology (λx. (fst x* o_CL fst x) ⊗o id-cblinfun)(Rep-kraus-family
F)
   ≤ infsum-in cweak-operator-topology (λ(E, x). E* o_CL E) (Rep-kraus-family F) ⊗o id-cblinfun
    by (metis (mono-tags, lifting) infsum-in-cong prod.case-eq-if)
  then have infsum-in cweak-operator-topology (λ(E, x). E* o_CL E) ((λ(x,-).
    (x ⊗o (id-cblinfun ::'b update), ())) ` (Rep-kraus-family F)) ≤
   (infsum-in cweak-operator-topology (λ(E, x). E* o_CL E) (Rep-kraus-family F)) ⊗o id-cblinfun
    by (subst infsum-in-reindex[OF inj])
      (auto simp add: o-def case-prod-beta tensor-op-adjoint comp-tensor-op)
  then show ?thesis
    by (simp add: kf-Fst.rep-eq kf-bound.rep-eq)
qed

lemma sandwich-tc-kf-apply-Fst:
  sandwich-tc (Snd (Q::'d update)) (kf-apply (kf-Fst F::
 (('a×'d) ell2, ('a×'d) ell2, unit) kraus-family) ϱ) =
 kf-apply (kf-Fst F) (sandwich-tc (Snd Q) ϱ)
proof −
```

have *sand*: *sandwich-tc* (*Snd Q*) (*sandwich-tc a* ϱ) =
sandwich-tc a (*sandwich-tc* (*Snd Q*) ϱ)
if (*a*, ()) ∈ *Rep-kraus-family* (*kf-Fst F*) **for** *a*
proof −
obtain *x* **where** *a*: *a* = *x* $\otimes_o$ *id-cblinfun*
using ‹(*a*, ()) ∈ *Rep-kraus-family* (*kf-Fst F*)› **unfolding** *kf-Fst.rep-eq* **by** *auto*
show *?thesis* **unfolding** *a sandwich-tc-compose′*[*symmetric*] *Snd-def* **by** (*auto simp add*: *comp-tensor-op*)
qed
have *1*: *sum* (*sandwich-tc* (*Snd Q*) *o* (λ*E*. (*sandwich-tc* (*fst E*) ϱ))) *F′* =
sandwich-tc (*Snd Q*) ($\sum$ *E*∈*F′*. *sandwich-tc* (*fst E*) ϱ)
if *finite F′ F′* ⊆ *Rep-kraus-family* (*kf-Fst F* ::
((*′a*×*′d*) *ell2*, (*′a*×*′d*) *ell2*, *unit*) *kraus-family*) **for** *F′*
by (*auto simp add*: *sandwich-tc-sum sandwich-tc-tensor intro*!: *sum.cong*)
have *2*: *isCont* (*sandwich-tc* (*Snd Q*))
($\sum_\infty$*E*∈*Rep-kraus-family* (*kf-Fst F*). *sandwich-tc* (*fst E*) ϱ)
using *isCont-sandwich-tc* **by** *auto*
have *3*: (λ*E*. *sandwich-tc* (*fst E*) ϱ) *summable-on Rep-kraus-family* (*kf-Fst F*)
by (*metis* (*no-types*, *lifting*) *cond-case-prod-eta fst-conv kf-apply-summable*)
then show *?thesis* **unfolding** *kf-apply.rep-eq*
by (*subst infsum-comm-additive-general*[*OF 1 2 3*, *symmetric*])
(*auto intro*!: *infsum-cong simp add*: *sand*)
qed

kraus family Fst is trace preserving.

lemma *kf-apply-Fst-tensor*:
‹*kf-apply* (*kf-Fst* $\mathfrak{E}$ ::((*′c* × *′b*) *ell2*, (*′a* × *′b*) *ell2*, *unit*) *kraus-family*)
(*tc-tensor* ϱ σ) = *tc-tensor* (*kf-apply* $\mathfrak{E}$ ϱ) σ›
proof −
have *inj*: ‹*inj-on* (λ(*E*, *x*). (*E* $\otimes_o$ *id-cblinfun*, ())) (*Rep-kraus-family* $\mathfrak{E}$)›
unfolding *inj-on-def* **by** (*auto simp add*: *inj-Fst-alt*[*OF id-cblinfun-not-0*])
have [*simp*]: ‹*bounded-linear* (λ*x*. *tc-tensor x* 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 ‹*kf-apply* (*kf-Fst* $\mathfrak{E}$ ::((*′c* × *′b*) *ell2*, (*′a* × *′b*) *ell2*, *unit*) *kraus-family*)
(*tc-tensor* ϱ σ)
= ($\sum_\infty$(*E*,*x*)∈*Rep-kraus-family* $\mathfrak{E}$. *sandwich-tc* (*E* $\otimes_o$ *id-cblinfun*) (*tc-tensor* ϱ σ))›
unfolding *kf-apply.rep-eq kf-Fst.rep-eq*
by (*subst infsum-reindex*[*OF inj*]) (*simp add*: *case-prod-unfold o-def*)
also have ‹. . . = ($\sum_\infty$(*E*,*x*)∈*Rep-kraus-family* $\mathfrak{E}$. *tc-tensor* (*sandwich-tc E* ϱ) σ)›
by (*simp add*: *sandwich-tc-tensor*)
finally have ‹*kf-apply* (*kf-Fst* $\mathfrak{E}$ ::((*′c* × *′b*) *ell2*, (*′a* × *′b*) *ell2*, *unit*) *kraus-family*)
(*tc-tensor* ϱ σ) = ($\sum_\infty$(*E*,*x*)∈*Rep-kraus-family* $\mathfrak{E}$. *tc-tensor* (*sandwich-tc E* ϱ) σ)›
by (*simp add*: *kf-apply-def case-prod-unfold*)
also have ‹. . . = *tc-tensor* ($\sum_\infty$(*E*,*x*)∈*Rep-kraus-family* $\mathfrak{E}$. *sandwich-tc E* ϱ) σ›

by (*subst infsum-bounded-linear*[**where** *h*=‹λ*x*. *tc-tensor x σ*›, *symmetric*])
(*use sum2* **in** ‹*auto simp add*: *o-def case-prod-unfold*›)
also have ‹. . . = *tc-tensor* (*kf-apply* 𝔈 *ϱ*) *σ*›
by (*simp add*: *kf-apply-def case-prod-unfold*)
finally show *?thesis* **by** *auto*
qed

lemma *partial-trace-ignore-trace-preserving-map-Fst*:
assumes ‹*kf-trace-preserving* 𝔈›
shows ‹*partial-trace* (*kf-apply* (*kf-Fst* 𝔈) *ϱ*) =
kf-apply 𝔈 (*partial-trace ϱ*)›
proof (*rule fun-cong*[**where** *x*=*ϱ*], *rule eq-from-separatingI2*[*OF separating-set-bounded-clinear-tc-tensor*])
show ‹*bounded-clinear* (λ*a*. *partial-trace* (*kf-apply* (*kf-Fst* 𝔈) *a*))›
by (*intro bounded-linear-intros*)
show ‹*bounded-clinear* (λ*a*. *kf-apply* 𝔈 (*partial-trace a*))›
by (*intro bounded-linear-intros*)
fix *ϱ* :: ‹('*a ell2*, '*a ell2*) *trace-class*› **and** *σ* :: ‹('*c ell2*, '*c ell2*) *trace-class*›
from *assms*
show ‹*partial-trace* (*kf-apply* (*kf-Fst* 𝔈) (*tc-tensor ϱ σ*)) =
kf-apply 𝔈 (*partial-trace* (*tc-tensor ϱ σ*))›
by (*simp add*: *kf-apply-Fst-tensor kf-apply-scaleC partial-trace-tensor*)
qed

lemma *trace-preserving-kf-Fst*:
assumes *km-trace-preserving* (*kf-apply E*)
shows *km-trace-preserving* (*kf-apply* (
kf-Fst E ::(('*a* × '*c*) *ell2*, ('*a* × '*c*) *ell2*, *unit*) *kraus-family*))
proof −
have *bounded*: *bounded-clinear* (λ*ϱ*. *trace-tc* (*kf-apply* (*kf-Fst E*) *ϱ*))
by (*simp add*: *bounded-clinear-compose kf-apply-bounded-clinear*)
have *trace*: *trace-tc* (*kf-apply* (*kf-Fst E* ::
(('*a* × '*c*) *ell2*, ('*a* × '*c*) *ell2*, *unit*) *kraus-family*) (*tc-tensor x y*)) =
trace-tc (*tc-tensor x y*) **for** *x y* **using** *assms*
apply (*simp add*: *kf-apply-Fst-tensor tc-tensor.rep-eq trace-tc.rep-eq trace-tensor km-trace-preserving-def*)

by (*metis kf-trace-preserving-def trace-tc.rep-eq*)

have (λ*ϱ*. *trace-tc* (*kf-apply* (*kf-Fst E* ::
(('*a* × '*c*) *ell2*, ('*a* × '*c*) *ell2*, *unit*) *kraus-family*) *ϱ*)) = *trace-tc*
by (*rule eq-from-separatingI2*[*OF separating-set-bounded-clinear-tc-tensor*])
(*auto simp add*: *bounded trace*)
then show *?thesis*
using *assms* **unfolding** *km-trace-preserving-def*
by (*metis kf-trace-preserving-def*)
qed

Summability on Kraus maps

lemma *finite-kf-apply-has-sum*:
assumes (*f has-sum x*) *A*

shows ((*kf-apply* $\mathfrak{F}$ *o f*) *has-sum kf-apply* $\mathfrak{F}$ *x*) *A*
unfolding *o-def* **by** (*intro has-sum-bounded-linear*[*OF - assms*])
(*auto simp add*: *bounded-clinear.bounded-linear kf-apply-bounded-clinear*)

lemma *finite-kf-apply-abs-summable-on*:
assumes *f abs-summable-on A*
shows (*kf-apply* $\mathfrak{F}$ *o f*) *abs-summable-on A*
by (*intro abs-summable-on-bounded-linear*)
(*auto simp add*: *assms bounded-clinear.bounded-linear kf-apply-bounded-clinear*)

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*

end
theory *Unitary-S*

imports *Definition-O2H*

begin

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

context *o2h-setting*
begin

1.2 Linear operator *US*

definition *Ub* :: *nat* ⇒ *'l update* **where**
Ub i = *classical-operator* (*Some o* (*flip i*))

lemma *Ub-exists*: *classical-operator-exists* (*Some* ∘ *flip i*)
by (*intro classical-operator-exists-inj*, *subst inj-map-total*)
(*simp add*: *inj-flip*)

lemma *isometry-Ub*:
isometry (*Ub k*)
unfolding *Ub-def* **by** (*auto simp add*: *inj-flip*)

lemma *Ub-ket-range*: (*Ub i* $*_V$ *ket y*) ∈ *range ket* **unfolding** *Ub-def*
by (*simp add*: *classical-operator-ket*[*OF Ub-exists*])

lemma *Ub-ket*:
Ub k $*_V$ (*ket x*) = *ket* (*flip k x*)
unfolding *Ub-def* **by** (*simp add*: *classical-operator-ket*[*OF Ub-exists*])

Using the operator *Ub*, we define the unitary U_S. Whenever we queried an element in the set S, we add a bit-flip in the register *'l*, otherwise not. The linear operator *Ub* works only on the second register part (the counting register). This is the operator between the oracle queries while running B.

definition *US* :: ‹(′*x* ⇒ *bool*) ⇒ *nat* ⇒ (′*mem* × ′*l*) *update*› **where**
‹*US S k* = *S-embed S* $\otimes_o$ *Ub k* + *not-S-embed S* $\otimes_o$ *id-cblinfun*›

lemma *isometry-US*:
isometry (*US S k*)
proof −
have *S-embed S* $\otimes_o$ *id-cblinfun* + *not-S-embed S* $\otimes_o$ *id-cblinfun* = *id-cblinfun*
by (*auto simp add*: *tensor-op-left-add*[*symmetric*])
then have ((*S-embed S*)∗ $\otimes_o$ (*Ub k*)∗ + (*not-S-embed S*)∗ $\otimes_o$ *id-cblinfun*) o_{CL}
(*S-embed S* $\otimes_o$ (*Ub k*) + *not-S-embed S* $\otimes_o$ *id-cblinfun*) = *id-cblinfun*
by (*auto simp add*: *cblinfun-compose-add-right cblinfun-compose-add-left*
comp-tensor-op S-embed-adj tensor-op-ell2 isometry-Ub not-S-embed-adj not-S-embed-idem)
then show *?thesis* **unfolding** *US-def isometry-def*
by (*auto simp add*: *cblinfun.add-left cblinfun.add-right*
tensor-ell2-ket[*symmetric*] *adj-plus tensor-op-adjoint*)
qed

lemma *US-ket-split*:
(*US S k*) $*_V$ *ket* (*x*,*y*) = (*S-embed S* $*_V$ *ket x*) $\otimes_s$ ((*Ub k*) $*_V$ *ket y*) + (*not-S-embed S* $*_V$ *ket x*) $\otimes_s$ *ket y*
unfolding *US-def*
by (*auto simp add*: *plus-cblinfun.rep-eq tensor-ell2-ket*[*symmetric*] *tensor-op-ell2*)

lemma *US-ket-only01*:
US S k (*v* $\otimes_s$ *ket n*) = (*not-S-embed S* $*_V$ *v*) $\otimes_s$ *ket n* + (*S-embed S* $*_V$ *v*) $\otimes_s$ *ket* (*flip k n*)
unfolding *US-def* **by** (*auto simp add*: *cblinfun.add-left tensor-op-ell2 Ub-ket*)

lemma *norm-US*:
assumes *i* < *Suc d* **shows** *norm* (*US S i*) = *1*
by (*simp add*: *isometry-US norm-isometry*)

Projection upto bit i

How the counting unitary *Ub* behaves with respect to projections on the counting register.

lemma *proj-classical-set-not-blog-Ub*:
assumes *n*<*d*
shows *proj-classical-set* (− *Collect blog*) o_{CL} *Ub n* =
Ub n o_{CL} *proj-classical-set* (− *Collect blog*)
proof (*intro equal-ket*, *goal-cases*)
case (*1 x*)
show *?case* **proof** (*cases blog x*)
case *True*
then show *?thesis* **by** (*simp add*: *blog-flip*[*OF* ‹*n*<*d*›] *True Ub-ket proj-classical-set-not-elem*)
next
case *False*

then show *?thesis* **by** (*simp add*: *Ub-ket not-blog-flip*[*OF* ‹*n*<*d*›] *proj-classical-set-elem*)
qed
qed

lemma *proj-classical-set-over-Ub*:
assumes *n*≤*d m*<*d*
shows *proj-classical-set* (*list-to-l* ‘ *has-bits* {*n*..<*d*}) o_{CL} *Ub m* =
Ub m o_{CL} *proj-classical-set* (*flip m* ‘ *list-to-l* ‘ *has-bits* {*n*..<*d*})
proof (*intro equal-ket*, *goal-cases*)
case (*1 x*)
show *?case* **proof** (*cases blog x*)
case *True*
obtain *j* **where** *j*∈*len-d-lists* **and** *x*: *x* = *list-to-l j*
by (*metis True image-iff mem-Collect-eq surj-list-to-l*)
consider (*elem*) *flip m x* ∈ *list-to-l* ‘ *has-bits* {*n*..<*d*} |
(*not-elem*) ¬ *flip m x* ∈ *list-to-l* ‘ *has-bits* {*n*..<*d*} **by** *auto*
then have *?thesis* **if** *n*<*d* **proof** (*cases*)
case *elem*
then have *x-in*: *x* ∈ *flip m* ‘ *list-to-l* ‘ *has-bits* {*n*..<*d*}
using *True assms*(*2*) *flip-flip* **by** *fastforce*
have (*proj-classical-set* (*list-to-l* ‘ *has-bits* {*n*..<*d*}) o_{CL} *Ub m*) $*_V$ *ket x* = *ket* (*flip m x*)
by (*simp add*: *Ub-ket local.elem proj-classical-set-elem*)
moreover have (*Ub m* o_{CL} *proj-classical-set* (*flip m* ‘ *list-to-l* ‘ *has-bits* {*n*..<*d*})) $*_V$ *ket*
x =
ket (*flip m x*)
using *proj-classical-set-elem*[*OF x-in*] **by** (*auto simp add*: *Ub-ket*)
ultimately show *?thesis* **by** *metis*
next
case *not-elem*
then have *x-in*: *x* ∉ *flip m* ‘ *list-to-l* ‘ *has-bits* {*n*..<*d*} **using** *flip-flip*[*OF* ‹*m*<*d*›]
by (*metis True imageE inj-def inj-flip*)
have (*proj-classical-set* (*list-to-l* ‘ *has-bits* {*n*..<*d*}) o_{CL} *Ub m*) $*_V$ *ket x* = *0*
by (*simp add*: *Ub-ket not-elem proj-classical-set-not-elem*)
moreover have (*Ub m* o_{CL} *proj-classical-set* (*flip m* ‘ *list-to-l* ‘ *has-bits* {*n*..<*d*})) $*_V$ *ket*
x = *0*
using *proj-classical-set-not-elem*[*OF x-in*] **by** (*auto simp add*: *Ub-ket*)
ultimately show *?thesis* **by** *metis*
qed
moreover have *?thesis* **if** *n*=*d* **unfolding** *that* **using** *True*
by (*auto simp add*: *Ub-ket proj-classical-set-not-elem*)
ultimately show *?thesis* **using** *assms* **by** *linarith*
next
case *False*
have *proj-classical-set* (*list-to-l* ‘ *has-bits* {*n*..<*d*}) $*_V$ *ket* (*flip m x*) = *0*
by (*intro proj-classical-set-not-elem*)(*metis* (*no-types*, *lifting*) *False*
image-iff mem-Collect-eq not-blog-flip[*OF* ‹*m*<*d*›] *has-bits-def surj-list-to-l*)
then have *left*: (*proj-classical-set* (*list-to-l* ‘ *has-bits* {*n*..<*d*}) o_{CL} *Ub m*) $*_V$ *ket x* = *0*

by (*auto simp add*: *Ub-ket*)
have *right*: *proj-classical-set* (*flip m* ‘ *list-to-l* ‘ *has-bits* {*n*..<*d*}) $*_V$ *ket x* = *0*
by (*intro proj-classical-set-not-elem*) (*smt* (*verit*, *del-insts*) *False assms*(*2*) *blog.simps imageE image-eqI mem-Collect-eq has-bits-def surj-list-to-l*)
show *?thesis* **unfolding** *left* **using** *right* **by** *auto*
qed
qed

lemma Ψ*s-US-Proj-ket-upto*:
assumes *i*<*d*
shows *tensor-ell2-right* (*ket empty*)∗ o_{CL} ((*US S i*) o_{CL} *Proj-ket-upto* (*has-bits-upto i*)) = *not-S-embed S* o_{CL} *tensor-ell2-right* (*ket empty*)∗
proof (*intro equal-ket*, *safe*, *goal-cases*)
case (*1 a b*)
have *split-a*: *ket a* = *S-embed S* (*ket a*) + *not-S-embed S* (*ket a*)
using *S-embed-not-S-embed-add* **by** *auto*
have *?case* (**is** *?left* = *?right*) **if** *b* ∈ *list-to-l* ‘ *has-bits-upto i*
proof −
have *Proj* (*ccspan* (*ket* ‘ *list-to-l* ‘ *has-bits-upto i*)) $*_V$ *ket b* = *ket b*
using *that* **by** (*simp add*: *Proj-fixes-image ccspan-superset*′)
then have *proj*: *proj-classical-set* (*list-to-l* ‘ *has-bits-upto i*) $*_V$ *ket b* = *ket b*
unfolding *proj-classical-set-def* **by** *auto*
have *?left* = (*tensor-ell2-right* (*ket empty*)∗ o_{CL} (*US S i*)) $*_V$ *ket* (*a*, *b*)
using *proj* **by** (*auto simp add*: *Proj-ket-upto-def Proj-ket-set-def tensor-ell2-ket*[*symmetric*]

tensor-op-ell2)
also have . . . = *tensor-ell2-right* (*ket empty*)∗ $*_V$
((*S-embed S* $*_V$ *ket a*) $\otimes_s$ (*Ub i*) $*_V$ *ket b* + ((*not-S-embed S* $*_V$ *ket a*) $\otimes_s$ *ket b*))
using *US-ket-split* **by** *auto*
also have . . . = *tensor-ell2-right* (*ket empty*)∗ $*_V$ ((*not-S-embed S* $*_V$ *ket a*) $\otimes_s$ *ket b*)
proof −
obtain *bs* **where** *b*: *bs*∈ *has-bits-upto i b* = *list-to-l bs*
using ‹*b* ∈ *list-to-l* ‘ *has-bits-upto i*› **by** *auto*
then have *bs*: *length bs* = *d* ¬ (*bs*!(*d*−*i*−*1*)) **unfolding** *has-bits-upto-def len-d-lists-def*
using *assms b*(*1*) *has-bits-upto-elem* **by** *auto*
then have *bit b i* = *bit empty i* **unfolding** *b*(*2*)
by (*subst bit-list-to-l*) (*auto simp add*: ‹*i*<*d*›)
then have *flip i b* ≠ *empty* **using** *assms bit-flip-same blog.intros*(*1*) *not-blog-flip* **by** *blast*
then have *tensor-ell2-right* (*ket empty*)∗ $*_V$ ((*S-embed S* $*_V$ *ket a*) $\otimes_s$ (*Ub i*) $*_V$ *ket b*) =
0
by (*simp add*: *Ub-def classical-operator-ket*[*OF Ub-exists*])
then show *?thesis* **by** (*simp add*: *cblinfun.real.add-right*)
qed

also have . . . = *?right*
by (*auto simp add*: *tensor-ell2-ket*[*symmetric*] *cinner-ket*)
finally show *?thesis* **by** *blast*
qed
moreover have *?case* (**is** *?left* = *?right*) **if** ¬ (*b* ∈ *list-to-l* ‘ *has-bits-upto i*) *blog b*

```
  proof –
    have b ≠ empty using that empty-list-to-l-has-bits-upto by force
    have b ∉ list-to-l ` has-bits-upto i using that by auto
    then have proj: proj-classical-set (list-to-l ` has-bits-upto i) *V ket b = 0
      unfolding proj-classical-set-def by (intro Proj-0-compl, intro mem-ortho-ccspanI) auto
    then have ?left = 0
      by (auto simp add: Proj-ket-upto-def Proj-ket-set-def tensor-ell2-ket[symmetric]
          tensor-op-ell2)
    moreover have ?right = 0 using ‹b ≠ empty›
      by (auto simp add: tensor-ell2-ket[symmetric] cinner-ket)
    ultimately show ?thesis by auto
  qed
  moreover have ?case (is ?left = ?right) if ¬ blog b
  proof –
    have b ≠ empty using that blog.intros(1) by auto
    have b ∉ list-to-l ` has-bits-upto i
      using has-bits-upto-def surj-list-to-l that by fastforce
    then have proj: proj-classical-set (list-to-l ` has-bits-upto i) *V ket b = 0
      unfolding proj-classical-set-def by (intro Proj-0-compl, intro mem-ortho-ccspanI) auto
    then have ?left = 0
      by (auto simp add: Proj-ket-upto-def Proj-ket-set-def tensor-ell2-ket[symmetric]
          tensor-op-ell2)
    moreover have ?right = 0 using ‹b ≠ empty›
      by (auto simp add:  tensor-ell2-ket[symmetric] cinner-ket)
    ultimately show ?thesis by auto
  qed
  ultimately show ?case by (cases b ∈ list-to-l ` has-bits-upto i, auto)
qed

end

unbundle no cblinfun-syntax
unbundle no lattice-syntax


end
theory Unitary-S-prime
  imports Definition-O2H
begin

unbundle cblinfun-syntax
unbundle lattice-syntax

context o2h-setting
begin
```

1.3 Towards the Definition of *U-S'*

For the definition of *U-S'*, we need a counting function on the additional register. We model this with the function *c-add* that works on *nat* as a addition of 1 modulo $d + 1$ (as long as we stay under the query depth d) and as an identity otherwise.

definition *c-add* :: *nat* ⇒ *nat* **where**
  c-add c = (*if c*<*d*+*1 then* (*c*+*1*) *mod* (*d*+*1*) *else c*)

lemma *surj-c-add-c-valid*: *c-add* ' {..<*d*+*1*} = {..<*d*+*1*}
proof −
  have *x* ∈ (λ*x*. *Suc x mod Suc d*) ' {..<*Suc d*} **if** *x* < *Suc d* **for** *x*
  proof (*cases x*=*0*)
    case *True*
    then show *?thesis* **by** (*simp add*: *True lessThan-Suc*)
  next
    case *False*
    then show *?thesis* **by** (*intro image-eqI*[*of* - - *x*−*1*])(*use False that* **in** ‹*auto*›)
  qed
  then show *?thesis* **unfolding** *c-add-def* **by** *auto*
qed

c-add needs to be bijective, so that the resulting operator is unitary.

lemma *inj-c-add*: *inj c-add*
proof −
  have *x* = *y* **if** *c-add x* = *c-add y x*<*d*+*1 y*<*d*+*1* **for** *x y*
  proof −
    have *c-add x* = (*if x* = *d then 0 else x*+*1*) **using** *that c-add-def* **by** *force*
    moreover have *c-add y* = (*if y* = *d then 0 else y*+*1*) **using** *that c-add-def* **by** *force*
    ultimately have (*if x* = *d then 0 else x*+*1*) = (*if y* = *d then 0 else y*+*1*) **using** *that* **by** *auto*
    then show *?thesis* **by** (*metis Suc-eq-plus1 diff-add-inverse2 nat.simps*(*3*))
  qed
  moreover have *False* **if** *c-add x* = *c-add y x*<*d*+*1 y*≥*d*+*1* **for** *x y*
    using *c-add-def surj-c-add-c-valid that*
    by (*metis add-pos-nonneg d-gr-0 mod-less-divisor not-less zero-less-one-class.zero-le-one*)
  moreover have *x* = *y* **if** *c-add x* = *c-add y x*≥*d*+*1 y*≥*d*+*1* **for** *x y*
    using *c-add-def that* **by** *auto*
  ultimately show *?thesis* **unfolding** *inj-def* **by** (*metis not-less*)
qed

lemma *surj-c-add*: *c-add* ' *UNIV* = *UNIV*
  using *surj-c-add-c-valid c-add-def* **by** *auto*

lemma *bij-c-add*: *bij c-add*
  by (*subst* (*2*) *surj-c-add*[*symmetric*])
    (*auto simp add*: *inj-c-add intro*: *inj-on-imp-bij-betw*)

lemma *c-add-0*: *c-add 0* ≠ *0*

unfolding *c-add-def* **by** (*simp add*: *d-gr-0*)

Finally, we can define the operator for the adversary B_{count}.

definition *Uc* = *classical-operator* (*Some o c-add*)

lemma *Uc-exists*:
classical-operator-exists (*Some o c-add*)
by (*intro classical-operator-exists-inj*, *subst inj-map-def*)
(*use inj-c-add* **in** ‹*auto simp add*: *inj-on-def*›)

lemma *unitary-Uc*:
unitary Uc
unfolding *Uc-def* **by** (*auto intro*!: *unitary-classical-operator simp add*: *bij-c-add*)

lemma *Uc-ket-d*:
$Uc *_V$ *ket d* = *ket 0*
unfolding *Uc-def* **by** (*subst classical-operator-ket*[*OF Uc-exists*])
(*simp add*: *c-add-def Suc-lessD*)

lemma *Uc-ket-less*:
assumes *n*<*d*
shows $Uc *_V$ *ket n* = *ket* (*n*+*1*)
unfolding *Uc-def* **by** (*subst classical-operator-ket*[*OF Uc-exists*])
(*simp add*: *c-add-def Suc-lessD assms*)

lemma *Uc-ket-leq*:
assumes *n*<*d*+*1*
shows $Uc *_V$ *ket n* = *ket* ((*n*+*1*) *mod* (*d*+*1*))
proof (*cases n*=*d*)
case *True* **show** *?thesis* **by** (*use Uc-ket-d* **in** ‹*auto simp add*: *True*›)
next
case *False* **show** *?thesis* **by** (*use Uc-ket-less assms False* **in** ‹*auto*›)
qed

lemma *Uc-ket-greater*:
assumes *n*>*d*
shows $Uc *_V$ *ket n* = *ket n*
unfolding *Uc-def* **by** (*subst classical-operator-ket*[*OF Uc-exists*])
(*use c-add-def assms* **in** ‹*auto*›)

lemma *Uc-ket-range*:
($Uc *_V$ *ket y*) ∈ *range ket* **unfolding** *Uc-def*
by (*subst classical-operator-ket*[*OF Uc-exists*])(*auto*)

lemma *Uc-ket-range-valid*:
assumes *y*<*d*+*1*
shows ($Uc *_V$ *ket y*) ∈ *ket* ‘ {..<*d*+*1*} **unfolding** *Uc-def*
by (*subst classical-operator-ket*[*OF Uc-exists*])(*use assms* **in** ‹*auto simp add*: *c-add-def*›)

Using the operator *Uc*, we define the unitary U'_S. Whenever, we queried an element in the set S, we add a count in the counting register, otherwise not. The linear operator *Uc* works only on the second register part (the counting register).

definition *U-S′* :: ‹(*′x* ⇒ *bool*) ⇒ (*′mem* × *nat*) *update*› **where**
‹*U-S′ S* = *S-embed S* $\otimes_o$ *Uc* + *not-S-embed S* $\otimes_o$ *id-cblinfun*›

lemma *unitary-U-S′*:
unitary (*U-S′ S*)
unfolding *U-S′-def unitary-def* **proof** (*safe*, *goal-cases*)
case *1*
have *S-embed S* $\otimes_o$ *id-cblinfun* + *not-S-embed S* $\otimes_o$ *id-cblinfun* = *id-cblinfun*
by (*auto simp add*: *tensor-op-left-add*[*symmetric*])
then have ((*S-embed S*)∗ $\otimes_o$ *Uc*∗ + (*not-S-embed S*)∗ $\otimes_o$ *id-cblinfun*) o_{CL}
(*S-embed S* $\otimes_o$ *Uc* + *not-S-embed S* $\otimes_o$ *id-cblinfun*) = *id-cblinfun*
by (*auto simp add*: *cblinfun-compose-add-right cblinfun-compose-add-left comp-tensor-op S-embed-adj tensor-op-ell2 unitary-Uc not-S-embed-adj not-S-embed-idem*)
then show *?case* **by** (*auto simp add*: *cblinfun.add-left cblinfun.add-right tensor-ell2-ket*[*symmetric*] *adj-plus tensor-op-adjoint*)
next
case *2*
have *S-embed S* $\otimes_o$ *id-cblinfun* + *not-S-embed S* $\otimes_o$ *id-cblinfun* = *id-cblinfun*
by (*auto simp add*: *tensor-op-left-add*[*symmetric*])
then have (*S-embed S* $\otimes_o$ *Uc* + *not-S-embed S* $\otimes_o$ *id-cblinfun*) o_{CL}
((*S-embed S*)∗ $\otimes_o$ *Uc*∗ + (*not-S-embed S*)∗ $\otimes_o$ *id-cblinfun*) = *id-cblinfun*
by (*auto simp add*: *cblinfun-compose-add-right cblinfun-compose-add-left comp-tensor-op S-embed-adj tensor-op-ell2 unitary-Uc not-S-embed-adj not-S-embed-idem*)
then show *?case* **by** (*auto simp add*: *cblinfun.add-left cblinfun.add-right tensor-ell2-ket*[*symmetric*] *adj-plus tensor-op-adjoint*)
qed

lemma *iso-U-S′*: *isometry* (*U-S′ S*)
by (*simp add*: *unitary-U-S′*)

lemma *U-S′-ket-split*:
U-S′ S $*_V$ *ket* (*x*,*y*) = (*S-embed S* $*_V$ *ket x*) $\otimes_s$ (*Uc* $*_V$ *ket y*) + (*not-S-embed S* $*_V$ *ket x*) $\otimes_s$ *ket y*
unfolding *U-S′-def*
by (*auto simp add*: *plus-cblinfun.rep-eq tensor-ell2-ket*[*symmetric*] *tensor-op-ell2*)

lemma *norm-U-S′*:
assumes *i* < *Suc d* **shows** *norm* (*U-S′ S*) = *1*
by (*simp add*: *iso-U-S′ norm-isometry*)

We ensure that the Φs is the same as the left part of Ψ_{count} (ie. *run-B-count*) with right part $|\,0\rangle$.

lemma *Ψs-U-S′-Proj-ket-upto*:
assumes *i*<*d*
shows *tensor-ell2-right* (*ket 0*)∗ o_{CL} (*U-S′ S* o_{CL} *Proj-ket-set* {..<*i*+*1*}) =

not-S-embed S o_{CL} *tensor-ell2-right* (*ket 0*)∗
proof (*intro equal-ket*, *safe*, *goal-cases*)
case (*1 a b*)
have *split-a*: *ket a* = *S-embed S* (*ket a*) + *not-S-embed S* (*ket a*)
using *S-embed-not-S-embed-add* **by** *auto*
have (*tensor-ell2-right* (*ket 0*)∗ o_{CL} (*U-S′ S* o_{CL} *Proj-ket-set* {..<*i*+*1*})) $*_V$ *ket* (*a*, *b*) =
(*not-S-embed S* o_{CL} *tensor-ell2-right* (*ket 0*)∗) $*_V$ *ket* (*a*, *b*) (**is** *?left* = *?right*)
if *b* < *i*+*1*
proof −
have *b* < *d* **using** *assms that* **by** *linarith*
then have *c-add b* ≠ *0* **unfolding** *c-add-def* **using** *c-add-0 c-add-def not-less-eq* **by** *force*
have *proj*: *proj-classical-set* {..<*i*+*1*} $*_V$ *ket b* = *ket b*
using *that* **unfolding** *proj-classical-set-def* **by** (*simp add*: *Proj-fixes-image ccspan-superset′*)
have *?left* = (*tensor-ell2-right* (*ket 0*)∗ o_{CL} *U-S′ S*) $*_V$ *ket* (*a*, *b*)
using *proj* **by** (*auto simp add*: *Proj-ket-set-def tensor-ell2-ket*[*symmetric*] *tensor-op-ell2*)
also have . . . = *tensor-ell2-right* (*ket 0*)∗ $*_V$
((*S-embed S* $*_V$ *ket a*) $\otimes_s$ *Uc* $*_V$ *ket b* + ((*not-S-embed S* $*_V$ *ket a*) $\otimes_s$ *ket b*))
using *U-S′-ket-split* **by** *auto*
also have . . . = *tensor-ell2-right* (*ket 0*)∗ $*_V$ ((*not-S-embed S* $*_V$ *ket a*) $\otimes_s$ *ket b*)
proof −
have *tensor-ell2-right* (*ket 0*)∗ $*_V$ ((*S-embed S* $*_V$ *ket a*) $\otimes_s$ *Uc* $*_V$ *ket b*) = *0*
by (*simp add*: *Uc-ket-less* ‹*b* < *d*›)
then show *?thesis* **by** (*simp add*: *cblinfun.real.add-right*)
qed

also have . . . = *?right*
by (*auto simp add*: *tensor-ell2-ket*[*symmetric*] *cinner-ket*)
finally show *?thesis* **by** *blast*
qed
moreover have (*tensor-ell2-right* (*ket 0*)∗ o_{CL} (*U-S′ S* o_{CL} *Proj-ket-set* {..<*i*+*1*})) $*_V$ *ket*
(*a*, *b*) =
(*not-S-embed S* o_{CL} *tensor-ell2-right* (*ket 0*)∗) $*_V$ *ket* (*a*, *b*) (**is** *?left* = *?right*)
if ¬ (*b* < *i*+*1*)
proof −
have *b*≠*0* **using** *that* **by** *auto*
have *proj*: *Proj* (*ccspan* (*ket* ‘ {..<*i*+*1*})) $*_V$ *ket b* = *0*
using *that*
by (*metis lessThan-iff proj-classical-set-def proj-classical-set-not-elem*)
then have *?left* = *0* **by** (*auto simp add*: *Proj-ket-set-def tensor-ell2-ket*[*symmetric*]
tensor-op-ell2 proj-classical-set-def)
moreover have *?right* = *0* **by** (*auto simp add*: ‹*b*≠*0*› *tensor-ell2-ket*[*symmetric*] *cinner-ket*)
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** (*cases b* < *i*+*1*, *auto*)
qed

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*

end
theory *Run-Adversary*

imports *Definition-O2H*
More-Kraus-Maps
Unitary-S
Unitary-S-prime

begin

unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
unbundle *register-syntax*

2 Running the Adversary

Modelling the adversary, some type synonyms.

type-synonym $'a$ *tc-op* = ($'a$ *ell2*,$'a$ *ell2*) *trace-class*
type-synonym $'a$ *kraus-adv* = *nat* ⇒ ($'a$ *ell2*,$'a$ *ell2*,*unit*) *kraus-family*
type-synonym $'a$ *pure-adv* = *nat* ⇒ (*nat* ⇒ $'a$ *update*) ⇒ $'a$ *ell2*

type-synonym $'a$ *mixed-adv* = *nat* ⇒ $'a$ *kraus-adv* ⇒ $'a$ *update*

We define the run of the quantum algorithm of our adversaries. Each adversary can make quantum calculations (in form of unitaries) before and after each query to the oracle *Uquery H*. Since the oracle function $H : X \longrightarrow Y$ works on (classical) registers, we need to embed the domain and target registers X and Y as well. (($X;Y$) (*Uquery H*) is the notation for the query to H applied to the registers X and Y.) *init* is the initial quantum state which may also be manipulated by the adversary in the first step.

Running the adversary with Uquerys

Definitions for pure adversaries

fun *run-pure-adv* :: *nat* ⇒ (*nat* ⇒ $'a$ *update*) ⇒ (*nat* ⇒ $'a$ *update*) ⇒ $'a$ *ell2* ⇒- ⇒ - ⇒ - ⇒ $'a$ *ell2* **where**
run-pure-adv 0 UAs UB init X Y H = (*UAs 0*) $*_V$ *init*
| *run-pure-adv* (*Suc n*) *UAs UB init X Y H* =
(*UAs* (*Suc n*)) $*_V$ ($X;Y$) (*Uquery H*) $*_V$ (*UB n*) $*_V$ (*run-pure-adv n UAs UB init X Y H*)

fun *run-pure-adv-update* :: *nat* ⇒ (*nat* ⇒ $'a$ *update*) ⇒ (*nat* ⇒ $'a$ *update*) ⇒ $'a$ *ell2* ⇒- ⇒ - ⇒ - ⇒ $'a$ *update* **where**
run-pure-adv-update 0 UAs UB init X Y H = *sandwich* (*UAs 0*) $*_V$ (*selfbutter init*)
| *run-pure-adv-update* (*Suc n*) *UAs UB init X Y H* =

sandwich (*UAs* (*Suc n*) o_{CL} (*X*;*Y*) (*Uquery H*) o_{CL} *UB n*) $*_V$ (*run-pure-adv-update n UAs UB init X Y H*)

fun *run-pure-adv-tc* :: *nat* ⇒ (*nat* ⇒ *'a update*) ⇒ (*nat* ⇒ *'a update*) ⇒ *'a ell2* ⇒- ⇒ - ⇒ - ⇒ *'a tc-op* **where**
run-pure-adv-tc 0 UAs UB init X Y H = *sandwich-tc* (*UAs 0*) (*tc-selfbutter init*)
| *run-pure-adv-tc* (*Suc n*) *UAs UB init X Y H* =
sandwich-tc (*UAs* (*Suc n*) o_{CL} (*X*;*Y*) (*Uquery H*) o_{CL} *UB n*) (*run-pure-adv-tc n UAs UB init X Y H*)

lemma *run-pure-adv-tc-pos*:
run-pure-adv-tc n UAs UB init X Y H ≥ *0*
by (*induct n*) (*auto simp add*: *Abs-trace-class-geq0I sandwich-tc-pos tc-selfbutter-def*)

How a pure run is mapped from ell2 to trace class.

lemma *run-pure-adv-ell2-update*:
run-pure-adv-update n UAs UB init X Y H = *selfbutter* (*run-pure-adv n UAs UB init X Y H*)
by (*induct n*) (*auto simp add*: *butterfly-comp-cblinfun cblinfun-comp-butterfly sandwich-apply*)

lemma *run-pure-adv-update-tc'*:
from-trace-class (*run-pure-adv-tc n UAs UB init X Y H*) = *run-pure-adv-update n UAs UB init X Y H*
by (*induct n*) (*auto simp add*: *Abs-trace-class-inverse from-trace-class-sandwich-tc tc-butterfly.abs-eq tc-selfbutter-def*)

lemma *run-pure-adv-update-tc*:
run-pure-adv-tc n UAs UB init X Y H = *Abs-trace-class* (*run-pure-adv-update n UAs UB init X Y H*)
using *Abs-trace-class-inverse* **unfolding** *run-pure-adv-update-tc'*[*symmetric*] *from-trace-class-inverse*

by *auto*

Definitions for mixed adversaries

fun *run-mixed-adv* ::
nat ⇒ *'a kraus-adv* ⇒ (*nat* ⇒ *'a update*) ⇒ *'a ell2* ⇒ - ⇒ - ⇒ - ⇒ *'a tc-op* **where**
run-mixed-adv 0 Es UB init X Y H = (*kf-apply* (*Es 0*)) (*tc-selfbutter init*)
| *run-mixed-adv* (*Suc n*) *Es UB init X Y H* = (*kf-apply* (*Es* (*Suc n*)))
(*sandwich-tc* ((*X*;*Y*) (*Uquery H*) o_{CL} *UB n*) (*run-mixed-adv n Es UB init X Y H*))

lemma *run-mixed-adv-pos*:
run-mixed-adv n Es UB init X Y H ≥ *0*
by (*induct n*) (*auto simp add*: *Abs-trace-class-geq0I sandwich-tc-pos kf-apply-pos tc-selfbutter-def*)

lemma (**in** *o2h-setting*) *norm-run-mixed-adv*:
assumes *Es-norm-id*: ⋀*i*. *i* < *d*+*1* ⟹ *kf-bound* (*Es i*) ≤ *id-cblinfun*

and *n*: *n* < *d+1*
and *normUB*: $\bigwedge$*i*. *i* < *d+1* $\Longrightarrow$ *norm* (*UB i*) ≤ *1*
and *register*: *register* (*X′*;*Y′*)
and *norm-init′*: *norm init′* = *1*
fixes *H* :: *′x* ⇒ *′y*
shows *norm* (*run-mixed-adv n Es UB init′ X′ Y′ H*) ≤ *1*
using *n* **proof** (*induct n*)
case *0*
have *norm1*:*norm* (*tc-selfbutter init′*) = *1* **using** *norm-init′*
by (*simp add*: *norm-tc-butterfly tc-selfbutter-def*)
have *init0*: *0* ≤ *tc-selfbutter init′* **by** (*simp add*: *tc-selfbutter-def*)
have *norm* (*run-mixed-adv 0 Es UB init′ X′ Y′ H*) ≤ *kf-norm* (*Es 0*)
using *kf-apply-bounded-pos*[*OF init0*] *norm1* **by** *auto*
also have ... ≤ *1* **unfolding** *kf-norm-def* **using** *Es-norm-id*[*of 0*]
by (*metis 0.prems Kraus-Families.kf-bound-pos norm-cblinfun-id norm-cblinfun-mono*)
finally show *?case* **by** *auto*
next
case (*Suc n*)
have *uniH*: *unitary* ((*X′*;*Y′*) (*Uquery H*)) **by** (*intro register-unitary*[*OF register unitary-H*])
let *?sand* = *sandwich-tc* ((*X′*;*Y′*) (*Uquery H*) o_{CL} *UB n*)
have *pos*: *?sand* (*run-mixed-adv n Es UB init′ X′ Y′ H*) ≥ *0*
using *sandwich-tc-pos*[*OF run-mixed-adv-pos*] **by** *blast*
have *norm* (*kf-apply* (*Es* (*Suc n*)) (*?sand* (*run-mixed-adv n Es UB init′ X′ Y′ H*))) ≤
kf-norm (*Es* (*Suc n*)) ∗ *norm* (*?sand* (*run-mixed-adv n Es UB init′ X′ Y′ H*))
using *kf-apply-bounded-pos*[*OF pos*] **by** *auto*
also have ... ≤ *norm* (*?sand* (*run-mixed-adv n Es UB init′ X′ Y′ H*))
unfolding *kf-norm-def* **using** *Es-norm-id*[*OF Suc*(*2*)]
by (*metis Kraus-Families.kf-bound-pos mult-left-le-one-le norm-cblinfun-id norm-cblinfun-mono norm-ge-zero*)
also have ... ≤ (*norm* ((*X′*;*Y′*) (*Uquery H*) o_{CL} *UB n*))^*2*
using *norm-sandwich-tc Suc* **by** (*smt* (*verit*, *best*) *Suc-lessD mult-less-cancel-left1 zero-le-power2*)
also have ... ≤ (*norm* ((*X′*;*Y′*) (*Uquery H*)))^*2* ∗ (*norm* (*UB n*))^*2*
by (*metis norm-cblinfun-compose norm-ge-zero power-mono power-mult-distrib*)
also have ... ≤ (*norm* (*UB n*))^*2* **by** (*simp add*: *norm-isometry uniH*)
also have ... ≤ *1* **using** *Suc*(*2*) *normUB*[*of n*] **by** (*auto simp add*: *power-le-one*)
finally show *?case* **by** *auto*
qed

Trace preserving Kraus maps/adversaries preserve the norm.

lemma *km-trace-preserving-kf-apply*:
assumes *km-trace-preserving* (*kf-apply F*) *ϱ* ≥ *0*
shows *norm* (*kf-apply F ϱ*) = *norm ϱ*
by (*metis assms*(*1*,*2*) *kf-apply-pos km-trace-preserving-iff norm-tc-pos-Re*)

lemma (**in** *o2h-setting*) *trace-preserving-norm-run-mixed-adv*:
assumes *trace-pres*: $\bigwedge$*i*. *i* < *d+1* $\Longrightarrow$ *km-trace-preserving* (*kf-apply* (*Es i*))
and *n*: *n* < *d+1*
and *iso-UB*: $\bigwedge$*i*. *i* < *d+1* $\Longrightarrow$ *isometry* (*UB i*)

and *register*: *register* $(X';Y')$
and *norm-init'*: *norm init'* = *1*
fixes H :: $'x \Rightarrow 'y$
shows *norm* (*run-mixed-adv n Es UB init' X' Y' H*) = *1*
using *n* **proof** (*induct n*)
case *0*
have *norm* (*kf-apply* (*Es 0*) (*tc-selfbutter init'*)) = *norm* (*tc-selfbutter init'*)
using *assms*(*1*)
by (*auto intro*!: *km-trace-preserving-kf-apply simp add*: *tc-selfbutter-def*)
then show *?case* **using** *assms* **by** *auto*
next
case (*Suc n*)
have *n<d+1* **using** *Suc* **by** *auto*
have *norm* (*kf-apply* (*Es* (*Suc n*))
(*sandwich-tc* ((X';Y') (*Uquery H*) o_{CL} *UB n*) (*run-mixed-adv n Es UB init' X' Y' H*)))
=
norm (*sandwich-tc* ((X';Y') (*Uquery H*) o_{CL} *UB n*) (*run-mixed-adv n Es UB init' X' Y' H*))
using *assms*(*1*)[*OF Suc*(*2*)]
by (*auto intro*!: *km-trace-preserving-kf-apply simp add*: *tc-selfbutter-def run-mixed-adv-pos sandwich-tc-pos*)
also have ... = *norm* (*run-mixed-adv n Es UB init' X' Y' H*)
by (*intro norm-sandwich-tc-unitary*) (*use iso-UB*[*OF* ‹*n<d+1*›]
unitary-isometry[*OF register-unitary*[*OF register unitary-H*]]
in ‹*auto simp add*: *run-mixed-adv-pos intro*!: *isometry-cblinfun-compose*›)
finally show *?case* **using** *Suc* **by** *auto*
qed

context *o2h-setting*
begin

The run of the adversaries A and B (= A with counting in register 'l) for mixed states.

For pure adversaries

definition *run-pure-A-ell2* **where**
run-pure-A-ell2 UA H = *run-pure-adv d UA* (λ-. *id-cblinfun*) *init X Y H*

definition *run-pure-A-update* :: (*nat* ⇒ '*mem update*) ⇒ ('*x* ⇒ '*y*) ⇒ '*mem update* **where**
run-pure-A-update UA H = *run-pure-adv-update d UA* (λ-. *id-cblinfun*) *init X Y H*

definition *run-pure-A-tc* :: (*nat* ⇒ '*mem update*) ⇒ ('*x* ⇒ '*y*) ⇒ '*mem tc-op* **where**
run-pure-A-tc UA H = *run-pure-adv-tc d UA* (λ-. *id-cblinfun*) *init X Y H*

lemma *run-pure-A-ell2-update*:
run-pure-A-update UA H = *selfbutter* (*run-pure-A-ell2 UA H*)
unfolding *run-pure-A-update-def run-pure-A-ell2-def* **by** (*rule run-pure-adv-ell2-update*)

lemma *run-pure-A-update-tc*:
  run-pure-A-tc UA H = *Abs-trace-class* (*run-pure-A-update UA H*)
  unfolding *run-pure-A-update-def run-pure-A-tc-def* **by** (*rule run-pure-adv-update-tc*)

lemma *run-pure-A-tc-pos*: *0* ≤ *run-pure-A-tc UA H*
  unfolding *run-pure-A-tc-def* **by** (*rule run-pure-adv-tc-pos*)

For mixed adversaries

definition *run-mixed-A* :: *'mem kraus-adv* ⇒ (*'x* ⇒ *'y*) ⇒ *'mem tc-op* **where**
  run-mixed-A kraus-A H = *run-mixed-adv d kraus-A* (λ-. *id-cblinfun*) *init X Y H*

lemma *run-mixed-A-pos*:
  0 ≤ *run-mixed-A kraus-A H*
  unfolding *run-mixed-A-def* **by** (*rule run-mixed-adv-pos*)

lemma *norm-run-mixed-A*:
  assumes *F-norm-id*: ⋀*i*. *i* < *d*+*1* ⟹ *kf-bound* (*F i*) ≤ *id-cblinfun*
  shows *norm* (*run-mixed-A F H*) ≤ *1*
  unfolding *run-mixed-A-def*
  by (*intro norm-run-mixed-adv*) (*auto simp add*: *norm-init assms*)

Embeddings of X and Y in the counting register of B

definition *X-for-B* :: ‹*'x update* ⇒ (*'mem* × *'l*) *update*› **where**
  ‹*X-for-B* = *Fst o X*›

definition *Y-for-B* :: ‹*'y update* ⇒ (*'mem* × *'l*) *update*› **where**
  ‹*Y-for-B* = *Fst o Y*›

lemma [*register*]: ‹*register X-for-B*›
  by (*simp add*: *X-for-B-def*)

lemma [*register*]: ‹*register Y-for-B*›
  by (*simp add*: *Y-for-B-def*)

lemma *register-XY-for-B*:
  register (*X-for-B*;*Y-for-B*) **by** (*simp add*: *X-for-B-def Y-for-B-def*)

Alternative representation of *Uquery H* on *'l*

lemma *UqueryH-tensor-id-cblinfunB*:
  (*X-for-B*;*Y-for-B*) (*Uquery H*) = (*X*;*Y*) (*Uquery H*) $\otimes_o$ *id-cblinfun*
  unfolding *X-for-B-def Y-for-B-def*
  by (*metis Fst-def comp-eq-dest-lhs compat register-Fst register-comp-pair*)

The oracle query on the extended register stays unitary.

lemma *unitary-H-B*: *unitary* ((*X-for-B*;*Y-for-B*) (*Uquery H*))
  by (*intro register-unitary*[*OF register-XY-for-B*]) (*auto simp add*: *unitary-H*)

lemma *iso-H-B*: *isometry* ((*X-for-B*;*Y-for-B*) (*Uquery H*))
by (*simp add*: *unitary-H-B*)

The initial register state *init* is extended by zeros in the register $'l$. Here, we need the embedding of the counter into the type $'l$ by *list-to-l*.

definition ‹*init-B* = *init* $\otimes_s$ *ket empty*›

lemma *norm-init-B*: *norm init-B* = *1*
unfolding *init-B-def* **using** *norm-init* **by** (*simp add*: *norm-tensor-ell2*)

Definition of adversary B

For pure adversaries

definition *run-pure-B-ell2* **where**
run-pure-B-ell2 UB H S =
run-pure-adv d (*Fst o UB*) (*US S*) *init-B X-for-B Y-for-B H*

definition *run-pure-B-update* ::
(*nat* ⇒ *'mem update*) ⇒ (*'x* ⇒ *'y*) ⇒ (*'x* ⇒ *bool*) ⇒ (*'mem* × *'l*) *update* **where**
run-pure-B-update UB H S = *run-pure-adv-update d* (*Fst o UB*) (*US S*) *init-B X-for-B Y-for-B H*

definition *run-pure-B-tc* ::
(*nat* ⇒ *'mem update*) ⇒ (*'x* ⇒ *'y*) ⇒ (*'x* ⇒ *bool*) ⇒ (*'mem*×*'l*) *tc-op* **where**
run-pure-B-tc UB H S = *run-pure-adv-tc d*(*Fst o UB*) (*US S*) *init-B X-for-B Y-for-B H*

lemma *run-pure-B-ell2-update*:
run-pure-B-update UB H S = *selfbutter* (*run-pure-B-ell2 UB H S*)
unfolding *run-pure-B-update-def run-pure-B-ell2-def* **by** (*rule run-pure-adv-ell2-update*)

lemma *run-pure-B-update-tc*:
run-pure-B-tc UB H S = *Abs-trace-class* (*run-pure-B-update UB H S*)
unfolding *run-pure-B-update-def run-pure-B-tc-def* **by** (*rule run-pure-adv-update-tc*)

lemma *run-pure-B-update-tc'*:
run-pure-B-update UB H S = *from-trace-class* (*run-pure-B-tc UB H S*)
by (*simp add*: *run-pure-B-tc-def run-pure-B-update-def run-pure-adv-update-tc'*)

lemma *run-pure-B-tc-pos*: *0* ≤ *run-pure-B-tc UB H S*
unfolding *run-pure-B-tc-def* **by** (*rule run-pure-adv-tc-pos*)

For mixed adversaries

definition *run-mixed-B* ::
'mem kraus-adv ⇒ (*'x* ⇒ *'y*) ⇒ (*'x* ⇒ *bool*) ⇒ (*'mem* × *'l*) *tc-op***where**
run-mixed-B kraus-B H S = *run-mixed-adv d* (λ*n*. *kf-Fst* (*kraus-B n*))
(*US S*) *init-B X-for-B Y-for-B H*

lemma *run-mixed-B-pos*:
  $0 \le$ *run-mixed-B kraus-B H S*
  unfolding *run-mixed-B-def* **by** (*rule run-mixed-adv-pos*)

lemma *norm-run-mixed-B*:
  assumes *F-norm-id*: $\bigwedge i.\ i < d+1 \Longrightarrow$ *kf-bound* (*F i*) $\le$ *id-cblinfun*
  shows *norm* (*run-mixed-B F H S*) ≤ 1
  unfolding *run-mixed-B-def* **proof** (*intro norm-run-mixed-adv*, *goal-cases*)
  case (*1 i*)
  have *kf-bound* (*kf-Fst* (*F i*)) $\le$ *kf-bound* (*F i*) $\otimes_o$ *id-cblinfun*
    using *kf-bound-kf-Fst* **by** *auto*
  also have ... $\le$ *id-cblinfun* $\otimes_o$ *id-cblinfun*
    by (*intro tensor-op-mono-left*[*OF assms*[*OF 1*]], *auto*)
  finally show *?case* **by** *auto*
qed (*auto simp add*: *register-XY-for-B norm-init-B norm-US assms*)

lemma *trace-preserving-norm-run-mixed-B*:
  assumes $\bigwedge i.\ i < d+1 \Longrightarrow$ *km-trace-preserving* (*kf-apply*
  (*kf-Fst* (*F i*)::(('*mem* × '*l*) *ell2*, ('*mem* × '*l*) *ell2*, *unit*) *kraus-family*))
  shows *norm* (*run-mixed-B F H S*) $= 1$
  unfolding *run-mixed-B-def*
  by (*auto intro*!: *trace-preserving-norm-run-mixed-adv*
    simp add: *assms isometry-US register-XY-for-B norm-init-B*
    simp del: *km-trace-preserving-apply*)

3 Definition of *B-count*

3.1 Defining the run of adversary *B*

Embeddings of X and Y in the counting register for B_{count}

definition *X-for-C* :: ‹'*x update* ⇒ ('*mem* × *nat*) *update*› **where**
  ‹*X-for-C* = *Fst o X*›

definition *Y-for-C* :: ‹'*y update* ⇒ ('*mem* × *nat*) *update*› **where**
  ‹*Y-for-C* = *Fst o Y*›

lemma [*register*]: ‹*register X-for-C*›
  by (*simp add*: *X-for-C-def*)

lemma [*register*]: ‹*register Y-for-C*›
  by (*simp add*: *Y-for-C-def*)

lemma *register-XY-for-C*:
  register (*X-for-C*;*Y-for-C*) **by** (*simp add*: *X-for-C-def Y-for-C-def*)

The oracle query on the extended register stays unitary.

lemma *unitary-H-C*: *unitary* ((*X-for-C*;*Y-for-C*) (*Uquery H*))

by (*intro register-unitary*[*OF register-XY-for-C*]) (*auto simp add*: *unitary-H*)

lemma *iso-H-C*: *isometry* ((*X-for-C*;*Y-for-C*) (*Uquery H*))
by (*simp add*: *unitary-H-C*)

Alternative representation of *Uquery H*.

lemma *UqueryH-tensor-id-cblinfunC*:
(*X-for-C*;*Y-for-C*) (*Uquery H*) = (*X*;*Y*) (*Uquery H*) $\otimes_o$ *id-cblinfun*
unfolding *X-for-C-def Y-for-C-def*
by (*metis Fst-def comp-eq-dest-lhs compat register-Fst register-comp-pair*)

The initial register for the adversary B with counting is the initial state and starting with 0 in the counting register.

definition *init-B-count* :: (*'mem* × *nat*) *ell2* **where**
‹*init-B-count* = *init* $\otimes_s$ *ket 0*›

lemma *norm-init-B-count*:
norm (*init-B-count*) = *1*
unfolding *init-B-count-def* **by** (*simp add*: *norm-init norm-tensor-ell2*)

Definition of adversary B with counting

For pure adversaries

definition *run-pure-B-count-ell2* **where**
run-pure-B-count-ell2 UB H S = *run-pure-adv d* (*Fst o UB*) (λ-. *U-S' S*) *init-B-count X-for-C Y-for-C H*

definition *run-pure-B-count-update* ::
(*nat* ⇒ *'mem update*) ⇒ (*'x* ⇒ *'y*) ⇒ (*'x* ⇒ *bool*) ⇒ (*'mem* × *nat*) *update* **where**
run-pure-B-count-update UB H S = *run-pure-adv-update d* (*Fst o UB*) (λ-. *U-S' S*) *init-B-count X-for-C Y-for-C H*

definition *run-pure-B-count-tc* ::
(*nat* ⇒ *'mem update*) ⇒ (*'x* ⇒ *'y*) ⇒ (*'x* ⇒ *bool*) ⇒ (*'mem*×*nat*) *tc-op* **where**
run-pure-B-count-tc UB H S = *run-pure-adv-tc d* (*Fst o UB*) (λ-. *U-S' S*) *init-B-count X-for-C Y-for-C H*

lemma *run-pure-B-count-ell2-update*:
run-pure-B-count-update UB H S = *selfbutter* (*run-pure-B-count-ell2 UB H S*)
unfolding *run-pure-B-count-update-def run-pure-B-count-ell2-def* **by** (*rule run-pure-adv-ell2-update*)

lemma *run-pure-B-count-update-tc*:
run-pure-B-count-tc UB H S = *Abs-trace-class* (*run-pure-B-count-update UB H S*)
unfolding *run-pure-B-count-update-def run-pure-B-count-tc-def* **by** (*rule run-pure-adv-update-tc*)

lemma *run-pure-B-count-update-tc'*:
run-pure-B-count-update UB H S = *from-trace-class* (*run-pure-B-count-tc UB H S*)

by (*simp add*: *run-pure-B-count-tc-def run-pure-B-count-update-def run-pure-adv-update-tc′*)

lemma *run-pure-B-count-tc-pos*: *0* ≤ *run-pure-B-count-tc UB H S*
unfolding *run-pure-B-count-tc-def* **by** (*rule run-pure-adv-tc-pos*)

For mixed adversaries

definition *run-mixed-B-count* ::
′mem kraus-adv ⇒ (*′x* ⇒ *′y*) ⇒ (*′x* ⇒ *bool*) ⇒ (*′mem* × *nat*) *tc-op* **where**
run-mixed-B-count kraus-B H S = *run-mixed-adv d* (*λn. kf-Fst* (*kraus-B n*))
(*λn. U-S′ S*) *init-B-count X-for-C Y-for-C H*

lemma *run-mixed-B-count-pos*:
0 ≤ *run-mixed-B-count kraus-B H S*
unfolding *run-mixed-B-count-def* **by** (*rule run-mixed-adv-pos*)

lemma *norm-run-mixed-B-count*:
assumes *F-norm-id*: ⋀*i. i* < *d+1* ⟹ *kf-bound* (*F i*) ≤ *id-cblinfun*
shows *norm* (*run-mixed-B-count F H S*) ≤ *1*
unfolding *run-mixed-B-count-def* **proof** (*intro norm-run-mixed-adv*, *goal-cases*)
case (*1 i*)
have *kf-bound* (*kf-Fst* (*F i*)) ≤ *kf-bound* (*F i*) $\otimes_o$ *id-cblinfun*
using *kf-bound-kf-Fst* **by** *auto*
also have ... ≤ *id-cblinfun* $\otimes_o$ *id-cblinfun*
by (*intro tensor-op-mono-left*[*OF assms*[*OF 1*]], *auto*)
finally show *?case* **by** *auto*
qed (*auto simp add*: *register-XY-for-C norm-init-B-count norm-U-S′ assms*)

lemma *trace-preserving-norm-run-mixed-B-count*:
assumes ⋀*i. i* < *d+1* ⟹ *km-trace-preserving* (*kf-apply*
(*kf-Fst* (*F i*)::((*′mem* × *nat*) *ell2*, (*′mem* × *nat*) *ell2*, *unit*) *kraus-family*))
shows *norm* (*run-mixed-B-count F H S*) = *1*
by (*auto intro*!: *trace-preserving-norm-run-mixed-adv*
simp add: *assms iso-U-S′ register-XY-for-C norm-init-B-count run-mixed-B-count-def*
simp del: *km-trace-preserving-apply*)

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*
unbundle *no register-syntax*

end
theory *Definition-Pure-O2H*

imports *Definition-O2H*
Run-Adversary

begin

unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
unbundle *register-syntax*

3.2 Locale for the pure O2H setting

For the pure state case, we define a separate locale for the pure one-way to hiding lemma.

locale *pure-o2h* = *o2h-setting* $TYPE('x)$ $TYPE('y{::}group\text{-}add)$ $TYPE('mem)$ $TYPE('l)$ +
— We fix the oracle function H, a subset of the oracle domain S and the sequence of operations the adversary A undertakes in the function UA.
fixes H :: ‹$'x \Rightarrow ('y{::}group\text{-}add)$›
and S :: ‹$'x \Rightarrow bool$›
and UA :: ‹$nat \Rightarrow 'mem\ update$›

— All operations by the adversary A must be isometries.
assumes *norm-UA*: ‹$\bigwedge i.\ i<d+1 \Longrightarrow norm\ (UA\ i) \leq 1$›
begin

Given the initial register state *init*, *run-A* returns the register state after performing the algorithm describing the adversary A.

definition ‹$run\text{-}A = run\text{-}pure\text{-}adv\ d\ UA\ (\lambda\text{-}.\ id\text{-}cblinfun{::}'mem\ update)\ init\ X\ Y\ H$›

lemma *norm-UA-Suc*: $n < d \Longrightarrow norm\ (UA\ (Suc\ n)) \leq 1$
by (*simp add*: *norm-UA*)

lemma *norm-UA-0-init*:
$norm\ (UA\ 0 *_V init) \leq 1$
using *norm-UA*[*of 0*] *norm-init d-gr-0*
by (*metis basic-trans-rules*(*23*) *mult-cancel-left2 norm-cblinfun semiring-norm*(*174*) *zero-less-Suc*)

lemma *tensor-proj-UA-tensor-commute*:
$(id\text{-}cblinfun \otimes_o proj\text{-}classical\text{-}set\ A)\ o_{CL}\ (UA\ (Suc\ n) \otimes_o id\text{-}cblinfun) =$
$(UA\ (Suc\ n) \otimes_o id\text{-}cblinfun)\ o_{CL}\ (id\text{-}cblinfun \otimes_o proj\text{-}classical\text{-}set\ A)$
by (*auto intro*!: *tensor-ell2-extensionality simp add*: *tensor-op-ell2*)

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*
unbundle *no register-syntax*

end
theory *Run-Pure-B*

imports *Definition-Pure-O2H*

begin

unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
unbundle *register-syntax*

context *pure-o2h*
begin

4 Defining and Representing the Adversary B

For the proof of the O2H, the final adversary B is restricted to information on the set S. That means, B takes note in a separate register of type $'l$ weather a value in S was queried and in which step with a unitary $U\text{-}S$.

Given the initial state $init \otimes_s ket\ 0 :: 'mem \times 'l$, we run the adversary with counting by performing consecutive bit-flips. *run-B-upto n* is the function that allows the adversary n calls to the query oracle. *run-B* allows exactly d query calls. The final state Ψ_{right} as in the paper is then *run-B*.

definition ‹*run-B-upto n* = *run-pure-adv n* (λi. *UA i* $\otimes_o$ *id-cblinfun*) (*US S*) *init-B X-for-B Y-for-B H*›

definition ‹*run-B* = *run-pure-adv d* (λi. *UA i* $\otimes_o$ *id-cblinfun*) (*US S*) *init-B X-for-B Y-for-B H*›

lemma *run-B-altdef*: *run-B* = *run-B-upto d*
  unfolding *run-B-def run-B-upto-def* **by** *auto*

lemma *run-B-upto-I*:
  run-B-upto (*Suc n*) = (*UA* (*Suc n*) $\otimes_o$ *id-cblinfun*) $*_V$ (*X-for-B*;*Y-for-B*) (*Uquery H*) $*_V$
    US S n $*_V$ *run-B-upto n*
  unfolding *run-B-upto-def* **by** *auto*

This version of the O2H is only for pure states. Therefore, the norm of states is always 1.

lemma *norm-run-B-upto*:
  assumes $n<d+1$
  shows *norm* (*run-B-upto n*) $\leq$ *1*
  using *assms* **proof** (*induct n*)
  case *0*
  then show *?case* **unfolding** *run-B-upto-def init-B-def* **using** *norm-UA-0-init*
    by (*auto simp add*: *tensor-op-ell2 norm-tensor-ell2 norm-init*)
next

case (*Suc n*)
have *norm* (*run-B-upto* (*Suc n*)) $\leq$ *norm* ((*UA* (*Suc n*) $\otimes_o$ (*id-cblinfun*::′*l update*))) $*$
norm ((*X-for-B*;*Y-for-B*) (*Uquery H*) $*_V$ *US S n* $*_V$
run-pure-adv n (λi. *UA i* $\otimes_o$ *id-cblinfun*) (*US S*) *init-B X-for-B Y-for-B H*)
unfolding *run-B-upto-def run-pure-adv.simps* **using** *norm-cblinfun* **by** *blast*
also have ... $\leq$ *norm* ((*UA* (*Suc n*) $\otimes_o$ (*id-cblinfun*::′*l update*))) $*$
norm (*run-pure-adv n* (λi. *UA i* $\otimes_o$ *id-cblinfun*) (*US S*) *init-B X-for-B Y-for-B H*)
by (*simp add*: *iso-H-B isometry-US isometry-preserves-norm norm-isometry*)
also have ... $\leq$ *1* **using** *norm-UA Suc* **by** (*simp add*: *mult-le-one run-B-upto-def tensor-op-norm*)
finally show *?case* **by** *linarith*
qed

lemma *norm-run-B*:
norm run-B $\leq$ *1*
unfolding *run-B-altdef* **using** *norm-run-B-upto* **by** *auto*

4.1 Representing the run of Adversary B as a finite sum

How the state after the n-th query behaves with respect to projections.

lemma *run-B-upto-proj-not-valid*:
assumes $n \leq d$
shows *Proj-ket-set* ($-$ *Collect blog*) $*_V$ *run-B-upto n* = *0*
using *le0*[*of n*] **proof** (*induct rule*: *dec-induct*)
case *base*
have *proj-classical-set* ($-$ *Collect blog*) $*_V$ *ket empty* = *0*
by (*intro proj-classical-set-not-elem*)(*auto simp add*: *blog-empty*)
then show *?case* **unfolding** *run-B-upto-def init-B-def Proj-ket-set-def*
by (*auto simp add*: *tensor-op-ell2*)
next
case (*step m*)
have *1*: *Proj-ket-set* ($-$ *Collect blog*) $*_V$ *run-B-upto* (*Suc m*) =
(*UA* (*Suc m*) $\otimes_o$ *id-cblinfun*) $*_V$ (*X-for-B*;*Y-for-B*) (*Uquery H*) $*_V$
Proj-ket-set ($-$ *Collect blog*) $*_V$
US S m $*_V$ *run-B-upto m* **unfolding** *Proj-ket-set-def*
by (*metis UqueryH-tensor-id-cblinfunB run-B-upto-I tensor-op-padding tensor-op-padding*′)
have $m<d$ **using** *step assms* **by** *auto*
have (*S-embed S* $\otimes_o$ (*proj-classical-set* ($-$ *Collect blog*) o_{CL} *Ub m*)) $*_V$ *run-B-upto m* = *0*
using *step*(*3*) **unfolding** *Proj-ket-set-def* **by** (*subst tensor-op-padding*,
subst proj-classical-set-not-blog-Ub[*OF* ‹$m<d$›])
(*metis* (*no-types*, *lifting*) *cblinfun.zero-right cblinfun-apply-cblinfun-compose*
cblinfun-compose-id-left comp-tensor-op)
moreover have (*not-S-embed S* $\otimes_o$ *proj-classical-set* ($-$ *Collect blog*)) $*_V$ *run-B-upto m* = *0*
using *step*(*3*) **unfolding** *Proj-ket-set-def* **by** (*subst tensor-op-padding*) *auto*
ultimately have *2*: *Proj-ket-set* ($-$ *Collect blog*) $*_V$ *US S m* $*_V$ *run-B-upto m* =
(*S-embed S* $\otimes_o$ *Ub m*) $*_V$ *Proj-ket-set* ($-$ *Collect blog*) $*_V$ *run-B-upto m* +
(*not-S-embed S* $\otimes_o$ *id-cblinfun*) $*_V$ *Proj-ket-set* ($-$ *Collect blog*) $*_V$ *run-B-upto m*
using *proj-classical-set-not-blog-Ub*[*symmetric*] *step assms* **unfolding** *US-def Proj-ket-set-def*

by (*auto simp add*: *cblinfun.add-left cblinfun.add-right comp-tensor-op*
cblinfun-apply-cblinfun-compose[*symmetric*] *simp del*: *cblinfun-apply-cblinfun-compose*)
show *?case* **by** (*subst 1*, *subst 2*) (*auto simp add*: *step*(*3*))
qed

lemma *orth-run-B-upto*:
fixes *C* :: *'mem update*
assumes *y*: $y \in$ *has-bits* {*Suc m..*<*d*} **and** *m*: *m*<*d*
shows *is-orthogonal* ((*C* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto m*) (*x* $\otimes_s$ *ket* (*list-to-l y*))
using *le0 y* **proof** (*induction m arbitrary*: *C y rule*: *Nat.dec-induct*)
case *base*
have *empty* $\neq$ *list-to-l y* **using** *base.prems has-bits-def has-bits-not-empty* **by** *fastforce*
then show *?case* **unfolding** *run-B-upto-def init-B-def* **by** (*auto simp add*: *tensor-op-ell2*)
next
case (*step n*)
define *A* **where** *A* = *C* o_{CL} *UA* (*Suc n*) o_{CL} (*X*;*Y*) (*Uquery H*) o_{CL} *S-embed S*
define *B* **where** *B* = *C* o_{CL} *UA* (*Suc n*) o_{CL} (*X*;*Y*) (*Uquery H*) o_{CL} *not-S-embed S*
then have *Suc*: (*C* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto* (*Suc n*) =
(*id-cblinfun* $\otimes_o$ *Ub n*) $*_V$ (*A* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto n* +
(*B* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto n*
apply (*subst tensor-op-padding'*[*symmetric*])
unfolding *run-B-upto-I UqueryH-tensor-id-cblinfunB US-def A-def B-def*
unfolding *cblinfun.add-left cblinfun.add-right*
by (*metis* (*mono-tags*, *lifting*) *cblinfun-apply-cblinfun-compose*
cblinfun-compose-id-left comp-tensor-op)
have *iso*: *isometry* (*id-cblinfun* $\otimes_o$ *Ub n*) **by** (*simp add*: *isometry-Ub*)
have *len-y*: *length y* = *d* **using** *step* **unfolding** *has-bits-def len-d-lists-def* **by** *auto*
have *blog-y*: *blog* (*list-to-l y*) **by** (*simp add*: *blog-list-to-l len-y*)
have *y-has-bits*: $y \in$ *has-bits* {*Suc n..*<*d*}
by (*metis Set.basic-monos*(*7*) *Suc-n-not-le-n has-bits-incl ivl-subset step*(*4*) *nat-le-linear*)
have *range*: *y*[*d*−*n*−*1*:=¬*y*!(*d*−*n*−*1*)] $\in$ *has-bits* {*Suc n..*<*d*}
by (*intro has-bits-not-elem*) (*use step y-has-bits m len-y len-d-lists-def* **in** ‹*auto*›)
have *no-flip*: ((*id-cblinfun* $\otimes_o$ *Ub n*) $*_V$ (*A* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto n*) $\cdot_C$
(*x* $\otimes_s$ *ket* (*list-to-l y*)) = *0* (**is** *?left* = *0*)
proof −
have *n*<*d* **using** *assms*(*2*) *step*(*2*) **by** *force*
have *?left* = ((*id-cblinfun* $\otimes_o$ *Ub n*) $*_V$ (*A* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto n*) $\cdot_C$
((*id-cblinfun* $\otimes_o$ *Ub n*) $*_V$ (*id-cblinfun* $\otimes_o$ *Ub n*) $*_V$ (*x* $\otimes_s$ *ket* (*list-to-l y*)))
by (*simp add*: *Ub-ket flip-flip*[*OF* ‹*n*<*d*› *blog-y*] *tensor-op-ell2*)
also have ... = ((*A* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto n*) $\cdot_C$
((*id-cblinfun* $\otimes_o$ *Ub n*) $*_V$ (*x* $\otimes_s$ *ket* (*list-to-l y*)))
using *isometry-cinner-both-sides*[*OF iso*] **by** *auto*
also have ... = ((*A* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto n*) $\cdot_C$
(*x* $\otimes_s$ *ket* (*flip n* (*list-to-l y*))) **by** (*simp add*: *Ub-ket tensor-op-ell2*)
also have ... = ((*A* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto n*) $\cdot_C$
(*x* $\otimes_s$ *ket* (*list-to-l* (*y*[*d*−*n*−*1*:=¬*y*!(*d*−*n*−*1*)])))
using ‹*n* < *d*› *flip-list-to-l le-eq-less-or-eq len-y* **by** *presburger*
also have ... = *0* **by** (*intro step*(*3*)) (*rule range*)

    finally show *?thesis* **by** *auto*
  qed
  show *?case* **using** *y-has-bits* **unfolding** *Suc cinner-add-left*
    by (*subst no-flip*, *subst step*(*3*))(*use step* **in** ‹*auto*›)
qed

lemma *orth-run-B-upto-ket*:
  assumes *y*: $y \in$ *has-bits* {*Suc m*..<*d*} **and** *Sucm*: *Suc m*<*d*
  shows *is-orthogonal* (*run-B-upto m*) ($x \otimes_s$ *ket* (*list-to-l y*))
proof −
  have *m*: *m*<*d* **using** *assms*(*2*) **by** *auto*
  have *id*: *run-B-upto m* = (*id-cblinfun* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto m* **by** *auto*
  then show *?thesis*
    by (*subst id*, *intro orth-run-B-upto*[*OF - m*]) (*use assms* **in** ‹*auto*›)
qed

lemma *orth-run-B-upto-flip*:
  assumes *y*: $y \in$ *has-bits* {*Suc m*..<*d*} **and** *Sucm*: *Suc m*<*d*
  shows *is-orthogonal* (*run-B-upto m*) ($x \otimes_s$ *ket* (*flip m* (*list-to-l y*)))
proof −
  have *len-d*: $y \in$ *len-d-lists* **using** *y* **unfolding** *has-bits-def* **by** *auto*
  then have *m*: *m* < *length y* **by** (*simp add*: *Suc-lessD Sucm len-d-lists-def*)
  have *yd*: *length y* $\leq$ *d* **using** *y has-bits-def len-d-lists-def* **by** *auto*
  have *id*: *run-B-upto m* = (*id-cblinfun* $\otimes_o$ *id-cblinfun*) $*_V$ *run-B-upto m* **by** *auto*
  have *range*: *y*[*d* − *m* − *1* := ¬ *y* ! (*d* − *m* − *1*)] $\in$ *has-bits* {*Suc m*..<*d*}
    by (*intro has-bits-not-elem*) (*use y Sucm len-d* **in** ‹*auto*›)
  then show *?thesis*
    by (*subst flip-list-to-l*[*OF m yd*], *subst id*, *intro orth-run-B-upto*)
      (*use len-d len-d-lists-def Sucm* **in** ‹*auto*›)
qed

lemma *run-B-upto-proj-over*:
  assumes *n*≤*d*
  shows *Proj-ket-set* (*list-to-l* ‘ *has-bits* {*n*..<*d*}) $*_V$ *run-B-upto n* = *0*
proof (*cases n*=*d*)
  case *True* **then show** *?thesis* **unfolding** ‹*n*=*d*› *Proj-ket-set-def proj-classical-set-def* **by** *auto*
next
  case *False*
  then have *n*<*d* **using** *assms* **by** *auto*
  show *?thesis*
    using *le0*[*of n*] **proof** (*induct rule*: *dec-induct*)
    case *base*
    have *empty* $\notin$ *list-to-l* ‘ *has-bits* {*0*..<*d*} **using** *has-bits-def has-bits-not-empty* **by** *fastforce*
    then have *proj-classical-set* (*list-to-l* ‘ *has-bits* {*0*..<*d*}) $*_V$ *ket* (*empty*) = *0*
      by (*intro proj-classical-set-not-elem*) *auto*
    then show *?case* **unfolding** *run-B-upto-def init-B-def Proj-ket-set-def*
      by (*auto simp add*: *tensor-op-ell2*)
  next

case (*step m*)
then have *m<d* **using** *assms* **by** *auto*
then have *Suc m≤d* **by** *auto*
have *Suc m < d* **using** ‹*n < d*› *step*(*2*) **by** *linarith*
have *1*: *Proj-ket-set* (*list-to-l* ‘ *has-bits* {*Suc m..<d*}) $*_V$ *run-B-upto* (*Suc m*) =
(*UA* (*Suc m*) $\otimes_o$ *id-cblinfun*) $*_V$ (*X-for-B*;*Y-for-B*) (*Uquery H*) $*_V$
Proj-ket-set (*list-to-l* ‘ *has-bits* {*Suc m..<d*}) $*_V$
US S m $*_V$ *run-B-upto m* **unfolding** *Proj-ket-set-def*
by (*smt* (*verit*, *del-insts*) *UqueryH-tensor-id-cblinfunB cblinfun-apply-cblinfun-compose*
cblinfun-compose-id-left cblinfun-compose-id-right comp-tensor-op run-B-upto-def
run-pure-adv.simps(*2*))
have *2*: *Proj-ket-set* (*list-to-l* ‘ *has-bits* {*Suc m..<d*}) $*_V$ *US S m* $*_V$ *run-B-upto m* =
(*S-embed S* $\otimes_o$ *Ub m*) $*_V$ *Proj-ket-set* (*flip m* ‘ *list-to-l* ‘ *has-bits* {*Suc m..<d*}) $*_V$ *run-B-upto*
m +
(*not-S-embed S* $\otimes_o$ *id-cblinfun*) $*_V$ *Proj-ket-set* (*list-to-l* ‘ *has-bits* {*Suc m..<d*}) $*_V$
run-B-upto m
using *proj-classical-set-over-Ub*[*OF* ‹*Suc m≤d*›, *symmetric*] *step assms*
unfolding *US-def Proj-ket-set-def*
by (*auto simp add*: *cblinfun.add-left cblinfun.add-right comp-tensor-op*
cblinfun-apply-cblinfun-compose[*symmetric*] *simp del*: *cblinfun-apply-cblinfun-compose*)
have *Proj-ket-set* (*flip m* ‘ *list-to-l* ‘ *has-bits* {*Suc m..<d*}) $*_V$ *run-B-upto m* = *0*
proof −
have *Proj-ket-set* (*flip m* ‘ *list-to-l* ‘ *has-bits* {*Suc m..<d*}) $*_V$ *run-B-upto m* =
Proj (⊤ $\otimes_S$ *ccspan* (*ket* ‘ *flip m* ‘ *list-to-l* ‘ *has-bits* {*Suc m..<d*})) $*_V$ *run-B-upto m*
by (*simp add*: *Proj-on-own-range is-Proj-tensor-op proj-classical-set-def*
tensor-ccsubspace-via-Proj Proj-ket-set-def)
also have ... = *0* **by** (*intro Proj-0-compl*, *unfold ccspan-UNIV*[*symmetric*],
subst tensor-ccsubspace-ccspan,*intro mem-ortho-ccspanI*)
(*auto intro*!: *orth-run-B-upto-flip simp add*: ‹*Suc m<d*›)
finally show *?thesis* **by** *auto*
qed
then have *3*: (*S-embed S* $\otimes_o$ *Ub m*) $*_V$ *Proj-ket-set* (*flip m* ‘ *list-to-l* ‘ *has-bits* {*Suc m..<d*})
$*_V$ *run-B-upto m* = *0*
by (*metis* (*no-types*, *lifting*) *cblinfun.real.zero-right cblinfun-apply-cblinfun-compose*)
have *Proj-ket-set* (*list-to-l* ‘ *has-bits* {*Suc m..<d*}) $*_V$ *run-B-upto m* = *0*
proof −
have *Proj-ket-set* (*list-to-l* ‘ *has-bits* {*Suc m..<d*}) $*_V$ *run-B-upto m* =
Proj (⊤ $\otimes_S$ *ccspan* (*ket* ‘ *list-to-l* ‘ *has-bits* {*Suc m..<d*})) $*_V$ *run-B-upto m*
by (*simp add*: *Proj-on-own-range is-Proj-tensor-op proj-classical-set-def*
tensor-ccsubspace-via-Proj Proj-ket-set-def)
also have ... = *0* **by** (*intro Proj-0-compl*, *unfold ccspan-UNIV*[*symmetric*],
subst tensor-ccsubspace-ccspan,*intro mem-ortho-ccspanI*)
(*auto intro*!: *orth-run-B-upto-ket simp add*: ‹*Suc m<d*›)
finally show *?thesis* **by** *auto*
qed
then have *4*: (*not-S-embed S* $\otimes_o$ *id-cblinfun*) $*_V$ *Proj-ket-set* (*list-to-l* ‘ *has-bits* {*Suc m..<d*})
$*_V$
run-B-upto m = *0* **by** (*subst tensor-op-padding*) *auto*
show *?case* **by** (*subst 1*, *subst 2*, *subst 3*, *subst 4*) *auto*

qed
qed

How Ψs relate to *run-B*. We can write *run-B* as a sum counting over all valid ket states in the counting register.

lemma *not-empty-list-nth*:
assumes $x{\in}len\text{-}d\text{-}lists$
shows $x \neq empty\text{-}list \longleftrightarrow (\exists\, i{<}d.\ x!i)$
using *assms* **unfolding** *len-d-lists-def empty-list-def* **by** (*simp add*: *list-eq-iff-nth-eq*)

First we show the mere existence of such a form.

lemma *run-B-upto-sum*:
assumes $n{<}d$
shows $\exists\, v.\ run\text{-}B\text{-}upto\ n = (\sum i{\in}has\text{-}bits\text{-}upto\ n.\ v\ i \otimes_s ket\ (list\text{-}to\text{-}l\ i))$
using *le0*[*of n*] *assms* **proof** (*induct n rule*: *Nat.dec-induct*)
case *base*
have $\exists\, xa{\in}\{0..{<}d\}.\ x\ !\ (d - Suc\ xa)$
if $x \neq empty\text{-}list\ x \in len\text{-}d\text{-}lists$ **for** x :: *bool list*
using *not-empty-list-nth*[*OF that*(*2*)] *that*(*1*) **by** (*metis Suc-pred atLeast0LessThan diff-diff-cancel diff-less-Suc lessThan-iff less-or-eq-imp-le not-less-eq zero-less-diff*)
then have *rew*: $has\text{-}bits\text{-}upto\ 0 = \{empty\text{-}list\}$
unfolding *has-bits-upto-def has-bits-def len-d-lists-def empty-list-def* **by** *auto*
show *?case* **by** (*subst rew*, *unfold run-B-upto-def init-B-def*)
(*auto simp add*: *empty-list-to-l tensor-op-ell2*)
next
case (*step n*)
let $?upto\text{-}n = has\text{-}bits\text{-}upto\ n$
let $?upto\text{-}Suc\text{-}n = has\text{-}bits\text{-}upto\ (Suc\ n)$
let $?only\text{-}n = has\text{-}bits\ \{n\} - has\text{-}bits\{Suc\ n..{<}d\}$
have [*simp*]: $n{<}d$ **by**(*rule Suc-lessD*[*OF step*(*4*)])
from *step* **obtain** *v* **where** *v*: $run\text{-}B\text{-}upto\ n =$
$(\sum i{\in}?upto\text{-}n.\ v\ i \otimes_s ket\ (list\text{-}to\text{-}l\ i))$
using *Suc-lessD* **by** *presburger*
define *v1* **where** $v1\ i = not\text{-}S\text{-}embed\ S *_V (v\ i)$ **for** *i*
define *v2* **where** $v2\ i = S\text{-}embed\ S *_V (v\ i)$ **for** *i*
have *US-ket*: $US\ S\ n *_V (v\ i) \otimes_s ket\ (list\text{-}to\text{-}l\ i) =$
$v1\ i \otimes_s ket\ (list\text{-}to\text{-}l\ i) + v2\ i \otimes_s ket\ (flip\ n\ (list\text{-}to\text{-}l\ i))$
if $i{\in}?upto\text{-}n$ **for** *i* **unfolding** *US-ket-only01*
by (*auto simp add*: *that v1-def v2-def*)
define v' **where** $v'\ i = (if\ i \in\ ?upto\text{-}n$
$then\ ((UA\ (Suc\ n)) *_V (X;Y)\ (Uquery\ H) *_V v1\ i)$
$else\ ((UA\ (Suc\ n)) *_V (X;Y)\ (Uquery\ H) *_V v2\ (i[d{-}n{-}1 := \neg i!(d{-}n{-}1)])))$ **for** *i*
have $(\sum i{\in}?upto\text{-}n.$
$((UA\ (Suc\ n)) *_V (X;Y)\ (Uquery\ H) *_V v1\ i) \otimes_s ket\ (list\text{-}to\text{-}l\ i) +$
$((UA\ (Suc\ n)) *_V (X;Y)\ (Uquery\ H) *_V v2\ i) \otimes_s ket\ (flip\ n\ (list\text{-}to\text{-}l\ i))) =$
$(\sum i{\in}?upto\text{-}Suc\text{-}n.\ v'\ i \otimes_s ket\ (list\text{-}to\text{-}l\ i))$ (**is** $?l = ?r$)
proof −
have *left*: $?l = (\sum i{\in}?upto\text{-}n.\ ((UA\ (Suc\ n)) *_V (X;Y)\ (Uquery\ H) *_V v1\ i) \otimes_s ket\ (list\text{-}to\text{-}l\ i)) +$

($\sum$ *i*∈*?upto-n*. (*UA* (*Suc n*) $*_V$ (*X*;*Y*) (*Uquery H*) $*_V$ *v2 i*) $\otimes_s$ *ket* (*flip n* (*list-to-l i*)))
(**is** *?l* = *?fst* + *?snd*)
by (*subst sum.distrib*)(*auto intro*!: *sum.cong*)
have *fst*: *?fst* = ($\sum$ *i*∈*?upto-n*. *v′ i* $\otimes_s$ *ket* (*list-to-l i*)) **unfolding** *v′-def* **by** *auto*

let *?reindex* = (λ*k*. *k*[*d*−*n*−*1*:= ¬*k*!(*d*−*n*−*1*)])
have *reindex-idem*: *?reindex* (*?reindex l*) = *l* **if** *l*∈*len-d-lists* **for** *l*
by (*smt* (*verit*, *best*) *list-update-id list-update-overwrite*)
have *snd′*: *?snd* = ($\sum$ *i*∈*?reindex*‘ *?upto-n*.
((*UA* (*Suc n*)) $*_V$ (*X*;*Y*) (*Uquery H*) $*_V$ *v2* (*?reindex i*)) $\otimes_s$
ket (*list-to-l i*))
proof (*subst sum.reindex*, *goal-cases*)
case *1*
then show *?case* **unfolding** *has-bits-upto-def*
by (*metis* (*no-types*, *lifting*) *inj-on-def inj-on-diff reindex-idem*)
next
case *2*
then show *?case*
proof (*intro sum.cong*, *goal-cases*)
case (*2 x*)
have *one*: *n* < *length x* **using** ‹*n*<*d*› *2 has-bits-upto-def len-d-lists-def* **by** *auto*
have *two*: *length x* ≤ *d* **using** *2 len-d-lists-def has-bits-upto-def* **by** *auto*
have *three*: *x* ∈ *len-d-lists* **using** *2 has-bits-upto-def* **by** *auto*
show *?case* **by** (*subst flip-list-to-l*[*OF one two*])
(*use reindex-idem*[*OF three*] *three* **in** ‹*auto simp add*: *len-d-lists-def*›)
qed *simp*
qed
have *set-rew*: *?reindex* ‘ *?upto-n* = *?only-n*
proof −
have *x*∈*?only-n* **if** *x*∈*?reindex* ‘ *?upto-n* **for** *x*
proof −
have *len-d-x*: *x* ∈ *len-d-lists* **using** *that* **unfolding** *len-d-lists-def has-bits-upto-def* **by** *auto*
obtain *x′* **where** *x′*:*x* = *?reindex x′* *x′*∈*?upto-n* **using** ‹*x* ∈ *?reindex* ‘ *?upto-n*› **by** *blast*
then have *len-x′*: *length x′* = *d* **unfolding** *len-d-lists-def has-bits-upto-def* **by** *auto*
have ¬ *x′*!(*d*−*n*−*1*) **by** (*intro has-bits-upto-elem* [*OF x′*(*2*)]) *auto*
then have *x*!(*d*−*n*−*1*) **unfolding** *x′* **by** (*subst nth-list-update-eq*)(*auto simp add*: *d-gr-0 len-x′*)
then have *a*: *x*∈*has-bits* {*n*} **unfolding** *has-bits-def* **using** *len-d-x* **by** *auto*
have *x′* ∈ *has-bits-upto* (*Suc n*) **using** *has-bits-split-Suc has-bits-upto-def x′*(*2*) **by** *force*
then have ¬ *x′*!(*d*−*i*−*1*) **if** *i*∈{*Suc n*..<*d*} **for** *i*
by (*intro has-bits-elem*[*of x′* {*Suc n*..<*d*}], *unfold has-bits-upto-def*[*symmetric*])
(*use that* **in** ‹*auto*›)
then have ∀ *i*∈{*Suc n*..<*d*}. ¬ *x*!(*d*−*i*−*1*) **using** *x′*(*1*) **by** *fastforce*
then have *b*: *x*∉*has-bits* {*Suc n*..<*d*} **by** (*simp add*: *has-bits-def*)
show *?thesis* **using** *a b* **by** *auto*
qed
moreover have*x*∈*?reindex* ‘ *?upto-n* **if** *x*∈*?only-n* **for** *x*
proof −

```
      have len-d-x: x ∈ len-d-lists using that unfolding has-bits-def len-d-lists-def by auto
      define x' where x':x' = ?reindex x
      then have x'∈?reindex ` ?only-n using ‹x ∈ ?only-n› reindex-idem by auto
      then have len-d-x': x' ∈ len-d-lists using that
        unfolding has-bits-def len-d-lists-def by auto
      have ¬x'!(d−i−1) if i∈{n..<d} for i
      proof (cases i = n)
        case True
        have ineq: d − n − 1 < length x using len-d-x len-d-lists-def by (simp add: d-gr-0)
        have x∈has-bits {n} using ‹x ∈ ?only-n› by blast
        then have x ! (d−n−1) unfolding has-bits-def by auto
        then show ?thesis unfolding True x' by (subst nth-list-update-eq) (use ineq in ‹auto›)
      next
        case False
        have set: i∈{Suc n..<d} using False ‹i∈{n..<d}› by auto
        have x∉has-bits {Suc n..<d} using ‹x ∈ ?only-n› by auto
        then have ¬x!(d−i−1) using has-bits-def set using len-d-x by blast
        moreover have x!(d−i−1) = x'!(d−i−1) using False ‹i∈{n..<d}› using x' by force
        ultimately show ?thesis by auto
      qed
      then have x'∉ has-bits {n..<d} by (simp add: has-bits-def)
      then have x'∈?upto-n using len-d-x' has-bits-upto-def by auto
      then show ?thesis using x'
        by (metis (no-types, lifting) image-iff len-d-x reindex-idem)
    qed
    ultimately show ?thesis by auto
  qed
  have snd: ?snd = (∑ i∈?only-n. v' i ⊗s ket (list-to-l i))
    unfolding v'-def snd' using set-rew
    by (auto intro!: sum.cong simp add: has-bits-upto-def has-bits-split-Suc)
  have incl: ?only-n ⊆ has-bits {n..<d}
    using has-bits-incl[of {n} {n..<d}] by (auto)
  have union: ?upto-n ∪ ?only-n = ?upto-Suc-n
    unfolding has-bits-split-Suc[OF ‹n<d›] has-bits-upto-def
    using has-bits-in-len-d-lists[of {n}] by blast
  show ?thesis unfolding left fst snd
    by (subst sum.union-disjoint[symmetric])(use incl union has-bits-upto-def in ‹auto›)
 qed
 then show ?case unfolding run-B-upto-def unfolding run-pure-adv.simps
  by (fold run-B-upto-def, subst v)
    (auto simp add: cblinfun.sum-right tensor-op-ell2 US-ket-only01
      UqueryH-tensor-id-cblinfunB cblinfun.add-right US-ket nth-append v1-def v2-def)
qed
```

As a shorthand, we define *Proj-ket-upto*.

```
lemma run-B-projection:
 assumes n<d
 shows Proj-ket-upto (has-bits-upto n) *V run-B-upto n = run-B-upto n
proof −
```

obtain *v* **where** *v*: *run-B-upto n* = ($\sum$ *i*∈*has-bits-upto n*. *v i* $\otimes_s$ *ket* (*list-to-l i*))
  using *run-B-upto-sum assms* **by** *auto*
  show *?thesis* **unfolding** *v* **by** (*subst cblinfun.sum-right*, *intro sum.cong*, *simp*,
    intro Proj-ket-upto-vec) *fastforce*
qed

How Ψ*s* relate to *run-B*.

lemma *run-B-upto-split*:
  assumes *n*≤*d*
  shows *run-B-upto n* = ($\sum$ *i*∈*has-bits-upto n*. Ψ*s* (*list-to-l i*) (*run-B-upto n*) $\otimes_s$ *ket* (*list-to-l i*))
proof −
  have *neg-set*: − *list-to-l* ‘ (*has-bits-upto n*) =
    list-to-l ‘ *has-bits* {*n*..<*d*} ∪ − *Collect blog* **unfolding** *has-bits-upto-def*
    by (*smt* (*verit*, *del-insts*) *Compl-Diff-eq Diff-subset Un-commute inj-list-to-l*
      inj-on-image-set-diff has-bits-in-len-d-lists surj-list-to-l)
  have *run-B-upto n* = *id-cblinfun* $*_V$ *run-B-upto n* **by** *auto*
  also have ... = ($\sum$ *i*∈*list-to-l* ‘ *has-bits-upto n*.
    (*tensor-ell2-right* (*ket i*)) o_{CL} (*tensor-ell2-right* (*ket i*)*)) $*_V$ *run-B-upto n* +
    Proj-ket-set (− *list-to-l* ‘ *has-bits-upto n*) $*_V$ *run-B-upto n*
    by (*subst id-cblinfun-tensor-split-finite*[*of list-to-l* ‘ *has-bits-upto n*])
      (*auto simp add*: *cblinfun.add-left*)
  also have ... = ($\sum$ *i*∈*list-to-l* ‘ *has-bits-upto n*.
    (*tensor-ell2-right* (*ket i*)) o_{CL} (*tensor-ell2-right* (*ket i*)*)) $*_V$ *run-B-upto n* +
    Proj-ket-set (*list-to-l* ‘ *has-bits* {*n*..<*d*}) $*_V$ *run-B-upto n*
  proof −
    have *orth*: *is-orthogonal x y*
      if *ass*: *x* ∈ *ket* ‘ *list-to-l* ‘ *has-bits* {*n*..<*d*} *y* ∈ *ket* ‘ (− *Collect blog*) **for** *x y*
    proof −
      obtain *j* **where** *j*: *j*∈*has-bits* {*n*..<*d*} **and** *x*: *x* = *ket* (*list-to-l j*) **using** *ass*(*1*) **by** *auto*
      then have *valid*: *blog* (*list-to-l j*)
        by (*metis Set.basic-monos*(*7*) *has-bits-in-len-d-lists imageI mem-Collect-eq surj-list-to-l*)
      obtain *k* **where** *k*: ¬ *blog k* **and** *y*: *y* = *ket k* **using** *ass*(*2*) **by** *auto*
      show *is-orthogonal x y* **unfolding** *x y* **by** (*subst orthogonal-ket*)(*use k valid* **in** ‹*blast*›)
    qed
    then show *?thesis* **using** *run-B-upto-proj-not-valid* **unfolding** *neg-set Proj-ket-set-def*
      by (*subst proj-classical-set-union*[*OF orth*])
        (*auto simp add*: *tensor-op-right-add cblinfun.add-left assms*)
  qed
  also have ... = ($\sum$ *i*∈*has-bits-upto n*. (*tensor-ell2-right* (*ket* (*list-to-l i*))) o_{CL}
    (*tensor-ell2-right* (*ket* (*list-to-l i*))*)) $*_V$ *run-B-upto n*
    unfolding *run-B-upto-proj-over*[*OF* ‹*n*≤*d*›] **proof** (*subst sum.reindex*, *goal-cases*)
    case *1* **show** *?case* **using** *bij-betw-imp-inj-on*[*OF bij-betw-list-to-l*]
      by (*simp add*: *has-bits-upto-def inj-on-diff*)
  qed (*auto simp add*: *Proj-ket-set-def*)
  finally have ∗: *run-B-upto n* =
    ($\sum$ *i*∈*has-bits-upto n*. (*tensor-ell2-right* (*ket* (*list-to-l i*))* $*_V$ *run-B-upto n*) $\otimes_s$ *ket* (*list-to-l i*))
    by (*smt* (*verit*, *ccfv-SIG*) *cblinfun.sum-left cblinfun-apply-cblinfun-compose sum.cong*

tensor-ell2-right.rep-eq)
show *?thesis* **unfolding** Ψ*s-def* **by** (*subst* ∗)(*use lessThan-Suc-atMost* **in** ‹*auto*›)
qed

lemma *run-B-split*:
run-B = ($\sum$ *i*∈*len-d-lists.* Ψ*s* (*list-to-l i*) *run-B* $\otimes_s$ (*ket* (*list-to-l i*)))
unfolding *run-B-altdef has-bits-upto-d*[*symmetric*] **by** (*subst run-B-upto-split*[*symmetric*]) *auto*

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*
unbundle *no register-syntax*

end
theory *Run-Pure-B-count*

imports *Definition-Pure-O2H*

begin

unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
unbundle *register-syntax*

context *pure-o2h*
begin

5 Defining and Representing the Adversary B with Counting

For the proof of the O2H, we need an intermediate operator *U-S′*. The operator *U-S′* counts, how many oracle queries were made so far in a separate register (modelled by *nat*).

Given the initial state *init* $\otimes_s$ *ket 0* :: ′*mem* × *nat*, we run the adversary with counting by adding +1 in {*0..<d+1*}. *run-B-count-upto n* is the function that allows the adversary n calls to the query oracle. *run-B-count* allows exactly d query calls (ie. queries up to the full query depth d). The final state called Ψ_{count} in the paper is represented by *run-B-count*.

definition ‹*run-B-count-upto n* =
run-pure-adv n (λ*i. UA i* $\otimes_o$ *id-cblinfun*) (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*›

definition ‹*run-B-count* = *run-pure-adv d* (λi. *UA i* $\otimes_o$ *id-cblinfun*) (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*›

lemma *run-B-count-altdef*: *run-B-count* = *run-B-count-upto d*
  unfolding *run-B-count-def run-B-count-upto-def* **by** *auto*

lemma *run-B-count-upto-I*:
  run-B-count-upto (*Suc n*) = (*UA* (*Suc n*) $\otimes_o$ *id-cblinfun*) $*_V$ (*X-for-C*;*Y-for-C*) (*Uquery H*) $*_V$
    U-S′ S $*_V$ *run-B-count-upto n*
  unfolding *run-B-count-upto-def* **by** *auto*

This version of the O2H is only for pure states. Therefore, the norm of states is always 1.

lemma *norm-run-B-count-upto*:
  assumes *n*<*d*+*1*
  shows *norm* (*run-B-count-upto n*) ≤ *1*
  using *assms* **proof** (*induct n*)
  case *0*
  then show *?case* **unfolding** *run-B-count-upto-def init-B-count-def* **using** *norm-UA-0-init*
    by (*auto simp add*: *tensor-op-ell2 norm-tensor-ell2 norm-init*)
next
  case (*Suc n*)
  have *norm* (*run-B-count-upto* (*Suc n*)) ≤ *norm* ((*UA* (*Suc n*) $\otimes_o$ (*id-cblinfun*::*nat update*))) *
    norm ((*X-for-C*;*Y-for-C*) (*Uquery H*) $*_V$ *U-S′ S* $*_V$
    run-pure-adv n (λi. *UA i* $\otimes_o$ *id-cblinfun*) (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*)
    unfolding *run-B-count-upto-def run-pure-adv.simps* **using** *norm-cblinfun* **by** *blast*
  also have ... ≤ *norm* ((*UA* (*Suc n*) $\otimes_o$ (*id-cblinfun*::*nat update*))) *
    norm (*run-pure-adv n* (λi. *UA i* $\otimes_o$ *id-cblinfun*) (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*)
    by (*simp add*: *iso-H-C iso-U-S′ isometry-preserves-norm norm-isometry*)
  also have ... ≤ *1* **using** *norm-UA Suc* **by** (*simp add*: *mult-le-one run-B-count-upto-def tensor-op-norm*)
  finally show *?case* **by** *linarith*
qed

lemma *norm-run-B-count*:
  norm run-B-count ≤ *1*
  unfolding *run-B-count-altdef* **using** *norm-run-B-count-upto* **by** *auto*

5.1 Representing the run of Adversary B with counting as a finite sum

Preparation for representation of *run-B-count*

lemma *tensor-proj-UqueryH-commute*:
  (*id-cblinfun* $\otimes_o$ *proj-classical-set A*) o_{CL} (*X-for-C*;*Y-for-C*) (*Uquery H*) =
    (*X-for-C*;*Y-for-C*) (*Uquery H*) o_{CL} (*id-cblinfun* $\otimes_o$ *proj-classical-set A*)

by (*subst UqueryH-tensor-id-cblinfunC*)+ (*auto intro*!: *tensor-ell2-extensionality simp add*: *tensor-op-ell2*)

How the counting unitary *Uc* behaves with respect to projections on the counting register.

lemma *proj-Uc*:
  assumes *m>0*
  shows *proj-classical-set* {*m*} o_{CL} *Uc* = *Uc* o_{CL}
  (*if m<d+1 then proj-classical-set* {*m−1*} *else proj-classical-set* {*m*})
proof (*intro equal-ket*, *goal-cases*)
  case (*1 x*)
  consider (*less*) *x<d* | (*eq*) *x=d* | (*greater*) *x>d* **by** *linarith*
  then show *?case*
  proof (*cases*)
    case *less*
    have *proj-classical-set* {*m*} $*_V$ *ket* (*Suc x*) = *ket* (*Suc x*) **if** *m=x+1* **using** *that*
        proj-classical-set-elem **by** *force*
    moreover have *proj-classical-set* {*m*} $*_V$ *ket* (*Suc x*) = *0* **if** *m≠x+1* **using** *that*
      by (*simp add*: *proj-classical-set-not-elem*)
    moreover have (*Uc* o_{CL}(*if m < d + 1 then proj-classical-set* {*m − 1*} *else proj-classical-set* {*m*}))
        $*_V$ *ket x* = *ket* (*Suc x*) **if** *m=x+1*
      by (*simp add*: *Uc-ket-less less proj-classical-set-elem that*)
    moreover have (*Uc* o_{CL}(*if m < d + 1 then proj-classical-set* {*m − 1*} *else proj-classical-set* {*m*}))
        $*_V$ *ket x* = *0* **if** *m≠x+1*
      using *assms less proj-classical-set-not-elem that*
      by (*smt* (*verit*) *Suc-eq-plus1 Suc-pred′ basic-trans-rules*(*20*) *cblinfun.real.zero-right*
          cblinfun-apply-cblinfun-compose not-less-eq singletonD)
    ultimately show *?thesis* **using** *Uc-ket-less*[*OF less*] *less* **by** *force*
  next
    case *eq*
    then show *?thesis* **using** *Uc-ket-d assms proj-classical-set-not-elem*
     by (*smt* (*verit*, *best*) *One-nat-def Set.ball-empty Suc-pred ab-semigroup-add-class.add-ac*(*1*)

        cblinfun.real.zero-right insertE less-add-eq-less less-add-same-cancel2 less-numeral-extra(*1*)
          nat-less-le plus-1-eq-Suc simp-a-oCL-b′)
  next
    case *greater*
    show *?thesis* **using** *Uc-ket-greater*[*OF greater*] *greater proj-classical-set-elem*
      by (*smt* (*verit*, *ccfv-SIG*) *Suc-eq-plus1 basic-trans-rules*(*20*) *cblinfun.real.zero-right*
          less-diff-conv not-less-eq proj-classical-set-not-elem simp-a-oCL-b′ singleton-iff)
  qed
qed

lemma *proj-classical-set-over-Uc*:
  proj-classical-set {*Suc n..*} o_{CL} *Uc* = *Uc* o_{CL} *proj-classical-set*
(*if n>d then* {*Suc n..*} *else* {*n..*}−{*d*})
proof (*intro equal-ket*, *goal-cases*)

case (*1 x*)
consider (*less*) *x<d* | (*eq*) *x=d* | (*greater*) *x>d* **by** *linarith*
then show *?case*
proof (*cases*)
case *less*
have *proj-classical-set* {*Suc n..*} $*_V$ *ket* (*Suc x*) = *0* **if** *x<n*
by (*simp add*: *proj-classical-set-upto that*)
moreover have *Uc* $*_V$ *proj-classical-set* ({*n..*} − {*d*}) $*_V$ *ket x* = *0* **if** *x<n*
by (*subst proj-classical-set-not-elem*) (*use that* **in** ‹*auto*›)
moreover have *proj-classical-set* {*Suc n..*} $*_V$ *ket* (*Suc x*) = *ket* (*Suc x*) **if** *x≥n*
by (*simp add*: *Proj-fixes-image ccspan-superset′ proj-classical-set-def that*)
moreover have *Uc* $*_V$ *proj-classical-set* ({*n..*} − {*d*}) $*_V$ *ket x* = *ket* (*Suc x*) **if** *x≥n*
by (*subst proj-classical-set-elem*) (*use that less Uc-ket-less*[*OF less*] **in** ‹*auto*›)
ultimately show *?thesis* **using** *Uc-ket-less*[*OF less*] *less proj-classical-set-upto* **by** *force*
next
case *eq*
have (*proj-classical-set* {*Suc n..*} o_{CL} *Uc*) $*_V$ *ket x* = *0*
using *Uc-ket-d proj-classical-set-not-elem eq* **by** (*simp add*: *proj-classical-set-upto*)
moreover have
(*Uc* o_{CL} *proj-classical-set* (*if d < n then* {*Suc n..*} *else* {*n..*} − {*d*})) $*_V$ *ket x* = *0*
using *proj-classical-set-not-elem eq*
by (*metis* (*full-types*) *Diff-not-in cblinfun.real.zero-right cblinfun-apply-cblinfun-compose less-SucI proj-classical-set-upto*)
ultimately show *?thesis* **by** *force*
next
case *greater*
have *proj-classical-set* {*Suc n..*} $*_V$ *ket x* = *Uc* $*_V$ *proj-classical-set* {*Suc n..*} $*_V$ *ket x*
if *d<n* **by** (*cases x<n+1*)(*auto simp add*: *proj-classical-set-not-elem Uc-ket-greater greater proj-classical-set-elem*)
then show *?thesis* **using** *Uc-ket-greater*[*OF greater*] *greater proj-classical-set-elem*
by (*smt* (*verit*, *ccfv-SIG*) *atLeast-iff basic-trans-rules*(*19*) *cblinfun-apply-cblinfun-compose diff-Suc-1 insertCI insertE insert-Diff-single le-simps*(*2*) *lessI less-natE linorder-neqE-nat*

linorder-not-less zero-less-Suc)
qed
qed

How the state after the *n*-th query behaves with respect to projections.

lemma *run-B-count-proj-gr*:
assumes *m>n*
shows *Proj-ket-set* {*m*} $*_V$ *run-B-count-upto n* = *0*
using *assms* **proof** (*induction n arbitrary*: *m*)
case *0*
have *proj-classical-set* {*m*} $*_V$ *ket 0* = *0*
by (*simp add*: *0.prems proj-classical-set-not-elem*)
then show *?case* **unfolding** *run-B-count-upto-def init-B-count-def Proj-ket-set-def*
by (*auto simp add*: *tensor-op-ell2*)
next
case (*Suc n*)

have *1*: *Proj-ket-set* {*m*} $*_V$ *run-B-count-upto* (*Suc n*) =
(*UA* (*Suc n*) $\otimes_o$ *id-cblinfun*) $*_V$ (*X-for-C*;*Y-for-C*) (*Uquery H*) $*_V$
Proj-ket-set {*m*} $*_V$ *U-S′ S* $*_V$ *run-B-count-upto n*
unfolding *Proj-ket-set-def*
by (*subst run-B-count-upto-I*)(*smt* (*verit*, *best*) *UqueryH-tensor-id-cblinfunC*
cblinfun-compose-id-left cblinfun-compose-id-right comp-tensor-op lift-cblinfun-comp(*4*))
have *m>0* **using** *Suc*(*2*) **by** *linarith*
then have *2*: *Proj-ket-set* {*m*} $*_V$ *U-S′ S* $*_V$ *run-B-count-upto n* =
(*S-embed S* $\otimes_o$ (*Uc* o_{CL} (*if m<d+1 then proj-classical-set* {*m−1*} *else proj-classical-set*
{*m*})))
$*_V$ *run-B-count-upto n* + (*not-S-embed S* $\otimes_o$ *proj-classical-set* {*m*}) $*_V$ *run-B-count-upto n*
unfolding *U-S′-def Proj-ket-set-def* **by** (*subst proj-Uc*[*symmetric*])
(*auto simp add*: *cblinfun.add-left cblinfun.add-right comp-tensor-op*
cblinfun-apply-cblinfun-compose[*symmetric*] *simp del*: *cblinfun-apply-cblinfun-compose*)
have *3*: (*S-embed S* $\otimes_o$ (*Uc* o_{CL} (*if m<d+1 then proj-classical-set* {*m−1*} *else proj-classical-set*
{*m*})))
$*_V$ *run-B-count-upto n* = *0*
proof (*cases m<d+1*)
case *True*
have ∗: (*S-embed S* $\otimes_o$ (*Uc* o_{CL} *proj-classical-set* {*m−1*})) $*_V$ *run-B-count-upto n* =
(*S-embed S* $\otimes_o$ *Uc*) $*_V$ *Proj-ket-set* {*m−1*} $*_V$ *run-B-count-upto n*
by (*simp add*: *Proj-ket-set-def comp-tensor-op lift-cblinfun-comp*(*4*))
have (*S-embed S* $\otimes_o$ *Uc*) $*_V$ *Proj-ket-set* {*m−1*} $*_V$ *run-B-count-upto n* = *0*
by (*simp add*: *Suc*(*1*) *Suc*(*2*) *less-diff-conv*)
then show *?thesis* **using** *True* ∗ **by** *auto*
next
case *False*
have *n<m* **using** *Suc*(*2*) **by** *auto*
have (*S-embed S* $\otimes_o$ (*Uc* o_{CL} *proj-classical-set* {*m*})) $*_V$ *run-B-count-upto n* = *0*
using *Suc*(*1*)[*OF* ‹*n<m*›] **unfolding** *Proj-ket-set-def*
by (*metis* (*no-types*, *opaque-lifting*) *cblinfun.zero-right*
cblinfun-apply-cblinfun-compose cblinfun-compose-id-right comp-tensor-op)
then show *?thesis* **using** *False* **by** *auto*
qed
have ∗: (*not-S-embed S* $\otimes_o$ *proj-classical-set* {*m*}) $*_V$ *run-B-count-upto n* =
(*not-S-embed S* $\otimes_o$ *id-cblinfun*) $*_V$ *Proj-ket-set* {*m*} $*_V$ *run-B-count-upto n*
unfolding *Proj-ket-set-def* **by** (*metis cblinfun-apply-cblinfun-compose cblinfun-compose-id-left*

cblinfun-compose-id-right comp-tensor-op)
have *4*: (*not-S-embed S* $\otimes_o$ *proj-classical-set* {*m*}) $*_V$ *run-B-count-upto n* = *0*
unfolding ∗ **by** (*simp add*: *Suc*(*1*) *Suc.prems Suc-lessD*)
show *?case* **by** (*subst 1*, *subst 2*, *subst 3*, *subst 4*) *auto*
qed

lemma *run-B-count-upto-proj-over*:
Proj-ket-set {*n+1..*} $*_V$ *run-B-count-upto n* = *0*
proof (*induction n*)
case *0*

then show *?case* **unfolding** *run-B-count-upto-def init-B-count-def Proj-ket-set-def*
using *proj-classical-set-upto*[*of 0*] **by** (*auto simp add*: *tensor-op-ell2*)
next
case (*Suc n*)
have *1*: *Proj-ket-set* $\{Suc\ n + 1..\}$ $*_V$ *run-B-count-upto* (*Suc n*) =
(*UA* (*Suc n*) $\otimes_o$ *id-cblinfun*) $*_V$ (*X-for-C*;*Y-for-C*) (*Uquery H*) $*_V$
Proj-ket-set $\{Suc\ n + 1..\}$ $*_V$ *U-S′ S* $*_V$ *run-B-count-upto n*
unfolding *Proj-ket-set-def*
by (*subst run-B-count-upto-I*) (*metis* (*no-types*, *lifting*)
cblinfun-apply-cblinfun-compose tensor-proj-UA-tensor-commute tensor-proj-UqueryH-commute)
have *2*: *Proj-ket-set* $\{Suc\ n + 1..\}$ $*_V$ *U-S′ S* $*_V$ *run-B-count-upto n* =
(*S-embed S* $\otimes_o$ *Uc*) $*_V$ *Proj-ket-set* (*if Suc n*>*d then* $\{Suc\ (Suc\ n)..\}$ *else* $\{Suc\ n..\}-\{d\}$)
$*_V$
run-B-count-upto n +
(*not-S-embed S* $\otimes_o$ *id-cblinfun*) $*_V$ *Proj-ket-set* $\{Suc\ n +1..\}$ $*_V$ *run-B-count-upto n*
using *proj-classical-set-over-Uc*[*symmetric*] **unfolding** *U-S′-def Proj-ket-set-def*
by (*auto simp add*: *cblinfun.add-left cblinfun.add-right comp-tensor-op*
cblinfun-apply-cblinfun-compose[*symmetric*] *simp del*: *cblinfun-apply-cblinfun-compose*)
have *Proj-ket-set* $\{Suc\ n\}$ $*_V$ *run-B-count-upto n* +
Proj-ket-set $\{Suc\ (Suc\ n)..\}$ $*_V$ *run-B-count-upto n* = *0*
using *proj-classical-set-split-Suc*[*of Suc n*] *Suc* **unfolding** *Proj-ket-set-def*
by (*simp add*: *cblinfun.add-left tensor-op-right-add proj-classical-set-def*)
then have *Suc-Suc*: *Proj-ket-set* $\{Suc\ (Suc\ n)..\}$ $*_V$ *run-B-count-upto n* = *0*
using *run-B-count-proj-gr*[*of n Suc n*] **by** *auto*
have *3*: (*S-embed S* $\otimes_o$ *Uc*) $*_V$ *Proj-ket-set* (*if Suc n*>*d then* $\{Suc\ (Suc\ n)..\}$ *else* $\{Suc$
$n..\}-\{d\}$)
$*_V$ *run-B-count-upto n* = *0*
proof (*cases Suc n* > *d*)
case *True*
show *?thesis* **using** *Suc-Suc True* **by** *auto*
next
case *False*
have *ket*: *ket* ‘ $\{Suc\ n..\}$ = *insert* (*ket d*) (*ket* ‘ ($\{Suc\ n..\} - \{d\}$)) **using** *False* **by** *auto*
have *proj-classical-set* $\{Suc\ n..\}$ = *proj* (*ket d*) + *proj-classical-set* ($\{Suc\ n..\} - \{d\}$)
unfolding *proj-classical-set-def ket* **by** (*intro Proj-orthog-ccspan-insert*) *auto*
then have *Proj-ket-set* $\{d\}$ $*_V$ *run-B-count-upto n* +
Proj-ket-set ($\{Suc\ n..\} - \{d\}$) $*_V$ *run-B-count-upto n* = *0*
by (*metis* (*no-types*, *opaque-lifting*) *One-nat-def Proj-ket-set-def Suc add-Suc-right*
cblinfun.add-left image-empty image-insert nat-arith.rule0 proj-classical-set-def
tensor-op-right-add)
then have *Proj-ket-set* ($\{Suc\ n..\} - \{d\}$) $*_V$ *run-B-count-upto n* = *0*
using *False run-B-count-proj-gr* **by** *auto*
then show *?thesis* **using** *False* **by** *auto*
qed
have *4*: (*not-S-embed S* $\otimes_o$ *id-cblinfun*) $*_V$ *Proj-ket-set* $\{Suc\ (Suc\ n)..\}$ $*_V$ *run-B-count-upto*
n = *0*
using *Suc-Suc* **by** *auto*
show *?case* **by** (*subst 1*, *subst 2*) (*auto simp add*: *3 4*)
qed

How Ψs relate to *run-B-count*. We can write *run-B-count* as a sum counting over all valid ket states in the counting register.

lemma *run-B-count-upto-split*:
run-B-count-upto n = ($\sum$ *i*<*n*+*1*. Ψ*s i* (*run-B-count-upto n*) $\otimes_s$ *ket i*)
proof −
have *run-B-count-upto n* =
($\sum$ *i*<*n*+*1*. (*tensor-ell2-right* (*ket i*)) o_{CL} (*tensor-ell2-right* (*ket i*)∗)) $*_V$ *run-B-count-upto n* +
Proj-ket-set {*n*+*1*..} $*_V$ *run-B-count-upto n*
using *id-cblinfun-tensor-split-finite*
by (*smt* (*verit*) *Compl-lessThan cblinfun.add-left cblinfun-id-cblinfun-apply finite-lessThan*)
also have ... = ($\sum$ *i*<*n*+*1*. (*tensor-ell2-right* (*ket i*)) o_{CL} (*tensor-ell2-right* (*ket i*)∗))
$*_V$ *run-B-count-upto n* **using** *run-B-count-upto-proj-over* **by** *auto*
finally have ∗: *run-B-count-upto n* =
($\sum$ *i*<*n*+*1*. (*tensor-ell2-right* (*ket i*)∗ $*_V$ *run-B-count-upto n*) $\otimes_s$ *ket i*)
by (*smt* (*verit*, *ccfv-SIG*) *cblinfun.sum-left cblinfun-apply-cblinfun-compose sum.cong tensor-ell2-right.rep-eq*)
show *?thesis* **unfolding** Ψ*s-def* **by** (*subst* ∗)(*use lessThan-Suc-atMost* **in** ‹*auto*›)
qed

lemma *run-B-count-split*:
run-B-count = ($\sum$ *i*<*d*+*1*. Ψ*s i run-B-count* $\otimes_s$ *ket i*)
unfolding *run-B-count-altdef* **by** (*rule run-B-count-upto-split*)

lemma *run-B-count-projection*:
Proj-ket-set {..<*n*+*1*} $*_V$ (*run-B-count-upto n*) = (*run-B-count-upto n*)
proof −
have *v*: *run-B-count-upto n* = ($\sum$ *i*<*n*+*1*. Ψ*s i* (*run-B-count-upto n*) $\otimes_s$ *ket i*)
using *run-B-count-upto-split* **by** *auto*
show *?thesis* **by** (*subst v*, *subst* (*2*) *v*, *subst cblinfun.sum-right*)
(*intro sum.cong*, *auto intro*!: *Proj-ket-set-vec*)
qed

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*
unbundle *no register-syntax*

end
theory *Pure-O2H*

imports *Run-Pure-B*

Run-Pure-B-count

begin

unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
unbundle *register-syntax*

context *pure-o2h*
begin

The probability that the find event occurs. That is the event that the adversary B notices that a query in S was made.

definition ‹$Pfind' = (norm\ (Snd\ (id\text{-}cblinfun\ -\ selfbutter\ (ket\ empty))\ *_V\ run\text{-}B))^2$›

What happens only to the first part of the memory when executing B or *B-count* is the same. This is recorded in Φ. The second registers only serve as counting registers.

definition Φ*s* **where**
Φ*s n* = *run-pure-adv n* (λ*i*. *UA i*) (λ-. *not-S-embed S*) *init X Y H*

We ensure that the Φ*s* is the same as the left part of Ψ_{count} (ie. *run-B-count*) with right part $|\ 0\rangle$.

lemma Ψ*s-run-B-count-upto-eq-*Φ*s*:
assumes *i*<*d*+*1*
shows Ψ*s 0* (*run-B-count-upto i*) = Φ*s i*
using *le0* **proof** (*induction i rule*: *Nat.dec-induct*)
case *base*
then show *?case* **unfolding** *run-B-count-upto-def init-B-count-def* Φ*s-def*
by (*auto simp add*: *tensor-op-ell2 tensor-ell2-ket* Ψ*s-def*)
next
case (*step n*)
then have *n*<*d* **using** *assms* **by** *auto*
have Ψ*s 0* (*run-B-count-upto* (*Suc n*)) = *UA* (*Suc n*) $*_V$ (*X*;*Y*) (*Uquery H*) $*_V$
Ψ*s 0* (*U-S′ S* $*_V$ *Proj-ket-set* {..<*n*+*1*} $*_V$
run-pure-adv n (λ*i*. *UA i* $\otimes_o$ *id-cblinfun*) (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*)
using *run-B-count-projection*
by (*auto simp add*: Ψ*s-id-cblinfun UqueryH-tensor-id-cblinfunC run-B-count-upto-def*)
also have ... = *UA* (*Suc n*) $*_V$ (*X*;*Y*) (*Uquery H*) $*_V$
(*not-S-embed S* $*_V$ *tensor-ell2-right* (*ket 0*)∗ $*_V$
run-pure-adv n (λ*i*. *UA i* $\otimes_o$ *id-cblinfun*) (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*)
using Ψ*s-U-S′-Proj-ket-upto*[*OF* ‹*n*<*d*›]
by (*metis* (*no-types*, *lifting*) Ψ*s-def cblinfun-apply-cblinfun-compose*)
also have ... = *UA* (*Suc n*) $*_V$ (*X*;*Y*) (*Uquery H*) $*_V$
not-S-embed S $*_V$ *run-pure-adv n UA* (λ-. *not-S-embed S*) *init X Y H*
using *step* **by** (*simp add*: Φ*s-def* Ψ*s-def run-B-count-upto-def*)
finally show *?case* **unfolding** Φ*s-def* **by** *auto*
qed

Analogously, Φs is the same as the left part of Ψ_{right} (ie. *run-B*) with right part | *embed* 0⟩.

lemma Ψ*s-run-B-upto-eq-*Φ*s*:
  assumes $i \leq d$
  shows Ψs *empty* (*run-B-upto i*) = Φs *i*
  using *le0* **proof** (*induction i rule*: *Nat.dec-induct*)
  case *base*
  then show *?case* **unfolding** *run-B-upto-def init-B-def* Φ*s-def*
    by (*auto simp add*: *tensor-op-ell2 tensor-ell2-ket* Ψ*s-def*)
next
  case (*step n*)
  then have $n<d$ **using** *assms* **by** *auto*
  have Ψs *empty* (*run-B-upto* (*Suc n*)) = *UA* (*Suc n*) $*_V$ ($X;Y$) (*Uquery H*) $*_V$
    Ψs *empty* ((*US S n*) $*_V$ *Proj-ket-upto* (*has-bits-upto n*) $*_V$ *run-B-upto n*)
    by (*subst run-B-upto-I*, *subst run-B-projection*[*OF* ‹$n<d$›])
      (*auto simp add*: Ψ*s-id-cblinfun UqueryH-tensor-id-cblinfunB*)
  also have ... = *UA* (*Suc n*) $*_V$ ($X;Y$) (*Uquery H*) $*_V$
    (*not-S-embed S* $*_V$ *tensor-ell2-right* (*ket empty*)$*$ $*_V$ *run-B-upto n*)
    using Ψ*s-US-Proj-ket-upto*[*OF* ‹$n<d$›]
    by (*metis* (*no-types*, *lifting*) Ψ*s-def cblinfun-apply-cblinfun-compose*)
  also have ... = *UA* (*Suc n*) $*_V$ ($X;Y$) (*Uquery H*) $*_V$
    not-S-embed S $*_V$ *run-pure-adv n UA* (λ-. *not-S-embed S*) *init X Y H*
    using *step* **by** (*simp add*: Φ*s-def* Ψ*s-def run-B-upto-def init-B-count-def init-B-def*)
  finally show *?case* **unfolding** Φ*s-def* **by** *auto*
qed

For the version of o2h with *norm UA* $\leq$ *1*, we need to introduce the following error term: when an adversary does not terminate, we get an additional term in the Pfind.

definition *P-nonterm* = (*norm run-B-count*)^2 − (*norm run-B*)^2

The One-Way-to-Hiding Lemma for pure states. Intuition: The difference of two games where we may change queries on a set S in game B can be bounded by the fining event *Pfind'*. Proof idea: We introduce an intermediate game B_{count} and show first equivalence between A and the left part of B_{count} in | 0⟩ and then equivalence of B_{count} and B in | 0⟩.

lemma *pure-o2h*: ‹(*norm* ((*run-A* $\otimes_s$ *ket empty*) − *run-B*))^2 $\leq$ (d+*1*) $*$ *Pfind'* + d $*$ *P-nonterm*›
proof −
  define $\Psi s'$ **where** $\Psi s'$ = (λi::*nat*. Ψs *i run-B-count*)
  have *eq16*: *run-B-count* = ($\sum i<d+1$. $\Psi s'$ *i* $\otimes_s$ (*ket i*))
    using *run-B-count-split* $\Psi s'$*-def* **by** *auto*
      — Equation (16)

— The operation N' connects the results of the game A and the counting game B_{count}.

  define N':: (*'mem* × *nat*) *update* **where**
    N' = (*id-cblinfun* $\otimes_o$ ($\sum i<d+1$. *butterfly* (*ket 0*) (*ket i*))) o_{CL} *Proj-ket-set* {..<d+*1*}

have ∗: *sum* (*Rep-ell2* (*ket c*)) {..<*d* + *1*} $*_C$ *ket 0* = *ket 0* **if** *c*<*d*+*1* **for** *c*
  by (*metis lessThan-iff proj-classical-set-elem sum-butterfly-ket0 sum-butterfly-ket0′ that*)
have *N′-ket*: *N′* (*ket* (*x*,*c*)) = *ket* (*x*,*0*) **if** *c*<*d*+*1* **for** *x c*
  unfolding *N′-def Proj-ket-set-def*
  apply (*subst tensor-ell2-ket*[*symmetric*], *subst comp-tensor-op*, *subst tensor-op-ell2*)
  apply (*subst* (*2*) *cblinfun-apply-cblinfun-compose*, *subst sum-butterfly-ket0′*)
  apply (*subst* ∗[*OF that*])
  by (*auto simp add*: *tensor-ell2-ket*[*symmetric*])
have *N′-tensor-ket*: *N′* $*_V$ *y* $\otimes_s$ *ket c* = *y* $\otimes_s$ *ket 0* **if** *c*<*d*+*1* **for** *c y* **unfolding** *N′-def*
proof (*subst cblinfun-apply-cblinfun-compose*, *subst Proj-ket-set-vec*)
  show *c* ∈ {..<*d* + *1*} **using** *that* **by** *auto*
  then show (*id-cblinfun* $\otimes_o$ (∑*i*<*d* + *1*. *butterfly* (*ket 0*) (*ket i*))) $*_V$ *y* $\otimes_s$ *ket c* = *y* $\otimes_s$ *ket 0*
    by (*subst tensor-op-ell2*, *subst sum-butterfly-ket0*) *auto*
qed

have *N′-UA*: *N′* o_{CL} (*UA i* $\otimes_o$ *id-cblinfun*) = (*UA i* $\otimes_o$ *id-cblinfun*) o_{CL} *N′* **for** *i*
  unfolding *N′-def* **by** (*simp add*: *Proj-ket-set-def comp-tensor-op*)
    — *N′* commutes with *UA*

have *N′-UqueryH*: *N′* o_{CL} (*X-for-C*;*Y-for-C*) (*Uquery H*) = (*X-for-C*;*Y-for-C*) (*Uquery H*) o_{CL} *N′*
  unfolding *UqueryH-tensor-id-cblinfunC* **by** (*simp add*: *N′-def Proj-ket-set-def comp-tensor-op*)
    — *N′* commutes with the oracle queries

have *N′-B-count*: *N′* o_{CL} *U-S′ S* = *N′*
proof (*unfold N′-def*, *intro equal-ket*, *safe*, *goal-cases*)
  case (*1 a b*)
  show *?case* **proof** (*cases b*<*d*+*1*)
    case *True*
    obtain *y* **where** *y*: *Uc* $*_V$ *ket b* = *ket y y*<*d*+*1*
      using *True Uc-ket-range-valid* **by** *auto*
    have *proj-y*:*proj-classical-set* {..<*Suc d*} $*_V$ *ket y* = *ket y* **using** ‹*y*<*d*+*1*›
      by (*metis Suc-eq-plus1 lessThan-iff proj-classical-set-elem*)
    have *proj-b*:*proj-classical-set* {..<*Suc d*} $*_V$ *ket b* = *ket b* **using** *True*
      by (*metis Suc-eq-plus1 lessThan-iff proj-classical-set-elem*)
    have *butter-y*: (∑*i*<*d*+*1*. *butterfly* (*ket 0*) (*ket i*)) $*_V$ *ket y* = *ket 0*
      using *sum-butterfly-ket0 y*(*2*) **by** *blast*
    have *butter-b*: (∑*i*<*d*+*1*. *butterfly* (*ket 0*) (*ket i*)) $*_V$ *ket b* = *ket 0*
      using *sum-butterfly-ket0 True* **by** *blast*
    have (*S-embed S* $*_V$ *ket a*) $\otimes_s$ (∑*i*<*d*+*1*. *butterfly* (*ket 0*) (*ket i*)) $*_V$ *ket y* +
      (*not-S-embed S* $*_V$ *ket a*) $\otimes_s$ (∑*i*<*d*+*1*. *butterfly* (*ket 0*) (*ket i*)) $*_V$ *ket b* =
      ket a $\otimes_s$ (∑*i*<*d*+*1*. *butterfly* (*ket 0*) (*ket i*)) $*_V$ *ket b*
      unfolding *butter-y butter-b* **by** (*metis S-embed-not-S-embed-add tensor-ell2-add1*)
    then show *?thesis* **using** *y proj-y proj-b*
      by(*auto simp add*: *tensor-op-ell2 tensor-op-ket cblinfun.add-right*
        U-S′-ket-split sum-butterfly-ket0 tensor-ell2-add1[*symmetric*] *Proj-ket-set-def*)
  next
    case *False*

then show *?thesis*
by (*metis* (*no-types*, *lifting*) *S-embed-not-S-embed-add U-S′-ket-split Uc-ket-greater lift-cblinfun-comp*(*4*) *not-less-eq semiring-norm*(*174*) *tensor-ell2-add1 tensor-ell2-ket*)
qed

qed
— N' *U-S′* = N'

have $0<d+1$ **using** *d-gr-0* **by** *auto*
have *N′-init-B-count*: $N' *_V$ *init-B-count* = *init-B-count*
unfolding *init-B-count-def* **using** *N′-def N′-tensor-ket*[*OF* ‹$0<d+1$›] **by** *blast*
— the initial state of *B-count* is invariant under N'

have *N′-run-B-count-upto-N′-run-A*: $N' *_V$ *run-B-count-upto n* =
$N' *_V$ (*run-pure-adv n UA* (λ-. *id-cblinfun*) *init X Y H* $\otimes_s$ *ket* (*0*)) **for** *n*
unfolding *run-B-count-upto-def run-A-def*
proof (*induction n*)
case *0*
have $N' *_V$ *U-S′ S* $*_V$ (*UA 0* $\otimes_o$ *id-cblinfun*) $*_V$ *init-B-count* =
(*UA 0* $\otimes_o$ *id-cblinfun* o_{CL} N') $*_V$ *init-B-count*
by (*auto simp add*: *cblinfun-apply-cblinfun-compose*[*symmetric*] *N′-B-count N′-init-B-count N′-UA cblinfun-compose-assoc*[*symmetric*] *simp del*: *cblinfun-apply-cblinfun-compose*)
also have ... = (*UA 0* $\otimes_o$ *id-cblinfun*) $*_V$ *init-B-count* **using** *N′-init-B-count* **by** *auto*
finally show *?case* **by** (*auto simp add*: *N′-UA N′-tensor-ket tensor-op-ell2 init-B-count-def*)
next
case (*Suc n*)
let *?run-pure-adv-B-count-d* =
run-pure-adv n (λ*i*. *UA i* $\otimes_o$ *id-cblinfun*) (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*
let *?run-pure-adv-A-d* = *run-pure-adv n UA* (λ-. *id-cblinfun*) *init X Y H*
have $N' *_V$ *run-pure-adv* (*n+1*) (λ*i*. *UA i* $\otimes_o$ *id-cblinfun*) (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H* =
(*UA* (*Suc n*) $\otimes_o$ *id-cblinfun* o_{CL} (*X-for-C*;*Y-for-C*) (*Uquery H*) o_{CL} N') $*_V$ *?run-pure-adv-B-count-d*

using *N′-B-count N′-UA N′-UqueryH*
by (*auto simp add*: *cblinfun-apply-cblinfun-compose*[*symmetric*] *cblinfun-compose-assoc*[*symmetric*] *simp del*: *cblinfun-apply-cblinfun-compose*)
(*auto simp add*: *cblinfun-compose-assoc*)
also have ... = (*UA* (*Suc n*) $\otimes_o$ *id-cblinfun* o_{CL} (*X-for-C*;*Y-for-C*) (*Uquery H*)) $*_V$
$N' *_V$ *?run-pure-adv-A-d* $\otimes_s$ *ket 0*
by (*simp add*: *Suc.IH*)
also have ... = (N' o_{CL} *UA* (*Suc n*) $\otimes_o$ *id-cblinfun* o_{CL} (*X-for-C*;*Y-for-C*) (*Uquery H*))
$*_V$
?run-pure-adv-A-d $\otimes_s$ *ket 0*
using *N′-B-count N′-UA N′-UqueryH*
by (*auto simp add*: *cblinfun-apply-cblinfun-compose*[*symmetric*] *cblinfun-compose-assoc*[*symmetric*] *simp del*: *cblinfun-apply-cblinfun-compose*)
(*auto simp add*: *cblinfun-compose-assoc*)
finally show *?case*

by (*auto simp add*: *UqueryH-tensor-id-cblinfunC tensor-op-ell2 nth-append*)
qed

have *N′-run-B-count-N′-run-A*: $N' *_V$ *run-B-count* = $N' *_V$ (*run-A* $\otimes_s$ *ket* (*0*))
unfolding *run-B-count-altdef* **by** (*subst N′-run-B-count-upto-N′-run-A*)
(*auto simp add*: *run-A-def*)
— N' does not touch the second part of the memory and *run-B-count* and *run-A* do the same on *′mem*

then have *N′-run-B-count-run-A*: $N' *_V$ *run-B-count* = *run-A* $\otimes_s$ *ket 0*
by (*simp add*: *N′-tensor-ket*)
— Relation between A and B_{count}

have ($\sum$ *i*<*d*+*1*. Ψ*s′ i* $\otimes_s$ *ket* (*0*::*nat*)) = $N' *_V$ *run-B-count* **unfolding** *eq16*
by (*subst cblinfun.sum-right*, *intro sum.cong*) (*auto simp add*: *N′-tensor-ket*)
moreover have $N' *_V$ *run-B-count* = *run-A* $\otimes_s$ *ket* (*0*::*nat*)
unfolding *N′-run-B-count-N′-run-A* **by** (*auto simp add*: *N′-tensor-ket*)
ultimately have Ψ*s′-run-A*:
($\sum$ *i*<*d*+*1*. Ψ*s′ i* $\otimes_s$ *ket* (*0*::*nat*)) = *run-A* $\otimes_s$ *ket* (*0*) **by** *auto*

have *eq17*: *run-A* = ($\sum$ *i*<*d*+*1*. Ψ*s′ i*)
proof −
have *run-A* = (*tensor-ell2-right* (*ket 0*))∗ $*_V$ (*run-A* $\otimes_s$ *ket* (*0*::*nat*)) **by** *auto*
also have ... = ($\sum$ *i*<*d*+*1*. Ψ*s′ i*)
unfolding Ψ*s′-run-A*[*symmetric*]
by (*subst tensor-ell2-sum-left*[*symmetric*], *subst tensor-ell2-right-adj-apply*, *auto*)
finally show *?thesis* **by** *blast*
qed
— Equation (17)
— Representation of A in terms of parts of states in B_{count}

define Ψ*sB* **where** Ψ*sB* = (λ*i*::*bool list*. Ψ*s* (*list-to-l i*) *run-B*)
have *eq18*: *run-B* = ($\sum$ *l*∈*len-d-lists*. Ψ*sB l* $\otimes_s$ *ket* (*list-to-l l*))
by (*subst run-B-split*, *unfold* Ψ*sB-def*) *auto*
— Equation (18)

have Ψ*sB*-Φ*s*: Ψ*sB empty-list* = Φ*s d* **by** (*simp add*: Ψ*sB-def* Ψ*s-run-B-upto-eq*-Φ*s run-B-altdef*)

have *eq19*: Ψ*s′ 0* = Ψ*sB empty-list*
proof −
have Ψ*s′ 0* = Ψ*s 0* (*run-B-count-upto d*) **unfolding** Ψ*s′-def run-B-count-altdef* **by** *auto*
also have ... = Φ*s d* **by** (*simp add*: Ψ*s-run-B-count-upto-eq*-Φ*s*)
also have ... = Ψ*sB empty-list* **unfolding** Ψ*sB*-Φ*s* **by** *auto*
finally show *?thesis* **by** *blast*
qed
— Equation (19)
— Relating the games B and B_{count}.

— Now, we argue about the probabilits of the find event and the outcome states.

have *eq20*:*norm* (Ψ*sB empty-list*)^*2* = (*norm run-B*)^*2* − *Pfind′*
proof −
have *norm* (Ψ*sB empty-list*)^*2* = (*norm* (Φ*s d*))^*2* **unfolding** Ψ*sB-def run-B-altdef*
by (*auto simp add*: Ψ*s-run-B-upto-eq-*Φ*s*)
also have ... = (*norm run-B*)^*2* − (*norm* (*run-B* − Φ*s d* $\otimes_s$ *ket empty*))^2
proof −
have *norm-B*: *Re* (*run-B* $\bullet_C$ *run-B*) = (*norm run-B*)^*2*
unfolding *power2-norm-eq-cinner*[*symmetric*] *norm-run-B* **by** *auto*
have *cinner-B-*Ψ: (*run-B*) $\bullet_C$ (Φ*s d* $\otimes_s$ *ket empty*) = (Φ*s d*) $\bullet_C$ (Φ*s d*)
proof −
have *list-to-l x* = *empty* ⟹ *x*∈*len-d-lists* ⟹ *x*=*empty-list* **for** *x*
using *inj-list-to-l inj-onD* **by** *fastforce*
then have ∗∗: Ψ*sB empty-list* $\bullet_C$ Ψ*sB empty-list* = *sum* ((λ*l*. Ψ*sB l* $\bullet_C$ Ψ*sB empty-list* ∗ (*ket* (*list-to-l l*) $\bullet_C$ *ket empty*))) *len-d-lists*
by (*subst sum.remove*[*OF finite-len-d-lists empty-list-len-d*]) (*auto intro*!: *sum.neutral*)
show *?thesis* **unfolding** *eq18* Ψ*sB-*Φ*s*[*symmetric*] **unfolding** ∗∗
by (*auto simp add*: *cinner-sum-left*)
qed
have (*norm* (Φ*s d*))^*2* + (*norm* (*run-B* − Φ*s d* $\otimes_s$ *ket empty*))^*2* =
Re (Φ*s d* $\bullet_C$ Φ*s d* + (*run-B* − Φ*s d* $\otimes_s$ *ket empty*) $\bullet_C$ (*run-B* − Φ*s d* $\otimes_s$ *ket empty*))
unfolding *power2-norm-eq-cinner′* **by** *auto*
also have ... = *Re* (*2* ∗ (Φ*s d* $\bullet_C$ Φ*s d*) + *run-B* $\bullet_C$ *run-B* − (*run-B*) $\bullet_C$ (Φ*s d* $\otimes_s$ *ket empty*) −
(Φ*s d* $\otimes_s$ *ket empty*) $\bullet_C$ (*run-B*))
by (*auto simp add*: *algebra-simps norm-B*)
also have ... = (*norm run-B*)^*2* **by** (*subst* (*3*)*cinner-commute*, *unfold cinner-B-*Ψ)
(*auto simp add*: *norm-B*)
finally have (*norm* (Φ*s d*))^*2* + (*norm* (*run-B* − Φ*s d* $\otimes_s$ *ket empty*))^*2* = (*norm run-B*)^*2* **by** *auto*
then show *?thesis* **by** *auto*
qed
also have ... = (*norm run-B*)^*2* − (*norm* (*run-B* − (*tensor-ell2-right* (*ket empty*)∗ $*_V$ *run-B*)
$\otimes_s$ *ket empty*))^2 **unfolding** *run-B-def*
by (*subst* Ψ*s-run-B-upto-eq-*Φ*s*[*symmetric*])(*auto simp add*: Ψ*s-def run-B-upto-def*)
also have ... = (*norm run-B*)^*2* − *Pfind′* **unfolding** *Pfind′-def*
by (*auto simp add*: *Snd-def tensor-op-right-minus cblinfun.diff-left*
id-cblinfun-selfbutter-tensor-ell2-right)
finally show *?thesis* **by** *auto*
qed
— Equation (20)

have *eq20′*:*norm* (Ψ*s′ 0*)^*2* = *norm* (*run-B*)^*2* − *Pfind′* **unfolding** *eq19* **using** *eq20* **by** *auto*
— Analog to Equation (20)

have *sum-to-1′*: (∑ *i*<*d*+*1*. *norm* (Ψ*s′ i*)^*2*) = (*norm run-B-count*)^*2*

proof −
  have ($\sum i<d+1.$ *norm* ($\Psi s'$ *i*)^2) =
    ($\sum i<d+1.$ *norm* (($\Psi s'$ *i*) $\otimes_s$ *ket* (*i*::*nat*))^2)
    by (*intro sum.cong*, *auto simp add*: *norm-tensor-ell2*)
  also have ... = *norm* ($\sum i<d+1.$ ($\Psi s'$ *i*) $\otimes_s$ *ket* (*i*::*nat*))^2
    by (*rule pythagorean-theorem-sum*[*symmetric*], *auto*)
  also have ... = *norm* (*run-B-count*)^2 **unfolding** *eq16* **by** *auto*
  finally show *?thesis* **by** *auto*
qed
then have *eq21'*: ($\sum i=1..<d+1.$ *norm* ($\Psi s'$ *i*)^2) = *Pfind'* + *P-nonterm*
proof −
  have ($\sum i<d+1.$ *norm* ($\Psi s'$ *i*)^2) =
    norm ($\Psi s'$ *0*)^2 + ($\sum i=1..<d+1.$ *norm* ($\Psi s'$ *i*)^2)
    unfolding *atLeast0AtMost lessThan-atLeast0*
    by (*subst sum.atLeast-Suc-lessThan*[*OF* ‹*0*<*d*+*1*›]) *auto*
  then show *?thesis* **using** *eq20' sum-to-1'* **unfolding** *P-nonterm-def* **by** *linarith*
qed
  — Part of Equation (21)

have *sum-to-1*:($\sum l\in$*len-d-lists*. *norm* (ΨsB *l*)^2) = (*norm run-B*)^2
proof −
  have ($\sum l\in$*len-d-lists*. *norm* (ΨsB *l*)^2) =
    ($\sum l\in$*len-d-lists*. *norm* ((ΨsB *l*) $\otimes_s$ *ket* (*list-to-l l*))^2)
    by (*intro sum.cong*, *auto simp add*: *norm-tensor-ell2*)
  also have ... = *norm* ($\sum l\in$*len-d-lists*. ΨsB *l* $\otimes_s$ *ket* (*list-to-l l*))^2
  proof −
    have $a \neq a' \Longrightarrow a\in$*len-d-lists* $\Longrightarrow a'\in$*len-d-lists* $\Longrightarrow$ *list-to-l a* $\neq$ (*list-to-l a'*)
      for *a a'* **by** (*meson inj-list-to-l inj-onD*)
    then show *?thesis* **by** (*subst pythagorean-theorem-sum*) *auto*
  qed
  also have ... = *norm* (*run-B*)^2 **unfolding** *eq18* **by** *auto*
  finally show *?thesis* **by** *auto*
qed
then have *eq21*: ($\sum l\in$*has-bits* {*0*..<*d*}. *norm* (ΨsB *l*)^2) = *Pfind'*
proof −
  have ($\sum l\in$*len-d-lists*. *norm* (ΨsB *l*)^2) =
    norm (ΨsB *empty-list*)^2 + ($\sum l\in$*has-bits* {*0*..<*d*}. *norm* (ΨsB *l*)^2)
    by (*subst sum.remove*[*of - empty-list*], *unfold len-d-empty-has-bits*) *auto*
  then show *?thesis* **using** *eq20 sum-to-1* **unfolding** *P-nonterm-def* **by** *auto*
qed
  — Part of Equation (21)

— Finally, we can subsume all our findings and prove the O2H Lemma.

show *?thesis*
proof −
  have (*norm* (*run-A* $\otimes_s$ *ket empty* − *run-B*))^2 =
      (*norm* (*run-B* − *run-A* $\otimes_s$ *ket empty*))^2
    by (*subst norm-minus-cancel*[*symmetric*], *auto*)

also have ... $=$ (*norm* ((Ψ*sB empty-list* $-$ *run-A*) $\otimes_s$ *ket empty* $+$
($\sum$ *l*$\in$*has-bits* {*0*..<*d*}. Ψ*sB l* $\otimes_s$ *ket* (*list-to-l l*))))^2
proof $-$
have $*$: (Ψ*sB empty-list* $-$ *run-A*) $\otimes_s$ *ket empty* $=$
Ψ*sB empty-list* $\otimes_s$ *ket empty* $-$ *run-A* $\otimes_s$ *ket empty*
using *tensor-ell2-diff1* **by** *blast*
show *?thesis* **unfolding** *eq18*
by (*subst sum.remove*[*of - empty-list*], *unfold* $*$ *len-d-empty-has-bits*)
(*auto simp add*: *algebra-simps*)
qed
also have ... $=$ (*norm* ((Ψ*sB empty-list* $-$ *run-A*) $\otimes_s$ *ket* (*empty*)))^2 $+$
(*norm* ($\sum$ *l*$\in$*has-bits* {*0*..<*d*}. Ψ*sB l* $\otimes_s$ *ket* (*list-to-l l*)))^2
proof $-$
have *l*: (*if empty* $=$ *list-to-l l then 1 else 0*) $=$ *0* **if** $l \in$ *has-bits* {*0*..<*d*} **for** *l*
using *that* **by** (*metis DiffE empty-iff has-bits-empty len-d-empty-has-bits has-bits-not-empty*)

then have $*$: ($\sum$ *l*$\in$*has-bits* {*0*..<*d*}.
(Ψ*sB empty-list* $-$ *run-A*) $\cdot_C$ Ψ*sB l* $*$ (*if empty* $=$ *list-to-l l then 1 else 0*)) $=$ *0*
by (*smt* (*verit*, *best*) *class-semiring.add.finprod-all1 semiring-norm*(*64*))
have ($\sum$ *l*$\in$*has-bits* {*0*..<*d*} $\cap$ {*l*. *empty* $=$ *list-to-l l*}.
(Ψ*sB empty-list* $-$ *run-A*) $\cdot_C$ Ψ*sB l*) $=$ *0* **by** (*subst sum.inter-restrict*, *simp*)
(*subst* (*2*) $*$[*symmetric*], *intro sum.cong*, *auto*)
then have *is-orthogonal* ((Ψ*sB empty-list* $-$ *run-A*) $\otimes_s$ *ket empty*)
($\sum$ *l*$\in$*has-bits* {*0*..<*d*}. Ψ*sB l* $\otimes_s$ *ket* (*list-to-l l*))
by (*auto simp add*: *cinner-sum-right cinner-ket*)
then show *?thesis* **by** (*rule pythagorean-theorem*)
qed
also have ... $=$ (*norm* ((Ψ*sB empty-list* $-$ *run-A*) $\otimes_s$ *ket empty*))^2 $+$
($\sum$ *l*$\in$*has-bits* {*0*..<*d*}. *norm* (Ψ*sB l*)^*2*)
proof $-$
have $a \neq a' \Longrightarrow$ *a*$\in$*has-bits* {*0*..<*d*} $\Longrightarrow$ *a'*$\in$*has-bits* {*0*..<*d*} $\Longrightarrow$
list-to-l a $\neq$ *list-to-l a'* **for** *a a'*
by (*metis DiffE inj-list-to-l inj-onD len-d-empty-has-bits*)
then show *?thesis*
by (*subst pythagorean-theorem-sum*) (*auto simp add*: *norm-tensor-ell2*)
qed
also have ... $=$ (*norm* ((Ψ*sB empty-list* $-$ *run-A*) $\otimes_s$ *ket empty*))^2 $+$ *Pfind'*
using *eq21* **by** *auto*
also have ... $=$ *norm* ($\sum$ *i*=*1*..<*d*+*1*. Ψ*s' i*)^*2* $+$ *Pfind'*
proof $-$
have $*$: Ψ*s' 0* $-$ ($\sum$ *i*<*d*+*1*. Ψ*s' i*) $=$ $-$ ($\sum$ *i*=*1*..<*d*+*1*. Ψ*s' i*)
unfolding *lessThan-atLeast0*
by (*subst sum.atLeast-Suc-lessThan*[*OF* ‹*0*<*d*+*1*›]) *auto*
have $**$: *norm* ((Ψ*s' 0* $-$ ($\sum$ *i*<*d*+*1*. Ψ*s' i*)) $\otimes_s$ *ket empty*) $=$
norm ($\sum$ *i* $=$ *1*..<*d*+*1*. Ψ*s' i*)
unfolding $*$ **by** (*simp add*: *norm-tensor-ell2*)
show *?thesis* **unfolding** *eq17 eq19*[*symmetric*] $**$ **by** *auto*
qed
also have ... $\leq$ ($\sum$ *i*=*1*..<*d*+*1*. *norm* (Ψ*s' i*))^*2* $+$ *Pfind'*

using *eq20 eq21′*
by (*smt* (*verit*) *power-mono*[*OF norm-sum norm-ge-zero*] *sum.cong*)
also have ... $\leq d * (\sum i{=}1..{<}d{+}1.\ norm\ (\Psi s'\ i)\hat{}2) + Pfind'$
by (*subst add-le-cancel-right*)
(*use arith-quad-mean-ineq*[*of* {*1..<d+1*} (λ*i. norm* (Ψ*s′ i*))] **in** ‹*auto*›)
also have ... $= (d{+}1) * Pfind' + d * P\text{-}nonterm$
unfolding *eq21′* **by** (*simp add*: *algebra-simps*)
finally show *?thesis* **by** *linarith*
qed
qed

lemma *pure-o2h-sqrt*: ‹*norm* ((*run-A* $\otimes_s$ *ket empty*) − *run-B*) ≤ *sqrt* ((*d+1*) ∗ *Pfind′* + *d* ∗ *P-nonterm*)›
using *pure-o2h real-le-rsqrt* **by** *blast*

lemma *error-term-pos*:
(*d+1*) ∗ *Pfind′* + *d* ∗ *P-nonterm* ≥ *0*
using *pure-o2h* **by** (*smt* (*verit*, *best*) *power2-diff sum-squares-bound*)

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*
unbundle *no register-syntax*

end
theory *Estimation*

imports *Complex-Main*

begin

6 Auxiliary lemma: Estimation

For the proof of the mixed state O2H, we need an auxiliary lemma on the square roots of sums.

lemma *abc-ineq*:
assumes *a≥0 b≥0 c≥0* |*sqrt a* − *sqrt b*| ≤ *sqrt c*
shows *a* + *b* ≤ *c* + *2* ∗ *sqrt* (*a* ∗ *b*)
proof −
have |*sqrt a* − *sqrt b*|^*2* ≤ *sqrt c* ^*2* **using** *assms* **by** (*simp add*: *sqrt-ge-absD*)
then show *?thesis* **by** (*auto simp add*: *algebra-simps power2-diff assms real-sqrt-mult*)
qed

lemma *two-ab-ineq*:
assumes *a≥0 b≥0*
shows *2* ∗ *sqrt* (*a* ∗ *b*) ≤ *a* + *b*
proof −

```
  have 0 ≤ (sqrt a − sqrt b)^2 by auto
  then show ?thesis by (auto simp add: algebra-simps power2-diff assms real-sqrt-mult)
qed

lemma sqrt-estimate-real:
  assumes fin-M: finite M
    and pos-t: ∀x∈M. t x ≥ (0::real)
    and pos-u: ∀x∈M. u x ≥ (0::real)
    and pos-v: ∀x∈M. v x ≥ (0::real)
    and pos-a: ∀x∈M. a x ≥ (0::real)
    and ineq: ∀x∈M. | sqrt (t x) − sqrt (u x)| ≤ sqrt (v x)
  shows |sqrt (∑x∈M. a x ∗ t x) − sqrt (∑x∈M. a x ∗ u x)| ≤ sqrt (∑x∈M. a x ∗ v x)
  using assms proof (induction M)
  case empty
  then show ?case by auto
next
  case (insert y N)
  define tN where tN = (∑x∈N. a x ∗ t x)
  have pos-tN[simp]: 0 ≤ tN unfolding tN-def by (simp add: insert.prems(1) insert.prems(4)
sum-nonneg)
  define uN where uN = (∑x∈N. a x ∗ u x)
  have pos-uN[simp]: 0 ≤ uN unfolding uN-def by (simp add: insert.prems(2) insert.prems(4)
sum-nonneg)
  define vN where vN = (∑x∈N. a x ∗ v x)
  have pos-vN[simp]: 0 ≤ vN unfolding vN-def by (simp add: insert.prems(3) insert.prems(4)
sum-nonneg)
  have ineqN: |sqrt tN − sqrt uN| ≤ sqrt vN
    by (simp add: insert.prems(1,4) insert(3,5,6,8) tN-def uN-def vN-def)
  have 2: tN + uN ≤ vN + 2 ∗ sqrt (tN ∗ uN)
    by (intro abc-ineq)(auto simp add: ineqN)
  have a y ≥ 0 using insert by auto
  have 3a: t y + u y ≤ v y + 2 ∗ sqrt (t y ∗ u y) by (intro abc-ineq) (auto simp add: insert)
  have 3: a y ∗ t y + a y ∗ u y ≤ a y ∗ v y + 2 ∗ a y ∗ sqrt (t y ∗ u y)
    by (use mult-left-mono[OF 3a ‹a y ≥ 0›] in ‹auto simp add: algebra-simps›)
  have 5: sqrt ((tN + a y ∗ t y) ∗ (uN + a y ∗ u y)) ≥ sqrt (tN ∗ uN) + a y ∗ sqrt (t y ∗ u y)
  proof −
    have (sqrt (tN ∗ uN) + a y ∗ sqrt (t y ∗ u y)) ^2 = tN ∗ uN + (a y)^2 ∗ t y ∗ u y
      + 2 ∗ a y ∗ sqrt (tN ∗ uN ∗ t y ∗ u y)
      by (auto simp add: algebra-simps power2-sum insert real-sqrt-mult[symmetric])
    also have ... = tN ∗ uN + (a y)^2 ∗ t y ∗ u y + 2 ∗ sqrt ((a y)^2 ∗ tN ∗ uN ∗ t y ∗ u y)
      by (auto simp add: real-sqrt-mult ‹a y ≥ 0›)
    also have ... = tN ∗ uN + (a y)^2 ∗ t y ∗ u y + 2 ∗ sqrt ((a y ∗ tN ∗ u y) ∗ (a y ∗ uN
∗ t y))
      by (auto simp add: algebra-simps power2-eq-square)
    also have ... ≤ tN ∗ uN + (a y)^2 ∗ t y ∗ u y + a y ∗ tN ∗ u y + a y ∗ uN ∗ t y
      using two-ab-ineq[of a y ∗ tN ∗ u y a y ∗ uN ∗ t y] by (auto simp add: insert)
    also have ... = sqrt ((tN + a y ∗ t y) ∗ (uN + a y ∗ u y))^2 using real-sqrt-pow2
      by (auto simp add: insert algebra-simps power2-eq-square)
    finally show ?thesis by (simp add: ‹0 ≤ a y› insert(4,5) real-le-rsqrt)
```

qed
have $|sqrt\ (\sum x{\in}insert\ y\ N.\ a\ x * t\ x) - sqrt\ (\sum x{\in}insert\ y\ N.\ a\ x * u\ x)| =$
$|sqrt\ (tN + a\ y * t\ y) - sqrt\ (uN + a\ y * u\ y)|$
by (*subst sum.insert*[*OF insert*(*1*,*2*)])+ (*auto simp add*: *tN-def uN-def algebra-simps*)
also have $\ldots \le sqrt\ (vN + a\ y * v\ y)$
proof −
have $|sqrt\ (tN + a\ y * t\ y) - sqrt\ (uN + a\ y * u\ y)|\hat{}2 =$
$tN + a\ y * t\ y + uN + a\ y * u\ y - 2 * sqrt((tN + a\ y * t\ y) * (uN + a\ y * u\ y))$
by (*auto simp add*: *algebra-simps power2-diff insert real-sqrt-mult*[*symmetric*])
also have $\ldots \le vN + a\ y * v\ y + 2 * sqrt(tN * uN) + 2 * a\ y * sqrt\ (t\ y * u\ y) -$
$2 * sqrt\ ((tN + a\ y * t\ y) * (uN + a\ y * u\ y))$
using *2 3* **by** *auto*
finally have $|sqrt\ (tN + a\ y * t\ y) - sqrt\ (uN + a\ y * u\ y)|\hat{}2 \le vN + a\ y * v\ y$
using *5* **by** *auto*
then show *?thesis* **using** *real-le-rsqrt* **by** *blast*
qed
also have $\ldots = sqrt\ (\sum x{\in}insert\ y\ N.\ a\ x * v\ x)$
by (*subst sum.insert*[*OF insert*(*1*,*2*)]) (*auto simp add*: *vN-def*)
finally show *?case* **by** *linarith*
qed

end
theory *Limit-Process*

imports *Run-Adversary*

begin

unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
unbundle *register-syntax*

7 Limit Processes

We need some concept of limes of Kraus families, i.e. finite Kraus maps tending to a Kraus map. Therefore, we define a filter on the Kraus family.

kf-elems is the set of Kraus maps with only one element that are part of the original Kraus map.

lift-definition *kf-elems* ::
('*a*::*chilbert-space*, '*b*::*chilbert-space*, *unit*) *kraus-family* ⇒ ('*a*, '*b*, *unit*) *kraus-family set* **is**
$\lambda E.\ (\lambda x.\ \{x\})$ ‘ E
apply (*intro CollectI kraus-family-if-finite*)
by (*auto simp*: *kraus-family-def*)

lemma *kf-elems-Rep-kraus-family*:

kf-elems $\mathfrak{E}$ = (λx. *Abs-kraus-family* $\{x\}$) ' *Rep-kraus-family* $\mathfrak{E}$
unfolding *kf-elems-def* **by** *auto*

lemma *kf-elems-finite*:
assumes $F \in$ *kf-elems* $\mathfrak{E}$
shows *finite* (*Rep-kraus-family* F)
using *assms* **by** *transfer auto*

lemma *kf-bound-of-elems*:
assumes $F \in$ *kf-elems* E
shows *kf-bound* $F \leq$ *kf-bound* E
proof −
have *subset*: *Rep-kraus-family* $F \subseteq$ *Rep-kraus-family* E **using** *assms* **by** *transfer auto*
have *kf-bound* F = ($\sum$(E, u)$\in$*Rep-kraus-family* F. $E*$ o_{CL} E)
using *assms kf-bound-finite kf-elems-finite* **by** *blast*
also have ... $\leq$ *kf-bound* E **using** *kf-bound-geq-sum*[*OF subset*] **by** *auto*
finally show *?thesis* **by** *linarith*
qed

lemma *kf-elems-card-1*:
assumes $F \in$ *kf-elems* E
shows *card* (*Rep-kraus-family* F) = *1*
using *assms* **by** *transfer auto*

lemma *inj-on-kf-singleton*:
inj-on (λx. *Abs-kraus-family* $\{x\}$) (*Rep-kraus-family* $\mathfrak{E}$)
apply (*rule inj-onI*)
apply (*subst* (*asm*) *Abs-kraus-family-inject*)
using *Rep-kraus-family kraus-family-def* **by** *auto*

lemma *kf-apply-singleton*:
fixes E :: ‹*'a*::*chilbert-space* $\Rightarrow_{CL}$ *'b*::*chilbert-space* × *'x*›
assumes ‹*fst* $E \neq$ *0*›
shows *kf-apply* (*Abs-kraus-family* $\{E\}$) ϱ = *sandwich-tc* (*fst* E) ϱ
apply (*subst kf-apply.abs-eq*)
using *assms*
apply (*simp add*: *eq-onp-same-args*)
by *simp*

lemma *kf-apply-summable-on-kf-elems*:
fixes $\mathfrak{E}$ :: (*'a*::*chilbert-space*,*'b*::*chilbert-space*,*unit*) *kraus-family*
shows ($\lambda\mathfrak{F}$. *kf-apply* $\mathfrak{F}$ ϱ) *summable-on* (*kf-elems* $\mathfrak{E}$)
proof −
have $*$: ‹*kf-apply* (*Abs-kraus-family* $\{E\}$) ϱ = *sandwich-tc* (*fst* E) ϱ›
if ‹$E \in$ *Rep-kraus-family* $\mathfrak{E}$› **for** E
apply (*subst kf-apply-singleton*)
using *that Rep-kraus-family*
apply (*force intro*!: *simp*: *kraus-family-def*)
by *simp*

show *?thesis*
unfolding *kf-elems-Rep-kraus-family*
apply (*subst summable-on-reindex*[*OF inj-on-kf-singleton*])
apply (*rule summable-on-cong*[**where** *g*=‹λE. *sandwich-tc* (*fst E*) ϱ›, *THEN iffD2*])
using ∗
apply *force*
using *kf-apply-summable*
by (*force simp*: *case-prod-unfold*)
qed

lemma *kf-apply-has-sum-kf-elems*:
fixes $\mathfrak{E}$ :: (*'a*::*chilbert-space*,*'b*::*chilbert-space*,*unit*) *kraus-family*
shows (($\lambda\mathfrak{F}$. *kf-apply* $\mathfrak{F}$ ϱ) *has-sum* (*kf-apply* $\mathfrak{E}$ ϱ)) (*kf-elems* $\mathfrak{E}$)
proof −
have ∗: ‹*kf-apply* (*Abs-kraus-family* {*E*}) ϱ = *sandwich-tc* (*fst E*) ϱ›
if ‹*E* ∈ *Rep-kraus-family* $\mathfrak{E}$› **for** *E*
apply (*subst kf-apply-singleton*)
using *that Rep-kraus-family*
apply (*force intro*!: *simp*: *kraus-family-def*)
by *simp*
show *?thesis*
unfolding *kf-elems-Rep-kraus-family*
apply (*subst has-sum-reindex*[*OF inj-on-kf-singleton*])
apply (*rule has-sum-cong*[**where** *g*=‹λE. *sandwich-tc* (*fst E*) ϱ›, *THEN iffD2*])
using ∗
apply *force*
by (*metis* (*no-types*, *lifting*) *has-sum-cong kf-apply-has-sum split-def*)
qed

lemma *kf-apply-abs-summable-on-kf-elems*:
fixes $\mathfrak{E}$ :: (*'a*::*chilbert-space*,*'b*::*chilbert-space*,*unit*) *kraus-family*
shows ($\lambda\mathfrak{F}$. *kf-apply* $\mathfrak{F}$ ϱ) *abs-summable-on* (*kf-elems* $\mathfrak{E}$)
proof −
have ∗: ‹*kf-apply* (*Abs-kraus-family* {*E*}) ϱ = *sandwich-tc* (*fst E*) ϱ›
if ‹*E* ∈ *Rep-kraus-family* $\mathfrak{E}$› **for** *E*
apply (*subst kf-apply-singleton*)
using *that Rep-kraus-family*
apply (*force intro*!: *simp*: *kraus-family-def*)
by *simp*
show *?thesis*
unfolding *kf-elems-Rep-kraus-family*
apply (*subst summable-on-reindex*[*OF inj-on-kf-singleton*])
apply (*subst o-def*)
apply (*rule summable-on-cong*[**where** *g*=‹λE. *norm* (*sandwich-tc* (*fst E*) ϱ)›, *THEN iffD2*])
using ∗
apply *force*
using *Rep-kraus-family kf-apply-abs-summable*
by (*force simp*: *case-prod-unfold*)
qed

Now, we can define a sub-adversary. An adversary is modeled by a sequence of n Kraus maps. A sub-adversary is then defined as a sequence of n elements of the respective Kraus maps. Adding all sub-adversaries together yields the original Kraus map.

definition *finite-kraus-subadv* :: *'a kraus-adv* ⇒ *nat* ⇒ *'a kraus-adv set* **where**
finite-kraus-subadv $\mathfrak{E}$ *n* = *PiE* {*0..<n+1*} (λ*i*. *kf-elems* ($\mathfrak{E}$ *i*))

lemma *finite-kraus-subadv-I*:
assumes *f* ∈ *finite-kraus-subadv* $\mathfrak{E}$ *n i<n+1*
shows *f i* ∈ *kf-elems* ($\mathfrak{E}$ *i*)
using *assms* **unfolding** *finite-kraus-subadv-def* **by** *auto*

lemma *finite-kraus-subadv-rewrite*:
finite-kraus-subadv $\mathfrak{E}$ (*Suc n*) =
(λ(*x*,*f*). *fun-upd f* (*Suc n*) *x*) ‘ (*kf-elems* ($\mathfrak{E}$ (*Suc n*)) × *finite-kraus-subadv* $\mathfrak{E}$ *n*)
by (*metis PiE-insert-eq Suc-eq-plus1 finite-kraus-subadv-def set-upt-Suc*)

lemma *finite-kraus-subadv-rewrite-inj*:
inj-on (λ(*x*, *f*). *f*(*Suc n* := *x*)) (*kf-elems* ($\mathfrak{E}$ (*Suc n*)) × *finite-kraus-subadv* $\mathfrak{E}$ *n*)
unfolding *inj-on-def* **proof** (*safe*, *goal-cases*)
case (*1 a b aa ba*) **then show** *?case* **by** (*metis fun-upd-eqD*)
next
case (*2 a b aa b'*)
then have *b x* = *b' x* **if** *x<Suc n* **for** *x*
by (*metis fun-upd-eqD fun-upd-triv fun-upd-twist nat-neq-iff that*)
moreover have *b x* = *undefined* **and** *b' x* = *undefined* **if** *x* ≥ *Suc n* **for** *x*
using *2*(*2*,*4*) **unfolding** *finite-kraus-subadv-def*
by (*metis PiE-arb Suc-eq-plus1 atLeastLessThan-iff not-le that*)+
ultimately show *?case* **by** (*intro ext*) (*metis not-le*)
qed

lemma *norm-kf-apply-singleton-trace-tc*:
assumes *0* ≤ ϱ **and** ‹*fst x* ≠ *0*›
shows *norm* (*kf-apply* (*Abs-kraus-family* {*x*}) ϱ) = *trace-tc* (*sandwich-tc* (*fst x*) ϱ)
apply (*subst norm-tc-pos*)
apply (*rule kf-apply-pos*[*OF assms*(*1*)])
using *kf-apply-singleton*
apply (*subst kf-apply-singleton*)
using *assms* **by** *auto*

lemma *infsum-norm-kf-apply-step*:
assumes *ϱn-summable*: *ϱn summable-on finite-kraus-subadv* $\mathfrak{E}$ *n*
and *pos*: ⋀*x*. *x* ∈ *finite-kraus-subadv* $\mathfrak{E}$ *n* ⟹ *0* ≤ *ϱn x*
shows (λ*x*. $\sum_\infty$*y*∈*finite-kraus-subadv* $\mathfrak{E}$ *n*. *norm* (*kf-apply x* (*ϱn y*)))
abs-summable-on kf-elems ($\mathfrak{E}$ (*Suc n*))
proof −

define ϱ **where** ϱ = *infsum* ϱ*n* (*finite-kraus-subadv* $\mathfrak{E}$ *n*)
have ((λ*y. trace-tc* (*sandwich-tc E y*)) *o* (λ*y.* ϱ*n y*) *has-sum trace-tc* (*sandwich-tc E* ϱ))
(*finite-kraus-subadv* $\mathfrak{E}$ *n*) **for** *E*::'*a ell2* $\Rightarrow_{CL}$ '*a ell2*
unfolding *o-def* **by** (*subst has-sum-bounded-linear*[*OF bounded-linear-trace-norm-sandwich-tc*])
(*auto simp add*: ϱ*-def* ϱ*n-summable*)
then have *sandwich-tc-lim*: ($\sum_\infty$*y*∈*finite-kraus-subadv* $\mathfrak{E}$ *n. trace-tc* (*sandwich-tc E* (ϱ*n y*)))
=
trace-tc (*sandwich-tc E* ($\sum_\infty$*y*∈*finite-kraus-subadv* $\mathfrak{E}$ *n.* ϱ*n y*))
for *E*::'*a ell2* $\Rightarrow_{CL}$ '*a ell2*
by (*intro infsumI*) (*auto simp add*: *o-def* ϱ*-def*)

let *?f1* = (λ(*E*,*x*). |$\sum_\infty$*y*∈*finite-kraus-subadv* $\mathfrak{E}$ *n. trace-tc* (*sandwich-tc E* (ϱ*n y*))|)
let *?f2* = (λ*x.* |$\sum_\infty$*y*∈*finite-kraus-subadv* $\mathfrak{E}$ *n. norm* (*kf-apply* (*Abs-kraus-family* {*x*}) (ϱ*n*
y))|)

have (λ(*E*, *x*). *sandwich-tc E* ϱ) *abs-summable-on Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*))
using *Rep-kraus-family kf-apply-abs-summable* **by** *blast*
then have *f1-summable*: *?f1 summable-on Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*))
unfolding *sandwich-tc-lim* ϱ*-def*[*symmetric*] **using** *trace-tc-abs-summable-on o-def*
by (*metis* (*mono-tags*, *lifting*) *abs-summable-summable norm-abs split-def summable-on-cong*)

then have *?f2 summable-on Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*))
proof −
have (*?f1 summable-on Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*))) = (*?f2 summable-on Rep-kraus-family*
($\mathfrak{E}$ (*Suc n*)))
proof (*subst summable-on-cong*[*of Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*)) *?f1 ?f2*], *goal-cases*)
case (*1 x*)
then have *neq0*: ‹*fst x* ≠ *0*›
using *Rep-kraus-family*
by (*force simp*: *kraus-family-def*)
have *infsum*: ($\sum_\infty$*y*∈*finite-kraus-subadv* $\mathfrak{E}$ *n. trace-tc* (*sandwich-tc* (*fst x*) (ϱ*n y*))) =
($\sum_\infty$*y*∈*finite-kraus-subadv* $\mathfrak{E}$ *n. norm* (*kf-apply* (*Abs-kraus-family* {*x*}) (ϱ*n y*)))
apply (*subst infsum-of-real*[*symmetric*])
apply (*rule infsum-cong*)
apply (*subst norm-kf-apply-singleton-trace-tc*)
using *pos neq0* **by** *auto*
then show *?case* **by** (*auto simp add*: *split-def abs-complex-def*)
next
case *2*
then show *?case* **using** *summable-on-iff-abs-summable-on-complex* **by** *force*
qed
then show *?thesis* **using** *f1-summable* **by** *auto*
qed
then show *?thesis* **unfolding** *kf-elems-Rep-kraus-family*
by (*subst summable-on-reindex*[*OF inj-on-kf-singleton*])
(*use kf-apply-singleton* **in** ‹*auto simp add*: *o-def*›)
qed

Run of adversary is summable on sub-adversaries.

lemma *run-mixed-adv-greater-indifferent*:
  assumes $m > n$
  shows *run-mixed-adv n* $(f(m := x))$ *UB init X Y H* = *run-mixed-adv n f UB init X Y H*
  using *assms* **by** (*induct n arbitrary*: *f m*) *auto*

lemma *run-mixed-adv-Suc-indifferent*:
  run-mixed-adv n $(f(Suc\ n := x))$ *UB init X Y H* = *run-mixed-adv n f UB init X Y H*
  by (*intro run-mixed-adv-greater-indifferent*) *auto*

lemma *run-mixed-adv-abs-summable*:
  fixes $\mathfrak{E}$ :: *'a kraus-adv*
  shows (λf. *run-mixed-adv n f UB init X Y H*) *abs-summable-on* (*finite-kraus-subadv* $\mathfrak{E}$ *n*)
proof (*induct n*)
  case *0*
  have *inj-on* (λf. *f 0*) (Π_E $i \in \{0\}$. *kf-elems* ($\mathfrak{E}$ *i*))
    unfolding *PiE-over-singleton-iff inj-on-def* **by** *auto*
  then have *inj*: *inj-on* (λf. *f 0*) (*finite-kraus-subadv* $\mathfrak{E}$ *0*)
    unfolding *finite-kraus-subadv-def* **by** *simp*
  have ($\lambda\mathfrak{F}$. *kf-apply* $\mathfrak{F}$ (*tc-selfbutter init*)) *abs-summable-on*
    (*kf-elems* ($\mathfrak{E}$ *0*)) **using** *kf-apply-abs-summable-on-kf-elems* **by** *auto*
  moreover {
    have $x \in$ *kf-elems* ($\mathfrak{E}$ *0*) $\Longrightarrow$
        $x \in$ (λx. *x 0*) ‘ (Π_E $i \in \{0\}$. *kf-elems* ($\mathfrak{E}$ *i*)) **for** *x*
      unfolding *PiE-over-singleton-iff* **by** (*simp add*: *image-iff*)
    then have (λf. *f 0*) ‘ *finite-kraus-subadv* $\mathfrak{E}$ *0* = *kf-elems* ($\mathfrak{E}$ *0*)
      by (*auto simp add*: *finite-kraus-subadv-I finite-kraus-subadv-def*)
  }
  ultimately have ($\lambda\mathfrak{F}$. *kf-apply* $\mathfrak{F}$ (*tc-selfbutter init*)) *o* (λf. *f 0*) *abs-summable-on* (*finite-kraus-subadv*
$\mathfrak{E}$ *0*)
    by (*subst abs-summable-on-reindex*[*OF inj*, *symmetric*]) *auto*
  then show *?case* **by** *auto*
next
  case (*Suc n*)
  define ϱn **where** $\varrho n\ f$ = *sandwich-tc* ((*X*;*Y*) (*Uquery H*) o_{CL} *UB n*)(*run-mixed-adv n f UB*
init X Y H)
    for *f*
  have ϱn*-Suc-indiff*:ϱn $(f(Suc\ n := x))$ = $\varrho n\ f$ **for** *f x*
    unfolding ϱn*-def run-mixed-adv-Suc-indifferent* **by** *auto*
  have ϱn*-abs-summable-on*:
    (λf. $\varrho n\ f$) *abs-summable-on finite-kraus-subadv* $\mathfrak{E}$ *n*
    unfolding ϱn*-def* **using** *sandwich-tc-abs-summable-on*[*OF Suc*] **by** (*auto simp add*: *o-def*)

  have *one*: (λxa. *kf-apply x* (ϱn $(xa(Suc\ n := x))$)) *abs-summable-on finite-kraus-subadv* $\mathfrak{E}$ *n*
    if $x \in$ *kf-elems* ($\mathfrak{E}$ (*Suc n*)) **for** *x*
    using ϱn*-Suc-indiff* ϱn*-abs-summable-on finite-kf-apply-abs-summable-on* **by** *fastforce*

  have ϱn*-summable*: ϱn *summable-on finite-kraus-subadv* $\mathfrak{E}$ *n*
    using *Suc* ϱn*-def* ϱn*-abs-summable-on abs-summable-summable* **by** *blast*

have *pos*: $x \in$ *finite-kraus-subadv* $\mathfrak{E}$ $n \Longrightarrow 0 \leq \varrho n\ x$ **for** x
  by (*simp add*: *ϱn-def run-mixed-adv-pos sandwich-tc-pos*)
have *two*: ($\lambda x.\ \sum_\infty y{\in}$*finite-kraus-subadv* $\mathfrak{E}$ n. *norm* (*kf-apply* x (ϱn (y(*Suc* n := x)))))
  abs-summable-on kf-elems ($\mathfrak{E}$ (*Suc* n))
  unfolding *ϱn-Suc-indiff* **by** (*rule infsum-norm-kf-apply-step*[*OF ϱn-summable pos*])

have *lim*: (λx. *kf-apply* (x (*Suc* n)) ($\varrho n\ x$)) *abs-summable-on finite-kraus-subadv* $\mathfrak{E}$ (*Suc* n)
  apply (*subst finite-kraus-subadv-rewrite*)
  apply (*subst abs-summable-on-reindex*[*OF finite-kraus-subadv-rewrite-inj*])
  apply (*unfold o-def case-prod-beta*)
  apply (*subst abs-summable-on-Sigma-iff*)
  using *one two* **by** *auto*
then have (λf. *kf-apply* (f (*Suc* n)) (*sandwich-tc* (($X;Y$) (*Uquery* H) o_{CL} *UB* n)
 (*run-mixed-adv* n f *UB init* X Y H))) *abs-summable-on*
 (*finite-kraus-subadv* $\mathfrak{E}$ (*Suc* n))
  unfolding *ϱn-def*[*symmetric*] **by** *auto*
then show *?case* **by** *auto*
qed

lemma *run-mixed-adv-summable*:
  fixes $\mathfrak{E}$ :: *'a kraus-adv*
  shows (λf. *run-mixed-adv* n f *UB init* X Y H) *summable-on* (*finite-kraus-subadv* $\mathfrak{E}$ n)
  using *abs-summable-summable*[*OF run-mixed-adv-abs-summable*] **by** *blast*

lemma *run-mixed-adv-has-sum*:
  fixes $\mathfrak{E}$ :: *'a kraus-adv*
  shows ((λf. *run-mixed-adv* n f *UB init* X Y H) *has-sum run-mixed-adv* n $\mathfrak{E}$ *UB init* X Y H)
  (*finite-kraus-subadv* $\mathfrak{E}$ n)
proof (*induct* n)
  case *0*
  have *inj-on* (λf. f *0*) (Π_E $i{\in}\{0\}$. *kf-elems* ($\mathfrak{E}$ i))
    unfolding *PiE-over-singleton-iff inj-on-def* **by** *auto*
  then have *inj*: *inj-on* (λf. f *0*) (*finite-kraus-subadv* $\mathfrak{E}$ *0*)
    unfolding *finite-kraus-subadv-def* **by** *simp*
  have *rew*: (λf. *kf-apply* (f *0*) (*tc-selfbutter init*)) =
    ($\lambda\mathfrak{F}$. *kf-apply* $\mathfrak{F}$ (*tc-selfbutter init*)) *o* (λf. f *0*) **by** *auto*
  have (($\lambda\mathfrak{F}$. *kf-apply* $\mathfrak{F}$ (*tc-selfbutter init*)) *has-sum*
     kf-apply ($\mathfrak{E}$ *0*) (*tc-selfbutter init*))
     (*kf-elems* ($\mathfrak{E}$ *0*)) **using** *kf-apply-has-sum-kf-elems* **by** *auto*
  moreover {
    have $x \in$ *kf-elems* ($\mathfrak{E}$ *0*) $\Longrightarrow$
        $x \in$ (λx. x *0*) ` (Π_E $i{\in}\{0\}$. *kf-elems* ($\mathfrak{E}$ i)) **for** x
      unfolding *PiE-over-singleton-iff* **by** (*simp add*: *image-iff*)
    then have (λf. f *0*) ` *finite-kraus-subadv* $\mathfrak{E}$ *0* = *kf-elems* ($\mathfrak{E}$ *0*)
      by (*auto simp add*: *finite-kraus-subadv-I finite-kraus-subadv-def*)
  }
  ultimately have ((λf. *kf-apply* (f *0*) (*tc-selfbutter init*)) *has-sum*
     kf-apply ($\mathfrak{E}$ *0*) (*tc-selfbutter init*)) (*finite-kraus-subadv* $\mathfrak{E}$ *0*)

unfolding *rew* **by** (*subst has-sum-reindex*[*OF inj*, *symmetric*]) *auto*
then show *?case* **by** *auto*
next
case (*Suc n*)
define ϱn **where** $\varrho n\ f$ = *sandwich-tc* ((*X*;*Y*) (*Uquery H*) o_{CL} *UB n*)(*run-mixed-adv n f UB init X Y H*)
for *f*
have ϱ*n-Suc-indiff*:ϱn (*f*(*Suc n* := *x*)) = ϱn *f* **for** *f x*
unfolding ϱ*n-def run-mixed-adv-Suc-indifferent* **by** *auto*

define ϱ **where** ϱ = *sandwich-tc* ((*X*;*Y*) (*Uquery H*) o_{CL} *UB n*) (*run-mixed-adv n* $\mathfrak{E}$ *UB init X Y H*)

have ϱ*n-has-sum-*ϱ: ((λf. ϱn *f*) *has-sum* ϱ) (*finite-kraus-subadv* $\mathfrak{E}$ *n*)
unfolding ϱ*n-def* ϱ*-def* **by** (*use sandwich-tc-has-sum*[*OF Suc*] **in** ‹*auto simp add*: *o-def*›)

have ϱ*n-abs-summable-on*:
(λf. ϱn *f*) *abs-summable-on finite-kraus-subadv* $\mathfrak{E}$ *n*
proof −
have ∀ *f c F*. (λ*fa*. *sandwich-tc c* (*f fa*)) *abs-summable-on F* ∨ ¬ *f abs-summable-on F*
using *sandwich-tc-abs-summable-on* **by** *auto*
then show *?thesis*
unfolding ϱ*n-def* **by** (*metis* (*no-types*) *run-mixed-adv-abs-summable*)
qed

have *one*: ((λy. *kf-apply x* (ϱn (*y*(*Suc n* := *x*)))) *has-sum kf-apply x* ϱ)
(*finite-kraus-subadv* $\mathfrak{E}$ *n*) **if** *x* ∈ *kf-elems* ($\mathfrak{E}$ (*Suc n*)) **for** *x*
unfolding ϱ*n-Suc-indiff* **by** (*smt* (*verit*, *best*) ϱ*n-has-sum-*ϱ *comp-eq-dest-lhs*
finite-kf-apply-has-sum has-sum-cong)
have *two*: ((λx. *kf-apply x* ϱ) *has-sum kf-apply* ($\mathfrak{E}$ (*Suc n*)) ϱ)
(*kf-elems* ($\mathfrak{E}$ (*Suc n*)))
by (*simp add*: *kf-apply-has-sum-kf-elems*)

have (λ(*x*,*f*). *kf-apply x* (ϱn (*f*(*Suc n* := *x*)))) *abs-summable-on*
kf-elems ($\mathfrak{E}$ (*Suc n*)) × *finite-kraus-subadv* $\mathfrak{E}$ *n*
proof (*unfold* ϱ*n-Suc-indiff*, *subst abs-summable-on-Sigma-iff*, *safe*, *goal-cases*)
case (*1 x*)
then show *?case* **using** ϱ*n-abs-summable-on finite-kf-apply-abs-summable-on* **by** *auto*
next
case *2*
then show *?case*
by (*intro infsum-norm-kf-apply-step*[*OF abs-summable-summable*[*OF* ϱ*n-abs-summable-on*]])

(*auto simp add*: ϱ*n-def run-mixed-adv-pos sandwich-tc-pos*)
qed
then have (λ(*x*,*f*). *kf-apply x* (ϱn (*f*(*Suc n* := *x*)))) *summable-on*
kf-elems ($\mathfrak{E}$ (*Suc n*)) × *finite-kraus-subadv* $\mathfrak{E}$ *n*
using *abs-summable-summable* **by** *blast*
then have *three*: (λx. *kf-apply* (*fst x*) (ϱn ((*snd x*)(*Suc n* := *fst x*)))) *summable-on*

kf-elems ($\mathfrak{E}$ (*Suc n*)) × *finite-kraus-subadv* $\mathfrak{E}$ *n*
by (*metis* (*no-types*, *lifting*) *split-def summable-on-cong*)

have *lim*:
((λ*f*. *kf-apply* (*f* (*Suc n*)) (ϱ*n f*)) *has-sum kf-apply* ($\mathfrak{E}$ (*Suc n*)) ϱ)
(*finite-kraus-subadv* $\mathfrak{E}$ (*Suc n*))
apply (*subst finite-kraus-subadv-rewrite*)
apply (*subst has-sum-reindex*[*OF finite-kraus-subadv-rewrite-inj*])
apply (*unfold o-def case-prod-beta*)
apply (*intro has-sum-SigmaI*[**where** *g* = (λ*x*. *kf-apply x* ϱ)])
by (*auto simp add*: *one two three*)
then have ((λ*f*. *kf-apply* (*f* (*Suc n*)) (*sandwich-tc* ((X;Y) (*Uquery H*) o_{CL} *UB n*)
(*run-mixed-adv n f UB init X Y H*))) *has-sum kf-apply* ($\mathfrak{E}$ (*Suc n*))
(*sandwich-tc* ((X;Y) (*Uquery H*) o_{CL} *UB n*) (*run-mixed-adv n* $\mathfrak{E}$ *UB init X Y H*)))
(*finite-kraus-subadv* $\mathfrak{E}$ (*Suc n*))
unfolding ϱ*n-def* ϱ*-def* **by** *auto*
then show *?case* **by** *auto*
qed

Now, we cover limits for adversary runs in the O2H setting.

context *o2h-setting*
begin

lemma *run-mixed-A-has-sum*:
((λ*f*. *run-mixed-A f H*) *has-sum run-mixed-A kraus-A H*) (*finite-kraus-subadv kraus-A d*)
unfolding *run-mixed-A-def* **by** (*rule run-mixed-adv-has-sum*)

lemma *run-mixed-B-has-sum*:
((λ*f*. *run-mixed-adv d f* (*US S*) *init-B X-for-B Y-for-B H*) *has-sum run-mixed-B kraus-B H S*)
(*finite-kraus-subadv* (λ*n*. *kf-Fst* (*kraus-B n*)) *d*)
unfolding *run-mixed-B-def* **by** (*rule run-mixed-adv-has-sum*)

lemma *run-mixed-B-count-has-sum*:
((λ*f*. *run-mixed-adv d f* (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*) *has-sum run-mixed-B-count kraus-B H S*)
(*finite-kraus-subadv* (λ*n*. *kf-Fst* (*kraus-B n*)) *d*)
unfolding *run-mixed-B-count-def* **by** (*rule run-mixed-adv-has-sum*)

lemma *kf-elems-kf-Fst*:
kf-elems (*kf-Fst* $\mathfrak{E}$) = (λ*f*. *kf-Fst f*) ‘ *kf-elems* $\mathfrak{E}$
by *transfer auto*

lemma *finite-kraus-subadv-Fst-invert*:
finite-kraus-subadv (λ*m*. (*kf-Fst* :: -⇒((′*a* × ′*c*) *ell2*,-,-) *kraus-family*) ($\mathfrak{E}$ *m*)) *n* =

```
 (λf. λi∈{0..<n+1}. kf-Fst (f i)) ` (finite-kraus-subadv 𝔈 n)
  unfolding finite-kraus-subadv-def kf-elems-kf-Fst
proof (induct n)
  case 0
  have (Π_E i∈{0..<0 + 1}. kf-Fst ` kf-elems (𝔈 i)) =
    (Π_E i∈{0}. kf-Fst ` kf-elems (𝔈 i)) by auto
  also have ... = (⋃b∈kf-elems (𝔈 0). {λx∈{0}. kf-Fst b})
    unfolding PiE-over-singleton-iff by auto
  also have ... = (⋃b∈kf-elems (𝔈 0). (λf. λi∈{0}. kf-Fst (f i)) ` {λx∈{0}. b})
  proof −
    have (λx∈{0}. kf-Fst b) = (λa∈{0}. kf-Fst (if a = 0 then b else undefined)) for b
      by fastforce
    then show ?thesis by (intro Union-cong) auto
  qed
  also have ... = (λf. λi∈{0}. kf-Fst (f i)) `(⋃b∈kf-elems (𝔈 0). {λx∈{0}. b})
    unfolding image-UN by auto
  also have ... = (λf. λi∈{0..<0+1}. kf-Fst (f i)) ` (Π_E i∈{0}. kf-elems (𝔈 i))
    unfolding PiE-over-singleton-iff by auto
  also have ... = (λf. λi∈{0..<0+1}. kf-Fst (f i)) ` (Π_E i∈{0..<0+1}. kf-elems (𝔈 i))
    by auto
  finally show ?case by blast
next
  case (Suc n)
  let ?prodset = kf-elems (𝔈 (Suc n)) ×(Π_E i∈{0..<n+1}. kf-elems (𝔈 i))
  have (Π_E i∈{0..<Suc n + 1}. (kf-Fst :: -⇒(('a × 'c) ell2,-,-) kraus-family) `
    kf-elems (𝔈 i)) =
    (Π_E i∈(insert (Suc n) {0..<Suc n}). kf-Fst ` kf-elems (𝔈 i))
    by (auto simp add: set-upt-Suc)
  also have ... = (λ(y, g). g(Suc n := y)) ` (kf-Fst ` kf-elems (𝔈 (Suc n)) ×
     (Π_E i∈{0..<n+1}. kf-Fst ` kf-elems (𝔈 i)))
    by (subst PiE-insert-eq) auto
  also have ... = (λ(y, g). g(Suc n := y)) ` (kf-Fst ` kf-elems (𝔈 (Suc n)) ×
     ((λf. λi∈{0..<n+1}. kf-Fst (f i)) ` (Π_E i∈{0..<n+1}. kf-elems (𝔈 i))))
    by (subst Suc) auto
  also have ... = (λ(y, g). g(Suc n := y)) `
    (λ(a,x). (kf-Fst a, restrict (λi. kf-Fst (x i)) {0..<n+1})) ` ?prodset
    by (simp add: image-paired-Times)
  also have ... = (λ(y, g). (restrict (λi. kf-Fst (g i)) {0..<n+1})
    (Suc n := kf-Fst y)) ` ?prodset
    by (subst image-image) (simp add: split-def)
  also have ... = (λ(y, g). restrict ((λi. kf-Fst (g i))(Suc n := kf-Fst y))
    (insert (Suc n) {0..<n+1})) ` ?prodset
    by (subst restrict-upd) auto
  also have ... = (λ(y, g). restrict ((λi. kf-Fst (g i))(Suc n := kf-Fst y))
    {0..<Suc n + 1}) ` ?prodset using semiring-norm(174) set-upt-Suc by presburger
  also have ... = (λ(y, g). restrict (λi. kf-Fst ((g(Suc n:=y)) i))
    {0..<Suc n + 1}) ` ?prodset
  proof −
    have rew: (λi. kf-Fst (g i))(Suc n := kf-Fst y) =
```

```
    (λi. (kf-Fst :: -⇒(('a × 'c) ell2,-,-) kraus-family) ((g(Suc n:=y)) i)) for g y
      by fastforce
    show ?thesis by (subst rew) auto
  qed
  also have ... = (λf. restrict (λi. kf-Fst (f i)) {0..<Suc n + 1}) `
    (λ(a,g). g(Suc n:= a)) ` ?prodset
    by (smt (verit, best) image-cong image-image restrict-ext split-def)
  also have ... = (λf. restrict (λi. kf-Fst (f i)) {0..<Suc n + 1}) `
    (Π_E i∈(insert (Suc n) {0..<Suc n}). kf-elems (𝔈 i))
    by (metis Suc-eq-plus1 finite-kraus-subadv-def finite-kraus-subadv-rewrite set-upt-Suc)
  also have ... = (λf. λi∈{0..<Suc n + 1}. kf-Fst (f i)) `
    (Π_E i∈{0..<Suc n+1}. kf-elems (𝔈 i)) by (simp add: set-upt-Suc)
  finally show ?case by blast
qed

lemma inj-kf-Fst: ‹E = F› if ‹kf-Fst E = kf-Fst F›
proof (insert that, transfer)
  fix E F :: ‹('a ell2 ⇒CL 'c ell2 × unit) set›
  assume asm: ‹(λ(x, -). (x ⊗o id-cblinfun, ())) ` E = (λ(x, -). (x ⊗o id-cblinfun, ())) ` F›
  have ‹inj (λ(x::'a ell2 ⇒CL 'c ell2, -::unit). (x ⊗o id-cblinfun, ()))›
   apply (rewrite at ‹inj ⌑› to ‹map-prod (λx. x ⊗o id-cblinfun) id› DEADID.rel-mono-strong)
     apply (force intro!: simp:)
    apply (rule prod.inj-map)
    by (simp-all add: inj-tensor-left)
  from inj-image-eq-iff[OF this] asm
  show ‹E = F›
    by blast
qed


lemma inj-on-kf-Fst:
  inj-on (λf. λn∈{0..<n+1}. (kf-Fst (f n) :: (('a × 'b) ell2, -, -) kraus-family))
  (finite-kraus-subadv 𝔈 n)
proof (rule inj-onI, rename-tac E F)
  fix E F :: ‹nat ⇒ ('a ell2, 'a ell2, unit) kraus-family›
  assume finE: ‹E ∈ finite-kraus-subadv 𝔈 n› and finF: ‹F ∈ finite-kraus-subadv 𝔈 n›
 assume eq: ‹(λi∈{0..<n + 1}. kf-Fst (E i)  :: (('a × 'b) ell2, -, -) kraus-family) = (λn∈{0..<n
+ 1}. kf-Fst (F n))›
  have ‹(kf-Fst (E i) :: (('a × 'b) ell2, -, -) kraus-family) = kf-Fst (F i)› if ‹i ∈ {0..<n+1}›
for i
    using eq[unfolded fun-eq-iff, rule-format, of i]
    unfolding restrict-def
    using that by (auto intro!: simp: fun-eq-iff)
  then have ‹E i = F i› if ‹i ∈ {0..<n+1}› for i
    using inj-kf-Fst that by blast
  moreover from finE finF
  have ‹E i = F i› if ‹i ∉ {0..<n+1}› for i
    using that
    by (simp add: finite-kraus-subadv-def PiE-def extensional-def)
```

ultimately **show** ‹$E = F$›
  by *blast*
qed

lemma *run-mixed-adv-kf-Fst-restricted*:
  run-mixed-adv m (λn. *kf-Fst* (*f n*)) *U init′ X′ Y′ H* =
  run-mixed-adv m ($\lambda n \in \{0..<m + 1\}$. *kf-Fst* (*f n*)) *U init′ X′ Y′ H*
proof (*induct m arbitrary*: *f*)
  case (*Suc m*)
  let *?f1* = (λa. *if a* < *Suc m then* (*kf-Fst* ::-⇒((′*a* × ′*b*) *ell2*,-,-) *kraus-family*) (*f a*)
    else undefined)
  let *?f2* = (λa. *if a* < *Suc* (*Suc m*) *then* (*kf-Fst* ::-⇒((′*a* × ′*b*) *ell2*,-,-) *kraus-family*) (*f a*)
    else undefined)
  have *f12*: *?f1*(*Suc m*:= *kf-Fst* (*f* (*Suc m*))) = *?f2* **by** *fastforce*
  have *eq*: *run-mixed-adv m ?f1 U init′ X′ Y′ H* =
      run-mixed-adv m ?f2 U init′ X′ Y′ H
    unfolding *f12*[*symmetric*] **by** (*subst run-mixed-adv-Suc-indifferent*) *auto*
  show *?case* **by** (*auto simp add*: *eq Suc*)
qed *auto*

lemma *run-mixed-B-has-sum′*:
  ((λf. *run-mixed-B f H S*) *has-sum run-mixed-B kraus-B H S*) (*finite-kraus-subadv kraus-B d*)
  (**is** (*?f has-sum ?x*) *?A*)
proof −
  have *inj*: *inj-on* (λf. $\lambda n \in \{0..<d + 1\}$. *kf-Fst* (*f n*)) (*finite-kraus-subadv kraus-B d*)
    using *inj-on-kf-Fst* **by** *auto*
  have *rew*: *?f* = (λf. *run-mixed-adv d f* (*US S*) *init-B X-for-B Y-for-B H*) *o*
    (λf. $\lambda n \in \{0..<d + 1\}$. *kf-Fst* (*f n*)) **unfolding** *run-mixed-B-def*
   using *run-mixed-adv-kf-Fst-restricted*[**where** *init′* = *init-B* **and** *X′* = *X-for-B* **and** *Y′*=*Y-for-B*]

    by *auto*
  show *?thesis* **unfolding** *rew* **by** (*subst has-sum-reindex*[*OF inj*, *symmetric*])
    (*unfold finite-kraus-subadv-Fst-invert*[*symmetric*], *rule run-mixed-B-has-sum*)
qed

lemma *run-mixed-B-count-has-sum′*:
  ((λf. *run-mixed-B-count f H S*) *has-sum run-mixed-B-count kraus-B H S*) (*finite-kraus-subadv kraus-B d*)
  (**is** (*?f has-sum ?x*) *?A*)
proof −
  have *inj*: *inj-on* (λf. $\lambda n \in \{0..<d + 1\}$. *kf-Fst* (*f n*)) (*finite-kraus-subadv kraus-B d*)
    using *inj-on-kf-Fst* **by** *auto*
  have *rew*: *?f* = (λf. *run-mixed-adv d f* (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*) *o*
    (λf. $\lambda n \in \{0..<d + 1\}$. *kf-Fst* (*f n*)) **unfolding** *run-mixed-B-count-def*
    using *run-mixed-adv-kf-Fst-restricted*[**where** *init′* = *init-B-count* **and** *X′* = *X-for-C* **and**

$Y' = Y\text{-}for\text{-}C$]
  by *auto*
 show *?thesis* **unfolding** *rew* **by** (*subst has-sum-reindex*[*OF inj*, *symmetric*])
   (*unfold finite-kraus-subadv-Fst-invert*[*symmetric*], *rule run-mixed-B-count-has-sum*)
qed

Limit with finite sums

lemma *has-sum-finite-sum*:
 fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c :: \{comm\text{-}monoid\text{-}add, topological\text{-}space, topological\text{-}comm\text{-}monoid\text{-}add\}$
 assumes $\bigwedge val.\ (f\ val\ has\text{-}sum\ g\ val)\ A$ $finite\ S$
 shows $((\lambda x.\ (\sum val \in S.\ f\ val\ x))\ has\text{-}sum\ (\sum val \in S.\ g\ val))\ A$
 using *assms*(*2*) **proof** (*induct S*)
 case *empty*
 show $((\lambda x.\ \sum val{\in}\{\}.\ f\ val\ x)\ has\text{-}sum\ sum\ g\ \{\})\ A$ **by** *auto*
next
 case (*insert x F*)
 show $((\lambda xa.\ \sum val{\in}insert\ x\ F.\ f\ val\ xa)\ has\text{-}sum\ sum\ g\ (insert\ x\ F))\ A$
  unfolding *sum.insert-remove*[*OF* ‹*finite F*›] **by** (*intro has-sum-add*[*of f x*])
   (*use assms insert* **in** ‹*auto*›)
qed

lemma *fin-subadv-fin-Rep-kraus-family*:
 assumes $F \in finite\text{-}kraus\text{-}subadv\ E\ n$ $i < n+1$ $n<d+1$
 shows *finite* (*Rep-kraus-family* (*F i*))
 using *assms* **unfolding** *finite-kraus-subadv-def* **using** *kf-elems-finite* **by** *fastforce*

lemma *fin-subadv-bound-leq-id*:
 assumes $F \in finite\text{-}kraus\text{-}subadv\ E\ d$
 assumes $i < d+1$
 assumes *E-norm-id*: $\bigwedge i.\ i < d+1 \Longrightarrow kf\text{-}bound\ (E\ i) \le id\text{-}cblinfun$
 shows $kf\text{-}bound\ (F\ i) \le id\text{-}cblinfun$
proof −
 have $F\ i \in kf\text{-}elems\ (E\ i)$ **using** *assms* **unfolding** *finite-kraus-subadv-def* **by** *auto*
 then have $kf\text{-}bound\ (F\ i) \le kf\text{-}bound\ (E\ i)$
  using *kf-bound-of-elems* **by** *auto*
 then show *?thesis* **using** *E-norm-id*[*OF assms*(*2*)] **by** *auto*
qed

lemma *fin-subadv-nonzero*:
 assumes $F \in finite\text{-}kraus\text{-}subadv\ E\ n$ $i < n+1$ $n<d+1$
 shows $Rep\text{-}kraus\text{-}family\ (F\ i) \neq \{\}$
proof −
 have $F\ i \in kf\text{-}elems\ (E\ i)$ **using** *assms* **unfolding** *finite-kraus-subadv-def* **by** *auto*
 then show *?thesis* **using** *kf-elems-card-1* **by** *fastforce*
qed

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*
unbundle *no register-syntax*

end
theory *Purification*

imports *Run-Adversary*

begin
context *o2h-setting*
begin

unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
unbundle *register-syntax*

8 Purification of the Adversary

Purification of composed kraus maps.

definition *purify-comp-kraus* ::
  $nat \Rightarrow (nat \Rightarrow ('a::chilbert\text{-}space, 'b::chilbert\text{-}space, 'c)\ kraus\text{-}family) \Rightarrow (nat \Rightarrow 'a \Rightarrow_{CL} 'b)$
set **where**
  $purify\text{-}comp\text{-}kraus\ n\ \mathfrak{E} = PiE\ \{0..<n+1\}\ (\lambda i.\ (fst\ `\ (Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ i))))$

definition *comp-upto* :: $(nat \Rightarrow ('a::chilbert\text{-}space) \Rightarrow_{CL} 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow_{CL} 'a$ **where**
  $comp\text{-}upto\ f\ n = fold\ (\lambda i\ x.\ f\ i\ o_{CL}\ x)\ [0..<n+1]\ id\text{-}cblinfun$

Some auxiliary lemmas on injectivity, Fst and finiteness.

lemma *Rep-kf-id*:
  $Rep\text{-}kraus\text{-}family\ kf\text{-}id = \{(id\text{-}cblinfun :: 'a \Rightarrow_{CL} 'a::\{chilbert\text{-}space,not\text{-}singleton\},())\}$
  by (*simp add*: *kf-id-def kf-of-op.rep-eq del*: *kf-of-op-id*)

lemma *fst-Rep-kf-Fst*:
  fixes $\mathfrak{E}$ :: ('*a ell2*, '*b ell2*, *unit*) *kraus-family*
  shows $fst\ `\ (Rep\text{-}kraus\text{-}family\ (kf\text{-}Fst\ \mathfrak{E})) = Fst\ `\ (fst\ `\ (Rep\text{-}kraus\text{-}family\ \mathfrak{E}))$
proof (*transfer*, *safe*, *goal-cases*)
  case (*1* $\mathfrak{E}$ *x a aa*)
  then have $aa \in fst\ `\mathfrak{E}$ **by** (*metis fst-conv image-eqI*)
  then show *?case* **by** (*auto simp add*: *Fst-def*)
next
  case (*2* $\mathfrak{E}$ *x xa a*)
  then show *?case* **by** (*auto simp add*: *Fst-def image-comp*)
qed

lemma *inj-on-Fst*:
  shows *inj-on Fst A*
  unfolding *Fst-def inj-on-def* **using** *inj-tensor-left*[*OF id-cblinfun-not-0*] **unfolding** *inj-def*
by *auto*

lemma *finite-kf-Fst*:
  fixes $\mathfrak{E}$ :: (*'mem ell2*, *'mem ell2*, *unit*) *kraus-family*
  assumes *finite* (*Rep-kraus-family* $\mathfrak{E}$)
  shows *finite* (*Rep-kraus-family* (*kf-Fst* $\mathfrak{E}$))
  using *assms* **by** *transfer auto*

lemma *finite-kf-id*:
  finite (*Rep-kraus-family kf-id*)
  by (*simp add*: *kf-of-op.rep-eq flip*: *kf-of-op-id*)

lemma *inj-on-fst-Rep-kraus-family*:
  fixes $\mathfrak{E}$ :: (*'a ell2*,*'b ell2*,*unit*) *kraus-family*
  shows *inj-on fst* (*Rep-kraus-family* $\mathfrak{E}$)
  unfolding *inj-on-def* **by** *fastforce*

lemma *comp-kraus-maps-set-finite*:
  assumes $\bigwedge i.\ i<n+1 \Longrightarrow$ *finite* (*Rep-kraus-family* ($\mathfrak{E}$ *i*))
  shows *finite* (*purify-comp-kraus n* $\mathfrak{E}$)
  unfolding *purify-comp-kraus-def* **by** (*intro finite-PiE*) (*auto simp add*: *assms*)

Showing conditions of Kraus maps.

lemma *norm-square-in-kraus-map*:
  fixes $\mathfrak{E}$ :: (*'a ell2*,*'a ell2*,*unit*) *kraus-family*
  assumes *kf-bound* $\mathfrak{E} \leq$ *id-cblinfun*
  assumes $U \in$ *fst* ‘ *Rep-kraus-family* $\mathfrak{E}$
  shows $U*\ o_{CL}\ U \leq$ *id-cblinfun*
proof −
  have $*$: $\{(U,\ ())\} \subseteq$ *Rep-kraus-family* $\mathfrak{E}$ **using** *assms*(*2*) **by** *auto*
  show *?thesis* **using** *kf-bound-geq-sum*[*OF* $*$] *assms*(*1*) **by** *auto*
qed

lemma *norm-in-kraus-map*:
  fixes $\mathfrak{E}$ :: (*'a ell2*,*'a ell2*,*unit*) *kraus-family*
  assumes *kf-bound* $\mathfrak{E} \leq$ *id-cblinfun*
  assumes $U \in$ *fst* ‘ *Rep-kraus-family* $\mathfrak{E}$
  shows *norm* $U \leq 1$
  using *norm-square-in-kraus-map*[*OF assms*] *cond-to-norm-1* **by** *auto*

lemma *purify-comp-kraus-in-kraus-family*:

assumes *UA* ∈ *purify-comp-kraus n* 𝔈 *j*<*n*+*1*
shows *UA j* ∈ *fst* ‘ *Rep-kraus-family* (𝔈 *j*)
proof (*intro PiE-mem*[*of UA* {*0*..<*n*+*1*}])
show *UA* ∈ (Π_E *a*∈{*0*..<*n*+*1*}. *fst* ‘ *Rep-kraus-family* (𝔈 *a*))
using *assms*(*1*) **unfolding** *purify-comp-kraus-def* **by** *auto*
show *j* ∈ {*0*..<*n*+*1*} **using** *assms*(*2*) **by** *auto*
qed

lemma *norm-in-purify-comp-kraus*:
fixes 𝔈 :: *nat* ⇒ (*'a ell2*, *'a ell2*, *unit*) *kraus-family*
assumes ⋀*i*. *i*<*n*+*1* ⟹ *kf-bound* (𝔈 *i*) ≤ *id-cblinfun*
assumes *UA* ∈ *purify-comp-kraus n* 𝔈
shows ⋀*i*. *i*<*n*+*1* ⟹ *norm* (*UA i*) ≤ *1*
proof −
fix *i* **assume** *i*: *i*<*n*+*1*
have *UA i* ∈ *fst* ‘ *Rep-kraus-family* (𝔈 *i*)
using *purify-comp-kraus-in-kraus-family*[*OF assms*(*2*) ‹*i*<*n*+*1*›] **by** *auto*
then show *norm* (*UA i*) ≤ *1* **using** *norm-in-kraus-map assms*(*1*) *i* **by** *auto*
qed

lemma *run-pure-adv-tc-over*:
assumes *m* > *n*
shows *run-pure-adv-tc n* (*UA*(*m* := *x*)) *UB init′ X′ Y′ H* = *run-pure-adv-tc n UA UB init′ X′ Y′ H*
using *assms* **by** (*induct n arbitrary*: *m x*) *auto*

lemma *run-pure-adv-tc-Fst-over*:
assumes *m* > *n*
shows *run-pure-adv-tc n* (*Fst o UA*(*m* := *x*)) *UB init′ X′ Y′ H* =
run-pure-adv-tc n (*Fst o UA*) *UB init′ X′ Y′ H*
using *assms* **by** (*induct n arbitrary*: *m x*) *auto*

Purifications of the adversarial runs.

lemma *purification-run-mixed-adv*:
assumes ⋀*i*. *i*<*n*+*1* ⟹ *finite* (*Rep-kraus-family* (𝔈 *i*))
assumes ⋀*i*. *i*<*n*+*1* ⟹ *fst* ‘ *Rep-kraus-family* (𝔈 *i*) ≠ {}
shows *run-mixed-adv n* 𝔈 *UB init′ X′ Y′ H* =
(∑ *UAs* ∈ *purify-comp-kraus n* 𝔈. *run-pure-adv-tc n UAs UB init′ X′ Y′ H*)
unfolding *purify-comp-kraus-def* **using** *assms*
proof (*induct n*)
case *0*
have *kf-apply* (𝔈 *0*) (*tc-selfbutter init′*) =
(∑*E*∈*fst* ‘ (*Rep-kraus-family* (𝔈 *0*)). *sandwich-tc E* (*tc-selfbutter init′*))
unfolding *kf-apply.rep-eq* **using** *assms*
by (*subst sum.reindex*) (*auto simp add*: *inj-on-fst-Rep-kraus-family d-gr-0*)
moreover have (∑ *UAs*∈(Π_E *i*∈{*0*}. *fst* ‘ *Rep-kraus-family* (𝔈 *i*)).
sandwich-tc (*UAs 0*) (*tc-selfbutter init′*)) =
(∑*E*∈*fst* ‘ (*Rep-kraus-family* (𝔈 *0*)). *sandwich-tc E* (*tc-selfbutter init′*))

(**is** *?left* = *?right*)
proof −
have *inj1*: *inj-on* (λ*UA*. *UA 0*) (Π_E *i*$\in$\{*0*\}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*))
by (*smt* (*verit*, *best*) *PiE-ext inj-on-def singletonD*)
have *non-empty*: (Π_E *i*$\in$\{*0*\}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)) $\neq$ \{\}
by (*metis* (*no-types*, *lifting*) *0*(*2*) *PiE-eq-empty-iff Suc-eq-plus1 singleton-iff zero-less-Suc*)
have *?left* = ($\sum$ *UA* $\in$ (λ*UA*. *UA 0*) ‘ (Π_E *i*$\in$\{*0*\}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)).
sandwich-tc UA (*tc-selfbutter init*$'$))
by (*subst sum.reindex*) (*auto simp add*: *inj1*)
also have ... = ($\sum$ *UA* $\in$ *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *0*).
sandwich-tc UA (*tc-selfbutter init*$'$))
by (*intro sum.cong*) (*auto simp add*: *image-projection-PiE non-empty*)
finally show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** *auto*
next
case (*Suc d*)
have *inj2*: *inj-on* (λ(*y*, *g*). *g*(*Suc d* := *y*))(*fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ (*Suc d*)) $\times$
(Π_E *i*$\in$\{*0*..<*Suc d*\}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)))
by (*metis atLeastLessThan-iff inj-combinator less-irrefl-nat*)
let *?*Φ = *sandwich-tc* ((*X*$'$;*Y*$'$) (*Uquery H*) o_{CL} *UB d*)(*run-mixed-adv d* $\mathfrak{E}$ *UB init*$'$ *X*$'$ *Y*$'$ *H*)
let *?*Ψ = (λ*UAs*. *run-pure-adv-tc d UAs UB init*$'$ *X*$'$ *Y*$'$ *H*)
have *kraus*: *kf-apply* ($\mathfrak{E}$ (*Suc d*)) *?*Φ =
($\sum$ *E*$\in$*fst* ‘ (*Rep-kraus-family* ($\mathfrak{E}$ (*Suc d*))). *sandwich-tc E ?*Φ)
unfolding *kf-apply.rep-eq* **using** *assms*
by (*subst sum.reindex*) (*auto simp add*: *inj-on-fst-Rep-kraus-family Suc*)
also have ... = ($\sum$ *UA*$\in$*fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ (*Suc d*)).
($\sum$ *UAs*$\in$(Π_E *i*$\in$\{*0*..<*Suc d*\}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)). *sandwich-tc UA*
(*sandwich-tc* ((*X*$'$;*Y*$'$) (*Uquery H*) o_{CL} *UB d*) (*?*Ψ *UAs*))))
using *Suc* **by** (*intro sum.cong*)(*auto simp add*: *sandwich-tc-sum*)
also have ... = ($\sum$ (*UA*,*UAs*)$\in$*fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ (*Suc d*))$\times$
(Π_E *i*$\in$\{*0*..<*Suc d*\}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)).
sandwich-tc UA (*sandwich-tc* ((*X*$'$;*Y*$'$) (*Uquery H*) o_{CL} *UB d*) (*?*Ψ *UAs*)))
by (*subst sum.cartesian-product*) *auto*
also have ... = ($\sum$ *UAs*$\in$(λ(*y*, *g*). *g*(*Suc d* := *y*)) ‘ (*fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ (*Suc d*)) $\times$
(Π_E *i*$\in$ \{*0*..<*Suc d*\}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*))).
sandwich-tc (*UAs* (*Suc d*) o_{CL} (*X*$'$;*Y*$'$) (*Uquery H*) o_{CL} *UB d*)(*?*Ψ *UAs*))
by (*subst sum.reindex*) (*auto intro*!: *sum.cong simp add*: *o-def sandwich-tc-compose*
run-pure-adv-tc-over inj2)
also have ... = ($\sum$ *UAs*$\in$(Π_E *i*$\in$*insert* (*Suc d*) \{*0*..<*Suc d*\}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)).
sandwich-tc (*UAs* (*Suc d*) o_{CL} (*X*$'$;*Y*$'$) (*Uquery H*) o_{CL} *UB d*) (*?*Ψ *UAs*))
by (*subst PiE-insert-eq*) *auto*
finally show *?case* **by** (*auto simp add*: *set-upt-Suc*)
qed

lemma *purification-run-mixed-A*:
assumes $\bigwedge$*i*. *i*<*d*+*1* $\Longrightarrow$ *finite* (*Rep-kraus-family* ($\mathfrak{E}$ *i*))
assumes $\bigwedge$*i*. *i*<*d*+*1* $\Longrightarrow$ *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*) $\neq$ \{\}

shows *run-mixed-A* $\mathfrak{E}$ *H* = ($\sum$ *UAs* ∈ *purify-comp-kraus d* $\mathfrak{E}$. *run-pure-A-tc UAs H*)
unfolding *run-mixed-A-def run-pure-A-tc-def* **using** *assms*
by (*auto intro*!: *purification-run-mixed-adv*)

lemma *purification-run-mixed-B*:
assumes $\bigwedge$*i. i<d+1* ⟹ *finite* (*Rep-kraus-family* ($\mathfrak{E}$ *i*))
assumes $\bigwedge$*i. i<d+1* ⟹ *fst ‘ Rep-kraus-family* ($\mathfrak{E}$ *i*) ≠ {}
shows *run-mixed-B* $\mathfrak{E}$ *H S* = ($\sum$ *UAs* ∈ *purify-comp-kraus d* $\mathfrak{E}$. *run-pure-B-tc UAs H S*)
unfolding *run-mixed-B-def run-pure-B-tc-def purify-comp-kraus-def*
using *assms*
proof (*induct d*)
case *0*
let *?*$\mathfrak{E}$*0* = *kf-Fst* ($\mathfrak{E}$ *0*)
have *finite*: *finite* (*Rep-kraus-family ?*$\mathfrak{E}$*0*)
using *finite-kf-Fst assms*(*1*) **by** *auto*
have *inj1*: *inj-on fst* (*Rep-kraus-family ?*$\mathfrak{E}$*0*)
by (*rule inj-on-fst-Rep-kraus-family*)
have *inj2*: *inj-on Fst* (*fst ‘ Rep-kraus-family* ($\mathfrak{E}$ *0*))
using *inj-on-Fst* **by** *auto*
have ∗: *kf-apply ?*$\mathfrak{E}$*0* (*tc-selfbutter init-B*) =
($\sum$*E*∈*fst ‘* (*Rep-kraus-family ?*$\mathfrak{E}$*0*). *sandwich-tc E* (*tc-selfbutter init-B*))
unfolding *kf-apply.rep-eq*
by (*subst sum.reindex*) (*auto simp add*: *infsum-finite*[*OF finite*] *inj1*)
have *kf-apply ?*$\mathfrak{E}$*0* (*tc-selfbutter init-B*) =
($\sum$*E*∈*fst ‘* (*Rep-kraus-family* ($\mathfrak{E}$ *0*)). *sandwich-tc* (*Fst E*) (*tc-selfbutter init-B*))
unfolding ∗ *fst-Rep-kf-Fst* **by** (*subst sum.reindex*) (*auto simp add*: *o-def inj2*)
moreover have ($\sum$*UAs*∈(Π_E *i*∈{*0*}. *fst ‘ Rep-kraus-family* ($\mathfrak{E}$ *i*)).
sandwich-tc (*Fst* (*UAs 0*)) (*tc-selfbutter init-B*)) =
($\sum$*E*∈*fst ‘* (*Rep-kraus-family* ($\mathfrak{E}$ *0*)). *sandwich-tc* (*Fst E*) (*tc-selfbutter init-B*))
(**is** *?left* = *?right*)
proof −
have *inj1*: *inj-on* (λ*UA. UA 0*) (Π_E *i*∈{*0*}. *fst ‘ Rep-kraus-family* ($\mathfrak{E}$ *i*))
by (*smt* (*verit*, *best*) *PiE-ext inj-on-def singletonD*)
have *non-empty*: (Π_E *i*∈{*0*}. *fst ‘ Rep-kraus-family* ($\mathfrak{E}$ *i*)) ≠ {}
by (*smt* (*verit*, *del-insts*) *PiE-eq-empty-iff add-is-0 assms*(*2*) *d-gr-0 emptyE insertE not-gr0*)
have *?left* = ($\sum$ *UA* ∈ (λ*UA. UA 0*) ‘ (Π_E *i*∈{*0*}. *fst ‘ Rep-kraus-family* ($\mathfrak{E}$ *i*)).
sandwich-tc (*Fst UA*) (*tc-selfbutter init-B*))
by (*subst sum.reindex*) (*auto simp add*: *inj1*)
also have . . . = ($\sum$ *UA* ∈ *fst ‘ Rep-kraus-family* ($\mathfrak{E}$ *0*).
sandwich-tc (*Fst UA*) (*tc-selfbutter init-B*))
by (*intro sum.cong*) (*auto simp add*: *image-projection-PiE non-empty*)
finally show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** *auto*
next
case (*Suc d*)
let *?*$\mathfrak{E}$ = (λ*i. kf-Fst* ($\mathfrak{E}$ *i*))
have *inj1*: *inj-on* (λ(*y*, *g*). *g*(*Suc d* := *y*))(*fst ‘ Rep-kraus-family* ($\mathfrak{E}$ (*Suc d*)) ×

$(\Pi_E\ i{\in}\{0..{<}Suc\ d\}.\ fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ i)))$
by (*metis atLeastLessThan-iff inj-combinator less-irrefl-nat*)
let $?\Phi = sandwich\text{-}tc\ ((X\text{-}for\text{-}B;Y\text{-}for\text{-}B)\ (Uquery\ H)\ o_{CL}\ US\ S\ d)$
$(run\text{-}mixed\text{-}adv\ d\ ?\mathfrak{E}\ (US\ S)\ init\text{-}B\ X\text{-}for\text{-}B\ Y\text{-}for\text{-}B\ H)$
let $?\Psi = (\lambda UAs.\ run\text{-}pure\text{-}adv\text{-}tc\ d\ UAs\ (US\ S)\ init\text{-}B\ X\text{-}for\text{-}B\ Y\text{-}for\text{-}B\ H)$
have *finite*: $finite\ (Rep\text{-}kraus\text{-}family\ (kf\text{-}Fst\ (\mathfrak{E}\ (Suc\ d))))$
using *Suc.prems*(*1*) *finite-kf-Fst* **by** *auto*
have $kf\text{-}apply\ (?\mathfrak{E}\ (Suc\ d))\ ?\Phi =$
$(\sum E{\in}fst\ `\ Rep\text{-}kraus\text{-}family\ (kf\text{-}Fst\ (\mathfrak{E}\ (Suc\ d))).\ sandwich\text{-}tc\ E\ ?\Phi)$
by (*subst sum.reindex*) (*auto simp add*: *kf-apply.rep-eq infsum-finite*[*OF finite*]
inj-on-fst-Rep-kraus-family)
also have $\ldots = (\sum E{\in}fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ (Suc\ d)).\ sandwich\text{-}tc\ (Fst\ E)\ ?\Phi)$
unfolding *fst-Rep-kf-Fst* **by** (*subst sum.reindex*) (*auto simp add*: *inj-on-Fst*)
also have $\ldots = (\sum UA{\in}fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ (Suc\ d)).$
$(\sum UAs{\in}(\Pi_E\ i{\in}\{0..{<}Suc\ d\}.\ fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ i)).\ sandwich\text{-}tc\ (Fst\ UA)$
$(sandwich\text{-}tc\ ((X\text{-}for\text{-}B;Y\text{-}for\text{-}B)\ (Uquery\ H)\ o_{CL}\ US\ S\ d)\ (?\Psi\ (Fst\ o\ UAs)))))$
using *Suc* **unfolding** *kf-Fst-def*[*symmetric*] **by** (*intro sum.cong*)(*auto simp add*: *sandwich-tc-sum o-def*)
also have $\ldots = (\sum (UA,UAs){\in}fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ (Suc\ d))\times$
$(\Pi_E\ i{\in}\{0..{<}Suc\ d\}.\ fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ i)).$
$sandwich\text{-}tc\ (Fst\ UA)\ (sandwich\text{-}tc\ ((X\text{-}for\text{-}B;Y\text{-}for\text{-}B)\ (Uquery\ H)\ o_{CL}\ US\ S\ d)\ (?\Psi\ (Fst\ o\ UAs))))$
by (*subst sum.cartesian-product*) *auto*
also have $\ldots = (\sum UAs{\in}(\lambda(y,\ g).\ g(Suc\ d := y))\ `\ (fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ (Suc\ d))\times$
$(\Pi_E\ i{\in}\ \{0..{<}Suc\ d\}.\ fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ i))).$
$sandwich\text{-}tc\ (Fst\ (UAs\ (Suc\ d))\ o_{CL}\ (X\text{-}for\text{-}B;Y\text{-}for\text{-}B)\ (Uquery\ H)\ o_{CL}\ US\ S\ d)(?\Psi\ (Fst\ o\ UAs)))$
by (*subst sum.reindex*)(*use run-pure-adv-tc-Fst-over*[**where** $init' = init\text{-}B$ **and** $X' = X\text{-}for\text{-}B$ **and** $Y' = Y\text{-}for\text{-}B$]
in ‹*auto intro*!: *sum.cong simp add*: *o-def sandwich-tc-compose inj1*›)
also have $\ldots = (\sum UAs{\in}(\Pi_E\ i{\in}insert\ (Suc\ d)\ \{0..{<}Suc\ d\}.\ fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ i)).$
$sandwich\text{-}tc\ (Fst\ (UAs\ (Suc\ d))\ o_{CL}\ (X\text{-}for\text{-}B;Y\text{-}for\text{-}B)\ (Uquery\ H)\ o_{CL}\ US\ S\ d)\ (?\Psi\ (Fst\ o\ UAs)))$
by (*subst PiE-insert-eq*) *auto*
finally show *?case* **unfolding** *kf-Fst-def* **by** (*auto simp add*: *set-upt-Suc o-def*)
qed

lemma *purification-run-mixed-B-count-prep*:
assumes $\bigwedge i.\ i{<}d{+}1 \Longrightarrow finite\ (Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ i))$
assumes $\bigwedge i.\ i{<}d{+}1 \Longrightarrow fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ i) \neq \{\}$
assumes $n{<}d{+}1$
shows $run\text{-}mixed\text{-}adv\ n\ (\lambda n.\ kf\text{-}Fst\ (\mathfrak{E}\ n))\ (\lambda n.\ U\text{-}S'\ S)$
$init\text{-}B\text{-}count\ X\text{-}for\text{-}C\ Y\text{-}for\text{-}C\ H =$
$(\sum UAs{\in}(\Pi_E\ i{\in}\{0..{<}n\ +\ 1\}.\ fst\ `\ Rep\text{-}kraus\text{-}family\ (\mathfrak{E}\ i)).$
$run\text{-}pure\text{-}adv\text{-}tc\ n\ (Fst \circ UAs)\ (\lambda\text{-}.\ U\text{-}S'\ S)\ init\text{-}B\text{-}count\ X\text{-}for\text{-}C$
$Y\text{-}for\text{-}C\ H)$
using *assms*
proof (*induct n*)

case *0*
let *?*$\mathfrak{E}$*0* = *kf-Fst* ($\mathfrak{E}$ *0*)
have *finite*: *finite* (*Rep-kraus-family ?*$\mathfrak{E}$*0*)
using *finite-kf-Fst assms* **by** *auto*
have *inj1*: *inj-on fst* (*Rep-kraus-family ?*$\mathfrak{E}$*0*)
by (*rule inj-on-fst-Rep-kraus-family*)
have *inj2*: *inj-on Fst* (*fst ' Rep-kraus-family* ($\mathfrak{E}$ *0*)) **using** *inj-on-Fst* **by** *auto*
have $*$: *kf-apply ?*$\mathfrak{E}$*0* (*tc-selfbutter init-B-count*) =
($\sum$ *E*∈*fst '* (*Rep-kraus-family ?*$\mathfrak{E}$*0*). *sandwich-tc E* (*tc-selfbutter init-B-count*))
unfolding *kf-apply.rep-eq*
by (*subst sum.reindex*) (*auto simp add*: *infsum-finite*[*OF finite*] *inj1*)
have *kf-apply ?*$\mathfrak{E}$*0* (*tc-selfbutter init-B-count*) =
($\sum$ *E*∈*fst '* (*Rep-kraus-family* ($\mathfrak{E}$ *0*)). *sandwich-tc* (*Fst E*) (*tc-selfbutter init-B-count*))
unfolding $*$ *fst-Rep-kf-Fst* **by** (*subst sum.reindex*) (*auto simp add*: *o-def inj2*)
moreover have ($\sum$ *UAs*∈(Π_E *i*∈{*0*}. *fst ' Rep-kraus-family* ($\mathfrak{E}$ *i*)).
sandwich-tc (*Fst* (*UAs 0*)) (*tc-selfbutter init-B-count*)) =
($\sum$ *E*∈*fst '* (*Rep-kraus-family* ($\mathfrak{E}$ *0*)). *sandwich-tc* (*Fst E*) (*tc-selfbutter init-B-count*))
(**is** *?left* = *?right*)
proof −
have *inj1*: *inj-on* (λ*UA*. *UA 0*) (Π_E *i*∈{*0*}. *fst ' Rep-kraus-family* ($\mathfrak{E}$ *i*))
by (*smt* (*verit*, *best*) *PiE-ext inj-on-def singletonD*)
have *non-empty*: (Π_E *i*∈{*0*}. *fst ' Rep-kraus-family* ($\mathfrak{E}$ *i*)) ≠ {}
by (*smt* (*verit*, *del-insts*) *PiE-eq-empty-iff add-is-0 assms*(*2*) *d-gr-0 emptyE insertE not-gr0*)
have *?left* = ($\sum$ *UA* ∈ (λ*UA*. *UA 0*) ' (Π_E *i*∈{*0*}. *fst ' Rep-kraus-family* ($\mathfrak{E}$ *i*)).
sandwich-tc (*Fst UA*) (*tc-selfbutter init-B-count*))
by (*subst sum.reindex*) (*auto simp add*: *inj1*)
also have . . . = ($\sum$ *UA* ∈ *fst ' Rep-kraus-family* ($\mathfrak{E}$ *0*).
sandwich-tc (*Fst UA*) (*tc-selfbutter init-B-count*))
by (*intro sum.cong*) (*auto simp add*: *image-projection-PiE non-empty*)
finally show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** *auto*
next
case (*Suc n*)
define $\mathfrak{E}'$ :: (*'mem*×*nat*) *kraus-adv* **where** $\mathfrak{E}'$ = (λ*i*. *kf-Fst* ($\mathfrak{E}$ *i*))
have *inj1*: *inj-on* (λ(*y*, *g*). *g*(*Suc n* := *y*))(*fst ' Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*)) ×
(Π_E *i*∈{*0*..<*Suc n*}. *fst ' Rep-kraus-family* ($\mathfrak{E}$ *i*)))
by (*metis atLeastLessThan-iff inj-combinator less-irrefl-nat*)
let *?*Φ = *sandwich-tc* ((*X-for-C*;*Y-for-C*) (*Uquery H*) o_{CL} *U-S' S*)
(*run-mixed-adv n* $\mathfrak{E}'$ (λ-. *U-S' S*) *init-B-count X-for-C Y-for-C H*)
define Ψ **where** Ψ = (λ*UAs*. *run-pure-adv-tc n UAs* (λ-. *U-S' S*) *init-B-count X-for-C Y-for-C H*)

have *finite*: *finite* (*Rep-kraus-family* (*kf-Fst* ($\mathfrak{E}$ (*Suc n*))))
using *Suc.prems finite-kf-Fst* **by** *auto*
have *kf-apply* ($\mathfrak{E}'$ (*Suc n*)) *?*Φ =
($\sum$ *E*∈*fst ' Rep-kraus-family* (*kf-Fst* ($\mathfrak{E}$ (*Suc n*))). *sandwich-tc E ?*Φ)
unfolding $\mathfrak{E}'$*-def*
by (*subst sum.reindex*) (*auto simp add*: *kf-apply.rep-eq infsum-finite*[*OF finite*]

inj-on-fst-Rep-kraus-family)
also have ... = $(\sum$ *E*∈*fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*)). *sandwich-tc* (*Fst E*) ?Φ)
unfolding *fst-Rep-kf-Fst* **by** (*subst sum.reindex*) (*auto simp add*: *inj-on-Fst*)
also have ... = $(\sum$ *UA*∈*fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*)).
$(\sum$ *UAs*∈(Π_E *i*∈{*0*..<*Suc n*}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)). *sandwich-tc* (*Fst UA*)
(*sandwich-tc* ((*X-for-C*;*Y-for-C*) (*Uquery H*) o_{CL} *U-S′ S*) (Ψ (*Fst o UAs*)))))
using *Suc* **unfolding** *kf-Fst-def*[*symmetric*] Ψ*-def* $\mathfrak{E}$′*-def*
by (*auto simp add*: *sandwich-tc-sum o-def intro*!: *sum.cong*)
also have ... = $(\sum$ (*UA*,*UAs*)∈*fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*))×
(Π_E *i*∈{*0*..<*Suc n*}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)). *sandwich-tc* (*Fst UA*)
(*sandwich-tc* ((*X-for-C*;*Y-for-C*) (*Uquery H*) o_{CL} *U-S′ S*) (Ψ (*Fst o UAs*))))
by (*subst sum.cartesian-product*) *auto*
also have ... = $(\sum$ *UAs*∈(λ(*y*, *g*). *g*(*Suc n* := *y*)) ‘ (*fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ (*Suc n*)) ×
(Π_E *i*∈ {*0*..<*Suc n*}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*))). *sandwich-tc* (*Fst* (*UAs* (*Suc n*)) o_{CL}
(*X-for-C*;*Y-for-C*) (*Uquery H*) o_{CL} *U-S′ S*)(Ψ (*Fst o UAs*)))
unfolding Ψ*-def* **by** (*subst sum.reindex*)
(*use run-pure-adv-tc-Fst-over*[**where** *init′* = *init-B-count* **and** *X′* = *X-for-C* **and** *Y′* = *Y-for-C*]
in ‹*auto intro*!: *sum.cong simp add*: *o-def sandwich-tc-compose inj1*›)
also have ... = $(\sum$ *UAs*∈(Π_E *i*∈*insert* (*Suc n*) {*0*..<*Suc n*}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)).
sandwich-tc (*Fst* (*UAs* (*Suc n*)) o_{CL} (*X-for-C*;*Y-for-C*) (*Uquery H*) o_{CL} *U-S′ S*)
(Ψ (*Fst o UAs*)))
by (*subst PiE-insert-eq*) *auto*
also have ... = $(\sum$ *UAs*∈(Π_E *i*∈{*0*..<*Suc n* + *1*}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)).
run-pure-adv-tc (*Suc n*) (*Fst* ∘ *UAs*) (λ-. *U-S′ S*) *init-B-count X-for-C Y-for-C H*)
unfolding *kf-Fst-def* Ψ*-def* **by** (*auto simp add*: *set-upt-Suc o-def intro*!: *sum.cong*)
finally show *?case* **unfolding** $\mathfrak{E}$′*-def* **by** (*auto simp add*: *comp-def*)
qed

lemma *purification-run-mixed-B-count*:
assumes $\bigwedge$*i*. *i*<*d*+*1* ⟹ *finite* (*Rep-kraus-family* ($\mathfrak{E}$ *i*))
assumes $\bigwedge$*i*. *i*<*d*+*1* ⟹ *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*) ≠ {}
shows *run-mixed-B-count* $\mathfrak{E}$ *H S* = $(\sum$ *UAs* ∈ *purify-comp-kraus d* $\mathfrak{E}$. *run-pure-B-count-tc UAs H S*)
unfolding *run-mixed-B-count-def run-pure-B-count-tc-def purify-comp-kraus-def*
using *purification-run-mixed-B-count-prep*[**where** *n*=*d*, *OF assms*] **by** *auto*

Purification of *kf-Fst*

lemma *purification-kf-Fst*:
assumes $\bigwedge$*i*. *i* < *n* + *1* ⟹ *fst* ‘ *Rep-kraus-family* (*F i*) ≠ {}
assumes *x* ∈ *purify-comp-kraus n* (λ*n*. *kf-Fst* (*F n*)::((′*a* × ′*c*) *ell2*, (′*b* × ′*c*) *ell2*, *unit*) *kraus-family*)
shows ∃ *UA*. *x* = (λ*a*. *if a*<*n*+*1 then* (*Fst* (*UA a*)::(′*a* × ′*c*) *ell2* $\Rightarrow_{CL}$ (′*b* × ′*c*) *ell2*) *else undefined*)
proof −
have *nonempty*: *PiE* {*0*..<*n*+*1*} (λ*i*. *Fst* ‘ *fst* ‘ *Rep-kraus-family* (*F i*)
::((′*a* × ′*c*) *ell2* $\Rightarrow_{CL}$ (′*b* × ′*c*) *ell2*) *set*) ≠ {}
by (*rule ccontr*) (*use assms* **in** ‹*auto simp add*: *PiE-eq-empty-iff*›)
have *x*∈ *PiE* {*0*..<*n*+*1*} (λ*i*. *Fst* ‘ *fst* ‘ *Rep-kraus-family* (*F i*))

using *assms* **unfolding** *purify-comp-kraus-def fst-Rep-kf-Fst* **by** *auto*
then have *elem*: $x\ a \in (\lambda f.\ f\ a)$ ‘ *PiE* $\{0..<n+1\}$ $(\lambda i.$ *Fst* ‘ *fst* ‘ *Rep-kraus-family* $(F\ i))$
for *a* **by** *blast*
have *rew*: $(\lambda f.\ f\ a)$ ‘ *PiE* $\{0..<n+1\}$ $(\lambda i.$ *Fst* ‘ *fst* ‘ *Rep-kraus-family* $(F\ i)$::
$(('a \times 'c)$ *ell2* $\Rightarrow_{CL}$ $('b \times 'c)$ *ell2*$)$ *set*$)$ =
$($*if* $a\in\{0..<n+1\}$*then Fst* ‘ *fst* ‘ *Rep-kraus-family* $(F\ a)$ *else* $\{$*undefined*$\})$
for *a* **by** $($*subst image-projection-PiE*$)$ $($*use nonempty* **in** ‹*auto*›$)$
have *el*: $x\ a \in ($*if* $a\in\{0..<n+1\}$ *then Fst* ‘ *fst* ‘ *Rep-kraus-family* $(F\ a)$ *else* $\{$*undefined*$\})$
for *a* **using** *elem* **unfolding** *rew* **by** *auto*
have *ins*: $a\in\{0..<n+1\} \Longrightarrow x\ a \in$ *Fst* ‘ *fst* ‘ *Rep-kraus-family* $(F\ a)$ **for** *a* **using** *el* **by** *meson*
then have $\exists\, v.\ (v \in$ *Rep-kraus-family* $(F\ a) \wedge x\ a =$ *Fst*$($*fst*$(v)))$ **if** $a\in\{0..<n+1\}$ **for** *a*
using *that* **by** *fastforce*
then obtain *v* **where** *v-in*: $a\in\{0..<n+1\} \Longrightarrow v\ a \in$ *Rep-kraus-family* $(F\ a)$
and *x*: $a\in\{0..<n+1\} \Longrightarrow x\ a = ($*Fst* $($*fst* $(v\ a))$::$('a \times 'c)$ *ell2* $\Rightarrow_{CL}$ $('b \times 'c)$ *ell2*$)$ **for** *a*
by *metis*
have *outs*: $\neg a\in\{0..<n+1\} \Longrightarrow x\ a =$ *undefined* **for** *a* **by** $($*meson el singletonD*$)$
define *val* **where** *val a* = $($*if* $a\in\{0..<n+1\}$ *then v a else undefined*$)$ **for** *a*
have $x\ a =$ *Fst* $($*fst* $($*val a*$))$ **if** $a\in\{0..<n+1\}$ **for** *a*
unfolding *val-def* **using** *x*[*OF that*] *that* **by** *auto*
then have $x\ a = ($*if* $a \in \{0..<n+1\}$ *then Fst* $($*fst* $($*val a*$))$ *else undefined*$)$
for *a* **by** $($*cases* $a\in\{0..<n+1\}$, *use outs* **in** ‹*auto*›$)$
then show *?thesis* **by** $($*intro exI*[*of* - $(\lambda a.$ *fst* $($*val a*$))$]$)$ *auto*
qed

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*
unbundle *no register-syntax*

end
theory *Mixed-O2H*

imports *Pure-O2H*
Estimation
Run-Adversary
Limit-Process
Purification

begin

9 Mixed O2H Setting and Preliminaries

hide-const (**open**) *Determinants.trace*

locale *mixed-o2h* = *o2h-setting* *TYPE*$('x)$ *TYPE*$('y$::*group-add*$)$ *TYPE*$('mem)$ *TYPE*$('l)$ +

— We fix the distributions on H and S. (They might be correlated.) So far, we assume that they are discrete distributions and model them in the following way:

fixes *carrier* :: $(('x \Rightarrow 'y)\times('x \Rightarrow bool)\times\text{-})$ *set*
fixes *distr* :: $(('x \Rightarrow 'y)\times('x \Rightarrow bool)\times\text{-}) \Rightarrow$ *real*

assumes *distr-pos*: $\forall (H,S,z)\in carrier.\ distr\ (H,S,z) \geq 0$
  and *distr-sum-1*: $(\sum (H,S,z)\in carrier.\ distr\ (H,S,z)) = 1$
  and *finite-carrier*: *finite carrier*

fixes *E*:: *'mem kraus-adv*
assumes *E-norm-id*: $\bigwedge i.\ i < d+1 \Longrightarrow$ *kf-bound* $(E\ i) \leq$ *id-cblinfun*
assumes *E-nonzero*: $\bigwedge i.\ i < d+1 \Longrightarrow$ *Rep-kraus-family* $(E\ i) \neq \{\}$

fixes *P*:: *'mem update*
assumes *is-Proj-P*: *is-Proj P*

begin

lemma *norm-P*:
  $norm\ P \leq 1$
  using *is-Proj-P* **by** (*simp add*: *norm-is-Proj*)

lemma *distr-pos'*:
  assumes $(H,S,z)\in carrier$ **shows** $distr\ (H,S,z) \geq 0$
  using *distr-pos assms* **by** *auto*

lemma *norm-Fst-P*:
  norm (*Fst P*:: $('mem \times 'a)$ *update*) ≤ 1
  by (*simp add*: *Fst-def norm-P tensor-op-norm*)

9.1 Final states

definition *ϱleft-pure* :: $(nat \Rightarrow 'mem\ update) \Rightarrow 'mem\ tc\text{-}op$ **where**
  ϱleft-pure UA $= (\sum (H,S,z)\in carrier.\ distr\ (H,S,z) *_C$ *run-pure-A-tc UA H*)

definition *ϱleft* :: *'mem kraus-adv* $\Rightarrow$ *'mem tc-op* **where**
  ϱleft F $= (\sum (H,S,z)\in carrier.\ distr\ (H,S,z) *_C$ *run-mixed-A F H*)

definition *ϱright-pure* :: $(nat \Rightarrow 'mem\ update) \Rightarrow ('mem \times 'l)\ tc\text{-}op$ **where**
  ϱright-pure UA $= (\sum (H,S,z)\in carrier.\ distr\ (H,S,z) *_C$ *run-pure-B-tc UA H S*)

definition *ϱright* :: *'mem kraus-adv* $\Rightarrow ('mem \times 'l)$ *tc-op* **where**

ϱright F = (∑(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) $*_C$ *run-mixed-B F H S*)

definition *ϱcount-pure* :: (*nat* ⇒ *'mem update*) ⇒ (*'mem* × *nat*) *tc-op* **where**
ϱcount-pure UA = (∑(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) $*_C$ *run-pure-B-count-tc UA H S*)

definition *ϱcount* :: *'mem kraus-adv* ⇒ (*'mem* × *nat*) *tc-op* **where**
ϱcount F = (∑(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) $*_C$ *run-mixed-B-count F H S*)

Positivity

lemma *ϱleft-pure-pos*: *0* ≤ *ϱleft-pure UA*
unfolding *ϱleft-pure-def* **by** (*intro sum-nonneg*) (*metis* (*mono-tags*, *lifting*) *complex-of-real-nn-iff*

distr-pos prod.case-eq-if run-pure-A-tc-pos scaleC-nonneg-nonneg)

lemma *ϱleft-pos*: *0* ≤ *ϱleft F*
unfolding *ϱleft-def* **by** (*intro sum-nonneg*) (*metis* (*mono-tags*, *lifting*) *complex-of-real-nn-iff*
distr-pos prod.case-eq-if run-mixed-A-pos scaleC-nonneg-nonneg)

lemma *ϱright-pure-pos*: *0* ≤ *ϱright-pure UA*
unfolding *ϱright-pure-def* **by** (*intro sum-nonneg*) (*metis* (*mono-tags*, *lifting*) *complex-of-real-nn-iff*

distr-pos prod.case-eq-if run-pure-B-tc-pos scaleC-nonneg-nonneg)

lemma *ϱright-pos*: *0* ≤ *ϱright F*
unfolding *ϱright-def* **by** (*intro sum-nonneg*) (*metis* (*mono-tags*, *lifting*) *complex-of-real-nn-iff*

distr-pos prod.case-eq-if run-mixed-B-pos scaleC-nonneg-nonneg)

lemma *ϱcount-pure-pos*: *0* ≤ *ϱcount-pure UA*
unfolding *ϱcount-pure-def* **by** (*intro sum-nonneg*)(*metis* (*mono-tags*, *lifting*) *complex-of-real-nn-iff*

distr-pos prod.case-eq-if run-pure-B-count-tc-pos scaleC-nonneg-nonneg)

lemma *ϱcount-pos*: *0* ≤ *ϱcount F*
unfolding *ϱcount-def* **by** (*intro sum-nonneg*) (*metis* (*mono-tags*, *lifting*) *complex-of-real-nn-iff*

distr-pos prod.case-eq-if run-mixed-B-count-pos scaleC-nonneg-nonneg)

Norm leq 1, trace-preserving adversary states have norm 1

lemma *norm-ϱleft*:
norm (*ϱleft E*) ≤ *1*
proof −
have *norm* (*ϱleft E*) ≤ (∑(*H*,*S*,*z*)∈*carrier*. *norm* ((*distr* (*H*,*S*,*z*)) $*_C$ *run-mixed-A E H*))
unfolding *ϱleft-def* **using** *norm-sum* **by** (*simp add*: *prod.case-eq-if sum-norm-le*)
also have ... = (∑(*H*,*S*,*z*)∈*carrier*. (*distr* (*H*,*S*,*z*)) $*_C$ *norm* (*run-mixed-A E H*))
by (*auto intro*!: *sum.cong simp add*: *distr-pos'*)
also have ... ≤ (∑(*H*,*S*,*z*)∈*carrier*. (*distr* (*H*,*S*,*z*)) $*_C$ *1*)
proof (*intro sum-mono*, *safe*, *goal-cases*)

```
    case (1 H S z)
    have one: 0 ≤ complex-of-real (distr (H, S, z)) using distr-pos' 1 by auto
    have two: complex-of-real (norm (run-mixed-A E H)) ≤ (1::complex)
      using norm-run-mixed-A[OF E-norm-id] 1 by auto
    then show ?case by (metis complex-scaleC-def one mult-left-mono)
  qed
  also have ... = 1 using distr-sum-1 by (auto simp add: of-real-sum[symmetric])
  finally show ?thesis by auto
qed

lemma norm-ϱright:
  norm (ϱright E) ≤ 1
proof –
  have norm (ϱright E) ≤ (∑(H,S,z)∈carrier. norm ((distr (H,S,z)) *C run-mixed-B E H S))
    unfolding ϱright-def using norm-sum by (simp add: prod.case-eq-if sum-norm-le)
  also have ... = (∑(H,S,z)∈carrier. (distr (H,S,z)) *C norm (run-mixed-B E H S))
    by (auto intro!: sum.cong simp add: distr-pos')
  also have ... ≤ (∑(H,S,z)∈carrier. (distr (H,S,z)) *C 1)
  proof (intro sum-mono, safe, goal-cases)
    case (1 H S z)
    have one: 0 ≤ complex-of-real (distr (H, S, z)) using distr-pos' 1 by auto
    have two: complex-of-real (norm (run-mixed-B E H S)) ≤ (1::complex)
      using norm-run-mixed-B[OF E-norm-id] 1 by auto
    then show ?case by (metis complex-scaleC-def one mult-left-mono)
  qed
  also have ... = 1 using distr-sum-1 by (auto simp add: of-real-sum[symmetric])
  finally show ?thesis by auto
qed

lemma norm-ϱcount:
  norm (ϱcount E) ≤ 1
proof –
  have norm (ϱcount E) ≤ (∑(H,S,z)∈carrier. norm ((distr (H,S,z)) *C run-mixed-B-count
E H S))
    unfolding ϱcount-def using norm-sum by (simp add: prod.case-eq-if sum-norm-le)
  also have ... = (∑(H,S,z)∈carrier. (distr (H,S,z)) *C norm (run-mixed-B-count E H S))
    by (auto intro!: sum.cong simp add: distr-pos')
  also have ... ≤ (∑(H,S,z)∈carrier. (distr (H,S,z)) *C 1)
  proof (intro sum-mono, safe, goal-cases)
    case (1 H S z)
    have one: 0 ≤ complex-of-real (distr (H, S, z)) using distr-pos' 1 by auto
    have two: complex-of-real (norm (run-mixed-B-count E H S)) ≤ (1::complex)
      using norm-run-mixed-B-count[OF E-norm-id] 1 by auto
    then show ?case by (metis complex-scaleC-def one mult-left-mono)
  qed
  also have ... = 1 using distr-sum-1 by (auto simp add: of-real-sum[symmetric])
  finally show ?thesis by auto
qed
```

lemma *trace-preserving-norm-ϱright*:
assumes $\bigwedge i.\ i < d+1 \Longrightarrow$ *km-trace-preserving* (*kf-apply*
(*kf-Fst* (*E i*)::(('*mem* × '*l*) *ell2*, ('*mem* × '*l*) *ell2*, *unit*) *kraus-family*))
shows *norm* (*ϱright E*) = *1*
proof −
have *norm* ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)) $*_C$ *run-mixed-B E H S*) =
trace-tc ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)) $*_C$ *run-mixed-B E H S*)
using *ϱright-def ϱright-pos norm-tc-pos* **by** *auto*
also have ... = ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)) ∗ *trace-tc* (*run-mixed-B E H S*))
by (*smt* (*verit*) *complex-scaleC-def prod.case-eq-if sum.cong trace-tc-scaleC trace-tc-sum*)
also have ... = ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)) ∗ *norm* (*run-mixed-B E H S*))
by (*subst of-real-sum*, *intro sum.cong*, *simp*)
(*simp add*: *norm-tc-pos prod.case-eq-if run-mixed-B-pos*)
also have ... = ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)))
using *trace-preserving-norm-run-mixed-B*[*OF assms*] **by** *auto*
also have ... = *1* **using** *distr-sum-1* **by** *blast*
finally show *?thesis* **unfolding** *ϱright-def* **by** *auto*
qed

lemma *trace-preserving-norm-ϱcount*:
assumes $\bigwedge i.\ i < d+1 \Longrightarrow$ *km-trace-preserving* (*kf-apply*
(*kf-Fst* (*E i*)::(('*mem* × *nat*) *ell2*, ('*mem* × *nat*) *ell2*, *unit*) *kraus-family*))
shows *norm* (*ϱcount E*) = *1*
proof −
have *norm* ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)) $*_C$ *run-mixed-B-count E H S*) =
trace-tc ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)) $*_C$ *run-mixed-B-count E H S*)
using *ϱcount-def ϱcount-pos norm-tc-pos* **by** *auto*
also have ... = ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)) ∗ *trace-tc* (*run-mixed-B-count E H S*))
by (*smt* (*verit*) *complex-scaleC-def prod.case-eq-if sum.cong trace-tc-scaleC trace-tc-sum*)
also have ... = ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)) ∗ *norm* (*run-mixed-B-count E H S*))
by (*subst of-real-sum*, *intro sum.cong*, *simp*)
(*simp add*: *norm-tc-pos prod.case-eq-if run-mixed-B-count-pos*)
also have ... = ($\sum$(*H*, *S*, *z*)∈*carrier*. (*distr* (*H*, *S*, *z*)))
using *trace-preserving-norm-run-mixed-B-count*[*OF assms*] **by** *auto*
also have ... = *1* **using** *distr-sum-1* **by** *blast*
finally show *?thesis* **unfolding** *ϱcount-def* **by** *auto*
qed

Summability and Infsums

lemma *from-trace-class-ϱright-pure*:
from-trace-class (*ϱright-pure UA*) = ($\sum$(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) $*_C$ *run-pure-B-update UA H S*)
by (*smt* (*verit*) *ϱright-pure-def from-trace-class-sum run-pure-B-tc-def prod.case-eq-if run-pure-B-update-def run-pure-adv-update-tc' scaleC-trace-class.rep-eq sum.cong*)

lemma *has-sum-scaleC-tc*:
fixes x :: (*'a::chilbert-space,'a*) *trace-class*
assumes (f *has-sum* x) A
shows (($\lambda y.$ c $*_C$ f y) *has-sum* c $*_C$ x) A
using *assms* **by** (*rule has-sum-scaleC-right*)

lemma *ϱleft-has-sum*:
(*ϱleft has-sum ϱleft E*) (*finite-kraus-subadv E d*)
proof −
have (($\lambda x.$ (*distr* (H,S,z)) $*_C$ *run-mixed-A x H*) *has-sum* (*distr* (H,S,z)) $*_C$ *run-mixed-A E H*)
(*finite-kraus-subadv E d*) **for** H S z
using *has-sum-scaleC-tc*[*OF run-mixed-A-has-sum*[*of H E*]] **by** *auto*
then show *?thesis* **unfolding** *ϱleft-def* **by** (*intro has-sum-finite-sum*[*OF - finite-carrier*])
(*auto simp add*: *case-prod-beta intro*!: *has-sum-cmult-right*)
qed

lemma *ϱright-has-sum*:
(($\lambda f.$ *ϱright f*) *has-sum ϱright E*) (*finite-kraus-subadv E d*)
proof −
have (($\lambda x.$ (*distr* (H,S,z)) $*_C$ *run-mixed-B x H S*) *has-sum* (*distr* (H,S,z)) $*_C$ *run-mixed-B E H S*)
(*finite-kraus-subadv E d*) **for** H S z
using *has-sum-scaleC-tc*[*OF run-mixed-B-has-sum'*[*of H S E*]] **by** *auto*
then show *?thesis* **unfolding** *ϱright-def* **by** (*intro has-sum-finite-sum*[*OF - finite-carrier*])
(*auto simp add*: *case-prod-beta intro*!: *has-sum-cmult-right*)
qed

lemma *ϱright-abs-summable*:
ϱright abs-summable-on (*finite-kraus-subadv E d*)
using *has-sum-imp-summable*[*OF ϱright-has-sum*] *ϱright-pos summable-abs-summable-tc* **by** *blast*

lemma *ϱcount-has-sum*:
(*ϱcount has-sum ϱcount E*) (*finite-kraus-subadv E d*)
proof −
have (($\lambda x.$ (*distr* (H,S,z)) $*_C$ *run-mixed-B-count x H S*) *has-sum* (*distr* (H,S,z)) $*_C$ *run-mixed-B-count E H S*)
(*finite-kraus-subadv E d*) **for** H S z
using *has-sum-scaleC-tc*[*OF run-mixed-B-count-has-sum'*[*of H S E*]] **by** *auto*
then show *?thesis* **unfolding** *ϱcount-def* **by** (*intro has-sum-finite-sum*[*OF - finite-carrier*])
(*auto simp add*: *case-prod-beta intro*!: *has-sum-cmult-right*)
qed

Connection pure and mixed states

lemma *ϱleft-pure-mixed*:
  assumes $\bigwedge i.\ i < d + 1 \Longrightarrow$ *finite* (*Rep-kraus-family* (*F i*))
    $\bigwedge i.\ i < d + 1 \Longrightarrow$ *fst* ' *Rep-kraus-family* (*F i*) $\neq$ {}
  shows *ϱleft F* = ($\sum$ *UAs* $\in$ *purify-comp-kraus d F*. *ϱleft-pure UAs*)
proof −
  have *run*: *run-mixed-A F H* = ($\sum$ *UAs*$\in$*purify-comp-kraus d F*. *run-pure-A-tc UAs H*) **for** *H*
    using *purification-run-mixed-A assms* **by** *auto*
  have *ϱleft F* = ($\sum$ *UAs*$\in$*purify-comp-kraus d F*. ($\sum$ (*H*,*S*,*z*)$\in$*carrier*.
      complex-of-real (*distr* (*H*,*S*,*z*)) $*_C$ *run-pure-A-tc UAs H*))
   unfolding *ϱleft-def run* **by** (*subst sum.swap*) (*auto intro*!: *sum.cong simp add*: *scaleC-sum-right*)
  then show *?thesis* **unfolding** *ϱleft-pure-def* **by** *auto*
qed

lemma *ϱright-pure-mixed*:
  assumes $\bigwedge i.\ i < d + 1 \Longrightarrow$ *finite* (*Rep-kraus-family* (*F i*))
    $\bigwedge i.\ i < d + 1 \Longrightarrow$ *fst* ' *Rep-kraus-family* (*F i*) $\neq$ {}
  shows *ϱright F* = ($\sum$ *UAs* $\in$ *purify-comp-kraus d F*. *ϱright-pure UAs*)
proof −
  have *run*: *run-mixed-B F H S* = ($\sum$ *UAs*$\in$*purify-comp-kraus d F*. *run-pure-B-tc UAs H S*)
for *H S*
    using *purification-run-mixed-B assms* **by** *blast*
  have *ϱright F* = ($\sum$ *UAs*$\in$*purify-comp-kraus d F*. ($\sum$ (*H*,*S*,*z*)$\in$*carrier*.
      complex-of-real (*distr* (*H*,*S*,*z*)) $*_C$ *run-pure-B-tc UAs H S*))
   unfolding *ϱright-def run* **by** (*subst sum.swap*) (*auto intro*!: *sum.cong simp add*: *scaleC-sum-right*)
  then show *?thesis* **unfolding** *ϱright-pure-def* **by** *auto*
qed

lemma *ϱcount-pure-mixed*:
  assumes $\bigwedge i.\ i < d + 1 \Longrightarrow$ *finite* (*Rep-kraus-family* (*F i*))
    $\bigwedge i.\ i < d + 1 \Longrightarrow$ *fst* ' *Rep-kraus-family* (*F i*) $\neq$ {}
  shows *ϱcount F* = ($\sum$ *UAs* $\in$ *purify-comp-kraus d F*. *ϱcount-pure UAs*)
proof −
  have *run*: *run-mixed-B-count F H S* = ($\sum$ *UAs*$\in$*purify-comp-kraus d F*. *run-pure-B-count-tc*
UAs H S) **for** *H S*
    using *purification-run-mixed-B-count assms* **by** *blast*
  have *ϱcount F* = ($\sum$ *UAs*$\in$*purify-comp-kraus d F*. ($\sum$ (*H*,*S*,*z*)$\in$*carrier*.
      complex-of-real (*distr* (*H*,*S*,*z*)) $*_C$ *run-pure-B-count-tc UAs H S*))
   unfolding *ϱcount-def run* **by** (*subst sum.swap*) (*auto intro*!: *sum.cong simp add*: *scaleC-sum-right*)
  then show *?thesis* **unfolding** *ϱcount-pure-def* **by** *auto*
qed

9.2 Measurement at the end

Measurement at the end of the adversary run. *end-measure* measures whether there was a find element (event "Find").

definition *end-measure* :: (*'mem* × *'l*) *update* **where**
  end-measure = *Snd* (*id-cblinfun* − *selfbutter* (*ket empty*))

lemma *is-Proj-Snd*:
assumes *is-Proj f*
shows *is-Proj* (*Snd f*)
by (*simp add*: *assms register-projector*)

lemma *is-Proj-end-measure*:
is-Proj end-measure
unfolding *end-measure-def* **by** (*auto intro*!: *is-Proj-Snd simp add*: *butterfly-is-Proj*)

lemma *Proj-end-measure*:
Proj (*end-measure* $*_S$ $\top$) = *end-measure*
using *Proj-on-own-range is-Proj-end-measure* **by** *auto*

lemma *norm-end-measure*:
norm (*end-measure*) $\leq$ *1*
using *norm-is-Proj is-Proj-end-measure* **by** *auto*

lemma *end-measure-butterfly*:
sandwich end-measure (*selfbutter* Ψ) = *selfbutter* (*end-measure* $*_V$ Ψ)
by (*rule sandwich-butterfly*)

lemma *trace-end-measure*:
trace (*end-measure* o_{CL} *selfbutter* Ψ) = (*complex-of-real* (*norm* (*end-measure* $*_V$ Ψ)))^2
by (*metis* (*no-types*, *lifting*) *cblinfun-apply-cblinfun-compose cinner-adj-left*
is-Proj-algebraic is-Proj-end-measure power2-norm-eq-cinner trace-butterfly-comp′)

lemma *trace-endmeasure-pos*:
assumes $\varrho \geq 0$
shows *trace-tc* (*compose-tcr end-measure* ϱ) $\geq$ *0*
by (*metis* (*no-types*, *opaque-lifting*) *assms compose-tcr.rep-eq from-trace-class-pos is-Proj-algebraic*

is-Proj-end-measure positive-cblinfun-squareI trace-class-from-trace-class trace-comp-pos
trace-tc.rep-eq)

lemma *trace-class-end-measure*:
assumes *trace-class a*
shows *trace-class* (*end-measure* o_{CL} *a*)
using *assms* **by** (*rule trace-class-comp-right*)

lemma *abs-op-id-cblinfun* [*simp*]:
abs-op id-cblinfun = *id-cblinfun*
by (*simp add*: *abs-op-id-on-pos*)

10 *empty-tc* is the trace-class representative of the 0.

definition *empty-tc* :: *'l tc-op* **where**
empty-tc = *Abs-trace-class* (*selfbutter* (*ket empty*))

lemma *norm-empty-tc*:
norm empty-tc = *1*
unfolding *empty-tc-def* **by** (*metis more-arith-simps*(*6*) *norm-ket norm-tc-butterfly tc-butterfly.abs-eq*)

lemma *empty-tc-pos*: *0* ≤ *empty-tc*
unfolding *empty-tc-def* **by** (*simp add*: *Abs-trace-class-geq0I*)

10.1 Projective measurement PM

The projective measurement PM Q at the end

definition *PM-update* :: (*'mem* × *'l*) *update* ⇒ (*'mem* × *'l*) *update* ⇒ *complex* **where**
PM-update Q ϱ = *trace* (*sandwich Q* ϱ)

lemma *PM-update-linear*:
assumes *trace-class* ϱ *trace-class* ψ
shows *PM-update Q* (ϱ + ψ) = *PM-update Q* ϱ + *PM-update Q* ψ
unfolding *PM-update-def*
by (*simp add*: *assms*(*1*) *assms*(*2*) *cblinfun.add-right trace-class-sandwich trace-plus*)

definition *PM* :: (*'mem* × *'l*) *update* ⇒ (*'mem* × *'l*) *tc-op* ⇒ *complex* **where**
PM Q = *PM-update Q o from-trace-class*

lemma *PM-altdef*:
PM Q ϱ = *trace-tc* (*sandwich-tc Q* ϱ)
unfolding *PM-def PM-update-def* **by** (*simp add*: *from-trace-class-sandwich-tc trace-tc.rep-eq*)

lemma *PM-linear*:
PM Q (ϱ + ψ) = *PM Q* ϱ + *PM Q* ψ
unfolding *PM-altdef* **by** (*simp add*: *sandwich-tc-plus trace-tc-plus*)

lemma *PM-sum-distr*:
PM Q (*sum f S*) = *sum* (*PM Q o f*) *S*
by (*metis PM-linear add-cancel-right-right sum-comp-morphism*)

lemma *PM-scale*:
PM Q ($a *_C \varrho$) = $a *$ *PM Q* ϱ
unfolding *PM-altdef* **by** (*simp add*: *sandwich-tc-scaleC-right trace-tc-scaleC*)

lemma *PM-case*:
PM Q (*case x of* (*H*,*S*,*z*) ⇒ *f H S*) = (*case x of* (*H*,*S*,*z*) ⇒ *PM Q* (*f H S*))
by (*simp add*: *prod.case-eq-if*)

lemma *PM-Re*:
assumes $\varrho \geq 0$

shows *Re* (*PM Q* ϱ) = *PM Q* ϱ
unfolding *PM-altdef* **by** (*simp add*: *assms complex-is-real-iff-compare0 sandwich-tc-pos trace-tc-pos*)

lemma *PM-pos*:
assumes $\varrho \geq 0$
shows *PM Q* $\varrho \geq 0$
by (*simp add*: *PM-def PM-update-def assms from-trace-class-pos sandwich-pos trace-pos*)

lemma *Re-PM-pos*:
assumes $\varrho \geq 0$
shows *Re* (*PM Q* ϱ) ≥ 0
using *PM-Re PM-pos*[*OF assms*] **by** (*simp add*: *less-eq-complex-def*)

lemma *norm-PM*:
assumes *norm* $\varrho \leq 1$ *norm* $Q \leq 1$
shows *norm* (*PM Q* ϱ) ≤ 1
proof −
have *norm* (*PM Q* ϱ) $\leq$ *norm* (*sandwich-tc* (*Q*:: ('*mem* × '*l*) *update*) ϱ)
unfolding *PM-altdef* **using** *trace-tc-norm* **by** *auto*
also have ... $\leq$ (*norm* (*Q* :: ('*mem* × '*l*) *update*))^*2* ∗ *norm* ϱ **by** (*rule norm-sandwich-tc*)
also have ... $\leq$ *norm* ϱ **using** ‹*norm* $Q \leq 1$› *mult-le-cancel-right2 power-mono* **by** *fastforce*
also have ... ≤ 1 **using** *assms* **by** *auto*
finally show *?thesis* **by** *auto*
qed

lemma *PM-bounded-linear*:
shows *bounded-linear* (*PM Q*)
unfolding *PM-altdef* **by** (*simp add*: *bounded-linear-trace-norm-sandwich-tc*)

has_sum property of PM

lemma *PM-has-sum*:
assumes (*f has-sum x*) *A*
shows (*PM Q o f has-sum PM Q x*) *A*
unfolding *o-def* **by** (*simp add*: *has-sum-bounded-linear PM-bounded-linear assms*)

10.2 Pright and Pleft'

definition *Pleft'* **where** *Pleft' Q* = *Re* (*PM Q* (*tc-tensor* (ϱ*left E*) *empty-tc*))

definition *Pleft* **where** *Pleft Q* = *Re* (*trace-tc* (*sandwich-tc Q* (ϱ*left E*)))

lemma *trace-tensor-tc*:
trace-tc (*tc-tensor a b*) = *trace-tc a* ∗ *trace-tc b*
by (*simp add*: *tc-tensor.rep-eq trace-tc.rep-eq trace-tensor*)

lemma *Pleft-Pleft'*:
assumes *sandwich-tc A empty-tc* = *tc-selfbutter* (*ket empty*)
shows *Pleft Q* = *Pleft'* ($Q \otimes_o A$)

proof −
  from *assms* **have** *trace-tc* (*sandwich-tc* Q (*ϱleft* E)) =
    trace-tc (*sandwich-tc* ($Q \otimes_o A$) (*tc-tensor* (*ϱleft* E) *empty-tc*))
   by (*metis empty-tc-def empty-tc-pos from-trace-class-inverse mult-cancel-left1 norm-empty-tc*

      norm-tc-pos of-real-1 sandwich-tc-tensor tc-butterfly.rep-eq tc-selfbutter-def trace-tensor-tc)
  then show *?thesis* **unfolding** *Pleft-def Pleft′-def PM-altdef* **by** *auto*
qed

lemma *Pleft-Pleft′-empty*:
  Pleft Q = *Pleft′* ($Q \otimes_o$ *selfbutter* (*ket empty*))
proof −
  have *sandwich-tc* (*selfbutter* (*ket empty*)) *empty-tc* = *tc-selfbutter* (*ket empty*)
    unfolding *empty-tc-def tc-selfbutter-def sandwich-tc-def tc-butterfly-def compose-tcl-def*
      compose-tcr-def **by** (*auto simp add*: *Abs-trace-class-inverse*)
  then show *?thesis* **using** *Pleft-Pleft′* **by** *auto*
qed

lemma *Pleft-Pleft′-id*:
  Pleft Q = *Pleft′* ($Q \otimes_o$ *id-cblinfun*)
proof −
  have *sandwich-tc* (*id-cblinfun*) *empty-tc* = *tc-selfbutter* (*ket empty*)
    unfolding *empty-tc-def tc-selfbutter-def sandwich-tc-def tc-butterfly-def compose-tcl-def*
      compose-tcr-def **by** (*auto simp add*: *Abs-trace-class-inverse*)
  then show *?thesis* **using** *Pleft-Pleft′* **by** *auto*
qed

lemma *Pleft-Pleft′-case5*:
  assumes *is-Proj* Q
  shows *Pleft* Q = *Pleft′* ($Q \otimes_o$ *selfbutter* (*ket empty*) + *end-measure*)
proof −
  define *Set* **where** *Set* = ($Q \otimes_o$ *id-cblinfun*) $*_S$ ⊤ ⊔
    (*id-cblinfun* $\otimes_o$ (*id-cblinfun* − *selfbutter* (*ket empty*))) $*_S$ ⊤
  have *rew*: $Q \otimes_o$ *selfbutter* (*ket empty*) + *end-measure* = *Proj* (*Set*) **unfolding** *Set-def*
   by (*subst splitting-Proj-or*) (*auto simp add*: *butterfly-is-Proj assms end-measure-def Snd-def*
      is-Proj-end-measure)
  have (*id-cblinfun* − *selfbutter* (*ket empty*)) o_{CL} *selfbutter* (*ket empty*) = *0*
   by (*simp add*: *cblinfun-compose-minus-left*)
  then have *zero*: *compose-tcr end-measure* (*tc-tensor* (*ϱleft* E) *empty-tc*) = *0*
    unfolding *compose-tcr-def empty-tc-def end-measure-def Snd-def*
   by (*auto simp add*: *Abs-trace-class-inverse comp-tensor-op tc-tensor.rep-eq zero-trace-class.abs-eq*)
  have *trace-tc* (*sandwich-tc* ($Q \otimes_o$ *selfbutter* (*ket empty*) + *end-measure*)
    (*tc-tensor* (*ϱleft* E) *empty-tc*)) =
    trace-tc (*compose-tcr* ($Q \otimes_o$ *selfbutter* (*ket empty*) + *end-measure*)
    (*tc-tensor* (*ϱleft* E) *empty-tc*)) **unfolding** *rew sandwich-tc-def trace-tc.rep-eq*
    compose-tcl.rep-eq compose-tcr.rep-eq
   by (*metis* (*no-types*, *lifting*) *Proj-idempotent adj-Proj cblinfun-assoc-left*(*1*) *circularity-of-trace*

compose-tcr.rep-eq trace-class-from-trace-class)
also have ... = *trace-tc* (*compose-tcr* ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*tc-tensor* (*ϱleft E*) *empty-tc*) +
compose-tcr end-measure (*tc-tensor* (*ϱleft E*) *empty-tc*))
by (*simp add*: *compose-tcr.add-left*)
also have ... = *trace-tc* (*compose-tcr* ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*tc-tensor* (*ϱleft E*) *empty-tc*))
using *zero* **by** *auto*
also have ... = *trace-tc* (*sandwich-tc* ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*tc-tensor* (*ϱleft E*) *empty-tc*))
unfolding *sandwich-tc-def trace-tc.rep-eq compose-tcl.rep-eq compose-tcr.rep-eq*
by (*metis* (*no-types*, *lifting*) *assms butterfly-is-Proj cblinfun-assoc-left*(*1*) *circularity-of-trace*

compose-tcr.rep-eq is-Proj-algebraic is-Proj-tensor-op norm-ket trace-class-from-trace-class)
finally have *trace-tc* (*sandwich-tc* ($Q \otimes_o$ *selfbutter* (*ket empty*) + *end-measure*)
(*tc-tensor* (*ϱleft E*) *empty-tc*)) =
trace-tc (*sandwich-tc* ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*tc-tensor* (*ϱleft E*) *empty-tc*))
by *auto*
then have *Pleft′* ($Q \otimes_o$ *selfbutter* (*ket empty*) + *end-measure*) = *Pleft′* ($Q \otimes_o$ *selfbutter* (*ket empty*))
unfolding *Pleft′-def PM-altdef* **by** *auto*
then show *?thesis* **using** *Pleft-Pleft′-empty* **by** *auto*
qed

definition *Pright* **where** *Pright Q* = *Re* (*PM Q* (*ϱright E*))

lemma *Re-PM-left-has-sum*:
((λ*F*. *Re* (*PM Q* (*tc-tensor* (*ϱleft F*) *empty-tc*))) *has-sum Pleft′ Q*)
(*finite-kraus-subadv E d*)
unfolding *Pleft′-def*
using *Re-has-sum*[*OF PM-has-sum*[*OF tc-tensor-has-sum* [*of* - - - *empty-tc*, *OF ϱleft-has-sum*]]]
PM-pos[*OF tc-tensor-pos*[*OF ϱleft-pos empty-tc-pos*]] **unfolding** *comp-def* **by** *auto*

lemma *Re-PM-right-has-sum*:
((λ*F*. *Re* (*PM Q* (*ϱright F*))) *has-sum Pright Q*) (*finite-kraus-subadv E d*)
unfolding *Pright-def*
using *Re-has-sum*[*OF PM-has-sum*[*OF ϱright-has-sum*]] *PM-pos*[*OF ϱright-pos*] **by** (*auto simp add*: *o-def*)

10.3 Pfind

The definition of the find event

definition *Pfind-update* ::
(*nat* ⇒ ′*mem update*) ⇒ (′*x* ⇒ ′*y*) ⇒ (′*x* ⇒ *bool*) ⇒ *complex* **where**
Pfind-update UA H S = *trace* (*end-measure* o_{CL} (*run-pure-B-update UA H S*))

definition *Pfind-pure* :: (*nat* ⇒ *'mem update*) ⇒ *complex* **where**
Pfind-pure UA = *trace-tc* (*compose-tcr end-measure* (*ϱright-pure UA*))

definition *Pfind* :: *'mem kraus-adv* ⇒ *complex* **where**
Pfind F = *trace-tc* (*compose-tcr end-measure* (*ϱright F*))

lemma *Pfind-altdef*:
Pfind E = *Pright end-measure*
unfolding *Pfind-def Pright-def PM-altdef*
by (*smt* (*verit*) *ϱright-pos cblinfun-assoc-left*(*1*) *circularity-of-trace complex-is-real-iff-compare0*

compose-tcr.rep-eq from-trace-class-sandwich-tc is-Proj-algebraic is-Proj-end-measure of-real-Re

sandwich-apply sandwich-tc-pos trace-class-from-trace-class trace-tc.rep-eq trace-tc-pos)

lemma *Pfind-Pright*:
Re (*Pfind E*) = *Pright end-measure*
unfolding *Pfind-altdef* **by** *auto*

Write mixed in pure states, pure in updates and connect updates to *pure-o2h* version.

lemma *Re-Pfind-update-altdef*:
assumes $\bigwedge$*i*. *i*<*d*+*1* ⟹ *norm* (*UA i*) ≤ *1*
shows *Re* (*Pfind-update UA H S*) = *pure-o2h.Pfind' X Y d init flip empty H S UA*
proof −
interpret *pure*: *pure-o2h X Y d init flip bit valid empty H S UA*
by *unfold-locales* (*auto simp add*: *assms*)
have *B*: *run-pure-B-update UA H S* = *selfbutter* (*pure.run-B*)
unfolding *run-pure-B-ell2-update*
by (*simp add*: *Fst-def o-def pure.run-B-def run-pure-B-ell2-def*)
show *?thesis* **unfolding** *Pfind-update-def pure.Pfind'-def B trace-selfbutter-norm trace-end-measure*
by (*auto simp add*: *end-measure-def*)
qed

lemma *Pfind-pure-update*:
Pfind-pure UA = ($\sum$(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Pfind-update UA H S*)
proof −
have *Pfind-pure UA* = *trace-tc* ($\sum$*x*∈*carrier*. *compose-tcr end-measure*
(*case x of* (*H*,*S*,*z*) ⇒ *complex-of-real* (*distr* (*H*,*S*,*z*)) $*_C$ *run-pure-B-tc UA H S*))
unfolding *Pfind-pure-def ϱright-pure-def*
by (*auto simp add*: *compose-tcr.sum-right*)
also have ... = *trace* ($\sum$*x*∈*carrier*. *end-measure* o_{CL} *from-trace-class*
(*case x of* (*H*,*S*,*z*) ⇒ *complex-of-real* (*distr* (*H*,*S*,*z*)) $*_C$ *run-pure-B-tc UA H S*))
unfolding *trace-tc.rep-eq from-trace-class-sum compose-tcr.rep-eq* **by** *auto*
also have ... = ($\sum$*i*∈*carrier*. *trace* (*end-measure* o_{CL} *from-trace-class*
(*case i of* (*H*,*S*,*z*) ⇒ *complex-of-real* (*distr* (*H*,*S*,*z*)) $*_C$ *run-pure-B-tc UA H S*)))
by (*subst trace-sum*) (*auto simp add*: *trace-class-end-measure*)

also have ... = ($\sum$ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Pfind-update UA H S*)
  unfolding *Pfind-update-def* **by** (*intro sum.cong*) (*auto simp add*: *Abs-trace-class-inverse run-pure-B-ell2-update run-pure-B-update-tc scaleC-trace-class.rep-eq trace-scaleC*)
finally show *?thesis* **by** *auto*
qed

lemma *Pfind-pure-mixed*:
  assumes $\bigwedge$*i*. *i* < *d* + *1* ⟹ *finite* (*Rep-kraus-family* (*F i*))
  $\bigwedge$*i*. *i* < *d* + *1* ⟹ *fst* ‘ *Rep-kraus-family* (*F i*) ≠ {}
  shows *Pfind F* = ($\sum$ *UA*∈*purify-comp-kraus d F*. *Pfind-pure UA*)
proof −
  have *run*: *ϱright F* = *sum ϱright-pure* (*purify-comp-kraus d F*)
    using *ϱright-pure-mixed assms* **by** *auto*
  show *?thesis* **unfolding** *Pfind-def Pfind-pure-def run*
    by (*auto simp add*: *compose-tcr.sum-right trace-tc-sum*)
qed

Pfind positivity

lemma *Pfind-pure-pos*:
  Pfind-pure UA ≥ *0*
proof −
  have *Pfind-pure UA* = *trace-tc* ($\sum$ *i*∈*carrier*. *compose-tcr end-measure* (*case i of* (*H*,*S*,*z*) ⇒ (*distr* (*H*,*S*,*z*)) $*_C$ *run-pure-B-tc UA H S*))
    unfolding *Pfind-pure-def ϱright-pure-def* **by** (*auto simp add*: *compose-tcr.sum-right*)
  also have ... = ($\sum$ *i*∈*carrier*. *trace-tc* (*compose-tcr end-measure* (*case i of* (*H*,*S*,*z*) ⇒ (*distr* (*H*,*S*,*z*)) $*_C$ *run-pure-B-tc UA H S*)))
    by (*intro trace-tc-sum*)
  also have ... = ($\sum$ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) $*_C$ *trace-tc* (*compose-tcr end-measure* (*run-pure-B-tc UA H S*)))
    by (*intro sum.cong*, *auto simp add*: *compose-tcr.scaleC-right trace-tc-scaleC*)
  also have ... ≥ *0* **by** (*intro sum-nonneg*)
    (*use distr-pos run-pure-B-tc-pos trace-endmeasure-pos* **in** ‹*fastforce*›)
  finally show *?thesis* **by** *linarith*
qed

lemma *Pfind-pos*:
  Pfind F ≥ *0* **by** (*simp add*: *Pfind-def ϱright-pos trace-endmeasure-pos*)

Pfind is already real

lemma *Re-Pfind-update*:
  Re (*Pfind-update UA H S*) = *Pfind-update UA H S*
  unfolding *Pfind-update-def*
  by (*simp add*: *run-pure-B-ell2-update trace-end-measure*)

lemma *Re-Pfind-pure*:
  Re (*Pfind-pure UA*) = *Pfind-pure UA*
  using *Pfind-pure-pos complex-eq-iff less-eq-complex-def* **by** *auto*

lemma *Re-Pfind*:

Re (*Pfind F*) = *Pfind F*
unfolding *Pfind-def*
using *ϱright-pos complex-eq-iff less-eq-complex-def trace-endmeasure-pos* **by** *force*

Pfind, (*has-sum*), and (*summable-on*) properties

lemma *Pfind-abs-summable-on*:
Pfind abs-summable-on (*finite-kraus-subadv E d*)
proof −
have *ϱright abs-summable-on* (*finite-kraus-subadv E d*) **using** *ϱright-abs-summable* **by** *auto*
then obtain *M* **where** *M*: *sum* (*λx. trace-tc* (*ϱright x*)) *F* ≤ *complex-of-real M*
if *F* ⊆ *finite-kraus-subadv E d finite F* **for** *F*
unfolding *norm-tc-pos*[*OF ϱright-pos*, *symmetric*] *of-real-sum*[*symmetric*]
abs-summable-iff-bdd-above bdd-above-def
by (*metis* (*no-types*, *lifting*) *bdd-above-def cSUP-upper complex-of-real-mono mem-Collect-eq*)
show *?thesis*
proof (*unfold Pfind-def abs-summable-iff-bdd-above*, *intro bdd-aboveI*[*of - M*])
fix *x* **assume** *x* ∈ *sum* (*λx. cmod* (*trace-tc* (*compose-tcr end-measure* (*ϱright x*)))) ‘
{*F*. *F* ⊆ *finite-kraus-subadv E d* ∧ *finite F*}
then obtain *F* **where** *assm*: *F* ⊆ *finite-kraus-subadv E d finite F*
and *x*: *x* = *sum* (*λx. cmod* (*trace-tc* (*compose-tcr end-measure* (*ϱright x*)))) *F*
by *auto*
have *x* ≤ *sum* (*λx. norm* (*end-measure*) ∗ (*trace-tc* (*ϱright x*))) *F*
unfolding *x* **using** *cmod-trace-times′* **by** (*subst of-real-sum*, *intro sum-mono*)
(*smt* (*verit*, *ccfv-threshold*) *ϱright-pos complex-of-real-mono norm-compose-tcr norm-tc-pos*

of-real-mult trace-tc-norm)
also have . . . ≤ *1* ∗ *sum* (*λx. trace-tc* (*ϱright x*)) *F*
by (*subst sum-distrib-left*[*symmetric*]) (*meson ϱright-pos complex-of-real-leq-1-iff mult-right-mono*

norm-end-measure sum-nonneg trace-tc-pos)
also have . . . ≤ *M* **using** *M*[*OF assm*] **by** (*simp add*: *trace-tc.rep-eq*)
finally show *x* ≤ *M* **by** *auto*
qed
qed

lemma *Pfind-summable-on*:
Pfind summable-on (*finite-kraus-subadv E d*)
using *Pfind-abs-summable-on abs-summable-summable* **by** *blast*

lemma *Pfind-has-sum*:
((*λF. Pfind F*) *has-sum Pfind E*) (*finite-kraus-subadv E d*)
proof −
have *lin*: *bounded-linear* (*λx. trace-tc* (*compose-tcr end-measure x*))
by (*simp add*: *bounded-clinear.bounded-linear bounded-linear-compose compose-tcr.real.bounded-linear-right*)
show *?thesis* **unfolding** *Pfind-def* **using** *ϱright-has-sum* **by** (*auto intro*!: *has-sum-bounded-linear*[*OF lin*])
qed

10.4 Nontermination Part

This introduces the non-termination part needed for pure o2h with *norm UA* $\leq$ *1*.

definition *P-nonterm-update*::$('x \Rightarrow 'y) \Rightarrow ('x \Rightarrow bool) \Rightarrow (nat \Rightarrow 'mem\ update) \Rightarrow real$ **where**
P-nonterm-update H S UA =
Re (*trace* (*run-pure-B-count-update UA H S*) $-$ *trace* (*run-pure-B-update UA H S*))

definition *P-nonterm-pure*::$(nat \Rightarrow 'mem\ ell2 \Rightarrow_{CL} 'mem\ ell2) \Rightarrow real$ **where**
P-nonterm-pure UA = *Re* (*trace-tc* (ϱ*count-pure UA*) $-$ *trace-tc* (ϱ*right-pure UA*))

definition *P-nonterm* :: *'mem kraus-adv* $\Rightarrow$ *real* **where**
P-nonterm F = *Re* (*trace-tc* (ϱ*count F*) $-$ *trace-tc* (ϱ*right F*))

Connecting mixed with pure, pure with updates and updates with *pure-o2h* version.

lemma *P-nonterm-update-altdef*:
assumes $\bigwedge i.\ i<d+1 \Longrightarrow norm\ (UA\ i) \leq 1$
shows *P-nonterm-update H S UA* = *pure-o2h.P-nonterm X Y d init flip empty H S UA*
proof $-$
interpret *pure*: *pure-o2h X Y d init flip bit valid empty H S UA*
by *unfold-locales* (*auto simp add*: *assms*)
have *Bcount*: *run-pure-B-count-update UA H S* = *selfbutter* (*pure.run-B-count*)
unfolding *run-pure-B-count-ell2-update*
by (*simp add*: *Fst-def o-def pure.run-B-count-def run-pure-B-count-ell2-def*)
have *B*: *run-pure-B-update UA H S* = *selfbutter* (*pure.run-B*)
unfolding *run-pure-B-ell2-update*
by (*simp add*: *Fst-def o-def pure.run-B-def run-pure-B-ell2-def*)
show *?thesis* **unfolding** *P-nonterm-update-def pure.P-nonterm-def Bcount B trace-selfbutter-norm*
by *auto*
qed

lemma *P-nonterm-pure-update*:
P-nonterm-pure UA = ($\sum$ $(H,S,z) \in carrier$. *distr* (H,S,z) $*$ *P-nonterm-update H S UA*)
proof $-$
have *1*: *trace-tc* (*run-pure-B-count-tc UA a b*) = *trace* (*from-trace-class* (*run-pure-B-count-tc UA a b*))
for *a b* **by** (*simp add*: *trace-tc.rep-eq*)
have *2*: *trace* (*from-trace-class* (*run-pure-B-tc UA a b*)) = *trace-tc* (*run-pure-B-tc UA a b*)
for *a b* **by** (*simp add*: *trace-tc.rep-eq*)
show *?thesis* **unfolding** *P-nonterm-pure-def* ϱ*count-pure-def* ϱ*right-pure-def P-nonterm-update-def*

by (*subst trace-tc-sum*, *subst trace-tc-sum*) (*auto simp add*: *case-prod-beta sum-subtractf*[*symmetric*] *trace-tc-scaleC algebra-simps run-pure-B-update-tc' run-pure-B-count-update-tc' 1 2 intro*!: *sum.cong*)
qed

lemma *P-nonterm-purification*:
assumes $\bigwedge i.\ i < d + 1 \Longrightarrow$ *finite* (*Rep-kraus-family* (*F i*))

$\bigwedge i.\ i < d + 1 \Longrightarrow$ *Rep-kraus-family* (*F i*) $\neq$ {}
shows *P-nonterm F* = ($\sum$ *UA*∈*purify-comp-kraus d F*. *P-nonterm-pure UA*)
proof −
have *r*: *ϱright F* = *sum ϱright-pure* (*purify-comp-kraus d F*)
using *ϱright-pure-mixed assms* **by** *auto*
have *c*: *ϱcount F* = *sum ϱcount-pure* (*purify-comp-kraus d F*)
using *ϱcount-pure-mixed assms* **by** *auto*
show *?thesis* **unfolding** *P-nonterm-def P-nonterm-pure-def* **using** *r*[*symmetric*] *c*[*symmetric*]
by (*auto simp add*: *sum-subtractf Re-sum*[*symmetric*] *trace-tc-sum*[*symmetric*] *simp del*: *Re-sum*)
qed

Positive error term

lemma *error-term-update-pos*:
assumes $\bigwedge i.\ i<d+1 \Longrightarrow$ *norm* (*UA i*) $\leq$ *1*
shows *0* $\leq$ (*d+1*) $*$ *Re* (*Pfind-update UA H S*) + *d* $*$ (*P-nonterm-update H S UA*)
proof −
interpret *pure*: *pure-o2h X Y d init flip bit valid empty H S UA*
by *unfold-locales* (*auto simp add*: *assms*)
have (*d+1*) $*$ *Re* (*Pfind-update UA H S*) + *d* $*$ (*P-nonterm-update H S UA*) =
(*d+1*) $*$ (*pure.Pfind′*) + *d* $*$ (*pure.P-nonterm*)
using *Re-Pfind-update-altdef P-nonterm-update-altdef assms* **by** *auto*
also have ... $\geq$ *0* **using** *pure.error-term-pos* **by** *auto*
finally show *?thesis* **by** *auto*
qed

lemma *error-term-pure-pos*:
assumes $\bigwedge i.\ i<d+1 \Longrightarrow$ *norm* (*UA i*) $\leq$ *1*
shows *0* $\leq$ (*d+1*) $*$ *Re* (*Pfind-pure UA*) + *d* $*$ (*P-nonterm-pure UA*)
proof −
have (*d+1*) $*$ *Re* (*Pfind-pure UA*) + *d* $*$ (*P-nonterm-pure UA*) = ($\sum$ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) $*$
((*d+1*) $*$ *Re* (*Pfind-update UA H S*) + *d* $*$ (*P-nonterm-update H S UA*)))
unfolding *P-nonterm-pure-update Pfind-pure-update*
unfolding *sum-distrib-left Re-sum sum.distrib*[*symmetric*]
by (*intro sum.cong*)(*auto simp add*: *algebra-simps*)
also have ... $\geq$ *0* **by** (*intro sum-nonneg*, *use error-term-update-pos assms distr-pos′* **in** ‹*auto*›)
finally show *?thesis* **by** *auto*
qed

lemma *error-term-pos*:
assumes *finite*: $\bigwedge i.\ i < d+1 \Longrightarrow$ *finite* (*Rep-kraus-family* (*F i*))
and *F-norm-id*:$\bigwedge i.\ i < d+1 \Longrightarrow$ *kf-bound* (*F i*) $\leq$ *id-cblinfun*
and *F-nonzero*: $\bigwedge i.\ i < d+1 \Longrightarrow$ *Rep-kraus-family* (*F i*) $\neq$ {}
shows *0* $\leq$ (*d* + *1*) $*$ *Re* (*Pfind F*) + *d* $*$ *P-nonterm F*
proof −
have (*d* + *1*) $*$ *Re* (*Pfind F*) + *d* $*$ *P-nonterm F* =
($\sum$ *UA*∈*purify-comp-kraus d F*. (*d+1*) $*$ *Re* (*Pfind-pure UA*)) + *d* $*$ *P-nonterm F*
using *assms* **by** (*subst Pfind-pure-mixed*) (*auto simp add*: *sum-distrib-left*)

also have ... $= (\sum$ *UA*$\in$*purify-comp-kraus d F*. $(d+1) * $ *Re* (*Pfind-pure UA*) $+ d *$ (*P-nonterm-pure UA*))
  using *assms* **by** (*subst P-nonterm-purification*) (*auto simp add*: *sum-distrib-left*)
 also have ... ≥ 0 **by** (*intro sum-nonneg*)
   (*use error-term-pure-pos norm-in-purify-comp-kraus*[**where** *n*=*d*] *assms* **in** ‹*auto*›)
 finally show *?thesis* **by** *auto*
qed

has sum property

lemma *P-nonterm-has-sum*:
 ((λF. *P-nonterm F*) *has-sum P-nonterm E*) (*finite-kraus-subadv E d*)
proof −
 have *lin*: *bounded-linear* (λx. *trace-tc x*)
  by (*simp add*: *bounded-clinear.bounded-linear*)
 show *?thesis* **unfolding** *P-nonterm-def* **using** *ϱright-has-sum ϱcount-has-sum*
  by (*auto intro*!: *has-sum-Re has-sum-diff has-sum-bounded-linear*[*OF lin*])
qed

11 Proof of Mixed O2H

We prove the mixed O2H in several steps.

Step 1: Connect the updates version to the *pure-o2h* lemma

lemma *estimate-Pfind-update-sqrt*:
 fixes *UA H S*
 assumes $\bigwedge i.\ i<d+1 \Longrightarrow$ *norm* (*UA i*) ≤ 1
  and *norm-Q*: *norm* $Q \leq 1$
 shows |*sqrt* (*Re* (*PM-update Q* ((*run-pure-A-update UA H*) $\otimes_o$ (*selfbutter* (*ket empty*))))) −
    sqrt (*Re* (*PM-update Q* (*run-pure-B-update UA H S*)))|
   $\leq$ *sqrt* (($d+1$) $*$ *Re* (*Pfind-update UA H S*) $+ d *$ *P-nonterm-update H S UA*)
proof −
 interpret *pure*: *pure-o2h X Y d init flip bit valid empty H S UA*
  by *unfold-locales* (*auto simp add*: *assms*)
 have *pure-A*: *run-pure-A-ell2 UA H* = *pure.run-A*
  unfolding *pure.run-A-def run-pure-A-ell2-def* **by** *auto*
 have *pure-B*: *run-pure-B-ell2 UA H S* = *pure.run-B*
  unfolding *pure.run-B-def run-pure-B-ell2-def Fst-def comp-def* **by** *auto*
 have *pure-find*: *Pfind-update UA H S* = *pure.Pfind′*
  unfolding *Pfind-update-def pure.Pfind′-def end-measure-def*[*symmetric*]
  unfolding *run-pure-B-ell2-update pure-B* **using** *trace-end-measure* **by** *auto*
 have *1*: |*sqrt* (*Re* (*PM-update Q* (*run-pure-A-update UA H* $\otimes_o$ *selfbutter* (*ket empty*)))) −
   sqrt (*Re* (*PM-update Q* (*run-pure-B-update UA H S*)))| =
  |*sqrt* (*Re* (*trace* (*sandwich Q* (*selfbutter* (*run-pure-A-ell2 UA H* $\otimes_s$ *ket empty*))))) −
   sqrt (*Re* (*trace* (*sandwich Q* (*selfbutter* (*run-pure-B-ell2 UA H S*)))))|
  unfolding *PM-update-def* **by** (*simp add*: *run-pure-A-ell2-update run-pure-B-ell2-update tensor-butterfly*)
 also have *2*: ... = |*sqrt* (*Re* (*trace* (*selfbutter* ($Q *_V$ (*run-pure-A-ell2 UA H* $\otimes_s$ *ket empty*)))))
−

sqrt (*Re* (*trace* (*selfbutter* ($Q *_V$ (*run-pure-B-ell2 UA H S*)))))|
by (*simp add*: *selfbutter-sandwich*)
also have *3*: ... = |(*norm* ($Q *_V$ (*run-pure-A-ell2 UA H* $\otimes_s$ *ket empty*))) −
norm ($Q *_V$ (*run-pure-B-ell2 UA H S*))|
unfolding *trace-butterfly power2-norm-eq-cinner*[*symmetric*] **by** (*simp add*: *norm-power*)
also have *5*: ... ≤ *norm* ($Q *_V$ (*run-pure-A-ell2 UA H* $\otimes_s$ *ket empty*) −
$Q *_V$ (*run-pure-B-ell2 UA H S*)) **by** (*simp add*: *norm-triangle-ineq3*)
also have ... = *norm* ($Q *_V$ (*run-pure-A-ell2 UA H* $\otimes_s$ *ket empty* − *run-pure-B-ell2 UA H S*))
by (*subst cblinfun.diff-right*) *auto*
also have ... ≤ *norm* (*Q*::(′*mem*×′*l*)*update*) ∗
norm (*run-pure-A-ell2 UA H* $\otimes_s$ *ket empty* − *run-pure-B-ell2 UA H S*)
by (*rule norm-cblinfun*)
also have ... ≤ *norm* (*run-pure-A-ell2 UA H* $\otimes_s$ *ket empty* − *run-pure-B-ell2 UA H S*)
using *norm-Q* **by** (*metis mult-le-cancel-right2 norm-not-less-zero*)
also have ... ≤ (*sqrt* (*real* (*d* + *1*) ∗ *pure.Pfind*′ + *real d* ∗ *pure.P-nonterm*))
unfolding *pure-A pure-B* **using** *pure.pure-o2h-sqrt* **by** *auto*
also have ... ≤ *sqrt* ((*d*+*1*) ∗ *Re* (*Pfind-update UA H S*) + *d* ∗ *pure.P-nonterm*)
unfolding *pure-find* **by** *simp*
also have ... ≤ *sqrt* ((*d* + *1*) ∗ *Re* (*Pfind-update UA H S*) + (*d* ∗ *P-nonterm-update H S UA*))
by (*subst P-nonterm-update-altdef*[*OF assms*(*1*)], *auto*)
finally show *?thesis* **using** *1 2 3 5* **by** *linarith*
qed

lemma *estimate-Pfind-tc-sqrt*:
fixes *UA H S*
assumes ⋀*i*. *i*<*d*+*1* ⟹ *norm* (*UA i*) ≤ *1 norm Q* ≤ *1*
shows |*sqrt* (*Re* (*PM Q* (*tc-tensor* (*run-pure-A-tc UA H*) *empty-tc*))) −
sqrt (*Re* (*PM Q* (*run-pure-B-tc UA H S*)))|
≤ *sqrt* ((*d*+*1*) ∗ *Re* (*Pfind-update UA H S*) + *d* ∗ (*P-nonterm-update H S UA*))
using *estimate-Pfind-update-sqrt*[*OF assms*] **unfolding** *empty-tc-def*
by (*smt* (*verit*) *PM-def assms*(*1*) *assms*(*2*) *from-trace-class-inverse estimate-Pfind-update-sqrt*

run-pure-B-tc-def run-pure-B-update-def o-def run-pure-A-tc-def run-pure-A-update-def
run-pure-adv-update-tc′ *tc-butterfly.rep-eq tc-tensor.rep-eq*)

Step 2: Connect the pure version with the update version by summation over the distribution of H and S

lemma *estimate-Pfind-pure-sqrt*:
fixes *UA*
assumes ⋀*i*. *i*<*d*+*1* ⟹ *norm* (*UA i*) ≤ *1 norm Q* ≤ *1*
shows |*sqrt* (*Re* (*PM Q* (*tc-tensor* (*ϱleft-pure UA*) *empty-tc*))) − *sqrt* (*Re* (*PM Q* (*ϱright-pure UA*)))|
≤ *sqrt* (*real* (*d* + *1*) ∗ *Re* (*Pfind-pure UA*) + *d* ∗ *P-nonterm-pure UA*)
proof −
let *?PMA* = (*λH*. *PM Q* (*tc-tensor* (*run-pure-A-tc UA H*) *empty-tc*))

let *?PMB* = (λ*H S*. *PM Q* (*run-pure-B-tc UA H S*))
have |*sqrt* (*Re* (*PM Q* (*tc-tensor* (*ϱleft-pure UA*) *empty-tc*))) − *sqrt* (*Re* (*PM Q* (*ϱright-pure UA*)))| =
|*sqrt* ($\sum$(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Re* (*?PMA H*)) −
sqrt ($\sum$(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Re* (*?PMB H S*))|
proof −
have *zeroA*: *0* ≤ *tc-tensor* (*run-pure-A-tc UA H*) *empty-tc* **for** *H*
by (*intro tc-tensor-pos*[*OF run-pure-A-tc-pos empty-tc-pos*])
have *Re* (*PM Q* (*tc-tensor* (*ϱleft-pure UA*) *empty-tc*)) = *Re* (*PM Q* ($\sum$ *i*∈*carrier*.
(*distr i*) $*_C$ *tc-tensor* (*run-pure-A-tc UA* (*fst i*)) *empty-tc*))
unfolding *ϱleft-pure-def*
by (*auto simp add*: *tc-tensor-scaleC-left tc-tensor-sum-left case-prod-beta*)
also have ... = *Re* ($\sum$ *x*∈*carrier*. (*distr x*) ∗ *PM Q* (*tc-tensor* (*run-pure-A-tc UA* (*fst x*)) *empty-tc*))
by (*subst PM-sum-distr*)+ (*auto simp add*: *prod.case-distrib comp-def PM-scale algebra-simps*)
also have ... = ($\sum$(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Re* (*?PMA H*))
by (*subst Re-sum*)
(*auto simp add*: *distr-pos′ PM-pos*[*OF zeroA*] *norm-mult algebra-simps intro*!: *sum.cong*)
finally have *1*: *Re* (*PM Q* (*tc-tensor* (*ϱleft-pure UA*) *empty-tc*)) =
($\sum$(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Re* (*?PMA H*)) **by** *auto*
have *Re* (*PM Q* (*ϱright-pure UA*)) = *Re* (*PM Q* ($\sum$ *i*∈*carrier*. (*distr i*) $*_C$ *run-pure-B-tc UA* (*fst i*) (*fst* (*snd i*))))
unfolding *ϱright-pure-def*
by (*auto simp add*: *tc-tensor-scaleC-left tc-tensor-sum-left case-prod-beta*)
also have ... = *Re* ($\sum$ *x*∈*carrier*. (*distr x*) ∗ *PM Q* (*run-pure-B-tc UA* (*fst x*) (*fst* (*snd x*))))
by (*subst PM-sum-distr*)+ (*auto simp add*: *prod.case-distrib comp-def PM-scale algebra-simps*)
also have ... = ($\sum$(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Re* (*?PMB H S*))
by (*subst Re-sum*) (*auto simp add*: *distr-pos′ PM-pos*[*OF run-pure-B-tc-pos*]
norm-mult algebra-simps intro!: *sum.cong*)
finally have *2*: *Re* (*PM Q* (*ϱright-pure UA*)) = ($\sum$(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Re* (*?PMB H S*))
by *auto*
show *?thesis* **unfolding** *1 2* **by** *auto*
qed
also have ... ≤ *sqrt* ($\sum$(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗
((*d*+*1*) ∗ *Re* (*Pfind-update UA H S*) + *d* ∗ (*P-nonterm-update H S UA*)))
proof −
have *ass1*: ∀ *x*∈*carrier*. *0* ≤ (*case x of* (*H*,*S*,*z*) ⇒ *Re* (*?PMA H*))
by (*metis* (*mono-tags*, *lifting*) *PM-pos cmod-Re empty-tc-pos norm-ge-zero prod.case-eq-if run-pure-A-tc-pos tc-tensor-pos*)
have *ass2*: ∀ *x*∈*carrier*. *0* ≤ (*case x of* (*H*,*S*,*z*) ⇒ *Re* (*?PMB H S*))
by (*metis* (*mono-tags*, *lifting*) *PM-pos cmod-Re norm-ge-zero prod.case-eq-if run-pure-B-tc-pos*)
have *ass3*: ∀ *x*∈*carrier*. *0* ≤ (*case x of* (*H*,*S*,*z*) ⇒ (*d* + *1*) ∗ *Re* (*Pfind-update UA H S*) +
d ∗ (*P-nonterm-update H S UA*))
using *assms*(*1*) *error-term-update-pos* **by** *auto*
have *ass4*: ∀ *x*∈*carrier*. *0* ≤ *distr x* **using** *distr-pos* **by** *fastforce*

have *ass5*: ∀ *x*∈*carrier*. |*sqrt* (*case x of* (*H*,*S*,*z*) ⇒ *Re* (*?PMA H*)) −
sqrt (*case x of* (*H*,*S*,*z*) ⇒ *Re* (*?PMB H S*))|
≤ *sqrt* (*case x of* (*H*,*S*,*z*) ⇒ *real* (*d* + *1*) ∗ *Re* (*Pfind-update UA H S*) + *d* ∗ (*P-nonterm-update H S UA*))
using *estimate-Pfind-tc-sqrt*[*OF assms*] **by** *auto*
have *rew-sum1*:
(∑ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Re* (*?PMA H*)) =
(∑ *x*∈*carrier*. *distr x* ∗ (*case x of* (*H*,*S*,*z*) ⇒ *Re* (*?PMA H*)))
by (*auto intro*: *sum.cong*)
have *rew-sum2*: (∑ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Re* (*?PMB H S*)) =
(∑ *x*∈*carrier*. *distr x* ∗ (*case x of* (*H*,*S*,*z*) ⇒ *Re* (*?PMB H S*))) **by** (*auto intro*: *sum.cong*)
have *rew-sum3*: (∑ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗
(*real* (*d* + *1*) ∗ *Re* (*Pfind-update UA H S*) + *d* ∗ (*P-nonterm-update H S UA*))) =
(∑ *x*∈*carrier*. *distr x* ∗ (*case x of* (*H*,*S*,*z*) ⇒ *real* (*d* + *1*) ∗ *Re* (*Pfind-update UA H S*) +
real d ∗ (*P-nonterm-update H S UA*))) **by** (*auto intro*!: *sum.cong*)
show *?thesis* **by** (*unfold rew-sum1 rew-sum2 rew-sum3*)
(*rule sqrt-estimate-real*[*OF finite-carrier ass1 ass2 ass3 ass4 ass5*])
qed
also have … ≤ *sqrt* (*real* (*d* + *1*) ∗ *Re* (*Pfind-pure UA*) + *d* ∗ *P-nonterm-pure UA*)
proof −
have (∑ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ ((*d* + *1*) ∗ *Re* (*Pfind-update UA H S*) +
d ∗ *P-nonterm-update H S UA*)) = (∑ (*H*,*S*,*z*)∈*carrier*.
(*d* + *1*) ∗ *distr* (*H*,*S*,*z*) ∗ *Re* (*Pfind-update UA H S*)) + (∑ (*H*,*S*,*z*)∈*carrier*.
d ∗ *distr* (*H*,*S*,*z*) ∗ *P-nonterm-update H S UA*)
by (*subst sum.distrib*[*symmetric*], *intro sum.cong*) (*auto simp add*: *algebra-simps*)
also have … = (*d* + *1*) ∗ *Re* ((∑ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Pfind-update UA H S*)) +
d ∗ (∑ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *P-nonterm-update H S UA*)
proof (*subst Re-sum*, *goal-cases*)
case *1*
have *1*: (∑ (*H*,*S*,*z*)∈*carrier*. (*d* + *1*) ∗ *distr* (*H*,*S*,*z*) ∗ *Re* (*Pfind-update UA H S*)) =
(*d* + *1*) ∗ (∑ (*H*,*S*,*z*)∈*carrier*. *Re* ((*distr* (*H*,*S*,*z*)) ∗ *Pfind-update UA H S*))
by (*auto simp add*: *sum-distrib-left distr-pos′ norm-mult intro*!: *sum.cong*)
then show *?case* **unfolding** *1* **by** (*auto simp add*: *sum-distrib-left prod.case-eq-if*
intro!: *sum.cong*)
qed
also have … ≤ (*d* + *1*) ∗ *Re* (*Pfind-pure UA*) + *d* ∗ (*P-nonterm-pure UA*)
unfolding *Pfind-pure-update* **using** *P-nonterm-pure-update d-gr-0* **by** *auto*
finally show *?thesis* **by** *auto*
qed
finally show *?thesis* **by** *auto*
qed

Step 3: prove the mixed O2H only for finite kraus maps using the pure version

lemma *estimate-Pfind-finite-sqrt*:
assumes *finite*: ⋀*i*. *i* < *d*+*1* ⟹ *finite* (*Rep-kraus-family* (*F i*))
and *F-norm-id*:⋀*i*. *i* < *d*+*1* ⟹ *kf-bound* (*F i*) ≤ *id-cblinfun*
and *F-nonzero*: ⋀*i*. *i* < *d*+*1* ⟹ *Rep-kraus-family* (*F i*) ≠ {}
and *norm-Q*: *norm Q* ≤ *1*

shows $|csqrt\ (PM\ Q\ (tc\text{-}tensor\ (\varrho left\ F)\ empty\text{-}tc)) - csqrt\ (PM\ Q\ (\varrho right\ F))| \leq$
$csqrt\ ((d+1) * Pfind\ F + d * P\text{-}nonterm\ F)$
proof –
let *?PMleft* = (λ*UA*. *PM Q* (*tc-tensor* (*ϱleft-pure UA*) *empty-tc*))
let *?PMright* = (λ*UA*. *PM Q* (*ϱright-pure UA*))
define *I* :: (*nat* ⇒ *'mem update*) *set* **where** *I* = *purify-comp-kraus d F*
have *finite I* **unfolding** *I-def* **using** *comp-kraus-maps-set-finite assms* **by** *auto*
have *norm-UA*: $\bigwedge i.\ i<d+1 \Longrightarrow norm\ (UA\ i) \leq 1$ **if** *UA*∈*I* **for** *UA*
by (*intro norm-in-purify-comp-kraus*) (*use that F-norm-id* **in** ‹*auto simp add*: *I-def*›)
have *A*: *PM Q* (*tc-tensor* (*ϱleft F*) *empty-tc*) = (∑ *UA*∈*I*. *?PMleft UA*) **unfolding** *I-def*
by (*subst ϱleft-pure-mixed*) (*auto simp add*: *tc-tensor-sum-left PM-sum-distr assms*)
have *B*: *PM Q* (*ϱright F*) = (∑ *UA*∈*I*. *?PMright UA*) **unfolding** *I-def*
by (*subst ϱright-pure-mixed*) (*auto simp add*: *PM-sum-distr assms*)
have *find*: $(d + 1) * (Pfind\ F) = (\sum UA{\in}I.\ (d + 1) * Pfind\text{-}pure\ UA)$ **unfolding** *I-def*
by (*subst Pfind-pure-mixed*) (*auto simp add*: *sum-distrib-left assms*)
have *nonterm*: $d * (P\text{-}nonterm\ F) = (\sum UA{\in}I.\ d * P\text{-}nonterm\text{-}pure\ UA)$ **unfolding** *I-def*
by (*subst P-nonterm-purification*) (*auto simp add*: *sum-distrib-left assms*)
have *A-pos*: $0 \leq$ *tc-tensor* (*ϱleft F*) *empty-tc* **using** *ϱleft-pos*
by (*simp add*: *empty-tc-pos tc-tensor-pos*)
have *PMleft-pos*: *?PMleft UA* ≥ *0* **for** *UA*
by (*auto intro*!: *PM-pos simp add*: *empty-tc-pos tc-tensor-pos ϱleft-pure-pos*)
have *PMright-pos*: *?PMright UA* ≥ *0* **for** *UA* **by** (*auto intro*!: *PM-pos simp add*: *ϱright-pure-pos*)
have $|csqrt\ (PM\ Q\ (tc\text{-}tensor\ (\varrho left\ F)\ empty\text{-}tc)) - csqrt\ (PM\ Q\ (\varrho right\ F))| =$
$|csqrt\ (Re\ (PM\ Q\ (tc\text{-}tensor\ (\varrho left\ F)\ empty\text{-}tc))) - csqrt\ (Re\ (PM\ Q\ (\varrho right\ F)))|$
by (*subst PM-Re*[*OF A-pos*], *subst PM-Re*[*OF ϱright-pos*]) *auto*
also have … $= |sqrt\ (Re\ (PM\ Q\ (tc\text{-}tensor\ (\varrho left\ F)\ empty\text{-}tc))) - sqrt\ (Re\ (PM\ Q\ (\varrho right\ F)))|$
using *complex-of-real-abs A-pos PM-pos ϱright-pos less-eq-complex-def* **by** *force*
also have … $= |sqrt\ ((\sum UA{\in}I.\ Re\ (?PMleft\ UA))) - sqrt\ ((\sum UA{\in}I.\ Re\ (?PMright\ UA)))|$
unfolding *A B* **by** (*subst Re-sum*, *subst Re-sum*) *auto*
also have … $\leq sqrt\ (\sum UA{\in}I.\ (d+1) * Re\ (Pfind\text{-}pure\ UA) + d * P\text{-}nonterm\text{-}pure\ UA)$
proof –
have *1*: $\forall UA{\in}I.\ 0 \leq Re\ (?PMleft\ UA)$ **using** *PMleft-pos* **by** (*simp add*: *less-eq-complex-def*)
have *2*: $\forall UA{\in}I.\ 0 \leq Re\ (?PMright\ UA)$ **using** *PMright-pos* **by** (*metis cmod-Re norm-ge-zero*)
have *3*: $\forall UA{\in}I.\ 0 \leq real\ (d + 1) * Re\ (Pfind\text{-}pure\ UA) + d * P\text{-}nonterm\text{-}pure\ UA$
using *error-term-pure-pos norm-UA* **by** *auto*
have *4*: $\forall UA{\in}I.\ 0 \leq (1{::}real)$ **by** *auto*
have *5*: $\forall UA{\in}I.$
$|sqrt\ (Re\ (PM\ Q\ (tc\text{-}tensor\ (\varrho left\text{-}pure\ UA)\ empty\text{-}tc))) - sqrt\ (Re\ (PM\ Q\ (\varrho right\text{-}pure\ UA)))|$
$\leq sqrt\ (real\ (d + 1) * Re\ (Pfind\text{-}pure\ UA) + d * P\text{-}nonterm\text{-}pure\ UA)$
using *estimate-Pfind-pure-sqrt norm-UA norm-Q* **by** *auto*
have $|sqrt\ ((\sum UA{\in}I.\ Re\ (?PMleft\ UA))) - sqrt\ ((\sum UA{\in}I.\ Re\ (?PMright\ UA)))|$
$\leq sqrt\ (\sum UA{\in}I.\ ((d+1) * Re\ (Pfind\text{-}pure\ UA) + d * P\text{-}nonterm\text{-}pure\ UA))$
using *sqrt-estimate-real*[*OF* ‹*finite I*› *1 2 3 4 5*] **by** *auto*
then show *?thesis* **by** (*auto intro*!: *complex-of-real-mono*)
qed
also have … $= csqrt\ (\sum UA{\in}I.\ (d+1) * (Pfind\text{-}pure\ UA) + d * P\text{-}nonterm\text{-}pure\ UA)$
proof (*subst of-real-sqrt*, *goal-cases*)

case *1* **then show** *?case* **by** (*intro sum-nonneg*) (*use error-term-pure-pos norm-UA* **in** ‹*auto*›)
next
case *2*
have $*$: *complex-of-real* (*real* (*d* + *1*) $*$ *Re* (*Pfind-pure x*) + *real d* $*$ *P-nonterm-pure x*) = *complex-of-nat* (*d* + *1*) $*$ *Pfind-pure x* + *complex-of-real* (*real d* $*$ *P-nonterm-pure x*) **for** *x*
by (*simp add*: *Re-Pfind-pure*)
then show *?case* **by** (*subst of-real-sum*, *subst* $*$) *auto*
qed
also have ... = *csqrt* (($\sum$ *UA*$\in$*I*. (*d*+*1*) $*$ *Pfind-pure UA*) + ($\sum$ *UA*$\in$*I*. *d* $*$ *P-nonterm-pure UA*))
by (*simp*)
finally show *?thesis* **by** (*subst find*, *subst nonterm*) *auto*
qed

lemma *estimate-Pfind-finite-sqrt'*:
assumes *finite*: $\bigwedge$*i*. *i* < *d*+*1* $\Longrightarrow$ *finite* (*Rep-kraus-family* (*F i*))
and *F-norm-id*:$\bigwedge$*i*. *i* < *d*+*1* $\Longrightarrow$ *kf-bound* (*F i*) $\leq$ *id-cblinfun*
and *F-nonzero*: $\bigwedge$*i*. *i* < *d*+*1* $\Longrightarrow$ *Rep-kraus-family* (*F i*) $\neq$ {}
and *norm-Q*: *norm Q* $\leq$ *1*
shows |*sqrt* (*Re* (*PM Q* (*tc-tensor* (*ϱleft F*) *empty-tc*))) − *sqrt* (*Re* (*PM Q* (*ϱright F*)))| $\leq$ *sqrt* ((*d*+*1*) $*$ *Re* (*Pfind F*) + *d* $*$ *P-nonterm F*)
proof −
let *?f* = *PM Q* (*tc-tensor* (*ϱleft F*) *empty-tc*)
have *rew1*: *sqrt* (*Re* (*?f*)) = *csqrt* (*?f*)
by (*metis PM-Re PM-pos ϱleft-pos complex-of-real-nn-iff empty-tc-pos of-real-sqrt tc-tensor-pos*)
have *rew2*: *sqrt* (*Re* (*PM Q* (*ϱright F*))) = *csqrt* (*PM Q* (*ϱright F*))
using *PM-pos ϱright-pos less-eq-complex-def* **by** *auto*
have *pos*: *0* $\leq$ *real* (*d* + *1*) $*$ *Re* (*Pfind F*) + *real d* $*$ *P-nonterm F*
using *error-term-pos assms* **by** *auto*
have *rew3*: *complex-of-real* (*sqrt* (*real* (*d* + *1*) $*$ *Re* (*Pfind F*) + *real d* $*$ *P-nonterm F*)) = *csqrt* (*real* (*d* + *1*) $*$ (*Pfind F*) + *real d* $*$ *P-nonterm F*)
by (*subst of-real-sqrt*[*OF pos*]) (*auto simp add*: *error-term-pos Re-Pfind*)
show *?thesis* **apply** (*subst complex-of-real-mono-iff*[*symmetric*], *subst complex-of-real-abs*)
apply (*subst of-real-diff*, *subst rew1*, *subst rew2*, *subst rew3*)
by (*use estimate-Pfind-finite-sqrt*[*OF assms*] **in** ‹*auto*›)
qed

Step 4: Prove the mixed O2H for possibly infinite kraus maps using a limit process from finite to infinite kraus maps

lemma *Re-Pfind-has-sum*:
((*λF*. (*1* + *real d*) $*$ *Re* (*Pfind F*)) *has-sum* (*1* + *real d*) $*$ *Re* (*Pfind E*)) (*finite-kraus-subadv E d*)
using *has-sum-cmult-right*[*OF Re-has-sum*[*OF Pfind-has-sum*]] *Pfind-pos*
by (*auto simp add*: *o-def*)

lemma *scale-P-nonterm-has-sum*:
((*λF*. *real d* $*$ *P-nonterm F*) *has-sum real d* $*$ *P-nonterm E*) (*finite-kraus-subadv E d*)
using *P-nonterm-has-sum has-sum-cmult-right* **by** *blast*

```
lemma estimate-Pfind-sqrt:
  assumes norm-Q: norm Q ≤ 1
  shows |sqrt (Pleft′ Q) − sqrt (Pright Q)| ≤
  sqrt ((d+1) ∗ Re (Pfind E) + d ∗ P-nonterm E)
    (is ?left ≤ ?right)
proof −
  have not-bot: finite-subsets-at-top (finite-kraus-subadv E d) ≠ ⊥ by auto
  let ?f = (λ𝔉. sqrt (sum (λF. (d+1) ∗ (Re (Pfind F)) + d ∗ (P-nonterm F)) 𝔉))
  have tendsto-right: (?f ⟶ sqrt (real (d + 1) ∗ Re (Pfind E) + real d ∗ P-nonterm E))
      (finite-subsets-at-top (finite-kraus-subadv E d))
    using Re-Pfind-has-sum scale-P-nonterm-has-sum unfolding has-sum-def
    by (auto intro!: tendsto-real-sqrt tendsto-add tendsto-mult-left tendsto-Re)
  let ?g = (λ𝔉. |sqrt (sum (λF. Re (PM Q (tc-tensor (ϱleft F) empty-tc))) 𝔉) −
    sqrt (sum (λF. Re (PM Q (ϱright F))) 𝔉)|)
  have tendsto-left:
    (?g ⟶ |sqrt (Pleft′ Q) − sqrt (Pright Q)|)
     (finite-subsets-at-top (finite-kraus-subadv E d))
    using Re-PM-left-has-sum Re-PM-right-has-sum unfolding has-sum-def
    by (auto intro!: tendsto-rabs tendsto-real-sqrt tendsto-diff)
  have eventually: ∀F x in finite-subsets-at-top (finite-kraus-subadv E d).
        |sqrt (∑F∈x. Re (PM Q (tc-tensor (ϱleft F) empty-tc))) − sqrt (∑F∈x. Re (PM Q
(ϱright F)))|
       ≤ sqrt (∑F∈x. real (d + 1) ∗ Re (Pfind F) + real d ∗ P-nonterm F)
  proof (intro eventually-finite-subsets-at-top-weakI, goal-cases)
    case (1 G)
    have fin-case: |sqrt (Re (PM Q (tc-tensor (ϱleft F) empty-tc))) − sqrt (Re (PM Q (ϱright
F)))|
         ≤ sqrt (real(d+1) ∗ Re (Pfind F) + real d ∗ P-nonterm F)
      if F∈G for F proof (intro estimate-Pfind-finite-sqrt′[OF - - - norm-Q], goal-cases)
      case (1 i)
      have F ∈ finite-kraus-subadv E d using ‹G ⊆ finite-kraus-subadv E d› that by auto
      then show ?case using fin-subadv-fin-Rep-kraus-family 1 by auto
    next
      case (2 i)
      have kf-bound (F i) ≤ kf-bound (E i)
        by (meson 1(2) 2 Set.basic-monos(7) finite-kraus-subadv-I kf-bound-of-elems that)
      also have kf-bound (E i) ≤ id-cblinfun using 2 E-norm-id by auto
      finally show ?case by linarith
    next
      case (3 i)
      then show ?case using 1(2) fin-subadv-nonzero[where n=d] that by auto
    qed
    then have |sqrt (∑F∈G. 1 ∗ (Re (PM Q (tc-tensor (ϱleft F) empty-tc)))) −
```

```
      sqrt (∑ F∈G. 1 ∗ (Re (PM Q (ϱright F))))|
    ≤ sqrt (∑ F∈G. 1 ∗ (real (d + 1) ∗ Re (Pfind F) + real d ∗ P-nonterm F))
    proof (intro sqrt-estimate-real[OF 1(1)], goal-cases)
      case 3
      then show ?case using error-term-pos by (smt (verit, best) real-sqrt-ge-0-iff)
    qed (auto intro!: Re-PM-pos ϱright-pos tc-tensor-pos empty-tc-pos ϱleft-pos)
    then show ?case by auto
  qed
  show ?thesis by (intro tendsto-le[OF not-bot tendsto-right tendsto-left eventually])
qed


lemma estimate-Pfind:
  assumes norm-Q: norm Q ≤ 1
  shows
    |Pleft′ Q − Pright Q| ≤  2 ∗ sqrt ((d+1) ∗ Re (Pfind E) + d∗ P-nonterm E)
proof −
  have sqrt:
    |sqrt (Pleft′ Q) − sqrt (Pright Q)| ≤ sqrt ((d+1) ∗ Re (Pfind E) + d∗ P-nonterm E)
    using estimate-Pfind-sqrt assms norm-Fst-P by auto
  have |(Pleft′ Q) − (Pright Q)| = |sqrt (Pleft′ Q) − sqrt (Pright Q)| ∗
    |sqrt (Pleft′ Q) + sqrt (Pright Q)|
    unfolding Pleft′-def Pright-def
    by (auto intro!: sqrt-binom Re-PM-pos ϱright-pos tc-tensor-pos ϱleft-pos empty-tc-pos)
  also have ... ≤ 2 ∗ |sqrt (Pleft′ Q) − sqrt (Pright Q)|
  proof −
    have norm (PM Q(tc-tensor (ϱleft E) empty-tc)) ≤ trace-norm (sandwich Q ∗V
        from-trace-class (tc-tensor (ϱleft E) empty-tc))  unfolding PM-def PM-update-def by
auto
    also have ... ≤ (norm (Q :: ('mem × 'l) ell2 ⇒CL ('mem × 'l) ell2))^2 ∗
      trace-norm (from-trace-class (tc-tensor (ϱleft E) empty-tc))
      using trace-norm-sandwich[of from-trace-class (tc-tensor (ϱleft E) empty-tc) Q,
          OF trace-class-from-trace-class] by auto
    also have ... ≤ 1 ∗ norm (tc-tensor (ϱleft E) empty-tc)
     by (smt (verit, ccfv-SIG) norm-Q mult-le-cancel-right2 norm-ge-zero norm-trace-class.rep-eq

          power2-eq-square)
    also have ... = norm (ϱleft E) unfolding norm-tc-tensor norm-empty-tc by auto
    have norm (PM Q(tc-tensor (ϱleft E) empty-tc)) ≤ 1
       by (intro norm-PM, unfold norm-tc-tensor) (auto simp add: norm-ϱleft norm-empty-tc
norm-Q)
    then have left: norm (sqrt (Pleft′ Q)) ≤ 1
      by (simp add: Pleft′-def PM-pos Re-PM-pos ϱleft-pos cmod-Re empty-tc-pos tc-tensor-pos)
    have norm (PM Q(ϱright E)) ≤ 1
      by (intro norm-PM) (auto simp add: norm-ϱright norm-Q)
    then have right: norm (sqrt(Pright Q)) ≤ 1
      by (simp add: Pright-def PM-pos Re-PM-pos ϱright-pos cmod-Re)
    have |sqrt (Pleft′ Q) + sqrt (Pright Q)| ≤ 2
```

using *left right* **by** *auto*
then show *?thesis* **by** (*subst mult.commute*[*of 2*], *intro mult-left-mono*) *auto*
qed
finally show $|Pleft'\ Q - Pright\ Q| \leq 2 * sqrt\ ((d+1) * Re\ (Pfind\ E) + d* P\text{-}nonterm\ E)$
using *sqrt* **by** *auto*
qed

end
end
theory *O2H-Theorem*

imports *Mixed-O2H*

begin

unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
unbundle *register-syntax*

12 General O2H Setting and Theorem

General O2H setting

locale *o2h-theorem* = *o2h-setting* $TYPE('x)$ $TYPE('y::group\text{-}add)$ $TYPE('mem)$ $TYPE('l)$ +
fixes *carrier* :: $(('x \Rightarrow 'y)\times('x \Rightarrow 'y)\times('x \Rightarrow bool)\times\text{-})$ *set*
fixes *distr* :: $(('x \Rightarrow 'y)\times('x \Rightarrow 'y)\times('x \Rightarrow bool)\times\text{-}) \Rightarrow real$

assumes *distr-pos*: $\forall (H,G,S,z)\in carrier.\ distr\ (H,G,S,z) \geq 0$
and *distr-sum-1*: $(\sum (H,G,S,z)\in carrier.\ distr\ (H,G,S,z)) = 1$
and *finite-carrier*: *finite carrier*

and *H-G-same-upto-S*:
$\bigwedge H\ G\ S\ z.\ (H,G,S,z)\in carrier \Longrightarrow x \in -\ Collect\ S \Longrightarrow H\ x = G\ x$

fixes *E*:: *'mem kraus-adv*
assumes *E-norm-id*: $\bigwedge i.\ i < d+1 \Longrightarrow kf\text{-}bound\ (E\ i) \leq id\text{-}cblinfun$
assumes *E-nonzero*: $\bigwedge i.\ i < d+1 \Longrightarrow Rep\text{-}kraus\text{-}family\ (E\ i) \neq \{\}$

fixes *P*:: *'mem update*
assumes *is-Proj-P*: *is-Proj P*

begin

lemma *Fst-E-nonzero*:
$\bigwedge i.\ i < d+1 \Longrightarrow Rep\text{-}kraus\text{-}family\ (kf\text{-}Fst\ (E\ i)) \neq \{\}$
using *E-nonzero* **by** (*simp add*: *kf-Fst.rep-eq*)

Some properties of the joint distribution.

lemma *Uquery-G-H-same-on-not-S-embed'*:

assumes $(H,G,S,z) \in$ *carrier*
shows
Uquery H o_{CL} *proj-classical-set* (− (*Collect S*)) $\otimes_o$ *id-cblinfun* =
Uquery G o_{CL} *proj-classical-set* (− (*Collect S*)) $\otimes_o$ *id-cblinfun*
proof (*intro equal-ket*, *safe*, *unfold tensor-ell2-ket*[*symmetric*], *goal-cases*)
case (*1 a b*)
let *?P* = *proj-classical-set* (− (*Collect S*))
have (*Uquery H* o_{CL} *?P* $\otimes_o$ *id-cblinfun*) $*_V$ *ket a* $\otimes_s$ *ket b* =
(*Uquery G* o_{CL} *?P* $\otimes_o$ *id-cblinfun*) $*_V$ *ket a* $\otimes_s$ *ket b*
if ¬ *S a*
proof −
have (*Uquery H* o_{CL} *?P* $\otimes_o$ *id-cblinfun*) $*_V$ *ket a* $\otimes_s$ *ket b* = *Uquery H* $*_V$ *ket a* $\otimes_s$ *ket b*
by (*simp add*: *proj-classical-set-elem tensor-op-ell2 that*)
also have ... = *ket a* $\otimes_s$ *ket* (*b* + *H a*) **using** *Uquery-ket* **by** *auto*
also have ... = *ket a* $\otimes_s$ *ket* (*b* + *G a*) **using** *H-G-same-upto-S*[*OF assms*] *that* **by** *auto*
also have ... = *Uquery G* $*_V$ *ket a* $\otimes_s$ *ket b* **using** *Uquery-ket* **by** *auto*
also have ... = (*Uquery G* o_{CL} *?P* $\otimes_o$ *id-cblinfun*) $*_V$ *ket a* $\otimes_s$ *ket b*
by (*simp add*: *proj-classical-set-elem tensor-op-ell2 that*)
finally show *?thesis* **by** *auto*
qed
moreover have (*Uquery H* o_{CL} *?P* $\otimes_o$ *id-cblinfun*) $*_V$ *ket a* $\otimes_s$ *ket b* =
(*Uquery G* o_{CL} *?P* $\otimes_o$ *id-cblinfun*) $*_V$ *ket a* $\otimes_s$ *ket b*
if *S a*
by (*simp add*: *proj-classical-set-not-elem tensor-op-ell2 that*)
ultimately show *?case* **by** (*cases S a*, *auto*)
qed

lemma *Uquery-G-H-same-on-not-S-embed*:
assumes $(H,G,S,z) \in$ *carrier*
shows $((X;Y)$ (*Uquery H*) o_{CL} (*not-S-embed S*)) = $((X;Y)$ (*Uquery G*) o_{CL} (*not-S-embed S*))
proof −
have $((X;Y)$ (*Uquery H*) o_{CL} (*not-S-embed S*)) =
$((X;Y)$ (*Uquery H*) o_{CL} $(X;Y)$ (*proj-classical-set* (− (*Collect S*)) $\otimes_o$ *id-cblinfun*))
unfolding *not-S-embed-def* **by** (*simp add*: *Laws-Quantum.register-pair-apply*)
also have ... = $(X;Y)$ (*Uquery H* o_{CL} *proj-classical-set* (− (*Collect S*)) $\otimes_o$ *id-cblinfun*)
by (*simp add*: *Axioms-Quantum.register-mult*)
also have ... = $(X;Y)$ (*Uquery G* o_{CL} *proj-classical-set* (− (*Collect S*)) $\otimes_o$ *id-cblinfun*)
using *Uquery-G-H-same-on-not-S-embed′ assms* **by** *auto*
also have ... = $((X;Y)$ (*Uquery G*) o_{CL} $(X;Y)$ (*proj-classical-set* (− (*Collect S*)) $\otimes_o$ *id-cblinfun*))
by (*simp add*: *Axioms-Quantum.register-mult*)
also have ... = $((X;Y)$ (*Uquery G*) o_{CL} (*not-S-embed S*))
unfolding *not-S-embed-def* **by** (*simp add*: *Laws-Quantum.register-pair-apply*)
finally show *?thesis* **by** *auto*
qed

lemma *Uquery-G-H-same-on-not-S-embed-tensor*:

assumes $(H,G,S,z) \in carrier$
shows $((X\text{-}for\text{-}B;Y\text{-}for\text{-}B)\ (Uquery\ H)\ o_{CL}\ Fst\ (not\text{-}S\text{-}embed\ S)) =$
$((X\text{-}for\text{-}B;Y\text{-}for\text{-}B)\ (Uquery\ G)\ o_{CL}\ Fst\ (not\text{-}S\text{-}embed\ S))$
using *Uquery-G-H-same-on-not-S-embed*[*OF assms*] **unfolding** *UqueryH-tensor-id-cblinfunB Fst-def*
by (*auto simp add*: *comp-tensor-op*)

Instantiations of mixed o2h locale for H and G

definition *carrier-G* **where** $carrier\text{-}G = (\lambda(H,G,S,z).\ (G,S,(H,z)))$ ‘ *carrier*
definition *distr-G* **where** $distr\text{-}G = (\lambda(G,S,(H,z)).\ distr\ (H,G,S,z))$

lemma *distr-G-pos*: $\forall (G,S,z) \in carrier\text{-}G.\ distr\text{-}G\ (G,S,z) \geq 0$
unfolding *carrier-G-def distr-G-def* **using** *distr-pos* **by** *auto*

lemma *distr-G-sum-1*: $(\sum (G,S,z) \in carrier\text{-}G.\ distr\text{-}G\ (G,S,z)) = 1$
unfolding *carrier-G-def distr-G-def* **using** *distr-sum-1*
by (*subst sum.reindex*, *auto simp add*: *inj-on-def case-prod-beta*)

lemma *finite-carrier-G*: *finite carrier-G*
unfolding *carrier-G-def* **by** (*auto simp add*: *inj-on-def finite-carrier*)

definition *carrier-H* **where** $carrier\text{-}H = (\lambda(H,G,S,z).\ (H,S,(G,z)))$ ‘ *carrier*
definition *distr-H* **where** $distr\text{-}H = (\lambda(H,S,(G,z)).\ distr\ (H,G,S,z))$

lemma *distr-H-pos*: $\forall (H,S,z) \in carrier\text{-}H.\ distr\text{-}H\ (H,S,z) \geq 0$
unfolding *carrier-H-def distr-H-def* **using** *distr-pos* **by** *auto*

lemma *distr-H-sum-1*: $(\sum (H,S,z) \in carrier\text{-}H.\ distr\text{-}H\ (H,S,z)) = 1$
unfolding *carrier-H-def distr-H-def* **using** *distr-sum-1*
by (*subst sum.reindex*[], *auto simp add*: *inj-on-def case-prod-beta*)

lemma *finite-carrier-H*: *finite carrier-H*
unfolding *carrier-H-def* **by** (*auto simp add*: *inj-on-def finite-carrier*)

interpretation *mixed-H*: *mixed-o2h X Y d init flip bit valid empty carrier-H distr-H E P*
apply *unfold-locales*
using *distr-H-pos distr-H-sum-1 finite-carrier-H E-norm-id E-nonzero is-Proj-P*
by *auto*

interpretation *mixed-G*: *mixed-o2h X Y d init flip bit valid empty carrier-G distr-G E P*
apply *unfold-locales*
using *distr-G-pos distr-G-sum-1 finite-carrier-G E-norm-id E-nonzero is-Proj-P*
by *auto*

Lemmas on *Proj-ket-upto* and *run-adv-mixed*. The adversary run upto i can be projected to the first i ket states in the counting register.

lemma *length-has-bits-upto*:
  assumes *l*∈*has-bits-upto n*
  shows *length l* = *d*
  using *assms* **unfolding** *has-bits-upto-def len-d-lists-def has-bits-def* **by** *auto*

lemma *empty-not-flip*:
  assumes *x* ∈ *list-to-l* ‘ *has-bits-upto n n*<*d*
  shows *empty* ≠ *flip n x*
proof −
  have *blog x* **using** *assms* **using** *has-bits-upto-def surj-list-to-l* **by** *auto*
  obtain *l* **where** *x*: *x* = *list-to-l l* **and** *l-in*:*l*∈*has-bits-upto n* **using** *assms*(*1*) **by** *blast*
  then have *len*: *length l* = *d* **using** *length-has-bits-upto assms* **by** *auto*
  have ¬ *l* ! (*length l* − *Suc n*) **unfolding** *len* **using** *assms*(*2*) *has-bits-upto-elem l-in* **by** *auto*
  then have *bit x n* = *bit empty n* **unfolding** *x* **by** (*subst bit-list-to-l*) (*auto simp add*: *assms len*)
  then have *bit* (*flip n x*) *n* ≠ *bit empty n* **by** (*subst bit-flip-same*[*OF* ‹*n*<*d*› ‹*blog x*›], *auto*)
  then show *?thesis* **by** *auto*
qed

lemma *empty-not-flip*′:
  assumes *x* ≠ *flip n empty n*<*d*
  shows *empty* ≠ *flip n x*
proof (*rule ccontr*, *safe*)
  assume *empty* = *flip n x*
  then have *flip n empty* = *flip n* (*flip n x*) **by** *auto*
  then have *flip n empty* = *x* **by** (*metis assms*(*2*) *blog.intros*(*1*) *flip-flip not-blog-flip*)
  then show *False* **using** *assms* **by** *auto*
qed

lemma *Proj-ket-upto-Snd*:
  Proj-ket-upto A = *Snd* (*proj-classical-set* (*list-to-l* ‘ *A*))
  unfolding *Proj-ket-upto-def Proj-ket-set-def Snd-def* **by** *auto*

lemma *from-trace-class-tc-selfbutter*:
  from-trace-class (*tc-selfbutter x*) = *selfbutter x*
  by (*simp add*: *tc-butterfly.rep-eq tc-selfbutter-def*)

lemma *selfbutter-empty-US-Proj-ket-upto*:
  assumes *i*<*d*
  shows *Snd* (*selfbutter* (*ket empty*)) o_{CL} ((*US S i*) o_{CL} *Proj-ket-upto* (*has-bits-upto i*)) =
  Fst (*not-S-embed S*) o_{CL} *Snd* (*selfbutter* (*ket empty*))
proof (*intro equal-ket*, *safe*, *goal-cases*)
  case (*1 a b*)
  have *split-a*: *ket a* = *S-embed S* (*ket a*) + *not-S-embed S* (*ket a*)
    using *S-embed-not-S-embed-add* **by** *auto*

have *?case* (**is** *?left* = *?right*) **if** $b \in$ *list-to-l* ‘ *has-bits-upto i*
proof −
  have *Proj* (*ccspan* (*ket* ‘ *list-to-l* ‘ *has-bits-upto i*)) $*_V$ *ket b* = *ket b*
    using *that* **by** (*simp add*: *Proj-fixes-image ccspan-superset′*)
  then have *proj*: *proj-classical-set* (*list-to-l* ‘ *has-bits-upto i*) $*_V$ *ket b* = *ket b*
    unfolding *proj-classical-set-def* **by** *auto*
  have *?left* = (*Snd* (*selfbutter* (*ket empty*)) o_{CL} (*US S i*)) $*_V$ *ket* (*a*, *b*)
   using *proj* **by** (*auto simp add*: *Proj-ket-upto-def Proj-ket-set-def tensor-ell2-ket*[*symmetric*]

      tensor-op-ell2)
  also have . . . = *Snd* (*selfbutter* (*ket empty*)) $*_V$
     ((*S-embed S* $*_V$ *ket a*) $\otimes_s$ (*Ub i*) $*_V$ *ket b* + ((*not-S-embed S* $*_V$ *ket a*) $\otimes_s$ *ket b*))
    using *US-ket-split* **by** *auto*
  also have . . . = *Snd* (*selfbutter* (*ket empty*)) $*_V$ ((*not-S-embed S* $*_V$ *ket a*) $\otimes_s$ *ket b*)
  proof −
    obtain *bs* **where** *b*: *bs*∈ *has-bits-upto i b* = *list-to-l bs*
      using ‹$b \in$ *list-to-l* ‘ *has-bits-upto i*› **by** *auto*
    then have *bs*: *length bs* = *d* ¬ (*bs*!(*d*−*i*−*1*)) **unfolding** *has-bits-upto-def len-d-lists-def*
      using *assms b*(*1*) *has-bits-upto-elem* **by** *auto*
    then have *bit b i* = *bit empty i* **unfolding** *b*(*2*)
      by (*subst bit-list-to-l*) (*auto simp add*: ‹*i*<*d*›)
    then have *flip i b* ≠ *empty* **using** *assms bit-flip-same blog.intros*(*1*) *not-blog-flip* **by** *blast*
   then have *Snd* (*selfbutter* (*ket empty*)) $*_V$ ((*S-embed S* $*_V$ *ket a*) $\otimes_s$ (*Ub i*) $*_V$ *ket b*) = *0*
      by (*simp add*: *Ub-def Snd-def classical-operator-ket*[*OF Ub-exists*] *tensor-op-ell2*
         tensor-ell2-scaleC2)
    then show *?thesis* **by** (*simp add*: *cblinfun.real.add-right*)
  qed

  also have . . . = *?right* **unfolding** *Fst-def Snd-def*
    by (*auto simp add*: *tensor-ell2-ket*[*symmetric*] *cinner-ket tensor-op-ell2*)
  finally show *?thesis* **by** *blast*
qed
moreover have *?case* (**is** *?left* = *?right*) **if** ¬ ($b \in$ *list-to-l* ‘ *has-bits-upto i*) *blog b*
proof −
  have *b* ≠ *empty* **using** *that empty-list-to-l-has-bits-upto* **by** *force*
  have *b* ∉ *list-to-l* ‘ *has-bits-upto i* **using** *that* **by** *auto*
  then have *proj*: *proj-classical-set* (*list-to-l* ‘ *has-bits-upto i*) $*_V$ *ket b* = *0*
    unfolding *proj-classical-set-def* **by** (*intro Proj-0-compl*, *intro mem-ortho-ccspanI*) *auto*
  then have *?left* = *0*
    by (*auto simp add*: *Proj-ket-upto-def Proj-ket-set-def tensor-ell2-ket*[*symmetric*]
      tensor-op-ell2)
  moreover have *?right* = *0* **using** ‹*b* ≠ *empty*› **unfolding** *Fst-def Snd-def*
    by (*auto simp add*: *tensor-ell2-ket*[*symmetric*] *cinner-ket tensor-op-ell2*)
  ultimately show *?thesis* **by** *auto*
qed
moreover have *?case* (**is** *?left* = *?right*) **if** ¬ *blog b*
proof −
  have *b* ≠ *empty* **using** *that blog.intros*(*1*) **by** *auto*
  have *b* ∉ *list-to-l* ‘ *has-bits-upto i*

using *has-bits-upto-def surj-list-to-l that* **by** *fastforce*
then have *proj*: *proj-classical-set* (*list-to-l* ‘ *has-bits-upto i*) $*_V$ *ket b* = *0*
unfolding *proj-classical-set-def* **by** (*intro Proj-0-compl*, *intro mem-ortho-ccspanI*) *auto*
then have *?left* = *0*
by (*auto simp add*: *Proj-ket-upto-def Proj-ket-set-def tensor-ell2-ket*[*symmetric*] *tensor-op-ell2*)
moreover have *?right* = *0* **using** ‹*b* ≠ *empty*› **unfolding** *Fst-def Snd-def*
by (*auto simp add*: *tensor-ell2-ket*[*symmetric*] *cinner-ket tensor-op-ell2*)
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** (*cases b* ∈ *list-to-l* ‘ *has-bits-upto i*, *auto*)
qed

lemma *list-to-l-has-bits-upto-flip*:
assumes *b* ∈ *list-to-l* ‘ *has-bits-upto n n*<*d*
shows *flip n b* ∈ *list-to-l* ‘ *has-bits-upto* (*Suc n*)
proof −
obtain *lb* **where** *lb*: *lb* ∈ *has-bits-upto n* **and** *b*: *b* = *list-to-l lb* **using** *assms* **by** *blast*
then have *len*:*length lb* = *d* **unfolding** *has-bits-upto-def len-d-lists-def* **by** *auto*
moreover have ¬ *lb* ! (*length lb* − *Suc n*) **using** *lb assms*(*2*) *calculation has-bits-upto-elem* **by** *auto*
ultimately have *flip*: *flip n b* = *list-to-l* (*lb*[*length lb* − *Suc n* := *True*])
unfolding *b* **by** (*subst flip-list-to-l*) (*auto simp add*: *assms*)
let *?lb′* = *lb*[*length lb* − *Suc n* := *True*]
have *len-lb′*: *?lb′* ∈ *len-d-lists* **unfolding** *len-d-lists-def* **using** *len* **by** *auto*
have ∀ *i*∈{*Suc n*..<*d*}. ¬ *lb*!(*d*−*i*−*1*) **using** *lb* **unfolding** *has-bits-upto-def has-bits-def* **by** *auto*
then have ∀ *i*∈{*Suc n*..<*d*}. ¬ *?lb′*!(*d*−*i*−*1*) **unfolding** *len* **by** *fastforce*
then have *?lb′* ∉ *has-bits* {*Suc n*..<*d*} **unfolding** *has-bits-def* **by** *auto*
then have *?lb′* ∈ *has-bits-upto* (*Suc n*) **using** *len-lb′* **unfolding** *has-bits-upto-def* **by** *auto*
then show *?thesis* **using** *flip* **by** *auto*
qed

lemma *Proj-ket-upto-US*:
assumes *n*<*d*
shows *US S n* o_{CL} *Proj-ket-upto* (*has-bits-upto n*) =
Proj-ket-upto (*has-bits-upto* (*Suc n*)) o_{CL} *US S n* o_{CL} *Proj-ket-upto* (*has-bits-upto n*)
proof (*intro equal-ket*, *safe*, *goal-cases*)
case (*1 a b*)
have *split-a*: *ket a* = *S-embed S* (*ket a*) + *not-S-embed S* (*ket a*)
using *S-embed-not-S-embed-add* **by** *auto*
have *?case* (**is** *?left* = *?right*) **if** *b* ∈ *list-to-l* ‘ *has-bits-upto n*
proof −
have *Proj* (*ccspan* (*ket* ‘ *list-to-l* ‘ *has-bits-upto n*)) $*_V$ *ket b* = *ket b*
using *that* **by** (*simp add*: *Proj-fixes-image ccspan-superset′*)

then have *Proj*: *Proj-ket-upto* (*has-bits-upto n*) $*_V$ *ket* (*a*,*b*) = *ket* (*a*,*b*)
unfolding *proj-classical-set-def Proj-ket-upto-Snd Snd-def*
by (*auto simp add*: *tensor-op-ell2 tensor-ell2-ket*[*symmetric*])
have *proj-Suc*: *proj-classical-set* (*list-to-l* ‘ *has-bits-upto* (*Suc n*)) $*_V$ *ket b* = *ket b*
unfolding *proj-classical-set-def* **by** (*metis has-bits-upto-incl image-mono le-simps*(*1*)
less-Suc-eq proj-classical-set-def proj-classical-set-elem subset-eq that)
have *proj-Suc-flip*: *proj-classical-set* (*list-to-l* ‘ *has-bits-upto* (*Suc n*)) $*_V$ *ket* (*flip n b*) =
ket (*flip n b*)
using *list-to-l-has-bits-upto-flip*[*OF that* ‹*n*<*d*›] **by** (*auto simp add*: *proj-classical-set-elem*)
have *?left* = *US S n* $*_V$ *ket* (*a*, *b*) **using** *Proj* **by** *auto*
also have ... = (*S-embed S* $*_V$ *ket a*) $\otimes_s$ *ket* (*flip n b*) + ((*not-S-embed S* $*_V$ *ket a*) $\otimes_s$ *ket b*)
using *US-ket-split Ub-ket* **by** *auto*
also have ... = ((*S-embed S* $*_V$ *ket a*) $\otimes_s$
proj-classical-set (*list-to-l* ‘ *has-bits-upto* (*Suc n*)) $*_V$ *ket* (*flip n b*) +
((*not-S-embed S* $*_V$ *ket a*) $\otimes_s$ *proj-classical-set* (*list-to-l* ‘ *has-bits-upto* (*Suc n*)) $*_V$ *ket b*))
unfolding *proj-Suc proj-Suc-flip* **by** *auto*
also have ... = (*Proj-ket-upto* (*has-bits-upto* (*Suc n*)) o_{CL} *US S n*) $*_V$ *ket* (*a*,*b*)
unfolding *Proj-ket-upto-Snd Snd-def US-def*
by (*auto simp add*: *tensor-op-ell2 cblinfun.add-right cblinfun.add-left tensor-ell2-ket*[*symmetric*]

Ub-ket)
also have ... = *?right* **using** *Proj* **by** *auto*
finally show *?thesis* **by** *blast*
qed
moreover have *?case* (**is** *?left* = *?right*) **if** ¬ (*b* ∈ *list-to-l* ‘ *has-bits-upto n*)
proof −
have *b* ∉ *list-to-l* ‘ *has-bits-upto n* **using** *that* **by** *auto*
then have *proj*: *proj-classical-set* (*list-to-l* ‘ *has-bits-upto n*) $*_V$ *ket b* = *0*
unfolding *proj-classical-set-def* **by** (*intro Proj-0-compl*, *intro mem-ortho-ccspanI*) *auto*
then have *Proj*:*Proj-ket-upto* (*has-bits-upto n*) $*_V$ *ket*(*a*,*b*) = *0*
unfolding *Proj-ket-upto-Snd Snd-def* **by** (*auto simp add*: *tensor-ell2-ket*[*symmetric*] *tensor-op-ell2*)
then have *?left* = *0* **by** *auto*
moreover have *?right* = *0* **using** *Proj* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** (*cases b* ∈ *list-to-l* ‘ *has-bits-upto n*, *auto*)
qed

lemma *run-pure-adv-projection*:
assumes *n*<*d*+*1*
and *ϱ*: *ϱ* = *run-pure-adv-tc n* (*λm. if m*<*n*+*1 then Fst* (*UA m*) *else UB m*) (*US S*) *init-B X-for-B Y-for-B H*
shows *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*)) *ϱ* = *ϱ*
using *assms* **proof** (*induct n arbitrary*: *ϱ*)
case *0*
let *?P* = *proj-classical-set* (*list-to-l* ‘ *has-bits-upto 0*)

have *sandwich* (*Snd ?P*) (*selfbutter init-B*) = *selfbutter init-B*
unfolding *init-B-def Proj-ket-upto-Snd*[*symmetric*]
by (*metis Proj-ket-upto-vec empty-list-has-bits-upto empty-list-to-l selfbutter-sandwich*)
then have *from-trace-class* (*sandwich-tc* (*Snd ?P*) (*tc-selfbutter init-B*)) =
from-trace-class (*tc-selfbutter init-B*)
unfolding *from-trace-class-sandwich-tc from-trace-class-tc-selfbutter* **by** *auto*
then have *sand*: *sandwich-tc* (*Snd ?P*) (*tc-selfbutter init-B*) = *tc-selfbutter init-B*
using *from-trace-class-inject* **by** *blast*
have ∗: (*if* (*0::nat*)<*0+1 then Fst* (*UA 0*) *else UB 0*) = *Fst* (*UA 0*) **by** *auto*
have *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto 0*)) ϱ =
sandwich-tc (*Fst* (*UA 0*)) (*sandwich-tc* (*Snd ?P*) (*tc-selfbutter init-B*))
unfolding *Proj-ket-upto-Snd 0 run-pure-adv-tc.simps*(*1*) **unfolding** ∗
by (*metis Fst-def Snd-def from-trace-class-inject from-trace-class-sandwich-tc id-cblinfun.rep-eq*

init-B-def from-trace-class-tc-selfbutter selfbutter-sandwich tensor-op-ell2)
also have . . . = *sandwich-tc* (*Fst* (*UA 0*)) (*tc-selfbutter init-B*)
unfolding *sand* **by** *auto*
finally show *?case* **unfolding** *0*(*2*) **by** *auto*
next
case (*Suc n*)
define *run* **where** *run* = *run-pure-adv-tc n* (λ*m. if m*<*Suc n then Fst* (*UA m*) *else UB m*) (*US S*)
init-B X-for-B Y-for-B H
have ∗: (λ*m. if m*<(*Suc n*)+*1 then Fst* (*UA m*) *else UB m*) (*Suc n*) = *Fst* (*UA* (*Suc n*))
using *Suc* **by** *auto*
have *n*<*d n*<*d+1* **using** *Suc*(*2*) **by** *auto*
have *rew*: (λ*m. if m* < *Suc* (*Suc n*) *then Fst* (*UA m*) *else UB m*) =
(λ*m. if m* < *Suc n then Fst* (*UA m*) *else UB m*)(*Suc n*:= *Fst* (*UA*(*Suc n*)))
by *auto*
have *over*: *run-pure-adv-tc n* (λ*m. if m* < *Suc* (*Suc n*) *then Fst* (*UA m*) *else UB m*) (*US S*) *init-B*
X-for-B Y-for-B H = *run* **unfolding** *run-def rew* **by** (*intro run-pure-adv-tc-over*) *auto*
have *sand-run*: *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*)) *run* = *run* **unfolding** *run-def*
by (*subst Suc*(*1*)[*OF* ‹*n*<*d+1*›]) *auto*
have *Suc′*: *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*)) *run* = *run*
unfolding *run-def* **using** *Suc* ‹*n*<*d+1*› **by** *auto*
have ϱ = *sandwich-tc* (*Fst* (*UA* (*Suc n*)) o_{CL} (*X-for-B*;*Y-for-B*) (*Uquery H*) o_{CL} *US S n* o_{CL}
Proj-ket-upto (*has-bits-upto n*)) *run*
unfolding *Suc*(*3*) **by** (*auto simp add*: *sand-run sandwich-tc-compose′* ‹*n*<*d*› *run-def*[*symmetric*] *over*)
also have . . . = *sandwich-tc* (*Fst* (*UA* (*Suc n*)) o_{CL} (*X-for-B*;*Y-for-B*) (*Uquery H*) o_{CL}
(*Proj-ket-upto* (*has-bits-upto* (*Suc n*)) o_{CL} *US S n* o_{CL} *Proj-ket-upto* (*has-bits-upto n*))) *run*

using *Proj-ket-upto-US*[*OF* ‹*n*<*d*›] **by** (*smt* (*verit*, *best*) *sandwich-tc-compose′*)
also have . . . = *sandwich-tc* (*Fst* (*UA* (*Suc n*)) o_{CL} (*X-for-B*;*Y-for-B*) (*Uquery H*) o_{CL}
Proj-ket-upto (*has-bits-upto* (*Suc n*)) o_{CL} *US S n*) *run*
by (*auto simp add*: *sand-run sandwich-tc-compose′*)
also have . . . = *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto* (*Suc n*)) o_{CL} *Fst* (*UA* (*Suc n*))

o_{CL}
((*X-for-B*;*Y-for-B*) (*Uquery H*)) o_{CL} *US S n*) *run*
unfolding *Proj-ket-upto-Snd Snd-def UqueryH-tensor-id-cblinfunB Fst-def*
by (*auto simp add*: *comp-tensor-op sandwich-tc-compose′*)
also have ... = *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto* (*Suc n*))) ϱ
unfolding *Suc*(*3*) **by** (*auto simp add*: *sand-run sandwich-tc-compose′ Suc* ‹*n*<*d*› *run-def*[*symmetric*] *over*)
finally show *?case* **by** *auto*
qed

lemma *run-mixed-adv-projection-finite*:
assumes ⋀*i*. *i* < *n* + *1* ⟹ *finite* (*Rep-kraus-family* (*kf-Fst* (*F i*)::
((′*mem* × ′*l*) *ell2*, (′*mem* × ′*l*) *ell2*, *unit*) *kraus-family*))
and ⋀*i*. *i* < *n* + *1* ⟹ *fst* ‘ *Rep-kraus-family* (*kf-Fst* (*F i*)::
((′*mem* × ′*l*) *ell2*, (′*mem* × ′*l*) *ell2*, *unit*) *kraus-family*) ≠ {}
assumes *n*<*d*+*1*
shows *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*))
(*run-mixed-adv n* (λ*n*. *kf-Fst* (*F n*)) (*US S*) *init-B X-for-B Y-for-B H*) =
run-mixed-adv n (λ*n*. *kf-Fst* (*F n*)) (*US S*) *init-B X-for-B Y-for-B H*
proof −
have *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*))
(*run-pure-adv-tc n x* (*US S*) *init-B X-for-B Y-for-B H*) =
run-pure-adv-tc n x (*US S*) *init-B X-for-B Y-for-B H*
if *x* ∈ *purify-comp-kraus n* (λ*n*. *kf-Fst* (*F n*)) **for** *x*
proof −
have ∗: (⋀*i*. *i* < *n* + *1* ⟹ *fst* ‘ *Rep-kraus-family* (*F i*) ≠ {})
using *assms*(*2*) **unfolding** *fst-Rep-kf-Fst* **by** *auto*
obtain *UA* **where** *x*:*x* = (λ*a*. *if a* < *n*+*1 then Fst* (*UA a*) *else undefined*) **using**
purification-kf-Fst[*OF* ∗ ‹*x* ∈ *purify-comp-kraus n* (λ*n*. *kf-Fst* (*F n*))›]
by *auto*
show *?thesis* **using** *assms* **by** (*intro run-pure-adv-projection*[*of n - UA* (λ-. *undefined*) *S H*])

(*auto simp add*: *x*)
qed
then show *?thesis* **by** (*subst purification-run-mixed-adv*[*OF assms*(*1*,*2*)], *simp*,
subst purification-run-mixed-adv[*OF assms*(*1*,*2*)], *simp*)
(*use assms* **in** ‹*auto simp add*: *sandwich-tc-sum intro*!: *sum.cong*›)
qed

lemma *run-mixed-adv-projection*:
assumes ⋀*i*. *i* < *d* + *1* ⟹ *fst* ‘ *Rep-kraus-family* (*kf-Fst* (*F i*)::
((′*mem* × ′*l*) *ell2*, (′*mem* × ′*l*) *ell2*, *unit*) *kraus-family*) ≠ {}
assumes *n*<*d*+*1*
shows *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*))
(*run-mixed-adv n* (λ*n*. *kf-Fst* (*F n*)) (*US S*) *init-B X-for-B Y-for-B H*) =
run-mixed-adv n (λ*n*. *kf-Fst* (*F n*)) (*US S*) *init-B X-for-B Y-for-B H*
proof −

define ϱ **where** ϱ = *run-mixed-adv n* (λn. *kf-Fst* (*F n*)) (*US S*) *init-B X-for-B Y-for-B H*
define ϱ*sum* **where**
ϱ*sum F′* = *run-mixed-adv n* (λn. *kf-Fst* (*F′ n*)) (*US S*) *init-B X-for-B Y-for-B H* **for** *F′*
have ϱ*-has-sum′*: (($\lambda F'$. *run-mixed-adv n F′* (*US S*) *init-B X-for-B Y-for-B H*) *has-sum* ϱ) (*finite-kraus-subadv* (λm. *kf-Fst* (*F m*)) *n*)
unfolding ϱ*-def* **using** *run-mixed-adv-has-sum* **by** *blast*
then have ϱ*-has-sum*: (ϱ*sum has-sum* ϱ) (*finite-kraus-subadv F n*)
proof −
have *inj*: *inj-on* (λf. $\lambda n \in \{0..<n+1\}$. *kf-Fst* (*f n*)) (*finite-kraus-subadv F n*)
using *inj-on-kf-Fst* **by** *auto*
have *rew*: ϱ*sum* = (λf. *run-mixed-adv n f* (*US S*) *init-B X-for-B Y-for-B H*) *o* (λf. $\lambda n \in \{0..<n+1\}$. *kf-Fst* (*f n*)) **unfolding** ϱ*sum-def*
using *run-mixed-adv-kf-Fst-restricted*[**where** *init′* = *init-B* **and** *X′* = *X-for-B* **and** *Y′*=*Y-for-B*]
by *auto*
show *?thesis* **unfolding** *rew* **by** (*subst has-sum-reindex*[*OF inj*, *symmetric*]) (*unfold finite-kraus-subadv-Fst-invert*[*symmetric*], *rule* ϱ*-has-sum′*)
qed
have *elem*: (*sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*)) *o* ϱ*sum*) *x* = ϱ*sum x*
if $x \in$ (*finite-kraus-subadv F n*) **for** *x* **unfolding** ϱ*sum-def o-def*
proof (*intro run-mixed-adv-projection-finite*, *goal-cases*)
case (*1 i*)
then show *?case* **using** *finite-kf-Fst*[*OF fin-subadv-fin-Rep-kraus-family*[*OF that*]] *assms*
by *auto*
next
case (*2 i*)
then show *?case* **using** *fin-subadv-nonzero*[*OF that*] *assms*
unfolding *fst-Rep-kf-Fst* **by** *auto*
qed (*use assms* **in** ‹*auto*›)
have *sand-has-sum-rho*: (*sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*)) *o* ϱ*sum has-sum* ϱ) (*finite-kraus-subadv* (*F*) *n*)
by (*subst has-sum-cong*[**where** *g* = ϱ*sum*]) (*use elem* ϱ*-has-sum* **in** ‹*auto*›)
have *sand-has-sum-sand*: (*sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*)) *o* ϱ*sum has-sum* (*sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*)) ϱ))
(*finite-kraus-subadv* (*F*) *n*) **by** (*intro sandwich-tc-has-sum*[*OF* ϱ*-has-sum*])
have *sandwich-tc* (*Proj-ket-upto* (*has-bits-upto n*)) ϱ = ϱ
using *has-sum-unique*[*OF sand-has-sum-sand sand-has-sum-rho*] **by** *auto*
then show *?thesis* **unfolding** ϱ*-def* **by** *auto*
qed

Lemmas of commutation with non-Find event

lemma *Proj-commutes-with-Uquery*:
Snd (*selfbutter* (*ket empty*)) o_{CL} (*X-for-B*;*Y-for-B*) (*Uquery G*) =
(*X-for-B*;*Y-for-B*) (*Uquery G*) o_{CL} *Snd* (*selfbutter* (*ket empty*))
unfolding *Snd-def* **by** (*simp add*: *UqueryH-tensor-id-cblinfunB comp-tensor-op*)

lemma *run-mixed-adv-G-H-same*:
assumes (*H*,*G*,*S*,*z*)∈*carrier n*<*d*+*1*

shows *sandwich-tc* (*Snd* (*selfbutter* (*ket empty*)))
(*run-mixed-adv n* (λ*n*. *kf-Fst* (*E n*)) (*US S*) *init-B X-for-B Y-for-B H*) =
sandwich-tc (*Snd* (*selfbutter* (*ket empty*)))
(*run-mixed-adv n* (λ*n*. *kf-Fst* (*E n*)) (*US S*) *init-B X-for-B Y-for-B G*)
using *assms*(*2*) **proof** (*induct n*)
case (*Suc n*)
have *n*<*d n*<*Suc d* **using** *Suc* **by** *auto*
let *?P* = *Snd* (*selfbutter* (*ket empty*))
let *?P′* = *Proj-ket-upto* (*has-bits-upto n*)
let *?ϱ* = (λ*x Y*. (*run-mixed-adv x* (λ*n*. *kf-Fst* (*E n*)) (*US S*) *init-B X-for-B Y-for-B Y*))
have *sandwich-tc ?P* (*?ϱ* (*Suc n*) *H*) = *kf-apply* (*kf-Fst* (*E* (*Suc n*)))
(*sandwich-tc* (*?P* o_{CL} (*X-for-B*;*Y-for-B*) (*Uquery H*) o_{CL} *US S n*) (*?ϱ n H*))
using *sandwich-tc-kf-apply-Fst* **by** (*auto simp add*: *sandwich-tc-compose′*)
also have ... = *kf-apply* (*kf-Fst* (*E* (*Suc n*)))
(*sandwich-tc* ((*X-for-B*;*Y-for-B*) (*Uquery H*) o_{CL} *?P* o_{CL} *US S n* o_{CL} *?P′*) (*?ϱ n H*))
by (*subst Proj-commutes-with-Uquery*, *subst run-mixed-adv-projection*[*symmetric*])
(*auto simp add*: *Fst-E-nonzero sandwich-tc-compose′* ‹*n*<*Suc d*›)
also have ... = *kf-apply* (*kf-Fst* (*E* (*Suc n*)))
(*sandwich-tc* ((*X-for-B*;*Y-for-B*) (*Uquery H*) o_{CL} *Fst* (*not-S-embed S*) o_{CL} *?P*) (*?ϱ n H*))
using *selfbutter-empty-US-Proj-ket-upto*[*OF* ‹*n*<*d*›]
by (*metis* (*no-types*, *lifting*) *sandwich-tc-compose′*)
also have ... = *kf-apply* (*kf-Fst* (*E* (*Suc n*)))
(*sandwich-tc* ((*X-for-B*;*Y-for-B*) (*Uquery G*) o_{CL} *Fst* (*not-S-embed S*) o_{CL} *?P*) (*?ϱ n H*))
using *Uquery-G-H-same-on-not-S-embed-tensor assms* **by** *auto*
also have ... = *kf-apply* (*kf-Fst* (*E* (*Suc n*)))
(*sandwich-tc* ((*X-for-B*;*Y-for-B*) (*Uquery G*) o_{CL} *Fst* (*not-S-embed S*) o_{CL} *?P*) (*?ϱ n G*))
using *Suc* **by** (*auto simp add*: *sandwich-tc-compose′*)
also have ... = *kf-apply* (*kf-Fst* (*E* (*Suc n*)))
(*sandwich-tc* ((*X-for-B*;*Y-for-B*) (*Uquery G*) o_{CL} *?P* o_{CL} *US S n* o_{CL} *?P′*) (*?ϱ n G*))
using *selfbutter-empty-US-Proj-ket-upto*[*OF* ‹*n*<*d*›]
by (*metis* (*no-types*, *lifting*) *sandwich-tc-compose′*)
also have ... = *kf-apply* (*kf-Fst* (*E* (*Suc n*)))
(*sandwich-tc* (*?P* o_{CL} (*X-for-B*;*Y-for-B*) (*Uquery G*) o_{CL} *US S n*) (*?ϱ n G*))
by (*subst Proj-commutes-with-Uquery*, *subst* (*2*) *run-mixed-adv-projection*[*symmetric*])
(*auto simp add*: *Fst-E-nonzero sandwich-tc-compose′* ‹*n*<*Suc d*›)
also have ... = *sandwich-tc ?P* (*?ϱ* (*Suc n*) *G*)
using *sandwich-tc-kf-apply-Fst*[*symmetric*] **by** (*auto simp add*: *sandwich-tc-compose′*)
finally show *?case* **by** *auto*
qed *auto*

lemma *run-mixed-B-G-H-same*:
assumes (*H*,*G*,*S*,*z*)∈*carrier*
shows *sandwich-tc* (*Q* $\otimes_o$ *selfbutter* (*ket empty*)) (*run-mixed-B E H S*) =
sandwich-tc (*Q* $\otimes_o$ *selfbutter* (*ket empty*)) (*run-mixed-B E G S*)
proof −
have *sandwich-tc* (*Q* $\otimes_o$ *selfbutter* (*ket empty*)) (*run-mixed-B E H S*) =
sandwich-tc (*Q* $\otimes_o$ *id-cblinfun*) (*sandwich-tc* (*Snd* (*selfbutter* (*ket empty*)))
(*run-mixed-adv d* (λ*n*. *kf-Fst* (*E n*)) (*US S*) *init-B X-for-B Y-for-B H*))

unfolding *run-mixed-B-def*
by (*auto simp add*: *sandwich-tc-compose′*[*symmetric*] *Snd-def comp-tensor-op*)
also have ... = *sandwich-tc* ($Q \otimes_o$ *id-cblinfun*) (*sandwich-tc* (*Snd* (*selfbutter* (*ket empty*)))
(*run-mixed-adv d* (λn. *kf-Fst* (*E n*)) (*US S*) *init-B X-for-B Y-for-B G*))
using *run-mixed-adv-G-H-same*[**where** *n*=*d*, *OF assms*] **by** *auto*
also have ... = *sandwich-tc* ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*run-mixed-B E G S*)
unfolding *run-mixed-B-def*
by (*auto simp add*: *sandwich-tc-compose′*[*symmetric*] *Snd-def comp-tensor-op*)
finally show *?thesis* **by** *auto*
qed

lemma *ϱright-G-H-same*:
sandwich-tc ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*mixed-H.ϱright E*) =
sandwich-tc ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*mixed-G.ϱright E*)
proof −
have *sandwich-tc* ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*mixed-H.ϱright E*) =
($\sum$ (*H*,*G*,*S*,*z*)∈*carrier*. *distr* (*H*,*G*,*S*,*z*) $*_C$
sandwich-tc ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*run-mixed-B E H S*))
unfolding *mixed-H.ϱright-def* **unfolding** *carrier-H-def distr-H-def*
by (*subst sum.reindex*) (*auto simp add*: *inj-on-def case-prod-beta sandwich-tc-sum*
sandwich-tc-scaleC-right intro!: *sum.cong*)
also have ... = ($\sum$ (*H*,*G*,*S*,*z*)∈*carrier*. *distr* (*H*,*G*,*S*,*z*) $*_C$
sandwich-tc ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*run-mixed-B E G S*))
using *run-mixed-B-G-H-same* **by** (*auto intro*!: *sum.cong*)
also have ... = *sandwich-tc* ($Q \otimes_o$ *selfbutter* (*ket empty*)) (*mixed-G.ϱright E*)
unfolding *mixed-G.ϱright-def* **unfolding** *carrier-G-def distr-G-def*
by (*subst sum.reindex*) (*auto simp add*: *inj-on-def case-prod-beta sandwich-tc-sum*
sandwich-tc-scaleC-right intro!: *sum.cong*)
finally show *?thesis* **by** *auto*
qed

lemma *trace-compose-tcr-H-G-same*:
trace-tc (*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-H.ϱright E*)) =
trace-tc (*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-G.ϱright E*))
proof −
have *trace* (*from-trace-class*
(*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-H.ϱright E*)) o_{CL} (*Snd* (*selfbutter* (*ket empty*)))∗) =
trace (*from-trace-class*
(*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-G.ϱright E*)) o_{CL} (*Snd* (*selfbutter* (*ket empty*)))∗)
by (*metis* (*no-types*, *opaque-lifting*) *Snd-def ϱright-G-H-same compose-tcr.rep-eq*
from-trace-class-sandwich-tc sandwich-apply)
then have *trace* (*from-trace-class*
(*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-H.ϱright E*))) =
trace (*from-trace-class*
(*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-G.ϱright E*)))

by (*smt* (*verit*, *best*) *Snd-def butterfly-is-Proj cblinfun-assoc-left*(*1*) *circularity-of-trace compose-tcr.rep-eq is-Proj-algebraic is-Proj-id is-Proj-tensor-op norm-ket trace-class-from-trace-class*)
then show *?thesis* **unfolding** *trace-tc.rep-eq* **by** *auto*
qed

The probability of not Find and the adversary succeeding for H§and G§are the same. $Pr[b \not{F}ind : b \leftarrow A^{H\backslash S}(z)] = Pr[b \not{F}ind : b \leftarrow A^{G\backslash S}(z)]$

lemma *Pright-G-H-same*:
mixed-H.Pright ($Q \otimes_o$ *selfbutter* (*ket empty*)) = *mixed-G.Pright* ($Q \otimes_o$ *selfbutter* (*ket empty*))
unfolding *mixed-H.Pright-def mixed-G.Pright-def mixed-G.PM-altdef*
using *ϱright-G-H-same*[**where** $Q = Q$] **by** *auto*

The finding event occurs with the same probability for G and H if the overall norm stays the same.

lemma *Pfind-G-H-same*:
assumes *norm* (*mixed-H.ϱright E*) = *norm* (*mixed-G.ϱright E*)
shows *mixed-H.Pfind E* = *mixed-G.Pfind E*
proof −
have *mixed-H.Pfind E* = *trace-tc* (*compose-tcr*
(*id-cblinfun* − *Snd* (*selfbutter* (*ket empty*))::(*'mem*×*'l*) *update*) (*mixed-H.ϱright E*))
unfolding *mixed-H.Pfind-def mixed-G.end-measure-def Snd-def*
by (*auto simp add*: *tensor-op-right-minus*)
also have ... = *trace-tc* (*mixed-H.ϱright E*) −
trace-tc (*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-H.ϱright E*))
by (*simp add*: *compose-tcr.diff-left trace-tc-minus*)
also have ... = *norm* (*mixed-H.ϱright E*) −
trace-tc (*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-H.ϱright E*))
by (*simp add*: *mixed-H.ϱright-pos norm-tc-pos*)
also have ... = *norm* (*mixed-G.ϱright E*) −
trace-tc (*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-G.ϱright E*))
unfolding *assms* **using** *trace-compose-tcr-H-G-same* **by** *auto*
also have ... = *trace-tc* (*mixed-G.ϱright E*) −
trace-tc (*compose-tcr* (*Snd* (*selfbutter* (*ket empty*))) (*mixed-G.ϱright E*))
by (*simp add*: *mixed-G.ϱright-pos norm-tc-pos*)
also have ... = *trace-tc* (*compose-tcr*
(*id-cblinfun* − *Snd* (*selfbutter* (*ket empty*))::(*'mem*×*'l*) *update*) (*mixed-G.ϱright E*))
by (*simp add*: *compose-tcr.diff-left trace-tc-minus*)
also have ... = *mixed-G.Pfind E*
unfolding *mixed-G.Pfind-def mixed-G.end-measure-def Snd-def*
by (*auto simp add*: *tensor-op-right-minus*)
finally show *?thesis* **by** *auto*
qed

lemma *Pfind-G-H-same-nonterm*:
shows (*mixed-H.Pfind E* − *mixed-G.Pfind E*) =
(*norm* (*mixed-H.ϱright E*) − *norm* (*mixed-G.ϱright E*))
proof −

have *mixed-H.Pfind E = trace-tc (compose-tcr*
(id-cblinfun − Snd (selfbutter (ket empty))::('mem×'l) update) (mixed-H.ϱright E))
unfolding *mixed-H.Pfind-def mixed-G.end-measure-def Snd-def*
by *(auto simp add: tensor-op-right-minus)*
also have *. . . = trace-tc (mixed-H.ϱright E) −*
trace-tc (compose-tcr (Snd (selfbutter (ket empty))) (mixed-H.ϱright E))
by *(simp add: compose-tcr.diff-left trace-tc-minus)*
also have *. . . = norm (mixed-H.ϱright E) −*
trace-tc (compose-tcr (Snd (selfbutter (ket empty))) (mixed-H.ϱright E))
by *(simp add: mixed-H.ϱright-pos norm-tc-pos)*
finally have *H: mixed-H.Pfind E = norm (mixed-H.ϱright E) −*
trace-tc (compose-tcr (Snd (selfbutter (ket empty))) (mixed-G.ϱright E))
using *trace-compose-tcr-H-G-same* **by** *auto*
have *mixed-G.Pfind E = trace-tc (compose-tcr*
(id-cblinfun − Snd (selfbutter (ket empty))::('mem×'l) update) (mixed-G.ϱright E))
unfolding *mixed-G.Pfind-def mixed-G.end-measure-def Snd-def*
by *(auto simp add: tensor-op-right-minus)*
also have *. . . = trace-tc (mixed-G.ϱright E) −*
trace-tc (compose-tcr (Snd (selfbutter (ket empty))) (mixed-G.ϱright E))
by *(simp add: compose-tcr.diff-left trace-tc-minus)*
finally have *G: mixed-G.Pfind E = norm (mixed-G.ϱright E) −*
trace-tc (compose-tcr (Snd (selfbutter (ket empty))) (mixed-G.ϱright E))
by *(simp add: mixed-G.ϱright-pos norm-tc-pos)*
show *?thesis* **unfolding** *H G* **using** *complex-of-real-abs* **by** *auto*
qed

The general version of the O2H with non-termination part.

theorem *mixed-o2h-nonterm*:
shows
|mixed-H.Pleft P − mixed-G.Pleft P| ≤
2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
+ 2 ∗ sqrt ((d+1) ∗ Re (mixed-G.Pfind E) + d∗ mixed-G.P-nonterm E)
and
|sqrt (mixed-H.Pleft P) − sqrt (mixed-G.Pleft P)| ≤
sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
+ sqrt ((d+1) ∗ Re (mixed-G.Pfind E) + d∗ mixed-G.P-nonterm E)
proof −
let *?P = P* $\otimes_o$ *selfbutter (ket empty)*
have *norm-P: norm ?P ≤ 1*
by *(simp add: butterfly-is-Proj is-Proj-tensor-op mixed-G.is-Proj-P norm-is-Proj)*
have *|mixed-H.Pleft P − mixed-G.Pleft P| =*
|mixed-H.Pleft' ?P − mixed-H.Pright ?P + mixed-G.Pright ?P − mixed-G.Pleft' ?P|
using *Pright-G-H-same* **unfolding** *mixed-G.Pleft-Pleft'-empty mixed-H.Pleft-Pleft'-empty*
by *auto*
also have *. . . ≤ |mixed-H.Pleft' ?P − mixed-H.Pright ?P| +*
|mixed-G.Pleft' ?P − mixed-G.Pright ?P| **by** *linarith*
also have *. . . ≤ 2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E) +*
2 ∗ sqrt ((d+1) ∗ Re (mixed-G.Pfind E) + d∗ mixed-G.P-nonterm E)
using *mixed-H.estimate-Pfind[OF norm-P] mixed-G.estimate-Pfind[OF norm-P]*

by *auto*
finally show |*mixed-H.Pleft P* − *mixed-G.Pleft P*| ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
+ *2* ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-G.Pfind E*) + *d*∗ *mixed-G.P-nonterm E*)
by *auto*
have |*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-G.Pleft P*)| =
|*sqrt* (*mixed-H.Pleft′ ?P*) − *sqrt* (*mixed-H.Pright ?P*) +
sqrt (*mixed-G.Pright ?P*) − *sqrt* (*mixed-G.Pleft′ ?P*)|
using *Pright-G-H-same* **unfolding** *mixed-G.Pleft-Pleft′-empty mixed-H.Pleft-Pleft′-empty*
by *auto*
also have ... ≤ |*sqrt* (*mixed-H.Pleft′ ?P*) − *sqrt* (*mixed-H.Pright ?P*)| +
|*sqrt* (*mixed-G.Pleft′ ?P*) − *sqrt* (*mixed-G.Pright ?P*)| **by** *linarith*
also have ... ≤ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
+ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-G.Pfind E*) + *d*∗ *mixed-G.P-nonterm E*)
using *mixed-H.estimate-Pfind-sqrt*[*OF norm-P*] *mixed-G.estimate-Pfind-sqrt*[*OF norm-P*]
by *auto*
finally show |*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-G.Pleft P*)| ≤
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
+ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-G.Pfind E*) + *d*∗ *mixed-G.P-nonterm E*)
by *auto*
qed

The general version of the O2H with terminating adversary. This formulation corresponds to Theorem 1.

theorem *mixed-o2h-term*:
assumes ⋀*i*. *i*<*d*+*1* ⟹ *km-trace-preserving* (*kf-apply* (*E i*))
shows
|*mixed-H.Pleft P* − *mixed-G.Pleft P*| ≤ *4* ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
and
|*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-G.Pleft P*)| ≤ *2* ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
proof −
have *normHright*: *norm* (*mixed-H.ϱright E*) = *1*
by (*rule mixed-H.trace-preserving-norm-ϱright*[*OF trace-preserving-kf-Fst*[*OF assms*]])
have *normHcount*: *norm* (*mixed-H.ϱcount E*) = *1*
by (*rule mixed-H.trace-preserving-norm-ϱcount*[*OF trace-preserving-kf-Fst*[*OF assms*]])
have *normGright*: *norm* (*mixed-G.ϱright E*) = *1*
by (*rule mixed-G.trace-preserving-norm-ϱright*[*OF trace-preserving-kf-Fst*[*OF assms*]])
have *normGcount*: *norm* (*mixed-G.ϱcount E*) = *1*
by (*rule mixed-G.trace-preserving-norm-ϱcount*[*OF trace-preserving-kf-Fst*[*OF assms*]])
have *norm*: *norm* (*mixed-H.ϱright E*) = *norm* (*mixed-G.ϱright E*) **using** *normHright normGright* **by** *auto*
have *terminH*: *mixed-H.P-nonterm E* = *0*
unfolding *mixed-H.P-nonterm-def* **using** *normHright normHcount*
by (*metis cmod-Re complex-of-real-nn-iff mixed-H.ϱcount-pos mixed-H.ϱright-pos norm-le-zero-iff*
norm-pths(*2*) *norm-tc-pos norm-zero of-real-0*)
have *terminG*: *mixed-G.P-nonterm E* = *0*
unfolding *mixed-G.P-nonterm-def* **using** *normGright normGcount*

by (*smt* (*verit*, *del-insts*) *Re-complex-of-real mixed-G.ϱcount-pos mixed-G.ϱright-pos norm-eq-zero*

norm-le-zero-iff norm-of-real norm-pths(*2*) *norm-tc-pos*)
show |*mixed-H.Pleft P* − *mixed-G.Pleft P*| ≤ *4* ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
using *mixed-o2h-nonterm*(*1*) *Pfind-G-H-same*[*OF norm*] *terminH terminG* **by** *auto*
show |*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-G.Pleft P*)| ≤ *2* ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
using *mixed-o2h-nonterm*(*2*) *Pfind-G-H-same*[*OF norm*] *terminH terminG* **by** *auto*
qed

Other formulations of the mixed o2h.

Theorem 1, definition of Pright (2)

definition *Proj-2* :: (*'mem* × *'l*) *ell2* $\Rightarrow_{CL}$ (*'mem* × *'l*) *ell2* **where**
Proj-2 = *P* $\otimes_o$ *id-cblinfun*

lemma *norm-Proj-2*:
norm Proj-2 ≤ *1*
unfolding *Proj-2-def* **using** *mixed-H.norm-P* **by** (*simp add*: *tensor-op-norm*)

theorem *mixed-o2h-nonterm-2*:
shows
|*mixed-H.Pleft P* − *mixed-H.Pright Proj-2*| ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
and
|*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-2*)| ≤
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
proof −
have |*mixed-H.Pleft P* − *mixed-H.Pright Proj-2*| =
|*mixed-H.Pleft' Proj-2* − *mixed-H.Pright Proj-2*|
unfolding *mixed-H.Pleft-Pleft'-id Proj-2-def* **by** *auto*
also have ... ≤ *2* ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
using *mixed-H.estimate-Pfind*[*OF norm-Proj-2*] **by** *auto*
finally show |*mixed-H.Pleft P* − *mixed-H.Pright Proj-2*| ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
by *auto*
have |*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-2*)| =
|*sqrt* (*mixed-H.Pleft' Proj-2*) − *sqrt* (*mixed-H.Pright Proj-2*)|
unfolding *mixed-H.Pleft-Pleft'-id Proj-2-def* **by** *auto*
also have ... ≤ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
using *mixed-H.estimate-Pfind-sqrt*[*OF norm-Proj-2*] **by** *auto*
finally show |*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-2*)| ≤
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
by *auto*
qed

theorem *mixed-o2h-term-2*:
assumes ⋀*i*. *i*<*d*+*1* ⟹ *km-trace-preserving* (*kf-apply* (*E i*))
shows

|*mixed-H.Pleft P* − *mixed-H.Pright Proj-2*| ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
and
|*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-2*)| ≤
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
proof −
have *normHright*: *norm* (*mixed-H.ϱright E*) = *1*
by (*rule mixed-H.trace-preserving-norm-ϱright*[*OF trace-preserving-kf-Fst*[*OF assms*]])
have *normHcount*: *norm* (*mixed-H.ϱcount E*) = *1*
by (*rule mixed-H.trace-preserving-norm-ϱcount*[*OF trace-preserving-kf-Fst*[*OF assms*]])
have *terminH*: *mixed-H.P-nonterm E* = *0*
unfolding *mixed-H.P-nonterm-def* **using** *normHright normHcount*
by (*metis cmod-Re complex-of-real-nn-iff mixed-H.ϱcount-pos mixed-H.ϱright-pos norm-le-zero-iff*

norm-pths(*2*) *norm-tc-pos norm-zero of-real-0*)
show |*mixed-H.Pleft P* − *mixed-H.Pright Proj-2*| ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
using *mixed-o2h-nonterm-2*(*1*) *terminH* **by** *auto*
show |*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-2*)| ≤
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
using *mixed-o2h-nonterm-2*(*2*) *terminH* **by** *auto*
qed

Theorem 1, definition of Pright (3)

definition *Proj-3* :: (*'mem* × *'l*) *ell2* $\Rightarrow_{CL}$ (*'mem* × *'l*) *ell2* **where**
Proj-3 = *P* $\otimes_o$ *selfbutter* (*ket empty*)

lemma *is-Proj-3*:
is-Proj Proj-3
unfolding *Proj-3-def*
by (*simp add*: *butterfly-is-Proj is-Proj-tensor-op mixed-G.is-Proj-P*)

lemma *Proj-3-altdef*:
Proj-3 = *Proj* ((*P* $\otimes_o$ *id-cblinfun*) $*_S$ ⊤ ⊔ (*id-cblinfun* $\otimes_o$ *selfbutter* (*ket empty*)) $*_S$ ⊤)
oops

lemma *norm-Proj-3*:
norm Proj-3 ≤ *1*
unfolding *Proj-3-def* **using** *mixed-H.norm-P* **by** (*simp add*: *norm-butterfly tensor-op-norm*)

theorem *mixed-o2h-nonterm-3*:
shows
|*mixed-H.Pleft P* − *mixed-H.Pright Proj-3*| ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
and
|*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-3*)| ≤
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
proof −
have |*mixed-H.Pleft P* − *mixed-H.Pright Proj-3*| =

```
    |mixed-H.Pleft' Proj-3 − mixed-H.Pright Proj-3|
    unfolding mixed-H.Pleft-Pleft'-empty Proj-3-def by auto
  also have ... ≤ 2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
    using mixed-H.estimate-Pfind[OF norm-Proj-3] by auto
  finally show |mixed-H.Pleft P − mixed-H.Pright Proj-3| ≤
    2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
    by auto
  have |sqrt (mixed-H.Pleft P) − sqrt (mixed-H.Pright Proj-3)| =
    |sqrt (mixed-H.Pleft' Proj-3) − sqrt (mixed-H.Pright Proj-3)|
    unfolding mixed-H.Pleft-Pleft'-empty Proj-3-def by auto
  also have ... ≤ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
    using mixed-H.estimate-Pfind-sqrt[OF norm-Proj-3] by auto
  finally show |sqrt (mixed-H.Pleft P) − sqrt (mixed-H.Pright Proj-3)| ≤
    sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
    by auto
qed

theorem mixed-o2h-term-3:
  assumes ⋀i. i<d+1 ⟹ km-trace-preserving (kf-apply (E i))
  shows
    |mixed-H.Pleft P − mixed-H.Pright Proj-3| ≤
  2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
    and
    |sqrt (mixed-H.Pleft P) − sqrt (mixed-H.Pright Proj-3)| ≤
  sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
proof −
  have normHright: norm (mixed-H.ϱright E) = 1
    by (rule mixed-H.trace-preserving-norm-ϱright[OF trace-preserving-kf-Fst[OF assms]])
  have normHcount: norm (mixed-H.ϱcount E) = 1
    by (rule mixed-H.trace-preserving-norm-ϱcount[OF trace-preserving-kf-Fst[OF assms]])
  have terminH: mixed-H.P-nonterm E = 0
    unfolding mixed-H.P-nonterm-def using normHright normHcount
   by (metis cmod-Re complex-of-real-nn-iff mixed-H.ϱcount-pos mixed-H.ϱright-pos norm-le-zero-iff

        norm-pths(2) norm-tc-pos norm-zero of-real-0)
  show |mixed-H.Pleft P − mixed-H.Pright Proj-3| ≤
    2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
    using mixed-o2h-nonterm-3(1) terminH by auto
  show |sqrt (mixed-H.Pleft P) − sqrt (mixed-H.Pright Proj-3)| ≤
    sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
    using mixed-o2h-nonterm-3(2) terminH by auto
qed
```

Theorem 1, definition of Pright (4)

```
theorem mixed-o2h-nonterm-4:
  shows
    |mixed-H.Pleft P − mixed-G.Pright Proj-3| ≤
  2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
    and
```

$|sqrt\ (mixed\text{-}H.Pleft\ P) - sqrt\ (mixed\text{-}G.Pright\ Proj\text{-}3)| \le$
sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
proof −
have *|mixed-H.Pleft P − mixed-G.Pright Proj-3| =*
|mixed-H.Pleft′ Proj-3 − mixed-H.Pright Proj-3|
using *Pright-G-H-same* **unfolding** *mixed-G.Pleft-Pleft′-empty mixed-H.Pleft-Pleft′-empty Proj-3-def* **by** *auto*
also have ... ≤ *2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)*
using *mixed-H.estimate-Pfind[OF norm-Proj-3]*
by *auto*
finally show *|mixed-H.Pleft P − mixed-G.Pright Proj-3| ≤*
2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
by *auto*
have *|sqrt (mixed-H.Pleft P) − sqrt (mixed-G.Pright Proj-3)| =*
|sqrt (mixed-H.Pleft′ Proj-3) − sqrt (mixed-H.Pright Proj-3)|
using *Pright-G-H-same* **unfolding** *mixed-G.Pleft-Pleft′-empty mixed-H.Pleft-Pleft′-empty Proj-3-def* **by** *auto*
also have ... ≤ *sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)*
using *mixed-H.estimate-Pfind-sqrt[OF norm-Proj-3]* **by** *auto*
finally show *|sqrt (mixed-H.Pleft P) − sqrt (mixed-G.Pright Proj-3)| ≤*
sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)
by *auto*
qed

theorem *mixed-o2h-term-4*:
assumes $\bigwedge i.\ i<d+1 \Longrightarrow$ *km-trace-preserving (kf-apply (E i))*
shows
|mixed-H.Pleft P − mixed-G.Pright Proj-3| ≤
2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
and
|sqrt (mixed-H.Pleft P) − sqrt (mixed-G.Pright Proj-3)| ≤
sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
proof −
have *normHright*: *norm (mixed-H.ϱright E) = 1*
by *(rule mixed-H.trace-preserving-norm-ϱright[OF trace-preserving-kf-Fst[OF assms]])*
have *normHcount*: *norm (mixed-H.ϱcount E) = 1*
by *(rule mixed-H.trace-preserving-norm-ϱcount[OF trace-preserving-kf-Fst[OF assms]])*
have *terminH*: *mixed-H.P-nonterm E = 0*
unfolding *mixed-H.P-nonterm-def* **using** *normHright normHcount*
by *(metis cmod-Re complex-of-real-nn-iff mixed-H.ϱcount-pos mixed-H.ϱright-pos norm-le-zero-iff*

norm-pths(2) norm-tc-pos norm-zero of-real-0)
show *|mixed-H.Pleft P − mixed-G.Pright Proj-3| ≤*
2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
using *mixed-o2h-nonterm-4(1) terminH* **by** *auto*
show *|sqrt (mixed-H.Pleft P) − sqrt (mixed-G.Pright Proj-3)| ≤*
sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
using *mixed-o2h-nonterm-4(2) terminH* **by** *auto*
qed

Theorem 1: the definition of Pright (5) is $Pright = P[find \vee b{=}1 \ for \ b{<}{-} \ A\hat{}\{H\backslash S\}] = P(find \ for \ b{<}{-} \ A\hat{}\{H\backslash S\}) + P(\neg \ find \wedge b{=}1 \ for \ b{<}{-} \ A\hat{}\{H\backslash S\})$

Careful: In general, we cannot state quantum events with and or or. However, in the case that the two projectors commute, we may say $Pr(A \wedge B) \equiv PM\text{-}A \ o \ PM\text{-}B \ Pr(A \vee B) \equiv PM\text{-}A + PM\text{-}B - PM\text{-}A \ o \ PM\text{-}B$

Still, for the projection, we need to joint the two projective spaces.

definition *Proj-5* :: ('*mem* × '*l*) *ell2* $\Rightarrow_{CL}$ ('*mem* × '*l*) *ell2* **where**
Proj-5 = *Proj* (((*P* $\otimes_o$ *id-cblinfun*) $*_S$ ⊤) ⊔ ((*id-cblinfun* $\otimes_o$ (*id-cblinfun* − *selfbutter* (*ket* *empty*))) $*_S$ ⊤))

lemma *is-Proj-5*:
is-Proj Proj-5
unfolding *Proj-5-def* **by** (*simp*)

lemma *Proj-5-altdef*:
Proj-5 = *Proj-3* + *mixed-H.end-measure*
unfolding *Proj-5-def Proj-3-def* **by** (*subst splitting-Proj-or*[*OF mixed-H.is-Proj-P*])
(*auto simp add*: *mixed-G.end-measure-def Snd-def butterfly-is-Proj*)

lemma *norm-Proj-5*:
norm Proj-5 ≤ *1*
unfolding *Proj-5-def* **by** (*simp add*: *norm-is-Proj*)

theorem *mixed-o2h-nonterm-5*:
shows
|*mixed-H.Pleft P* − (*mixed-H.Pright Proj-5*)| ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
and
|*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-5*)| ≤
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
proof −
have |*mixed-H.Pleft P* − *mixed-H.Pright Proj-5*| =
|*mixed-H.Pleft' Proj-5* − *mixed-H.Pright Proj-5*|
unfolding *Proj-5-altdef Proj-3-def mixed-H.Pleft-Pleft'-case5*[*OF mixed-H.is-Proj-P*]
mixed-H.Pfind-Pright Fst-def **by** *auto*
also have ... ≤ *2* ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
using *mixed-H.estimate-Pfind*[*OF norm-Proj-5*] **by** *auto*
finally show |*mixed-H.Pleft P* − *mixed-H.Pright Proj-5*| ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
by *auto*
have |*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-5*)| =
|*sqrt* (*mixed-H.Pleft' Proj-5*) − *sqrt* (*mixed-H.Pright Proj-5*)|

unfolding *mixed-H.Pleft-Pleft'-case5*[*OF mixed-H.is-Proj-P*] *Proj-5-altdef Proj-3-def* **by** *auto*
also have ... $\leq$ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
using *mixed-H.estimate-Pfind-sqrt*[*OF norm-Proj-5*] **by** *auto*
finally show |*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-5*)| $\leq$
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
by *auto*
qed

theorem *mixed-o2h-term-5*:
assumes ⋀*i*. *i*<*d*+*1* ⟹ *km-trace-preserving* (*kf-apply* (*E i*))
shows
|*mixed-H.Pleft P* − *mixed-H.Pright Proj-5*| $\leq$
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
and
|*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-5*)| $\leq$
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
proof −
have *normHright*: *norm* (*mixed-H.ϱright E*) = *1*
by (*rule mixed-H.trace-preserving-norm-ϱright*[*OF trace-preserving-kf-Fst*[*OF assms*]])
have *normHcount*: *norm* (*mixed-H.ϱcount E*) = *1*
by (*rule mixed-H.trace-preserving-norm-ϱcount*[*OF trace-preserving-kf-Fst*[*OF assms*]])
have *terminH*: *mixed-H.P-nonterm E* = *0*
unfolding *mixed-H.P-nonterm-def* **using** *normHright normHcount*
by (*metis cmod-Re complex-of-real-nn-iff mixed-H.ϱcount-pos mixed-H.ϱright-pos norm-le-zero-iff*

norm-pths(*2*) *norm-tc-pos norm-zero of-real-0*)
show |*mixed-H.Pleft P* − *mixed-H.Pright Proj-5*| $\leq$
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
using *mixed-o2h-nonterm-5*(*1*) *terminH* **by** *auto*
show |*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-5*)| $\leq$
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
using *mixed-o2h-nonterm-5*(*2*) *terminH* **by** *auto*
qed

Theorem 1, definition of Pright (6)

lemma *Pright-G-H-case5-nonterm*:
mixed-H.Pright Proj-5 − *mixed-G.Pright Proj-5* = *norm* (*mixed-H.ϱright E*) − *norm* (*mixed-G.ϱright E*)
proof −
have *mixed-H.Pright Proj-5* − *mixed-G.Pright Proj-5* =
Re (*trace-tc* (*compose-tcr Proj-5* (*mixed-H.ϱright E*))) −
Re (*trace-tc* (*compose-tcr Proj-5* (*mixed-G.ϱright E*)))
unfolding *mixed-H.Pright-def mixed-G.Pright-def mixed-H.PM-altdef mixed-G.PM-altdef*
by (*smt* (*verit*, *best*) *cblinfun-assoc-left*(*1*) *circularity-of-trace compose-tcr.rep-eq*
from-trace-class-sandwich-tc is-Proj-5 is-Proj-algebraic sandwich-apply trace-class-from-trace-class

trace-tc.rep-eq)
also have ... = *Re* (*trace-tc* (*compose-tcr Proj-3* (*mixed-H.ϱright E*))) +

Re (*trace-tc* (*compose-tcr mixed-G.end-measure* (*mixed-H.ϱright E*))) −
Re (*trace-tc* (*compose-tcr Proj-3* (*mixed-G.ϱright E*))) −
Re (*trace-tc* (*compose-tcr mixed-G.end-measure* (*mixed-G.ϱright E*)))
unfolding *Proj-5-altdef* **by** (*auto simp add*: *compose-tcr.add-left trace-tc-plus*)
also have ... = *Re* (*trace-tc* (*sandwich-tc* ($P \otimes_o$ *selfbutter* (*ket empty*)) (*mixed-H.ϱright E*)))
+
Re (*trace-tc* (*compose-tcr mixed-G.end-measure* (*mixed-H.ϱright E*))) −
Re (*trace-tc* (*sandwich-tc* ($P \otimes_o$ *selfbutter* (*ket empty*)) (*mixed-G.ϱright E*))) −
Re (*trace-tc* (*compose-tcr mixed-G.end-measure* (*mixed-G.ϱright E*)))
by (*smt* (*verit*, *del-insts*) *Proj-3-def cblinfun-assoc-left*(*1*) *circularity-of-trace compose-tcr.rep-eq*

from-trace-class-sandwich-tc is-Proj-3 is-Proj-algebraic sandwich-apply trace-class-from-trace-class

trace-tc.rep-eq)
also have ... = *mixed-H.Pright Proj-3* − *mixed-G.Pright Proj-3* + *Re* (*mixed-H.Pfind E* −
mixed-G.Pfind E)
by (*simp add*: *Pright-G-H-same Proj-3-def ϱright-G-H-same mixed-G.Pfind-def mixed-H.Pfind-def*)
also have ... = *Re* (*norm* (*mixed-H.ϱright E*) − *norm* (*mixed-G.ϱright E*))
unfolding *Proj-3-def* **by** (*simp add*: *Pfind-G-H-same-nonterm Pright-G-H-same*)
finally show *?thesis* **by** *auto*
qed

lemma *Pright-G-H-case5*:
assumes $\bigwedge i.\ i<d+1 \Longrightarrow$ *km-trace-preserving* (*kf-apply* (*E i*))
shows *mixed-H.Pright Proj-5* = *mixed-G.Pright Proj-5*
proof −
have *normHright*: *norm* (*mixed-H.ϱright E*) = *1*
by (*rule mixed-H.trace-preserving-norm-ϱright*[*OF trace-preserving-kf-Fst*[*OF assms*]])
moreover have *normGright*: *norm* (*mixed-G.ϱright E*) = *1*
by (*rule mixed-G.trace-preserving-norm-ϱright*[*OF trace-preserving-kf-Fst*[*OF assms*]])
ultimately show *?thesis* **using** *Pright-G-H-case5-nonterm* **by** *auto*
qed

theorem *mixed-o2h-nonterm-6*:
shows
|*mixed-H.Pleft P* − *mixed-G.Pright Proj-5*| ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*) +
|*norm* (*mixed-H.ϱright E*) − *norm* (*mixed-G.ϱright E*)|

proof −
have |*mixed-H.Pleft P* − *mixed-G.Pright Proj-5*| =
|*mixed-H.Pleft P* − *mixed-H.Pright Proj-5* + *norm* (*mixed-H.ϱright E*) − *norm* (*mixed-G.ϱright*
E)|
using *Pright-G-H-case5-nonterm* **by** *auto*
also have ... ≤ |*mixed-H.Pleft′ Proj-5* − *mixed-H.Pright Proj-5*| +
|*norm* (*mixed-H.ϱright E*) − *norm* (*mixed-G.ϱright E*)|
using *Proj-3-def Proj-5-altdef mixed-G.is-Proj-P mixed-H.Pleft-Pleft′-case5* **by** *force*

also have ... $\leq$ *2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E) +*
|norm (mixed-H.ϱright E) − norm (mixed-G.ϱright E)|
using *mixed-H.estimate-Pfind[OF norm-Proj-5]* **by** *auto*
finally show *|mixed-H.Pleft P − mixed-G.Pright Proj-5| ≤*
2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E) +
|norm (mixed-H.ϱright E) − norm (mixed-G.ϱright E)|
by *auto*
qed

theorem *mixed-o2h-term-6*:
assumes *⋀i. i<d+1 ⟹ km-trace-preserving (kf-apply (E i))*
shows
|mixed-H.Pleft P − mixed-G.Pright Proj-5| ≤
2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
and
|sqrt (mixed-H.Pleft P) − sqrt (mixed-G.Pright Proj-5)| ≤
sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
proof −
have *normHright*: *norm (mixed-H.ϱright E) = 1*
by *(rule mixed-H.trace-preserving-norm-ϱright[OF trace-preserving-kf-Fst[OF assms]])*
have *normGright*: *norm (mixed-G.ϱright E) = 1*
by *(rule mixed-G.trace-preserving-norm-ϱright[OF trace-preserving-kf-Fst[OF assms]])*
have *normHcount*: *norm (mixed-H.ϱcount E) = 1*
by *(rule mixed-H.trace-preserving-norm-ϱcount[OF trace-preserving-kf-Fst[OF assms]])*
have *terminH*: *mixed-H.P-nonterm E = 0*
unfolding *mixed-H.P-nonterm-def* **using** *normHright normHcount*
by *(metis cmod-Re complex-of-real-nn-iff mixed-H.ϱcount-pos mixed-H.ϱright-pos norm-le-zero-iff*

norm-pths(2) norm-tc-pos norm-zero of-real-0)
show *|mixed-H.Pleft P − mixed-G.Pright Proj-5| ≤*
2 ∗ sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
using *mixed-o2h-nonterm-6(1) terminH* **unfolding** *normHright normGright* **by** *auto*

have *nonterm-zero*: *mixed-H.P-nonterm E = 0*
unfolding *mixed-H.P-nonterm-def* **using** *normHright normHcount*
mixed-H.P-nonterm-def terminH **by** *presburger*
have *|sqrt (mixed-H.Pleft P) − sqrt (mixed-G.Pright Proj-5)| =*
|sqrt (mixed-H.Pleft′ Proj-5) − sqrt (mixed-H.Pright Proj-5)|
using *Pright-G-H-case5[OF assms, symmetric]*
unfolding *mixed-H.Pleft-Pleft′-case5[OF mixed-H.is-Proj-P]*
Proj-5-altdef Proj-3-def **by** *auto*
also have ... *≤ sqrt ((d+1) ∗ Re (mixed-H.Pfind E) + d∗ mixed-H.P-nonterm E)*
using *mixed-H.estimate-Pfind-sqrt[OF norm-Proj-5]* **by** *auto*
finally show *|sqrt (mixed-H.Pleft P) − sqrt (mixed-G.Pright Proj-5)| ≤*
sqrt ((d+1) ∗ Re (mixed-H.Pfind E))
using *nonterm-zero* **by** *auto*
qed

end

unbundle *no cblinfun-syntax*
unbundle *no lattice-syntax*
unbundle *no register-syntax*

end

References

[1] A. Ambainis, M. Hamburg, and D. Unruh. *Quantum Security Proofs Using Semi-classical Oracles*, page 269295. Springer International Publishing, 2019.

[2] K. Heidler and D. Unruh. Formalizing the one-way to hiding theorem. In K. Stark, A. Timany, S. Blazy, and N. Tabareau, editors, *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025*, pages 243–256. ACM, 2025.