The Oneway to Hiding Theorem*

Katharina Heidler Dominique Unruh

February 6, 2026

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*) ⟷ (*g* ∘ *h*) *abs-summable-on A*›
⟨*proof*⟩

lemma *abs-summable-norm*:
assumes ‹*f abs-summable-on A*›
shows ‹(λ*x*. *norm* (*f x*)) *abs-summable-on A*›
⟨*proof*⟩

lemma *abs-summable-on-add*:
assumes ‹*f abs-summable-on A*› **and** ‹*g abs-summable-on A*›
shows ‹(λ*x*. *f x* + *g x*) *abs-summable-on A*›
⟨*proof*⟩

lemma *sandwich-tc-has-sum*:
assumes (*f has-sum x*) *A*
shows ((*sandwich-tc* ϱ *o f*) *has-sum sandwich-tc* ϱ *x*) *A*
⟨*proof*⟩

lemma *sandwich-tc-abs-summable-on*:
assumes *f abs-summable-on A*
shows (*sandwich-tc* ϱ *o f*) *abs-summable-on A*
⟨*proof*⟩

lemma *trace-tc-abs-summable-on*:
assumes *f abs-summable-on A*
shows (*trace-tc o f*) *abs-summable-on A*
⟨*proof*⟩

Defining a self butterfly on trace class.

lemma *trace-selfbutter-norm*:
trace (*selfbutter A*) = *norm A* ^2
⟨*proof*⟩

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

lemma *norm-tc-selfbutter*[*simp*]:
norm (*tc-selfbutter a*) = (*norm a*)^2
⟨*proof*⟩

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

lemma *norm-sandwich-tc-unitary*:
assumes *isometry U* ϱ ≥ *0*
shows *norm* (*sandwich-tc U* ϱ) = *norm* ϱ
⟨*proof*⟩

Lemmas on *trace-tc*

lemma *trace-tc-minus*:
trace-tc (*a*−*b*) = *trace-tc a* − *trace-tc b*
⟨*proof*⟩

lemma *trace-tc-sum*:
trace-tc (*sum a I*) = (∑ *i*∈*I*. *trace-tc* (*a i*))
⟨*proof*⟩

lemma *selfbutter-sandwich*:
fixes *A* :: *'a ell2* ⇒$_{CL}$ *'a ell2* **and** *B* :: *'a ell2*
shows *selfbutter* (*A* ∗$_V$ *B*) = *sandwich A* ∗$_V$ *selfbutter B*
⟨*proof*⟩

lemma *tc-tensor-sum-left*:
tc-tensor (*sum g S*) *x* = (∑ *i*∈*S*. *tc-tensor* (*g i*) *x*)
⟨*proof*⟩

lemma *tc-tensor-sum-right*:
tc-tensor x (*sum g S*) = (∑ *i*∈*S*. *tc-tensor x* (*g i*))
⟨*proof*⟩

lemma *complex-of-real-abs*: *complex-of-real* |*f*| = |*complex-of-real f*|
⟨*proof*⟩

Additional Lemmas on Tensors

lemma *tensor-op-padding*:
(*A* ⊗$_o$ *B*) ∗$_V$ *v* = (*A* ⊗$_o$ *id-cblinfun*) ∗$_V$ (*id-cblinfun* ⊗$_o$ *B*) ∗$_V$ *v*
⟨*proof*⟩

lemma *tensor-op-padding'*:
(*A* ⊗$_o$ *B*) ∗$_V$ *v* = (*id-cblinfun* ⊗$_o$ *B*) ∗$_V$ (*A* ⊗$_o$ *id-cblinfun*) ∗$_V$ *v*
⟨*proof*⟩

lemma *tensor-op-left-minus*: (*x* − *y*) ⊗$_o$ *b* = *x* ⊗$_o$ *b* − *y* ⊗$_o$ *b*
⟨*proof*⟩

lemma *tensor-op-right-minus*: *b* ⊗$_o$ (*x* − *y*) = *b* ⊗$_o$ *x* − *b* ⊗$_o$ *y*
⟨*proof*⟩

lemma *id-cblinfun-selfbutter-tensor-ell2-right*:
id-cblinfun ⊗$_o$ *selfbutter* (*ket i*) = (*tensor-ell2-right* (*ket i*)) *o*$_{CL}$ (*tensor-ell2-right* (*ket i*)∗)
⟨*proof*⟩

A lot lof lemmas on limit processes with several functions.

lemma *tendsto-Re*:
assumes (*f* ⟶ *x*) *F*
shows ((λ*x*. *Re* (*f x*)) ⟶ *Re x*) *F*
⟨*proof*⟩

lemma *tendsto-tc-tensor*:
assumes (*f* ⟶ *x*) *F*
shows ((λ*x*. *tc-tensor* (*f x*) *a*) ⟶ *tc-tensor x a*) *F*

⟨*proof*⟩

lemma *tc-tensor-has-sum*:
assumes (*f has-sum x*) *A*
shows ((λ*y*. *tc-tensor y a*) *o f has-sum tc-tensor x a*) *A*
⟨*proof*⟩

lemma *Re-has-sum*:
assumes (*f has-sum x*) *A*
shows ((λ*n*. *Re n*) *o f has-sum Re x*) *A*
⟨*proof*⟩

Relationship norm and condition

lemma *norm-1-to-cond*:
fixes *A* :: *'a ell2* ⇒$_{CL}$ *'a ell2*
assumes *norm A* ≤ *1*
shows *A*∗ *o*$_{CL}$ *A* ≤ *id-cblinfun*
⟨*proof*⟩

lemma *cond-to-norm-1*:
fixes *A* :: *'a ell2* ⇒$_{CL}$ *'a ell2*
assumes *A*∗ *o*$_{CL}$ *A* ≤ *id-cblinfun*
shows *norm A* ≤ *1*
⟨*proof*⟩

Arithmetic mean - quadratic mean inequality (AM-QM)

lemma *arith-quad-mean-ineq*:
fixes *n*::*nat* **and** *x* :: *nat* ⇒ *real*
assumes ⋀*i*. *i*∈*I* ⟹ *x i* ≥ *0*
shows (∑ *i*∈*I*. *x i*)^2 ≤ (*card I*) ∗ (∑ *i*∈*I*. (*x i*) ^2)
⟨*proof*⟩

lemma *sqrt-binom*:
assumes *a* ≥ *0 b* ≥ *0*
shows |*a* − *b*| = |*sqrt a* − *sqrt b*| ∗ |*sqrt a* + *sqrt b*|
⟨*proof*⟩

Lemmas on *sandwich-tc*

lemma *sandwich-tc-compose'*:
sandwich-tc (*A o*$_{CL}$ *B*) ϱ = *sandwich-tc A* (*sandwich-tc B* ϱ)
⟨*proof*⟩

lemma *sandwich-tc-sum*:
sandwich-tc E (*sum f A*) = *sum* (*sandwich-tc E o f*) *A*
⟨*proof*⟩

lemma *isCont-sandwich-tc*:
isCont (*sandwich-tc A*) *x*
⟨*proof*⟩

lemma *bounded-linear-trace-norm-sandwich-tc*:
bounded-linear (λ*y*. *trace-tc* (*sandwich-tc E y*))
⟨*proof*⟩

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*∗
⟨*proof*⟩

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*∗)
⟨*proof*⟩

lemma *sandwich-add2*:
sandwich A (*B*+*C*) = *sandwich A B* + *sandwich A C*
⟨*proof*⟩

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

lemma *splitting-Proj-and*:
assumes *is-Proj P is-Proj Q*
shows *Proj* (((*P* ⊗$_o$ *id-cblinfun*) ∗$_S$ ⊤) ⊓ ((*id-cblinfun* ⊗$_o$ *Q*) ∗$_S$ ⊤)) = *P* ⊗$_o$ *Q*
⟨*proof*⟩

lemma *splitting-Proj-or*:
assumes *is-Proj P is-Proj Q*
shows *Proj* (((*P* ⊗$_o$ *id-cblinfun*) ∗$_S$ ⊤) ⊔ ((*id-cblinfun* ⊗$_o$ *Q*) ∗$_S$ ⊤)) =
P ⊗$_o$ (*id-cblinfun* − *Q*) + *id-cblinfun* ⊗$_o$ *Q*
⟨*proof*⟩

Additional stuff

lemma *Union-cong*:
assumes ⋀*i*. *i*∈*A* ⟹ *f i* = *g i*
shows (⋃ *i*∈*A*. *f i*) = (⋃ *i*∈*A*. *g i*)
⟨*proof*⟩

lemma *proj-ket-apply*:
proj (*ket i*) ∗$_V$ *ket j* = (*if i*=*j then ket i else* 0)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *clinear-of-complex*[*iff*]: ‹*clinear of-complex*›
⟨*proof*⟩

lemma *inj-on-CARD-1*[*iff*]: ‹*inj-on f X*› **for** *X* :: ‹*'a*::*CARD-1 set*›
⟨*proof*⟩

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* ⇒$_{CL}$ (*'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*
  ⟨*proof*⟩

lemma *blog-flip*: $i<d \Longrightarrow blog\ l \Longrightarrow blog\ (flip\ i\ l)$
  ⟨*proof*⟩

lemma *blog-valid*:
  $blog\ l \Longrightarrow valid\ l$
  ⟨*proof*⟩

lemma *flip-flip*: $i<d \Longrightarrow blog\ l \Longrightarrow flip\ i\ (flip\ i\ l) = l$
  ⟨*proof*⟩

lemma *bit-flip-same*: $i<d \Longrightarrow blog\ l \Longrightarrow bit\ (flip\ i\ l)\ i = (\neg\ (bit\ l\ i))$
  ⟨*proof*⟩

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

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*⟩

lemma *finite-len-d-lists*[*simp*]:
finite len-d-lists
⟨*proof*⟩

lemma *blog-list-to-l*:
assumes *length ls* ≤ *d*
shows *blog* (*list-to-l ls*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)])
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *empty-list-to-l* [*simp*]:
list-to-l empty-list = *empty*
⟨*proof*⟩

lemma *empty-list-len-d*[*simp*]:
empty-list ∈ *len-d-lists*
⟨*proof*⟩

lemma *empty-list-to-l-elem* [*simp*]:
empty ∈ *list-to-l* ‘ *len-d-lists*
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *surj-list-to-l*: *list-to-l ` len-d-lists = Collect blog*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *list-to-l-eq*:
assumes *list-to-l xs* = *list-to-l ys length xs* = *d length ys* = *d*
shows *xs* = *ys*
⟨*proof*⟩

lemma *inj-list-to-l*: *inj-on list-to-l* (*len-d-lists*)
⟨*proof*⟩

lemma *bij-betw-list-to-l*: *bij-betw list-to-l len-d-lists* (*Collect blog*)
⟨*proof*⟩

lemma *card-blog*: *card* (*Collect blog*) = *2^d*
⟨*proof*⟩

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 {} = {}
⟨*proof*⟩

lemma *has-bits-not-empty*:
assumes *y* ∈ *has-bits A A*≠{} *y*∈*len-d-lists*
shows *list-to-l y* ≠ *empty*
⟨*proof*⟩

lemma *has-bits-empty-list*:

empty-list $\notin$ *has-bits* $\{0..<d\}$
⟨*proof*⟩

lemma *has-bits-incl*:
assumes $A \subseteq B$
shows *has-bits* $A \subseteq$ *has-bits* B
⟨*proof*⟩

lemma *has-bits-in-len-d-lists*[*simp*]:
has-bits $A \subseteq$ *len-d-lists*
⟨*proof*⟩

lemma *finite-has-bits*[*simp*]:
finite (*has-bits* A)
⟨*proof*⟩

lemma *has-bits-not-elem*:
assumes $y \in$ *has-bits* A $A \neq \{\}$ $A \subseteq \{0..<d\}$ $y \in$ *len-d-lists* $n \notin A$ $n<d$
shows $y[d-n-1:=\neg y!(d-n-1)] \in$ *has-bits* A
⟨*proof*⟩

lemma *has-bits-split-Suc*:
assumes $n<d$
shows *has-bits* $\{n..<d\}$ = *has-bits* $\{n\} \cup$ *has-bits* $\{Suc\ n..<d\}$
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *has-bits-elem*:
assumes $x \in$ *len-d-lists* $-$ *has-bits* A $a \in A$
shows $\neg x!(d-a-1)$
⟨*proof*⟩

lemma *has-bits-upto-elem*:
assumes $x \in$ *has-bits-upto* n $n<d$
shows $\neg x!(d-n-1)$
⟨*proof*⟩

lemma *has-bits-upto-incl*:
assumes $n \leq m$
shows *has-bits-upto* $n \subseteq$ *has-bits-upto* m
⟨*proof*⟩

lemma *has-bits-upto-d*:
has-bits-upto d = len-d-lists
⟨*proof*⟩

lemma *empty-list-has-bits-upto*:
empty-list ∈ has-bits-upto n
⟨*proof*⟩

lemma *empty-list-to-l-has-bits-upto*:
empty ∈ list-to-l ‘ has-bits-upto n
⟨*proof*⟩

lemma *len-d-empty-has-bits*:
len-d-lists − {empty-list} = has-bits {0..<d}
⟨*proof*⟩

Properties of d

lemma *two-d-gr-1*:
2^d > (1::nat)
⟨*proof*⟩

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

lemma *valid-inv*: *− Collect valid = valid −‘ {False}* ⟨*proof*⟩

lemma *blog-inv*: *− Collect blog = blog −‘ {False}* ⟨*proof*⟩

lemma *not-blog-flip*: *i<d ⟹ (¬ blog l) ⟹ (¬ blog (flip i l))*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *register-Y*:
register Y
⟨*proof*⟩

lemma *X-0*:
X 0 = 0 ⟨*proof*⟩

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)
⟨*proof*⟩

lemma *proj-classical-set-split-id*:
id-cblinfun = *proj-classical-set* M + *proj-classical-set* $(-M)$
⟨*proof*⟩

lemma *proj-classical-set-sum-ket-finite*:
assumes *finite* A
shows *proj-classical-set* $A = (\sum i{\in}A.$ *selfbutter* (*ket* $i))$
⟨*proof*⟩

lemma *proj-classical-set-not-elem*:
assumes $i{\notin}A$
shows *proj-classical-set* $A *_V$ *ket* $i = 0$
⟨*proof*⟩

lemma *proj-classical-set-elem*:
assumes $i{\in}A$
shows *proj-classical-set* $A *_V$ *ket* i = *ket* i
⟨*proof*⟩

lemma *proj-classical-set-upto*:
assumes $i{<}j$
shows *proj-classical-set* $\{j..\} *_V$ *ket* $(i{::}nat) = 0$
⟨*proof*⟩

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* $i)$
⟨*proof*⟩

lemma *proj-classical-set-split-Suc*:
proj-classical-set $\{n..\}$ = *proj* (*ket* n) + *proj-classical-set* $\{Suc\ n..\}$
⟨*proof*⟩

lemma *proj-classical-set-union*:
assumes $\bigwedge x\ y.\ x \in ket\ `\ A \Longrightarrow y \in ket\ `\ B \Longrightarrow$ *is-orthogonal* $x\ y$
shows *proj-classical-set* $(A \cup B)$ = *proj-classical-set* A + *proj-classical-set* B
⟨*proof*⟩

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

definition *Proj-ket-set* :: $'a$ *set* ⇒ $('mem \times 'a)$ *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*
⟨*proof*⟩

definition *Proj-ket-upto* :: *bool list set* ⇒ (′*mem* × ′*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-to-l y*)) *= v* $\otimes_s$ *ket* (*list-to-l y*)
⟨*proof*⟩

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*⟩

lemma *S-embed-not-S-embed-add*:
S-embed S′ (*ket a*) + *not-S-embed S′* (*ket a*) = *ket a*
⟨*proof*⟩

lemma *S-embed-idem* [*simp*]:
S-embed S′ o_{CL} *S-embed S′ = S-embed S′*
⟨*proof*⟩

lemma *S-embed-adj*:
(*S-embed S′*)∗ = *S-embed S′*
⟨*proof*⟩

lemma *not-S-embed-idem*:
not-S-embed S′ o_{CL} *not-S-embed S′ = not-S-embed S′*
⟨*proof*⟩

lemma *not-S-embed-adj*:
(*not-S-embed S′*)∗ = *not-S-embed S′*
⟨*proof*⟩

lemma *not-S-embed-S-embed* [*simp*]:
not-S-embed S′ o_{CL} *S-embed S′ = 0*
⟨*proof*⟩

lemma *S-embed-not-S-embed* [*simp*]:
S-embed S′ o_{CL} *not-S-embed S′ = 0*
⟨*proof*⟩

lemma *not-S-embed-Proj*:
not-S-embed S = Proj (not-S-embed S $*_S$ ⊤)
⟨*proof*⟩

lemma *not-S-embed-in-X-image*:
assumes *a ∈ space-as-set (− (not-S-embed S $*_S$ ⊤))*
shows *(not-S-embed S)$*_V$ a = 0*
⟨*proof*⟩

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-ell2-right (ket i)*) $*_V$ v*

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

lemma Ψ*s-id-cblinfun*:
$\Psi s\ a\ ((A \otimes_o$ *id-cblinfun*$)\ *_V\ v) = A\ *_V\ (\Psi s\ a\ v)$
⟨*proof*⟩

Additional Lemmas

lemma *id-cblinfun-tensor-split-finite*:
assumes *finite A*
shows *(id-cblinfun:: ('mem × 'a) ell2 ⇒$_{CL}$ ('mem × 'a) ell2) =*
($\sum$ i∈A. (tensor-ell2-right (ket i)) o_{CL} (tensor-ell2-right (ket i))) +*
Proj-ket-set (−A)
⟨*proof*⟩

Lemmas on sums of butterflys

lemma *sum-butterfly-ket0*:
assumes *(y::nat)<d+1*
shows *($\sum$ i<d+1. butterfly (ket 0) (ket i)) $*_V$ (ket y) = ket 0*
⟨*proof*⟩

lemma *sum-butterfly-ket0'*:
($\sum$ i<d+1. butterfly (ket 0) (ket i))$*_V$ proj-classical-set {..<d+1} $*_V$ y =
($\sum$ i<d+1. Rep-ell2 y i) $*_C$ ket 0
for *y::nat ell2*
⟨*proof*⟩

The oracle query is a unitary.

lemma *inj-Uquery-map*:
inj (λ(x, (y::'y)). (x, y + H x))
⟨*proof*⟩

lemma *classical-operator-exists-Uquery*:

classical-operator-exists (*Some o* (λ(*x*,(*y*::'*y*)). (*x*, *y* + (*H x*))))
⟨*proof*⟩

lemma *Uquery-ket*:
Uquery F ∗$_V$ *ket* (*a*::'*x*) ⊗$_s$ *ket* (*b*::'*y*) = *ket a* ⊗$_s$ *ket* (*b* + *F a*)
⟨*proof*⟩

lemma *unitary-H*: *unitary* (*Uquery* (*H*::'*x* ⇒ '*y*))
⟨*proof*⟩

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*
⟨*proof*⟩

lift-definition *kf-Fst* :: ('*a ell2*, '*c ell2*, *unit*) *kraus-family* ⇒
(('*a* × '*b*) *ell2*, ('*c* × '*b*) *ell2*, *unit*) *kraus-family* **is**
λ*E*. (λ(*x*,-). (*x* ⊗$_o$ *id-cblinfun*, ())) ‘ *E*
⟨*proof*⟩

lemma *summable-on-in-kf-Fst*:
fixes *f* :: '*c* ⇒ '*a ell2* ⇒$_{CL}$ '*a ell2*
and *b* :: '*b ell2* ⇒$_{CL}$ '*b ell2*
shows *summable-on-in cweak-operator-topology* (λ*x*. (*fst x*∗ *o*$_{CL}$ *fst x*) ⊗$_o$ *id-cblinfun*) (*Rep-kraus-family G*)
⟨*proof*⟩

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* $(\lambda x.\ (fst\ x*\ o_{CL}\ fst\ x) \otimes_o id\text{-}cblinfun)$ *(Rep-kraus-family G)* $\leq$
(infsum-in cweak-operator-topology $(\lambda x.\ fst\ x*\ o_{CL}\ fst\ x)$ *(Rep-kraus-family G)) ⊗o id-cblinfun*
⟨*proof*⟩

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*⟩

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*⟩

kraus family Fst is trace preserving.

lemma *kf-apply-Fst-tensor*:
‹kf-apply (kf-Fst 𝔈 ::(('c × 'b) ell2, ('a × 'b) ell2, unit) kraus-family)
(tc-tensor ϱ σ) = tc-tensor (kf-apply 𝔈 ϱ) σ›
⟨*proof*⟩

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*⟩

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*⟩

Summability on Kraus maps

lemma *finite-kf-apply-has-sum*:
assumes *(f has-sum x) A*
shows *((kf-apply 𝔉 o f) has-sum kf-apply 𝔉 x) A*
⟨*proof*⟩

lemma *finite-kf-apply-abs-summable-on*:
assumes *f abs-summable-on A*
shows *(kf-apply 𝔉 o f) abs-summable-on A*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *isometry-Ub*:
isometry (*Ub k*)
⟨*proof*⟩

lemma *Ub-ket-range*: (*Ub i* $*_V$ *ket y*) ∈ *range ket* ⟨*proof*⟩

lemma *Ub-ket*:
Ub k $*_V$ (*ket x*) = *ket* (*flip k x*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *norm-US*:
assumes *i* < *Suc d* **shows** *norm* (*US S i*) = *1*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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 $U\text{-}S'$

For the definition of $U\text{-}S'$, 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*⟩

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

lemma *inj-c-add*: *inj c-add*
⟨*proof*⟩

lemma *surj-c-add*: *c-add* ‘ *UNIV* = *UNIV*
⟨*proof*⟩

lemma *bij-c-add*: *bij c-add*
⟨*proof*⟩

lemma *c-add-0*: *c-add 0* ≠ *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *unitary-Uc*:
unitary Uc
⟨*proof*⟩

lemma *Uc-ket-d*:
Uc ∗$_V$ *ket d* = *ket 0*
⟨*proof*⟩

lemma *Uc-ket-less*:
assumes *n*<*d*
shows *Uc* ∗$_V$ *ket n* = *ket* (*n*+*1*)
⟨*proof*⟩

lemma *Uc-ket-leq*:
assumes *n*<*d*+*1*
shows *Uc* ∗$_V$ *ket n* = *ket* ((*n*+*1*) *mod* (*d*+*1*))
⟨*proof*⟩

lemma *Uc-ket-greater*:
assumes $n>d$
shows $Uc *_V ket\ n = ket\ n$
⟨*proof*⟩

lemma *Uc-ket-range*:
$(Uc *_V ket\ y) \in range\ ket$ ⟨*proof*⟩

lemma *Uc-ket-range-valid*:
assumes $y<d+1$
shows $(Uc *_V ket\ y) \in ket\ `\ \{..<d+1\}$ ⟨*proof*⟩

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* ⊗$_o$ *Uc* + *not-S-embed S* ⊗$_o$ *id-cblinfun*›

lemma *unitary-U-S'*:
unitary (*U-S' S*)
⟨*proof*⟩

lemma *iso-U-S'*: *isometry* (*U-S' S*)
⟨*proof*⟩

lemma *U-S'-ket-split*:
U-S' S $*_V$ *ket* (x,y) = (*S-embed S* $*_V$ *ket x*) ⊗$_s$ (*Uc* $*_V$ *ket y*) + (*not-S-embed S* $*_V$ *ket x*) ⊗$_s$ *ket y*
⟨*proof*⟩

lemma *norm-U-S'*:
assumes $i <$ *Suc d* **shows** *norm* (*U-S' S*) = *1*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*: ⋀*i*. *i* < *d+1* ⟹ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma (**in** *o2h-setting*) *trace-preserving-norm-run-mixed-adv*:
assumes *trace-pres*: ⋀*i*. *i* < *d+1* ⟹ *km-trace-preserving* (*kf-apply* (*Es i*))
and *n*: *n* < *d+1*
and *iso-UB*: ⋀*i*. *i* < *d+1* ⟹ *isometry* (*UB i*)
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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *run-pure-A-update-tc*:
run-pure-A-tc UA H = *Abs-trace-class* (*run-pure-A-update UA H*)
⟨*proof*⟩

lemma *run-pure-A-tc-pos*: *0* ≤ *run-pure-A-tc UA H*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*register*]: ‹*register Y-for-B*›
⟨*proof*⟩

lemma *register-XY-for-B*:
register (*X-for-B*;*Y-for-B*) ⟨*proof*⟩

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

lemma *UqueryH-tensor-id-cblinfunB*:
(*X-for-B*;*Y-for-B*) (*Uquery H*) = (*X*;*Y*) (*Uquery H*) ⊗$_o$ *id-cblinfun*
⟨*proof*⟩

The oracle query on the extended register stays unitary.

lemma *unitary-H-B*: *unitary* ((*X-for-B*;*Y-for-B*) (*Uquery H*))
⟨*proof*⟩

lemma *iso-H-B*: *isometry* ((*X-for-B*;*Y-for-B*) (*Uquery H*))
⟨*proof*⟩

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* ⊗$_s$ *ket empty*›

lemma *norm-init-B*: *norm init-B* = 1
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *run-pure-B-update-tc*:
run-pure-B-tc UB H S = *Abs-trace-class* (*run-pure-B-update UB H S*)
⟨*proof*⟩

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

lemma *run-pure-B-tc-pos*: *0* ≤ *run-pure-B-tc UB H S*
⟨*proof*⟩

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 ≤ *run-mixed-B kraus-B H S*
⟨*proof*⟩

lemma *norm-run-mixed-B*:
assumes *F-norm-id*: ⋀*i*. *i* < *d*+*1* ⟹ *kf-bound* (*F i*) ≤ *id-cblinfun*
shows *norm* (*run-mixed-B F H S*) ≤ *1*
⟨*proof*⟩

lemma *trace-preserving-norm-run-mixed-B*:
assumes ⋀*i*. *i* < *d*+*1* ⟹ *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*
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*register*]: ‹*register Y-for-C*›
⟨*proof*⟩

lemma *register-XY-for-C*:
register (*X-for-C*; *Y-for-C*) ⟨*proof*⟩

The oracle query on the extended register stays unitary.

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

lemma *iso-H-C*: *isometry* ((*X-for-C*; *Y-for-C*) (*Uquery H*))
⟨*proof*⟩

Alternative representation of *Uquery H*.

lemma *UqueryH-tensor-id-cblinfunC*:
(*X-for-C*; *Y-for-C*) (*Uquery H*) = (*X*; *Y*) (*Uquery H*) ⊗$_o$ *id-cblinfun*
⟨*proof*⟩

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* ⊗$_s$ *ket 0*›

lemma *norm-init-B-count*:
norm (*init-B-count*) = *1*
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *run-pure-B-count-tc-pos*: *0 ≤ run-pure-B-count-tc UB H S*
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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-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* ⇒ ('*y*::*group-add*)›
and *S* :: ‹'*x* ⇒ *bool*›
and *UA* :: ‹*nat* ⇒ '*mem update*›

— All operations by the adversary *A* must be isometries.
assumes *norm-UA*: ‹⋀*i*. *i*<*d*+*1* ⟹ *norm* (*UA i*) ≤ *1*›
begin

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

definition ‹*run-A* = *run-pure-adv d UA* (λ-. *id-cblinfun*::'*mem update*) *init X Y H*›

lemma *norm-UA-Suc*: *n* <*d* ⟹ *norm* (*UA* (*Suc n*)) ≤ *1*
⟨*proof*⟩

lemma *norm-UA-0-init*:
norm (*UA 0* ∗$_V$ *init*) ≤ *1*
⟨*proof*⟩

lemma *tensor-proj-UA-tensor-commute*:
(*id-cblinfun* $\otimes_o$ *proj-classical-set A*) o_{CL} (*UA* (*Suc n*) $\otimes_o$ *id-cblinfun*) =
(*UA* (*Suc n*) $\otimes_o$ *id-cblinfun*) o_{CL} (*id-cblinfun* $\otimes_o$ *proj-classical-set A*)
⟨*proof*⟩

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-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* (λ*i*. *UA i* $\otimes_o$ *id-cblinfun*) (*US S*) *init-B X-for-B Y-for-B H*›

definition ‹*run-B* = *run-pure-adv d* (λ*i*. *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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*) ≤ *1*
⟨*proof*⟩

lemma *norm-run-B*:
norm run-B ≤ *1*
⟨*proof*⟩

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*≤*d*
shows *Proj-ket-set* (− *Collect blog*) $*_V$ *run-B-upto n* = *0*
⟨*proof*⟩

lemma *orth-run-B-upto*:
fixes *C* :: *'mem update*
assumes *y*: *y* ∈ *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*))
⟨*proof*⟩

lemma *orth-run-B-upto-ket*:
assumes *y*: *y* ∈ *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*⟩

lemma *orth-run-B-upto-flip*:
assumes *y*: *y* ∈ *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*⟩

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*⟩

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-d-lists*
shows $x \neq$ *empty-list* $\longleftrightarrow$ $(\exists\, i{<}d.\ x!i)$
⟨*proof*⟩

First we show the mere existence of such a form.

lemma *run-B-upto-sum*:
assumes $n{<}d$
shows $\exists\, v.$ *run-B-upto* $n = (\sum i{\in}$*has-bits-upto* $n.$ $v\ i \otimes_s$ *ket* (*list-to-l* i))
⟨*proof*⟩

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*⟩

How Ψs relate to *run-B*.

lemma *run-B-upto-split*:
assumes $n{\leq}d$
shows *run-B-upto* $n = (\sum i{\in}$*has-bits-upto* $n.$ Ψs (*list-to-l* i) (*run-B-upto* n) $\otimes_s$ *ket* (*list-to-l* i))
⟨*proof*⟩

lemma *run-B-split*:
run-B $= (\sum i{\in}$*len-d-lists*$.$ Ψs (*list-to-l* i) *run-B* $\otimes_s$ (*ket* (*list-to-l* i)))
⟨*proof*⟩

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* $\times$ *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 (*λi. 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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*) $\leq$ *1*
⟨*proof*⟩

lemma *norm-run-B-count*:
norm run-B-count $\leq$ *1*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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

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*. Ψs *i* (*run-B-count-upto n*) $\otimes_s$ *ket i*)
⟨*proof*⟩

lemma *run-B-count-split*:
run-B-count = ($\sum$ *i*<*d*+*1*. Ψs *i run-B-count* $\otimes_s$ *ket i*)
⟨*proof*⟩

lemma *run-B-count-projection*:
Proj-ket-set {..<*n*+*1*} $*_V$ (*run-B-count-upto n*) = (*run-B-count-upto n*)
⟨*proof*⟩

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-cblinfun* − *selfbutter* (*ket empty*)) $*_V$ *run-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 Φs **where**
$\Phi s\ n$ = *run-pure-adv n* (λi. *UA i*) (λ-. *not-S-embed S*) *init X Y H*

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*-run-B-count-upto-eq-*Φs:
assumes *i*<*d*+*1*
shows Ψs *0* (*run-B-count-upto i*) = Φs *i*
⟨*proof*⟩

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

lemma Ψs*-run-B-upto-eq-*Φs:
assumes *i*≤*d*
shows Ψs *empty* (*run-B-upto i*) = Φs *i*
⟨*proof*⟩

For the version of o2h with *norm UA* ≤ *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 $|\ 0\rangle$ and then equivalence of B_{count} and B in $|\ 0\rangle$.

lemma *pure-o2h*: ‹(*norm* ((*run-A* $\otimes_s$ *ket empty*) − *run-B*))^2 ≤ (*d*+*1*) ∗ *Pfind'* + *d* ∗ *P-nonterm*›
⟨*proof*⟩

lemma *pure-o2h-sqrt*: ‹*norm* ((*run-A* $\otimes_s$ *ket empty*) − *run-B*) ≤ *sqrt* ((*d*+*1*) ∗ *Pfind'* + *d* ∗ *P-nonterm*)›
⟨*proof*⟩

lemma *error-term-pos*:
(*d*+*1*) ∗ *Pfind'* + *d* ∗ *P-nonterm* ≥ *0*
⟨*proof*⟩

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*⟩

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

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*: $\forall x{\in}M.\ v\ x \geq (0{::}real)$
and *pos-a*: $\forall x{\in}M.\ a\ x \geq (0{::}real)$
and *ineq*: $\forall x{\in}M.\ |\ sqrt\ (t\ x) - sqrt\ (u\ x)| \leq sqrt\ (v\ x)$
shows $|sqrt\ (\sum x{\in}M.\ a\ x * t\ x) - sqrt\ (\sum x{\in}M.\ a\ x * u\ x)| \leq sqrt\ (\sum x{\in}M.\ a\ x * v\ x)$
⟨*proof*⟩

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* ::
($'a{::}chilbert\text{-}space$, $'b{::}chilbert\text{-}space$, *unit*) *kraus-family* ⇒ ($'a$, $'b$, *unit*) *kraus-family set* **is**
$\lambda E.\ (\lambda x.\ \{x\})\ `\ E$
⟨*proof*⟩

lemma *kf-elems-Rep-kraus-family*:
kf-elems $\mathfrak{E}$ = $(\lambda x.\ Abs\text{-}kraus\text{-}family\ \{x\})\ `\ Rep\text{-}kraus\text{-}family\ \mathfrak{E}$
⟨*proof*⟩

lemma *kf-elems-finite*:
assumes $F \in kf\text{-}elems\ \mathfrak{E}$
shows *finite* (*Rep-kraus-family F*)
⟨*proof*⟩

lemma *kf-bound-of-elems*:
assumes $F \in kf\text{-}elems\ E$
shows $kf\text{-}bound\ F \leq kf\text{-}bound\ E$
⟨*proof*⟩

lemma *kf-elems-card-1*:
assumes $F \in kf\text{-}elems\ E$
shows $card\ (Rep\text{-}kraus\text{-}family\ F) = 1$

⟨*proof*⟩

lemma *inj-on-kf-singleton*:
inj-on (λ*x*. *Abs-kraus-family* {*x*}) (*Rep-kraus-family* 𝔈)
⟨*proof*⟩

lemma *kf-apply-singleton*:
fixes *E* :: ‹'*a*::*chilbert-space* ⇒$_{CL}$ '*b*::*chilbert-space* × '*x*›
assumes ‹*fst E* ≠ *0*›
shows *kf-apply* (*Abs-kraus-family* {*E*}) ϱ = *sandwich-tc* (*fst E*) ϱ
⟨*proof*⟩

lemma *kf-apply-summable-on-kf-elems*:
fixes 𝔈 :: ('*a*::*chilbert-space*,'*b*::*chilbert-space*,*unit*) *kraus-family*
shows (λ𝔉. *kf-apply* 𝔉 ϱ) *summable-on* (*kf-elems* 𝔈)
⟨*proof*⟩

lemma *kf-apply-has-sum-kf-elems*:
fixes 𝔈 :: ('*a*::*chilbert-space*,'*b*::*chilbert-space*,*unit*) *kraus-family*
shows ((λ𝔉. *kf-apply* 𝔉 ϱ) *has-sum* (*kf-apply* 𝔈 ϱ)) (*kf-elems* 𝔈)
⟨*proof*⟩

lemma *kf-apply-abs-summable-on-kf-elems*:
fixes 𝔈 :: ('*a*::*chilbert-space*,'*b*::*chilbert-space*,*unit*) *kraus-family*
shows (λ𝔉. *kf-apply* 𝔉 ϱ) *abs-summable-on* (*kf-elems* 𝔈)
⟨*proof*⟩

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 𝔈 *n* = *PiE* {*0*..<*n*+*1*} (λ*i*. *kf-elems* (𝔈 *i*))

lemma *finite-kraus-subadv-I*:
assumes *f* ∈ *finite-kraus-subadv* 𝔈 *n* *i*<*n*+*1*
shows *f i* ∈ *kf-elems* (𝔈 *i*)
⟨*proof*⟩

lemma *finite-kraus-subadv-rewrite*:
finite-kraus-subadv 𝔈 (*Suc n*) =
(λ(*x*,*f*). *fun-upd f* (*Suc n*) *x*) ‘ (*kf-elems* (𝔈 (*Suc n*)) × *finite-kraus-subadv* 𝔈 *n*)
⟨*proof*⟩

lemma *finite-kraus-subadv-rewrite-inj*:
inj-on (λ(*x*, *f*). *f*(*Suc n* := *x*)) (*kf-elems* (𝔈 (*Suc n*)) × *finite-kraus-subadv* 𝔈 *n*)
⟨*proof*⟩

lemma *norm-kf-apply-singleton-trace-tc*:
assumes $0 \leq \varrho$ **and** ‹*fst x* $\neq$ *0*›
shows *norm* (*kf-apply* (*Abs-kraus-family* {*x*}) ϱ) = *trace-tc* (*sandwich-tc* (*fst x*) ϱ)
⟨*proof*⟩

lemma *infsum-norm-kf-apply-step*:
assumes *ϱn-summable*: *ϱn summable-on finite-kraus-subadv* $\mathfrak{E}$ *n*
and *pos*: $\bigwedge x.\ x \in$ *finite-kraus-subadv* $\mathfrak{E}$ *n* $\Longrightarrow$ *0* $\leq$ *ϱn x*
shows ($\lambda x.\ \sum_\infty y \in$ *finite-kraus-subadv* $\mathfrak{E}$ *n. norm* (*kf-apply x* (*ϱn y*)))
abs-summable-on kf-elems ($\mathfrak{E}$ (*Suc n*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *run-mixed-adv-abs-summable*:
fixes $\mathfrak{E}$ :: *'a kraus-adv*
shows ($\lambda f.$ *run-mixed-adv n f UB init X Y H*) *abs-summable-on* (*finite-kraus-subadv* $\mathfrak{E}$ *n*)
⟨*proof*⟩

lemma *run-mixed-adv-summable*:
fixes $\mathfrak{E}$ :: *'a kraus-adv*
shows ($\lambda f.$ *run-mixed-adv n f UB init X Y H*) *summable-on* (*finite-kraus-subadv* $\mathfrak{E}$ *n*)
⟨*proof*⟩

lemma *run-mixed-adv-has-sum*:
fixes $\mathfrak{E}$ :: *'a kraus-adv*
shows (($\lambda f.$ *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*⟩

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

context *o2h-setting*
begin

lemma *run-mixed-A-has-sum*:
(($\lambda f.$ *run-mixed-A f H*) *has-sum run-mixed-A kraus-A H*) (*finite-kraus-subadv kraus-A d*)
⟨*proof*⟩

lemma *run-mixed-B-has-sum*:
((λf. *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* (λn. *kf-Fst* (*kraus-B n*)) *d*)
⟨*proof*⟩

lemma *run-mixed-B-count-has-sum*:
((λf. *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* (λn. *kf-Fst* (*kraus-B n*)) *d*)
⟨*proof*⟩

lemma *kf-elems-kf-Fst*:
kf-elems (*kf-Fst* $\mathfrak{E}$) = (λf. *kf-Fst f*) ‘ *kf-elems* $\mathfrak{E}$
⟨*proof*⟩

lemma *finite-kraus-subadv-Fst-invert*:
finite-kraus-subadv (λm. (*kf-Fst* :: -⇒(('*a* × '*c*) *ell2*,-,-) *kraus-family*) ($\mathfrak{E}$ *m*)) *n* =
(λf. $\lambda i{\in}\{0..{<}n{+}1\}$. *kf-Fst* (*f i*)) ‘ (*finite-kraus-subadv* $\mathfrak{E}$ *n*)
⟨*proof*⟩

lemma *inj-kf-Fst*: ‹*E* = *F*› **if** ‹*kf-Fst E* = *kf-Fst F*›
⟨*proof*⟩

lemma *inj-on-kf-Fst*:
inj-on (λf. $\lambda n{\in}\{0..{<}n{+}1\}$. (*kf-Fst* (*f n*) :: (('*a* × '*b*) *ell2*, -, -) *kraus-family*))
(*finite-kraus-subadv* $\mathfrak{E}$ *n*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *run-mixed-B-count-has-sum'*:

((λ*f*. *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*⟩

Limit with finite sums

lemma *has-sum-finite-sum*:
fixes *f* :: *'a* ⇒ *'b* ⇒ *'c*:: {*comm-monoid-add*,*topological-space*, *topological-comm-monoid-add*}
assumes ⋀*val*. (*f val has-sum g val*) *A finite S*
shows ((λ*x*. (∑ *val* ∈ *S*. *f val x*)) *has-sum* (∑ *val* ∈ *S*. *g val*)) *A*
⟨*proof*⟩

lemma *fin-subadv-fin-Rep-kraus-family*:
assumes *F* ∈ *finite-kraus-subadv E n i* < *n*+*1 n*<*d*+*1*
shows *finite* (*Rep-kraus-family* (*F i*))
⟨*proof*⟩

lemma *fin-subadv-bound-leq-id*:
assumes *F* ∈ *finite-kraus-subadv E d*
assumes *i* < *d*+*1*
assumes *E-norm-id*: ⋀*i*. *i* < *d*+*1* ⟹ *kf-bound* (*E i*) ≤ *id-cblinfun*
shows *kf-bound* (*F i*) ≤ *id-cblinfun*
⟨*proof*⟩

lemma *fin-subadv-nonzero*:
assumes *F* ∈ *finite-kraus-subadv E n i* < *n*+*1 n*<*d*+*1*
shows *Rep-kraus-family* (*F i*) ≠ {}
⟨*proof*⟩

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 ⇒ (*nat* ⇒ ('*a*::*chilbert-space*, '*b*::*chilbert-space*, '*c*) *kraus-family*) ⇒ (*nat* ⇒ '*a* ⇒$_{CL}$ '*b*) *set* **where**
  purify-comp-kraus n 𝔈 = *PiE* {*0*..<*n*+*1*} (λ*i*. (*fst* ‘ (*Rep-kraus-family* (𝔈 *i*))))

definition *comp-upto* :: (*nat* ⇒ ('*a*::*chilbert-space*) ⇒$_{CL}$ '*a*) ⇒ *nat* ⇒ '*a* ⇒$_{CL}$ '*a* **where**
  comp-upto f n = *fold* (λ*i x*. *f i* ∘$_{CL}$ *x*) [*0*..<*n*+*1*] *id-cblinfun*

Some auxiliary lemmas on injectivity, Fst and finiteness.

lemma *Rep-kf-id*:
  Rep-kraus-family kf-id = {(*id-cblinfun* :: '*a* ⇒$_{CL}$ '*a*::{*chilbert-space*,*not-singleton*},())}
  ⟨*proof*⟩

lemma *fst-Rep-kf-Fst*:
  fixes 𝔈 :: ('*a ell2*, '*b ell2*, *unit*) *kraus-family*
  shows *fst* ‘ (*Rep-kraus-family* (*kf-Fst* 𝔈)) = *Fst* ‘ (*fst* ‘ (*Rep-kraus-family* 𝔈))
⟨*proof*⟩

lemma *inj-on-Fst*:
  shows *inj-on Fst A*
  ⟨*proof*⟩

lemma *finite-kf-Fst*:
  fixes 𝔈 :: ('*mem ell2*, '*mem ell2*, *unit*) *kraus-family*
  assumes *finite* (*Rep-kraus-family* 𝔈)
  shows *finite* (*Rep-kraus-family* (*kf-Fst* 𝔈))
  ⟨*proof*⟩

lemma *finite-kf-id*:
  finite (*Rep-kraus-family kf-id*)
  ⟨*proof*⟩

lemma *inj-on-fst-Rep-kraus-family*:
  fixes 𝔈 :: ('*a ell2*,'*b ell2*,*unit*) *kraus-family*
  shows *inj-on fst* (*Rep-kraus-family* 𝔈)
  ⟨*proof*⟩

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* $\mathfrak{E}$)
⟨*proof*⟩

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* ∈ *fst* ‘ *Rep-kraus-family* $\mathfrak{E}$
shows *U*∗ o_{CL} *U* ≤ *id-cblinfun*
⟨*proof*⟩

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* ∈ *fst* ‘ *Rep-kraus-family* $\mathfrak{E}$
shows *norm U* ≤ *1*
⟨*proof*⟩

lemma *purify-comp-kraus-in-kraus-family*:
assumes *UA* ∈ *purify-comp-kraus n* $\mathfrak{E}$ *j*<*n*+*1*
shows *UA j* ∈ *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *j*)
⟨*proof*⟩

lemma *norm-in-purify-comp-kraus*:
fixes $\mathfrak{E}$ :: *nat* ⇒ ('*a ell2*, '*a ell2*, *unit*) *kraus-family*
assumes $\bigwedge i.\ i<n+1 \Longrightarrow$ *kf-bound* ($\mathfrak{E}$ *i*) ≤ *id-cblinfun*
assumes *UA* ∈ *purify-comp-kraus n* $\mathfrak{E}$
shows $\bigwedge i.\ i<n+1 \Longrightarrow$ *norm* (*UA i*) ≤ *1*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

Purifications of the adversarial runs.

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

shows *run-mixed-adv n* $\mathfrak{E}$ *UB init' X' Y' H* =
(∑ *UAs* ∈ *purify-comp-kraus n* $\mathfrak{E}$. *run-pure-adv-tc n UAs UB init' X' Y' H*)
⟨*proof*⟩

lemma *purification-run-mixed-A*:
assumes ⋀*i*. *i*<*d*+1 ⟹ *finite* (*Rep-kraus-family* ($\mathfrak{E}$ *i*))
assumes ⋀*i*. *i*<*d*+1 ⟹ *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*) ≠ {}
shows *run-mixed-A* $\mathfrak{E}$ *H* = (∑ *UAs* ∈ *purify-comp-kraus d* $\mathfrak{E}$. *run-pure-A-tc UAs H*)
⟨*proof*⟩

lemma *purification-run-mixed-B*:
assumes ⋀*i*. *i*<*d*+1 ⟹ *finite* (*Rep-kraus-family* ($\mathfrak{E}$ *i*))
assumes ⋀*i*. *i*<*d*+1 ⟹ *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*) ≠ {}
shows *run-mixed-B* $\mathfrak{E}$ *H S* = (∑ *UAs* ∈ *purify-comp-kraus d* $\mathfrak{E}$. *run-pure-B-tc UAs H S*)
⟨*proof*⟩

lemma *purification-run-mixed-B-count-prep*:
assumes ⋀*i*. *i*<*d*+1 ⟹ *finite* (*Rep-kraus-family* ($\mathfrak{E}$ *i*))
assumes ⋀*i*. *i*<*d*+1 ⟹ *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*) ≠ {}
assumes *n*<*d*+1
shows *run-mixed-adv n* (λ*n*. *kf-Fst* ($\mathfrak{E}$ *n*)) (λ*n*. *U-S' S*)
init-B-count X-for-C Y-for-C H =
(∑ *UAs*∈(Π$_E$ *i*∈{*0*..<*n* + *1*}. *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*)).
run-pure-adv-tc n (*Fst* ∘ *UAs*) (λ-. *U-S' S*) *init-B-count X-for-C*
Y-for-C H)
⟨*proof*⟩

lemma *purification-run-mixed-B-count*:
assumes ⋀*i*. *i*<*d*+1 ⟹ *finite* (*Rep-kraus-family* ($\mathfrak{E}$ *i*))
assumes ⋀*i*. *i*<*d*+1 ⟹ *fst* ‘ *Rep-kraus-family* ($\mathfrak{E}$ *i*) ≠ {}
shows *run-mixed-B-count* $\mathfrak{E}$ *H S* = (∑ *UAs* ∈ *purify-comp-kraus d* $\mathfrak{E}$. *run-pure-B-count-tc UAs H S*)
⟨*proof*⟩

Purification of *kf-Fst*

lemma *purification-kf-Fst*:
assumes ⋀*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* ⇒$_{CL}$ ('*b* × '*c*) *ell2*) *else undefined*)
⟨*proof*⟩

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 ⇒ 'y)×('x ⇒ *bool*)×-) *set*
fixes *distr* :: (('x ⇒ 'y)×('x ⇒ *bool*)×-) ⇒ *real*

assumes *distr-pos*: ∀(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ≥ *0*
and *distr-sum-1*: (∑(*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*)) = *1*
and *finite-carrier*: *finite carrier*

fixes *E*:: 'mem *kraus-adv*
assumes *E-norm-id*: ⋀*i*. *i* < *d+1* ⟹ *kf-bound* (*E i*) ≤ *id-cblinfun*
assumes *E-nonzero*: ⋀*i*. *i* < *d+1* ⟹ *Rep-kraus-family* (*E i*) ≠ {}

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

begin

lemma *norm-P*:
norm P ≤ *1*
⟨*proof*⟩

lemma *distr-pos'*:
assumes $(H,S,z) \in carrier$ **shows** $distr\ (H,S,z) \geq 0$
⟨*proof*⟩

lemma *norm-Fst-P*:
$norm\ (Fst\ P :: ('mem \times 'a)\ update) \leq 1$
⟨*proof*⟩

9.1 Final states

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

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

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

definition *ϱright* :: *'mem kraus-adv* ⇒ $('mem \times 'l)\ tc\text{-}op$ **where**
$\varrho right\ F = (\sum (H,S,z) \in carrier.\ distr\ (H,S,z) *_C run\text{-}mixed\text{-}B\ F\ H\ S)$

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

definition *ϱcount* :: *'mem kraus-adv* ⇒ $('mem \times nat)\ tc\text{-}op$ **where**
$\varrho count\ F = (\sum (H,S,z) \in carrier.\ distr\ (H,S,z) *_C run\text{-}mixed\text{-}B\text{-}count\ F\ H\ S)$

Positivity

lemma *ϱleft-pure-pos*: $0 \leq \varrho left\text{-}pure\ UA$
⟨*proof*⟩

lemma *ϱleft-pos*: $0 \leq \varrho left\ F$
⟨*proof*⟩

lemma *ϱright-pure-pos*: $0 \leq \varrho right\text{-}pure\ UA$
⟨*proof*⟩

lemma *ϱright-pos*: $0 \leq \varrho right\ F$
⟨*proof*⟩

lemma *ϱcount-pure-pos*: $0 \leq \varrho count\text{-}pure\ UA$
⟨*proof*⟩

lemma *ϱcount-pos*: $0 \leq \varrho count\ F$
⟨*proof*⟩

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

lemma *norm-ϱleft*:
norm (*ϱleft E*) ≤ *1*
⟨*proof*⟩

lemma *norm-ϱright*:
norm (*ϱright E*) ≤ *1*
⟨*proof*⟩

lemma *norm-ϱcount*:
norm (*ϱcount E*) ≤ *1*
⟨*proof*⟩

lemma *trace-preserving-norm-ϱright*:
assumes ⋀*i*. *i* < *d+1* ⟹ *km-trace-preserving* (*kf-apply*
(*kf-Fst* (*E i*)::(('*mem* × '*l*) *ell2*, ('*mem* × '*l*) *ell2*, *unit*) *kraus-family*))
shows *norm* (*ϱright E*) = *1*
⟨*proof*⟩

lemma *trace-preserving-norm-ϱcount*:
assumes ⋀*i*. *i* < *d+1* ⟹ *km-trace-preserving* (*kf-apply*
(*kf-Fst* (*E i*)::(('*mem* × *nat*) *ell2*, ('*mem* × *nat*) *ell2*, *unit*) *kraus-family*))
shows *norm* (*ϱcount E*) = *1*
⟨*proof*⟩

Summability and Infsums

lemma *from-trace-class-ϱright-pure*:
from-trace-class (*ϱright-pure UA*) = (∑ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗$_C$ *run-pure-B-update*
UA H S)
⟨*proof*⟩

lemma *has-sum-scaleC-tc*:
fixes *x* :: ('*a*::*chilbert-space*,'*a*) *trace-class*
assumes (*f has-sum x*) *A*
shows ((λ*y*. *c* ∗$_C$ *f y*) *has-sum c* ∗$_C$ *x*) *A*
⟨*proof*⟩

lemma *ϱleft-has-sum*:
(*ϱleft has-sum ϱleft E*) (*finite-kraus-subadv E d*)
⟨*proof*⟩

lemma *ϱright-has-sum*:
((λ*f*. *ϱright f*) *has-sum ϱright E*) (*finite-kraus-subadv E d*)

⟨*proof*⟩

lemma *ϱright-abs-summable*:
ϱright abs-summable-on (*finite-kraus-subadv E d*)
⟨*proof*⟩

lemma *ϱcount-has-sum*:
(*ϱcount has-sum ϱcount E*) (*finite-kraus-subadv E d*)
⟨*proof*⟩

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* ∈ *purify-comp-kraus d F*. *ϱleft-pure UAs*)
⟨*proof*⟩

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* ∈ *purify-comp-kraus d F*. *ϱright-pure UAs*)
⟨*proof*⟩

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* ∈ *purify-comp-kraus d F*. *ϱcount-pure UAs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *is-Proj-end-measure*:
is-Proj end-measure
⟨*proof*⟩

lemma *Proj-end-measure*:
Proj (*end-measure* $*_S$ $\top$) = *end-measure*
⟨*proof*⟩

lemma *norm-end-measure*:
norm (*end-measure*) $\leq$ *1*
⟨*proof*⟩

lemma *end-measure-butterfly*:
sandwich end-measure (*selfbutter* Ψ) = *selfbutter* (*end-measure* $*_V$ Ψ)
⟨*proof*⟩

lemma *trace-end-measure*:
trace (*end-measure* o_{CL} *selfbutter* Ψ) = (*complex-of-real* (*norm* (*end-measure* $*_V$ Ψ)))^2
⟨*proof*⟩

lemma *trace-endmeasure-pos*:
assumes $\varrho \geq 0$
shows *trace-tc* (*compose-tcr end-measure* ϱ) $\geq$ *0*
⟨*proof*⟩

lemma *trace-class-end-measure*:
assumes *trace-class a*
shows *trace-class* (*end-measure* o_{CL} *a*)
⟨*proof*⟩

lemma *abs-op-id-cblinfun* [*simp*]:
abs-op id-cblinfun = *id-cblinfun*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *empty-tc-pos*: *0* $\leq$ *empty-tc*
⟨*proof*⟩

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 $(\varrho + \psi)$ = *PM-update* Q ϱ + *PM-update* Q ψ
⟨*proof*⟩

definition *PM* :: $('mem \times 'l)$ *update* ⇒ $('mem \times 'l)$ *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)
⟨*proof*⟩

lemma *PM-linear*:
PM Q $(\varrho + \psi)$ = *PM* Q ϱ + *PM* Q ψ
⟨*proof*⟩

lemma *PM-sum-distr*:
PM Q (*sum* f S) = *sum* (*PM* Q *o* f) S
⟨*proof*⟩

lemma *PM-scale*:
PM Q $(a *_C \varrho)$ = a * *PM* Q ϱ
⟨*proof*⟩

lemma *PM-case*:
PM Q (*case* x *of* $(H,S,z) \Rightarrow f\ H\ S$) = (*case* x *of* $(H,S,z) \Rightarrow$ *PM* Q $(f\ H\ S)$)
⟨*proof*⟩

lemma *PM-Re*:
assumes $\varrho \geq 0$
shows *Re* (*PM* Q ϱ) = *PM* Q ϱ
⟨*proof*⟩

lemma *PM-pos*:
assumes $\varrho \geq 0$
shows *PM* Q $\varrho \geq 0$
⟨*proof*⟩

lemma *Re-PM-pos*:
assumes $\varrho \geq 0$
shows *Re* (*PM* Q ϱ) ≥ 0
⟨*proof*⟩

lemma *norm-PM*:
assumes *norm* $\varrho \leq 1$ *norm* $Q \leq 1$
shows *norm* (*PM* Q ϱ) ≤ 1
⟨*proof*⟩

lemma *PM-bounded-linear*:
shows *bounded-linear* (*PM Q*)
⟨*proof*⟩

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*
⟨*proof*⟩

10.2 Pright and Pleft'

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

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

lemma *trace-tensor-tc*:
trace-tc (*tc-tensor a b*) = *trace-tc a* ∗ *trace-tc b*
⟨*proof*⟩

lemma *Pleft-Pleft'*:
assumes *sandwich-tc A empty-tc* = *tc-selfbutter* (*ket empty*)
shows *Pleft Q* = *Pleft'* (*Q* ⊗*o A*)
⟨*proof*⟩

lemma *Pleft-Pleft'-empty*:
Pleft Q = *Pleft'* (*Q* ⊗*o selfbutter* (*ket empty*))
⟨*proof*⟩

lemma *Pleft-Pleft'-id*:
Pleft Q = *Pleft'* (*Q* ⊗*o id-cblinfun*)
⟨*proof*⟩

lemma *Pleft-Pleft'-case5*:
assumes *is-Proj Q*
shows *Pleft Q* = *Pleft'* (*Q* ⊗*o selfbutter* (*ket empty*) + *end-measure*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Re-PM-right-has-sum*:
((λF. *Re* (*PM Q* (*ϱright F*))) *has-sum Pright Q*) (*finite-kraus-subadv E d*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Pfind-Pright*:
Re (*Pfind E*) = *Pright end-measure*
⟨*proof*⟩

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

lemma *Re-Pfind-update-altdef*:
assumes ⋀*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*⟩

lemma *Pfind-pure-update*:
Pfind-pure UA = (∑ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *Pfind-update UA H S*)
⟨*proof*⟩

lemma *Pfind-pure-mixed*:
assumes ⋀*i*. *i* < *d* + *1* ⟹ *finite* (*Rep-kraus-family* (*F i*))
⋀*i*. *i* < *d* + *1* ⟹ *fst* ‘ *Rep-kraus-family* (*F i*) ≠ {}
shows *Pfind F* = (∑ *UA*∈*purify-comp-kraus d F*. *Pfind-pure UA*)
⟨*proof*⟩

Pfind positivity

lemma *Pfind-pure-pos*:
Pfind-pure UA ≥ *0*
⟨*proof*⟩

lemma *Pfind-pos*:
Pfind F $\geq$ *0* ⟨*proof*⟩

Pfind is already real

lemma *Re-Pfind-update*:
Re (*Pfind-update UA H S*) = *Pfind-update UA H S*
⟨*proof*⟩

lemma *Re-Pfind-pure*:
Re (*Pfind-pure UA*) = *Pfind-pure UA*
⟨*proof*⟩

lemma *Re-Pfind*:
Re (*Pfind F*) = *Pfind F*
⟨*proof*⟩

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

lemma *Pfind-abs-summable-on*:
Pfind abs-summable-on (*finite-kraus-subadv E d*)
⟨*proof*⟩

lemma *Pfind-summable-on*:
Pfind summable-on (*finite-kraus-subadv E d*)
⟨*proof*⟩

lemma *Pfind-has-sum*:
((λ*F*. *Pfind F*) *has-sum Pfind E*) (*finite-kraus-subadv E d*)
⟨*proof*⟩

10.4 Nontermination Part

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

definition *P-nonterm-update*::('*x* ⇒ '*y*) ⇒ ('*x* ⇒ *bool*) ⇒ (*nat* ⇒ '*mem update*) ⇒ *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* ⇒ '*mem ell2* ⇒$_{CL}$ '*mem ell2*) ⇒ *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* ⇒ *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 ⋀*i*. *i*<*d*+*1* ⟹ *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*⟩

lemma *P-nonterm-pure-update*:
P-nonterm-pure UA = (∑ (*H*,*S*,*z*)∈*carrier*. *distr* (*H*,*S*,*z*) ∗ *P-nonterm-update H S UA*)
⟨*proof*⟩

lemma *P-nonterm-purification*:
assumes ⋀*i*. *i* < *d* + *1* ⟹ *finite* (*Rep-kraus-family* (*F i*))
⋀*i*. *i* < *d* + *1* ⟹ *Rep-kraus-family* (*F i*) ≠ {}
shows *P-nonterm F* = (∑ *UA*∈*purify-comp-kraus d F*. *P-nonterm-pure UA*)
⟨*proof*⟩

Positive error term

lemma *error-term-update-pos*:
assumes ⋀*i*. *i*<*d*+*1* ⟹ *norm* (*UA i*) ≤ *1*
shows *0* ≤ (*d*+*1*) ∗ *Re* (*Pfind-update UA H S*) + *d* ∗ (*P-nonterm-update H S UA*)
⟨*proof*⟩

lemma *error-term-pure-pos*:
assumes ⋀*i*. *i*<*d*+*1* ⟹ *norm* (*UA i*) ≤ *1*
shows *0* ≤ (*d*+*1*) ∗ *Re* (*Pfind-pure UA*) + *d* ∗ (*P-nonterm-pure UA*)
⟨*proof*⟩

lemma *error-term-pos*:
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*) ≠ {}
shows *0* ≤ (*d* + *1*) ∗ *Re* (*Pfind F*) + *d* ∗ *P-nonterm F*
⟨*proof*⟩

has sum property

lemma *P-nonterm-has-sum*:
((λ*F*. *P-nonterm F*) *has-sum P-nonterm E*) (*finite-kraus-subadv E d*)
⟨*proof*⟩

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 ⋀*i*. *i*<*d*+*1* ⟹ *norm* (*UA i*) ≤ *1*
and *norm-Q*: *norm Q* ≤ *1*
shows |*sqrt* (*Re* (*PM-update Q* ((*run-pure-A-update UA H*) ⊗*o* (*selfbutter* (*ket empty*))))) −

$sqrt\ (Re\ (PM\text{-}update\ Q\ (run\text{-}pure\text{-}B\text{-}update\ UA\ H\ S)))|$
$\leq sqrt\ ((d{+}1) * Re\ (Pfind\text{-}update\ UA\ H\ S) + d * P\text{-}nonterm\text{-}update\ H\ S\ UA)$
⟨*proof*⟩

lemma *estimate-Pfind-tc-sqrt*:
fixes *UA H S*
assumes $\bigwedge i.\ i{<}d{+}1 \Longrightarrow norm\ (UA\ i) \leq 1\ norm\ Q \leq 1$
shows $|sqrt\ (Re\ (PM\ Q\ (tc\text{-}tensor\ (run\text{-}pure\text{-}A\text{-}tc\ UA\ H)\ empty\text{-}tc))) -$
$sqrt\ (Re\ (PM\ Q\ (run\text{-}pure\text{-}B\text{-}tc\ UA\ H\ S)))|$
$\leq sqrt\ ((d{+}1) * Re\ (Pfind\text{-}update\ UA\ H\ S) + d * (P\text{-}nonterm\text{-}update\ H\ S\ UA))$
⟨*proof*⟩

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 $\bigwedge i.\ i{<}d{+}1 \Longrightarrow norm\ (UA\ i) \leq 1\ norm\ Q \leq 1$
shows $|sqrt\ (Re\ (PM\ Q\ (tc\text{-}tensor\ (\varrho left\text{-}pure\ UA)\ empty\text{-}tc))) - sqrt\ (Re\ (PM\ Q\ (\varrho right\text{-}pure\ UA)))|$
$\leq sqrt\ (real\ (d\ +\ 1) * Re\ (Pfind\text{-}pure\ UA) + d * P\text{-}nonterm\text{-}pure\ UA)$
⟨*proof*⟩

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

lemma *estimate-Pfind-finite-sqrt*:
assumes *finite*: $\bigwedge i.\ i < d{+}1 \Longrightarrow finite\ (Rep\text{-}kraus\text{-}family\ (F\ i))$
and *F-norm-id*:$\bigwedge i.\ i < d{+}1 \Longrightarrow kf\text{-}bound\ (F\ i) \leq id\text{-}cblinfun$
and *F-nonzero*: $\bigwedge i.\ i < d{+}1 \Longrightarrow Rep\text{-}kraus\text{-}family\ (F\ i) \neq \{\}$
and *norm-Q*: $norm\ Q \leq 1$
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*⟩

lemma *estimate-Pfind-finite-sqrt'*:
assumes *finite*: $\bigwedge i.\ i < d{+}1 \Longrightarrow finite\ (Rep\text{-}kraus\text{-}family\ (F\ i))$
and *F-norm-id*:$\bigwedge i.\ i < d{+}1 \Longrightarrow kf\text{-}bound\ (F\ i) \leq id\text{-}cblinfun$
and *F-nonzero*: $\bigwedge i.\ i < d{+}1 \Longrightarrow Rep\text{-}kraus\text{-}family\ (F\ i) \neq \{\}$
and *norm-Q*: $norm\ Q \leq 1$
shows $|sqrt\ (Re\ (PM\ Q\ (tc\text{-}tensor\ (\varrho left\ F)\ empty\text{-}tc))) - sqrt\ (Re\ (PM\ Q\ (\varrho right\ F)))| \leq$
$sqrt\ ((d{+}1) * Re\ (Pfind\ F) + d * P\text{-}nonterm\ F)$
⟨*proof*⟩

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*:
$((\lambda F.\ (1\ +\ real\ d) * Re\ (Pfind\ F))\ has\text{-}sum\ (1\ +\ real\ d) * Re\ (Pfind\ E))\ (finite\text{-}kraus\text{-}subadv\ E\ d)$

⟨*proof*⟩

lemma *scale-P-nonterm-has-sum*:
((λ*F*. *real d* ∗ *P-nonterm F*) *has-sum real d* ∗ *P-nonterm E*) (*finite-kraus-subadv E d*)
⟨*proof*⟩

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*⟩

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*⟩

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-add*) *TYPE*('*mem*) *TYPE*('*l*) +
fixes *carrier* :: (('*x* ⇒ '*y*)×('*x* ⇒ '*y*)×('*x* ⇒ *bool*)×-) *set*
fixes *distr* :: (('*x* ⇒ '*y*)×('*x* ⇒ '*y*)×('*x* ⇒ *bool*)×-) ⇒ *real*

assumes *distr-pos*: ∀(*H*,*G*,*S*,*z*)∈*carrier*. *distr* (*H*,*G*,*S*,*z*) ≥ *0*
and *distr-sum-1*: (∑(*H*,*G*,*S*,*z*)∈*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*:
$\bigwedge i.\ i < d+1 \Longrightarrow Rep\text{-}kraus\text{-}family\ (kf\text{-}Fst\ (E\ i)) \neq \{\}$
⟨*proof*⟩

Some properties of the joint distribution.

lemma *Uquery-G-H-same-on-not-S-embed'*:
assumes $(H,G,S,z) \in carrier$
shows
$Uquery\ H\ o_{CL}\ proj\text{-}classical\text{-}set\ (-\ (Collect\ S)) \otimes_o id\text{-}cblinfun =$
$Uquery\ G\ o_{CL}\ proj\text{-}classical\text{-}set\ (-\ (Collect\ S)) \otimes_o id\text{-}cblinfun$
⟨*proof*⟩

lemma *Uquery-G-H-same-on-not-S-embed*:
assumes $(H,G,S,z) \in carrier$
shows $((X;Y)\ (Uquery\ H)\ o_{CL}\ (not\text{-}S\text{-}embed\ S)) = ((X;Y)\ (Uquery\ G)\ o_{CL}\ (not\text{-}S\text{-}embed\ S))$
⟨*proof*⟩

lemma *Uquery-G-H-same-on-not-S-embed-tensor*:
assumes $(H,G,S,z) \in carrier$
shows $((X\text{-}for\text{-}B;Y\text{-}for\text{-}B)\ (Uquery\ H)\ o_{CL}\ Fst\ (not\text{-}S\text{-}embed\ S)) =$
$((X\text{-}for\text{-}B;Y\text{-}for\text{-}B)\ (Uquery\ G)\ o_{CL}\ Fst\ (not\text{-}S\text{-}embed\ S))$
⟨*proof*⟩

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*: $\forall (G,S,z) \in carrier\text{-}G.\ distr\text{-}G\ (G,S,z) \geq 0$
⟨*proof*⟩

lemma *distr-G-sum-1*: $(\sum (G,S,z) \in carrier\text{-}G.\ distr\text{-}G\ (G,S,z)) = 1$
⟨*proof*⟩

lemma *finite-carrier-G*: *finite carrier-G*

⟨*proof*⟩

definition *carrier-H* **where** *carrier-H* = (λ(*H*,*G*,*S*,*z*). (*H*,*S*,(*G*,*z*))) ‘ *carrier*
definition *distr-H* **where** *distr-H* = (λ(*H*,*S*,(*G*,*z*)). *distr* (*H*,*G*,*S*,*z*))

lemma *distr-H-pos*: ∀ (*H*,*S*,*z*)∈*carrier-H*. *distr-H* (*H*,*S*,*z*) ≥ *0*
⟨*proof*⟩

lemma *distr-H-sum-1*: (∑ (*H*,*S*,*z*)∈*carrier-H*. *distr-H* (*H*,*S*,*z*)) = *1*
⟨*proof*⟩

lemma *finite-carrier-H*: *finite carrier-H*
⟨*proof*⟩

interpretation *mixed-H*: *mixed-o2h X Y d init flip bit valid empty carrier-H distr-H E P*
⟨*proof*⟩

interpretation *mixed-G*: *mixed-o2h X Y d init flip bit valid empty carrier-G distr-G E P*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *empty-not-flip*:
assumes *x* ∈ *list-to-l* ‘ *has-bits-upto n n*<*d*
shows *empty* ≠ *flip n x*
⟨*proof*⟩

lemma *empty-not-flip′*:
assumes *x* ≠ *flip n empty n*<*d*
shows *empty* ≠ *flip n x*
⟨*proof*⟩

lemma *Proj-ket-upto-Snd*:
Proj-ket-upto A = *Snd* (*proj-classical-set* (*list-to-l* ‘ *A*))
⟨*proof*⟩

lemma *from-trace-class-tc-selfbutter*:
from-trace-class (*tc-selfbutter x*) = *selfbutter x*
⟨*proof*⟩

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*⟩

lemma *list-to-l-has-bits-upto-flip*:
assumes $b \in$ *list-to-l* ‘ *has-bits-upto n* $n<d$
shows *flip n b* $\in$ *list-to-l* ‘ *has-bits-upto* (*Suc n*)
⟨*proof*⟩

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*⟩

lemma *run-pure-adv-projection*:
assumes $n<d+1$
and ϱ: ϱ = *run-pure-adv-tc n* (λ*m*. *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
⟨*proof*⟩

lemma *run-mixed-adv-projection-finite*:
assumes $\bigwedge i.\ i < n + 1 \Longrightarrow$ *finite* (*Rep-kraus-family* (*kf-Fst* (*F i*)::
((*'mem* × *'l*) *ell2*, (*'mem* × *'l*) *ell2*, *unit*) *kraus-family*))
and $\bigwedge i.\ i < n + 1 \Longrightarrow$ *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*⟩

lemma *run-mixed-adv-projection*:
assumes $\bigwedge i.\ i < d + 1 \Longrightarrow$ *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*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *run-mixed-B-G-H-same*:
assumes (*H*,*G*,*S*,*z*)∈*carrier*
shows *sandwich-tc* (*Q* ⊗$_o$ *selfbutter* (*ket empty*)) (*run-mixed-B E H S*) =
sandwich-tc (*Q* ⊗$_o$ *selfbutter* (*ket empty*)) (*run-mixed-B E G S*)
⟨*proof*⟩

lemma *ϱright-G-H-same*:
sandwich-tc (*Q* ⊗$_o$ *selfbutter* (*ket empty*)) (*mixed-H.ϱright E*) =
sandwich-tc (*Q* ⊗$_o$ *selfbutter* (*ket empty*)) (*mixed-G.ϱright E*)
⟨*proof*⟩

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*⟩

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

lemma *Pright-G-H-same*:
mixed-H.Pright (*Q* ⊗$_o$ *selfbutter* (*ket empty*)) = *mixed-G.Pright* (*Q* ⊗$_o$ *selfbutter* (*ket empty*))
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

Other formulations of the mixed o2h.

Theorem 1, definition of Pright (2)

definition *Proj-2* :: (*'mem* × *'l*) *ell2* ⇒$_{CL}$ (*'mem* × *'l*) *ell2* **where**
Proj-2 = *P* ⊗$_o$ *id-cblinfun*

lemma *norm-Proj-2*:
norm Proj-2 ≤ *1*
⟨*proof*⟩

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*⟩

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*⟩

Theorem 1, definition of Pright (3)

definition *Proj-3* :: (*'mem* × *'l*) *ell2* ⇒$_{CL}$ (*'mem* × *'l*) *ell2* **where**
Proj-3 = *P* ⊗$_o$ *selfbutter* (*ket empty*)

lemma *is-Proj-3*:
is-Proj Proj-3
⟨*proof*⟩

lemma *Proj-3-altdef*:
Proj-3 = *Proj* ((*P* ⊗$_o$ *id-cblinfun*) ∗$_S$ ⊤ ⊔ (*id-cblinfun* ⊗$_o$ *selfbutter* (*ket empty*)) ∗$_S$ ⊤)
⟨*proof*⟩

lemma *norm-Proj-3*:
norm Proj-3 ≤ *1*
⟨*proof*⟩

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*⟩

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*⟩

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-H.Pleft P*) − *sqrt* (*mixed-G.Pright Proj-3*)$|$ ≤
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*) + *d*∗ *mixed-H.P-nonterm E*)
⟨*proof*⟩

theorem *mixed-o2h-term-4*:
assumes ⋀*i*. *i*<*d*+*1* ⟹ *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*⟩

Theorem 1: the definition of Pright (5) is *Pright* = *P*[*find* ∨ *b*=*1 for b*<− *A*^{*H**S*}] = *P*(*find for b*<− *A*^{*H**S*}) + *P*(¬ *find* ∧ *b*=*1 for b*<− *A*^{*H**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* ∧ *B*) ≡ *PM-A o PM-B Pr*(*A* ∨ *B*) ≡ *PM-A* + *PM-B* − *PM-A o PM-B*

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

definition *Proj-5* :: (*'mem* × *'l*) *ell2* ⇒$_{CL}$ (*'mem* × *'l*) *ell2* **where**
Proj-5 = *Proj* (((*P* ⊗$_o$ *id-cblinfun*) ∗$_S$ ⊤) ⊔ ((*id-cblinfun* ⊗$_o$ (*id-cblinfun* − *selfbutter* (*ket empty*))) ∗$_S$ ⊤))

lemma *is-Proj-5*:
is-Proj Proj-5
⟨*proof*⟩

lemma *Proj-5-altdef*:
Proj-5 = *Proj-3* + *mixed-H.end-measure*
⟨*proof*⟩

lemma *norm-Proj-5*:
norm Proj-5 ≤ *1*
⟨*proof*⟩

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*⟩

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*$|$ ≤
2 ∗ *sqrt* ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
and
$|$*sqrt* (*mixed-H.Pleft P*) − *sqrt* (*mixed-H.Pright Proj-5*)$|$ ≤
sqrt ((*d*+*1*) ∗ *Re* (*mixed-H.Pfind E*))
⟨*proof*⟩

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*⟩

lemma *Pright-G-H-case5*:
assumes ⋀*i*. *i*<*d*+*1* ⟹ *km-trace-preserving* (*kf-apply* (*E i*))
shows *mixed-H.Pright Proj-5* = *mixed-G.Pright Proj-5*
⟨*proof*⟩

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*⟩

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*⟩

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.