

The Oneway to Hiding Theorem*

Katharina Heidler Dominique Unruh

October 13, 2025

Abstract

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

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

Contents

1	Definitions for the one-way to Hiding (O2H) Lemma	7
1.1	Locale for the general O2H setting	7
1.2	Linear operator US	19
1.3	Towards the Definition of $U-S'$	21
2	Running the Adversary	23
3	Definition of B-count	28
3.1	Defining the run of adversary B	28
3.2	Locale for the pure O2H setting	30

*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).

4	Defining and Representing the Adversary B	31
4.1	Representing the run of Adversary B as a finite sum	32
5	Defining and Representing the Adversary B with Counting	34
5.1	Representing the run of Adversary B with counting as a finite sum	34
6	Auxiliary lemma: Estimation	37
7	Limit Processes	38
8	Purification of the Adversary	43
9	Mixed O2H Setting and Preliminaries	46
9.1	Final states	47
9.2	Measurement at the end	49
10	<i>empty-tc</i> is the trace-class representative of the 0.	50
10.1	Projective measurement PM	50
10.2	Prigh and Pleft'	52
10.3	Pfind	53
10.4	Nontermination Part	54
11	Proof of Mixed O2H	55
12	General O2H Setting and Theorem	57
theory <i>O2H-Additional-Lemmas</i>		
imports <i>Registers.Pure-States</i>		
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 $\langle inj\text{-}on\ h\ A \rangle$
 shows $\langle g\ abs\text{-}summable\text{-}on\ (h\ \text{'}\ A) \longleftrightarrow (g \circ h)\ abs\text{-}summable\text{-}on\ A \rangle$
 $\langle proof \rangle$

lemma *abs-summable-norm*:
 assumes $\langle f\ abs\text{-}summable\text{-}on\ A \rangle$
 shows $\langle (\lambda x.\ norm\ (f\ x))\ abs\text{-}summable\text{-}on\ A \rangle$
 $\langle proof \rangle$

lemma *abs-summable-on-add*:
assumes $\langle f \text{ abs-summable-on } A \rangle$ **and** $\langle g \text{ abs-summable-on } A \rangle$
shows $\langle (\lambda x. f\ x + g\ x) \text{ abs-summable-on } A \rangle$
 $\langle \text{proof} \rangle$

lemma *sandwich-tc-has-sum*:
assumes $\langle f \text{ has-sum } x \rangle$ A
shows $\langle (\text{sandwich-tc } \varrho \circ f) \text{ has-sum sandwich-tc } \varrho\ x \rangle A$
 $\langle \text{proof} \rangle$

lemma *sandwich-tc-abs-summable-on*:
assumes $f \text{ abs-summable-on } A$
shows $(\text{sandwich-tc } \varrho \circ f) \text{ abs-summable-on } A$
 $\langle \text{proof} \rangle$

lemma *trace-tc-abs-summable-on*:
assumes $f \text{ abs-summable-on } A$
shows $(\text{trace-tc } \circ f) \text{ abs-summable-on } A$
 $\langle \text{proof} \rangle$

Defining a self butterfly on trace class.

lemma *trace-selfbutter-norm*:
 $\text{trace } (\text{selfbutter } A) = \text{norm } A \wedge^2$
 $\langle \text{proof} \rangle$

definition *tc-selfbutter* **where** $\text{tc-selfbutter } a = \text{tc-butterfly } a\ a$

lemma *norm-tc-selfbutter[simp]*:
 $\text{norm } (\text{tc-selfbutter } a) = (\text{norm } a) \wedge^2$
 $\langle \text{proof} \rangle$

lemma *trace-tc-sandwich-tc-isometry*:
assumes *isometry* U
shows $\text{trace-tc } (\text{sandwich-tc } U\ A) = \text{trace-tc } A$
 $\langle \text{proof} \rangle$

lemma *norm-sandwich-tc-unitary*:
assumes *isometry* U $\varrho \geq 0$
shows $\text{norm } (\text{sandwich-tc } U\ \varrho) = \text{norm } \varrho$
 $\langle \text{proof} \rangle$

Lemmas on *trace-tc*

lemma *trace-tc-minus*:
 $\text{trace-tc } (a - b) = \text{trace-tc } a - \text{trace-tc } b$
 $\langle \text{proof} \rangle$

lemma *trace-tc-sum*:

trace-tc (*sum a I*) = ($\sum i \in I. \text{trace-tc } (a \ i)$)
 ⟨*proof*⟩

lemma *selfbutter-sandwich*:

fixes *A* :: 'a ell2 $\Rightarrow_{CL}$ 'a ell2 **and** *B* :: 'a ell2
shows *selfbutter* (*A* *_V *B*) = *sandwich A* *_V *selfbutter B*
 ⟨*proof*⟩

lemma *tc-tensor-sum-left*:

tc-tensor (*sum g S*) *x* = ($\sum i \in S. \text{tc-tensor } (g \ i) \ x$)
 ⟨*proof*⟩

lemma *tc-tensor-sum-right*:

tc-tensor *x* (*sum g S*) = ($\sum i \in S. \text{tc-tensor } x \ (g \ i)$)
 ⟨*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 of lemmas on limit processes with several functions.

lemma *tendsto-Re*:

assumes (*f* $\longrightarrow$ *x*) *F*
shows (($\lambda x. \text{Re } (f \ x)$) $\longrightarrow$ *Re x*) *F*
 ⟨*proof*⟩

lemma *tendsto-tc-tensor*:

assumes (*f* $\longrightarrow$ *x*) *F*
shows (($\lambda x. \text{tc-tensor } (f \ x) \ a$) $\longrightarrow$ *tc-tensor x a*) *F*

$\langle proof \rangle$

lemma *tc-tensor-has-sum*:

assumes $(f \text{ has-sum } x) \ A$

shows $((\lambda y. \text{tc-tensor } y \ a) \ o \ f \text{ has-sum } \text{tc-tensor } x \ a) \ A$

$\langle proof \rangle$

lemma *Re-has-sum*:

assumes $(f \text{ has-sum } x) \ A$

shows $((\lambda n. \text{Re } n) \ o \ f \text{ has-sum } \text{Re } x) \ A$

$\langle proof \rangle$

Relationship norm and condition

lemma *norm-1-to-cond*:

fixes $A :: 'a \ \text{ell2} \Rightarrow_{CL} 'a \ \text{ell2}$

assumes $\text{norm } A \leq 1$

shows $A^* \ o_{CL} \ A \leq \text{id-cblinfun}$

$\langle proof \rangle$

lemma *cond-to-norm-1*:

fixes $A :: 'a \ \text{ell2} \Rightarrow_{CL} 'a \ \text{ell2}$

assumes $A^* \ o_{CL} \ A \leq \text{id-cblinfun}$

shows $\text{norm } A \leq 1$

$\langle proof \rangle$

Arithmetic mean - quadratic mean inequality (AM-QM)

lemma *arith-quad-mean-ineq*:

fixes $n::\text{nat}$ **and** $x :: \text{nat} \Rightarrow \text{real}$

assumes $\bigwedge i. i \in I \Rightarrow x \ i \geq 0$

shows $(\sum i \in I. x \ i)^{\wedge 2} \leq (\text{card } I) * (\sum i \in I. (x \ i)^{\wedge 2})$

$\langle proof \rangle$

lemma *sqrt-binom*:

assumes $a \geq 0 \ b \geq 0$

shows $|a - b| = |\text{sqrt } a - \text{sqrt } b| * |\text{sqrt } a + \text{sqrt } b|$

$\langle proof \rangle$

Lemmas on *sandwich-tc*

lemma *sandwich-tc-compose'*:

$\text{sandwich-tc } (A \ o_{CL} \ B) \ \varrho = \text{sandwich-tc } A \ (\text{sandwich-tc } B \ \varrho)$

$\langle proof \rangle$

lemma *sandwich-tc-sum*:

$\text{sandwich-tc } E \ (\text{sum } f \ A) = \text{sum } (\text{sandwich-tc } E \ o \ f) \ A$

$\langle proof \rangle$

lemma *isCont-sandwich-tc:*

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

lemma *bounded-linear-trace-norm-sandwich-tc:*

bounded-linear ($\lambda y. \text{trace-tc } (\text{sandwich-tc } E \ y)$)
 ⟨*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* $\otimes_o$ *id-cblinfun*) $\ast_S$ $\top$) $\sqcap$ ((*id-cblinfun* $\otimes_o$ *Q*) $\ast_S$ $\top$)) = *P* $\otimes_o$ *Q*
 ⟨*proof*⟩

lemma *splitting-Proj-or:*

assumes *is-Proj* *P* *is-Proj* *Q*
shows *Proj* (((*P* $\otimes_o$ *id-cblinfun*) $\ast_S$ $\top$) $\sqcup$ ((*id-cblinfun* $\otimes_o$ *Q*) $\ast_S$ $\top$)) =
P $\otimes_o$ (*id-cblinfun* - *Q*) + *id-cblinfun* $\otimes_o$ *Q*
 ⟨*proof*⟩

Additional stuff

lemma *Union-cong:*

assumes $\bigwedge i. i \in A \implies f \ i = g \ i$
shows $(\bigcup i \in A. f \ i) = (\bigcup i \in A. g \ i)$
 ⟨*proof*⟩

lemma *proj-ket-apply:*

proj (*ket* *i*) $\ast_V$ *ket* *j* = (if *i=j* then *ket* *i* else 0)
 ⟨*proof*⟩

Missing lemmas for Kraus maps

```

lemma infsum-in-finite:
  assumes finite F
  assumes  $\langle \text{Hausdorff-space } T \rangle$ 
  assumes  $\langle \text{sum } f \ F \in \text{topspace } T \rangle$ 
  shows  $\langle \text{infsum-in } T \ f \ F = \text{sum } f \ F \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma bdd-above-transform-mono-pos:
  assumes bdd:  $\langle \text{bdd-above } ((\lambda x. g \ x) \ 'M) \rangle$ 
  assumes gpos:  $\langle \bigwedge x. x \in M \implies g \ x \geq 0 \rangle$ 
  assumes mono:  $\langle \text{mono-on } (\text{Collect } ((\leq) \ 0)) \ f \rangle$ 
  shows  $\langle \text{bdd-above } ((\lambda x. f \ (g \ x)) \ 'M) \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma clinear-of-complex[iff]:  $\langle \text{clinear of-complex} \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma inj-on-CARD-1[iff]:  $\langle \text{inj-on } f \ X \rangle$  for  $X :: \langle 'a :: \text{CARD-1 set} \rangle$ 
   $\langle \text{proof} \rangle$ 

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* :: $\langle ('x \Rightarrow ('y::\text{plus})) \Rightarrow ((x \times y) \text{ ell2} \Rightarrow_{CL} (x \times y) \text{ ell2}) \rangle$ **where**
Uquery *H* = *classical-operator* (*Some* *o* ($\lambda(x,y). (x, y + (H \ x))$))

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 $\Rightarrow$ '*mem* update

fixes *Y* :: ('*y*::group-add) update $\Rightarrow$ '*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 $\Rightarrow$ '*l* $\Rightarrow$ '*l*⟩

fixes *bit*:: ⟨'*l* $\Rightarrow$ nat $\Rightarrow$ bool⟩

fixes *valid*:: ⟨'*l* $\Rightarrow$ 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* $\Rightarrow$ valid *l* $\Rightarrow$ valid (*flip* *i* *l*)

— The *flip* operation must be idempotent.

assumes *inj-flip*: inj (*flip* *i*)

assumes *valid-flip-flip*: *i* < *d* $\Rightarrow$ valid *l* $\Rightarrow$ *flip* *i* (*flip* *i* *l*) = *l*

— The *flip* operation must be commutative with itself.

assumes *valid-flip-comm*: *i* < *d* $\Rightarrow$ *j* < *d* $\Rightarrow$ valid *l* $\Rightarrow$ *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*: $i < d \implies \text{valid } l \implies \text{bit } (\text{flip } i \ l) \ i = (\neg (\text{bit } l \ i))$

assumes *valid-bit-flip-diff*: $i < d \implies \text{valid } l \implies i \neq j \implies \text{bit } (\text{flip } i \ l) \ j = \text{bit } l \ j$

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* $\implies i < d \implies \text{blog } (\text{flip } i \ l)$

lemma *blog-empty*: *blog empty*

<proof>

lemma *blog-flip*: $i < d \implies \text{blog } l \implies \text{blog } (\text{flip } i \ l)$

<proof>

lemma *blog-valid*:

blog l $\implies \text{valid } l$

<proof>

lemma *flip-flip*: $i < d \implies \text{blog } l \implies \text{flip } i \ (\text{flip } i \ l) = l$

<proof>

lemma *bit-flip-same*: $i < d \implies \text{blog } l \implies \text{bit } (\text{flip } i \ l) \ i = (\neg (\text{bit } l \ i))$

<proof>

lemma *bit-flip-diff*: $i < d \implies \text{blog } l \implies i \neq j \implies \text{bit } (\text{flip } i \ l) \ j = \text{bit } l \ j$

<proof>

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

fun *list-to-l* :: bool list $\Rightarrow$ '*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*:

$\text{card } (\text{len-d-lists}) = (2::\text{nat})^d$
 $\langle \text{proof} \rangle$

lemma *finite-len-d-lists*[simp]:
finite len-d-lists
 $\langle \text{proof} \rangle$

lemma *blog-list-to-l*:
assumes $\text{length } ls \leq d$
shows $\text{blog } (\text{list-to-l } ls)$
 $\langle \text{proof} \rangle$

lemma *flip-commute*:
assumes $i \neq j \ i < d \ j < d \ \text{length } ls \leq d$
shows $\text{flip } i \ (\text{flip } j \ (\text{list-to-l } ls)) = \text{flip } j \ (\text{flip } i \ (\text{list-to-l } ls))$
 $\langle \text{proof} \rangle$

lemma *flip-list-to-l*:
assumes $i < \text{length } ls \ \text{length } ls \leq d$
shows $\text{flip } i \ (\text{list-to-l } ls) = \text{list-to-l } (ls[\text{length } ls - i - 1 := \neg ls ! (\text{length } ls - i - 1)])$
 $\langle \text{proof} \rangle$

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

definition *empty-list* **where**
 $\text{empty-list} = \text{replicate } d \ \text{False}$

lemma *empty-list-to-l-replicate*:
 $\text{list-to-l } (\text{replicate } n \ \text{False}) = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *empty-list-to-l* [simp]:
 $\text{list-to-l } \text{empty-list} = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *empty-list-len-d*[simp]:
 $\text{empty-list} \in \text{len-d-lists}$
 $\langle \text{proof} \rangle$

lemma *empty-list-to-l-elem* [simp]:
 $\text{empty} \in \text{list-to-l } \text{'len-d-lists}$
 $\langle \text{proof} \rangle$

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

lemma *list-to-l-flip*:
assumes $i < \text{length } ls \ \text{length } ls \leq d$
shows $\text{list-to-l } (ls[i := \neg ls ! i]) = \text{flip } (\text{length } ls - 1 - i) \ (\text{list-to-l } ls)$

$\langle proof \rangle$

lemma *surj-list-to-l*: *list-to-l* ‘ *len-d-lists* = *Collect blog*
 $\langle proof \rangle$

lemma *bit-list-to-l-over*:
 assumes *length l* $\leq d$ *i* $< d$ *length l* $\leq i$
 shows *bit (list-to-l l) i* = *bit empty i*
 $\langle proof \rangle$

lemma *bit-list-to-l*:
 assumes *length l* $\leq d$ *i* $< \text{length } l$
 shows *bit (list-to-l l) i* = (if $l!(\text{length } l - i - 1)$ then $\neg$ *bit empty i* else *bit empty i*)
 $\langle proof \rangle$

lemma *list-to-l-eq*:
 assumes *list-to-l xs* = *list-to-l ys* *length xs* = *d* *length ys* = *d*
 shows *xs* = *ys*
 $\langle proof \rangle$

lemma *inj-list-to-l*: *inj-on list-to-l (len-d-lists)*
 $\langle proof \rangle$

lemma *bij-betw-list-to-l*: *bij-betw list-to-l len-d-lists (Collect blog)*
 $\langle proof \rangle$

lemma *card-blog*: *card (Collect blog)* = 2^d
 $\langle proof \rangle$

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* $\Rightarrow$ *bool list set* **where**
 has-bits A = $\{l. l \in \text{len-d-lists} \wedge \text{True} \in (\lambda i. l!(d - i - 1)) \text{ ‘ } A\}$

lemma *has-bits-empty[simp]*:
 has-bits {} = $\{\}$
 $\langle proof \rangle$

lemma *has-bits-not-empty*:
 assumes *y* $\in$ *has-bits A* *A* $\neq \{\}$ *y* $\in$ *len-d-lists*
 shows *list-to-l y* $\neq$ *empty*
 $\langle proof \rangle$

lemma *has-bits-empty-list*:

empty-list $\notin$ *has-bits* $\{0..<d\}$
 $\langle \text{proof} \rangle$

lemma *has-bits-incl*:
assumes $A \subseteq B$
shows *has-bits* $A \subseteq$ *has-bits* B
 $\langle \text{proof} \rangle$

lemma *has-bits-in-len-d-lists*[*simp*]:
has-bits $A \subseteq$ *len-d-lists*
 $\langle \text{proof} \rangle$

lemma *finite-has-bits*[*simp*]:
finite (*has-bits* A)
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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

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)
 $\langle \text{proof} \rangle$

lemma *has-bits-elem*:
assumes $x \in$ *len-d-lists* $-$ *has-bits* A $a \in A$
shows $\neg x!(d-a-1)$
 $\langle \text{proof} \rangle$

lemma *has-bits-upto-elem*:
assumes $x \in$ *has-bits-upto* n $n < d$
shows $\neg x!(d-n-1)$
 $\langle \text{proof} \rangle$

lemma *has-bits-upto-incl*:
assumes $n \leq m$
shows *has-bits-upto* $n \subseteq$ *has-bits-upto* m
 $\langle \text{proof} \rangle$

lemma *has-bits-upto-d*:
 $has-bits-upto\ d = len-d-lists$
 $\langle proof \rangle$

lemma *empty-list-has-bits-upto*:
 $empty-list \in has-bits-upto\ n$
 $\langle proof \rangle$

lemma *empty-list-to-l-has-bits-upto*:
 $empty \in list-to-l\ 'has-bits-upto\ n$
 $\langle proof \rangle$

lemma *len-d-empty-has-bits*:
 $len-d-lists - \{empty-list\} = has-bits\ \{0..<d\}$
 $\langle proof \rangle$

Properties of d

lemma *two-d-gr-1*:
 $2^d > (1::nat)$
 $\langle proof \rangle$

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

lemma *valid-inv*: $- Collect\ valid = valid - \{False\}$ $\langle proof \rangle$

lemma *blog-inv*: $- Collect\ blog = blog - \{False\}$ $\langle proof \rangle$

lemma *not-blog-flip*: $i < d \implies (\neg\ blog\ l) \implies (\neg\ blog\ (flip\ i\ l))$
 $\langle proof \rangle$

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*:
 $register\ X$
 $\langle proof \rangle$

lemma *register-Y*:
 $register\ Y$
 $\langle proof \rangle$

lemma *X-0*:
 $X\ 0 = 0$ $\langle proof \rangle$

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* $S' = X \text{ (proj-classical-set } (- \text{ (Collect } S')))$

lemma *is-Proj-proj-classical-set*:

is-Proj (proj-classical-set M)

$\langle \text{proof} \rangle$

lemma *proj-classical-set-split-id*:

id-cblinfun = proj-classical-set $M + \text{proj-classical-set } (-M)$

$\langle \text{proof} \rangle$

lemma *proj-classical-set-sum-ket-finite*:

assumes *finite* A

shows proj-classical-set $A = (\sum i \in A. \text{selfbutter } (\text{ket } i))$

$\langle \text{proof} \rangle$

lemma *proj-classical-set-not-elem*:

assumes $i \notin A$

shows proj-classical-set $A *_{\mathcal{V}} \text{ket } i = 0$

$\langle \text{proof} \rangle$

lemma *proj-classical-set-elem*:

assumes $i \in A$

shows proj-classical-set $A *_{\mathcal{V}} \text{ket } i = \text{ket } i$

$\langle \text{proof} \rangle$

lemma *proj-classical-set-upto*:

assumes $i < j$

shows proj-classical-set $\{j..\} *_{\mathcal{V}} \text{ket } (i::\text{nat}) = 0$

$\langle \text{proof} \rangle$

lemma *proj-classical-set-apply*:

assumes *finite* A

shows proj-classical-set $A *_{\mathcal{V}} y = (\sum i \in A. \text{Rep-ell2 } y \ i *_{\mathcal{C}} \text{ket } i)$

$\langle \text{proof} \rangle$

lemma *proj-classical-set-split-Suc*:

proj-classical-set $\{n..\} = \text{proj } (\text{ket } n) + \text{proj-classical-set } \{\text{Suc } n..\}$

$\langle \text{proof} \rangle$

lemma *proj-classical-set-union*:

assumes $\bigwedge x \ y. x \in \text{ket } 'A \implies y \in \text{ket } 'B \implies \text{is-orthogonal } x \ y$

shows proj-classical-set $(A \cup B) = \text{proj-classical-set } A + \text{proj-classical-set } B$

$\langle \text{proof} \rangle$

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

definition *Proj-ket-set* :: $'a \text{ set} \implies ('mem \times 'a) \text{ update where}$

$$\text{Proj-ket-set } A = \text{id-cblinfun } \otimes_o \text{ proj-classical-set } A$$

lemma *Proj-ket-set-vec*:

assumes $y \in A$

shows $\text{Proj-ket-set } A *_V (v \otimes_s \text{ket } y) = v \otimes_s \text{ket } y$

$\langle \text{proof} \rangle$

definition *Proj-ket-upto* :: $\text{bool list set} \Rightarrow ('mem \times 'l) \text{ update}$ **where**

$\text{Proj-ket-upto } A = \text{Proj-ket-set } (\text{list-to-l } 'A)$

lemma *Proj-ket-upto-vec*:

assumes $y \in A$

shows $\text{Proj-ket-upto } A *_V (v \otimes_s \text{ket } (\text{list-to-l } y)) = v \otimes_s \text{ket } (\text{list-to-l } y)$

$\langle \text{proof} \rangle$

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]:

$S\text{-embed } S' + \text{not-}S\text{-embed } S' = \text{id-cblinfun}$

$\langle \text{proof} \rangle$

lemma *S-embed-not-S-embed-add*:

$S\text{-embed } S' (\text{ket } a) + \text{not-}S\text{-embed } S' (\text{ket } a) = \text{ket } a$

$\langle \text{proof} \rangle$

lemma *S-embed-idem* [simp]:

$S\text{-embed } S' \circ_{CL} S\text{-embed } S' = S\text{-embed } S'$

$\langle \text{proof} \rangle$

lemma *S-embed-adj*:

$(S\text{-embed } S')^* = S\text{-embed } S'$

$\langle \text{proof} \rangle$

lemma *not-S-embed-idem*:

$\text{not-}S\text{-embed } S' \circ_{CL} \text{not-}S\text{-embed } S' = \text{not-}S\text{-embed } S'$

$\langle \text{proof} \rangle$

lemma *not-S-embed-adj*:

$(\text{not-}S\text{-embed } S')^* = \text{not-}S\text{-embed } S'$

$\langle \text{proof} \rangle$

lemma *not-S-embed-S-embed* [simp]:

$\text{not-}S\text{-embed } S' \circ_{CL} S\text{-embed } S' = 0$

$\langle \text{proof} \rangle$

lemma *S-embed-not-S-embed* [simp]:

$S\text{-embed } S' \circ_{CL} \text{not-}S\text{-embed } S' = 0$

$\langle \text{proof} \rangle$

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** $\Psi s\ i\ v = (\text{tensor-ell2-right } (\text{ket } i)*) *_{\mathcal{V}} v$

lemma *tensor-ell2-right-compose-id-cblinfun*:

tensor-ell2-right (ket a) o_{CL} A ⊗_o id-cblinfun = A o_{CL} tensor-ell2-right (ket a)**
<proof>

lemma *Ψs-id-cblinfun*:

$\Psi s\ a\ ((A \otimes_o \text{id-cblinfun}) *_{\mathcal{V}} v) = A *_{\mathcal{V}} (\Psi s\ a\ v)$
<proof>

Additional Lemmas

lemma *id-cblinfun-tensor-split-finite*:

assumes *finite A*
shows *(id-cblinfun:: ('mem × 'a) ell2 ⇒_{CL} ('mem × 'a) ell2) =*
(∑ i ∈ A. (tensor-ell2-right (ket i)) o_{CL} (tensor-ell2-right (ket i))) +*
Proj-ket-set (−A)
<proof>

Lemmas on sums of butterflies

lemma *sum-butterfly-ket0*:

assumes *(y::nat) < d+1*
shows *(∑ i < d+1. butterfly (ket 0) (ket i)) *_V (ket y) = ket 0*
<proof>

lemma *sum-butterfly-ket0'*:

*(∑ i < d+1. butterfly (ket 0) (ket i))*_V proj-classical-set {..<d+1} *_V y =*
*(∑ 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 ($\lambda(x,(y::'y)). (x, y + (H x)))$)
 <proof>

lemma *Uquery-ket*:

$Uquery\ F\ *_V\ ket\ (a::'x)\ \otimes_s\ ket\ (b::'y) = ket\ a\ \otimes_s\ ket\ (b + F\ a)$
 <proof>

lemma *unitary-H*: *unitary* ($Uquery\ (H::'x \Rightarrow 'y)$)
 <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 $c \neq 0$

shows $a \otimes_o c = b \otimes_o c \implies a = b$

<proof>

lift-definition *kf-Fst* :: $('a\ ell2, 'c\ ell2, unit)\ kraus-family \Rightarrow$

$(('a \times 'b)\ ell2, ('c \times 'b)\ ell2, unit)\ kraus-family\ is$

$\lambda E. (\lambda(x,-). (x \otimes_o id_cblinfun, ()))\ 'E$

<proof>

lemma *summable-on-in-kf-Fst*:

fixes $f :: 'c \Rightarrow 'a\ ell2 \Rightarrow_{CL} 'a\ ell2$

and $b :: 'b\ ell2 \Rightarrow_{CL} 'b\ ell2$

shows *summable-on-in cweak-operator-topology* ($\lambda x. (fst\ x * o_{CL}\ fst\ x) \otimes_o id_cblinfun$) (*Rep-kraus-family* G)

<proof>

lemma *infsum-in-kf-Fst*:

fixes $f :: 'c \Rightarrow 'a \text{ ell2} \Rightarrow_{CL} 'a \text{ ell2}$
and $b :: 'b \text{ ell2} \Rightarrow_{CL} 'b \text{ ell2}$
shows $\text{infsum-in cweak-operator-topology } (\lambda x. \text{fst } x * o_{CL} \text{fst } x) \otimes_o \text{id-cblinfun} \text{ (Rep-kraus-family } G) \leq$
 $(\text{infsum-in cweak-operator-topology } (\lambda x. \text{fst } x * o_{CL} \text{fst } x) \text{ (Rep-kraus-family } G)) \otimes_o$
 id-cblinfun
 $\langle \text{proof} \rangle$

lemma *kf-bound-kf-Fst*:
 $\text{kf-bound } (\text{kf-Fst } F :: ('a \times 'b) \text{ ell2}, ('c \times 'b) \text{ ell2}, \text{unit}) \text{ kraus-family} \leq$
 $\text{kf-bound } F \otimes_o \text{id-cblinfun}$
 $\langle \text{proof} \rangle$

lemma *sandwich-tc-kf-apply-Fst*:
 $\text{sandwich-tc } (\text{Snd } (Q :: 'd \text{ update})) (\text{kf-apply } (\text{kf-Fst } F ::$
 $('a \times 'd) \text{ ell2}, ('a \times 'd) \text{ ell2}, \text{unit}) \text{ kraus-family} \varrho) =$
 $\text{kf-apply } (\text{kf-Fst } F) (\text{sandwich-tc } (\text{Snd } Q) \varrho)$
 $\langle \text{proof} \rangle$

kraus family Fst is trace preserving.

lemma *kf-apply-Fst-tensor*:
 $\langle \text{kf-apply } (\text{kf-Fst } \mathfrak{E} :: ('c \times 'b) \text{ ell2}, ('a \times 'b) \text{ ell2}, \text{unit}) \text{ kraus-family}$
 $(\text{tc-tensor } \varrho \sigma) = \text{tc-tensor } (\text{kf-apply } \mathfrak{E} \varrho) \sigma \rangle$
 $\langle \text{proof} \rangle$

lemma *partial-trace-ignore-trace-preserving-map-Fst*:
assumes $\langle \text{kf-trace-preserving } \mathfrak{E} \rangle$
shows $\langle \text{partial-trace } (\text{kf-apply } (\text{kf-Fst } \mathfrak{E}) \varrho) =$
 $\text{kf-apply } \mathfrak{E} (\text{partial-trace } \varrho) \rangle$
 $\langle \text{proof} \rangle$

lemma *trace-preserving-kf-Fst*:
assumes $\text{km-trace-preserving } (\text{kf-apply } E)$
shows $\text{km-trace-preserving } (\text{kf-apply } ($
 $\text{kf-Fst } E :: ('a \times 'c) \text{ ell2}, ('a \times 'c) \text{ ell2}, \text{unit}) \text{ kraus-family}))$
 $\langle \text{proof} \rangle$

Summability on Kraus maps

lemma *finite-kf-apply-has-sum*:
assumes $(f \text{ has-sum } x) \ A$
shows $((\text{kf-apply } \mathfrak{F} \circ f) \text{ has-sum } \text{kf-apply } \mathfrak{F} \ x) \ A$
 $\langle \text{proof} \rangle$

lemma *finite-kf-apply-abs-summable-on*:
assumes $f \text{ abs-summable-on } A$
shows $(\text{kf-apply } \mathfrak{F} \circ f) \text{ abs-summable-on } A$
 $\langle \text{proof} \rangle$

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 $Ub :: nat \Rightarrow 'l \text{ update}$ **where**
 $Ub\ i = \text{classical-operator } (Some\ o\ (flip\ i))$

lemma *Ub-exists: classical-operator-exists* $(Some\ o\ flip\ i)$
 $\langle proof \rangle$

lemma *isometry-Ub:*
 $isometry\ (Ub\ k)$
 $\langle proof \rangle$

lemma *Ub-ket-range:* $(Ub\ i *_{\mathcal{V}} ket\ y) \in range\ ket\ \langle proof \rangle$

lemma *Ub-ket:*
 $Ub\ k *_{\mathcal{V}} (ket\ x) = ket\ (flip\ k\ x)$
 $\langle proof \rangle$

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 $US :: \langle 'x \Rightarrow bool \rangle \Rightarrow nat \Rightarrow ('mem \times 'l) \text{ update}$ **where**
 $\langle US\ S\ k = S\text{-embed } S \otimes_o Ub\ k + not\text{-}S\text{-embed } S \otimes_o id\text{-}cblinfun \rangle$

lemma *isometry-US:*
 $isometry\ (US\ S\ k)$
 $\langle proof \rangle$

lemma *US-ket-split:*
 $(US\ S\ k) *_{\mathcal{V}} ket\ (x, y) = (S\text{-embed } S *_{\mathcal{V}} ket\ x) \otimes_s ((Ub\ k) *_{\mathcal{V}} ket\ y) + (not\text{-}S\text{-embed } S *_{\mathcal{V}} ket\ x) \otimes_s ket\ y$
 $\langle proof \rangle$

lemma *US-ket-only01*:

$US\ S\ k\ (v \otimes_s ket\ n) = (not\text{-}S\text{-}embed\ S\ *_V\ v) \otimes_s ket\ n + (S\text{-}embed\ S\ *_V\ v) \otimes_s ket\ (flip\ k\ n)$
 $\langle proof \rangle$

lemma *norm-US*:

assumes $i < Suc\ d$ **shows** $norm\ (US\ S\ i) = 1$
 $\langle proof \rangle$

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 $proj\text{-}classical\text{-}set\ (-\ Collect\ blog)\ o_{CL}\ Ub\ n =$
 $Ub\ n\ o_{CL}\ proj\text{-}classical\text{-}set\ (-\ Collect\ blog)$
 $\langle proof \rangle$

lemma *proj-classical-set-over-Ub*:

assumes $n \leq d\ m < d$
shows $proj\text{-}classical\text{-}set\ (list\text{-}to\text{-}l\ 'has\text{-}bits\ \{n..<d\})\ o_{CL}\ Ub\ m =$
 $Ub\ m\ o_{CL}\ proj\text{-}classical\text{-}set\ (flip\ m\ 'list\text{-}to\text{-}l\ 'has\text{-}bits\ \{n..<d\})$
 $\langle proof \rangle$

lemma *Ψ_S -US-Proj-ket-upto*:

assumes $i < d$
shows $tensor\text{-}ell2\text{-}right\ (ket\ empty)*\ o_{CL}\ ((US\ S\ i)\ o_{CL}\ Proj\text{-}ket\text{-}upto\ (has\text{-}bits\text{-}upto\ i)) =$
 $not\text{-}S\text{-}embed\ S\ o_{CL}\ tensor\text{-}ell2\text{-}right\ (ket\ empty)*$
 $\langle proof \rangle$

end

unbundle *no cblinfun-syntax*

unbundle *no lattice-syntax*

end

theory *Unitary-S-prime*

imports *Definition-O2H*

begin

unbundle *cblinfun-syntax*

unbundle *lattice-syntax*

context *o2h-setting*

begin

1.3 Towards the Definition of $U-S'$

For the definition of $U-S'$, we need a counting function on the additional register. We model this with the function $c\text{-add}$ 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 $c\text{-add} :: \text{nat} \Rightarrow \text{nat}$ **where**
 $c\text{-add } c = (\text{if } c < d+1 \text{ then } (c+1) \text{ mod } (d+1) \text{ else } c)$

lemma $\text{surj-}c\text{-add-}c\text{-valid}$: $c\text{-add } \{..<d+1\} = \{..<d+1\}$
 $\langle \text{proof} \rangle$

$c\text{-add}$ needs to be bijective, so that the resulting operator is unitary.

lemma $\text{inj-}c\text{-add}$: $\text{inj } c\text{-add}$
 $\langle \text{proof} \rangle$

lemma $\text{surj-}c\text{-add}$: $c\text{-add } UNIV = UNIV$
 $\langle \text{proof} \rangle$

lemma $\text{bij-}c\text{-add}$: $\text{bij } c\text{-add}$
 $\langle \text{proof} \rangle$

lemma $c\text{-add-}0$: $c\text{-add } 0 \neq 0$
 $\langle \text{proof} \rangle$

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

definition $Uc = \text{classical-operator } (\text{Some } o \text{ } c\text{-add})$

lemma $Uc\text{-exists}$:
 $\text{classical-operator-exists } (\text{Some } o \text{ } c\text{-add})$
 $\langle \text{proof} \rangle$

lemma $\text{unitary-}Uc$:
 $\text{unitary } Uc$
 $\langle \text{proof} \rangle$

lemma $Uc\text{-ket-}d$:
 $Uc *_{\text{V}} \text{ket } d = \text{ket } 0$
 $\langle \text{proof} \rangle$

lemma $Uc\text{-ket-less}$:
assumes $n < d$
shows $Uc *_{\text{V}} \text{ket } n = \text{ket } (n+1)$
 $\langle \text{proof} \rangle$

lemma $Uc\text{-ket-leq}$:
assumes $n < d+1$
shows $Uc *_{\text{V}} \text{ket } n = \text{ket } ((n+1) \text{ mod } (d+1))$
 $\langle \text{proof} \rangle$

lemma *Uc-ket-greater*:
assumes $n > d$
shows $Uc *_{\mathcal{V}} \text{ket } n = \text{ket } n$
 $\langle \text{proof} \rangle$

lemma *Uc-ket-range*:
 $(Uc *_{\mathcal{V}} \text{ket } y) \in \text{range } \text{ket } \langle \text{proof} \rangle$

lemma *Uc-ket-range-valid*:
assumes $y < d+1$
shows $(Uc *_{\mathcal{V}} \text{ket } y) \in \text{ket } \{..<d+1\} \langle \text{proof} \rangle$

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 $U-S' :: \langle ('x \Rightarrow \text{bool}) \Rightarrow ('mem \times nat) \text{ update} \rangle$ **where**
 $\langle U-S' S = S\text{-embed } S \otimes_o Uc + \text{not-}S\text{-embed } S \otimes_o \text{id-cblinfun} \rangle$

lemma *unitary-U-S'*:
 $\text{unitary } (U-S' S)$
 $\langle \text{proof} \rangle$

lemma *iso-U-S': isometry (U-S' S)*
 $\langle \text{proof} \rangle$

lemma *U-S'-ket-split*:
 $U-S' S *_{\mathcal{V}} \text{ket } (x,y) = (S\text{-embed } S *_{\mathcal{V}} \text{ket } x) \otimes_s (Uc *_{\mathcal{V}} \text{ket } y) + (\text{not-}S\text{-embed } S *_{\mathcal{V}} \text{ket } x) \otimes_s \text{ket } y$
 $\langle \text{proof} \rangle$

lemma *norm-U-S'*:
assumes $i < \text{Suc } d$ **shows** $\text{norm } (U-S' S) = 1$
 $\langle \text{proof} \rangle$

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 $\text{tensor-ell2-right } (\text{ket } 0) * o_{CL} (U-S' S o_{CL} \text{Proj-ket-set } \{..<i+1\}) = \text{not-}S\text{-embed } S o_{CL} \text{tensor-ell2-right } (\text{ket } 0) *$
 $\langle \text{proof} \rangle$

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* $\Rightarrow$ (*'a ell2*, *'a ell2*, *unit*) *kraus-family*

type-synonym *'a pure-adv* = *nat* $\Rightarrow$ (*nat* $\Rightarrow$ *'a update*) $\Rightarrow$ *'a ell2*

type-synonym *'a mixed-adv* = *nat* $\Rightarrow$ *'a kraus-adv* $\Rightarrow$ *'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 $H : X \rightarrow Y$ 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* $\Rightarrow$ (*nat* $\Rightarrow$ *'a update*) $\Rightarrow$ (*nat* $\Rightarrow$ *'a update*) $\Rightarrow$ *'a ell2* $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ *'a ell2* **where**

run-pure-adv 0 *UAs UB init X Y H* = (*UAs* 0) $\ast_V$ *init*

| *run-pure-adv* (*Suc n*) *UAs UB init X Y H* =

(*UAs* (*Suc n*)) $\ast_V$ (*X; Y*) (*Uquery H*) $\ast_V$ (*UB n*) $\ast_V$ (*run-pure-adv n UAs UB init X Y H*)

fun *run-pure-adv-update* :: *nat* $\Rightarrow$ (*nat* $\Rightarrow$ *'a update*) $\Rightarrow$ (*nat* $\Rightarrow$ *'a update*) $\Rightarrow$ *'a ell2* $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ *'a update* **where**

run-pure-adv-update 0 *UAs UB init X Y H* = *sandwich* (*UAs* 0) $\ast_V$ (*selfbutter init*)

| *run-pure-adv-update* (*Suc n*) *UAs UB init X Y H* =

sandwich (*UAs* (*Suc n*) *oCL* (*X; Y*) (*Uquery H*) *oCL* *UB n*) $\ast_V$ (*run-pure-adv-update n UAs UB init X Y H*)

fun *run-pure-adv-tc* :: $\text{nat} \Rightarrow (\text{nat} \Rightarrow 'a \text{ update}) \Rightarrow (\text{nat} \Rightarrow 'a \text{ update}) \Rightarrow 'a \text{ ell2} \Rightarrow - \Rightarrow - \Rightarrow -$
 $\Rightarrow 'a \text{ tc-op}$ **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
 $\langle \text{proof} \rangle$

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*)
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

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*)
 $\langle \text{proof} \rangle$

Definitions for mixed adversaries

fun *run-mixed-adv* ::

$\text{nat} \Rightarrow 'a \text{ kraus-adv} \Rightarrow (\text{nat} \Rightarrow 'a \text{ update}) \Rightarrow 'a \text{ ell2} \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow 'a \text{ tc-op}$ **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
 $\langle \text{proof} \rangle$

lemma (**in** *o2h-setting*) *norm-run-mixed-adv*:

assumes *Es-norm-id*: $\bigwedge i. i < d+1 \implies \text{kf-bound } (\text{Es } i) \leq \text{id-cblinfun}$
and *n*: $n < d+1$
and *normUB*: $\bigwedge i. i < d+1 \implies \text{norm } (\text{UB } i) \leq 1$
and *register*: *register* (*X'*; *Y'*)
and *norm-init'*: $\text{norm } \text{init}' = 1$
fixes *H* :: $'x \Rightarrow 'y$

shows $\text{norm } (\text{run-mixed-adv } n \text{ } Es \text{ } UB \text{ } init' \text{ } X' \text{ } Y' \text{ } H) \leq 1$
 $\langle \text{proof} \rangle$

Trace preserving Kraus maps/adversaries preserve the norm.

lemma *km-trace-preserving-kf-apply*:
assumes *km-trace-preserving* (*kf-apply* *F*) $\varrho \geq 0$
shows $\text{norm } (\text{kf-apply } F \text{ } \varrho) = \text{norm } \varrho$
 $\langle \text{proof} \rangle$

lemma (*in o2h-setting*) *trace-preserving-norm-run-mixed-adv*:
assumes *trace-pres*: $\bigwedge i. i < d+1 \implies \text{km-trace-preserving } (\text{kf-apply } (Es \text{ } i))$
and *n*: $n < d+1$
and *iso-UB*: $\bigwedge i. i < d+1 \implies \text{isometry } (UB \text{ } i)$
and *register*: *register* ($X'; Y'$)
and *norm-init'*: $\text{norm } init' = 1$
fixes $H :: 'x \Rightarrow 'y$
shows $\text{norm } (\text{run-mixed-adv } n \text{ } Es \text{ } UB \text{ } init' \text{ } X' \text{ } Y' \text{ } H) = 1$
 $\langle \text{proof} \rangle$

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**
 $\text{run-pure-A-ell2 } UA \text{ } H = \text{run-pure-adv } d \text{ } UA \text{ } (\lambda-. \text{id-cblinfun}) \text{ } init \text{ } X \text{ } Y \text{ } H$

definition *run-pure-A-update* $:: (nat \Rightarrow 'mem \text{ update}) \Rightarrow ('x \Rightarrow 'y) \Rightarrow 'mem \text{ update}$ **where**
 $\text{run-pure-A-update } UA \text{ } H = \text{run-pure-adv-update } d \text{ } UA \text{ } (\lambda-. \text{id-cblinfun}) \text{ } init \text{ } X \text{ } Y \text{ } H$

definition *run-pure-A-tc* $:: (nat \Rightarrow 'mem \text{ update}) \Rightarrow ('x \Rightarrow 'y) \Rightarrow 'mem \text{ tc-op}$ **where**
 $\text{run-pure-A-tc } UA \text{ } H = \text{run-pure-adv-tc } d \text{ } UA \text{ } (\lambda-. \text{id-cblinfun}) \text{ } init \text{ } X \text{ } Y \text{ } H$

lemma *run-pure-A-ell2-update*:
 $\text{run-pure-A-update } UA \text{ } H = \text{selfbutter } (\text{run-pure-A-ell2 } UA \text{ } H)$
 $\langle \text{proof} \rangle$

lemma *run-pure-A-update-tc*:
 $\text{run-pure-A-tc } UA \text{ } H = \text{Abs-trace-class } (\text{run-pure-A-update } UA \text{ } H)$
 $\langle \text{proof} \rangle$

lemma *run-pure-A-tc-pos*: $0 \leq \text{run-pure-A-tc } UA \text{ } H$
 $\langle \text{proof} \rangle$

For mixed adversaries

definition *run-mixed-A* :: *'mem kraus-adv* $\Rightarrow$ (*'x* $\Rightarrow$ *'y*) $\Rightarrow$ *'mem tc-op* **where**
run-mixed-A kraus-A H = *run-mixed-adv d kraus-A* ($\lambda\cdot$. *id-cblinfun*) *init X Y H*

lemma *run-mixed-A-pos*:
 $0 \leq \text{run-mixed-A kraus-A } H$
 $\langle \text{proof} \rangle$

lemma *norm-run-mixed-A*:
assumes *F-norm-id*: $\bigwedge i. i < d+1 \implies \text{kf-bound } (F \ i) \leq \text{id-cblinfun}$
shows *norm (run-mixed-A F H)* ≤ 1
 $\langle \text{proof} \rangle$

Embeddings of *X* and *Y* in the counting register of *B*

definition *X-for-B* :: $\langle 'x \text{ update} \Rightarrow ('mem \times 'l) \text{ update} \rangle$ **where**
 $\langle X\text{-for-B} = \text{Fst} \circ X \rangle$

definition *Y-for-B* :: $\langle 'y \text{ update} \Rightarrow ('mem \times 'l) \text{ update} \rangle$ **where**
 $\langle Y\text{-for-B} = \text{Fst} \circ Y \rangle$

lemma [*register*]: $\langle \text{register } X\text{-for-B} \rangle$
 $\langle \text{proof} \rangle$

lemma [*register*]: $\langle \text{register } Y\text{-for-B} \rangle$
 $\langle \text{proof} \rangle$

lemma *register-XY-for-B*:
 $\text{register } (X\text{-for-B}; Y\text{-for-B}) \langle \text{proof} \rangle$

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

lemma *UqueryH-tensor-id-cblinfunB*:
 $(X\text{-for-B}; Y\text{-for-B}) (U\text{query } H) = (X; Y) (U\text{query } H) \otimes_o \text{id-cblinfun}$
 $\langle \text{proof} \rangle$

The oracle query on the extended register stays unitary.

lemma *unitary-H-B*: *unitary* $((X\text{-for-B}; Y\text{-for-B}) (U\text{query } H))$
 $\langle \text{proof} \rangle$

lemma *iso-H-B*: *isometry* $((X\text{-for-B}; Y\text{-for-B}) (U\text{query } H))$
 $\langle \text{proof} \rangle$

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 $\langle \text{init-B} = \text{init} \otimes_s \text{ket empty} \rangle$

lemma *norm-init-B*: *norm init-B* = 1
 $\langle \text{proof} \rangle$

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* $\Rightarrow$ '*mem* *update*) $\Rightarrow$ ('*x* $\Rightarrow$ '*y*) $\Rightarrow$ ('*x* $\Rightarrow$ *bool*) $\Rightarrow$ ('*mem* $\times$ '*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* $\Rightarrow$ '*mem* *update*) $\Rightarrow$ ('*x* $\Rightarrow$ '*y*) $\Rightarrow$ ('*x* $\Rightarrow$ *bool*) $\Rightarrow$ ('*mem* $\times$ '*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*)
 $\langle \text{proof} \rangle$

lemma *run-pure-B-update-tc*:

run-pure-B-tc *UB H S* = *Abs-trace-class* (*run-pure-B-update* *UB H S*)
 $\langle \text{proof} \rangle$

lemma *run-pure-B-update-tc'*:

run-pure-B-update *UB H S* = *from-trace-class* (*run-pure-B-tc* *UB H S*)
 $\langle \text{proof} \rangle$

lemma *run-pure-B-tc-pos*: $0 \leq \text{run-pure-B-tc } UB H S$

$\langle \text{proof} \rangle$

For mixed adversaries

definition *run-mixed-B* ::

'*mem* *kraus-adv* $\Rightarrow$ ('*x* $\Rightarrow$ '*y*) $\Rightarrow$ ('*x* $\Rightarrow$ *bool*) $\Rightarrow$ ('*mem* $\times$ '*l*) *tc-op* **where**
run-mixed-B *kraus-B* *H S* = *run-mixed-adv* *d* ($\lambda n. \text{kf-Fst } (\text{kraus-B } n)$)
(*US S*) *init-B* *X-for-B* *Y-for-B* *H*

lemma *run-mixed-B-pos*:

$0 \leq \text{run-mixed-B } \text{kraus-B } H S$
 $\langle \text{proof} \rangle$

lemma *norm-run-mixed-B*:

assumes *F-norm-id*: $\bigwedge i. i < d+1 \Rightarrow \text{kf-bound } (F i) \leq \text{id-cblinfun}$
shows *norm* (*run-mixed-B* *F* *H S*) ≤ 1
 $\langle \text{proof} \rangle$

lemma *trace-preserving-norm-run-mixed-B*:

assumes $\bigwedge i. i < d+1 \Rightarrow \text{km-trace-preserving } (\text{kf-apply } (\text{kf-Fst } (F i)) : (('mem \times 'l) \text{ ell2}, ('mem \times 'l) \text{ ell2}, \text{unit}) \text{ kraus-family}))$

shows $\text{norm } (\text{run-mixed-}B \ F \ H \ S) = 1$
 $\langle \text{proof} \rangle$

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 $X\text{-for-}C :: \langle 'x \text{ update} \Rightarrow ('mem \times nat) \text{ update} \rangle$ **where**
 $\langle X\text{-for-}C = Fst \circ X \rangle$

definition $Y\text{-for-}C :: \langle 'y \text{ update} \Rightarrow ('mem \times nat) \text{ update} \rangle$ **where**
 $\langle Y\text{-for-}C = Fst \circ Y \rangle$

lemma $[register]: \langle \text{register } X\text{-for-}C \rangle$
 $\langle \text{proof} \rangle$

lemma $[register]: \langle \text{register } Y\text{-for-}C \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{register-}XY\text{-for-}C:$
 $\text{register } (X\text{-for-}C; Y\text{-for-}C) \langle \text{proof} \rangle$

The oracle query on the extended register stays unitary.

lemma $\text{unitary-}H\text{-}C: \text{unitary } ((X\text{-for-}C; Y\text{-for-}C) (U_{\text{query}} H))$
 $\langle \text{proof} \rangle$

lemma $\text{iso-}H\text{-}C: \text{isometry } ((X\text{-for-}C; Y\text{-for-}C) (U_{\text{query}} H))$
 $\langle \text{proof} \rangle$

Alternative representation of $U_{\text{query}} H$.

lemma $U_{\text{query}}H\text{-tensor-id-cblinfun}C:$
 $(X\text{-for-}C; Y\text{-for-}C) (U_{\text{query}} H) = (X; Y) (U_{\text{query}} H) \otimes_o \text{id-cblinfun}$
 $\langle \text{proof} \rangle$

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

definition $\text{init-}B\text{-count} :: ('mem \times nat) \text{ ell2}$ **where**
 $\langle \text{init-}B\text{-count} = \text{init} \otimes_s \text{ket } 0 \rangle$

lemma $\text{norm-init-}B\text{-count}:$
 $\text{norm } (\text{init-}B\text{-count}) = 1$
 $\langle \text{proof} \rangle$

Definition of adversary B with counting

For pure adversaries

definition *run-pure-B-count-ell2* **where**

run-pure-B-count-ell2 $UB\ H\ S = \text{run-pure-adv}\ d\ (Fst\ o\ UB)\ (\lambda\cdot. U\text{-}S'\ S)\ \text{init-B-count}\ X\text{-for-}C\ Y\text{-for-}C\ H$

definition *run-pure-B-count-update* ::

$(nat \Rightarrow 'mem\ update) \Rightarrow ('x \Rightarrow 'y) \Rightarrow ('x \Rightarrow bool) \Rightarrow ('mem \times nat)\ update$ **where**
run-pure-B-count-update $UB\ H\ S = \text{run-pure-adv-update}\ d\ (Fst\ o\ UB)\ (\lambda\cdot. U\text{-}S'\ S)\ \text{init-B-count}\ X\text{-for-}C\ Y\text{-for-}C\ H$

definition *run-pure-B-count-tc* ::

$(nat \Rightarrow 'mem\ update) \Rightarrow ('x \Rightarrow 'y) \Rightarrow ('x \Rightarrow bool) \Rightarrow ('mem \times nat)\ tc\text{-op}$ **where**
run-pure-B-count-tc $UB\ H\ S = \text{run-pure-adv-tc}\ d\ (Fst\ o\ UB)\ (\lambda\cdot. U\text{-}S'\ S)\ \text{init-B-count}\ X\text{-for-}C\ Y\text{-for-}C\ H$

lemma *run-pure-B-count-ell2-update*:

run-pure-B-count-update $UB\ H\ S = \text{selfbutter}\ (\text{run-pure-B-count-ell2}\ UB\ H\ S)$
 $\langle \text{proof} \rangle$

lemma *run-pure-B-count-update-tc*:

run-pure-B-count-tc $UB\ H\ S = \text{Abs-trace-class}\ (\text{run-pure-B-count-update}\ UB\ H\ S)$
 $\langle \text{proof} \rangle$

lemma *run-pure-B-count-update-tc'*:

run-pure-B-count-update $UB\ H\ S = \text{from-trace-class}\ (\text{run-pure-B-count-tc}\ UB\ H\ S)$
 $\langle \text{proof} \rangle$

lemma *run-pure-B-count-tc-pos*: $0 \leq \text{run-pure-B-count-tc}\ UB\ H\ S$

$\langle \text{proof} \rangle$

For mixed adversaries

definition *run-mixed-B-count* ::

$'mem\ \text{kraus-adv} \Rightarrow ('x \Rightarrow 'y) \Rightarrow ('x \Rightarrow bool) \Rightarrow ('mem \times nat)\ tc\text{-op}$ **where**
run-mixed-B-count $\text{kraus-B}\ H\ S = \text{run-mixed-adv}\ d\ (\lambda n. \text{kf-Fst}\ (\text{kraus-B}\ n))$
 $(\lambda n. U\text{-}S'\ S)\ \text{init-B-count}\ X\text{-for-}C\ Y\text{-for-}C\ H$

lemma *run-mixed-B-count-pos*:

$0 \leq \text{run-mixed-B-count}\ \text{kraus-B}\ H\ S$
 $\langle \text{proof} \rangle$

lemma *norm-run-mixed-B-count*:

assumes $F\text{-norm-id}: \bigwedge i. i < d+1 \implies \text{kf-bound}\ (F\ i) \leq \text{id-cblinfun}$
shows $\text{norm}\ (\text{run-mixed-B-count}\ F\ H\ S) \leq 1$
 $\langle \text{proof} \rangle$

lemma *trace-preserving-norm-run-mixed-B-count*:

assumes $\bigwedge i. i < d+1 \implies \text{km-trace-preserving}\ (\text{kf-apply}$

```

(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: ⟨ $\bigwedge i. i < d+1 \implies \text{norm } (UA\ i) \leq 1$ ⟩
begin

```

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

definition $\langle run-A = run-pure-adv\ d\ UA\ (\lambda-. id-cblinfun::'mem\ update)\ init\ X\ Y\ H \rangle$

```

lemma norm-UA-Suc:  $n < d \implies \text{norm } (UA\ (Suc\ n)) \leq 1$ 
⟨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)  $\circ_{CL}$  (UA (Suc n)  $\otimes_o$  id-cblinfun) =
  (UA (Suc n)  $\otimes_o$  id-cblinfun)  $\circ_{CL}$  (id-cblinfun  $\otimes_o$  proj-classical-set A)
   $\langle$ proof $\rangle$ 

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 $'l$ whether a value in S was queried and in which step with a unitary $U\text{-}S$.

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

definition $\langle \textit{run-B-upto } n = \textit{run-pure-adv } n \ (\lambda i. \textit{UA } i \otimes_o \textit{id-cblinfun}) \ (\textit{US } S) \ \textit{init-B } X\text{-for-B } Y\text{-for-B } H \rangle$

definition $\langle \textit{run-B} = \textit{run-pure-adv } d \ (\lambda i. \textit{UA } i \otimes_o \textit{id-cblinfun}) \ (\textit{US } S) \ \textit{init-B } X\text{-for-B } Y\text{-for-B } H \rangle$

lemma $\textit{run-B-altdef}$: $\textit{run-B} = \textit{run-B-upto } d$
 $\langle$ *proof* $\rangle$

lemma *run-B-upto-I*:

run-B-upto (Suc *n*) = (UA (Suc *n*) $\otimes_o$ id-cblinfun) $\ast_V$ (X-for-B; Y-for-B) (Uquery *H*) $\ast_V$
 US *S n* $\ast_V$ *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 $\text{norm } (\text{run-B-upto } n) \leq 1$
 ⟨proof⟩

lemma *norm-run-B*:

$\text{norm run-B} \leq 1$
 ⟨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 $n \leq d$
shows *Proj-ket-set* (– *Collect blog*) $\ast_V$ *run-B-upto n* = 0
 ⟨proof⟩

lemma *orth-run-B-upto*:

fixes *C* :: 'mem update
assumes *y*: $y \in \text{has-bits } \{\text{Suc } m..<d\}$ **and** *m*: $m < d$
shows *is-orthogonal* ((*C* $\otimes_o$ id-cblinfun) $\ast_V$ *run-B-upto m*) ($x \otimes_s \text{ket } (\text{list-to-l } y)$)
 ⟨proof⟩

lemma *orth-run-B-upto-ket*:

assumes *y*: $y \in \text{has-bits } \{\text{Suc } m..<d\}$ **and** *Sucm*: $\text{Suc } m < d$
shows *is-orthogonal* (*run-B-upto m*) ($x \otimes_s \text{ket } (\text{list-to-l } y)$)
 ⟨proof⟩

lemma *orth-run-B-upto-flip*:

assumes *y*: $y \in \text{has-bits } \{\text{Suc } m..<d\}$ **and** *Sucm*: $\text{Suc } m < d$
shows *is-orthogonal* (*run-B-upto m*) ($x \otimes_s \text{ket } (\text{flip } m (\text{list-to-l } y))$)
 ⟨proof⟩

lemma *run-B-upto-proj-over*:

assumes $n \leq d$
shows *Proj-ket-set* (*list-to-l* ‘ *has-bits* {*n*..*d*}) $\ast_V$ *run-B-upto n* = 0
 ⟨proof⟩

How Ψ s 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 $x \in \text{len-}d\text{-lists}$

shows $x \neq \text{empty-list} \longleftrightarrow (\exists i < d. x!i)$

<proof>

First we show the mere existence of such a form.

lemma *run-B-upto-sum*:

assumes $n < d$

shows $\exists v. \text{run-B-upto } n = (\sum_{i \in \text{has-bits-upto } n} v \ i \otimes_s \text{ket } (\text{list-to-l } i))$

<proof>

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

lemma *run-B-projection*:

assumes $n < d$

shows $\text{Proj-ket-upto } (\text{has-bits-upto } n) *_V \text{run-B-upto } n = \text{run-B-upto } n$

<proof>

How Ψ s relate to *run-B*.

lemma *run-B-upto-split*:

assumes $n \leq d$

shows $\text{run-B-upto } n = (\sum_{i \in \text{has-bits-upto } n} \Psi s \ (\text{list-to-l } i) \ (\text{run-B-upto } n) \otimes_s \text{ket } (\text{list-to-l } i))$

<proof>

lemma *run-B-split*:

$\text{run-B} = (\sum_{i \in \text{len-}d\text{-lists}} \Psi s \ (\text{list-to-l } i) \ \text{run-B} \otimes_s (\text{ket } (\text{list-to-l } i)))$

<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 $\text{init} \otimes_s \text{ket } 0 :: 'mem \times \text{nat}$, we run the adversary with counting by adding $+1$ in $\{0..<d+1\}$. $\text{run-B-count-upto } n$ 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 $\langle \text{run-B-count-upto } n =$
 $\text{run-pure-adv } n (\lambda i. \text{UA } i \otimes_o \text{id-cblinfun}) (\lambda-. \text{U-S'} S) \text{init-B-count } X\text{-for-C } Y\text{-for-C } H \rangle$

definition $\langle \text{run-B-count} = \text{run-pure-adv } d (\lambda i. \text{UA } i \otimes_o \text{id-cblinfun}) (\lambda-. \text{U-S'} S) \text{init-B-count}$
 $X\text{-for-C } Y\text{-for-C } H \rangle$

lemma $\text{run-B-count-altdef: run-B-count} = \text{run-B-count-upto } d$
 $\langle \text{proof} \rangle$

lemma $\text{run-B-count-upto-I:}$
 $\text{run-B-count-upto } (\text{Suc } n) = (\text{UA } (\text{Suc } n) \otimes_o \text{id-cblinfun}) *_V (X\text{-for-C}; Y\text{-for-C}) (\text{Uquery } H)$
 $*_V$
 $\text{U-S'} S *_V \text{run-B-count-upto } n$
 $\langle \text{proof} \rangle$

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

lemma $\text{norm-run-B-count-upto:}$
assumes $n < d+1$
shows $\text{norm } (\text{run-B-count-upto } n) \leq 1$
 $\langle \text{proof} \rangle$

lemma norm-run-B-count:
 $\text{norm run-B-count} \leq 1$
 $\langle \text{proof} \rangle$

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

Preparation for representation of run-B-count

lemma $\text{tensor-proj-UqueryH-commute:}$

$(id\text{-}cblinfun \otimes_o proj\text{-}classical\text{-}set A) \circ_{CL} (X\text{-}for\text{-}C; Y\text{-}for\text{-}C) (Uquery H) =$
 $(X\text{-}for\text{-}C; Y\text{-}for\text{-}C) (Uquery H) \circ_{CL} (id\text{-}cblinfun \otimes_o proj\text{-}classical\text{-}set A)$
 $\langle proof \rangle$

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

lemma *proj-Uc:*

assumes $m > 0$

shows $proj\text{-}classical\text{-}set \{m\} \circ_{CL} Uc = Uc \circ_{CL}$

$(if\ m < d+1\ then\ proj\text{-}classical\text{-}set \{m-1\}\ else\ proj\text{-}classical\text{-}set \{m\})$

$\langle proof \rangle$

lemma *proj-classical-set-over-Uc:*

$proj\text{-}classical\text{-}set \{Suc\ n..\} \circ_{CL} Uc = Uc \circ_{CL} proj\text{-}classical\text{-}set$

$(if\ n > d\ then\ \{Suc\ n..\}\ else\ \{n..\} - \{d\})$

$\langle proof \rangle$

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

lemma *run-B-count-proj-gr:*

assumes $m > n$

shows $Proj\text{-}ket\text{-}set \{m\} *_V run\text{-}B\text{-}count\text{-}upto\ n = 0$

$\langle proof \rangle$

lemma *run-B-count-upto-proj-over:*

$Proj\text{-}ket\text{-}set \{n+1..\} *_V run\text{-}B\text{-}count\text{-}upto\ n = 0$

$\langle proof \rangle$

How Ψ s 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:*

$run\text{-}B\text{-}count\text{-}upto\ n = (\sum i < n+1. \Psi\ s\ i\ (run\text{-}B\text{-}count\text{-}upto\ n) \otimes_s ket\ i)$

$\langle proof \rangle$

lemma *run-B-count-split:*

$run\text{-}B\text{-}count = (\sum i < d+1. \Psi\ s\ i\ run\text{-}B\text{-}count \otimes_s ket\ i)$

$\langle proof \rangle$

lemma *run-B-count-projection:*

$Proj\text{-}ket\text{-}set \{.. < n+1\} *_V (run\text{-}B\text{-}count\text{-}upto\ n) = (run\text{-}B\text{-}count\text{-}upto\ n)$

$\langle proof \rangle$

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 $\langle Pfind' = (norm (Snd (id-cblinfun - selfbutter (ket empty)) *_V run-B))^2 \rangle$

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**

$\Phi s\ n = run-pure-adv\ n\ (\lambda i. UA\ i)\ (\lambda -. not-S-embed\ S)\ init\ X\ Y\ H$

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 $\Psi s-run-B-count-upto-eq-\Phi s$:

assumes $i < d+1$

shows $\Psi s\ 0\ (run-B-count-upto\ i) = \Phi s\ i$

$\langle proof \rangle$

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

lemma $\Psi s-run-B-upto-eq-\Phi s$:

assumes $i \leq d$

shows $\Psi s\ empty\ (run-B-upto\ i) = \Phi s\ i$

$\langle proof \rangle$

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

definition $P-nonterm = (norm\ run-B-count)^2 - (norm\ run-B)^2$

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

lemma *pure-o2h*: $\langle (norm ((run-A \otimes_s ket\ empty) - run-B))^2 \leq (d+1) * Pfind' + d * P-nonterm \rangle$
 $\langle proof \rangle$

lemma *pure-o2h-sqrt*: $\langle norm ((run-A \otimes_s ket\ empty) - run-B) \leq sqrt ((d+1) * Pfind' + d * P-nonterm) \rangle$
 $\langle proof \rangle$

lemma *error-term-pos*:
 $(d+1) * Pfind' + d * P-nonterm \geq 0$
 $\langle proof \rangle$

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 $a \geq 0 \ b \geq 0 \ c \geq 0 \ |sqrt\ a - sqrt\ b| \leq sqrt\ c$
shows $a + b \leq c + 2 * sqrt\ (a * b)$
 $\langle proof \rangle$

lemma *two-ab-ineq*:
assumes $a \geq 0 \ b \geq 0$
shows $2 * sqrt\ (a * b) \leq a + b$
 $\langle proof \rangle$

lemma *sqrt-estimate-real*:
assumes *fin-M*: *finite M*
and *pos-t*: $\forall x \in M. t\ x \geq (0::real)$
and *pos-u*: $\forall x \in M. u\ x \geq (0::real)$

```

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:  $\forall x \in M. | \text{sqrt} \ (t \ x) - \text{sqrt} \ (u \ x) | \leq \text{sqrt} \ (v \ x)$ 
shows  $| \text{sqrt} \ (\sum x \in M. a \ x * t \ x) - \text{sqrt} \ (\sum x \in M. a \ x * u \ x) | \leq \text{sqrt} \ (\sum x \in M. a \ x * v \ x)$ 
 $\langle proof \rangle$ 

```

```

end
theory Limit-Process

```

```

imports Run-Adversary

```

```

begin

```

```

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

```

7 Limit Processes

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

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

```

lift-definition kf-elems ::
  ('a::chilbert-space, 'b::chilbert-space, unit) kraus-family  $\Rightarrow$  ('a, 'b, unit) kraus-family set is
   $\lambda E. (\lambda x. \{x\}) \text{ ' } E$ 
 $\langle proof \rangle$ 

```

```

lemma kf-elems-Rep-kraus-family:
  kf-elems  $\mathfrak{E} = (\lambda x. \text{Abs-kraus-family} \ \{x\}) \text{ ' } \text{Rep-kraus-family} \ \mathfrak{E}$ 
 $\langle proof \rangle$ 

```

```

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

```

```

lemma kf-bound-of-elems:
  assumes  $F \in \text{kf-elems} \ E$ 
  shows kf-bound F  $\leq$  kf-bound E
 $\langle proof \rangle$ 

```

```

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

```

$\langle \text{proof} \rangle$

lemma *inj-on-kf-singleton*:

inj-on $(\lambda x. \text{Abs-kraus-family } \{x\}) (\text{Rep-kraus-family } \mathfrak{E})$
 $\langle \text{proof} \rangle$

lemma *kf-apply-singleton*:

fixes $E :: \langle 'a::\text{chilbert-space} \Rightarrow_{CL} 'b::\text{chilbert-space} \times 'x \rangle$
assumes $\langle \text{fst } E \neq 0 \rangle$
shows $\text{kf-apply } (\text{Abs-kraus-family } \{E\}) \varrho = \text{sandwich-tc } (\text{fst } E) \varrho$
 $\langle \text{proof} \rangle$

lemma *kf-apply-summable-on-kf-elems*:

fixes $\mathfrak{E} :: (\text{'a}::\text{chilbert-space}, \text{'b}::\text{chilbert-space}, \text{unit}) \text{ kraus-family}$
shows $(\lambda \mathfrak{F}. \text{kf-apply } \mathfrak{F} \varrho) \text{ summable-on } (\text{kf-elems } \mathfrak{E})$
 $\langle \text{proof} \rangle$

lemma *kf-apply-has-sum-kf-elems*:

fixes $\mathfrak{E} :: (\text{'a}::\text{chilbert-space}, \text{'b}::\text{chilbert-space}, \text{unit}) \text{ kraus-family}$
shows $((\lambda \mathfrak{F}. \text{kf-apply } \mathfrak{F} \varrho) \text{ has-sum } (\text{kf-apply } \mathfrak{E} \varrho)) (\text{kf-elems } \mathfrak{E})$
 $\langle \text{proof} \rangle$

lemma *kf-apply-abs-summable-on-kf-elems*:

fixes $\mathfrak{E} :: (\text{'a}::\text{chilbert-space}, \text{'b}::\text{chilbert-space}, \text{unit}) \text{ kraus-family}$
shows $(\lambda \mathfrak{F}. \text{kf-apply } \mathfrak{F} \varrho) \text{ abs-summable-on } (\text{kf-elems } \mathfrak{E})$
 $\langle \text{proof} \rangle$

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

lemma *finite-kraus-subadv-I*:

assumes $f \in \text{finite-kraus-subadv } \mathfrak{E} \ n \ i < n+1$
shows $f \ i \in \text{kf-elems } (\mathfrak{E} \ i)$
 $\langle \text{proof} \rangle$

lemma *finite-kraus-subadv-rewrite*:

$\text{finite-kraus-subadv } \mathfrak{E} (\text{Suc } n) =$
 $(\lambda(x,f). \text{fun-upd } f (\text{Suc } n) \ x) \text{ ' } (\text{kf-elems } (\mathfrak{E} (\text{Suc } n)) \times \text{finite-kraus-subadv } \mathfrak{E} \ n)$
 $\langle \text{proof} \rangle$

lemma *finite-kraus-subadv-rewrite-inj*:

inj-on $(\lambda(x, f). f(\text{Suc } n := x)) (\text{kf-elems } (\mathfrak{E} (\text{Suc } n)) \times \text{finite-kraus-subadv } \mathfrak{E} \ n)$
 $\langle \text{proof} \rangle$

lemma *norm-kf-apply-singleton-trace-tc*:

assumes $0 \leq \varrho$ **and** $\langle \text{fst } x \neq 0 \rangle$

shows $\text{norm } (\text{kf-apply } (\text{Abs-kraus-family } \{x\}) \varrho) = \text{trace-tc } (\text{sandwich-tc } (\text{fst } x) \varrho)$
 $\langle \text{proof} \rangle$

lemma *infsum-norm-kf-apply-step*:

assumes ϱn -summable: ϱn summable-on finite-kraus-subadv $\mathfrak{E} n$

and pos: $\bigwedge x. x \in \text{finite-kraus-subadv } \mathfrak{E} n \implies 0 \leq \varrho n x$

shows $(\lambda x. \sum_{y \in \text{finite-kraus-subadv } \mathfrak{E} n} \text{norm } (\text{kf-apply } x (\varrho n y)))$
 $\text{abs-summable-on kf-elems } (\mathfrak{E} (\text{Suc } n))$
 $\langle \text{proof} \rangle$

Run of adversary is summable on sub-adversaries.

lemma *run-mixed-adv-greater-indifferent*:

assumes $m > n$

shows $\text{run-mixed-adv } n (f(m := x)) \text{ UB init } X Y H = \text{run-mixed-adv } n f \text{ UB init } X Y H$
 $\langle \text{proof} \rangle$

lemma *run-mixed-adv-Suc-indifferent*:

$\text{run-mixed-adv } n (f(\text{Suc } n := x)) \text{ UB init } X Y H = \text{run-mixed-adv } n f \text{ UB init } X Y H$
 $\langle \text{proof} \rangle$

lemma *run-mixed-adv-abs-summable*:

fixes $\mathfrak{E} :: 'a \text{ kraus-adv}$

shows $(\lambda f. \text{run-mixed-adv } n f \text{ UB init } X Y H) \text{ abs-summable-on } (\text{finite-kraus-subadv } \mathfrak{E} n)$
 $\langle \text{proof} \rangle$

lemma *run-mixed-adv-summable*:

fixes $\mathfrak{E} :: 'a \text{ kraus-adv}$

shows $(\lambda f. \text{run-mixed-adv } n f \text{ UB init } X Y H) \text{ summable-on } (\text{finite-kraus-subadv } \mathfrak{E} n)$
 $\langle \text{proof} \rangle$

lemma *run-mixed-adv-has-sum*:

fixes $\mathfrak{E} :: 'a \text{ kraus-adv}$

shows $((\lambda f. \text{run-mixed-adv } n f \text{ UB init } X Y H) \text{ has-sum run-mixed-adv } n \mathfrak{E} \text{ UB init } X Y H)$
 $(\text{finite-kraus-subadv } \mathfrak{E} n)$
 $\langle \text{proof} \rangle$

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

context *o2h-setting*

begin

lemma *run-mixed-A-has-sum*:

$((\lambda f. \text{run-mixed-A } f H) \text{ has-sum run-mixed-A kraus-A } H) (\text{finite-kraus-subadv kraus-A } d)$
 $\langle \text{proof} \rangle$

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

((λf . *run-mixed-adv* d f (U 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)
 $\langle$ *proof* $\rangle$

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)
 $\langle$ *proof* $\rangle$

lemma *kf-elems-kf-Fst*:

kf-elems (*kf-Fst* $\mathfrak{E}$) = (λf . *kf-Fst* f) ‘ *kf-elems* $\mathfrak{E}$
 $\langle$ *proof* $\rangle$

lemma *finite-kraus-subadv-Fst-invert*:

finite-kraus-subadv (λm . (*kf-Fst* :: $\Rightarrow$ (($'a \times 'c$) *ell2*, -, -) *kraus-family*) ($\mathfrak{E}$ m)) n =
 (λf . $\lambda i \in \{0..<n+1\}$. *kf-Fst* (f i)) ‘ (*finite-kraus-subadv* $\mathfrak{E}$ n)
 $\langle$ *proof* $\rangle$

lemma *inj-kf-Fst*: $\langle E = F \rangle$ **if** $\langle$ *kf-Fst* E = *kf-Fst* F $\rangle$

$\langle$ *proof* $\rangle$

lemma *inj-on-kf-Fst*:

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

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
 $\langle$ *proof* $\rangle$

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* $?x$) $?A$)
 $\langle$ *proof* $\rangle$

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

```

  (( $\lambda f$ . 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  $f :: 'a \Rightarrow 'b \Rightarrow 'c :: \{comm-monoid-add, topological-space, topological-comm-monoid-add\}$ 
  assumes  $\bigwedge val. (f\ val\ has-sum\ g\ val)\ A\ finite\ S$ 
  shows  $((\lambda x. (\sum val \in S. f\ val\ x))\ has-sum\ (\sum val \in S. g\ val))\ A$ 
  <proof>

```

lemma *fin-subadv-fin-Rep-kraus-family*:

```

  assumes  $F \in finite-kraus-subadv\ E\ n\ i < n+1\ n < d+1$ 
  shows finite (Rep-kraus-family ( $F\ i$ ))
  <proof>

```

lemma *fin-subadv-bound-leq-id*:

```

  assumes  $F \in finite-kraus-subadv\ E\ d$ 
  assumes  $i < d+1$ 
  assumes E-norm-id:  $\bigwedge i. i < d+1 \implies kf-bound\ (E\ i) \leq id-cblinfun$ 
  shows  $kf-bound\ (F\ i) \leq id-cblinfun$ 
  <proof>

```

lemma *fin-subadv-nonzero*:

```

  assumes  $F \in finite-kraus-subadv\ E\ n\ i < n+1\ n < d+1$ 
  shows Rep-kraus-family ( $F\ i$ )  $\neq \{\}$ 
  <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* ::

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

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

Some auxiliary lemmas on injectivity, Fst and finiteness.

lemma *Rep-kf-id*:

$Rep-kraus-family\ kf-id = \{(id-cblinfun :: 'a \Rightarrow_{CL} 'a :: \{chilbert-space, not-singleton\}, ())\}$
 $\langle proof \rangle$

lemma *fst-Rep-kf-Fst*:

fixes $\mathfrak{E} :: ('a\ ell2, 'b\ ell2, unit) \text{ kraus-family}$
shows $fst\ '\ (Rep-kraus-family\ (kf-Fst\ \mathfrak{E})) = Fst\ '\ (fst\ '\ (Rep-kraus-family\ \mathfrak{E}))$
 $\langle proof \rangle$

lemma *inj-on-Fst*:

shows $inj-on\ Fst\ A$
 $\langle proof \rangle$

lemma *finite-kf-Fst*:

fixes $\mathfrak{E} :: ('mem\ ell2, 'mem\ ell2, unit) \text{ kraus-family}$
assumes $finite\ (Rep-kraus-family\ \mathfrak{E})$
shows $finite\ (Rep-kraus-family\ (kf-Fst\ \mathfrak{E}))$
 $\langle proof \rangle$

lemma *finite-kf-id*:

$finite\ (Rep-kraus-family\ kf-id)$
 $\langle proof \rangle$

lemma *inj-on-fst-Rep-kraus-family*:

fixes $\mathfrak{E} :: ('a\ ell2, 'b\ ell2, unit) \text{ kraus-family}$
shows $inj-on\ fst\ (Rep-kraus-family\ \mathfrak{E})$
 $\langle proof \rangle$

lemma *comp-kraus-maps-set-finite*:
assumes $\bigwedge i. i < n+1 \implies \text{finite } (\text{Rep-kraus-family } (\mathfrak{E} \ i))$
shows $\text{finite } (\text{purify-comp-kraus } n \ \mathfrak{E})$
 $\langle \text{proof} \rangle$

Showing conditions of Kraus maps.

lemma *norm-square-in-kraus-map*:
fixes $\mathfrak{E} :: ('a \ \text{ell2}, 'a \ \text{ell2}, \text{unit}) \ \text{kraus-family}$
assumes $\text{kf-bound } \mathfrak{E} \leq \text{id-cblinfun}$
assumes $U \in \text{fst } ' \text{Rep-kraus-family } \mathfrak{E}$
shows $U * o_{CL} \ U \leq \text{id-cblinfun}$
 $\langle \text{proof} \rangle$

lemma *norm-in-kraus-map*:
fixes $\mathfrak{E} :: ('a \ \text{ell2}, 'a \ \text{ell2}, \text{unit}) \ \text{kraus-family}$
assumes $\text{kf-bound } \mathfrak{E} \leq \text{id-cblinfun}$
assumes $U \in \text{fst } ' \text{Rep-kraus-family } \mathfrak{E}$
shows $\text{norm } U \leq 1$
 $\langle \text{proof} \rangle$

lemma *purify-comp-kraus-in-kraus-family*:
assumes $UA \in \text{purify-comp-kraus } n \ \mathfrak{E} \ j < n+1$
shows $UA \ j \in \text{fst } ' \text{Rep-kraus-family } (\mathfrak{E} \ j)$
 $\langle \text{proof} \rangle$

lemma *norm-in-purify-comp-kraus*:
fixes $\mathfrak{E} :: \text{nat} \Rightarrow ('a \ \text{ell2}, 'a \ \text{ell2}, \text{unit}) \ \text{kraus-family}$
assumes $\bigwedge i. i < n+1 \implies \text{kf-bound } (\mathfrak{E} \ i) \leq \text{id-cblinfun}$
assumes $UA \in \text{purify-comp-kraus } n \ \mathfrak{E}$
shows $\bigwedge i. i < n+1 \implies \text{norm } (UA \ i) \leq 1$
 $\langle \text{proof} \rangle$

lemma *run-pure-adv-tc-over*:
assumes $m > n$
shows $\text{run-pure-adv-tc } n \ (UA(m := x)) \ UB \ \text{init}' \ X' \ Y' \ H = \text{run-pure-adv-tc } n \ UA \ UB \ \text{init}' \ X' \ Y' \ H$
 $\langle \text{proof} \rangle$

lemma *run-pure-adv-tc-Fst-over*:
assumes $m > n$
shows $\text{run-pure-adv-tc } n \ (Fst \ o \ UA(m := x)) \ UB \ \text{init}' \ X' \ Y' \ H = \text{run-pure-adv-tc } n \ (Fst \ o \ UA) \ UB \ \text{init}' \ X' \ Y' \ H$
 $\langle \text{proof} \rangle$

Purifications of the adversarial runs.

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

shows *run-mixed-adv* n $\mathfrak{E}$ *UB init'* X' Y' $H =$
 $(\sum UAs \in \text{purify-comp-kraus } n \ \mathfrak{E}. \text{run-pure-adv-tc } n \ UAs \ \text{UB init}' \ X' \ Y' \ H)$
 $\langle \text{proof} \rangle$

lemma *purification-run-mixed-A*:
assumes $\bigwedge i. i < d+1 \implies \text{finite } (\text{Rep-kraus-family } (\mathfrak{E} \ i))$
assumes $\bigwedge i. i < d+1 \implies \text{fst } ' \text{Rep-kraus-family } (\mathfrak{E} \ i) \neq \{\}$
shows *run-mixed-A* $\mathfrak{E} \ H = (\sum UAs \in \text{purify-comp-kraus } d \ \mathfrak{E}. \text{run-pure-A-tc } UAs \ H)$
 $\langle \text{proof} \rangle$

lemma *purification-run-mixed-B*:
assumes $\bigwedge i. i < d+1 \implies \text{finite } (\text{Rep-kraus-family } (\mathfrak{E} \ i))$
assumes $\bigwedge i. i < d+1 \implies \text{fst } ' \text{Rep-kraus-family } (\mathfrak{E} \ i) \neq \{\}$
shows *run-mixed-B* $\mathfrak{E} \ H \ S = (\sum UAs \in \text{purify-comp-kraus } d \ \mathfrak{E}. \text{run-pure-B-tc } UAs \ H \ S)$
 $\langle \text{proof} \rangle$

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

lemma *purification-run-mixed-B-count*:
assumes $\bigwedge i. i < d+1 \implies \text{finite } (\text{Rep-kraus-family } (\mathfrak{E} \ i))$
assumes $\bigwedge i. i < d+1 \implies \text{fst } ' \text{Rep-kraus-family } (\mathfrak{E} \ i) \neq \{\}$
shows *run-mixed-B-count* $\mathfrak{E} \ H \ S = (\sum UAs \in \text{purify-comp-kraus } d \ \mathfrak{E}. \text{run-pure-B-count-tc}$
 $UAs \ H \ S)$
 $\langle \text{proof} \rangle$

Purification of *kf-Fst*

lemma *purification-kf-Fst*:
assumes $\bigwedge i. i < n+1 \implies \text{fst } ' \text{Rep-kraus-family } (F \ i) \neq \{\}$
assumes $x \in \text{purify-comp-kraus } n \ (\lambda n. \text{kf-Fst } (F \ n)::('a \times 'c) \ \text{ell2}, ('b \times 'c) \ \text{ell2}, \text{unit})$
kraus-family
shows $\exists UA. x = (\lambda a. \text{if } a < n+1 \text{ then } (\text{Fst } (UA \ a)::('a \times 'c) \ \text{ell2}) \Rightarrow_{CL} ('b \times 'c) \ \text{ell2}) \text{ else undefined})$
 $\langle \text{proof} \rangle$

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* $\text{TYPE}('x)$ $\text{TYPE}('y::\text{group-add})$ $\text{TYPE}('mem)$ $\text{TYPE}('l)$ +
 — We fix the distributions on H and S. (They might be correlated.) So far, we assume that they are discrete distributions and model them in the following way:

```

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

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

```

```

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

```

```

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

```

```

begin

```

```

lemma norm-P:
  norm P  $\leq 1$ 
  <proof>

```

lemma *distr-pos'*:
assumes $(H, S, z) \in \text{carrier}$ **shows** $\text{distr } (H, S, z) \geq 0$
 $\langle \text{proof} \rangle$

lemma *norm-Fst-P*:
 $\text{norm } (\text{Fst } P :: ('mem \times 'a) \text{ update}) \leq 1$
 $\langle \text{proof} \rangle$

9.1 Final states

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

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

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

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

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

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

Positivity

lemma *qlleft-pure-pos*: $0 \leq \text{qlleft-pure } UA$
 $\langle \text{proof} \rangle$

lemma *qlleft-pos*: $0 \leq \text{qlleft } F$
 $\langle \text{proof} \rangle$

lemma *qright-pure-pos*: $0 \leq \text{qright-pure } UA$
 $\langle \text{proof} \rangle$

lemma *qright-pos*: $0 \leq \text{qright } F$
 $\langle \text{proof} \rangle$

lemma *qcount-pure-pos*: $0 \leq \text{qcount-pure } UA$
 $\langle \text{proof} \rangle$

lemma *qcount-pos*: $0 \leq \text{qcount } F$
 $\langle \text{proof} \rangle$

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

lemma *norm-qleft:*

$\text{norm } (\text{qleft } E) \leq 1$

$\langle \text{proof} \rangle$

lemma *norm-qright:*

$\text{norm } (\text{qright } E) \leq 1$

$\langle \text{proof} \rangle$

lemma *norm-qcount:*

$\text{norm } (\text{qcount } E) \leq 1$

$\langle \text{proof} \rangle$

lemma *trace-preserving-norm-qright:*

assumes $\bigwedge i. i < d+1 \implies \text{km-trace-preserving } (\text{kf-apply } (\text{kf-Fst } (E \ i))::('mem \times 'l) \text{ ell2}, ('mem \times 'l) \text{ ell2}, \text{unit}) \text{ kraus-family})$

shows $\text{norm } (\text{qright } E) = 1$

$\langle \text{proof} \rangle$

lemma *trace-preserving-norm-qcount:*

assumes $\bigwedge i. i < d+1 \implies \text{km-trace-preserving } (\text{kf-apply } (\text{kf-Fst } (E \ i))::('mem \times \text{nat}) \text{ ell2}, ('mem \times \text{nat}) \text{ ell2}, \text{unit}) \text{ kraus-family})$

shows $\text{norm } (\text{qcount } E) = 1$

$\langle \text{proof} \rangle$

Summability and Infsums

lemma *from-trace-class-qright-pure:*

$\text{from-trace-class } (\text{qright-pure } UA) = (\sum (H,S,z) \in \text{carrier}. \text{distr } (H,S,z) *_C \text{run-pure-B-update } UA \ H \ S)$

$\langle \text{proof} \rangle$

lemma *has-sum-scaleC-tc:*

fixes $x :: ('a::\text{chilbert-space}, 'a) \text{ trace-class}$

assumes $(f \text{ has-sum } x) \ A$

shows $((\lambda y. c *_C f \ y) \text{ has-sum } c *_C x) \ A$

$\langle \text{proof} \rangle$

lemma *qleft-has-sum:*

$(\text{qleft has-sum } \text{qleft } E) \ (\text{finite-kraus-subadv } E \ d)$

$\langle \text{proof} \rangle$

lemma *qright-has-sum:*

$((\lambda f. \text{qright } f) \text{ has-sum } \text{qright } E) \ (\text{finite-kraus-subadv } E \ d)$

<proof>

lemma *qright-abs-summable:*

qright abs-summable-on (finite-kraus-subadv E d)

<proof>

lemma *qcount-has-sum:*

(qcount has-sum qcount E) (finite-kraus-subadv E d)

<proof>

Connection pure and mixed states

lemma *qleft-pure-mixed:*

assumes $\bigwedge i. i < d + 1 \implies \text{finite } (\text{Rep-kraus-family } (F\ i))$

$\bigwedge i. i < d + 1 \implies \text{fst } \text{Rep-kraus-family } (F\ i) \neq \{\}$

shows $q\text{left } F = (\sum UAs \in \text{purify-comp-kraus } d\ F. q\text{left-pure } UAs)$

<proof>

lemma *qright-pure-mixed:*

assumes $\bigwedge i. i < d + 1 \implies \text{finite } (\text{Rep-kraus-family } (F\ i))$

$\bigwedge i. i < d + 1 \implies \text{fst } \text{Rep-kraus-family } (F\ i) \neq \{\}$

shows $q\text{right } F = (\sum UAs \in \text{purify-comp-kraus } d\ F. q\text{right-pure } UAs)$

<proof>

lemma *qcount-pure-mixed:*

assumes $\bigwedge i. i < d + 1 \implies \text{finite } (\text{Rep-kraus-family } (F\ i))$

$\bigwedge i. i < d + 1 \implies \text{fst } \text{Rep-kraus-family } (F\ i) \neq \{\}$

shows $q\text{count } F = (\sum UAs \in \text{purify-comp-kraus } d\ F. q\text{count-pure } UAs)$

<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* :: $('mem \times 'l)$ 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*:

$\text{Proj } (\text{end-measure} *_S \top) = \text{end-measure}$
 $\langle \text{proof} \rangle$

lemma *norm-end-measure*:

$\text{norm } (\text{end-measure}) \leq 1$
 $\langle \text{proof} \rangle$

lemma *end-measure-butterfly*:

$\text{sandwich } \text{end-measure } (\text{selfbutter } \Psi) = \text{selfbutter } (\text{end-measure} *_V \Psi)$
 $\langle \text{proof} \rangle$

lemma *trace-end-measure*:

$\text{trace } (\text{end-measure } o_{CL} \text{ selfbutter } \Psi) = (\text{complex-of-real } (\text{norm } (\text{end-measure} *_V \Psi)))^2$
 $\langle \text{proof} \rangle$

lemma *trace-endmeasure-pos*:

assumes $\varrho \geq 0$
shows $\text{trace-tc } (\text{compose-tcr } \text{end-measure } \varrho) \geq 0$
 $\langle \text{proof} \rangle$

lemma *trace-class-end-measure*:

assumes *trace-class* a
shows $\text{trace-class } (\text{end-measure } o_{CL} a)$
 $\langle \text{proof} \rangle$

lemma *abs-op-id-cblinfun [simp]*:

$\text{abs-op } \text{id-cblinfun} = \text{id-cblinfun}$
 $\langle \text{proof} \rangle$

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

definition *empty-tc* :: 'l tc-op **where**

$\text{empty-tc} = \text{Abs-trace-class } (\text{selfbutter } (\text{ket } \text{empty}))$

lemma *norm-empty-tc*:

$\text{norm } \text{empty-tc} = 1$
 $\langle \text{proof} \rangle$

lemma *empty-tc-pos*: $0 \leq \text{empty-tc}$

$\langle \text{proof} \rangle$

10.1 Projective measurement PM

The projective measurement PM Q at the end

definition *PM-update* :: ('mem $\times$ 'l) update $\Rightarrow$ ('mem $\times$ 'l) update $\Rightarrow$ complex **where**

$\text{PM-update } Q \varrho = \text{trace } (\text{sandwich } Q \varrho)$

lemma *PM-update-linear*:

assumes *trace-class* ϱ *trace-class* ψ

shows $PM\text{-}update\ Q\ (\varrho + \psi) = PM\text{-}update\ Q\ \varrho + PM\text{-}update\ Q\ \psi$

<proof>

definition $PM :: ('mem \times 'l)\ update \Rightarrow ('mem \times 'l)\ tc\text{-}op \Rightarrow complex$ **where**

$PM\ Q = PM\text{-}update\ Q\ o\ from\text{-}trace\text{-}class$

lemma *PM-altdef*:

$PM\ Q\ \varrho = trace\text{-}tc\ (sandwich\text{-}tc\ Q\ \varrho)$

<proof>

lemma *PM-linear*:

$PM\ Q\ (\varrho + \psi) = PM\ Q\ \varrho + PM\ Q\ \psi$

<proof>

lemma *PM-sum-distr*:

$PM\ Q\ (sum\ f\ S) = sum\ (PM\ Q\ o\ f)\ S$

<proof>

lemma *PM-scale*:

$PM\ Q\ (a *_{\mathcal{C}} \varrho) = a * PM\ Q\ \varrho$

<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 $Re\ (PM\ Q\ \varrho) = PM\ Q\ \varrho$

<proof>

lemma *PM-pos*:

assumes $\varrho \geq 0$

shows $PM\ Q\ \varrho \geq 0$

<proof>

lemma *Re-PM-pos*:

assumes $\varrho \geq 0$

shows $Re\ (PM\ Q\ \varrho) \geq 0$

<proof>

lemma *norm-PM*:

assumes $norm\ \varrho \leq 1\ norm\ Q \leq 1$

shows $norm\ (PM\ Q\ \varrho) \leq 1$

<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** $Pleft' Q = Re (PM Q (tc\text{-}tensor (\rho left E) empty\text{-}tc))$

definition *Pleft* **where** $Pleft Q = Re (trace\text{-}tc (sandwich\text{-}tc Q (\rho left E)))$

lemma *trace-tensor-tc*:
 $trace\text{-}tc (tc\text{-}tensor a b) = trace\text{-}tc a * trace\text{-}tc b$
 ⟨proof⟩

lemma *Pleft-Pleft'*:
assumes $sandwich\text{-}tc A empty\text{-}tc = tc\text{-}selfbutter (ket empty)$
shows $Pleft Q = Pleft' (Q \otimes_o A)$
 ⟨proof⟩

lemma *Pleft-Pleft'-empty*:
 $Pleft Q = Pleft' (Q \otimes_o selfbutter (ket empty))$
 ⟨proof⟩

lemma *Pleft-Pleft'-id*:
 $Pleft Q = Pleft' (Q \otimes_o id\text{-}cblinfun)$
 ⟨proof⟩

lemma *Pleft-Pleft'-case5*:
assumes *is-Proj* Q
shows $Pleft Q = Pleft' (Q \otimes_o selfbutter (ket empty) + end\text{-}measure)$
 ⟨proof⟩

definition *Pright* **where** $Pright Q = Re (PM Q (\rho right E))$

lemma *Re-PM-left-has-sum*:
 $((\lambda F. Re (PM Q (tc\text{-}tensor (\rho left F) empty\text{-}tc))) has\text{-}sum Pleft' Q)$
 $(finite\text{-}kraus\text{-}subadv E d)$
 ⟨proof⟩

lemma *Re-PM-right-has-sum*:

$((\lambda F. \text{Re } (PM \ Q \ (\varrho\text{right } F))) \text{ has-sum } Pright \ Q) \ (\text{finite-kraus-subadv } E \ d)$
 $\langle \text{proof} \rangle$

10.3 Pfind

The definition of the find event

definition *Pfind-update* ::

$(nat \Rightarrow 'mem \ \text{update}) \Rightarrow ('x \Rightarrow 'y) \Rightarrow ('x \Rightarrow bool) \Rightarrow \text{complex}$ **where**
 $Pfind\text{-update } UA \ H \ S = \text{trace } (\text{end-measure } o_{CL} \ (\text{run-pure-B-update } UA \ H \ S))$

definition *Pfind-pure* :: $(nat \Rightarrow 'mem \ \text{update}) \Rightarrow \text{complex}$ **where**

$Pfind\text{-pure } UA = \text{trace-tc } (\text{compose-tcr } \text{end-measure } (\varrho\text{right-pure } UA))$

definition *Pfind* :: $'mem \ \text{kraus-adv} \Rightarrow \text{complex}$ **where**

$Pfind \ F = \text{trace-tc } (\text{compose-tcr } \text{end-measure } (\varrho\text{right } F))$

lemma *Pfind-altdef*:

$Pfind \ E = Pright \ \text{end-measure}$
 $\langle \text{proof} \rangle$

lemma *Pfind-Pright*:

$Re \ (Pfind \ E) = Pright \ \text{end-measure}$
 $\langle \text{proof} \rangle$

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

lemma *Re-Pfind-update-altdef*:

assumes $\bigwedge i. i < d + 1 \implies \text{norm } (UA \ i) \leq 1$
shows $Re \ (Pfind\text{-update } UA \ H \ S) = \text{pure-o2h}.Pfind' \ X \ Y \ d \ \text{init flip empty } H \ S \ UA$
 $\langle \text{proof} \rangle$

lemma *Pfind-pure-update*:

$Pfind\text{-pure } UA = (\sum (H,S,z) \in \text{carrier}. \text{distr } (H,S,z) * Pfind\text{-update } UA \ H \ S)$
 $\langle \text{proof} \rangle$

lemma *Pfind-pure-mixed*:

assumes $\bigwedge i. i < d + 1 \implies \text{finite } (\text{Rep-kraus-family } (F \ i))$
 $\bigwedge i. i < d + 1 \implies \text{fst } ' \text{Rep-kraus-family } (F \ i) \neq \{\}$
shows $Pfind \ F = (\sum UA \in \text{purify-comp-kraus } d \ F. Pfind\text{-pure } UA)$
 $\langle \text{proof} \rangle$

Pfind positivity

lemma *Pfind-pure-pos*:

$Pfind\text{-pure } UA \geq 0$
 $\langle \text{proof} \rangle$

lemma *Pfind-pos:*

$Pfind\ F \geq 0$ *<proof>*

Pfind is already real

lemma *Re-Pfind-update:*

$Re\ (Pfind\text{-}update\ UA\ H\ S) = Pfind\text{-}update\ UA\ H\ S$
<proof>

lemma *Re-Pfind-pure:*

$Re\ (Pfind\text{-}pure\ UA) = Pfind\text{-}pure\ UA$
<proof>

lemma *Re-Pfind:*

$Re\ (Pfind\ F) = Pfind\ F$
<proof>

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

lemma *Pfind-abs-summable-on:*

$Pfind\ abs\text{-}summable\text{-}on\ (finite\text{-}kraus\text{-}subadv\ E\ d)$
<proof>

lemma *Pfind-summable-on:*

$Pfind\ summable\text{-}on\ (finite\text{-}kraus\text{-}subadv\ E\ d)$
<proof>

lemma *Pfind-has-sum:*

$((\lambda F. Pfind\ F)\ has\text{-}sum\ Pfind\ E)\ (finite\text{-}kraus\text{-}subadv\ E\ d)$
<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 $\Rightarrow$ 'y) $\Rightarrow$ ('x $\Rightarrow$ bool) $\Rightarrow$ (nat $\Rightarrow$ 'mem update) $\Rightarrow$ real* **where**

$P\text{-}nonterm\text{-}update\ H\ S\ UA =$
 $Re\ (trace\ (run\text{-}pure\text{-}B\text{-}count\text{-}update\ UA\ H\ S) - trace\ (run\text{-}pure\text{-}B\text{-}update\ UA\ H\ S))$

definition *P-nonterm-pure::(nat $\Rightarrow$ 'mem ell2 $\Rightarrow_{CL}$ 'mem ell2) $\Rightarrow$ real* **where**

$P\text{-}nonterm\text{-}pure\ UA = Re\ (trace\text{-}tc\ (\varrho count\text{-}pure\ UA) - trace\text{-}tc\ (\varrho right\text{-}pure\ UA))$

definition *P-nonterm :: 'mem kraus-adv $\Rightarrow$ real* **where**

$P\text{-}nonterm\ F = Re\ (trace\text{-}tc\ (\varrho count\ F) - trace\text{-}tc\ (\varrho right\ F))$

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

lemma *P-nonterm-update-altdef:*

assumes $\bigwedge i. i < d+1 \implies norm\ (UA\ i) \leq 1$

shows $P\text{-}nonterm\text{-}update\ H\ S\ UA = pure\text{-}o2h.P\text{-}nonterm\ X\ Y\ d\ init\ flip\ empty\ H\ S\ UA$

$\langle \text{proof} \rangle$

lemma *P-nonterm-pure-update:*

$P\text{-nonterm-pure } UA = (\sum (H,S,z) \in \text{carrier}. \text{distr } (H,S,z) * P\text{-nonterm-update } H S UA)$

$\langle \text{proof} \rangle$

lemma *P-nonterm-purification:*

assumes $\bigwedge i. i < d + 1 \implies \text{finite } (\text{Rep-kraus-family } (F i))$

$\bigwedge i. i < d + 1 \implies \text{Rep-kraus-family } (F i) \neq \{\}$

shows $P\text{-nonterm } F = (\sum UA \in \text{purify-comp-kraus } d F. P\text{-nonterm-pure } UA)$

$\langle \text{proof} \rangle$

Positive error term

lemma *error-term-update-pos:*

assumes $\bigwedge i. i < d + 1 \implies \text{norm } (UA i) \leq 1$

shows $0 \leq (d + 1) * \text{Re } (P\text{find-update } UA H S) + d * (P\text{-nonterm-update } H S UA)$

$\langle \text{proof} \rangle$

lemma *error-term-pure-pos:*

assumes $\bigwedge i. i < d + 1 \implies \text{norm } (UA i) \leq 1$

shows $0 \leq (d + 1) * \text{Re } (P\text{find-pure } UA) + d * (P\text{-nonterm-pure } UA)$

$\langle \text{proof} \rangle$

lemma *error-term-pos:*

assumes *finite:* $\bigwedge i. i < d + 1 \implies \text{finite } (\text{Rep-kraus-family } (F i))$

and *F-norm-id:* $\bigwedge i. i < d + 1 \implies \text{kf-bound } (F i) \leq \text{id-cblinfun}$

and *F-nonzero:* $\bigwedge i. i < d + 1 \implies \text{Rep-kraus-family } (F i) \neq \{\}$

shows $0 \leq (d + 1) * \text{Re } (P\text{find } F) + d * P\text{-nonterm } F$

$\langle \text{proof} \rangle$

has sum property

lemma *P-nonterm-has-sum:*

$((\lambda F. P\text{-nonterm } F) \text{ has-sum } P\text{-nonterm } E) (\text{finite-kraus-subadv } E d)$

$\langle \text{proof} \rangle$

11 Proof of Mixed O2H

We prove the mixed O2H in several steps.

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

lemma *estimate-Pfind-update-sqrt:*

fixes $UA H S$

assumes $\bigwedge i. i < d + 1 \implies \text{norm } (UA i) \leq 1$

and *norm-Q:* $\text{norm } Q \leq 1$

shows $|\text{sqrt } (\text{Re } (P\text{M-update } Q ((\text{run-pure-A-update } UA H) \otimes_o (\text{selfbutter } (\text{ket empty})))))) -$

$$\begin{aligned}
& \text{sqrt } (\text{Re } (\text{PM-update } Q \text{ (run-pure-B-update } UA \ H \ S)))| \\
& \leq \text{sqrt } ((d+1) * \text{Re } (\text{Pfind-update } UA \ H \ S) + d * \text{P-nonterm-update } H \ S \ UA) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *estimate-Pfind-tc-sqrt*:

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

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 $\bigwedge i. i < d+1 \implies \text{norm } (UA \ i) \leq 1 \ \text{norm } Q \leq 1$
shows $|\text{sqrt } (\text{Re } (\text{PM } Q \text{ (tc-tensor (qlift-pure } UA) \ \text{empty-tc}))) - \text{sqrt } (\text{Re } (\text{PM } Q \text{ (qright-pure } UA)))|$
 $\leq \text{sqrt } (\text{real } (d + 1) * \text{Re } (\text{Pfind-pure } UA) + d * \text{P-nonterm-pure } UA)$
 $\langle \text{proof} \rangle$

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

lemma *estimate-Pfind-finite-sqrt*:

assumes *finite*: $\bigwedge i. i < d+1 \implies \text{finite } (\text{Rep-kraus-family } (F \ i))$
and *F-norm-id*: $\bigwedge i. i < d+1 \implies \text{kf-bound } (F \ i) \leq \text{id-cblinfun}$
and *F-nonzero*: $\bigwedge i. i < d+1 \implies \text{Rep-kraus-family } (F \ i) \neq \{\}$
and *norm-Q*: $\text{norm } Q \leq 1$
shows $|\text{csqrt } (\text{PM } Q \text{ (tc-tensor (qlift } F) \ \text{empty-tc})) - \text{csqrt } (\text{PM } Q \text{ (qright } F))| \leq$
 $\text{csqrt } ((d+1) * \text{Pfind } F + d * \text{P-nonterm } F)$
 $\langle \text{proof} \rangle$

lemma *estimate-Pfind-finite-sqrt'*:

assumes *finite*: $\bigwedge i. i < d+1 \implies \text{finite } (\text{Rep-kraus-family } (F \ i))$
and *F-norm-id*: $\bigwedge i. i < d+1 \implies \text{kf-bound } (F \ i) \leq \text{id-cblinfun}$
and *F-nonzero*: $\bigwedge i. i < d+1 \implies \text{Rep-kraus-family } (F \ i) \neq \{\}$
and *norm-Q*: $\text{norm } Q \leq 1$
shows $|\text{sqrt } (\text{Re } (\text{PM } Q \text{ (tc-tensor (qlift } F) \ \text{empty-tc}))) - \text{sqrt } (\text{Re } (\text{PM } Q \text{ (qright } F)))| \leq$
 $\text{sqrt } ((d+1) * \text{Re } (\text{Pfind } F) + d * \text{P-nonterm } F)$
 $\langle \text{proof} \rangle$

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*:

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

$\langle proof \rangle$

lemma *scale-P-nonterm-has-sum*:

$((\lambda F. \text{real } d * P\text{-nonterm } F) \text{ has-sum real } d * P\text{-nonterm } E) (\text{finite-kraus-subadv } E \ d)$

$\langle proof \rangle$

lemma *estimate-Pfind-sqrt*:

assumes *norm-Q*: $\text{norm } Q \leq 1$

shows $|\text{sqrt } (P\text{left}' Q) - \text{sqrt } (P\text{right } Q)| \leq$

$\text{sqrt } ((d+1) * \text{Re } (P\text{find } E) + d * P\text{-nonterm } E)$

(**is** $?left \leq ?right$)

$\langle proof \rangle$

lemma *estimate-Pfind*:

assumes *norm-Q*: $\text{norm } Q \leq 1$

shows

$|P\text{left}' Q - P\text{right } Q| \leq 2 * \text{sqrt } ((d+1) * \text{Re } (P\text{find } E) + d * P\text{-nonterm } E)$

$\langle proof \rangle$

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* $\text{TYPE}('x)$ $\text{TYPE}('y::\text{group-add})$ $\text{TYPE}('mem)$ $\text{TYPE}('l)$ +

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

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

assumes *distr-pos*: $\forall (H, G, S, z) \in \text{carrier}. \text{distr } (H, G, S, z) \geq 0$

and *distr-sum-1*: $(\sum (H, G, S, z) \in \text{carrier}. \text{distr } (H, G, S, z)) = 1$

and *finite-carrier*: *finite carrier*

and H - G -same-upto- S :

$\bigwedge H \ G \ S \ z. (H, G, S, z) \in \text{carrier} \implies x \in - \text{Collect } S \implies H \ x = G \ x$

fixes E :: 'mem kraus-adv

assumes E -norm-id: $\bigwedge i. i < d+1 \implies \text{kf-bound } (E \ i) \leq \text{id-cblinfun}$

assumes E -nonzero: $\bigwedge i. i < d+1 \implies \text{Rep-kraus-family } (E \ i) \neq \{\}$

fixes P :: 'mem update

assumes $\text{is-Proj-}P$: $\text{is-Proj } P$

begin

lemma $\text{Fst-}E$ -nonzero:

$\bigwedge i. i < d+1 \implies \text{Rep-kraus-family } (\text{kf-Fst } (E \ i)) \neq \{\}$
 $\langle \text{proof} \rangle$

Some properties of the joint distribution.

lemma $\text{Uquery-}G$ - H -same-on-not- S -embed':

assumes $(H, G, S, z) \in \text{carrier}$

shows

$\text{Uquery } H \ o_{CL} \ \text{proj-classical-set } (- \ (\text{Collect } S)) \otimes_o \text{id-cblinfun} =$
 $\text{Uquery } G \ o_{CL} \ \text{proj-classical-set } (- \ (\text{Collect } S)) \otimes_o \text{id-cblinfun}$

$\langle \text{proof} \rangle$

lemma $\text{Uquery-}G$ - H -same-on-not- S -embed:

assumes $(H, G, S, z) \in \text{carrier}$

shows $((X; Y) \ (\text{Uquery } H) \ o_{CL} \ (\text{not-}S\text{-embed } S)) = ((X; Y) \ (\text{Uquery } G) \ o_{CL} \ (\text{not-}S\text{-embed } S))$

$\langle \text{proof} \rangle$

lemma $\text{Uquery-}G$ - H -same-on-not- S -embed-tensor:

assumes $(H, G, S, z) \in \text{carrier}$

shows $((X\text{-for-}B; Y\text{-for-}B) \ (\text{Uquery } H) \ o_{CL} \ \text{Fst } (\text{not-}S\text{-embed } S)) =$
 $((X\text{-for-}B; Y\text{-for-}B) \ (\text{Uquery } G) \ o_{CL} \ \text{Fst } (\text{not-}S\text{-embed } S))$

$\langle \text{proof} \rangle$

Instantiations of mixed o2h locale for H and G

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

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

lemma $\text{distr-}G$ -pos: $\forall (G, S, z) \in \text{carrier-}G. \text{distr-}G \ (G, S, z) \geq 0$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma $\text{finite-carrier-}G$: $\text{finite carrier-}G$

$\langle proof \rangle$

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

definition *distr-H* **where** $distr-H = (\lambda(H, S, (G, z)). distr (H, G, S, z))$

lemma *distr-H-pos*: $\forall (H, S, z) \in carrier-H. distr-H (H, S, z) \geq 0$

$\langle proof \rangle$

lemma *distr-H-sum-1*: $(\sum (H, S, z) \in carrier-H. distr-H (H, S, z)) = 1$

$\langle proof \rangle$

lemma *finite-carrier-H*: *finite carrier-H*

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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 $l \in has-bits-upto n$

shows $length\ l = d$

$\langle proof \rangle$

lemma *empty-not-flip*:

assumes $x \in list-to-l$ ‘ *has-bits-upto n n < d*

shows $empty \neq flip\ n\ x$

$\langle proof \rangle$

lemma *empty-not-flip'*:

assumes $x \neq flip\ n\ empty\ n < d$

shows $empty \neq flip\ n\ x$

$\langle proof \rangle$

lemma *Proj-ket-upto-Snd*:

$Proj-ket-upto\ A = Snd\ (proj-classical-set\ (list-to-l\ 'A))$

$\langle proof \rangle$

lemma *from-trace-class-tc-selfbutter*:

$from-trace-class\ (tc-selfbutter\ x) = selfbutter\ x$

$\langle proof \rangle$

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 $\varrho = run-pure-adv-tc\ n\ (\lambda m. if\ m < n + 1\ then\ Fst\ (UA\ m)\ else\ UB\ m)\ (US\ S)\ init-B$
 $X-for-B\ Y-for-B\ H$

shows $sandwich-tc\ (Proj-ket-upto\ (has-bits-upto\ n))\ \varrho = \varrho$

<proof>

lemma *run-mixed-adv-projection-finite*:

assumes $\bigwedge i. i < n + 1 \implies finite\ (Rep-kraus-family\ (kf-Fst\ (F\ i))::$

$((mem \times 'l)\ ell2, (mem \times 'l)\ ell2, unit)\ kraus-family))$

and $\bigwedge i. i < n + 1 \implies fst\ 'Rep-kraus-family\ (kf-Fst\ (F\ i))::$

$((mem \times 'l)\ ell2, (mem \times 'l)\ ell2, unit)\ kraus-family) \neq \{\}$

assumes $n < d + 1$

shows $sandwich-tc\ (Proj-ket-upto\ (has-bits-upto\ n))$

$(run-mixed-adv\ n\ (\lambda n. kf-Fst\ (F\ n))\ (US\ S)\ init-B\ X-for-B\ Y-for-B\ H) =$

$run-mixed-adv\ n\ (\lambda n. kf-Fst\ (F\ n))\ (US\ S)\ init-B\ X-for-B\ Y-for-B\ H$

<proof>

lemma *run-mixed-adv-projection*:

assumes $\bigwedge i. i < d + 1 \implies fst\ 'Rep-kraus-family\ (kf-Fst\ (F\ i))::$

$((mem \times 'l)\ ell2, (mem \times 'l)\ ell2, unit)\ kraus-family) \neq \{\}$

assumes $n < d + 1$

shows sandwich-tc (Proj-ket-upto (has-bits-upto n))
 (run-mixed-adv n ($\lambda n. \text{kf-Fst } (F \ n)$) (US S) init-B X-for-B Y-for-B H) =
 run-mixed-adv n ($\lambda n. \text{kf-Fst } (F \ n)$) (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)) $\circ_{CL}$ (X-for-B; Y-for-B) (Uquery G) =
 (X-for-B; Y-for-B) (Uquery G) $\circ_{CL}$ Snd (selfbutter (ket empty))
 <proof>

lemma run-mixed-adv-G-H-same:

assumes (H, G, S, z) $\in$ carrier n < d+1

shows sandwich-tc (Snd (selfbutter (ket empty)))
 (run-mixed-adv n ($\lambda n. \text{kf-Fst } (E \ n)$) (US S) init-B X-for-B Y-for-B H) =
 sandwich-tc (Snd (selfbutter (ket empty)))
 (run-mixed-adv n ($\lambda n. \text{kf-Fst } (E \ n)$) (US S) init-B X-for-B Y-for-B G)
 <proof>

lemma run-mixed-B-G-H-same:

assumes (H, G, S, z) $\in$ carrier

shows sandwich-tc (Q $\otimes_o$ selfbutter (ket empty)) (run-mixed-B E H S) =
 sandwich-tc (Q $\otimes_o$ selfbutter (ket empty)) (run-mixed-B E G S)
 <proof>

lemma pright-G-H-same:

sandwich-tc (Q $\otimes_o$ selfbutter (ket empty)) (mixed-H.pright E) =
 sandwich-tc (Q $\otimes_o$ selfbutter (ket empty)) (mixed-G.pright E)
 <proof>

lemma trace-compose-tcr-H-G-same:

trace-tc (compose-tcr (Snd (selfbutter (ket empty))) (mixed-H.pright E)) =
 trace-tc (compose-tcr (Snd (selfbutter (ket empty))) (mixed-G.pright E))
 <proof>

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

lemma Pright-G-H-same:

mixed-H.Pright (Q $\otimes_o$ selfbutter (ket empty)) = mixed-G.Pright (Q $\otimes_o$ selfbutter (ket empty))
 <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 $\text{norm } (\text{mixed-H.qright } E) = \text{norm } (\text{mixed-G.qright } E)$
shows $\text{mixed-H.Pfind } E = \text{mixed-G.Pfind } E$
 $\langle \text{proof} \rangle$

lemma *Pfind-G-H-same-nonterm*:
shows $(\text{mixed-H.Pfind } E - \text{mixed-G.Pfind } E) =$
 $(\text{norm } (\text{mixed-H.qright } E) - \text{norm } (\text{mixed-G.qright } E))$
 $\langle \text{proof} \rangle$

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

theorem *mixed-o2h-nonterm*:
shows
 $|\text{mixed-H.Pleft } P - \text{mixed-G.Pleft } P| \leq$
 $2 * \text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E) + d * \text{mixed-H.P-nonterm } E)$
 $+ 2 * \text{sqrt } ((d+1) * \text{Re } (\text{mixed-G.Pfind } E) + d * \text{mixed-G.P-nonterm } E)$
and
 $|\text{sqrt } (\text{mixed-H.Pleft } P) - \text{sqrt } (\text{mixed-G.Pleft } P)| \leq$
 $\text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E) + d * \text{mixed-H.P-nonterm } E)$
 $+ \text{sqrt } ((d+1) * \text{Re } (\text{mixed-G.Pfind } E) + d * \text{mixed-G.P-nonterm } E)$
 $\langle \text{proof} \rangle$

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

theorem *mixed-o2h-term*:
assumes $\bigwedge i. i < d+1 \implies \text{km-trace-preserving } (\text{kf-apply } (E \ i))$
shows
 $|\text{mixed-H.Pleft } P - \text{mixed-G.Pleft } P| \leq 4 * \text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E))$
and
 $|\text{sqrt } (\text{mixed-H.Pleft } P) - \text{sqrt } (\text{mixed-G.Pleft } P)| \leq 2 * \text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E))$
 $\langle \text{proof} \rangle$

Other formulations of the mixed o2h.

Theorem 1, definition of Pright (2)

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

lemma *norm-Proj-2*:
 $\text{norm } \text{Proj-2} \leq 1$
 $\langle \text{proof} \rangle$

theorem *mixed-o2h-nonterm-2*:
shows
 $|\text{mixed-H.Pleft } P - \text{mixed-H.Prigh } \text{Proj-2}| \leq$
 $2 * \text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E) + d * \text{mixed-H.P-nonterm } E)$
and
 $|\text{sqrt } (\text{mixed-H.Pleft } P) - \text{sqrt } (\text{mixed-H.Prigh } \text{Proj-2})| \leq$

$\text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E) + d * \text{mixed-H.P-nonterm } E)$
 $\langle \text{proof} \rangle$

theorem *mixed-o2h-term-2*:

assumes $\bigwedge i. i < d+1 \implies \text{km-trace-preserving } (\text{kf-apply } (E \ i))$

shows

$|\text{mixed-H.Pleft } P - \text{mixed-H.Pright Proj-2}| \leq$
 $2 * \text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E))$

and

$|\text{sqrt } (\text{mixed-H.Pleft } P) - \text{sqrt } (\text{mixed-H.Pright Proj-2})| \leq$
 $\text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E))$

$\langle \text{proof} \rangle$

Theorem 1, definition of Pright (3)

definition *Proj-3* :: $('mem \times 'l) \text{ ell2} \Rightarrow_{CL} ('mem \times 'l) \text{ ell2}$ **where**

$\text{Proj-3} = P \otimes_o \text{selfbutter } (\text{ket empty})$

lemma *is-Proj-3*:

is-Proj Proj-3

$\langle \text{proof} \rangle$

lemma *Proj-3-altdef*:

$\text{Proj-3} = \text{Proj } ((P \otimes_o \text{id-cblinfun}) *_S \top \sqcup (\text{id-cblinfun} \otimes_o \text{selfbutter } (\text{ket empty})) *_S \top)$

$\langle \text{proof} \rangle$

lemma *norm-Proj-3*:

$\text{norm Proj-3} \leq 1$

$\langle \text{proof} \rangle$

theorem *mixed-o2h-nonterm-3*:

shows

$|\text{mixed-H.Pleft } P - \text{mixed-H.Pright Proj-3}| \leq$
 $2 * \text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E) + d * \text{mixed-H.P-nonterm } E)$

and

$|\text{sqrt } (\text{mixed-H.Pleft } P) - \text{sqrt } (\text{mixed-H.Pright Proj-3})| \leq$
 $\text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E) + d * \text{mixed-H.P-nonterm } E)$

$\langle \text{proof} \rangle$

theorem *mixed-o2h-term-3*:

assumes $\bigwedge i. i < d+1 \implies \text{km-trace-preserving } (\text{kf-apply } (E \ i))$

shows

$|\text{mixed-H.Pleft } P - \text{mixed-H.Pright Proj-3}| \leq$
 $2 * \text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E))$

and

$|\text{sqrt } (\text{mixed-H.Pleft } P) - \text{sqrt } (\text{mixed-H.Pright Proj-3})| \leq$
 $\text{sqrt } ((d+1) * \text{Re } (\text{mixed-H.Pfind } E))$

$\langle \text{proof} \rangle$

Theorem 1, definition of Pright (4)

theorem *mixed-o2h-nonterm-4*:

shows

$$|mixed-H.Pleft P - mixed-G.Pright Proj-3| \leq \\ 2 * sqrt ((d+1) * Re (mixed-H.Pfind E) + d * mixed-H.P-nonterm E)$$

and

$$|sqrt (mixed-H.Pleft P) - sqrt (mixed-G.Pright Proj-3)| \leq \\ sqrt ((d+1) * Re (mixed-H.Pfind E) + d * mixed-H.P-nonterm E)$$

$\langle proof \rangle$

theorem *mixed-o2h-term-4*:

assumes $\bigwedge i. i < d+1 \implies km\text{-trace-preserving } (kf\text{-apply } (E \ i))$

shows

$$|mixed-H.Pleft P - mixed-G.Pright Proj-3| \leq \\ 2 * sqrt ((d+1) * Re (mixed-H.Pfind E))$$

and

$$|sqrt (mixed-H.Pleft P) - sqrt (mixed-G.Pright Proj-3)| \leq \\ sqrt ((d+1) * Re (mixed-H.Pfind E))$$

$\langle proof \rangle$

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

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

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

definition *Proj-5* :: $('mem \times 'l) \text{ ell2} \Rightarrow_{CL} ('mem \times 'l) \text{ ell2}$ **where**

$Proj-5 = Proj \ (((P \otimes_o id\text{-cblinfun}) *_S \top) \sqcup ((id\text{-cblinfun} \otimes_o (id\text{-cblinfun} - selfbutter \ (ket \ empty))) *_S \top))$

lemma *is-Proj-5*:

is-Proj Proj-5

$\langle proof \rangle$

lemma *Proj-5-altdef*:

$Proj-5 = Proj-3 + mixed-H.end\text{-measure}$

$\langle proof \rangle$

lemma *norm-Proj-5*:

$norm \ Proj-5 \leq 1$

$\langle proof \rangle$

theorem *mixed-o2h-nonterm-5*:

shows

$|mixed-H.Pleft\ P - (mixed-H.Prigh\ Proj-5)| \leq$
 $2 * sqrt\ ((d+1) * Re\ (mixed-H.Pfind\ E) + d * mixed-H.P-nonterm\ E)$
and
 $|sqrt\ (mixed-H.Pleft\ P) - sqrt\ (mixed-H.Prigh\ Proj-5)| \leq$
 $sqrt\ ((d+1) * Re\ (mixed-H.Pfind\ E) + d * mixed-H.P-nonterm\ E)$
 <proof>

theorem *mixed-o2h-term-5:*

assumes $\bigwedge i. i < d+1 \implies km\text{-}trace\text{-}preserving\ (kf\text{-}apply\ (E\ i))$
shows
 $|mixed-H.Pleft\ P - mixed-H.Prigh\ Proj-5| \leq$
 $2 * sqrt\ ((d+1) * Re\ (mixed-H.Pfind\ E))$
and
 $|sqrt\ (mixed-H.Pleft\ P) - sqrt\ (mixed-H.Prigh\ Proj-5)| \leq$
 $sqrt\ ((d+1) * Re\ (mixed-H.Pfind\ E))$
 <proof>

Theorem 1, definition of Prigh (6)

lemma *Prigh-G-H-case5-nonterm:*

$mixed-H.Prigh\ Proj-5 - mixed-G.Prigh\ Proj-5 = norm\ (mixed-H.qright\ E) - norm\ (mixed-G.qright\ E)$
 <proof>

lemma *Prigh-G-H-case5:*

assumes $\bigwedge i. i < d+1 \implies km\text{-}trace\text{-}preserving\ (kf\text{-}apply\ (E\ i))$
shows $mixed-H.Prigh\ Proj-5 = mixed-G.Prigh\ Proj-5$
 <proof>

theorem *mixed-o2h-nonterm-6:*

shows
 $|mixed-H.Pleft\ P - mixed-G.Prigh\ Proj-5| \leq$
 $2 * sqrt\ ((d+1) * Re\ (mixed-H.Pfind\ E) + d * mixed-H.P-nonterm\ E) +$
 $|norm\ (mixed-H.qright\ E) - norm\ (mixed-G.qright\ E)|$
 <proof>

theorem *mixed-o2h-term-6:*

assumes $\bigwedge i. i < d+1 \implies km\text{-}trace\text{-}preserving\ (kf\text{-}apply\ (E\ i))$
shows
 $|mixed-H.Pleft\ P - mixed-G.Prigh\ Proj-5| \leq$
 $2 * sqrt\ ((d+1) * Re\ (mixed-H.Pfind\ E))$
and
 $|sqrt\ (mixed-H.Pleft\ P) - sqrt\ (mixed-G.Prigh\ Proj-5)| \leq$
 $sqrt\ ((d+1) * Re\ (mixed-H.Pfind\ E))$
 <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 SIG-PLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025*, pages 243–256. ACM, 2025.