Compressed Random Oracles

Dominique Unruh*

January 26, 2026

Abstract

We formalize the compressed quantum random oracle methodology by Zhandry (Crypto 2019). This is a formalism for modeling quantum random oracles to make quantum cryptographic proofs feasible. Our definition of the compressed oracles is loosely based on the presentation from Unruh (arXiv 2021), but with a considerable amount of new definitions and results. In particular, we make extensive use of the quantum references formalism (Unruh, arXiv 2024, AFP 2021) to enable reasoning about queries on arbitrary subsystems, something which is left very informal in pen-and-paper formalizations of the compressed oracles.

We use the developed formalism to prove that finding x with $H(x) = 0$, and finding collisions in H, is hard for quantum adversaries with oracle access to a random function H.

Contents

*Supported by the ERC consolidator grant CerQuS (819317), the PRG team grant Secure Quantum Technology (PRG946) from the Estonian Research Council, and the Estonian Cluster of Excellence "Foundations of the Universe" (TK202).

1 *Misc-Compressed-Oracle* – Miscellaneous required theorems

theory *Misc-Compressed-Oracle*
imports *Registers.Quantum-Extra2*
begin

declare [[*simproc del*: *Laws-Quantum.compatibility-warn*]]

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

1.1 Misc

lemma *assoc-ell2'-ket*[*simp*]: ‹*assoc-ell2∗* $*_V$ *ket* (*x*,*y*,*z*) = *ket* ((*x*,*y*),*z*)›

lemma *assoc-ell2-ket*[*simp*]: ‹*assoc-ell2* $*_V$ *ket* ((*x*,*y*),*z*) = *ket* (*x*,*y*,*z*)›

lemma *sandwich-tensor*:
fixes *a* :: ‹*'a::finite ell2* $\Rightarrow_{CL}$ *'c::finite ell2*› **and** *b* :: ‹*'b::finite ell2* $\Rightarrow_{CL}$ *'d::finite ell2*›
assumes ‹*unitary a*› ‹*unitary b*›
shows *cblinfun-apply* (*sandwich* (*a* $\otimes_o$ *b*)) = *cblinfun-apply* (*sandwich a*) $\otimes_r$ *cblinfun-apply* (*sandwich b*)

lemma *sandwich-grow-left*:
fixes *a* :: ‹*'a::finite ell2* $\Rightarrow_{CL}$ *'b::finite ell2*›
assumes *unitary a*
shows *sandwich a* $\otimes_r$ *id* = *sandwich* (*a* $\otimes_o$ (*id-cblinfun* :: (-::*finite ell2* $\Rightarrow_{CL}$ -)))

lemma *Snd-apply-tensor-ell2*[*simp*]: ‹*Snd a* $*_V$ ($\psi \otimes_s \varphi$) = $\psi \otimes_s$ (*a* $*_V$ φ)›

⟨*ML*⟩

syntax *-register-n-of-m* :: ‹$'a \Rightarrow 'a \Rightarrow 'b$› ([-_])
⟨*ML*⟩

lemma *sum-if*: ‹($\sum x \in X$. *P* (*if Q x then a x else b x*)) = ($\sum x \in X$. *if Q x then P* (*a x*) *else P* (*b x*))›

lemma *sum-if'*: ‹($\sum x \in X$. *P* (*if Q x then a x else b x*) *x*) = ($\sum x \in X$. *if Q x then P* (*a x*) *x else P* (*b x*) *x*)›

lemma *bij-plus*: ‹*bij* ((+) *y*)› **for** *y* :: ‹-::*group-add*›

lemma *tensor-ell2-diff2*: ‹*tensor-ell2 a* (*b* − *c*) = *tensor-ell2 a b* − *tensor-ell2 a c*›
lemma *tensor-ell2-diff1*: ‹*tensor-ell2* (*a* − *b*) *c* = *tensor-ell2 a c* − *tensor-ell2 b c*›

lemma *aminus-bminusc*: ‹*a* − (*b* − *c*) = *a* − *b* + *c*› **for** *a b c* :: ‹- :: *ab-group-add*›

lemma *sum-case'*:
  fixes *a* :: ‹- ⇒ - ⇒ -::*ab-group-add*›
  assumes ‹*finite X*›
  shows ‹($\sum x \in X$. *P* (*case Q x of Some z* ⇒ *a z x* | *None* ⇒ *b x*))
  = ($\sum x \in X \cap \{x.\ Q\ x \neq None\}$. *P* (*a* (*the* (*Q x*)) *x*)) + ($\sum x \in X \cap \{x.\ Q\ x = None\}$. *P* (*b x*))›
  (**is** *?lhs=?rhs*)

lemma *register-isometry*:
  assumes *register F*
  assumes *isometry a*
  shows *isometry* (*F a*)

lemma *register-coisometry*:
  assumes *register F*
  assumes *isometry* (*a*∗)
  shows *isometry* (*F a*∗)

lemma *card-complement*:
  fixes *M* :: ‹*'a::finite set*›
  shows ‹*card* (−*M*) = *CARD*(*'a*) − *card M*›

lemma *register-minus*: ‹*register F* ⟹ *F* (*a* − *b*) = *F a* − *F b*›

lemma *vimage-singleton-inj*: ‹*inj f* ⟹ *f* −‘ {*f x*} = {*x*}›

lemma *has-ell2-norm-0*[*iff*]: ‹*has-ell2-norm* (λ*x*. *0*)›

lemma *ell2-norm-0I*[*simp*]: ‹*ell2-norm* (λ*x*. *0*) = *0*›

lemma *ran-smaller-dom*: ‹*finite* (*dom m*) ⟹ *card* (*ran m*) ≤ *card* (*dom m*)›

lemma *option-sum-split*: ‹($\sum x{\in}X.\ f\ x$) = ($\sum x{\in}Some -‘ X.\ f\ (Some\ x)$) + (*if None* ∈ *X then f None else 0*)› **if** ‹*finite X*› **for** *f X*

lemma *sum-sum-if-eq*: ‹($\sum x{\in}X.\ \sum y{\in}Y\ x.\ if\ x{=}a\ then\ f\ x\ y\ else\ 0$) = (*if a*∈*X then* ($\sum y{\in}Y\ a.\ f\ a\ y$) *else 0*)› **if** ‹*finite X*› **for** *X Y f*

lemma *sum-if-eq-else*: ‹($\sum x{\in}X.\ if\ x{=}a\ then\ 0\ else\ f\ x$) = ($\sum x{\in}X{-}\{a\}.\ f\ x$)› **for** *X f*

lemma *fun-upd-comp-left*:
 assumes ‹*inj g*›
 shows ‹($f \circ g$)($x := y$) = f($g\ x := y$) *o g*›

definition *reg-1-3* :: ‹- ⇒ (*'a::finite* × *'b::finite* × *'c::finite*) *ell2* $\Rightarrow_{CL}$ (*'a* × *'b* × *'c*) *ell2*› **where** ‹*reg-1-3* = *Fst*›
lemma *register-1-3*[*simp*]: ‹*register reg-1-3*›

lemma *comp-reg-1-3*[*simp*]: ‹(*F*;*G*) *o reg-1-3* = *F*› **if** [*register*]: ‹*compatible F G*›

definition *reg-2-3* :: ‹- ⇒ (*'a::finite* × *'b::finite* × *'c::finite*) *ell2* $\Rightarrow_{CL}$ (*'a* × *'b* × *'c*) *ell2*› **where** ‹*reg-2-3* = *Snd o Fst*›
lemma *register-2-3*[*simp*]: ‹*register reg-2-3*›
lemma *comp-reg-2-3*[*simp*]: ‹(*F*;(*G*;*H*)) *o reg-2-3* = *G*› **if** [*register*]: ‹*compatible F G*› ‹*compatible F H*› ‹*compatible G H*›

definition *reg-3-3* :: ‹- ⇒ (*'a::finite* × *'b::finite* × *'c::finite*) *ell2* $\Rightarrow_{CL}$ (*'a* × *'b* × *'c*) *ell2*› **where** ‹*reg-3-3* = *Snd o Snd*›
lemma *register-3-3*[*simp*]: ‹*register reg-3-3*›
lemma *comp-reg-3-3*[*simp*]: ‹(*F*;(*G*;*H*)) *o reg-3-3* = *H*› **if** [*register*]: ‹*compatible F G*› ‹*compatible F H*› ‹*compatible G H*›

lemma [*simp*]: ‹*mutually compatible* (*reg-1-3*, *reg-2-3*, *reg-3-3*)›

lemma *pair-o-tensor-right*:
 assumes [*simp*]: ‹*compatible F G*› ‹*register H*›
 shows ‹(*F*; *G o H*) = (*F*; *G*) *o* (*id* $\otimes_r$ *H*)›

lemma *register-tensor-distrib-right*:
 assumes [*simp*]: ‹*register F*› ‹*register H*› ‹*register L*›
 shows ‹*F* $\otimes_r$ (*H o L*) = (*F* $\otimes_r$ *H*) *o* (*id* $\otimes_r$ *L*)›

lemma *sandwich-apply'*:
 ‹*sandwich U A* $*_V$ ψ = *U* $*_V$ *A* $*_V$ *U*∗ $*_V$ ψ›

lemma *csubspace-set-sum*:
 assumes ‹$\bigwedge x.\ x \in X \Longrightarrow$ *csubspace* (*A x*)›
 shows ‹*csubspace* ($\sum x{\in}X.\ A\ x$)›

lemma *Rep-ell2-vector-to-cblinfun-ket*: ‹*Rep-ell2* ψ *x* = *bra x* $*_V$ ψ›

lemma *trunc-ell2-insert*: ‹*trunc-ell2* (*insert x M*) ψ = *Rep-ell2* ψ *x* $*_C$ *ket x* + *trunc-ell2 M* ψ› **if** ‹$x \notin M$›

lemma *trunc-ell2-in-cspan*:
  assumes ‹*finite S*›
  shows ‹$\mathit{trunc\text{-}ell2}\ S\ \psi \in \mathit{cspan}\ (\mathit{ket}\ `\ S)$›

lemma *space-ccspan-ket*: ‹$\mathit{space\text{-}as\text{-}set}\ (\mathit{ccspan}\ (\mathit{ket}\ `\ M)) = \{\psi.\ \forall\, x \in -M.\ \mathit{Rep\text{-}ell2}\ \psi\ x = 0\}$›

lemma *space-as-set-ccspan-memberI*: ‹$\psi \in \mathit{space\text{-}as\text{-}set}\ (\mathit{ccspan}\ X)$› **if** ‹$\psi \in X$›

lemma *closure-subset-remove-closure*: ‹$X \subseteq \mathit{closure}\ Y \Longrightarrow \mathit{closure}\ X \subseteq \mathit{closure}\ Y$›

lemma *closure-cspan-closure*: ‹$\mathit{closure}\ (\mathit{cspan}\ (\mathit{closure}\ X)) = \mathit{closure}\ (\mathit{cspan}\ X)$›
  for X :: ‹$'a$::*complex-normed-vector set*›

lemma *closure-Sup-closure*: ‹$\mathit{closure}\ (\mathit{Sup}\ (\mathit{closure}\ `\ X)) = \mathit{closure}\ (\mathit{Sup}\ X)$›

lemma *cspan-closure-cspan*: ‹$\mathit{cspan}\ (\mathit{closure}\ (\mathit{cspan}\ X)) = \mathit{closure}\ (\mathit{cspan}\ X)$›
  for X :: ‹$'a$::*complex-normed-vector set*›

lemma *cblinfun-image-SUP*: ‹$A *_S (\mathit{SUP}\ x{\in}X.\ I\ x) = (\mathit{SUP}\ x{\in}X.\ A *_S I\ x)$›

lemma *cspan-Sup-cspan*: ‹$\mathit{cspan}\ (\mathit{Sup}\ (\mathit{cspan}\ `\ X)) = \mathit{cspan}\ (\mathit{Sup}\ X)$›

lemma *ccspan-Sup*: ‹$\mathit{ccspan}\ (\bigcup X) = \mathit{Sup}\ (\mathit{ccspan}\ `\ X)$›

lemma *ccspan-space-as-set*[*simp*]: ‹$\mathit{ccspan}\ (\mathit{space\text{-}as\text{-}set}\ S) = S$›

lemma *cblinfun-image-Sup*: ‹$A *_S \mathit{Sup}\ \mathit{II} = (\mathit{SUP}\ I{\in}\mathit{II}.\ A *_S I)$›

lemma *space-as-set-mono*: ‹$I \le J \Longrightarrow \mathit{space\text{-}as\text{-}set}\ I \le \mathit{space\text{-}as\text{-}set}\ J$›

lemma *square-into-sum*:
  fixes $X\ Y$ **and** f :: ‹$\text{-} \Rightarrow \mathit{real}$›
  assumes ‹$\bigwedge x.\ f\ x \ge 0$›
  shows ‹$(\sum x{\in}X.\ f\ x)^2 \le \mathit{card}\ X * (\sum x{\in}X.\ (f\ x)^2)$›

lemma *bound-coeff-sum2*:
  fixes $X\ Y$ **and** f :: ‹$'a \Rightarrow \mathit{complex}$›
  assumes [*simp*]: ‹*finite Y*›
  assumes XY: ‹$X \subseteq Y$›
  assumes *sum1*: ‹$(\sum x{\in}Y.\ (\mathit{cmod}\ (f\ x))^2) \le 1$›
  assumes k: ‹$\bigwedge x.\ x \in X \Longrightarrow \mathit{card}\ \{y.\ g\ x = g\ y \wedge y \in X\} \le k$›
  shows ‹$\mathit{norm}\ (\sum x{\in}X.\ f\ x *_C \mathit{ket}\ (g\ x)) \le \mathit{sqrt}\ k$›

lemma *norm-ortho-sum-bound*:
  fixes X
  assumes *bound*: ‹$\bigwedge x.\ x{\in}X \Longrightarrow \mathit{norm}\ (\psi\ x) \le b'$›
  assumes *b'geq0*: ‹$b' \ge 0$›
  assumes *ortho*: ‹$\bigwedge x\ y.\ x{\in}X \Longrightarrow y{\in}X \Longrightarrow x{\neq}y \Longrightarrow \mathit{is\text{-}orthogonal}\ (\psi\ x)\ (\psi\ y)$›
  assumes *b'b*: ‹$\mathit{sqrt}\ (\mathit{card}\ X) * b' \le b$›
  shows ‹$\mathit{norm}\ (\sum x{\in}X.\ \psi\ x) \le b$›

lemma *compatible-project1*: ‹*compatible F G*›
  if ‹*compatible F* (*G*;*H*)› **and** [*register*]: ‹*compatible G H*› **for** *F G H*

lemma *compatible-project2*: ‹*compatible F H*›
if ‹*compatible F* (*G*;*H*)› **and** [*register*]: ‹*compatible G H*› **for** *F G H*

lemma *proj-ket-x-y*: ‹*proj* (*ket x*) $*_V$ (*ket y*) = *0*› **if** ‹$x \neq y$›

lemma *proj-ket-x-y-ofbool*: ‹*proj* (*ket x*) $*_V$ (*ket y*) = *of-bool* (*x*=*y*) $*_C$ *ket y*›

lemma *proj-x-x*[*simp*]: ‹*proj x* $*_V$ *x* = *x*›

lemma *in-ortho-ccspan*: ‹*y* ∈ *space-as-set* (− *ccspan X*)› **if** ‹∀ *x*∈*X*. *is-orthogonal y x*›

lemma *swap-sandwich-swap-ell2*: *swap* = *sandwich swap-ell2*

lemma *is-Proj-sandwich*: ‹*is-Proj* (*sandwich U P*)› **if** ‹*isometry U*› **and** ‹*is-Proj P*›
for *P* :: ‹*'a*::*chilbert-space* $\Rightarrow_{CL}$ *'a*› **and** *U* :: ‹*'a* $\Rightarrow_{CL}$ *'b*::*chilbert-space*›

lemma *is-Proj-swap*[*simp*]: ‹*is-Proj* (*swap P*)› **if** ‹*is-Proj P*›

lemma *iso-register-complement-pair*: ‹*iso-register* (*complement X*; *X*)› **if** ‹*register X*›

lemma *swap-Snd*: ‹*swap* (*Snd x*) = *Fst x*›

lemma *sandwich-butterfly*: ‹*sandwich a* (*butterfly g h*) = *butterfly* (*a g*) (*a h*)›

lemma *register0*:
assumes ‹*register Q*›
shows ‹*Q a* = *0* ⟷ *a* = *0*›

lemma *le-back-subst*:
assumes ‹*a* ≤ *c*›
assumes ‹*a* = *b*›
shows ‹*b* ≤ *c*›

lemma *le-back-subst-le*:
fixes *a b c* :: ‹- :: *order*›
assumes ‹*a* ≤ *c*›
assumes ‹*b* ≤ *a*›
shows ‹*b* ≤ *c*›

lemma *arg-cong4*: ‹*f a b c d* = *f a′ b′ c′ d′*› **if** ‹*a* = *a′*› **and** ‹*b* = *b′*› **and** ‹*c* = *c′*› **and** ‹*d* = *d′*›

1.2 Controlled operations

definition *controlled-op* :: ‹(*'a* ⇒ (*'b ell2* $\Rightarrow_{CL}$ *'c ell2*)) ⇒ ((*'a*×*'b*) *ell2* $\Rightarrow_{CL}$ (*'a*×*'c*) *ell2*)› **where**
‹*controlled-op A* = *infsum-in cstrong-operator-topology* (λ*x*. *selfbutter* (*ket x*) $\otimes_o$ *A x*) *UNIV*›

lemma *trunc-ell2-prod-tensor*: ‹*trunc-ell2* (*A*×*B*) (*g* $\otimes_s$ *h*) = *trunc-ell2 A g* $\otimes_s$ *trunc-ell2 B h*›

lemma *trunc-ell2-ket*: ‹*trunc-ell2 S* (*ket x*) = *of-bool* (*x*∈*S*) $*_C$ *ket x*›

lemma *summable-on-in-0*[*iff*]: ‹*summable-on-in T* (λ*x*. *0*) *A*› **if** ‹*0* ∈ *topspace T*›

lemma *sum-of-bool-scaleC*: ‹($\sum$ *x*∈*S*. *of-bool* (*x*=*a*) $*_C$ *f x*) = (*if a*∈*S* ∧ *finite S then f a else 0*)›
for *f* :: ‹- ⇒ -::*complex-vector*›

lemma
fixes *A* :: ‹′*x* ⇒ (′*a ell2* $\Rightarrow_{CL}$ ′*b ell2*)›

assumes ‹⋀*x*. *norm* (*A x*) ≤ *B*›
shows *controlled-op-has-sum-aux*: ‹*has-sum-in cstrong-operator-topology* (λ*x*. *selfbutter* (*ket x*) $\otimes_o$ *A x*) *UNIV* (*controlled-op A*)›
and *controlled-op-norm-leq*: ‹*norm* (*controlled-op A*) ≤ *B*›

lemma *controlled-op-has-sum*:
fixes *A* :: ‹′*x* ⇒ (′*a ell2* $\Rightarrow_{CL}$ ′*b ell2*)›
assumes ‹*bdd-above* (*range* (λ*x*. *norm* (*A x*)))›
shows ‹*has-sum-in cstrong-operator-topology* (λ*x*. *selfbutter* (*ket x*) $\otimes_o$ *A x*) *UNIV* (*controlled-op A*)›

hide-fact *controlled-op-has-sum-aux*

lemma *controlled-op-summable*:
fixes *A* :: ‹′*x* ⇒ (′*a ell2* $\Rightarrow_{CL}$ ′*b ell2*)›
assumes ‹*bdd-above* (*range* (λ*x*. *norm* (*A x*)))›
shows ‹*summable-on-in cstrong-operator-topology* (λ*x*. *selfbutter* (*ket x*) $\otimes_o$ *A x*) *UNIV*›

lemma *infsum-sot-cblinfun-apply*:
assumes ‹*summable-on-in cstrong-operator-topology f UNIV*›
shows ‹*infsum-in cstrong-operator-topology f UNIV* $*_V$ ψ = ($\sum_\infty$*x*. *f x* $*_V$ ψ)›

lemma *controlled-op-ket*[*simp*]:
assumes ‹*bdd-above* (*range* (λ*x*. *norm* (*A x*)))›
shows ‹*controlled-op A* $*_V$ (*ket x* $\otimes_s$ ψ) = *ket x* $\otimes_s$ (*A x* $*_V$ ψ)›

lemma *controlled-op-ket′*[*simp*]:
assumes ‹*bdd-above* (*range* (λ*x*. *norm* (*A x*)))›
shows ‹*controlled-op A* $*_V$ (*ket* (*x*, *y*)) = *ket x* $\otimes_s$ (*A x* $*_V$ *ket y*)›

lemma *controlled-op-compose*[*simp*]:
assumes [*simp*]: ‹*bdd-above* (*range* (λ*x*. *norm* (*A x*)))›
assumes [*simp*]: ‹*bdd-above* (*range* (λ*x*. *norm* (*B x*)))›
shows ‹*controlled-op A* o_{CL} *controlled-op B* = *controlled-op* (λ*x*. *A x* o_{CL} *B x*)›

lemma *controlled-op-adj*[*simp*]:
assumes [*simp*]: ‹*bdd-above* (*range* (λ*x*. *norm* (*A x*)))›
shows ‹(*controlled-op A*)∗ = *controlled-op* (λ*x*. (*A x*)∗)›

lemma *controlled-op-id*[*simp*]: ‹*controlled-op* (λ-. *id-cblinfun*) = *id-cblinfun*›

lemma *controlled-op-unitary*[*simp*]: ‹*unitary* (*controlled-op* U)› **if** [*simp*]: ‹$\bigwedge x$. *unitary* (U x)›

lemma *controlled-op-is-Proj*[*simp*]: ‹*is-Proj* (*controlled-op* P)› **if** [*simp*]: ‹$\bigwedge x$. *is-Proj* (P x)›

lemma *controlled-op-comp-butter*:
  assumes ‹*bdd-above* (*range* (λ*x*. *norm* (A x)))›
  shows ‹*controlled-op* A o_{CL} *Fst* (*selfbutter* (*ket* x)) = *Snd* (A x) o_{CL} *Fst* (*selfbutter* (*ket* x))›

lemma *norm-ell2-finite*: ‹*norm* ψ = *sqrt* ($\sum i \in UNIV$. (*cmod* (*Rep-ell2* ψ i))^2)› **for** ψ :: ‹-::*finite ell2*›

lemma *controlled-op-ket-swap*[*simp*]:
  assumes ‹*bdd-above* (*range* (λ*x*. *norm* (U x)))›
  shows ‹*swap* (*controlled-op* U) $*_V$ (A $\otimes_s$ *ket* x) = (U x $*_V$ A) $\otimes_s$ *ket* x›

lemma *controlled-op-const*: ‹*controlled-op* (λ-. P) = *Snd* P›

1.3 Superpositions

lift-definition *uniform-superpos* :: ‹*'a set* ⇒ *'a ell2*› **is** ‹λA x. *complex-of-real* (*of-bool* ($x \in A$) / *sqrt* (*of-nat* (*card* A)))›

lemma *norm-uniform-superpos*: ‹*norm* (*uniform-superpos* A) = *1*› **if** ‹*finite* A› **and** ‹$A \neq \{\}$›

lemma *uniform-superpos-infinite*: ‹*uniform-superpos* A = *0*› **if** ‹*infinite* A›

lemma *uniform-superpos-empty*: ‹*uniform-superpos* A = *0*› **if** ‹$A = \{\}$›

Alternative definition.

lemma *uniform-superpos-def2*: ‹*uniform-superpos* A = ($\sum f \in A$. *ket* f $/_C$ *csqrt* (*card* A))›

1.4 Lifting ell2 to option type

lift-definition *lift-ell2'* :: ‹*'a ell2* ⇒ *'a option ell2*› **is** ‹λψ x. *case* x *of Some* x' ⇒ ψ x' | *None* ⇒ *0*›

lemma *clinear-lift-ell2'*: ‹*clinear lift-ell2'*›

lemma *lift-ell2'-norm*[*simp*]: ‹*norm* (*lift-ell2'* ψ) = *norm* ψ›

lemma *bounded-clinear-lift-ell2'*[*bounded-clinear*, *simp*]: ‹*bounded-clinear lift-ell2'*›

lift-definition *lift-ell2* :: ‹*'a ell2* $\Rightarrow_{CL}$ *'a option ell2*› **is** *lift-ell2'*

definition *lift-op* :: ‹(*'a ell2* $\Rightarrow_{CL}$ *'b ell2*) ⇒ (*'a option ell2* $\Rightarrow_{CL}$ *'b option ell2*)› **where**
  ‹*lift-op* A = (*lift-ell2* o_{CL} A o_{CL} *lift-ell2*∗) + *butterfly* (*ket None*) (*ket None*)›

lemma *lift-ell2-ket*[*simp*]: ‹*lift-ell2* $*_V$ *ket* x = *ket* (*Some* x)›

lemma *isometry-lift-ell2*[*simp*]: ‹*isometry lift-ell2*›

lemma *lift-op-adj*: ‹(*lift-op A*)∗ = *lift-op* (*A*∗)›

lemma *bra-None-lift-ell2*: ‹*bra None* o_{CL} *lift-ell2* = *0*›

lemma *lift-op-mult*: ‹*lift-op A* o_{CL} *lift-op B* = *lift-op* (*A* o_{CL} *B*)›

lemma *lift-ell2-adj-None*[*simp*]: ‹*lift-ell2*∗ $*_V$ *ket None* = *0*›

lemma *lift-ell2-adj-Some*[*simp*]: ‹*lift-ell2*∗ $*_V$ *ket* (*Some x*) = *ket x*›

lemma *lift-op-id*[*simp*]: ‹*lift-op id-cblinfun* = *id-cblinfun*›

lemma *isometry-lift-op*[*simp*]: ‹*isometry* (*lift-op A*)› **if** ‹*isometry A*›

lemma *unitary-lift-op*[*simp*]: ‹*unitary* (*lift-op A*)› **if** ‹*unitary A*›

lemma *lift-op-None*[*simp*]: ‹*lift-op A* $*_V$ *ket None* = *ket None*›

lemma *lift-op-Some*[*simp*]: ‹*lift-op A* $*_V$ *ket* (*Some x*) = *lift-ell2* $*_V$ *A* $*_V$ *ket x*›

declare *register-tensor-is-register*[*simp*]

lemma *sum-sqrt*: ‹($\sum$ *i*<*n*. *sqrt i*) ≤ *2*/*3* ∗ (*sqrt n*)^*3*› **for** *n* :: *nat*

lemma *register-inj′*:
  assumes ‹*register Q*›
  shows ‹*Q a* = *Q b* ⟷ *a* = *b*›

lemma *norm-cblinfun-apply-leq1I*:
  assumes ‹*norm U* ≤ *1*›
  assumes ‹*norm ψ* ≤ *1*›
  shows ‹*norm* (*U* $*_V$ *ψ*) ≤ *1*›

lemma *times-sqrtn-div-n*[*simp*]:
  assumes ‹*n* ≥ *0*›
  shows ‹*a* ∗ *sqrt n* / *n* = *a* / *sqrt n*›

lemma *Proj-tensor-Proj*: ‹*Proj I* $\otimes_o$ *Proj J* = *Proj* (*I* $\otimes_S$ *J*)›

lemma *extend-mult-rule*: ‹*a* ∗ *b* = *c* ⟹ *a* ∗ (*b* ∗ *d*) = *c* ∗ *d*› **for** *a b c d* :: ‹-::*semigroup-mult*›

end

2 *Function-At* – Function values as individual registers

theory *Function-At*
imports *Registers.Quantum-Extra Misc-Compressed-Oracle*
begin

unbundle *no m-inv-syntax*

typedef (′*a*,′*b*) *punctured-function* = ‹*extensional* (−{*undefined*}) :: (′*a*⇒′*b*) *set*›

setup-lifting *type-definition-punctured-function*
instance *punctured-function* :: (*finite*, *finite*) *finite*

lift-definition *fix-punctured-function* :: ‹$'a \Rightarrow ('b \times ('a,'b)$ *punctured-function*$) \Rightarrow ('a \Rightarrow 'b)$› **is**
‹$\lambda x\ (y, f).$ (*Fun.swap x undefined f*) $(x := y)$›

lift-definition *puncture-function* :: ‹$'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b \times ('a,'b)$ *punctured-function*› **is**
‹$\lambda x\, f.$ ($f\, x$, (*Fun.swap x undefined f*) (*undefined* := *undefined*))›

lemma *puncture-function-recombine*:
‹(y, *snd* (*puncture-function x f*)) = *puncture-function x* ($f(x:=y)$)›

lemma *snd-puncture-function-upd*: ‹*snd* (*puncture-function x* ($f(x:=y)$)) = *snd* (*puncture-function x f*)›

lemma *puncture-function-split*: ‹*puncture-function x f* = ($f\, x$, *snd* (*puncture-function x f*))›

lemma *puncture-function-inverse*[*simp*]: ‹*fix-punctured-function x* (*puncture-function x f*) = f›

lemma *fix-punctured-function-inverse*[*simp*]: ‹*puncture-function x* (*fix-punctured-function x yf*) = *yf*›

lemma *bij-fix-punctured-function*[*simp*]: ‹*bij* (*fix-punctured-function x*)›

lemma *inj-fix-punctured-function*[*simp*]: ‹*inj* (*fix-punctured-function x*)›

lemma *surj-fix-punctured-function*[*simp*]: ‹*surj* (*fix-punctured-function x*)›

The following *function-at-U x* provides an unitary isomorphism between $('a \Rightarrow 'b)$ *ell2* (superposition of functions) and $('b \times ('a, 'b)$ *punctured-function*) *ell2* (superposition of pairs of the value of the function at x and the rest of the function). This allows to then apply a some operation to the first part of that pair and thus lifting it to an application to the whole function. (The "rest of the function" part is to be considered opaque.)

definition *function-at-U* :: ‹$'a \Rightarrow ('b \times ('a, 'b)$ *punctured-function*) *ell2* $\Rightarrow_{CL}$ $('a \Rightarrow 'b)$ *ell2*› **where**
‹*function-at-U x* = *classical-operator* (*Some o fix-punctured-function x*)›

lemma *unitary-function-at-U*[*simp*]: ‹*unitary* (*function-at-U x*)›

lemma *function-at-U-ket*[*simp*]: ‹*function-at-U x* $*_V$ *ket y* = *ket* (*fix-punctured-function x y*)›

lemma *function-at-U-adj-ket*[*simp*]: ‹(*function-at-U x*)$*$ $*_V$ *ket y* = *ket* (*puncture-function x y*)›

The reference *function-at x* lifts an operation U on $'a$ *ell2* to an operation on $('a \Rightarrow 'b)$ *ell2* (superposition of functions). The resulting operation applies U only to the x-output of the function.

definition *function-at* :: ‹$'a \Rightarrow ('b$ *update* $\Rightarrow ('a \Rightarrow 'b)$ *update*)› **where**
‹*function-at x* = *sandwich* (*function-at-U x*) *o Fst*›

lemma *Rep-ell2-function-at-ket*:
‹*Rep-ell2* (*function-at x U* $*_V$ *ket f*) *g* =
of-bool (*snd* (*puncture-function x f*) = *snd* (*puncture-function x g*)) $*$ *Rep-ell2* (U $*_V$ *ket* ($f\, x$)) (g x)›

lemma *function-at-ket*:

shows ‹*function-at x U* $*_V$ *ket f* = ($\sum_\infty$*y*∈*UNIV*. *Rep-ell2* (*U* $*_V$ *ket* (*f x*)) *y* $*_C$ *ket* (*f* (*x* := *y*)))›

lemma *register-function-at*[*simp*, *register*]: ‹*register* (*function-at x* :: ′*b update* ⇒ (′*a*⇒′*b*) *update*)› **for** *x* :: ′*a*

lemma *function-at-comm*:
  fixes *U V* :: ‹′*b ell2* $\Rightarrow_{CL}$ ′*b ell2*› **and** *x y* :: ′*a*
  assumes ‹*x* ≠ *y*›
  shows ‹*function-at x U* o_{CL} *function-at y V* = *function-at y V* o_{CL} *function-at x U*›

lemma *compatible-function-at*[*simp*]:
  assumes ‹*x* ≠ *y*›
  shows ‹*compatible* (*function-at x*) (*function-at y*)›

lemma *inv-fix-punctured-function*[*simp*]: ‹*inv* (*fix-punctured-function x*) = *puncture-function x*›

lemma *bij-puncture-function*[*simp*]: ‹*bij* (*puncture-function x*)›

lemma *fst-puncture-function*[*simp*]: ‹*fst* (*puncture-function x H*) = *H x*›

2.1 *apply-every*

Analogue to classical λ*M u f x*. *if x* ∈ *M then u x* (*f x*) *else f x*.

Note that the definition only makes sense when *M* is finite. In fact, a definition that works for infinite *M* is impossible as the following example shows: Let *H* denote the Hadamard matrix. Let *M* = *UNIV*. Then, by symmetry, a meaningful definition of *apply-every* would have that *apply-every M H* (*ket* (λ-. *0*)) would be a vector in (*nat* ⇒ *bit*) *ell2* with all coefficients equal. But the only such vector is *0*. But a meaningful definition should not map *ket* (λ-. *0*) to *0*.

definition *apply-every* **where** ‹*apply-every M U* = (*if finite M then Finite-Set.fold* (λ*x a*. *function-at x* (*U x*) o_{CL} *a*) *id-cblinfun M else 0*)›

lemma *apply-every-empty*[*simp*]: ‹*apply-every* {} *U* = *id-cblinfun*›

interpretation *apply-every-aux*: *comp-fun-commute* ‹(λ*x*. (o_{CL}) (*function-at x* (*U x*)))›

lemma *apply-every-unitary*: ‹*unitary* (*apply-every M U*)› **if** ‹*finite M*› **and** [*simp*]: ‹⋀*x*. *x*∈*M* ⟹ *unitary* (*U x*)›

lemma *apply-every-comm*: ‹*apply-every M U* o_{CL} *V* = *V* o_{CL} *apply-every M U*›
  if ‹*finite M*› **and** ‹⋀*x*. *x*∈*M* ⟹ *function-at x* (*U x*) o_{CL} *V* = *V* o_{CL} *function-at x* (*U x*)›

lemma *apply-every-infinite*: ‹*apply-every M U* = *0*› **if** ‹*infinite M*›

lemma *apply-every-split*: ‹*apply-every M U* o_{CL} *apply-every N U* = *apply-every* (*M* ∪ *N*) *U*› **if** ‹*M* ∩ *N* = {}› **for** *M N U*

lemma *apply-every-single*[*simp*]: ‹*apply-every* {*x*} *U* = *function-at x* (*U x*)›

lemma *apply-every-insert*: ‹*apply-every* (*insert x M*) *U* = *function-at x* (*U x*) o_{CL} *apply-every M U*› **if** ‹*x* ∉ *M*› **and** ‹*finite M*›

lemma *apply-every-mult*: ‹$apply\text{-}every\ M\ U\ o_{CL}\ apply\text{-}every\ M\ V = apply\text{-}every\ M\ (\lambda x.\ U\ x\ o_{CL}\ V\ x)$›

lemma *apply-every-id*[*simp*]: ‹$apply\text{-}every\ M\ (\lambda\text{-}.\ id\text{-}cblinfun) = id\text{-}cblinfun$› **if** ‹*finite M*›

lemma *apply-every-function-at-comm*:
  assumes ‹$x \notin M$›
  shows ‹$function\text{-}at\ x\ U\ o_{CL}\ apply\text{-}every\ M\ f = apply\text{-}every\ M\ f\ o_{CL}\ function\text{-}at\ x\ U$›

lemma *apply-every-adj*: ‹$(apply\text{-}every\ M\ f)* = apply\text{-}every\ M\ (\lambda i.\ (f\ i)*)$›

end

3 *Invariant-Preservation* Preservation of invariants under queries

theory *Invariant-Preservation*
  imports *Function-At Misc-Compressed-Oracle*
begin

hide-const (**open**) *Order.top*
no-notation *Order.bottom* ($\perp_1$)
unbundle *no m-inv-syntax*
unbundle *lattice-syntax*

3.1 Invariants

definition ‹$preserves\ U\ I\ J\ \varepsilon \longleftrightarrow \varepsilon \geq 0 \land (\forall \psi \in space\text{-}as\text{-}set\ I.\ norm\ (U *_V \psi - Proj\ J *_V U *_V \psi) \leq \varepsilon * norm\ \psi)$›
  for U :: ‹$'a::chilbert\text{-}space \Rightarrow_{CL} 'b::chilbert\text{-}space$›

lemma *preserves-def-closure*:
  assumes ‹$space\text{-}as\text{-}set\ I = closure\ I'$›
  shows ‹$preserves\ U\ I\ J\ \varepsilon \longleftrightarrow \varepsilon \geq 0 \land (\forall \psi \in I'.\ norm\ (U *_V \psi - Proj\ J *_V U *_V \psi) \leq \varepsilon * norm\ \psi)$›

lemma *preservesI-closure*:
  assumes ‹$\varepsilon \geq 0$›
  assumes *closure*: ‹$space\text{-}as\text{-}set\ I \subseteq closure\ I'$›
  assumes ‹*csubspace I'*›
  assumes *bound*: ‹$\bigwedge\psi.\ \psi \in I' \Longrightarrow norm\ \psi = 1 \Longrightarrow norm\ (U *_V \psi - Proj\ J *_V U *_V \psi) \leq \varepsilon$›
  shows ‹$preserves\ U\ I\ J\ \varepsilon$›

lemma *preservesI*:
  assumes ‹$\varepsilon \geq 0$›
  assumes ‹$\bigwedge\psi.\ \psi \in space\text{-}as\text{-}set\ I \Longrightarrow norm\ \psi = 1 \Longrightarrow norm\ (U *_V \psi - Proj\ J *_V U *_V \psi) \leq \varepsilon$›
  shows ‹$preserves\ U\ I\ J\ \varepsilon$›

lemma *preservesI'*:
  assumes ‹$\varepsilon \geq 0$›
  assumes ‹$\bigwedge\psi.\ \psi \in space\text{-}as\text{-}set\ I \Longrightarrow norm\ \psi = 1 \Longrightarrow norm\ (Proj\ (-J) *_V U *_V \psi) \leq \varepsilon$›
  shows ‹$preserves\ U\ I\ J\ \varepsilon$›

lemma *preserves-onorm*: ‹$preserves\ U\ I\ J\ \varepsilon \longleftrightarrow norm\ ((id\text{-}cblinfun - Proj\ J)\ o_{CL}\ U\ o_{CL}\ Proj\ I) \leq \varepsilon$›

lemma *preserves-cong*:
  assumes ‹$\bigwedge\psi.\ \psi \in$ *space-as-set* $I \Longrightarrow U *_V \psi = U' *_V \psi$›
  shows ‹*preserves* $U\ I\ J\ \varepsilon \longleftrightarrow$ *preserves* $U'\ I\ J\ \varepsilon$›

lemma *preserves-mono*:
  assumes ‹*preserves* $U\ I\ J\ \varepsilon$›
  assumes ‹$I \geq I'$›
  assumes ‹$J \leq J'$›
  assumes ‹$\varepsilon \leq \varepsilon'$›
  shows ‹*preserves* $U\ I'\ J'\ \varepsilon'$›

The next lemma allows us to decompose the preservation of an invariant into the preservation of simpler invariants. The main requirement is that the simpler invariants are all orthogonal.

This is in particular useful when one wants to show the preservation of an invariant that refers to the oracle input register and other unrelated registers. One can then decompose the invariant into many invariants that fix the input and unrelated registers to specific computational basis states. (I.e., wlog the input register is in a state of the form *ket x*.

Unfortunately, we have a proof only in the case of finitely many simpler invariants. This excludes, e.g., infinite oracle input registers etc. (e.g., quantum ints, quantum lists).

lemma *invariant-splitting*:
  fixes X :: ‹$'i$ *set*›
  fixes $I\ S$ :: ‹$'i \Rightarrow$ $'a$::*chilbert-space ccsubspace*›
  fixes J :: ‹$'i \Rightarrow$ $'b$::*chilbert-space ccsubspace*›
  assumes *ortho-S*: ‹$\bigwedge x\ y.\ x{\in}X \Longrightarrow y{\in}X \Longrightarrow x \neq y \Longrightarrow$ *orthogonal-spaces* $(S\ x)\ (S\ y)$›
  assumes *ortho-S'*: ‹$\bigwedge x\ y.\ x{\in}X \Longrightarrow y{\in}X \Longrightarrow x \neq y \Longrightarrow$ *orthogonal-spaces* $(S'\ x)\ (S'\ y)$›
  assumes *IS*: ‹$\bigwedge x.\ x{\in}X \Longrightarrow I\ x \leq S\ x$›
  assumes *JS'*: ‹$\bigwedge x.\ x{\in}X \Longrightarrow J\ x \leq S'\ x$›
  assumes *USS'*: ‹$\bigwedge x.\ x{\in}X \Longrightarrow U *_S S\ x \leq S'\ x$›
  assumes *II*: ‹$II \leq (\sum x{\in}X.\ I\ x)$›
  assumes *JJ*: ‹$JJ \geq (\sum x{\in}X.\ J\ x)$›
  assumes *ε0*: ‹$\varepsilon \geq 0$›
  assumes [*iff*]: ‹*finite* X›
  assumes *pres*: ‹$\bigwedge x.\ x{\in}X \Longrightarrow$ *preserves* $U\ (I\ x)\ (J\ x)\ \varepsilon$›
  shows ‹*preserves* $U\ II\ JJ\ \varepsilon$›

An invariant that is consists of all states that are the superposition of computational basis states.

Useful for representing a classically formulated condition (e.g., $x \neq 0$) as an invariant (*ket-invariant* $\{x.\ x \neq 0\}$).

definition ‹*ket-invariant* M = *ccspan* (*ket* ‘ M)›

lemma *ket-invariant-UNIV*[*simp*]: ‹*ket-invariant UNIV* = $\top$›

lemma *ket-invariant-empty*[*simp*]: ‹*ket-invariant* $\{\} = \bot$›

lemma *ket-invariant-Rep-ell2*: ‹$\psi \in$ *space-as-set* (*ket-invariant* I) $\longleftrightarrow$ ($\forall\, i{\in}{-I}.$ *Rep-ell2* $\psi\ i = 0$)›

lemma *ket-invariant-compl*: ‹*ket-invariant* $(-M) = -$ *ket-invariant* M›

lemma *ket-invariant-tensor*: ‹*ket-invariant* $I \otimes_S$ *ket-invariant* J = *ket-invariant* $(I \times J)$›

abbreviation ‹*preserves-ket U I J* $\varepsilon \equiv$ *preserves U* (*ket-invariant I*) (*ket-invariant J*) ε›

lemma *orthogonal-spaces-ket*[*simp*]: ‹*orthogonal-spaces* (*ket-invariant M*) (*ket-invariant N*) $\longleftrightarrow M \cap N = \{\}$› **for** *M N*

lemma *ket-invariant-le*[*simp*]: ‹*ket-invariant M* $\leq$ *ket-invariant N* $\longleftrightarrow M \subseteq N$› **for** *M N*

lemma *ket-invariant-mono*:
  assumes ‹$I \subseteq J$›
  shows ‹*ket-invariant I* $\leq$ *ket-invariant J*›

lemma *ket-invariant-Inf*: ‹*ket-invariant* (*Inf M*) = *Inf* (*ket-invariant* ‘ *M*)›

lemma *ket-invariant-INF*: ‹*ket-invariant* (*INF* $x \in M$. *f x*) = (*INF* $x \in M$. *ket-invariant* (*f x*))›

lemma *ket-invariant-Sup*: ‹*ket-invariant* (*Sup M*) = *Sup* (*ket-invariant* ‘ *M*)›

lemma *ket-invariant-SUP*: ‹*ket-invariant* (*SUP* $x \in M$. *f x*) = (*SUP* $x \in M$. *ket-invariant* (*f x*))›

lemma *ket-invariant-inter*: ‹*ket-invariant M* $\sqcap$ *ket-invariant N* = *ket-invariant* ($M \cap N$)› **for** *M N*

lemma *ket-invariant-union*: ‹*ket-invariant M* $\sqcup$ *ket-invariant N* = *ket-invariant* ($M \cup N$)› **for** *M N*

lemma *sum-ket-invariant*[*simp*]:
  assumes ‹*finite X*›
  shows ‹($\sum x \in X$. *ket-invariant* (*M x*)) = *ket-invariant* ($\bigcup x \in X$. *M x*)›

lemma *ket-invariant-inj*[*simp*]:
  ‹*ket-invariant M* = *ket-invariant N* $\longleftrightarrow M = N$› **for** *M N*

Given an invariant on the content of a register, this gives the corresponding invariant on the whole state. Useful for plugging together several invariants on different subsystems.

definition ‹*lift-invariant F I* = *F* (*Proj I*) $*_S$ $\top$›

lemma *lift-invariant-comp*:
  assumes [*simp*]: ‹*register G*›
  shows ‹*lift-invariant* (*F o G*) = *lift-invariant F o lift-invariant G*›

lemma *lift-invariant-top*[*simp*]: ‹*register F* $\Longrightarrow$ *lift-invariant F* $\top = \top$›

lemma *Proj-lift-invariant*: ‹*register F* $\Longrightarrow$ *Proj* (*lift-invariant F I*) = *F* (*Proj I*)›

lemma *ket-invariant-image-assoc*:
  ‹*ket-invariant* ((λ((*a*, *b*), *c*). (*a*, *b*, *c*)) ‘ *X*) = *lift-invariant assoc* (*ket-invariant X*)›

lemma *lift-invariant-inj*[*simp*]: ‹*lift-invariant F I* = *lift-invariant F J* $\longleftrightarrow I = J$› **if** [*register*]: ‹*register F*›

lemma *lift-invariant-decomp*:
  fixes *U* :: ‹- $\Rightarrow_{CL}$ -::*chilbert-space*›

assumes ‹$\bigwedge\vartheta$. *F* ϑ = *sandwich U* $*_V$ ($\vartheta \otimes_o$ *id-cblinfun*)›
assumes ‹*unitary U*›
shows ‹*lift-invariant F I* = *U* $*_S$ (*I* $\otimes_S$ $\top$)›

Invariants are compatible if their projectors commute, i.e., if you can simultaneously measure them. This can happen if they refer to different parts of the system. (E.g., one talks about register X, the other about register Y.) But also for example for any ket-invariants.

See lemma *preserves-intersect* below for a useful consequence.

definition ‹*compatible-invariants A B* $\longleftrightarrow$ *Proj A* o_{CL} *Proj B* = *Proj B* o_{CL} *Proj A*›

lemma *compatible-invariants-inter*: ‹*Proj A* o_{CL} *Proj B* = *Proj* (*A* $\sqcap$ *B*)› **if** ‹*compatible-invariants A B*›

lemma *compatible-invariants-ket*[*iff*]: ‹*compatible-invariants* (*ket-invariant I*) (*ket-invariant J*)›

lemma *preserves-intersect*:
assumes ‹*compatible-invariants J1 J2*›
assumes *pres1*: ‹*preserves U I J1* ε*1*›
assumes *pres2*: ‹*preserves U I J2* ε*2*›
shows ‹*preserves U I* (*J1* $\sqcap$ *J2*) (ε*1* + ε*2*)›

lemma *preserves-intersect-ket*:
assumes ‹*preserves-ket U I J1* ε*1*›
assumes ‹*preserves-ket U I J2* ε*2*›
shows ‹*preserves-ket U I* (*J1* $\sqcap$ *J2*) (ε*1* + ε*2*)›

An invariant is compatible with a register intuitively if the invariant only talks about parts of the quantum state outside the register.

definition ‹*compatible-register-invariant F I* $\longleftrightarrow$ ($\forall$ *A*. *Proj I* o_{CL} *F A* = *F A* o_{CL} *Proj I*)›
for *F* :: ‹*'a update* $\Rightarrow$ *'b update*›

lemma *compatible-register-invariant-top*[*simp*]:
‹*compatible-register-invariant F* $\top$›

lemma *compatible-register-invariant-bot*[*simp*]:
‹*compatible-register-invariant F* $\bot$›

lemma *compatible-register-invariant-id*:
assumes ‹$\bigwedge$*y*. *I* = *UNIV* $\vee$ *I* = {}›
shows ‹*compatible-register-invariant id* (*ket-invariant I*)›

lemma *compatible-register-invariant-compatible-register*:
assumes ‹*compatible F G*›
shows ‹*compatible-register-invariant F* (*lift-invariant G I*)›

lemma *compatible-register-invariant-chain*[*simp*]:
‹*compatible-register-invariant* (*F o G*) (*lift-invariant F I*) $\longleftrightarrow$ *compatible-register-invariant G I*› **if** [*simp*]: ‹*register F*›

Allows to decompose the preservation of an invariant into a part that is preserved inside a register, and a part outside of it.

lemma *preserves-register*:
  fixes F :: ‹$'a$ *update* $\Rightarrow$ $'b$ *update*›
  assumes *pres*: ‹*preserves* U' I' J' ε›
  assumes *reg*[*register*]: ‹*register* F›
  assumes *compat*: ‹*compatible-register-invariant* F K›
  assumes FU': ‹$\forall\,\psi\in$*space-as-set* I. F U' $*_V$ ψ = U $*_V$ ψ›
  assumes FI'-I: ‹*lift-invariant* F $I' \geq I$›
  assumes *KI*: ‹$K \geq I$›
  assumes $FJ'K$-I: ‹*lift-invariant* F $J' \sqcap K \leq J$›
  shows ‹*preserves* U I J ε›

lemma *preserves-top*[*simp*]: ‹$\varepsilon \geq 0 \Longrightarrow$ *preserves* U I $\top$ ε›

lemma *preserves-bot*[*simp*]: ‹$\varepsilon \geq 0 \Longrightarrow$ *preserves* U $\bot$ J ε›

lemma *preserves-0*[*simp*]: ‹$\varepsilon \geq 0 \Longrightarrow$ *preserves* 0 I J ε›

Tensor product of two invariants: The invariant that requires the first part of the system to satisfy invariant I and the second to satisfy J.

definition ‹*tensor-invariant* I J = *ccspan* $\{x \otimes_s y \mid x\ y.\ x \in$ *space-as-set* $I \wedge y \in$ *space-as-set* $J\}$›

lemma *tensor-invariant-via-Proj*: ‹*tensor-invariant* I J = (*Proj* I $\otimes_o$ *Proj* J) $*_S$ $\top$›

lemma *tensor-invariant-mono-left*: ‹$I \leq I' \Longrightarrow$ *tensor-invariant* I J $\leq$ *tensor-invariant* I' J›

lemma *swap-tensor-invariant*[*simp*]: ‹*swap-ell2* $*_S$ *tensor-invariant* I J = *tensor-invariant* J I›

lemma *tensor-invariant-SUP-left*: ‹*tensor-invariant* (*SUP* $x\in X$. I x) J = (*SUP* $x\in X$. *tensor-invariant* (I x) J)›

lemma *tensor-invariant-SUP-right*: ‹*tensor-invariant* I (*SUP* $x\in X$. J x) = (*SUP* $x\in X$. *tensor-invariant* I (J x))›

lemma *tensor-invariant-bot-left*[*simp*]: ‹*tensor-invariant* $\bot$ J = $\bot$›

lemma *tensor-invariant-bot-right*[*simp*]: ‹*tensor-invariant* I $\bot$ = $\bot$›

lemma *tensor-invariant-Sup-left*: ‹*tensor-invariant* (*Sup* *II*) J = (*SUP* $I\in$*II*. *tensor-invariant* I J)›

lemma *tensor-invariant-Sup-right*: ‹*tensor-invariant* I (*Sup* *JJ*) = (*SUP* $J\in$*JJ*. *tensor-invariant* I J)›

lemma *tensor-invariant-sup-left*: ‹*tensor-invariant* ($I1 \sqcup I2$) J = *tensor-invariant* $I1$ J $\sqcup$ *tensor-invariant* $I2$ J›

lemma *tensor-invariant-sup-right*: ‹*tensor-invariant* I ($J1 \sqcup J2$) = *tensor-invariant* I $J1$ $\sqcup$ *tensor-invariant* I $J2$›

lemma *compatible-register-invariant-compl*: ‹*compatible-register-invariant* F I $\Longrightarrow$ *compatible-register-invariant* F $(-I)$›

lemma *compatible-register-invariant-SUP*:
  assumes [*simp*]: ‹*register* F›
  assumes *compat*: ‹$\bigwedge x.\ x \in X \Longrightarrow$ *compatible-register-invariant* F (I x)›
  shows ‹*compatible-register-invariant* F (*SUP* $x\in X$. I x)›

lemma *compatible-register-invariant-INF*:
  assumes [*simp*]: ‹*register F*›
  assumes *compat*: ‹$\bigwedge x.\ x \in X \Longrightarrow$ *compatible-register-invariant F* (*I x*)›
  shows ‹*compatible-register-invariant F* (*INF x*∈*X*. *I x*)›

lemma *compatible-register-invariant-Sup*:
  assumes ‹*register F*›
  assumes ‹$\bigwedge I.\ I{\in}II \Longrightarrow$ *compatible-register-invariant F I*›
  shows ‹*compatible-register-invariant F* (*Sup II*)›

lemma *compatible-register-invariant-Inf*:
  assumes ‹*register F*›
  assumes ‹$\bigwedge I.\ I{\in}II \Longrightarrow$ *compatible-register-invariant F I*›
  shows ‹*compatible-register-invariant F* (*Inf II*)›

lemma *compatible-register-invariant-inter*:
  assumes ‹*register F*›
  assumes ‹*compatible-register-invariant F I*›
  assumes ‹*compatible-register-invariant F J*›
  shows ‹*compatible-register-invariant F* ($I \sqcap J$)›

lemma *compatible-register-invariant-pair*:
  assumes ‹*compatible-register-invariant F I*›
  assumes ‹*compatible-register-invariant G I*›
  shows ‹*compatible-register-invariant* (*F*;*G*) *I*›

lemma *compatible-register-invariant-tensor*:
  assumes [*register*]: ‹*register F*› ‹*register G*›
  assumes ‹*compatible-register-invariant F I*›
  assumes ‹*compatible-register-invariant G J*›
  shows ‹*compatible-register-invariant* ($F \otimes_r G$) ($I \otimes_S J$)›

lemma *compatible-register-invariant-image-shrinks*:
  assumes ‹*compatible-register-invariant F I*›
  shows ‹$F\ U *_S I \le I$›

lemma *sum-eq-SUP-ccsubspace*:
  fixes *I* :: ‹$'a \Rightarrow\ 'b$::*complex-normed-vector ccsubspace*›
  assumes ‹*finite X*›
  shows ‹($\sum x{\in}X.\ I\ x$) = (*SUP x*∈*X*. *I x*)›

Variant of *invariant-splitting* (see there) that allows the operation that is applied to depend on the state of some other register.

lemma *inv-split-reg*:
  fixes *X* :: ‹$'x$ *update* ⇒ $'m$ *update*› — register containing the index for the unitary
    and *Y* :: ‹$'z \Rightarrow\ 'y$ *update* ⇒ $'m$ *update*› — register on which the unitary operates
    and *K* :: ‹$'z \Rightarrow\ 'm$ *ell2 ccsubspace*› — additional invariants
    and *M* :: ‹$'z$ *set*›

  assumes *U1-U*: ‹$\bigwedge z\ \psi.\ z{\in}M \Longrightarrow \psi \in$ *space-as-set* (*K z*) $\Longrightarrow$ (*Y z* (*U1 z*)) $*_V \psi = U *_V \psi$›
  assumes *pres-I1*: ‹$\bigwedge z.\ z{\in}M \Longrightarrow$ *preserves* (*U1 z*) (*I1 z*) (*J1 z*) ε›
  assumes *I-leq*: ‹$I \le$ (*SUP z*∈*M*. *K z* ⊓ *lift-invariant* (*Y z*) (*I1 z*))›

assumes *J-geq*: ‹$\bigwedge z.\ z{\in}M \Longrightarrow J \geq K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (J1\ z)$›
assumes *YK*: ‹$\bigwedge z.\ z{\in}M \Longrightarrow$ *compatible-register-invariant* $(Y\ z)\ (K\ z)$›
assumes *regY*: ‹$\bigwedge z.\ z{\in}M \Longrightarrow$ *register* $(Y\ z)$›
assumes *orthoK*: ‹$\bigwedge z\ z'.\ z{\in}M \Longrightarrow z'{\in}M \Longrightarrow z \neq z' \Longrightarrow$ *orthogonal-spaces* $(K\ z)\ (K\ z')$›
assumes ‹$\varepsilon \geq 0$›
assumes [*iff*]: ‹*finite M*›
shows ‹*preserves* $U\ I\ J\ \varepsilon$›

lemma *Proj-ket-invariant-ket*: ‹*Proj* (*ket-invariant* X) $*_V$ *ket* i = (*if* $i{\in}X$ *then ket* i *else* 0)›

lemma *lift-invariant-function-at-ket-inv*: ‹*lift-invariant* (*function-at* x) (*ket-invariant* I) = *ket-invariant* $\{f.\ f\ x \in I\}$›

lemma *ket-invariant-prod*: ‹*Proj* (*ket-invariant* $(A \times B)$) = *Proj* (*ket-invariant* A) $\otimes_o$ *Proj* (*ket-invariant* B)›

lemma *lift-Fst-inv*: ‹*lift-invariant Fst* $I = I \otimes_S \top$›
lemma *lift-Snd-inv*: ‹*lift-invariant Snd* $I = \top \otimes_S I$›

lemma *lift-Snd-ket-inv*: ‹*lift-invariant Snd* (*ket-invariant* I) = *ket-invariant* $(UNIV \times I)$›
lemma *lift-Fst-ket-inv*: ‹*lift-invariant Fst* (*ket-invariant* I) = *ket-invariant* $(I \times UNIV)$›

lemma *lift-inv-prod*:
assumes [*simp*]: ‹*compatible F G*›
shows ‹*lift-invariant* $(F;G)$ (*ket-invariant* $(I \times J)$) =
lift-invariant F (*ket-invariant* I) $\sqcap$ *lift-invariant G* (*ket-invariant* J)›

lemma *lift-inv-tensor*:
assumes [*register*]: ‹*register F*› ‹*register G*›
shows ‹*lift-invariant* $(F \otimes_r G)$ (*ket-invariant* $(I \times J)$) =
lift-invariant F (*ket-invariant* I) $\otimes_S$ *lift-invariant G* (*ket-invariant* J)›

lemma *lift-invariant-sup*:
fixes F :: ‹$('a\ ell2 \Rightarrow_{CL} 'a\ ell2) \Rightarrow ('b\ ell2 \Rightarrow_{CL} 'b\ ell2)$›
assumes [*simp*]: ‹*register F*›
shows ‹*lift-invariant F* $(I \sqcup J)$ = *lift-invariant F I* $\sqcup$ *lift-invariant F J*›

lemma *lift-invariant-SUP*:
fixes F :: ‹$('a\ ell2 \Rightarrow_{CL} 'a\ ell2) \Rightarrow ('b\ ell2 \Rightarrow_{CL} 'b\ ell2)$›
assumes ‹*register F*›
shows ‹*lift-invariant F* $(SUP\ x{\in}X.\ I\ x)$ = $(SUP\ x{\in}X.$ *lift-invariant F* $(I\ x))$›

lemma *lift-invariant-compl*: ‹*lift-invariant R* $(-\ U) = -$ *lift-invariant R U*› **if** ‹*register R*›

lemma *lift-invariant-INF*:
assumes ‹*register F*›
shows ‹*lift-invariant F* $(\bigsqcap x{\in}A.\ I\ x) = (\bigsqcap x{\in}A.$ *lift-invariant F* $(I\ x))$›

lemma *lift-invariant-inf*:
assumes ‹*register F*›
shows ‹*lift-invariant F* $(I \sqcap J)$ = *lift-invariant F I* $\sqcap$ *lift-invariant F J*›

lemma *lift-invariant-mono*:
  assumes ‹*register F*›
  assumes ‹$I \leq J$›
  shows ‹*lift-invariant F I* $\leq$ *lift-invariant F J*›

lemma *lift-inv-prod′*:
  fixes F :: ‹(′*a ell2* $\Rightarrow_{CL}$ ′*a ell2*) $\Rightarrow$ (′*c ell2* $\Rightarrow_{CL}$ ′*c ell2*)›
  fixes G :: ‹(′*b ell2* $\Rightarrow_{CL}$ ′*b ell2*) $\Rightarrow$ (′*c ell2* $\Rightarrow_{CL}$ ′*c ell2*)›
  assumes [*simp*]: ‹*compatible F G*›
  shows ‹*lift-invariant* (F;G) (*ket-invariant I*) =
    (*SUP* (*x*,*y*)$\in$*I*. *lift-invariant F* (*ket-invariant* $\{x\}$) $\sqcap$ *lift-invariant G* (*ket-invariant* $\{y\}$))›

lemma *lift-inv-tensor′*:
  assumes [*register*]: ‹*register F*› ‹*register G*›
  shows ‹*lift-invariant* ($F \otimes_r G$) (*ket-invariant I*) =
    (*SUP* (*x*,*y*)$\in$*I*. *lift-invariant F* (*ket-invariant* $\{x\}$) $\otimes_S$ *lift-invariant G* (*ket-invariant* $\{y\}$))›

lemma *classical-operator-ket-invariant*:
  assumes ‹*inj-map f*›
  shows ‹*classical-operator f* $*_S$ *ket-invariant I* = *ket-invariant* (*Some* −‘ *f* ‘ *I*)›

lemma *Proj-ket-invariant-singleton*: ‹*Proj* (*ket-invariant* $\{x\}$) = *selfbutter* (*ket x*)›

lemma *lift-inv-classical*:
  fixes F :: ‹′*a ell2* $\Rightarrow_{CL}$ ′*a ell2* $\Rightarrow$ ′*b ell2* $\Rightarrow_{CL}$ ′*b ell2*› **and** f :: ‹′*a* $\times$ ′*c* $\Rightarrow$ ′*b*›
  assumes [*register*]: ‹*register F*›
  assumes ‹*inj f*›
  assumes ‹$\bigwedge$*x*::′*a*. $x \in I \Longrightarrow$ *F* (*selfbutter* (*ket x*)) = *sandwich* (*classical-operator* (*Some o f*)) (*selfbutter* (*ket x*) $\otimes_o$ *id-cblinfun*)›
  shows ‹*lift-invariant F* (*ket-invariant I*) = *ket-invariant* (*f* ‘ (*I* $\times$ *UNIV*))›

lemma *register-image-lift-invariant*:
  assumes ‹*register F*›
  assumes ‹*isometry U*›
  shows ‹*F U* $*_S$ *lift-invariant F I* = *lift-invariant F* (*U* $*_S$ *I*)›

lemma *ell2-sum-ket-ket-invariant*:
  fixes ψ :: ‹′*a ell2*›
  assumes ‹$\psi \in$ *space-as-set* (*ket-invariant X*)›
  shows ‹ψ = ($\sum_\infty i{\in}X$. *Rep-ell2* ψ *i* $*_C$ *ket i*)›

lemma *compatible-register-invariant-Fst-comp*:
  fixes I :: ‹(′*a* $\times$ ′*b*) *set*›
  assumes [*simp*]: ‹*register F*›
  assumes ‹$\bigwedge$*y*. *compatible-register-invariant F* (*ket-invariant* ((λx. (*x*,*y*)) −‘ *I*))›
  shows ‹*compatible-register-invariant* (*Fst o F*) (*ket-invariant I*)›

lemma *compatible-register-invariant-Fst*:

assumes ‹$\bigwedge y$. ((λx. (x,y)) −‘ I) = *UNIV* ∨ ((λx. (x,y)) −‘ I) = {}›
shows ‹*compatible-register-invariant Fst* (*ket-invariant I*)›

lemma *compatible-register-invariant-Snd-comp*:
fixes I :: ‹($'a \times 'b$) *set*›
assumes [*simp*]: ‹*register F*›
assumes ‹$\bigwedge x$. *compatible-register-invariant F* (*ket-invariant* ((λy. (x,y)) −‘ I))›
shows ‹*compatible-register-invariant* (*Snd o F*) (*ket-invariant I*)›

lemma *compatible-register-invariant-Snd*:
assumes ‹$\bigwedge x$. ((λy. (x,y)) −‘ I) = *UNIV* ∨ ((λy. (x,y)) −‘ I) = {}›
shows ‹*compatible-register-invariant Snd* (*ket-invariant I*)›

lemma *compatible-register-invariant-Fst-tensor*[*simp*]:
shows ‹*compatible-register-invariant Fst* ($\top \otimes_S I$)›

lemma *compatible-register-invariant-Snd-tensor*[*simp*]:
shows ‹*compatible-register-invariant Snd* ($I \otimes_S \top$)›

lemma *compatible-register-invariant-sandwich-comp*:
fixes U :: ‹$'a$ *ell2* $\Rightarrow_{CL}$ $'b$ *ell2*›
assumes [*simp*]: ‹*unitary U*›
assumes ‹*compatible-register-invariant F* ($U* *_S I$)›
shows ‹*compatible-register-invariant* (*sandwich U o F*) *I*›

lemma *compatible-register-invariant-function-at-comp*:
assumes [*simp*]: ‹*register F*›
assumes ‹$\bigwedge z$. *compatible-register-invariant F* (*ket-invariant* {*f x* |*f*. *f* ∈ *I* ∧ *z*(*x* := *undefined*) = *f*(*x* := *undefined*)})›
shows ‹*compatible-register-invariant* (*function-at x o F*) (*ket-invariant I*)›

lemma *compatible-register-invariant-function-at*:
assumes ‹$\bigwedge f\ y$. $f \in I \Longrightarrow f(x{:=}y) \in I$›
shows ‹*compatible-register-invariant* (*function-at x*) (*ket-invariant I*)›

The following lemma allows show that an invariant is preserved across several consecutive operations. Usually, *norm V* and *norm U* $\leq$ *1*, so the lemma essentially says that the errors are additive.

lemma *preserves-trans*[*trans*]:
assumes *presU*: ‹*preserves U I J* ε›
assumes *presV*: ‹*preserves V J K* δ›
shows ‹*preserves* ($V\ o_{CL}\ U$) *I K* (*norm V* $*\ \varepsilon$ + *norm U* $*\ \delta$)›

An operation that operates on a register that is outside the invariant preserves the invariant perfectly.

lemma *preserves-compatible*:
assumes *compat*: ‹*compatible-register-invariant F I*›
assumes ‹$\varepsilon \geq 0$›
shows ‹*preserves* (*F U*) *I I* ε›

lemma *Proj-ket-invariant-butterfly*: ‹*Proj* (*ket-invariant* {*x*}) = *selfbutter* (*ket x*)›

lemma *ket-in-ket-invariantI*: ‹*ket x* ∈ *space-as-set* (*ket-invariant I*)› **if** ‹*x* ∈ *I*›

lemma *cblinfun-image-ket-invariant-leqI*:
  assumes ‹$\bigwedge x.\ x \in I \Longrightarrow U *_V$ *ket* $x \in$ *space-as-set* J›
  shows ‹$U *_S$ *ket-invariant* $I \leq J$›

lemma *preserves0I*: ‹*preserves* $U\ I\ J\ 0 \longleftrightarrow U *_S I \leq J$›

lemma *lift-invariant-id*[*simp*]: ‹*lift-invariant id* $I = I$›

lemma *lift-invariant-pair-tensor*:
  assumes ‹*compatible* $X\ Y$›
  shows ‹*lift-invariant* $(X;Y)\ (I \otimes_S J) =$ *lift-invariant* $X\ I \sqcap$ *lift-invariant* $Y\ J$›

lemma *lift-invariant-tensor-tensor*:
  assumes [*register*]: ‹*register* X› ‹*register* Y›
  shows ‹*lift-invariant* $(X \otimes_r Y)\ (I \otimes_S J) =$ *lift-invariant* $X\ I \otimes_S$ *lift-invariant* $Y\ J$›

lemma *orthogonal-spaces-lift-invariant*[*simp*]:
  assumes ‹*register* Q›
  shows ‹*orthogonal-spaces* (*lift-invariant* $Q\ S$) (*lift-invariant* $Q\ T$) $\longleftrightarrow$ *orthogonal-spaces* $S\ T$›

3.2 Distance from invariants

definition *dist-inv* **where** ‹*dist-inv* $R\ I\ \psi =$ *norm* $(R\ (Proj\ (-I)) *_V \psi)$›
  for R :: ‹$('a\ ell2 \Rightarrow_{CL} 'a\ ell2) \Rightarrow ('b\ ell2 \Rightarrow_{CL} 'b\ ell2)$›
definition *dist-inv-avg* **where** ‹*dist-inv-avg* $R\ I\ \psi =$ *sqrt* $((\sum x{\in}UNIV.\ (\textit{dist-inv}\ R\ (I\ x)\ (\psi\ x))^2)\ /$ $CARD('x))$› **for** ψ :: ‹$'x$::*finite* $\Rightarrow$ -›

lemma *dist-inv-pos*[*iff*]: ‹*dist-inv* $R\ I\ \psi \geq 0$›
lemma *dist-inv-avg-pos*[*iff*]: ‹*dist-inv-avg* $R\ I\ \psi \geq 0$›

lemma *dist-inv-0-iff*:
  assumes ‹*register* R›
  shows ‹*dist-inv* $R\ I\ \psi = 0 \longleftrightarrow \psi \in$ *space-as-set* (*lift-invariant* $R\ I$)›

lemma *dist-inv-avg-0-iff*:
  assumes ‹*register* R›
  shows ‹*dist-inv-avg* $R\ I\ \psi = 0 \longleftrightarrow (\forall h.\ \psi\ h \in$ *space-as-set* (*lift-invariant* $R\ (I\ h)$))›

lemma *dist-inv-mono*:
  assumes ‹$I \leq J$›
  assumes [*register*]: ‹*register* Q›
  shows ‹*dist-inv* $Q\ J\ \psi \leq$ *dist-inv* $Q\ I\ \psi$›

lemma *dist-inv-avg-mono*:
  assumes ‹$\bigwedge h.\ I\ h \leq J\ h$›
  assumes [*register*]: ‹*register* Q›
  shows ‹*dist-inv-avg* $Q\ J\ \psi \leq$ *dist-inv-avg* $Q\ I\ \psi$›

lemma *dist-inv-Fst-tensor*:
  assumes ‹*norm* $\varphi = 1$›
  shows ‹*dist-inv* $(Fst\ o\ R)\ I\ (\psi \otimes_s \varphi) =$ *dist-inv* $R\ I\ \psi$›

lemma *dist-inv-avg-Fst-tensor*:

assumes ‹$\bigwedge h.$ *norm* (φ *h*) = *1*›
shows ‹*dist-inv-avg* (*Fst o R*) *I* ($\lambda h.\ \psi\ h \otimes_s \varphi\ h$) = *dist-inv-avg R I* ψ›

lemma *dist-inv-register-rewrite*:
assumes ‹*register Q*› **and** ‹*register R*›
assumes ‹*lift-invariant Q I* = *lift-invariant R J*›
shows ‹*dist-inv Q I* ψ = *dist-inv R J* ψ›

lemma *dist-inv-avg-register-rewrite*:
assumes ‹*register Q*› **and** ‹*register R*›
assumes ‹$\bigwedge h.$ *lift-invariant Q* (*I h*) = *lift-invariant R* (*J h*)›
shows ‹*dist-inv-avg Q I* ψ = *dist-inv-avg R J* ψ›

lemma *distance-from-inv-avg0I*:
‹*dist-inv-avg Q I* ψ = *0* $\longleftrightarrow$ ($\forall h.$ *dist-inv Q* (*I h*) (ψ *h*) = *0*)› **for** *h* :: ‹*'h::finite*› **and** ψ :: ‹*'h* $\Rightarrow$ -›

lemma *dist-inv-apply*:
assumes [*register*]: ‹*register Q*› ‹*register S*›
assumes [*iff*]: ‹*unitary U*›
assumes *QSR*: ‹*Q o S* = *R*›
shows ‹*dist-inv Q I* (*R U* $*_V$ ψ) = *dist-inv Q* (*S U** $*_S$ *I*) ψ›

lemma *dist-inv-apply-iff*:
assumes [*register*]: ‹*register Q*›
assumes [*iff*]: ‹*unitary U*›
shows ‹*dist-inv Q I* (*Q U* $*_V$ ψ) = *dist-inv Q* (*U** $*_S$ *I*) ψ›

lemma *dist-inv-avg-apply*:
assumes [*register*]: ‹*register Q*› ‹*register S*›
assumes [*iff*]: ‹$\bigwedge h.$ *unitary* (*U h*)›
assumes ‹*Q o S* = *R*›
shows ‹*dist-inv-avg Q I* ($\lambda h.$ *R* (*U h*) $*_V$ ψ *h*) = *dist-inv-avg Q* ($\lambda h.$ *S* (*U h*)* $*_S$ *I h*) ψ›

lemma *dist-inv-avg-apply-iff*:
assumes [*register*]: ‹*register Q*›
assumes [*iff*]: ‹$\bigwedge h.$ *unitary* (*U h*)›
shows ‹*dist-inv-avg Q I* ($\lambda h.$ *Q* (*U h*) $*_V$ ψ *h*) = *dist-inv-avg Q* ($\lambda h.$ *U h** $*_S$ *I h*) ψ›

lemma *dist-inv-intersect-onesided*:
assumes ‹*compatible-invariants I J*›
assumes ‹*register Q*›
assumes ‹*dist-inv Q I* ψ = *0*›
shows ‹*dist-inv Q* (*J* $\sqcap$ *I*) ψ = *dist-inv Q J* ψ›

lemma *dist-inv-avg-intersect*:

assumes ‹$\bigwedge h$. *compatible-invariants* $(I\ h)$ $(J\ h)$›
assumes ‹*register* Q›
assumes ‹*dist-inv-avg* $Q\ I\ \psi = 0$›
shows ‹*dist-inv-avg* Q $(\lambda h.\ J\ h \sqcap I\ h)$ ψ = *dist-inv-avg* $Q\ J\ \psi$›

lemma *dist-inv-avg-const*: ‹*dist-inv-avg* Q $(\lambda\text{-}.\ I)$ $(\lambda\text{-}.\ \psi)$ = *dist-inv* $Q\ I\ \psi$›

lemma *register-plus*:
assumes ‹*register* Q›
shows ‹$Q\ (a + b) = Q\ a + Q\ b$›

lemma *compatible-invariants-uminus-left*[*simp*]: ‹*compatible-invariants* $(-I)$ $J \longleftrightarrow$ *compatible-invariants* $I\ J$›

lemma *compatible-invariants-uminus-right*[*simp*]: ‹*compatible-invariants* I $(-J) \longleftrightarrow$ *compatible-invariants* $I\ J$›

lemma *compatible-invariants-sup*: ‹*Proj* $(A \sqcup B)$ = *Proj* A + *Proj* B − *Proj* A o_{CL} *Proj* B› **if** ‹*compatible-invariants* $A\ B$›

lemma *compatible-invariants-sym*: ‹*compatible-invariants* $S\ T \longleftrightarrow$ *compatible-invariants* $T\ S$›

lemma *compatible-invariants-refl*[*iff*]: ‹*compatible-invariants* $S\ S$›

lemma *compatible-invariants-infI*:
assumes [*iff*]: ‹*compatible-invariants* $S\ U$›
assumes [*iff*]: ‹*compatible-invariants* $S\ T$›
assumes [*iff*]: ‹*compatible-invariants* $T\ U$›
shows ‹*compatible-invariants* S $(T \sqcap U)$›

lemma *compatible-invariants-supI*:
assumes [*iff*]: ‹*compatible-invariants* $S\ U$›
assumes [*iff*]: ‹*compatible-invariants* $S\ T$›
assumes [*iff*]: ‹*compatible-invariants* $T\ U$›
shows ‹*compatible-invariants* S $(T \sqcup U)$›

lemma *compatible-invariants-inf-sup-distrib1*:
fixes $S\ T\ U$ :: ‹*'a::chilbert-space ccsubspace*›
assumes ‹*compatible-invariants* $S\ U$›
assumes ‹*compatible-invariants* $S\ T$›
assumes ‹*compatible-invariants* $T\ U$›
shows ‹$S \sqcap (T \sqcup U) = (S \sqcap T) \sqcup (S \sqcap U)$›

lemma *compatible-invariants-inf-sup-distrib2*:
fixes $S\ T\ U$ :: ‹*'a::chilbert-space ccsubspace*›
assumes [*iff*]: ‹*compatible-invariants* $S\ U$›
assumes [*iff*]: ‹*compatible-invariants* $S\ T$›
assumes [*iff*]: ‹*compatible-invariants* $T\ U$›
shows ‹$(T \sqcup U) \sqcap S = (T \sqcap S) \sqcup (U \sqcap S)$›

lemma *compatible-invariants-sup-inf-distrib1*:

fixes S T U :: ‹$'a$::*chilbert-space ccsubspace*›
assumes ‹*compatible-invariants* S U›
assumes ‹*compatible-invariants* S T›
assumes ‹*compatible-invariants* T U›
shows ‹$S \sqcup (T \sqcap U) = (S \sqcup T) \sqcap (S \sqcup U)$›

lemma *compatible-invariants-sup-inf-distrib2*:
fixes S T U :: ‹$'a$::*chilbert-space ccsubspace*›
assumes ‹*compatible-invariants* S U›
assumes ‹*compatible-invariants* S T›
assumes ‹*compatible-invariants* T U›
shows ‹$(T \sqcap U) \sqcup S = (T \sqcup S) \sqcap (U \sqcup S)$›

lemma *is-orthogonal-Proj-orthogonal-spaces*:
assumes ‹*orthogonal-spaces* S T›
shows ‹*is-orthogonal* (*Proj* $S *_V \psi$) (*Proj* $T *_V \psi$)›

lemma *dist-inv-intersect*:
assumes [*register*]: ‹*register* Q›
assumes [*iff*]: ‹*compatible-invariants* I J›
shows ‹*dist-inv* Q $(I \sqcap J)$ $\psi \leq$ *sqrt* ((*dist-inv* Q I $\psi)^2$ + (*dist-inv* Q J $\psi)^2$)›

3.3 Preservation of invariants

lemma *preserves-lift-invariant*:
assumes [*register*]: ‹*register* Q›
shows ‹*preserves* (Q U) (*lift-invariant* Q I) (*lift-invariant* Q J) $\varepsilon \longleftrightarrow$ *preserves* U I J ε›

lemma *dist-inv-leq-if-preserves*:
assumes *pres*: ‹*preserves* U (*lift-invariant* S J) (*lift-invariant* R I) γ›
assumes [*register*]: ‹*register* S› ‹*register* R›
shows ‹*dist-inv* R I $(U *_V \psi) \leq$ *norm* $U *$ *dist-inv* S J $\psi + \gamma *$ *norm* ψ›

lemma *dist-inv-preservesI*:
assumes ‹*dist-inv* S J $\psi \leq \varepsilon$›
assumes *pres*: ‹*preserves* U (*lift-invariant* S J) (*lift-invariant* R I) γ›
assumes ‹*norm* $U \leq 1$›
assumes ‹*norm* $\psi \leq 1$›
assumes ‹$\gamma + \varepsilon \leq \delta$›
assumes [*register*]: ‹*register* S› ‹*register* R›
shows ‹*dist-inv* R I $(U *_V \psi) \leq \delta$›

lemma *dist-inv-apply-compatible*:
assumes ‹*compatible* Q R›
shows ‹*dist-inv* Q I $(R\ U *_V \psi) \leq$ *norm* $U *$ *dist-inv* Q I ψ›

lemma *dist-inv-avg-apply-compatible*:
assumes ‹$\bigwedge h.$ *compatible* Q $(R\ h)$›
shows ‹*dist-inv-avg* Q I $(\lambda h.\ R\ h\ (U\ h) *_V \psi\ h) \leq$ (*MAX* h. *norm* $(U\ h)$) $*$ *dist-inv-avg* Q I ψ›

end

4 *CO-Operations* Definition of the compressed oracle and related unitaries

theory *CO-Operations* **imports**
Complex-Bounded-Operators.Complex-L2
HOL.Map
Registers.Quantum-Extra2

Misc-Compressed-Oracle
Function-At
begin

unbundle *cblinfun-syntax*

4.1 *function-oracle* - Querying a fixed function

definition *function-oracle* :: ‹(′x ⇒ ′y::ab-group-add) ⇒ ((′x × ′y) ell2 $\Rightarrow_{CL}$ (′x × ′y) ell2)› **where**
‹*function-oracle h = classical-operator* (λ(x,y). *Some* (x, y + h x))›

lemma *function-oracle-apply*: ‹*function-oracle h* (*ket* (x, y)) = *ket* (x, y + h x)›

lemma *function-oracle-adj-apply*: ‹*function-oracle h*∗ $*_V$ *ket* (x, y) = *ket* (x, y − h x)›

lemma *unitary-function-oracle*[*iff*]: ‹*unitary* (*function-oracle h*)›

lemma *norm-function-oracle*[*simp*]: ‹*norm* (*function-oracle h*) = *1*›

lemma *function-oracle-adj*[*simp*]: ‹*function-oracle h*∗ = *function-oracle* (λx. − h x)› **for** *h* :: ‹′x ⇒ ′y::ab-group-add›

4.2 Setup for compressed oracles

consts *trafo* :: ‹′a ell2 $\Rightarrow_{CL}$ ′a::{zero,finite} ell2›
specification (*trafo*)
unitary-trafo[*simp*]: ‹*unitary trafo*›
trafo-0[*simp*]: ‹*trafo* $*_V$ *ket 0* = *uniform-superpos UNIV*›

Set of total functions

definition ‹*total-functions* = {f::′x⇀′y. *None* ∉ *range f*}›

lemma *total-functions-def2*: ‹*total-functions* = (*comp Some*) ‘ *UNIV*›

lemma *total-functions-def3*: ‹*total-functions* = {f. *dom f* = *UNIV*}›

lemma *card-total-functions*: ‹*card* (*total-functions* :: (′x ⇒ ′y *option*) *set*) = *CARD*(′y) ^ *CARD*(′x::*finite*)›

abbreviation *superpos-total* :: ‹(′x::finite ⇒ ′y::finite option) ell2› **where** ‹*superpos-total* ≡ *uniform-superpos total-functions*›

Sets up the locale for defining the compressed oracle. We use a locale because the compressed oracle can depend on some arbitrary unitary *trafo*. The choice of *trafo* usually doesn't matter; in this case the default transformation *trafo* above can be used.

locale *compressed-oracle* =
  fixes *dummy-constant* :: ‹(*'x::finite* × *'y::{finite,ab-group-add}*) *itself*›
  fixes *trafo* :: ‹*'y::{finite,ab-group-add} ell2* $\Rightarrow_{CL}$ *'y ell2*›
  assumes *unitary-trafo*[*simp*]: ‹*unitary trafo*›
  assumes *trafo-0*: ‹*trafo* $*_V$ *ket 0* = *uniform-superpos UNIV*›
  assumes *y-cancel*[*simp*]: ‹(*y::'y*) + *y* = *0*›
begin

definition *dummy2* :: ‹*'y update* ⇒ (*'y set* ⇒ *nat*) ⇒ (*'y set* ⇒ *nat*)›
  where ‹*dummy2 x y* = *y*›
definition *N-def0*: ‹*N* = *dummy2 trafo card UNIV*›

N is the cardinality of the oracle outputs. (Intuitively, $N = 2^n$ for an *n*-bit output.

lemma *N-def*: ‹*N* = *CARD*(*'y*)›

lemma *Nneq0*[*iff*]: ‹*N* ≠ *0*›

definition ‹α = *complex-of-real* (*1* / *sqrt* (*of-nat N*))›
  — We use this term very often, so this abbreviation comes in handy.

lemma (**in** *compressed-oracle*) *uminus-y*[*simp*]: ‹− *y* = *y*› **for** *y* :: *'y*

4.3 *switch0* - Operator exchanging *ket* (*Some 0*) and *ket None*

switch0 maps *ket None* to *ket* (*Some 0*) and vice versa. It leaves all other *ket* (*Some y*) unchanged.

definition *switch0* :: ‹*'y option update*› **where**
  ‹*switch0* = *classical-operator* (*Some o Fun.swap* (*Some 0*) *None id*)›

lemma *switch0-None*[*simp*]: ‹*switch0* $*_V$ *ket None* = *ket* (*Some 0*)›

lemma *switch0-0*[*simp*]: ‹*switch0* $*_V$ *ket* (*Some 0*) = *ket None*›

lemma *switch0-other*: ‹*switch0* $*_V$ *ket* (*Some x*) = *ket* (*Some x*)› **if** ‹*x* ≠ *0*›

lemma *unitary-switch0*[*simp*]: ‹*unitary switch0*›

lemma *switch0-adj*[*simp*]: ‹*switch0*∗ = *switch0*›

4.4 *compress1* - Operator to compress a single RO-output

This unitary maps *ket None* onto the uniform superposition of all *ket* (*Some y*) and vice versa, and leaves everything orthogonal to these unchanged.

This is the operation that deals with compressing a single oracle output.

definition *compress1* :: ‹*'y option ell2* $\Rightarrow_{CL}$ *'y option ell2*› **where**
  ‹*compress1* = *lift-op trafo* o_{CL} *switch0* o_{CL} (*lift-op trafo*)∗›

lemma *uniform-superpos-y-sum*: ‹$uniform\text{-}superpos\ UNIV = (\sum d{\in}UNIV.\ \alpha *_C ket\ (d{::}'y))$›

lemma *compress1-None*[*simp*]: ‹$compress1 *_V ket\ None = (\sum d{\in}UNIV.\ \alpha *_C ket\ (Some\ d))$›

lemma *compress1-Some*[*simp*]: ‹$compress1 *_V ket\ (Some\ d) =$
$ket\ (Some\ d) - (\sum d{\in}UNIV.\ \alpha^2 *_C ket\ (Some\ d)) + \alpha *_C ket\ None$›

lemma *unitary-compress1*[*simp*]: ‹*unitary compress1*›

lemma *compress1-adj*[*simp*]: ‹$compress1* = compress1$›

lemma *compress1-square*: ‹$compress1\ o_{CL}\ compress1 = id\text{-}cblinfun$›

4.5 *compress* - Operator for compressing the RO

This is the unitary that maps between the compressed representation of the random oracle (in which the initial state is *ket* (λ-. *None*)) and the uncompressed one (in which the initial state is the uniform superposition of all total functions).

It works by simply applying *compress1* to each output separately.

definition *compress* :: ‹$('x \rightharpoonup 'y)$ *update*›
where ‹*compress* = *apply-every UNIV* (λ-. *compress1*)›

lemma *unitary-compress*[*simp*]: ‹*unitary compress*›

lemma *compress-selfinverse*: ‹$compress\ o_{CL}\ compress = id\text{-}cblinfun$›

lemma *compress-adj*: ‹$compress* = compress$›

lemma *compress-empty*: ‹$compress *_V ket\ Map.empty = superpos\text{-}total$›

4.6 *standard-query1* - Operator for uncompressed query of a single RO-output

We define the operation *standard-query1* of querying the oracle, but first in the special case of an oracle that has no input register. That is, the oracle state consists of just one output value (or *None*) and this value is simply added to the query output register.

Roughly speaking, it thus is the unitary $|y, h\rangle \mapsto |y \oplus h, h\rangle$. In comparison, a “normal” oracle query would be defined by $|x, y, h\rangle \mapsto |x, y \oplus h(x), h\rangle$.

That is: If one starts with a three-partite state $\psi \otimes_s ket\ 0 \otimes_s superpos\text{-}total$ and keeps performing operations U_i on the parts 1, 2 of the state, interleaved with *standard-query1* invocations on parts 2, 3, this is a simulation of starting with state $\psi \otimes_s 0$ and performing U_i interleaved with invocations of the unitary $|y\rangle \mapsto |y \oplus h\rangle$ on part 2 where h is chosen uniformly at random in the beginning.

When $h = None$, there are various natural choices how to define the behavior of *standard-query1*. This is because intuitively, this should not happen, because this operation intended to be applied to uncompressed oracles which are superpositions of total functions. Yet, due to errors introduced by projecting onto invariants, one can get situations where this is not perfectly the case, so the behavior on *None* matters. Here, we choose to let *standard-query1* be the identity in that case.

definition *standard-query1* :: ‹$('y \times 'y\ option)$ *update*› **where**

‹*standard-query1* = *classical-operator* (*Some o* (λ(*y*,*z*). *case z of None* ⇒ (*y*,*None*) | *Some z'* ⇒ (*y* + *z'*, *z*)))›

The operation *standard-query1'* is defined like *standard-query1* (and the motivation and properties mentioned there also hold here), except that in the case $h = None$ (see discussion for *standard-query1*), instead of being the identify, *standard-query1'* returns the 0-vector (not *ket 0*!). In particular, this operation is not a unitary which can make some things more awkward. But on the plus side, we can achieve better bounds in some situations when using *standard-query1'*.

definition *standard-query1'* :: ‹(*'y* × *'y option*) *update*› **where**
‹*standard-query1'* = *classical-operator* (λ(*y*,*z*). *case z of None* ⇒ *None* | *Some z'* ⇒ *Some* (*y* + *z'*, *z*))›

lemma *standard-query1-Some*[*simp*]: ‹*standard-query1* $*_V$ *ket* (*y*, *Some z*) = *ket* (*y* + *z*, *Some z*)›

lemma *standard-query1-None*[*simp*]: ‹*standard-query1* $*_V$ *ket* (*y*, *None*) = *ket* (*y*, *None*)›

lemma *standard-query1'-Some*[*simp*]: ‹*standard-query1'* $*_V$ *ket* (*y*, *Some z*) = *ket* (*y* + *z*, *Some z*)›

lemma *standard-query1'-None*[*simp*]: ‹*standard-query1'* $*_V$ *ket* (*y*, *None*) = *0*›

lemma *unitary-standard-query1*[*simp*]: ‹*unitary standard-query1*›

lemma *norm-standard-query1'*[*simp*]: ‹*norm standard-query1'* = *1*›

lemma *standard-query1-selfinverse*[*simp*]: ‹*standard-query1* o_{CL} *standard-query1* = *id-cblinfun*›

4.7 *standard-query* - Operator for uncompressed query of the RO

We can now define the operation of querying the (non-compressed) oracle, i.e., the operation $|x, y, h\rangle \mapsto |x, y \oplus h(x), h\rangle$. Most of the work has already been done when defining *standard-query1*. We just need to apply *standard-query1* onto the Y-register and the x-output of the H-register, where x is the content of the X-register (in the computational basis).

The various lemmas below (e.g., *standard-query-ket*) show that this definition actually achieves this.

That is: If one starts with a four-partite state $\psi \otimes_s$ *ket 0* $\otimes_s$ *ket 0* $\otimes_s$ *superpos-total* and keeps performing operations U_i on the parts 1–3 of the state, interleaved with *standard-query* invocations on parts 2–4, this is a simulation of starting with state $\psi \otimes_s 0$ and performing U_i interleaved with invocations of the unitary $|x, y\rangle \mapsto |x, y \oplus h(x)\rangle$ on parts 2, 3 where h is a function chosen uniformly at random in the beginning.

definition *standard-query* :: ‹(*'x* × *'y* × (*'x* ⇀ *'y*)) *ell2* $\Rightarrow_{CL}$ (*'x* × *'y* × (*'x* ⇀ *'y*)) *ell2*› **where**
‹*standard-query* = *controlled-op* (λ*x*. (*Fst*; *Snd o function-at x*) *standard-query1*)›

Analogous to *standard-query* but using the variant *standard-query1'*.

definition *standard-query'* :: ‹(*'x* × *'y* × (*'x* ⇀ *'y*)) *ell2* $\Rightarrow_{CL}$ (*'x* × *'y* × (*'x* ⇀ *'y*)) *ell2*› **where**
‹*standard-query'* = *controlled-op* (λ*x*. (*Fst*; *Snd o function-at x*) *standard-query1'*)›

lemma *standard-query-ket*: ‹*standard-query* $*_V$ (*ket x* $\otimes_s$ ψ) = *ket x* $\otimes_s$ ((*Fst*; *Snd o function-at x*) *standard-query1* $*_V$ ψ)›

lemma *standard-query-ket-full-Some*:

assumes ‹$H\ x = Some\ z$›
shows ‹*standard-query* $*_V$ (*ket* (x,y,H)) = *ket* $(x,\ y + z,\ H)$›

lemma *standard-query-ket-full-None*:
assumes ‹$H\ x = None$›
shows ‹*standard-query* $*_V$ (*ket* (x,y,H)) = *ket* $(x,\ y,\ H)$›

lemma *standard-query'-ket*: ‹*standard-query'* $*_V$ (*ket* $x \otimes_s \psi$) = *ket* $x \otimes_s$ ((*Fst*; *Snd o function-at x*) *standard-query1'* $*_V$ ψ)›

lemma *standard-query'-ket-full-Some*:
assumes ‹$H\ x = Some\ z$›
shows ‹*standard-query'* $*_V$ (*ket* (x,y,H)) = *ket* $(x,\ y + z,\ H)$›

lemma *standard-query'-ket-full-None*:
assumes ‹$H\ x = None$›
shows ‹*standard-query'* $*_V$ (*ket* (x,y,H)) = *0*›

lemma *standard-query-selfinverse*[*simp*]: ‹*standard-query* o_{CL} *standard-query* = *id-cblinfun*›

lemma *unitary-standard-query*[*simp*]: ‹*unitary standard-query*›

lemma *contracting-standard'-query*[*simp*]: ‹*norm standard-query'* = *1*›

4.8 *query1* - Query the compressed oracle at a single output

Before we formulate the compressed oracle itself, we define a scaled down version where the function in the oracle has only a single output (and there's no input register). Cf. *standard-query1*. This is done by decompressing the oracle register, applying *standard-query1*, and then recompressing the oracle register.

That is: If one starts with a three-partite state $\psi \otimes_s$ *ket 0* $\otimes_s$ *ket None* and keeps performing operations U_i on the parts 1, 2 of the state, interleaved with *query1* invocations on parts 2, 3, this is a simulation of starting with state $\psi \otimes_s$ *0* and performing U_i interleaved with invocations of the unitary $|y\rangle \mapsto |y \oplus h\rangle$ on part 2 where h is chosen uniformly at random in the beginning.

definition *query1* **where** ‹*query1* = *Snd compress1* o_{CL} *standard-query1* o_{CL} *Snd compress1*›

The operation *query1'* is defined like *query1* (and the motivation and properties mentioned there also hold here), except that it is based on *standard-query1'* instead of *standard-query1*. See the comment at *standard-query1'* for a discussion of the difference.

definition *query1'* **where** ‹*query1'* = *Snd compress1* o_{CL} *standard-query1'* o_{CL} *Snd compress1*›

lemma *unitary-query1*[*simp*]: ‹*unitary query1*›

lemma *norm-query1'*[*simp*]: ‹*norm query1'* = *1*›

The following lemmas give explicit formulas for the result of applying *query1* and *query1'* to computational basis states (*ket trafo*). While the definitions of *query1* and *query1'* are useful for showing structural properties of these operations (e.g., the fact that they actually simulate a random oracle), for doing computations in concrete cases (e.g., the preservation of an invariant), the explicit formulas can be more useful.

lemma *query1-None*: ‹$query1 *_V\ ket\ (y{,}None) =$
$\alpha *_C (\sum d{\in}UNIV.\ ket\ (y + d,\ Some\ d))$
$-\ \alpha\hat{}3 *_C (\sum y'{\in}UNIV.\ \sum d{\in}UNIV.\ ket\ (y',\ Some\ d))$
$+\ \alpha^2 *_C (\sum d{\in}UNIV.\ ket\ (d,\ None))$› (**is** ‹- = ?rhs›)

lemma *query1-Some*: ‹$query1 *_V\ ket\ (y,\ Some\ d) =$
$ket\ (y + d,\ Some\ d)$
$+\ \alpha *_C ket\ (y + d,\ None)$
$-\ \alpha\hat{}3 *_C (\sum y'{\in}UNIV.\ ket\ (y',\ None))$
$-\ \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y + d',\ Some\ d'))$
$-\ \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y + d,\ Some\ d'))$
$+\ \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y,\ Some\ d'))$
$+\ \alpha\hat{}4 *_C (\sum y'{\in}UNIV.\ \sum d'{\in}UNIV.\ ket\ (y',\ Some\ d'))$›
(**is** ‹- = ?rhs›)

lemma *query1*:
shows ‹$query1 *_V (ket\ yd) = (case\ yd\ of$
$(y,\ None) \Rightarrow$
$\alpha *_C (\sum d{\in}UNIV.\ ket\ (y + d,\ Some\ d))$
$-\ \alpha\hat{}3 *_C (\sum y'{\in}UNIV.\ \sum d{\in}UNIV.\ ket\ (y',\ Some\ d))$
$+\ \alpha^2 *_C (\sum d{\in}UNIV.\ ket\ (d,\ None))$
$|\ (y,\ Some\ d) \Rightarrow$
$ket\ (y + d,\ Some\ d)$
$+\ \alpha *_C ket\ (y + d,\ None)$
$-\ \alpha\hat{}3 *_C (\sum y'{\in}UNIV.\ ket\ (y',\ None))$
$-\ \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y + d',\ Some\ d'))$
$-\ \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y + d,\ Some\ d'))$
$+\ \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y,\ Some\ d'))$
$+\ \alpha\hat{}4 *_C (\sum y'{\in}UNIV.\ \sum d'{\in}UNIV.\ ket\ (y',\ Some\ d')))$›

lemma *query1'-None*: ‹$query1' *_V\ ket\ (y{,}None) =$
$\alpha *_C (\sum d{\in}UNIV.\ ket\ (y + d,\ Some\ d))$
$-\ \alpha\hat{}3 *_C (\sum y'{\in}UNIV.\ \sum d{\in}UNIV.\ ket\ (y',\ Some\ d))$
$+\ \alpha^2 *_C (\sum d{\in}UNIV.\ ket\ (d,\ None))$› (**is** ‹- = ?rhs›)

lemma *query1'-Some*: ‹$query1' *_V\ ket\ (y,\ Some\ d) =$
$ket\ (y + d,\ Some\ d)$
$+\ \alpha *_C ket\ (y + d,\ None)$
$-\ \alpha\hat{}3 *_C (\sum y'{\in}UNIV.\ ket\ (y',\ None))$
$-\ \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y + d',\ Some\ d'))$
$-\ \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y + d,\ Some\ d'))$
$+\ \alpha\hat{}4 *_C (\sum y'{\in}UNIV.\ \sum d'{\in}UNIV.\ ket\ (y',\ Some\ d'))$›
(**is** ‹- = ?rhs›)

lemma *query1'*:
shows ‹$query1' *_V (ket\ yd) = (case\ yd\ of$
$(y,\ None) \Rightarrow$
$\alpha *_C (\sum d{\in}UNIV.\ ket\ (y + d,\ Some\ d))$
$-\ \alpha\hat{}3 *_C (\sum y'{\in}UNIV.\ \sum d{\in}UNIV.\ ket\ (y',\ Some\ d))$
$+\ \alpha^2 *_C (\sum d{\in}UNIV.\ ket\ (d,\ None))$
$|\ (y,\ Some\ d) \Rightarrow$
$ket\ (y + d,\ Some\ d)$

$+ \alpha *_C$ *ket* $(y + d,$ *None*$)$
$- \alpha\hat{}3 *_C (\sum y' \in UNIV.$ *ket* $(y',$ *None*$))$
$- \alpha^2 *_C (\sum d' \in UNIV.$ *ket* $(y + d',$ *Some* $d'))$
$- \alpha^2 *_C (\sum d' \in UNIV.$ *ket* $(y + d,$ *Some* $d'))$
$+ \alpha\hat{}4 *_C (\sum y' \in UNIV. \sum d' \in UNIV.$ *ket* $(y',$ *Some* $d')))$›

4.9 *query* - Query the compressed oracle

We define the compressed oracle itself.

Analogous to the definition of *query1* above (decompress, *standard-query1*, recompress), the compressed oracle is defined by decompressing the oracle register (now a superposition of functions), applying *standard-query*, and recompressing.

That is: If one starts with a four-partite state $\psi \otimes_s$ *ket 0* $\otimes_s$ *ket 0* $\otimes_s$ *ket None* and keeps performing operations U_i on the parts 1–3 of the state, interleaved with *query* invocations on parts 2–4, this is a simulation of starting with state $\psi \otimes_s 0$ and performing U_i interleaved with invocations of the unitary $|x, y\rangle \mapsto |x, y \oplus h(x)\rangle$ on parts 2, 3 where h is a function chosen uniformly at random in the beginning.

Note that there is an alternative way of defining the compressed oracle, namely by decompressing not the whole oracle register, but only the specific oracle output that we are querying. This is closer to an efficient implementation of the compressed oracle. We show that this definition is equivalent below (lemma *query-local*).

definition *query* **where** ‹*query = reg-3-3 compress* o_{CL} *standard-query* o_{CL} *reg-3-3 compress*›

query′ is defined like *query*, except that it's based on *standard-query1′* instead of *standard-query1*. See the discussion of *standard-query1′* for the difference.

definition *query′* **where** ‹*query′ = reg-3-3 compress* o_{CL} *standard-query′* o_{CL} *reg-3-3 compress*›

lemma *unitary-query*[*simp*]: ‹*unitary query*›

lemma *norm-query*[*simp*]: ‹*norm query = 1*›

lemma *norm-query′*[*simp*]: ‹*norm query′ = 1*›

lemma *query-local-generic*:
— A generalization of lemmas *query-local* and *query′-local* below. We prove this first because it avoids a duplication of the proof because *query-local* and *query′-local* have very similar proofs.
fixes *query* :: ‹$('x \times 'y \times ('x \rightharpoonup 'y))$ *update*› **and** *query1*
and *standard-query* **and** *standard-query1*
assumes *query-def*: ‹*query = reg-3-3 compress* o_{CL} *standard-query* o_{CL} *reg-3-3 compress*›
assumes *query1-def*: ‹*query1 = Snd compress1* o_{CL} *standard-query1* o_{CL} *Snd compress1*›
assumes *standard-query-ket*: ‹$\bigwedge x\ \psi.$ *standard-query* $*_V$ (*ket* $x \otimes_s \psi$) = *ket* $x \otimes_s$ ((*Fst*; *Snd o function-at x*) *standard-query1* $*_V \psi$)›
shows ‹*query = controlled-op* ($\lambda x.$ (*Fst*; *Snd o function-at x*) *query1*)›

We give an alternate (equivalent) definition of the compressed oracle *query*. Instead of decompressing the whole oracle, we decompress only the output we need. Specifically, this is implemented by – if the query register contains *ket x* – performing *query1* on the output register and on the register H_x which is the part of the oracle register which corresponds to the output for input x.

And analogously for *query1'*.

lemma *query-local*: ‹*query* = *controlled-op* ($\lambda x.$ (*Fst*; *Snd o function-at x*) *query1*)›

lemma *query'-local*: ‹*query'* = *controlled-op* ($\lambda x.$ (*Fst*; *Snd o function-at x*) *query1'*)›

lemma (**in** *compressed-oracle*) *standard-query-compress*: ‹*standard-query* o_{CL} *reg-3-3 compress* = *reg-3-3 compress* o_{CL} *query*›

lemma (**in** *compressed-oracle*) *standard-query'-compress*: ‹*standard-query'* o_{CL} *reg-3-3 compress* = *reg-3-3 compress* o_{CL} *query'*›

end

end

5 *CO-Invariants* Preservation of invariants under compressed oracle queries

theory *CO-Invariants* **imports**
  Invariant-Preservation
  CO-Operations
begin

lemma *function-oracle-ket-invariant*: ‹*function-oracle h* $*_S$ *ket-invariant I* = *ket-invariant* (($\lambda(x,y).$ $(x,y + h\ x)$) ‘ *I*)›

lemma *function-oracle-Snd-ket-invariant*: ‹*Snd* (*function-oracle h*) $*_S$ *ket-invariant I* = *ket-invariant* (($\lambda(w,x,y).$ $(w,x,y{+}h\ x)$) ‘ *I*)›

context *compressed-oracle* **begin**

This lemma allows to simplify the preservation of invariants under invocations of the compressed oracle.

Given an invariant *I*, it can be split into many invariants *I1 z* for which preservation is shown then with respect to a fixed oracle input *x z*, using the simpler oracle *query1* instead.

This allows to reduce complex cases to more elementary ones that talk about a single output of the oracle.

Lemmas *inv-split-reg-query* and *inv-split-reg-query'* are the specific instantiations of this for the two compressed oracle variants *query* and *query'*.

lemma *inv-split-reg-query-generic*:
  fixes *query query1*
  assumes *query-local*: ‹*query* = *controlled-op* ($\lambda x.$ (*Fst*; *Snd o function-at x*) *query1*)›
  fixes *X* :: ‹*'x update* ⇒ *'m update*›
    and *Y* :: ‹*'y update* ⇒ *'m update*›
    and *H* :: ‹(*'x*⇀*'y*) *update* ⇒ *'m update*›
    and *K* :: ‹*'z* ⇒ *'m ell2 ccsubspace*›
    and *x* :: ‹*'z* ⇒ *'x*›
    and *M* :: ‹*'z set*›
  assumes *XK*: ‹$\bigwedge z.\ z{\in}M \Longrightarrow K\ z \le$ *lift-invariant X* (*ket-invariant* $\{x\ z\}$)›
  assumes *pres-I1*: ‹$\bigwedge z.\ z{\in}M \Longrightarrow$ *preserves query1* (*I1 z*) (*J1 z*) ε›

assumes *I-leq*: ‹*I* ≤ (*SUP z*∈*M*. *K z* ⊓ *lift-invariant* (*Y*;*H o function-at* (*x z*)) (*I1 z*))›
assumes *J-geq*: ‹⋀*z*. *z*∈*M* ⟹ *J* ≥ *K z* ⊓ *lift-invariant* (*Y*;*H o function-at* (*x z*)) (*J1 z*)›
assumes *YK*: ‹⋀*z*. *z*∈*M* ⟹ *compatible-register-invariant Y* (*K z*)›
assumes *HK*: ‹⋀*z*. *z*∈*M* ⟹ *compatible-register-invariant* (*H o function-at* (*x z*)) (*K z*)›
assumes [*simp*]: ‹*compatible X Y*› ‹*compatible X H*› ‹*compatible Y H*›
assumes *U*: ‹*U* = ((*X*;(*Y*;*H*)) *query*)›
assumes *orthoK*: ‹⋀*z z*′. *z*∈*M* ⟹ *z*′∈*M* ⟹ *z* ≠ *z*′ ⟹ *orthogonal-spaces* (*K z*) (*K z*′)›
assumes ‹ε ≥ *0*›
assumes ‹*finite M*›
shows ‹*preserves U I J* ε›

lemmas *inv-split-reg-query* = *inv-split-reg-query-generic*[*OF query-local*]
lemmas *inv-split-reg-query*′ = *inv-split-reg-query-generic*[*OF query*′*-local*]

definition ‹*num-queries q* = {(*x*::′*x*, *y*::′*y*, *D*::′*x*⇀′*y*). *card* (*dom D*) ≤ *q*}›
definition ‹*num-queries*′ *q* = {*D*::′*x*⇀′*y*. *card* (*dom D*) ≤ *q*}›

lemma *num-queries-num-queries*′: ‹*num-queries q* = *UNIV* × *UNIV* × (*num-queries*′ *q*)›

lemma *ket-invariant-num-queries-num-queries*′: ‹*ket-invariant* (*num-queries q*) = ⊤ $\otimes_S$ ⊤ $\otimes_S$ *ket-invariant* (*num-queries*′ *q*)›

This lemma shows that the number of recorded queries (defined outputs in the oracle register) increases at most by 1 upon each query of the compressed oracle.

The two instantiations for the two compressed oracle variants are given afterwards.

lemma *preserves-num-generic*:
fixes *query query1*
assumes *query-local*: ‹*query* = *controlled-op* (λ*x*. (*Fst*; *Snd o function-at x*) *query1*)›
shows ‹*preserves-ket query* (*num-queries q*) (*num-queries* (*q*+*1*)) *0*›

lemmas *preserves-num* = *preserves-num-generic*[*OF query-local*]
lemmas *preserves-num*′ = *preserves-num-generic*[*OF query*′*-local*]

We now present various lemmas that give concrete bounds for the preservation of invariants under various conditions, for *query1* (and *query1*′).

The invariants are formulated specifically for an application of *query1* to a two-partite system with query output register and oracle register only.

These can be applied to derive invariant preservation for full compressed oracle queries on arbitrary systems by first splitting the invariant using *inv-split-reg-query*.

The first bound is applicable for ket-invariants that do not put any conditions on the output register and that not not require that the output register is defined (not *None*) after the query.

Lemmas *preserve-query1-bound* and *preserve-query1*′*-bound*; with slightly simplified bounds in *preserve-query1-simplified*, *preserve-query1*′*-simplified*.

definition ‹*preserve-query1-bound NoneI* b_i b_{j0} = *4* ∗ *sqrt* b_{j0} ∗ *sqrt* b_i / *N* + *2* ∗ *of-bool NoneI* ∗ *sqrt* b_{j0} / *sqrt N*›
lemma *preserve-query1*:
assumes *IJ*: ‹*I* ⊆ *J*›
assumes [*simp*]: ‹*None* ∈ *J*›
assumes b_i: ‹*card* (*Some* −‘ *I*) ≤ b_i›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) ≤ b_{j0}›
assumes ε: ‹ε ≥ *preserve-query1-bound* (*None*∈*I*) b_i b_{j0}›

shows ‹*preserves-ket query1* (*UNIV* × *I*) (*UNIV* × *J*) ε›

definition ‹*preserve-query1′-bound NoneI* b_i b_{j0} = *3* ∗ *sqrt* b_{j0} ∗ *sqrt* b_i / *N* + *2* ∗ *of-bool NoneI* ∗ *sqrt* b_{j0} / *sqrt N*›
lemma *preserve-query1′*:
assumes *IJ*: ‹*I* ⊆ *J*›
assumes [*simp*]: ‹*None* ∈ *J*›
assumes b_i: ‹*card* (*Some* −‘ *I*) ≤ b_i›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) ≤ b_{j0}›
assumes ε: ‹ε ≥ *preserve-query1′-bound* (*None*∈*I*) b_i b_{j0}›
shows ‹*preserves-ket query1′* (*UNIV* × *I*) (*UNIV* × *J*) ε›

lemma *preserve-query1-simplified*:
assumes ‹*I* ⊆ *J*›
assumes ‹*None* ∈ *J*›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) ≤ b_{j0}›
shows ‹*preserves-ket query1* (*UNIV* × *I*) (*UNIV* × *J*) (*6* ∗ *sqrt* b_{j0} / *sqrt N*)›

lemma *preserve-query1′-simplified*:
assumes ‹*I* ⊆ *J*›
assumes ‹*None* ∈ *J*›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) ≤ b_{j0}›
shows ‹*preserves-ket query1′* (*UNIV* × *I*) (*UNIV* × *J*) (*5* ∗ *sqrt* b_{j0} / *sqrt N*)›

The next bound is applicable for ket-invariants assume the output register to have a specific value *ket* y_0 (typically *ket 0*) before the query and do not put any conditions on the output register after the query.

Lemmas *preserve-query1-fixY* and *preserve-query1′-fixY*.

definition ‹*preserve-query1-fixY-bound NoneI NoneJ* b_i b_{j0} = *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*)
+ *3* ∗ *sqrt* b_{j0} ∗ *sqrt* b_i / *N* + *of-bool NoneI* ∗ *sqrt* b_{j0} / *sqrt N* + *of-bool NoneI* ∗ *sqrt* b_{j0} / *N*
+ *of-bool NoneJ* / *sqrt N* + *of-bool NoneJ* ∗ *sqrt* b_i / *N* + *of-bool* (*NoneI* ∧ *NoneJ*) / *sqrt N*›
lemma *preserve-query1-fixY*:
assumes *IJ*: ‹*I* ⊆ *J*›
assumes b_i: ‹*card* (*Some* −‘ *I*) ≤ b_i›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) ≤ b_{j0}›
assumes ε: ‹ε ≥ *preserve-query1-fixY-bound* (*None*∈*I*) (*None*∉*J*) b_i b_{j0}›
shows ‹*preserves-ket query1* ({y_0} × *I*) (*UNIV* × *J*) ε›

definition ‹*preserve-query1′-fixY-bound NoneI NoneJ* b_i b_{j0} = *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*)
+ *2* ∗ *sqrt* b_{j0} ∗ *sqrt* b_i / *N* + *of-bool NoneI* ∗ *sqrt* b_{j0} / *sqrt N* + *of-bool NoneI* ∗ *sqrt* b_{j0} / *N*
+ *of-bool NoneJ* / *sqrt N* + *of-bool NoneJ* ∗ *sqrt* b_i / *N* + *of-bool* (*NoneI* ∧ *NoneJ*) / *sqrt N*›
lemma *preserve-query1′-fixY*:
assumes *IJ*: ‹*I* ⊆ *J*›
assumes b_i: ‹*card* (*Some* −‘ *I*) ≤ b_i›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) ≤ b_{j0}›
assumes ε: ‹ε ≥ *preserve-query1′-fixY-bound* (*None*∈*I*) (*None*∉*J*) b_i b_{j0}›

shows ‹*preserves-ket query1′* ({y_0} × *I*) (*UNIV* × *J*) ε›

The next bound is applicable for ket-invariants assume the output register to have a specific value *ket* y_0 (typically *ket 0*) before the query and require that after the query, the oracle register is not *None* and the output register has the correct value given that oracle register content.

Notice that this invariant is only available for *query1'*, not for *query1*!

definition ‹*preserve-query1'-fixY-bound-output* b_i = *4 / sqrt N + 2 ∗ sqrt* b_i */ N*›
lemma *preserve-query1'-fixY-output*:
  assumes b_i: ‹*card (Some −' I)* $\leq b_i$›
  assumes ε: ‹$\varepsilon \geq$ *preserve-query1'-fixY-bound-output* b_i›
  shows ‹*preserves-ket query1'* $(\{y_0\} \times I)$ $\{(y_0{+}d,\ Some\ d) |\ d.\ True\}$ ε›

A simpler to understand (and sometimes simpler to use) special case of *preserve-query1'-fixY-output* in terms of *query'* and ket-invariants.

lemma (**in** *compressed-oracle*) *preserves-ket-query'-output-simple*:
  ‹*preserves-ket query'* $\{(x,\ y,\ D).\ y = 0\}$ $\{(x,\ y,\ D).\ D\ x = Some\ y\}$ $(6\ /\ sqrt\ N)$›

A strengthened form of *preserves-ket-query'-output-simple* that additionally maintains a property P on the already existing oracle register (that can depend also on some auxiliary register and on the query input register).

This comes with the condition on P that when P accepts some oracle function that is undefined at the query input x, then it needs to accept the updated oracle function with any output at x. And if P doesn't accept the oracle function to be undefined at x, then it must accept either only a small amount of outputs or all but a small amount of outputs for x.

lemma (**in** *compressed-oracle*) *preserves-ket-query'-output*:
  fixes F :: ‹$('x \times 'y \times ('x \rightharpoonup 'y))\ update \Rightarrow\ 'mem\ update$›
    and P :: ‹$'w{::}finite \Rightarrow\ 'x \Rightarrow ('x \rightharpoonup 'y) \Rightarrow bool$›
    and b :: *nat*
  assumes [*register*]: ‹*register G*›
  assumes ‹$F = G\ o\ Snd$›
  assumes *PNone*: ‹$\bigwedge w\ x\ D.\ P\ w\ x\ (D(x{:=}None)) \Longrightarrow P\ w\ x\ D$›
  assumes *PSome*: ‹$\bigwedge w\ x\ D.\ D\ x = None \Longrightarrow \neg\ P\ w\ x\ D \Longrightarrow let\ c = card\ \{y.\ P\ w\ x\ (D(x{:=}Some\ y))\}$ $in\ c{*}(N{-}c) \leq b$›
  shows ‹*preserves* $(F\ query')$ $(lift\text{-}invariant\ G\ (ket\text{-}invariant\ \{(w,\ x,\ y,\ D).\ y = 0 \wedge P\ w\ x\ D\}))$
    $(lift\text{-}invariant\ G\ (ket\text{-}invariant\ \{(w,\ x,\ y,\ D).\ D\ x = Some\ y \wedge P\ w\ x\ D\}))$
    $(9\ /\ sqrt\ N + 2 * sqrt\ b\ /\ N)$›

This is an example of how *preserves-ket-query'-output* is used to deal with more complex query sequences. It is also useful in its own right (we use it in *Collision.thy*).

It shows that if we make two queries, then the oracle function contains the outputs of both queries. (In contrast, *preserves-ket-query'-output-simple* shows this only for a single query.)

lemma *dist-inv-double-query'*:
  fixes *X1 X2 Y1 Y2 H* **and** *state1* :: ‹*'mem ell2*›
  defines ‹$state2 \equiv (X1;(Y1;H))\ query' *_V\ state1$›
  defines ‹$state3 \equiv (X2;(Y2;H))\ query' *_V\ state2$›
  assumes [*register*]: ‹*mutually compatible (X1,X2,Y1,Y2,H)*›
  assumes [*iff*]: ‹$norm\ state1 \leq 1$›
  assumes *dist1*: ‹$dist\text{-}inv\ ((X1;X2);((Y1;Y2);H))\ (ket\text{-}invariant\ \{((x1,x2),(y1,y2),D).\ y1 = 0 \wedge y2 = 0\})\ state1 \leq \varepsilon$›
  shows ‹$dist\text{-}inv\ ((X1;X2);((Y1;Y2);H))\ (ket\text{-}invariant\ \{((x1,x2),(y1,y2),D).\ D\ x1 = Some\ y1 \wedge D\ x2 = Some\ y2\})\ state3 \leq \varepsilon + 20\ /\ sqrt\ N$›

The next bound is applicable for ket-invariants assume the output register to have a value *ket d* that matches what is in the output register before the query and require that after the query, the oracle register is not *None* and the output register has the correct value given that oracle register content. (I.e., before an uncomputation step.)

Notice that this invariant is only available for *query1'*, not for *query1*!

definition ‹*preserve-query1'-uncompute-bound NoneJ* b_i b_{j0} =
of-bool NoneJ ∗ *sqrt* b_i / *sqrt N* + *of-bool NoneJ* ∗ *sqrt* b_i / *N*
+ *sqrt* b_{j0} / *N* + *sqrt* b_i ∗ *sqrt* b_{j0} / *N* + *sqrt* b_i ∗ *sqrt* b_{j0} / (*N* ∗ *sqrt N*)›
lemma *preserve-query1'-uncompute*:
assumes *IJ*: ‹$I \subseteq J$›
assumes b_i: ‹*card* (*Some* −‘ *I*) ≤ b_i›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) ≤ b_{j0}›
assumes ε: ‹ε ≥ *preserve-query1'-uncompute-bound* (*None*∉*J*) b_i b_{j0}›
shows ‹*preserves-ket query1'* ((*UNIV* × *I*) ∩ {(*d*, *Some d*)| *d*. *True*}) (*UNIV* × *J*) ε›

end

end

6 *Compressed-Oracle-Is-RO* – Equivalence of compressed oracle and regular random oracle

theory *Compressed-Oracle-Is-RO* **imports**
Registers.Pure-States
CO-Operations
begin

lemma *swap-function-oracle-measure-generic*:
fixes *standard-query*
fixes *X* :: ‹*'x update* ⇒ *'mem update*› **and** *Y* :: ‹*'y*::*ab-group-add update* ⇒ *'mem update*›
assumes *std-query-Some*: ‹⋀*H x y z*. *H x* = *Some z* ⟹ *standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket* (*x*, *y* + *z*, *H*)›
assumes [*register*]: ‹*compatible X Y*›
shows ‹(*Fst o X*; (*Fst o Y*; *Snd*)) *standard-query* o_{CL} *Snd* (*proj* (*ket* (*Some o h*)))
= *Fst* ((*X*;*Y*) (*function-oracle h*)) o_{CL} *Snd* (*proj* (*ket* (*Some o h*)))›

lemma *standard-query-for-fixed-func-generic*:
fixes *standard-query*
fixes *X* :: ‹*'x update* ⇒ *'mem update*› **and** *Y* :: ‹*'y*::*ab-group-add update* ⇒ *'mem update*›
assumes ‹⋀*H x y z*. *H x* = *Some z* ⟹ *standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket* (*x*, *y* + *z*, *H*)›
assumes ‹*compatible X Y*›
shows ‹(*Fst o X*; (*Fst o Y*; *Snd*)) *standard-query* $*_V$ ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))
= *Fst* ((*X*;*Y*) (*function-oracle h*)) $*_V$ ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))›

end

7 *Oracle-Programs* – Oracle programs and their execution

theory *Oracle-Programs* **imports**
CO-Operations
Invariant-Preservation
Compressed-Oracle-Is-RO
begin

7.1 Oracle programs

datatype (*'mem*, *'x*, *'y*) *program-step* = *ProgramStep* ‹*'mem update*› | *QueryStep* ‹*'x update* ⇒ *'mem update*› ‹*'y update* ⇒ *'mem update*›
type-synonym (*'mem*, *'x*, *'y*) *program* = ‹(*'mem*, *'x*, *'y*) *program-step list*›

inductive *is-QueryStep* :: ‹(*'mem*,*'x*,*'y*::*ab-group-add*) *program-step* ⇒ *bool*› **where** *is-QueryStep-QueryStep*[*iff*]: ‹*is-QueryStep* (*QueryStep X Y*)›
inductive *is-ProgramStep* :: ‹(*'mem*,*'x*,*'y*::*ab-group-add*) *program-step* ⇒ *bool*› **where** *is-ProgramStep-ProgramStep*[*iff*]: ‹*is-ProgramStep* (*ProgramStep U*)›

lemma *is-QueryStep-ProgramStep*[*iff*]: ‹¬ *is-QueryStep* (*ProgramStep U*)›

lemma *is-ProgramStep-QueryStep*[*iff*]: ‹¬ *is-ProgramStep* (*QueryStep X Y*)›

fun *valid-program-step* **where** ‹*valid-program-step* (*QueryStep X Y*) = *compatible X Y*› | ‹*valid-program-step* (*ProgramStep U*) = *isometry U*›
definition *valid-program* **where** ‹*valid-program prog* = *list-all valid-program-step prog*›

lemma *valid-program-cons*[*simp*]: ‹*valid-program* (*p* # *ps*) ⟷ *valid-program-step p* ∧ *valid-program ps*›

lemma *valid-program-append*: ‹*valid-program* (*p* @ *q*) ⟷ *valid-program p* ∧ *valid-program q*›

lemma *valid-program-empty*[*iff*]: ‹*valid-program* []›

fun *exec-program-step* :: ‹(*'x* ⇒ *'y*) ⇒ (*'mem*,*'x*,*'y*::*ab-group-add*) *program-step* ⇒ *'mem ell2* ⇒ *'mem ell2*› **where**
‹*exec-program-step h* (*ProgramStep U*) ψ = $U *_V \psi$›
| ‹*exec-program-step h* (*QueryStep X Y*) ψ = $(X;Y)$ (*function-oracle h*) $*_V \psi$›

fun *exec-program-step-with* :: ‹(*'x* × *'y* × *'o*) *update* ⇒ (*'mem*,*'x*,*'y*) *program-step* ⇒ (*'mem* × *'o*) *ell2* ⇒ (*'mem* × *'o*) *ell2*› **where**
‹*exec-program-step-with Q* (*ProgramStep U*) ψ = *Fst U* $*_V \psi$›
| ‹*exec-program-step-with Q* (*QueryStep X Y*) ψ = (*Fst o X*; (*Fst o Y*; *Snd*)) *Q* ψ›

definition *exec-program* :: ‹(*'x* ⇒ *'y*::*ab-group-add*) ⇒ (*'mem*,*'x*,*'y*) *program* ⇒ *'mem ell2* ⇒ *'mem ell2*› **where**
‹*exec-program h program* ψ = *fold* (*exec-program-step h*) *program* ψ›
definition *exec-program-with* :: ‹(*'x* × *'y* × *'o*) *update* ⇒ (*'mem*,*'x*,*'y*) *program* ⇒ (*'mem*×*'o*) *ell2* ⇒ (*'mem*×*'o*) *ell2*› **where**
‹*exec-program-with Q program* ψ = *fold* (*exec-program-step-with Q*) *program* ψ›

lemma *bounded-clinear-exec-program-step-with*[*bounded-clinear*]: ‹*bounded-clinear* (*exec-program-step-with Q step*)›

lemma *exec-program-empty*[*simp*]: ‹*exec-program h* [] ψ = ψ›

lemma *exec-program-with-empty*[*simp*]: ‹*exec-program-with* Q [] $\psi = \psi$›
lemma *exec-program-append*: ‹*exec-program* h (p @ q) ψ = *exec-program* h q (*exec-program* h p ψ)›
lemma *exec-program-with-append*: ‹*exec-program-with* Q (p @ q) ψ = *exec-program-with* Q q (*exec-program-with* Q p ψ)›
lemma *exec-program-cons*[*simp*]: ‹*exec-program* h (*step*#*prog*) ψ = *exec-program* h *prog* (*exec-program-step* h *step* ψ)›
lemma *exec-program-with-cons*[*simp*]: ‹*exec-program-with* Q (*step*#*prog*) ψ = *exec-program-with* Q *prog* (*exec-program-step-with* Q *step* ψ)›

lemma *norm-exec-program-step-with*: ‹*norm* (*exec-program-step-with oracle program-step* ψ) $\leq$ *norm* ψ›
if ‹*valid-program-step program-step*› **and** ‹*norm oracle* $\leq$ *1*›

lemma *norm-exec-program-with*:
‹*norm* (*exec-program-with oracle program* ψ) $\leq$ *norm* ψ› **if** ‹*norm oracle* $\leq$ *1*› **and** ‹*valid-program program*› **for** *program*

lemma *norm-exec-program-step-with-isometry*:
assumes ‹*valid-program-step program-step*›
assumes ‹*isometry query*›
shows ‹*norm* (*exec-program-step-with query program-step* ψ) = *norm* ψ›

7.2 Lifting

fun *lift-program-step* :: ‹('*a update* ⇒ '*mem update*) ⇒ ('*a*,'*x*,'*y*::*ab-group-add*) *program-step* ⇒ ('*mem*,'*x*,'*y*) *program-step*› **where**
‹*lift-program-step* Q (*ProgramStep* U) = *ProgramStep* (Q U)›
| ‹*lift-program-step* Q (*QueryStep* X Y) = *QueryStep* (Q *o* X) (Q *o* Y)›

definition *lift-program* :: ‹('*a update* ⇒ '*mem update*) ⇒ ('*a*,'*x*,'*y*::*ab-group-add*) *program-step list* ⇒ ('*mem*,'*x*,'*y*) *program*› **where**
‹*lift-program* Q p = *map* (*lift-program-step* Q) p›

lemma *valid-program-step-lift*:
assumes ‹*register* Q› **and** ‹*valid-program-step* p›
shows ‹*valid-program-step* (*lift-program-step* Q p)›

lemma *valid-program-lift*:
assumes ‹*register* Q› **and** ‹*valid-program* p›
shows ‹*valid-program* (*lift-program* Q p)›

lemma *lift-program-empty*[*simp*]: ‹*lift-program* Q [] = []›

lemma *lift-program-cons*: ‹*lift-program* Q (*program-step* # *program*) = *lift-program-step* Q *program-step* # *lift-program* Q *program*›

lemma *lift-program-append*: ‹*lift-program* Q (*program1* @ *program2*) = *lift-program* Q *program1* @ *lift-program* Q *program2*›

lemma *is-QueryStep-lift-program-step*[*simp*]: ‹*is-QueryStep* (*lift-program-step* Q *program-step*) ⟷ *is-QueryStep program-step*›

lemma *filter-is-QueryStep-lift-program*: ‹*filter is-QueryStep* (*lift-program* Q *program*) = *lift-program* Q (*filter is-QueryStep program*)›

lemma *length-lift-program*[*simp*]: ‹*length* (*lift-program Q program*) = *length program*›

definition ‹*query-count program* = *length* (*filter is-QueryStep program*)›

lemma *query-count-append*[*simp*]: ‹*query-count* (*p* @ *q*) = *query-count p* + *query-count q*›
lemma *query-count-nil*[*simp*]: ‹*query-count* [] = *0*›
lemma *query-count-cons-QueryStep*[*simp*]: ‹*query-count* (*QueryStep X Y* # *p*) = *query-count p* + *1*›
lemma *query-count-cons-ProgramStep*[*simp*]: ‹*query-count* (*ProgramStep U* # *p*) = *query-count p*›
lemma *query-count-lift-program*[*simp*]: ‹*query-count* (*lift-program Q p*) = *query-count p*›

lemma *exec-lift-program-step-Fst*:
 assumes ‹*valid-program-step program-step*›
 shows ‹*exec-program-step h* (*lift-program-step Fst program-step*) ($\psi \otimes_s \varphi$) = *exec-program-step h program-step* $\psi \otimes_s \varphi$›

lemma *exec-lift-program-Fst*:
 assumes ‹*valid-program program*›
 shows ‹*exec-program h* (*lift-program Fst program*) ($\psi \otimes_s \varphi$) = *exec-program h program* $\psi \otimes_s \varphi$›

7.3 Final measurement

definition *measurement-probability* :: ‹(*'a update* ⇒ *'mem update*) ⇒ *'mem ell2* ⇒ *'a* ⇒ *real*› **where**
 ‹*measurement-probability Q* ψ *x* = (*norm* (*Q* (*proj* (*ket x*)) ψ))^2›

lemma *measurement-probability-nonneg*: ‹*measurement-probability Q* ψ *x* ≥ *0*›

lemma *norm-register-Proj-ket-invariant-union*:
 — Helper lemma
 assumes ‹*register Q*› **and** ‹$A \cap B = \{\}$›
 shows ‹(*norm* (*Q* (*Proj* (*ket-invariant* ($A \cup B$))) ψ))^2 = (*norm* (*Q* (*Proj* (*ket-invariant A*)) ψ))^2 + (*norm* (*Q* (*Proj* (*ket-invariant B*)) ψ))^2›

lemma *measurement-probability-sum*:
 assumes ‹*register Q*› **and** ‹*finite F*›
 shows ‹($\sum x \in F$. *measurement-probability Q* ψ *x*) = (*norm* (*Q* (*Proj* (*ket-invariant F*)) ψ))^2›

lemma
 assumes ‹*register Q*›
 shows *measurement-probability-summable*: ‹*measurement-probability Q* ψ *summable-on A*›
 and *measurement-probability-infsum-leq*: ‹($\sum_\infty x \in A$. *measurement-probability Q* ψ *x*) ≤ (*norm* (*Q* (*Proj* (*ket-invariant A*)) ψ))^2›

lemma *dist-inv-measurement-probability*:
 fixes *I* :: ‹*'i*::*finite set*›
 assumes [*register*]: ‹*register Q*›
 shows ‹($\sum x \in I$. *measurement-probability Q* ψ *x*) = (*dist-inv Q* (*ket-invariant* ($-I$)) ψ)^2›

lemma *dist-inv-avg-measurement-probability*:
 fixes *I* :: ‹*'h*::*finite* ⇒ *'i*::*finite set*›
 assumes [*register*]: ‹*register Q*›
 shows ‹($\sum h \in UNIV$. $\sum x \in I\ h$. *measurement-probability Q* (ψ *h*) *x*) / *CARD*(*'h*)
  = (*dist-inv-avg Q* (λh. *ket-invariant* ($-$ *I h*)) ψ)^2›

7.4 Preservation

lemma *dist-inv-avg-exec-compatible*:
fixes *prog*
assumes ‹*valid-program prog*›
assumes [*register*]: ‹*compatible Q R*›
shows ‹*dist-inv-avg Q I* (λh::′*x*::*finite*⇒′*y*::{*finite*,*ab-group-add*}. *exec-program h* (*lift-program R prog*) (ψ *h*))
$\leq$ *dist-inv-avg Q I* ψ›

lemma *dist-inv-exec′-compatible*:
fixes *prog*
assumes ‹*valid-program prog*›
assumes *normU*: ‹*norm U* $\leq$ *1*›
assumes [*register*]: ‹*register R*›
assumes *compatQ1*[*register*]: ‹*compatible Q* (*Fst o R*)›
assumes *compatQ2*[*register*]: ‹*compatible Q Snd*›
shows ‹*dist-inv Q I* (*exec-program-with U* (*lift-program R prog*) ψ) $\leq$ *dist-inv Q I* ψ›

7.5 Misc

lemma *dist-inv-induct*:
fixes *oracle* :: ‹(′*x* × ′*y*::*ab-group-add* × (′*x* ⇒ ′*y option*)) *update*›
assumes ‹*compatible R Fst*›
assumes ‹($\sum$ *i*<*query-count program. g i*) $\leq \varepsilon$›
assumes *init*: ‹ψ*0* ∈ *space-as-set* (*lift-invariant R* (*J 0*))›
assumes ‹*J* (*query-count program*) $\leq$ *I*›
assumes ‹*valid-program program*›
assumes ‹$\bigwedge$*X Y i. compatible X Y* $\Longrightarrow$ *preserves* ((*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *oracle* :: (′*m* × -) *update*) (*lift-invariant R* (*J i*))
(*lift-invariant R* (*J* (*Suc i*))) (*g i*)›
assumes ‹*norm oracle* $\leq$ *1*›
assumes ‹*norm* ψ*0* $\leq$ *1*›
shows ‹*dist-inv R I* (*exec-program-with oracle program* ψ*0*) $\leq \varepsilon$›

7.6 Random Oracles

lemma *standard-query-for-fixed-function-generic*:
fixes *standard-query*
assumes ‹$\bigwedge$*H x y z. H x* = *Some z* $\Longrightarrow$ *standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket* (*x*, *y* + *z*, *H*)›
assumes ‹*valid-program program*›
shows ‹*exec-program h program initial-state* $\otimes_s$ *ket* (*Some o h*)
= *exec-program-with standard-query program* (*initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*))›

lemma *standard-query-for-fixed-function-dist-inv-generic*:
assumes ‹$\bigwedge$*H x y z. H x* = *Some z* $\Longrightarrow$ *standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket* (*x*, *y* + *z*, *H*)›
assumes ‹*valid-program program*›
assumes *compat*: ‹*compatible-invariants* ($\top \otimes_S$ *ccspan* {*ket* (*Some* ∘ *h*)}) *J*›
assumes *IJ*: ‹*J* ⊓ ($\top \otimes_S$ *ccspan*{*ket* (*Some o h*)}) = *I* $\otimes_S$ *ccspan*{*ket* (*Some o h*)}›
assumes [*register*]: ‹*register Q*›
shows ‹*dist-inv Q I* (*exec-program h program initial-state*) =
dist-inv (*Fst o Q*; *Snd*) *J* (*exec-program-with standard-query program* (*initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*)))›

lemma *standard-query-is-ro-generic*:
fixes *standard-query*
assumes ‹$\bigwedge$*H x y z. H x = Some z* $\Longrightarrow$ *standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket* (*x*, *y* + *z*, *H*)›
assumes ‹*valid-program program*›
shows ‹*exec-program-with standard-query program* (*initial-state* $\otimes_s$ (*superpos-total* :: ('*x*::*finite* ⇒ '*y*::{*finite*,*ab-group-add*} *option*) *ell2*))
= ($\sum$ *h*∈*UNIV*. (*exec-program h program initial-state* $\otimes_s$ *ket* (*Some o h*)) $/_R$ *sqrt CARD*('*x* ⇒ '*y*))›

lemma *standard-query-is-ro-dist-inv-generic*:
fixes *standard-query* :: ‹('*x*::*finite* × '*y*::{*finite*,*ab-group-add*} × ('*x* ⇀ '*y*)) *ell2* $\Rightarrow_{CL}$ -›
assumes ‹$\bigwedge$*H x y z. H x = Some z* $\Longrightarrow$ *standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket* (*x*, *y* + *z*, *H*)›
assumes ‹*valid-program program*›
assumes [*register*]: ‹*register Q*›
shows ‹*dist-inv-avg Q* (λ-. *I*) (λ*h*. *exec-program h program initial-state*) =
dist-inv (*Fst o Q*) *I* (*exec-program-with standard-query program* (*initial-state* $\otimes_s$ *superpos-total*))› (**is** ‹*?lhs* = *?rhs*›)

lemma (**in** *compressed-oracle*) *standard-query-is-ro-dist-inv*:
assumes ‹*valid-program program*›
assumes [*register*]: ‹*register Q*›
shows ‹*dist-inv-avg Q* (λ-. *I*) (λ*h*. *exec-program h program initial-state*) =
dist-inv (*Fst o Q*) *I* (*exec-program-with standard-query program* (*initial-state* $\otimes_s$ *superpos-total*))› (**is** ‹*?lhs* = *?rhs*›)

lemma (**in** *compressed-oracle*) *standard-query'-is-ro-dist-inv*:
assumes ‹*valid-program program*›
assumes [*register*]: ‹*register Q*›
shows ‹*dist-inv-avg Q* (λ-. *I*) (λ*h*. *exec-program h program initial-state*) =
dist-inv (*Fst o Q*) *I* (*exec-program-with standard-query' program* (*initial-state* $\otimes_s$ *superpos-total*))› (**is** ‹*?lhs* = *?rhs*›)

lemma (**in** *compressed-oracle*) *compress-query-is-standard-query-generic*:
fixes *query standard-query*
assumes ‹*valid-program program*›
assumes ‹*standard-query* o_{CL} *reg-3-3 compress* = *reg-3-3 compress* o_{CL} *query*›
shows ‹*exec-program-with standard-query program* (*initial-state* $\otimes_s$ *superpos-total*)
= *Snd compress* $*_V$ *exec-program-with query program* (*initial-state* $\otimes_s$ *ket* (λ*x*. *None*))›

lemma (**in** *compressed-oracle*) *query-is-standard-query-generic*:
fixes *query standard-query*
assumes ‹*valid-program program*›
assumes ‹*standard-query* o_{CL} *reg-3-3 compress* = *reg-3-3 compress* o_{CL} *query*›
shows ‹*dist-inv Fst I* (*exec-program-with standard-query program* (*initial-state* $\otimes_s$ *superpos-total*))
= *dist-inv Fst I* (*exec-program-with query program* (*initial-state* $\otimes_s$ *ket* (λ*x*. *None*)))›

lemma (**in** *compressed-oracle*) *query-is-standard-query*:
  assumes ‹*valid-program program*›
  shows
  ‹*dist-inv Fst I* (*exec-program-with standard-query program* (*initial-state* $\otimes_s$ *superpos-total*)) =
   dist-inv Fst I (*exec-program-with query program* (*initial-state* $\otimes_s$ *ket* (λx. *None*)))›

lemma (**in** *compressed-oracle*) *query′-is-standard-query*:
  assumes ‹*valid-program program*›
  shows
  ‹*dist-inv Fst I* (*exec-program-with standard-query′ program* (*initial-state* $\otimes_s$ *superpos-total*)) =
   dist-inv Fst I (*exec-program-with query′ program* (*initial-state* $\otimes_s$ *ket* (λx. *None*)))›

lemma (**in** *compressed-oracle*) *query-is-random-oracle*:
  assumes ‹*valid-program program*›
  shows ‹*dist-inv-avg id* (λ-. *I*) (λh. *exec-program h program initial-state*) =
        dist-inv Fst I (*exec-program-with query program* (*initial-state* $\otimes_s$ *ket* (λ-. *None*)))›

lemma (**in** *compressed-oracle*) *query′-is-random-oracle*:
  assumes ‹*valid-program program*›
  shows ‹*dist-inv-avg id* (λ-. *I*) (λh. *exec-program h program initial-state*) =
        dist-inv Fst I (*exec-program-with query′ program* (*initial-state* $\otimes_s$ *ket* (λ-. *None*)))›

lemma (**in** *compressed-oracle*) *dist-inv-exec-query-exec-fixed*:
  fixes *program* :: ‹(*′mem*, *′x*::*finite*, *′y*::{*finite*,*ab-group-add*}) *program-step list*›
  fixes *Q* :: ‹*′a ell2* $\Rightarrow_{CL}$ *′a ell2* $\Rightarrow$ *′mem ell2* $\Rightarrow_{CL}$ *′mem ell2*›
  assumes ‹*valid-program program*›
  assumes [*register*]: ‹*register Q*›
  shows ‹*dist-inv* (*Fst* $\circ$ *Q*) *I* (*exec-program-with query program* (ψ $\otimes_s$ *ket* (λ-. *None*)))
      = *dist-inv-avg Q* (λ-. *I*) (λh. *exec-program h program* ψ)›

lemma (**in** *compressed-oracle*) *dist-inv-exec-query′-exec-fixed*:
  fixes *program* :: ‹(*′mem*, *′x*::*finite*, *′y*::{*finite*,*ab-group-add*}) *program-step list*›
  fixes *Q* :: ‹*′a ell2* $\Rightarrow_{CL}$ *′a ell2* $\Rightarrow$ *′mem ell2* $\Rightarrow_{CL}$ *′mem ell2*›
  assumes ‹*valid-program program*›
  assumes [*register*]: ‹*register Q*›
  shows ‹*dist-inv* (*Fst* $\circ$ *Q*) *I* (*exec-program-with query′ program* (ψ $\otimes_s$ *ket* (λ-. *None*)))
      = *dist-inv-avg Q* (λ-. *I*) (λh. *exec-program h program* ψ)›

end

8 *Find-Zero* Invariant preservation for zero-finding

theory *Find-Zero*
  imports *CO-Invariants Oracle-Programs*
begin

context *compressed-oracle* **begin**

definition ‹*no-zero* = {(*x*::*′x*,*y*::*′y*,*D*::*′x*⇀*′y*). *0* ∉ *ran D*}›
definition ‹*no-zero′* = {*D*::*′x*⇀*′y*. *0* ∉ *ran D*}›

lemma *no-zero-no-zero'*: ‹*no-zero* = *UNIV* × *UNIV* × *no-zero'*›

lemma *ket-invariant-no-zero-no-zero'*: ‹*ket-invariant no-zero* = ⊤ $\otimes_S$ ⊤ $\otimes_S$ *ket-invariant no-zero'*›

We show the preservation of the *no-zero* invariant. We show it first with respect to the oracle *query*.

lemma *preserves-no-zero*: ‹*preserves-ket query no-zero no-zero* (*6* / *sqrt N*)›

Like *preserves-no-zero* but with respect to the oracle *query*.

lemma *preserves-no-zero'*: ‹*preserves-ket query' no-zero no-zero* (*5* / *sqrt N*)›

lemma *preserves-no-zero-num*: ‹*preserves-ket query* (*no-zero* ∩ *num-queries q*) (*no-zero* ∩ *num-queries* (*q+1*)) (*6* / *sqrt N*)›

lemma *preserves-no-zero-num'*: ‹*preserves-ket query'* (*no-zero* ∩ *num-queries q*) (*no-zero* ∩ *num-queries* (*q+1*)) (*5* / *sqrt N*)›

8.1 Zero-finding is hard for q-query adversaries

lemma *zero-finding-is-hard*:
  fixes *program* :: ‹(*'mem*, *'x*, *'y*) *program*›
    and *adv-output* :: ‹*'x update* ⇒ *'mem update*›
    and *initial-state*
  assumes [*iff*]: ‹*valid-program program*›
  assumes ‹*norm initial-state* = *1*›
  assumes [*register*]: ‹*register adv-output*›
  shows ‹($\sum h{\in}UNIV.\ \sum x|h\ x = 0.$ *measurement-probability adv-output* (*exec-program h program initial-state*) *x*) / *CARD*(*'x* ⇒ *'y*)
        ≤ (*5* ∗ *real* (*query-count program*) + *11*)^2 / *N*›

end

end

9 *Aux-Sturm-Calculation* – Auxiliary theory for technical reasons.

theory *Aux-Sturm-Calculation* **imports**
  Sturm-Sequences.Sturm
begin

We prove this fact in a separate theory because in *Collision.thy*, the *sturm* method fails with an internal error.

lemma *sturm-calculation*: ‹*12* ∗ (r^2+*154*)^*3* − (*10/3* ∗ (*r+2*)^*3* + *20*)^2 ≠ *0*› **if** ‹*r* ≥ *0*› **for** *r* :: *real*

end

10 *Collision* Invariant preservation for collision resistance

theory *Collision* **imports**
CO-Invariants
Oracle-Programs
Aux-Sturm-Calculation
begin

context *compressed-oracle* **begin**

definition ‹$no\text{-}collision = \{(x,y,D::'x \rightharpoonup 'y).\ inj\text{-}map\ D\}$›
definition ‹$no\text{-}collision' = \{D::'x \rightharpoonup 'y.\ inj\text{-}map\ D\}$›

lemma *no-collision-no-collision'*: ‹$no\text{-}collision = UNIV \times UNIV \times no\text{-}collision'$›

lemma *ket-invariant-no-collision-no-collision'*: ‹$ket\text{-}invariant\ no\text{-}collision = \top \otimes_S \top \otimes_S ket\text{-}invariant\ no\text{-}collision'$›

We show the preservation of the *no-collision* invariant. We show it with respect to the oracle *query* first.

lemma *preserves-no-collision*: ‹$preserves\text{-}ket\ query\ (no\text{-}collision \cap num\text{-}queries\ q)\ no\text{-}collision\ (6 * sqrt\ q\ /\ sqrt\ N)$›

Like *preserves-no-collision* but with respect to the oracle *query*.

lemma *preserves-no-collision'*: ‹$preserves\text{-}ket\ query'\ (no\text{-}collision \cap num\text{-}queries\ q)\ no\text{-}collision\ (5 * sqrt\ q\ /\ sqrt\ N)$›

lemma *preserves-no-collision-num*: ‹$preserves\text{-}ket\ query\ (no\text{-}collision \cap num\text{-}queries\ q)\ (no\text{-}collision \cap num\text{-}queries\ (q+1))\ (6 * sqrt\ q\ /\ sqrt\ N)$›

lemma *preserves-no-collision'-num*: ‹$preserves\text{-}ket\ query'\ (no\text{-}collision \cap num\text{-}queries\ q)\ (no\text{-}collision \cap num\text{-}queries\ (q+1))\ (5 * sqrt\ q\ /\ sqrt\ N)$›

10.1 Collision-finding is hard for q-query adversaries

lemma *collision-finding-is-hard*:
fixes *program* :: ‹$('mem, 'x, 'y)\ program$›
and *adv-output* :: ‹$('x \times 'x)\ update \Rightarrow 'mem\ update$›
and *initial-state*
assumes [*iff*]: ‹*valid-program program*›
assumes ‹$norm\ initial\text{-}state = 1$›
assumes [*register*]: ‹*register adv-output*›
shows ‹$(\sum h\in UNIV.\ \sum (x1,x2)|x1 \neq x2 \wedge h\ x1 = h\ x2.\ measurement\text{-}probability\ adv\text{-}output\ (exec\text{-}program\ h\ program\ initial\text{-}state)\ (x1,x2))\ /\ CARD('x \Rightarrow 'y)$
$\leq 12 * (query\text{-}count\ program + 154)\hat{\ }3\ /\ N$›

end

end

References

[1] Dominique Unruh. Compressed permutation oracles (and the collision-resistance of sponge/sha3). IACR Cryptology ePrint Archive, 2021/062, 2021.

[2] Dominique Unruh. Quantum and classical registers. Archive for Formal Proofs, https://www.isa-afp.org/entries/Registers.html, 2021. Formalization of parts of the present paper. For historic reasons, "references" are called "registers" and "disjoint" is called "compatible" in the formalization.

[3] Dominique Unruh. Quantum references. arXiv:2105.10914v3 [cs.LO], 2024.

[4] Mark Zhandry. How to record quantum queries, and applications to quantum indifferentiability. In *Crypto 2019*, pages 239–268. Springer, 2019. Eprint is IACR ePrint 2018/276.