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*)›
by (*metis assoc-ell2′-tensor tensor-ell2-ket*)

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

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*)
apply (*rule tensor-extensionality*)
by (*auto simp*: *unitary-sandwich-register assms sandwich-apply register-tensor-is-register comp-tensor-op tensor-op-adjoint unitary-tensor-op*)

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* $\Rightarrow_{CL}$ -)))
by (*simp add*: *unitary-sandwich-register assms sandwich-tensor id-def*)

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

ML ‹
```
fun register-n-of-m n m = let
  val - = n > 0 orelse error register-n-of-m: n<=0
  val - = m >= n orelse error register-n-of-m: n>m
  val id-op = Const(const-name ‹id-cblinfun›,dummyT)
  val tensor-op = Const(const-name ‹tensor-op›,dummyT)
  fun add-ids 0 t = t
    | add-ids i t = tensor-op $ id-op $ add-ids (i−1) t
  val body = if n=m then add-ids (n−1) (Bound 0)
                 else add-ids (n−1) (tensor-op $ Bound 0 $ add-ids (m−n−1) id-op)
in
  Abs(a, dummyT, body)
end
;;
register-n-of-m 5 5 |> Syntax.string-of-term context |> writeln
```
›

ML ‹
```
fun dest-numeral-syntax (Const(const-syntax ‹Num.num.One›, -)) = 1
  | dest-numeral-syntax (Const(const-syntax ‹Num.num.Bit0›, -) $ bs) = 2 ∗ dest-numeral-syntax bs
  | dest-numeral-syntax (Const (const-syntax ‹Num.num.Bit1›, -) $ bs) = 2 ∗ dest-numeral-syntax bs
+ 1
  | dest-numeral-syntax (Const (-constrain, -) $ t $ -) = dest-numeral-syntax t
  | dest-numeral-syntax t = raise TERM (dest-numeral-syntax, [t]);

fun dest-number-syntax (Const (const-syntax ‹Groups.zero-class.zero›, -)) = 0
  | dest-number-syntax (Const (const-syntax ‹Groups.one-class.one›, -)) = 1
  | dest-number-syntax (Const (const-syntax ‹Num.numeral-class.numeral›, -) $ t) =
      dest-numeral-syntax t
  | dest-number-syntax (Const (const-syntax ‹Groups.uminus-class.uminus›, -) $ t) =
      ~ (dest-number-syntax t)
  | dest-number-syntax (Const (-constrain, -) $ t $ -) = dest-number-syntax t
  | dest-number-syntax t = raise TERM (dest-number-syntax, [t])
```
›

syntax *-register-n-of-m* :: ‹′*a* ⇒ ′*a* ⇒ ′*b*› ([-_])
parse-translation ‹[(*syntax-const* ‹*-register-n-of-m*›, *fn ctxt* => *fn* [*nt*,*mt*] => *let*
```
  val n = dest-number-syntax nt
  val m = dest-number-syntax mt
  in register-n-of-m n m end
```
)]›
ML ‹
```
Syntax.read-term context [8₉] |> Thm.cterm-of context
```
›

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*))›
  by (*smt* (*verit*) *Finite-Cartesian-Product.sum-cong-aux*)

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*)›
  by (*smt* (*verit*) *Finite-Cartesian-Product.sum-cong-aux*)

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

lemma *tensor-ell2-diff2*: ‹*tensor-ell2 a* (*b* − *c*) = *tensor-ell2 a b* − *tensor-ell2 a c*›
  by (*metis add-diff-cancel-right' diff-add-cancel tensor-ell2-add2*)
lemma *tensor-ell2-diff1*: ‹*tensor-ell2* (*a* − *b*) *c* = *tensor-ell2 a c* − *tensor-ell2 b c*›
  by (*metis add-right-cancel diff-add-cancel tensor-ell2-add1*)

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

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*)
proof −
  have ‹*?lhs* = ($\sum x{\in}X \cap (Q$ −‘ *Some* ‘ *UNIV*). *P* (*case Q x of Some z* ⇒ *a z x* | *None* ⇒ *b x*)) + ($\sum x{\in}X \cap (Q$ −‘ {*None*}). *P* (*case Q x of Some z* ⇒ *a z x* | *None* ⇒ *b x*))›
    apply (*subst sum.union-disjoint*[*symmetric*])
    using *assms* **apply** *auto*
    by (*metis Int-UNIV-right Int-Un-distrib UNIV-option-conv insert-union vimage-UNIV vimage-Un*)
  also have ‹... = *?rhs*›
    apply (*rule arg-cong2*[**where** *f=plus*])
     apply (*rule sum.cong*)
      apply *auto*[*2*]
    apply (*rule sum.cong*)
    by *auto*
  finally show *?thesis*
    by −
qed

lemma *register-isometry*:
  assumes *register F*
  assumes *isometry a*
  shows *isometry* (*F a*)
  using *assms* **by** (*smt* (*verit*, *best*) *register-def isometry-def*)

lemma *register-coisometry*:
  assumes *register F*
  assumes *isometry* (*a*∗)
  shows *isometry* (*F a*∗)
  using *assms* **by** (*smt* (*verit*, *best*) *register-def isometry-def*)

lemma *card-complement*:
fixes M :: ‹$'a$::*finite set*›
shows ‹$card\ (-M) = CARD('a) - card\ M$›
by (*simp add*: *Compl-eq-Diff-UNIV card-Diff-subset*)

lemma *register-minus*: ‹$register\ F \Longrightarrow F\ (a - b) = F\ a - F\ b$›
using *clinear-register complex-vector.linear-diff* **by** *blast*

lemma *vimage-singleton-inj*: ‹$inj\ f \Longrightarrow f -` \{f\ x\} = \{x\}$›
using *inj-vimage-singleton subset-singletonD* **by** *fastforce*

lemma *has-ell2-norm-0*[*iff*]: ‹$has\text{-}ell2\text{-}norm\ (\lambda x.\ 0)$›
by (*metis eq-onp-same-args zero-ell2.rsp*)

lemma *ell2-norm-0I*[*simp*]: ‹$ell2\text{-}norm\ (\lambda x.\ 0) = 0$›
using *ell2-norm-0* **by** *blast*

lemma *ran-smaller-dom*: ‹$finite\ (dom\ m) \Longrightarrow card\ (ran\ m) \le card\ (dom\ m)$›
apply (*rule surj-card-le*[**where** f=‹*the o m*›], *simp*)
unfolding *dom-def ran-def* **by** *force*

lemma *option-sum-split*: ‹$(\sum x\in X.\ f\ x) = (\sum x\in Some -` X.\ f\ (Some\ x)) + (if\ None \in X\ then\ f\ None\ else\ 0)$› **if** ‹*finite X*› **for** *f X*
apply (*subst asm-rl*[*of* ‹$X = (Some\ `\ Some -`\ X) \cup (\{None\} \cap X)$›])
 apply *auto*[*1*]
apply (*subst sum.union-disjoint*)
   apply (*auto simp*: *that*)[*3*]
apply (*subst sum.reindex*)
by *auto*

lemma *sum-sum-if-eq*: ‹$(\sum x\in X.\ \sum y\in Y\ x.\ if\ x{=}a\ then\ f\ x\ y\ else\ 0) = (if\ a\in X\ then\ (\sum y\in Y\ a.\ f\ a\ y)\ else\ 0)$› **if** ‹*finite X*› **for** *X Y f*
by (*subst sum-single*[**where** $i{=}a$], *auto simp*: *that*)

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*
apply (*cases* ‹*finite X*›)
 apply (*rule sum.mono-neutral-cong-right*)
by *auto*

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

definition *reg-1-3* :: ‹$- \Rightarrow ('a::finite \times 'b::finite \times 'c::finite)\ ell2 \Rightarrow_{CL} ('a \times 'b \times 'c)\ ell2$› **where** ‹$reg\text{-}1\text{-}3 = Fst$›
lemma *register-1-3*[*simp*]: ‹*register reg-1-3*›
by (*simp add*: *reg-1-3-def*)

lemma *comp-reg-1-3*[*simp*]: ‹$(F;G)\ o\ reg\text{-}1\text{-}3 = F$› **if** [*register*]: ‹*compatible F G*›
by (*simp add*: *reg-1-3-def register-pair-Fst*)

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*›
  by (*simp add*: *reg-2-3-def*)
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*›
  by (*simp add*: *reg-2-3-def register-pair-Fst register-pair-Snd flip*: *comp-assoc*)

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*›
  by (*simp add*: *reg-3-3-def*)
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*›
  by (*simp add*: *reg-3-3-def register-pair-Snd flip*: *comp-assoc*)

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

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

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*)›
  apply (*subst register-tensor-distrib*)
  by *auto*

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

lemma *csubspace-set-sum*:
  assumes ‹$\bigwedge$*x*. *x* ∈ *X* ⟹ *csubspace* (*A x*)›
  shows ‹*csubspace* ($\sum$*x*∈*X*. *A x*)›
  using *assms*
  apply (*induction X rule*:*infinite-finite-induct*)
  by (*auto simp*: *csubspace-set-plus*)

lemma *Rep-ell2-vector-to-cblinfun-ket*: ‹*Rep-ell2* ψ *x* = *bra x* $*_V$ ψ›
  by (*simp add*: *cinner-ket-left*)

lemma *trunc-ell2-insert*: ‹*trunc-ell2* (*insert x M*) ψ = *Rep-ell2* ψ *x* $*_C$ *ket x* + *trunc-ell2 M* ψ› **if** ‹*x* ∉ *M*›
  using *trunc-ell2-union-disjoint*[**where** *M*=‹{*x*}› **and** *N*=*M* **and** ψ=ψ] *that*
  by (*auto simp*: *trunc-ell2-singleton*)

lemma *trunc-ell2-in-cspan*:
  assumes ‹*finite S*›
  shows ‹*trunc-ell2 S* ψ ∈ *cspan* (*ket* ‘ *S*)›
  using *assms*

proof *induction*
  case *empty*
  show *?case*
    by *simp*
next
  case (*insert x F*)
  then have ‹*Rep-ell2 ψ x* $*_C$ *ket x* + *trunc-ell2 F ψ* ∈ *cspan* (*insert* (*ket x*) (*ket* ‘ *F*))›
    by (*metis add-diff-cancel-left′ complex-vector.span-breakdown-eq*)
  with *insert* **show** *?case*
    by (*auto simp*: *trunc-ell2-insert*)
qed

lemma *space-ccspan-ket*: ‹*space-as-set* (*ccspan* (*ket* ‘ *M*)) = {ψ. ∀ *x* ∈ −*M*. *Rep-ell2 ψ x* = *0*}›
proof (*intro Set.set-eqI iffI*; *rename-tac ψ*)
  fix ψ
  assume *ψ-in-ccspan*: ‹ψ ∈ *space-as-set* (*ccspan* (*ket* ‘ *M*))›
  have ‹*Rep-ell2 ψ x* = *0*› **if** ‹*x* ∈ −*M*› **for** *x*
  proof −
    have ‹*Rep-ell2 ψ x* = *vector-to-cblinfun* (*ket x*)∗ $*_V$ ψ›
      by (*simp add*: *Rep-ell2-vector-to-cblinfun-ket*)
    also have ‹. . . ∈ *vector-to-cblinfun* (*ket x*)∗ ‘ *space-as-set* (*ccspan* (*ket* ‘ *M*))›
      using *ψ-in-ccspan* **by** *blast*
    also have ‹. . . ⊆ *space-as-set* (*vector-to-cblinfun* (*ket x*)∗ $*_S$ *ccspan* (*ket* ‘ *M*))›
      by (*simp add*: *cblinfun-image.rep-eq closure-subset*)
    also have ‹. . . = *space-as-set* (*ccspan* (*vector-to-cblinfun* (*ket x*)∗ ‘ *ket* ‘ *M*))›
      by (*simp add*: *cblinfun-image-ccspan*)
    also have ‹. . . = *space-as-set* (*ccspan* (*if M*={} *then* {} *else* {*0*}))›
      apply (*rule arg-cong*[**where** *f*=‹λ*x*. *space-as-set* (*ccspan x*)›])
      using ‹*x* ∈ −*M*› **apply** *auto*
      by (*metis imageI orthogonal-ket*)
    also have ‹. . . = *0*›
      by *simp*
    finally show ‹*Rep-ell2 ψ x* = *0*›
      by *auto*
  qed
  then show ‹ψ ∈ {ψ. ∀ *x*∈− *M*. *Rep-ell2 ψ x* = *0*}›
    by *simp*
next
  fix ψ
  assume ‹ψ ∈ {ψ. ∀ *x*∈− *M*. *Rep-ell2 ψ x* = *0*}›
  then have ‹ψ = *trunc-ell2 M ψ*›
    by (*auto intro*!: *Rep-ell2-inject*[*THEN iffD1*] *ext simp*: *trunc-ell2.rep-eq*)
  then have *lim*: ‹((λ*S*. *trunc-ell2 S ψ*) ⟶ ψ) (*finite-subsets-at-top M*)›
    using *trunc-ell2-lim*[*of ψ M*]
    by *auto*
  have ‹*trunc-ell2 S ψ* ∈ *cspan* (*ket* ‘ *S*)› **if** ‹*finite S*› **for** *S*
    by (*simp add*: *that trunc-ell2-in-cspan*)
  also have ‹. . . *S* ⊆ *space-as-set* (*ccspan* (*ket* ‘ *M*))› **if** ‹*finite S*› **and** ‹*S* ⊆ *M*› **for** *S*
   by (*metis ccspan.rep-eq closure-subset complex-vector.span-mono dual-order.trans image-mono that*(*2*))
  finally show ‹ψ ∈ *space-as-set* (*ccspan* (*ket* ‘ *M*))›
    apply (*rule-tac Lim-in-closed-set*[*OF* - - - *lim*])
    by (*auto intro*!: *eventually-finite-subsets-at-top-weakI*)
qed

lemma *space-as-set-ccspan-memberI*: ‹ψ ∈ *space-as-set* (*ccspan X*)› **if** ‹ψ ∈ *X*›

using *that* **apply** *transfer*
by (*meson closure-subset complex-vector.span-superset subset-iff*)

lemma *closure-subset-remove-closure*: ‹$X \subseteq$ *closure* $Y \Longrightarrow$ *closure* $X \subseteq$ *closure* Y›
using *closure-minimal* **by** *blast*

lemma *closure-cspan-closure*: ‹*closure* (*cspan* (*closure* X)) = *closure* (*cspan* X)›
for X :: ‹′*a*::*complex-normed-vector set*›
apply (*rule antisym*)
apply (*meson closure-subset-remove-closure closure-is-csubspace closure-mono complex-vector.span-minimal*

complex-vector.span-superset complex-vector.subspace-span)
by (*simp add*: *closure-mono closure-subset complex-vector.span-mono*)

lemma *closure-Sup-closure*: ‹*closure* (*Sup* (*closure* ‘ X)) = *closure* (*Sup* X)›
by (*auto simp*: *hull-def closure-hull*)

lemma *cspan-closure-cspan*: ‹*cspan* (*closure* (*cspan* X)) = *closure* (*cspan* X)›
for X :: ‹′*a*::*complex-normed-vector set*›
by (*metis* (*full-types*) *closure-cspan-closure closure-subset complex-vector.span-span complex-vector.span-superset subset-antisym*)

lemma *cblinfun-image-SUP*: ‹$A *_S$ (*SUP* $x \in X$. I x) = (*SUP* $x \in X$. $A *_S I$ x)›
proof (*rule antisym*)
show ‹$A *_S$ (*SUP* $x \in X$. I x) $\leq$ (*SUP* $x \in X$. $A *_S I$ x)›
proof (*transfer*, *rule closure-subset-remove-closure*)
fix A :: ‹′$b \Rightarrow$ ′a› **and** I :: ‹′$c \Rightarrow$ ′b *set*› **and** X
assume [*simp*]: ‹*bounded-clinear* A›
assume ‹*pred-fun top closed-csubspace* I›
then have [*simp*]: ‹*closed-csubspace* (I x)› **for** x
by *simp*
have ‹A ‘ *closure* (*cspan* ($\bigcup x \in X$. I x)) $\subseteq$ *closure* (A ‘ *cspan* ($\bigcup x \in X$. I x))›
apply (*rule closure-bounded-linear-image-subset*)
by (*simp add*: *bounded-clinear.bounded-linear*)
also have ‹... = *closure* (*cspan* (A ‘ ($\bigcup x \in X$. I x)))›
by (*simp add*: *bounded-clinear.clinear complex-vector.linear-span-image*)
also have ‹... = *closure* (*cspan* ($\bigcup x \in X$. A ‘ I x))›
by (*metis image-UN*)
also have ‹... = *closure* (*cspan* (*closure* ($\bigcup x \in X$. A ‘ I x)))›
using *closure-cspan-closure* **by** *blast*
also have ‹... = *closure* (*cspan* (*closure* ($\bigcup x \in X$. *closure* (A ‘ I x))))›
apply (*subst closure-Sup-closure*[*symmetric*])
by (*simp add*: *image-image*)
also have ‹... = *closure* (*cspan* ($\bigcup x \in X$. *closure* (A ‘ I x)))›
using *closure-cspan-closure* **by** *blast*
finally show ‹A ‘ *closure* (*cspan* ($\bigcup$ (I ‘ X))) $\subseteq$ *closure* (*cspan* ($\bigcup x \in X$. *closure* (A ‘ I x)))›
by −
qed

show ‹(*SUP* $x \in X$. $A *_S I$ x) $\leq A *_S$ (*SUP* $x \in X$. I x)›
by (*simp add*: *SUP-least SUP-upper cblinfun-image-mono*)
qed

lemma *cspan-Sup-cspan*: ‹*cspan* (*Sup* (*cspan* ‘ X)) = *cspan* (*Sup* X)›
by (*auto simp*: *hull-def complex-vector.span-def*)

```
lemma ccspan-Sup: ‹ccspan (⋃X) = Sup (ccspan ‘ X)›
proof (transfer fixing: X)
  have ‹closure (cspan (⋃X)) = closure (cspan (⋃ (cspan ‘ X)))›
    by (simp add: cspan-Sup-cspan)
  also have ‹... = closure (cspan (closure (⋃ (cspan ‘ X))))›
    using closure-cspan-closure by blast
  also have ‹... = closure (cspan (closure (⋃ (closure ‘ cspan ‘ X))))›
    by (metis closure-Sup-closure)
  also have ‹... = closure (cspan (⋃ (closure ‘ cspan ‘ X)))›
    by (meson closure-cspan-closure)
  also have ‹... = closure (cspan (⋃G∈X. closure (cspan G)))›
    by (metis image-image)
  finally show ‹closure (cspan (⋃ X)) = closure (cspan (⋃G∈X. closure (cspan G)))›
    by −
qed

lemma ccspan-space-as-set[simp]: ‹ccspan (space-as-set S) = S›
  apply transfer
  by (metis closed-csubspace-def closure-closed complex-vector.span-eq-iff)

lemma cblinfun-image-Sup: ‹A *S Sup II = (SUP I∈II. A *S I)›
  using cblinfun-image-SUP[where X=II and I=id and A=A]
  by simp

lemma space-as-set-mono: ‹I ≤ J ⟹ space-as-set I ≤ space-as-set J›
  by (simp add: less-eq-ccsubspace.rep-eq)

lemma square-into-sum:
  fixes X Y and f :: ‹- ⇒ real›
  assumes ‹⋀x. f x ≥ 0›
  shows ‹(∑x∈X. f x)² ≤ card X * (∑x∈X. (f x)²)›
proof −
  have ‹(∑x∈X. f x)² = (∑x∈X. f x * 1)²›
    by simp
  also have ‹... ≤ (∑x∈X. (f x)²) * (∑x∈X. 1²)›
    by (rule Cauchy-Schwarz-ineq-sum)
  also have ‹... = (∑x∈X. (f x)²) * (card X)›
    by simp
  also have ‹... = card X * (∑x∈X. (f x)²)›
    by auto
  finally show ?thesis
    by −
qed

lemma bound-coeff-sum2:
  fixes X Y and f :: ‹'a ⇒ complex›
  assumes [simp]: ‹finite Y›
  assumes XY: ‹X ⊆ Y›
  assumes sum1: ‹(∑x∈Y. (cmod (f x))²) ≤ 1›
  assumes k: ‹⋀x. x ∈ X ⟹ card {y. g x = g y ∧ y ∈ X} ≤ k›
  shows ‹norm (∑x∈X. f x *C ket (g x)) ≤ sqrt k›
proof −
  define eq where ‹eq = {(x,y). g x = g y ∧ x ∈ X ∧ y ∈ X}›
  have [simp]: ‹equiv X eq›
```

```
  by (auto simp: eq-def equiv-def refl-on-def sym-def trans-def)
have [simp]: ‹finite X›
  using ‹finite Y› XY infinite-super by blast
then have [simp]: ‹finite (X // eq)›
  by (simp add: equiv-type finite-quotient)
have [simp]: ‹x ∈ X // eq ⟹ finite x› for x
  by (meson ‹equiv X eq› ‹finite X› equiv-def finite-equiv-class refl-on-def)
have card-c: ‹c ∈ X//eq ⟹ card c ≤ k› for c
  using k
  by (auto simp: Image-def quotient-def eq-def)

define g' where ‹g' c = g (SOME x. x∈c)› for c :: ‹'a set›
have g-g': ‹c ∈ X//eq ⟹ x ∈ c ⟹ g x = g' c› for x c
  apply (simp add: g'-def quotient-def eq-def)
  by (metis (mono-tags, lifting) mem-Collect-eq verit-sko-ex)
have g'-inj: ‹c ∈ X//eq ⟹ d ∈ X//eq ⟹ g' c = g' d ⟹ c = d› (is ‹PROP ?goal›) for c d
proof -
  have aux1: ‹⋀x xa xb.
      g (SOME x. g xb = g x ∧ x ∈ X) = g (SOME x. g xa = g x ∧ x ∈ X) ⟹
      xa ∈ X ⟹ xb ∈ X ⟹ g xa = g xb›
    by (metis (mono-tags, lifting) verit-sko-ex)
  have aux2: ‹⋀x xa xb.
      g (SOME xa. g x = g xa ∧ xa ∈ X) = g (SOME x. g xb = g x ∧ x ∈ X) ⟹
      x ∈ X ⟹ xb ∈ X ⟹ g x = g xb›
    by (metis (mono-tags, lifting) someI2)
  show ‹PROP ?goal›
    by (auto intro: aux1 aux2 simp add: g'-def quotient-def eq-def image-iff)
qed

have SIGMA: ‹(SIGMA x:X // eq. x) = (λx. (eq``{x},x)) ` X›
  by (auto simp: quotient-def eq-def)

have ‹(norm (∑x∈X. f x *C ket (g x)))^2 = (norm (∑c∈X//eq. ∑x∈c. f x *C ket (g x)))^2›
  apply (subst sum.Sigma)
    apply auto[2]
  apply (subst SIGMA)
  apply (subst sum.reindex)
  using inj-on-def by auto
also have ‹... = (norm (∑c∈X//eq. ∑x∈c. f x *C ket (g' c)))^2›
  by (simp add: g-g')
also have ‹... = (norm (∑c∈X//eq. (∑x∈c. f x) *C ket (g' c)))^2›
  by (simp add: scaleC-sum-left)
also have ‹... = (∑c∈X//eq. (norm ((∑x∈c. f x) *C ket (g' c)))^2)›
  apply (rule pythagorean-theorem-sum)
  by (auto dest: g'-inj)
also have ‹... = (∑c∈X//eq. (cmod (∑x∈c. f x))^2)›
  by force
also have ‹... ≤ (∑c∈X//eq. (∑x∈c. cmod (f x))^2)›
  by (simp add: power-mono sum-mono sum-norm-le)
also have ‹... ≤ (∑c∈X//eq. card c * (∑x∈c. (cmod (f x))^2))›
  apply (rule sum-mono)
  apply (rule square-into-sum)
  by simp
also have ‹... ≤ (∑c∈X//eq. k * (∑x∈c. (cmod (f x))^2))›
  apply (rule sum-mono)
```

```
    apply (rule mult-right-mono)
    by (simp-all add: card-c sum-nonneg)
  also have ‹... = k * (∑c∈X//eq. (∑x∈c. (cmod (f x))²))›
    by (rule sum-distrib-left[symmetric])
  also have ‹... ≤ k * (∑x∈X. (cmod (f x))²)›
    apply (subst sum.Sigma)
      apply auto[2]
    apply (subst SIGMA)
    apply (subst sum.reindex)
    using inj-on-def by auto
  also have ‹... ≤ k * (∑x∈Y. (cmod (f x))²)›
    apply (rule mult-left-mono)
     apply (rule sum-mono2)
    using XY by auto
  also have ‹... ≤ k›
    using sum1
    by (metis mult.right-neutral mult-left-mono of-nat-0-le-iff)
  finally show ?thesis
    using real-le-rsqrt by blast
qed


lemma norm-ortho-sum-bound:
  fixes X
  assumes bound: ‹⋀x. x∈X ⟹ norm (ψ x) ≤ b'›
  assumes b'geq0: ‹b' ≥ 0›
  assumes ortho: ‹⋀x y. x∈X ⟹ y∈X ⟹ x≠y ⟹ is-orthogonal (ψ x) (ψ y)›
  assumes b'b: ‹sqrt (card X) * b' ≤ b›
  shows ‹norm (∑x∈X. ψ x) ≤ b›
proof (cases ‹finite X›)
  case True
  have ‹b ≥ 0›
    by (metis b'b b'geq0 mult-nonneg-nonneg of-nat-0-le-iff order-trans real-sqrt-ge-0-iff)
  have ‹(norm (∑x∈X. ψ x))² = (∑a∈X. (norm (ψ a))²)›
    apply (subst pythagorean-theorem-sum)
    using assms True by auto
  also have ‹... ≤ (∑a∈X. b'²)›
    by (meson bound norm-ge-zero power-mono sum-mono)
  also have ‹... ≤ (sqrt (card X) * b')²›
    by (simp add: power-mult-distrib)
  also have ‹... ≤ b²›
    by (meson b'b b'geq0 mult-nonneg-nonneg of-nat-0-le-iff power-mono real-sqrt-ge-0-iff)
  finally show ?thesis
    apply (rule-tac power2-le-imp-le)
    apply force
    using ‹0 ≤ b› by force
next
  case False
  then show ?thesis
    using assms by auto
qed

lemma compatible-project1: ‹compatible F G›
  if ‹compatible F (G;H)› and [register]: ‹compatible G H› for F G H
proof (rule compatibleI)
```

show ‹*register F*›
  using *compatible-register1 that*(*1*) **by** *blast*
show ‹*register G*›
  using *compatible-register1 that*(*2*) **by** *blast*
fix *a b*
from ‹*compatible F* (*G*;*H*)›
have ‹$F\ a\ o_{CL}\ (G;H)\ (b \otimes_o \textit{id-cblinfun}) = (G;H)\ (b \otimes_o \textit{id-cblinfun})\ o_{CL}\ F\ a$›
  using *swap-registers* **by** *blast*
then show ‹$F\ a\ o_{CL}\ G\ b = G\ b\ o_{CL}\ F\ a$›
  by (*simp add*: *register-pair-apply*)
qed

lemma *compatible-project2*: ‹*compatible F H*›
  if ‹*compatible F* (*G*;*H*)› **and** [*register*]: ‹*compatible G H*› **for** *F G H*
proof (*rule compatibleI*)
  show ‹*register F*›
    using *compatible-register1 that*(*1*) **by** *blast*
  show ‹*register H*›
    using *compatible-register2 that*(*2*) **by** *blast*
  fix *a b*
  from ‹*compatible F* (*G*;*H*)›
  have ‹$F\ a\ o_{CL}\ (G;H)\ (\textit{id-cblinfun} \otimes_o b) = (G;H)\ (\textit{id-cblinfun} \otimes_o b)\ o_{CL}\ F\ a$›
    using *swap-registers* **by** *blast*
  then show ‹$F\ a\ o_{CL}\ H\ b = H\ b\ o_{CL}\ F\ a$›
    by (*simp add*: *register-pair-apply*)
qed

lemma *proj-ket-x-y*: ‹*proj* (*ket x*) $*_V$ (*ket y*) = *0*› **if** ‹$x \neq y$›
  by (*metis kernel-Proj kernel-memberD mem-ortho-ccspanI orthogonal-ket singletonD that*)

lemma *proj-ket-x-y-ofbool*: ‹*proj* (*ket x*) $*_V$ (*ket y*) = *of-bool* (*x*=*y*) $*_C$ *ket y*›
  by (*simp add*: *Proj-fixes-image ccspan-superset′ proj-ket-x-y*)

lemma *proj-x-x*[*simp*]: ‹*proj x* $*_V$ *x* = *x*›
  by (*meson Proj-fixes-image ccspan-superset insert-subset*)

lemma *in-ortho-ccspan*: ‹$y \in$ *space-as-set* (− *ccspan X*)› **if** ‹$\forall x \in X$. *is-orthogonal y x*›
  using *that* **apply** *transfer*
  by (*metis orthogonal-complementI orthogonal-complement-of-closure orthogonal-complement-of-cspan*)

lemma *swap-sandwich-swap-ell2*: *swap* = *sandwich swap-ell2*
  apply (*rule tensor-extensionality*)
    apply (*auto simp*: *sandwich-apply unitary-sandwich-register*)[*2*]
  apply (*rule tensor-ell2-extensionality*)
  by *simp*

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*›
  using *that*
  by (*simp add*: *is-Proj-algebraic sandwich-apply*

lift-cblinfun-comp[*OF isometryD*] *lift-cblinfun-comp*[*OF is-Proj-idempotent*] *flip*: *cblinfun-compose-assoc*)

lemma *is-Proj-swap*[*simp*]: ‹*is-Proj* (*swap P*)› **if** ‹*is-Proj P*›
using *that*
by (*simp add*: *swap-sandwich-swap-ell2 is-Proj-sandwich*)

lemma *iso-register-complement-pair*: ‹*iso-register* (*complement X*; *X*)› **if** ‹*register X*›
using *complement-is-complement complements-def complements-sym that* **by** *blast*

lemma *swap-Snd*: ‹*swap* (*Snd x*) = *Fst x*›
by (*simp add*: *Fst-def Snd-def*)

lemma *sandwich-butterfly*: ‹*sandwich a* (*butterfly g h*) = *butterfly* (*a g*) (*a h*)›
by (*simp add*: *butterfly-comp-cblinfun cblinfun-comp-butterfly sandwich-apply*)

lemma *register0*:
assumes ‹*register Q*›
shows ‹$Q\ a = 0 \longleftrightarrow a = 0$›
by (*metis assms norm-eq-zero register-norm*)

lemma *le-back-subst*:
assumes ‹$a \leq c$›
assumes ‹$a = b$›
shows ‹$b \leq c$›
using *assms* **by** *simp*

lemma *le-back-subst-le*:
fixes *a b c* :: ‹- :: *order*›
assumes ‹$a \leq c$›
assumes ‹$b \leq a$›
shows ‹$b \leq c$›
using *assms* **by** *order*

lemma *arg-cong4*: ‹$f\ a\ b\ c\ d = f\ a'\ b'\ c'\ d'$› **if** ‹$a = a'$› **and** ‹$b = b'$› **and** ‹$c = c'$› **and** ‹$d = d'$›
using *that* **by** *simp*

1.2 Controlled operations

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

lemma *trunc-ell2-prod-tensor*: ‹*trunc-ell2* ($A \times B$) ($g \otimes_s h$) = *trunc-ell2 A g* $\otimes_s$ *trunc-ell2 B h*›
apply *transfer'*
by *auto*

lemma *trunc-ell2-ket*: ‹*trunc-ell2 S* (*ket x*) = *of-bool* ($x \in S$) $*_C$ *ket x*›
apply *transfer'*
by *auto*

lemma *summable-on-in-0*[*iff*]: ‹*summable-on-in* T (λx. *0*) A› **if** ‹*0* ∈ *topspace* T›
  using *has-sum-in-0*[*of* T A ‹λ-. *0*›] *summable-on-in-def that* **by** *blast*

lemma *sum-of-bool-scaleC*: ‹($\sum x{\in}S$. *of-bool* ($x{=}a$) $*_C$ f x) = (*if* $a{\in}S$ ∧ *finite* S *then* f a *else* *0*)›
  for f :: ‹- ⇒ -::*complex-vector*›
  apply (*cases* ‹*finite* S›)
   apply (*subst sum-single*[**where** $i{=}a$])
  by *auto*

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

  assumes ‹$\bigwedge x$. *norm* (A x) ≤ B›
  shows *controlled-op-has-sum-aux*: ‹*has-sum-in cstrong-operator-topology* (λx. *selfbutter* (*ket* x) $\otimes_o$ A x) *UNIV* (*controlled-op* A)›
    and *controlled-op-norm-leq*: ‹*norm* (*controlled-op* A) ≤ B›
proof −
  have [*iff*]: ‹B ≥ *0*›
    using *assms*[*of undefined*] *norm-ge-zero*[*of* ‹A *undefined*›]
    by *argo*

  define A' **where** ‹A' x = *selfbutter* (*ket* x) $\otimes_o$ A x› **for** x
  have A'*summ*: ‹(λx. A' x $*_V$ h) *summable-on UNIV*› **for** h
  proof −
    wlog [*iff*]: ‹B ≠ *0*›
      using *negation assms* **by** (*simp add*: A'-*def*)
    have ‹∃ P. *eventually* P (*finite-subsets-at-top UNIV*) ∧ (∀ F G. P F ∧ P G ⟶ *dist* ($\sum x{\in}F$. A' x $*_V$ h) ($\sum x{\in}G$. A' x $*_V$ h) < e)› **if** ‹e > *0*› **for** e
    proof −
      have ‹((λS. *trunc-ell2* S h) ⟶ h) (*finite-subsets-at-top UNIV*)›
        by (*rule trunc-ell2-lim-at-UNIV*)
      from *tendsto-iff*[*THEN iffD1*, *OF this*, *rule-format*, *of* ‹e/B›]
      have ‹$\forall_F$ S *in finite-subsets-at-top UNIV*. *dist* (*trunc-ell2* S h) h < e/B›
        using ‹B ≠ *0*› ‹B ≥ *0*› *that* **by** *force*
      then obtain S **where** [*iff*]: ‹*finite* S› **and** *S-prop*: ‹*norm* (*trunc-ell2* S h − h) < e/B› **for** G
        apply *atomize-elim*
        by (*force simp add*: *eventually-finite-subsets-at-top dist-norm*)
      define P :: ‹$'x$ *set* ⇒ *bool*› **where** ‹P F ⟷ *finite* F ∧ F ⊇ *fst* ‘ S› **for** F
      have *evP*: ‹*eventually* P (*finite-subsets-at-top UNIV*)›
       by (*auto intro*!: *exI*[*of* - ‹*fst*‘S›] ‹*finite* S› *simp add*: *eventually-finite-subsets-at-top P-def*[*abs-def*])
      have ‹*dist* ($\sum x{\in}F$. A' x $*_V$ h) ($\sum x{\in}G$. A' x $*_V$ h) < e› **if** ‹P F› **and** ‹P G› **for** F G
      proof −
        have [*iff*]: ‹*finite* F› ‹*finite* G›
          using *that* **by** (*simp-all add*: *P-def*)
        define *FG* **where** ‹*FG* = *sym-diff* F G›
        then have [*iff*]: ‹*finite FG*›
          by *simp*
        define h' **where** ‹h' x = (*tensor-ell2-left* (*ket* x)∗) h› **for** x
        have $A'h$: ‹A' x $*_V$ h = *ket* x $\otimes_s$ (A x $*_V$ h' x)› **for** x
          unfolding h'-*def*
          apply (*rule fun-cong*[*of* - - h])
          apply (*rule bounded-clinear-equal-ket*)
            apply (*auto intro*!: *bounded-linear-intros*)[*2*]
          by (*auto simp add*: A'-*def tensor-op-ket tensor-op-ell2 cinner-ket simp flip*: *tensor-ell2-ket*)

```
    have ‹(dist (∑x∈F. A' x *V h) (∑x∈G. A' x *V h))² = (norm ((∑x∈F. A' x *V h) − (∑x∈G.
  A' x *V h)))²›
      by (simp add: dist-norm)
    also have ‹... = (norm ((∑x∈FG. (if x∈F then 1 else −1) *C (A' x *V h))))²›
      apply (rule arg-cong[where f=‹λx. (norm x)²›])
      apply (rewrite at F at ‹∑x∈⌑. -› to ‹(F−G)∪(F∩G)› DEADID.rel-mono-strong, blast)
      apply (rewrite at G at ‹∑x∈⌑. -› to ‹(G−F)∪(F∩G)› DEADID.rel-mono-strong, blast)
      apply (rewrite at FG at ‹∑x∈⌑. -› FG-def)
      apply (subst sum-Un, simp, simp)
      apply (subst sum-Un, simp, simp)
      apply (subst sum-Un, simp, simp)
      apply (rewrite at ‹(∑x∈F − G. (if x ∈ F then 1 else − 1) *C (A' x *V h))› to ‹(∑x∈F−G.
  A' x *V h)› sum.cong, simp, simp)
      apply (rewrite at ‹(∑x∈G − F. (if x ∈ F then 1 else − 1) *C (A' x *V h))› to ‹(∑x∈G−F.
  − (A' x *V h))› sum.cong, simp, simp)
      apply (rewrite at ‹(F − G) ∩ (G − F)› to ‹{}› DEADID.rel-mono-strong, blast)
      apply (rewrite at ‹(F − G) ∩ (F ∩ G)› to ‹{}› DEADID.rel-mono-strong, blast)
      apply (rewrite at ‹(G − F) ∩ (F ∩ G)› to ‹{}› DEADID.rel-mono-strong, blast)
      by (simp add: sum-negf)
    also have ‹... = (∑x∈FG. (norm ((if x ∈ F then 1 else − 1) *C (A' x *V h)))²)›
      apply (rule pythagorean-theorem-sum)
       apply (simp add: A'-def butterfly-adjoint tensor-op-adjoint comp-tensor-op cinner-ket
          flip: cinner-adj-right cblinfun-apply-cblinfun-compose)
      by (simp add: FG-def)
    also have ‹... = (∑x∈FG. (norm (A' x *V h))²)›
      apply (rule sum.cong, simp)
      by (simp add: norm-scaleC)
    also have ‹... = (∑x∈FG. (norm (A x *V h' x))²)›
      by (simp add: A'h norm-tensor-ell2)
    also have ‹... ≤ (∑x∈FG. (B * norm (h' x))²)›
      using assms
      by (auto intro!: sum-mono power-mono norm-cblinfun[THEN order-trans] mult-right-mono)
    also have ‹... = B² * (∑x∈FG. (norm (h' x))²)›
      by (simp add: sum-distrib-left power-mult-distrib)
    also have ‹... = B² * (∑x∈FG. (norm (ket x ⊗s h' x))²)›
      by (simp add: norm-tensor-ell2 norm-ket)
    also have ‹... = B² * (norm (∑x∈FG. ket x ⊗s h' x))²›
      apply (subst pythagorean-theorem-sum)
      by (simp-all add: FG-def)
    also have ‹... = B² * (norm (trunc-ell2 (FG×UNIV) h))²›
      apply (rule arg-cong[where f=‹λx. - * (norm x)²›])
      apply (rule cinner-ket-eqI)
      apply (rename-tac ab)
    proof −
      fix ab :: ‹'x × 'a›
      obtain a b where ab: ‹ab = (a,b)›
        by (auto simp: prod-eq-iff)
      have ‹ket ab •C (∑x∈FG. ket x ⊗s h' x) = (∑x∈FG. ket ab •C (ket x ⊗s h' x))›
        using cinner-sum-right by blast
      also have ‹... = of-bool (a∈FG) * (ket b •C h' a)›
        apply (subst sum-single[where i=a])
        by (auto simp add: ab simp flip: tensor-ell2-ket)
      also have ‹... = of-bool (a∈FG) * Rep-ell2 h (a, b)›
        by (simp add: h'-def cinner-adj-right tensor-ell2-ket cinner-ket-left)
      also have ‹... = ket ab •C trunc-ell2 (FG × UNIV) h›
```

```
        by (simp add: ab cinner-ket-left trunc-ell2.rep-eq)
      finally show ‹ket ab •C (∑x∈FG. ket x ⊗s h′ x) = ket ab •C trunc-ell2 (FG × UNIV) h›
        by −
    qed
    also have ‹... ≤ B² ∗ (norm (trunc-ell2 (−S) h))²›
      apply (rule mult-left-mono[rotated], simp)
      apply (rule power-mono[rotated], simp)
      apply (rule trunc-ell2-norm-mono)
      using ‹P F› ‹P G› by (force simp: P-def FG-def)
    also have ‹... = B² ∗ (norm (trunc-ell2 S h − h))²›
      by (smt (verit, best) norm-id-minus-trunc-ell2 norm-minus-commute trunc-ell2-uminus)
    also have ‹... < B² ∗ (e/B)²›
      apply (rule mult-strict-left-mono[rotated], simp)
      apply (rule power-strict-mono[rotated], simp, simp)
      by (rule S-prop)
    also have ‹... = e²›
      by (simp add: power-divide)
    finally show ?thesis
      by (smt (verit, del-insts) ‹0 < e› power-mono)
  qed
  with evP show ?thesis
    by blast
qed
then show ?thesis
  unfolding summable-on-def has-sum-def filterlim-def
  apply (rule-tac convergent-filter-iff[THEN iffD1])
  apply (subst convergent-filter-iff-cauchy)
  by (rule cauchy-filter-metric-filtermapI)
qed
define C where ‹C h = (∑∞x. A′ x ∗V h)› for h
then have C-hassum: ‹((λx. A′ x ∗V h) has-sum (C h)) UNIV› for h
  using A′summ by auto

have norm-C: ‹norm (C g) ≤ B ∗ norm g› for g
proof −
  define g′ where ‹g′ x = (tensor-ell2-left (ket x)∗) g› for x
  have A′g: ‹A′ x ∗V g = ket x ⊗s (A x ∗V g′ x)› for x
    unfolding g′-def
    apply (rule fun-cong[of - - g])
    apply (rule bounded-clinear-equal-ket)
      apply (simp add: cblinfun.bounded-clinear-right)
     apply (metis bounded-clinear-compose bounded-clinear-tensor-ell21 cblinfun.bounded-clinear-right)

    by (auto simp add: A′-def tensor-op-ket tensor-op-ell2 cinner-ket simp flip: tensor-ell2-ket)
  have norm-trunc: ‹norm (trunc-ell2 F (C g)) ≤ B ∗ norm g› if ‹finite F› for F
  proof −
    define S where ‹S = fst ‘ F›
    then have [iff]: ‹finite S›
      using that by blast
    have ‹(norm (trunc-ell2 F (C g)))² ≤ (norm (trunc-ell2 (S × UNIV) (C g)))²›
      apply (rule power-mono[rotated], simp)
      apply (rule trunc-ell2-norm-mono)
      by (force simp: S-def)
    also have ‹... = (norm (∑x∈S. ket x ⊗s (A x ∗V g′ x)))²›
    proof (rule arg-cong[where f=‹λx. (norm x)²›])
```

```
      have ‹trunc-ell2 (S×UNIV) (C g) = (∑∞x. trunc-ell2 (S×UNIV) (A′ x *V g))›
        apply (simp add: C-def)
        apply (rule infsum-bounded-linear[symmetric])
         apply (intro bounded-clinear.bounded-linear bounded-clinear-trunc-ell2)
        using A′summ by −
      also have ‹... = (∑∞x∈S. ket x ⊗s (A x *V g′ x))›
        apply (rule infsum-cong-neutral)
        by (auto simp add: A′g trunc-ell2-prod-tensor trunc-ell2-ket)
      also have ‹... = (∑x∈S. ket x ⊗s (A x *V g′ x))›
        by (auto intro!: infsum-finite simp: that)
      finally show ‹trunc-ell2 (S × UNIV) (C g) = (∑x∈S. ket x ⊗s A x *V g′ x)›
        by −
    qed
    also have ‹... = (∑x∈S. (norm (ket x ⊗s A x *V g′ x))²)›
      apply (subst pythagorean-theorem-sum)
      by auto
    also have ‹... = (∑x∈S. (norm (A x *V g′ x))²)›
      by (simp add: norm-tensor-ell2)
    also have ‹... ≤ (∑x∈S. (B * norm (g′ x))²)›
      using assms
      by (auto intro!: sum-mono power-mono norm-cblinfun[THEN order-trans] mult-right-mono)
    also have ‹... = (∑x∈S. (norm (g′ x))²) * B²›
      by (simp add: power-mult-distrib mult.commute sum-distrib-left)
    also have ‹... = (∑x∈S. (norm (ket x ⊗s g′ x))²) * B²›
      by (simp add: norm-tensor-ell2)
    also have ‹... = (norm (∑x∈S. ket x ⊗s g′ x))² * B²›
      apply (subst pythagorean-theorem-sum[symmetric])
      by (auto simp add: g′-def)
    also have ‹... ≤ (norm g)² * B²›
    proof −
      have ‹(∑x∈S. ket x ⊗s g′ x) = trunc-ell2 (S×UNIV) g›
        unfolding g′-def
        apply (rule fun-cong[where x=g])
        apply (rule bounded-clinear-equal-ket)
          apply (auto intro!: bounded-linear-intros)[2]
        by (auto intro!: simp: cinner-ket trunc-ell2-prod-tensor trunc-ell2-ket
            tensor-ell2-scaleC2 sum-of-bool-scaleC
            simp flip: tensor-ell2-ket
            split!: if-split-asm)
      then show ?thesis
       by (auto intro!: trunc-ell2-reduces-norm mult-right-mono power-mono sum-nonneg norm-ge-zero
zero-le-power2)
    qed
    also have ‹... ≤ (norm g * B)²›
      by (simp add: power-mult-distrib)
    finally show ?thesis
    by (metis Extra-Ordered-Fields.sign-simps(5) ‹0 ≤ B› norm-ge-zero power2-le-imp-le zero-compare-simps(4))
  qed
  have ‹((λF. trunc-ell2 F (C g)) ⟶ C g) (finite-subsets-at-top UNIV)›
    by (rule trunc-ell2-lim-at-UNIV)
  then have ‹((λF. norm (trunc-ell2 F (C g))) ⟶ norm (C g)) (finite-subsets-at-top UNIV)›
    by (rule tendsto-norm)
  then show ‹norm (C g) ≤ B * norm g›
    apply (rule tendsto-upperbound)
    using norm-trunc by (auto intro!: eventually-finite-subsets-at-top-weakI simp: )
```

qed

have ‹*bounded-clinear C*›
proof (*intro bounded-clinearI allI*)
fix *g h* :: ‹(*'x* × *'a*) *ell2*› **and** *c* :: *complex*
from *C-hassum*[*of g*] *C-hassum*[*of h*]
have ‹((λ*x*. *A' x* $*_V$ *g* + *A' x* $*_V$ *h*) *has-sum C g* + *C h*) *UNIV*›
by (*simp add*: *has-sum-add*)
with *C-hassum*[*of* ‹*g* + *h*›]
show ‹*C* (*g* + *h*) = *C g* + *C h*›
by (*metis* (*no-types*, *lifting*) *cblinfun.add-right has-sum-cong infsumI*)
from *C-hassum*[*of g*]
have ‹((λ*x*. *c* $*_C$ (*A' x* $*_V$ *g*)) *has-sum c* $*_C$ *C g*) *UNIV*›
by (*metis cblinfun-scaleC-right.rep-eq has-sum-cblinfun-apply*)
with *C-hassum*[*of* ‹*c* $*_C$ *g*›]
show ‹*C* (*c* $*_C$ *g*) = *c* $*_C$ *C g*›
by (*metis* (*no-types*, *lifting*) *cblinfun.scaleC-right has-sum-cong infsumI*)
from *norm-C* **show** ‹*norm* (*C g*) ≤ *norm g* ∗ *B*›
by (*simp add*: *sign-simps*(*5*))
qed
define *D* **where** ‹*D* = *CBlinfun C*›
with ‹*bounded-clinear C*› **have** *DC*: ‹*D* $*_V$ *f* = *C f*› **for** *f*
by (*simp add*: *bounded-clinear-CBlinfun-apply*)
have *D-hassum*: ‹*has-sum-in cstrong-operator-topology A' UNIV D*›
using *C-hassum* **by** (*simp add*: *has-sum-in-cstrong-operator-topology DC*)
then show ‹*has-sum-in cstrong-operator-topology A' UNIV* (*controlled-op A*)›
using *controlled-op-def A'-def*
by (*metis* (*no-types*, *lifting*) *has-sum-in-infsum-in hausdorff-sot infsum-in-cong summable-on-in-def*)
with *D-hassum* **have** *DA*: ‹*D* = *controlled-op A*›
apply (*rule-tac has-sum-in-unique*)
by *auto*
show ‹*norm* (*controlled-op A*) ≤ *B*›
apply (*rule norm-cblinfun-bound*, *simp*)
using *norm-C* **by** (*simp add*: *DC flip*: *DA*)
qed

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*)›
proof −
from *assms* **obtain** *B* **where** ‹*norm* (*A x*) ≤ *B*› **for** *x*
by (*auto intro*!: *simp*: *bdd-above-def*)
then show *?thesis*
by (*rule controlled-op-has-sum-aux*)
qed

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*›
using *controlled-op-has-sum*[*OF assms*] *summable-on-in-def* **by** *blast*

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)›
  by (*metis assms has-sum-in-cstrong-operator-topology has-sum-in-infsum-in hausdorff-sot infsumI*)

lemma *controlled-op-ket*[*simp*]:
  assumes ‹*bdd-above* (*range* (λx. *norm* (*A x*)))›
  shows ‹*controlled-op A* $*_V$ (*ket x* $\otimes_s$ ψ) = *ket x* $\otimes_s$ (*A x* $*_V$ ψ)›
proof −
  have ‹*controlled-op A* $*_V$ (*ket x* $\otimes_s$ ψ) = ($\sum_\infty y$. (*selfbutter* (*ket y*) $\otimes_o$ *A y*) $*_V$ (*ket x* $\otimes_s$ ψ))›
    by (*simp add*: *controlled-op-def assms infsum-sot-cblinfun-apply controlled-op-summable*)
  also have ‹. . . = *ket x* $\otimes_s$ (*A x* $*_V$ ψ)›
    apply (*subst infsum-single*[**where** *i*=*x*])
    by (*simp-all add*: *tensor-op-ell2 cinner-ket*)
  finally show *?thesis*
    by −
qed

lemma *controlled-op-ket'*[*simp*]:
  assumes ‹*bdd-above* (*range* (λx. *norm* (*A x*)))›
  shows ‹*controlled-op A* $*_V$ (*ket* (*x*, *y*)) = *ket x* $\otimes_s$ (*A x* $*_V$ *ket y*)›
  by (*metis assms controlled-op-ket tensor-ell2-ket*)

lemma *controlled-op-compose*[*simp*]:
  assumes [*simp*]: ‹*bdd-above* (*range* (λx. *norm* (*A x*)))›
  assumes [*simp*]: ‹*bdd-above* (*range* (λx. *norm* (*B x*)))›
  shows ‹*controlled-op A* o_{CL} *controlled-op B* = *controlled-op* (λx. *A x* o_{CL} *B x*)›
proof −
  from *assms*(*1*) **obtain** *a* **where** ‹*norm* (*A x*) ≤ *a*› **for** *x*
    by (*auto simp*: *bdd-above-def*)
  moreover from *assms*(*2*) **obtain** *b* **where** ‹*norm* (*B x*) ≤ *b*› **for** *x*
    by (*auto simp*: *bdd-above-def*)
  ultimately have [*simp*]: ‹*bdd-above* (*range* (λx. *norm* (*A x* o_{CL} *B x*)))›
    apply (*rule-tac bdd-aboveI*[*of* - ‹*a*∗*b*›])
    by (*smt* (*verit*, *ccfv-SIG*) *Multiseries-Expansion-Bounds.mult-monos*(*1*) *imageE norm-cblinfun-compose norm-ge-zero*)
  show *?thesis*
    apply (*rule equal-ket*)
    apply (*case-tac x*)
    by *simp*
qed

lemma *controlled-op-adj*[*simp*]:
  assumes [*simp*]: ‹*bdd-above* (*range* (λx. *norm* (*A x*)))›
  shows ‹(*controlled-op A*)∗ = *controlled-op* (λx. (*A x*)∗)›
  apply (*rule cinner-ket-adjointI*[*symmetric*])
  by (*auto intro*!: *simp*: *controlled-op-ket cinner-adj-left*
    simp flip: *tensor-ell2-ket*)

lemma *controlled-op-id*[*simp*]: ‹*controlled-op* (λ-. *id-cblinfun*) = *id-cblinfun*›
  apply (*rule equal-ket*)
  apply (*case-tac x*)
  by (*simp add*: *tensor-ell2-ket*)

lemma *controlled-op-unitary*[*simp*]: ‹*unitary* (*controlled-op U*)› **if** [*simp*]: ‹⋀*x*. *unitary* (*U x*)›
proof −
  have [*iff*]: ‹*bdd-above* (*range* (λ*x*. *norm* (*U x*)))›
    by (*simp add*: *norm-isometry*)
  show *?thesis*
    unfolding *unitary-def* **by** *auto*
qed

lemma *controlled-op-is-Proj*[*simp*]: ‹*is-Proj* (*controlled-op P*)› **if** [*simp*]: ‹⋀*x*. *is-Proj* (*P x*)›
proof −
  have [*iff*]: ‹*bdd-above* (*range* (λ*x*. *norm* (*P x*)))›
    using *norm-is-Proj*[*OF that*]
    by (*auto intro*!: *bdd-aboveI simp*:)
  show *?thesis*
    using *that* **unfolding** *is-Proj-algebraic* **by** *auto*
qed

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*))›
  using *assms* **by** (*auto intro*!: *equal-ket simp*: *Fst-def tensor-op-ket cinner-ket*)

lemma *norm-ell2-finite*: ‹*norm* ψ = *sqrt* ($\sum$ *i*∈*UNIV*. (*cmod* (*Rep-ell2* ψ *i*))^2)› **for** ψ :: ‹-::*finite ell2*›
  apply *transfer*
  by (*simp add*: *ell2-norm-finite*)

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*›
  by (*simp add*: *assms swap-sandwich-swap-ell2 sandwich-apply*)

lemma *controlled-op-const*: ‹*controlled-op* (λ-. *P*) = *Snd P*›
  by (*auto intro*!: *equal-ket simp*: *Snd-def tensor-op-ell2 simp flip*: *tensor-ell2-ket*)

1.3 Superpositions

lift-definition *uniform-superpos* :: ‹*'a set* ⇒ *'a ell2*› **is** ‹λ*A x*. *complex-of-real* (*of-bool* (*x* ∈ *A*) / *sqrt* (*of-nat* (*card A*)))›
proof (*rename-tac A*)
  fix *A* :: ‹*'a set*›
  show ‹*has-ell2-norm* (λ*x*. *complex-of-real* (*of-bool* (*x* ∈ *A*) / *sqrt* (*real* (*card A*))))›
  proof (*cases* ‹*finite A*›)
    case *True*
    show *?thesis*
      unfolding *has-ell2-norm-def*
      apply (*rule finite-nonzero-values-imp-summable-on*)
      using *True* **by** *auto*
  next
    case *False*
    then show *?thesis*
      by *simp*
  qed
qed

lemma *norm-uniform-superpos*: ‹*norm* (*uniform-superpos A*) = *1*› **if** ‹*finite A*› **and** ‹$A \neq \{\}$›
proof (*transfer′ fixing*: *A*)
  have ‹*ell2-norm* ($\lambda x.$ *complex-of-real* (*of-bool* ($x \in A$) / *sqrt* (*real* (*card A*))))
      = *sqrt* ($\sum_\infty x.$ (*cmod* (*complex-of-real* (*of-bool* ($x \in A$) / *sqrt* (*real* (*card A*)))))^2)›
    by (*simp add*: *ell2-norm-def*)
  also have ‹... = *sqrt* ($\sum_\infty x{\in}A.$ (*cmod* (*complex-of-real* (*1* / *sqrt* (*real* (*card A*)))))^2)›
    apply (*rule arg-cong*[**where** *f*=*sqrt*])
    apply (*rule infsum-cong-neutral*)
    by *auto*
  also have ‹... = *sqrt* ($\sum x{\in}A.$ (*cmod* (*complex-of-real* (*1* / *sqrt* (*real* (*card A*)))))^2)›
    by *simp*
  also have ‹... = *sqrt* (*real* (*card A*) ∗ (*cmod* (*1* / *complex-of-real* (*sqrt* (*real* (*card A*)))))^2)›
    by (*simp add*: *that*)
  also have ‹... = *sqrt* (*real* (*card A*) ∗ ((*1* / *sqrt* (*real* (*card A*)))^2))›
    by (*simp add*: *cmod-def*)
  also have ‹... = *1*›
    using *that*
    by (*simp add*: *power-one-over*)
  finally show ‹*ell2-norm* ($\lambda x.$ *complex-of-real* (*of-bool* ($x \in A$) / *sqrt* (*real* (*card A*)))) = *1*›
    by −
qed

lemma *uniform-superpos-infinite*: ‹*uniform-superpos A* = *0*› **if** ‹*infinite A*›
  apply (*transfer′ fixing*: *A*)
  using *that*
  by *simp*

lemma *uniform-superpos-empty*: ‹*uniform-superpos A* = *0*› **if** ‹$A = \{\}$›
  apply (*transfer′ fixing*: *A*)
  using *that*
  by *simp*

Alternative definition.

lemma *uniform-superpos-def2*: ‹*uniform-superpos A* = ($\sum f{\in}A.$ *ket f* $/_C$ *csqrt* (*card A*))›
proof −
  wlog [*simp*]: ‹*finite A*› ‹$A \neq \{\}$›
    using *negation uniform-superpos-empty uniform-superpos-infinite* **by** *force*
  show *?thesis*
  proof (*rule cinner-ket-eqI*)
    fix *f*
    show ‹*ket f* $\bullet_C$ (*uniform-superpos A*) = *ket f* $\bullet_C$ ($\sum f{\in}A.$ *ket f* $/_C$ *csqrt* (*card A*))›
    proof (*cases* ‹$f \in A$›)
      case *True*
      then have ‹*ket f* $\bullet_C$ (*uniform-superpos A*) = *1* / *csqrt* (*card A*)›
        apply (*subst cinner-ket-left*)
        apply (*transfer fixing*: *f*)
        by *auto*
      moreover have ‹*ket f* $\bullet_C$ ($\sum f{\in}A.$ *ket f* $/_C$ *csqrt* (*card A*)) = *1* / *csqrt* (*card A*)›
        apply (*subst cinner-sum-right*)
        apply (*subst sum-single*[**where** *i*=*f*])
        using *True* **by** (*auto simp*: *inverse-eq-divide*)
      finally show *?thesis*
        by *simp*
    next
      case *False*

then have ‹*ket f* $\bullet_C$ (*uniform-superpos A*) = *0*›
apply (*subst cinner-ket-left*)
apply (*transfer fixing*: *f*)
by *auto*
moreover have ‹*ket f* $\bullet_C$ ($\sum$*f*∈*A*. *ket f* $/_C$ *csqrt* (*card A*)) = *0*›
apply (*subst cinner-sum-right*)
apply (*rule sum.neutral*)
using *False* **by** *auto*
finally show *?thesis*
by *simp*
qed
qed
qed

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*›
proof −
fix ψ :: ‹*′a* ⇒ *complex*›
assume ‹*has-ell2-norm* ψ›
then have ‹(λ*x*. *norm* $((\psi\ x)^2)$) *summable-on UNIV*›
by (*simp add*: *has-ell2-norm-def*)
then have ‹(λ*x*. *case x of Some x′* ⇒ *norm* $((\psi\ x')^2)$ | *None* ⇒ *0*) *o Some summable-on UNIV*›
by (*metis comp-eq-dest-lhs option.simps*(*5*) *summable-on-cong*)
then have ‹(λ*x*. *case x of Some x′* ⇒ *norm* $((\psi\ x')^2)$ | *None* ⇒ *0*) *summable-on Some* ‘ *UNIV*›
by (*meson inj-Some summable-on-reindex*)
then have ‹(λ*x*. *case x of Some x′* ⇒ *norm* $((\psi\ x')^2)$ | *None* ⇒ *0*) *summable-on UNIV*›
apply (*rule summable-on-cong-neutral*[*THEN iffD1*, *rotated* −*1*])
by (*auto simp add*: *notin-range-Some*)
then show ‹*has-ell2-norm* (*case-option 0* ψ)›
apply (*simp add*: *has-ell2-norm-def*)
by (*metis* (*mono-tags*, *lifting*) *norm-zero option.case-eq-if summable-on-cong zero-power2*)
qed

lemma *clinear-lift-ell2′*: ‹*clinear lift-ell2′*›
apply (*rule clinearI*; *transfer*)
by (*auto intro*!: *ext simp add*: *option.case-eq-if*)

lemma *lift-ell2′-norm*[*simp*]: ‹*norm* (*lift-ell2′* ψ) = *norm* ψ›
proof *transfer*
fix ψ :: ‹*′a* ⇒ *complex*›
have ‹$(\textit{ell2-norm}\ \psi)^2$ = *infsum* (λ*x*. $(\textit{norm}\ (\psi\ x))^2$) *UNIV*›
apply (*simp add*: *ell2-norm-def*)
by (*meson infsum-nonneg zero-le-power2*)
also have ‹... = *infsum* ((λ*x*. *case x of Some x′* ⇒ $(\textit{norm}\ (\psi\ x'))^2$ | *None* ⇒ *0*) *o Some*) *UNIV*›
apply (*rule infsum-cong*) **by** *auto*
also have ‹... = *infsum* (λ*x*. *case x of Some x′* ⇒ $(\textit{norm}\ (\psi\ x'))^2$ | *None* ⇒ *0*) (*Some* ‘ *UNIV*)›
by (*simp add*: *infsum-reindex*)
also have ‹... = *infsum* (λ*x*. *case x of Some x′* ⇒ $(\textit{norm}\ (\psi\ x'))^2$ | *None* ⇒ *0*) *UNIV*›
apply (*rule infsum-cong-neutral*)
by (*auto simp add*: *notin-range-Some*)
also have ‹... = $(\textit{ell2-norm}\ (\textit{case-option}\ 0\ \psi))^2$›
apply (*simp add*: *ell2-norm-def*)
by (*smt* (*verit*, *ccfv-SIG*) *infsum-nonneg infsum-cong norm-zero option.case-eq-if real-sqrt-pow2-iff zero-le-power2 zero-power2*)
finally show ‹*ell2-norm* (*case-option 0* ψ) = *ell2-norm* ψ›

  by (*simp add*: *ell2-norm-geq0*)
qed

lemma *bounded-clinear-lift-ell2'*[*bounded-clinear*, *simp*]: ‹*bounded-clinear lift-ell2'*›
 by (*metis bounded-clinear.intro bounded-clinear-axioms.intro clinear-lift-ell2' lift-ell2'-norm mult.commute mult-1 order.refl*)

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

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*)›
 unfolding *lift-ell2.rep-eq* **apply** *transfer*
 by (*auto intro*!: *ext simp*: *of-bool-def split*!: *option.split if-split-asm*)

lemma *isometry-lift-ell2*[*simp*]: ‹*isometry lift-ell2*›
 apply (*rule norm-preserving-isometry*)
 by (*simp add*: *lift-ell2.rep-eq*)

lemma *lift-op-adj*: ‹(*lift-op A*)∗ = *lift-op* (*A*∗)›
 unfolding *lift-op-def*
 apply (*simp add*: *adj-plus*)
 by (*simp add*: *cblinfun-assoc-left*(*1*))

lemma *bra-None-lift-ell2*: ‹*bra None* o_{CL} *lift-ell2* = *0*›
 apply (*rule cblinfun-eqI*)
 apply (*simp add*: *lift-ell2.rep-eq*)
 apply *transfer*
 by (*simp add*: *infsum-0*)

lemma *lift-op-mult*: ‹*lift-op A* o_{CL} *lift-op B* = *lift-op* (*A* o_{CL} *B*)›
proof −
 have ‹*lift-op A* o_{CL} *lift-op B* =
   (*lift-ell2* o_{CL} *A* o_{CL} (*lift-ell2*∗ o_{CL} *lift-ell2*) o_{CL} *B* o_{CL} *lift-ell2*∗)
  + (*lift-ell2* o_{CL} *A* o_{CL} (*bra None* o_{CL} *lift-ell2*)∗ o_{CL} *bra None*)
  + (*vector-to-cblinfun* (*ket None*) o_{CL} (*bra None* o_{CL} *lift-ell2*) o_{CL} *B* o_{CL} *lift-ell2*∗)
  + *butterfly* (*ket None*) (*ket None*)›
  unfolding *lift-op-def*
  apply (*simp add*: *adj-plus cblinfun-compose-add-right cblinfun-compose-add-left del*: *isometryD*)
  apply (*simp add*: *butterfly-def cblinfun-compose-assoc del*: *isometryD*)
  by (*metis butterfly-def cblinfun-comp-butterfly*)
 also have ‹. . . = (*lift-ell2* o_{CL} (*A* o_{CL} *B*) o_{CL} *lift-ell2*∗) + *butterfly* (*ket None*) (*ket None*)›
  by (*simp add*: *bra-None-lift-ell2 cblinfun-compose-assoc*)
 also have ‹. . . = *lift-op* (*A* o_{CL} *B*)›
  by (*simp add*: *lift-op-def*)
 finally show *?thesis*
  by −
qed

lemma *lift-ell2-adj-None*[*simp*]: ‹*lift-ell2*∗ $*_V$ *ket None* = *0*›
 by (*simp add*: *cinner-adj-right cinner-ket-eqI lift-ell2-ket*)

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

by (*simp add*: *cinner-adj-right cinner-ket cinner-ket-eqI lift-ell2-ket*)

lemma *lift-op-id*[*simp*]: ‹*lift-op id-cblinfun* = *id-cblinfun*›
apply (*rule equal-ket*, *case-tac x*)
by (*auto simp*: *lift-op-def cblinfun.add-left cblinfun-compose-add-right lift-ell2-adj-None lift-ell2-ket*)

lemma *isometry-lift-op*[*simp*]: ‹*isometry* (*lift-op A*)› **if** ‹*isometry A*›
by (*simp add*: *isometry-def lift-op-mult lift-op-adj isometryD*[*OF that*])

lemma *unitary-lift-op*[*simp*]: ‹*unitary* (*lift-op A*)› **if** ‹*unitary A*›
by (*metis isometry-lift-op lift-op-adj that unitary-twosided-isometry*)

lemma *lift-op-None*[*simp*]: ‹*lift-op A* $*_V$ *ket None* = *ket None*›
unfolding *lift-op-def* **by** (*auto simp*: *cblinfun.add-left*)

lemma *lift-op-Some*[*simp*]: ‹*lift-op A* $*_V$ *ket* (*Some x*) = *lift-ell2* $*_V$ *A* $*_V$ *ket x*›
unfolding *lift-op-def* **by** (*auto simp*: *cblinfun.add-left*)

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

lemma *sum-sqrt*: ‹($\sum$ *i*<*n*. *sqrt i*) ≤ *2/3* ∗ (*sqrt n*)^*3*› **for** *n* :: *nat*
proof (*induction n*)
  case *0*
  show *?case*
    by *simp*
next
  case (*Suc n*)
  have ‹($\sum$ *i*<*Suc n*. *sqrt i*) ≤ *2/3* ∗ *sqrt* (*real n*) ^ *3* + *sqrt n*›
    using *Suc*
    by *simp*
  also have ‹. . . ≤ *2/3* ∗ *sqrt* (*Suc n*) ^ *3*›
  proof −
    define *x* :: *real* **where** ‹*x* = *n*›
    define *f* **where** ‹*f z* = *2/3* ∗ (*sqrt z*)^*3*› **for** *z*
    have *f'*: ‹(*f has-real-derivative sqrt z*) (*at z*)› **if** ‹*z* > *0*› **for** *z*
      apply (*rule ssubst*[*of* ‹*sqrt z*›, *rotated*])
      unfolding *f-def*
       apply (*rule that DERIV-real-sqrt derivative-eq-intros refl*)+
      using *that*
      apply *simp*
     by (*smt* (*verit*, *del-insts*) *Extra-Ordered-Fields.sign-simps*(*5*) *nonzero-eq-divide-eq sqrt-divide-self-eq*)
    have *cont*: ‹*continuous-on* {*x*..*x*+*1*} *f*›
      unfolding *f-def*
      by (*intro continuous-intros*)
    have ‹*x* ≥ *0*›
      using *x-def* **by** *auto*
    obtain *l z* **where** ‹*x* < *z*› ‹*z* < *x* + *1*› **and** *f'l*: ‹(*f has-real-derivative l*) (*at z*)› **and** *fdelta*: ‹*f* (*x* + *1*) − *f x* = (*x* + *1* − *x*) ∗ *l*›
      apply *atomize-elim*
      apply (*subst ex-comm*)
      apply (*rule MVT*)
        apply *simp*
       apply (*rule cont*)
      using *f'*

by (*smt* (*verit*, *best*) ‹*0* ≤ *x*› *real-differentiable-def*)
then have ‹*z* > *0*›
using ‹*0* ≤ *x*› **by** *linarith*
from *f'*[*OF this*] *f'l* **have** [*simp*]: ‹*l* = *sqrt z*›
using *DERIV-unique* **by** *blast*
from *fdelta*
have ‹*f* (*x* + *1*) − *f x* ≥ *sqrt x*›
using ‹*x* < *z*› **by** *auto*
then show *?thesis*
unfolding *x-def f-def*
by (*smt* (*verit*, *best*) *Num.of-nat-simps*(*3*))
qed
finally show *?case*
by −
qed

lemma *register-inj'*:
assumes ‹*register Q*›
shows ‹*Q a* = *Q b* ⟷ *a* = *b*›
using *register-inj*[*OF assms*] **by** *blast*

lemma *norm-cblinfun-apply-leq1I*:
assumes ‹*norm U* ≤ *1*›
assumes ‹*norm* ψ ≤ *1*›
shows ‹*norm* (*U* $*_V$ ψ) ≤ *1*›
by (*smt* (*verit*, *best*) *assms*(*1*,*2*) *mult-left-le-one-le norm-cblinfun norm-ge-zero*)

lemma *times-sqrtn-div-n*[*simp*]:
assumes ‹*n* ≥ *0*›
shows ‹*a* ∗ *sqrt n* / *n* = *a* / *sqrt n*›
by (*metis assms divide-divide-eq-right real-div-sqrt*)

lemma *Proj-tensor-Proj*: ‹*Proj I* $\otimes_o$ *Proj J* = *Proj* (*I* $\otimes_S$ *J*)›
by (*simp add*: *Proj-on-own-range is-Proj-tensor-op*
tensor-ccsubspace-via-Proj)

lemma *extend-mult-rule*: ‹*a* ∗ *b* = *c* ⟹ *a* ∗ (*b* ∗ *d*) = *c* ∗ *d*› **for** *a b c d* :: ‹-::*semigroup-mult*›
by (*metis Groups.mult-ac*(*1*))

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

apply *standard* **apply** (*rule finite-imageD*[**where** *f*=*Rep-punctured-function*])
by (*auto simp add*: *Rep-punctured-function-inject inj-on-def*)

lift-definition *fix-punctured-function* :: ‹'*a* ⇒ ('*b* × ('*a*,'*b*) *punctured-function*) ⇒ ('*a*⇒'*b*)› **is**
‹λ*x* (*y*, *f*). (*Fun.swap x undefined f*) (*x* := *y*)›.

lift-definition *puncture-function* :: ‹'*a* ⇒ ('*a*⇒'*b*) ⇒ '*b* × ('*a*,'*b*) *punctured-function*› **is**
‹λ*x f*. (*f x*, (*Fun.swap x undefined f*) (*undefined* := *undefined*))›
by (*simp add*: *Compl-eq-Diff-UNIV*)

lemma *puncture-function-recombine*:
‹(*y*, *snd* (*puncture-function x f*)) = *puncture-function x* (*f*(*x*:=*y*))›
apply *transfer*
by (*auto intro*!: *ext simp*: *Transposition.transpose-def*)

lemma *snd-puncture-function-upd*: ‹*snd* (*puncture-function x* (*f*(*x*:=*y*))) = *snd* (*puncture-function x f*)›
apply *transfer*
by (*auto intro*!: *ext simp*: *Transposition.transpose-def*)

lemma *puncture-function-split*: ‹*puncture-function x f* = (*f x*, *snd* (*puncture-function x f*))›
using *puncture-function-recombine*[**where** *x*=*x* **and** *f*=*f* **and** *y*=‹*f x*›]
by *simp*

lemma *puncture-function-inverse*[*simp*]: ‹*fix-punctured-function x* (*puncture-function x f*) = *f*›
apply *transfer* **by** (*auto intro*!: *ext simp*: *Transposition.transpose-def*)

lemma *fix-punctured-function-inverse*[*simp*]: ‹*puncture-function x* (*fix-punctured-function x yf*) = *yf*›
apply *transfer*
by (*auto intro*!: *ext simp*: *Transposition.transpose-def extensional-def*)

lemma *bij-fix-punctured-function*[*simp*]: ‹*bij* (*fix-punctured-function x*)›
by (*metis bijI' fix-punctured-function-inverse puncture-function-inverse*)

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

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

The following *function-at-U x* provides an unitary isomorphism between ('*a* ⇒ '*b*) *ell2* (superposition of functions) and ('*b* × ('*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* ⇒ ('*b* × ('*a*, '*b*) *punctured-function*) *ell2* $\Rightarrow_{CL}$ ('*a* ⇒ '*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*)›
by (*auto simp*: *function-at-U-def intro*!: *unitary-classical-operator*)

lemma *function-at-U-ket*[*simp*]: ‹*function-at-U x* $*_V$ *ket y* = *ket* (*fix-punctured-function x y*)›
by (*simp add*: *function-at-U-def classical-operator-ket classical-operator-exists-inj*)

lemma *function-at-U-adj-ket*[*simp*]: ‹(*function-at-U x*)* $*_V$ *ket y* = *ket* (*puncture-function x y*)›
apply (*simp add*: *function-at-U-def inv-map-total classical-operator-ket classical-operator-exists-inj*)

by (*metis* (*no-types*, *lifting*) *bij-betw-inv-into bij-def bij-fix-punctured-function classical-operator-exists-inj classical-operator-ket inj-map-total inv-f-f o-def option.case*(*2*) *puncture-function-inverse*)

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*)›
proof −
have ‹*Rep-ell2* (*function-at x U* $*_V$ *ket f*) *g* = *Rep-ell2* (*function-at-U x* $*_V$ (*U* $\otimes_o$ *id-cblinfun*) $*_V$ *ket* (*puncture-function x f*)) *g*›
by (*simp add*: *function-at-def Fst-def sandwich-apply*)
also have ‹... = (*function-at-U x*$*$ $*_V$ *ket g*) $\bullet_C$ ((*U* $\otimes_o$ *id-cblinfun*) $*_V$ *ket* (*puncture-function x f*))›
by (*metis cinner-adj-left cinner-ket-left*)
also have ‹... = (*ket* (*puncture-function x g*)) $\bullet_C$ ((*U* $\otimes_o$ *id-cblinfun*) $*_V$ *ket* (*puncture-function x f*))›
by (*simp add*: *function-at-def*)
also have ‹... = (*ket* (*g x*, *snd* (*puncture-function x g*))) $\bullet_C$ ((*U* $\otimes_o$ *id-cblinfun*) $*_V$ *ket* (*f x*, *snd* (*puncture-function x f*)))›
by (*simp flip*: *puncture-function-split*)
also have ‹... = *of-bool* (*snd* (*puncture-function x f*) = *snd* (*puncture-function x g*)) $*$ (*ket* (*g x*) $\bullet_C$ (*U* $*_V$ *ket* (*f x*)))›
by (*simp add*: *tensor-op-ell2 cinner-ket flip*: *tensor-ell2-ket*)
also have ‹... = *of-bool* (*snd* (*puncture-function x f*) = *snd* (*puncture-function x g*)) $*$ *Rep-ell2* (*U* $*_V$ *ket* (*f x*)) (*g x*)›
by (*simp add*: *cinner-ket-left*)
finally show *?thesis*
by −
qed

lemma *function-at-ket*:
shows ‹*function-at x U* $*_V$ *ket f* = ($\sum_\infty y \in UNIV$. *Rep-ell2* (*U* $*_V$ *ket* (*f x*)) *y* $*_C$ *ket* (*f* (*x* := *y*)))›
proof −
have ‹*function-at x U* $*_V$ *ket f* = *function-at-U x* $*_V$ (*U* $\otimes_o$ *id-cblinfun*) $*_V$ *ket* (*puncture-function x f*)›
by (*simp add*: *function-at-def Fst-def sandwich-apply*)
also have ‹... = *function-at-U x* $*_V$ (*U* $\otimes_o$ *id-cblinfun*) $*_V$ *ket* (*f x*, *snd* (*puncture-function x f*))›
by (*metis puncture-function-split*)
also have ‹... = *function-at-U x* $*_V$ ((*U* $*_V$ *ket* (*f x*)) $\otimes_s$ *ket* (*snd* (*puncture-function x f*)))›
by (*simp add*: *tensor-op-ket*)
also have ‹... = *function-at-U x* $*_V$ (($\sum_\infty y \in UNIV$. *Rep-ell2* (*U* $*_V$ *ket* (*f x*)) *y* $*_C$ *ket y*) $\otimes_s$ *ket* (*snd* (*puncture-function x f*)))›
by (*simp flip*: *ell2-decompose-infsum*)
also have ‹... = ($\sum_\infty y \in UNIV$. *Rep-ell2* (*U* $*_V$ *ket* (*f x*)) *y* $*_C$ (*function-at-U x* $*_V$ (*ket y* $\otimes_s$ *ket* (*snd* (*puncture-function x f*)))))›
by (*simp del*: *function-at-U-ket*
add: *tensor-ell2-scaleC1 invertible-cblinfun-isometry infsum-cblinfun-apply-invertible infsum-tensor-ell2-left*
flip: *cblinfun.scaleC-right*)
also have ‹... = ($\sum_\infty y \in UNIV$. *Rep-ell2* (*U* $*_V$ *ket* (*f x*)) *y* $*_C$ *ket* (*f* (*x* := *y*)))›

by (*simp add*: *puncture-function-recombine tensor-ell2-ket*)
finally show *?thesis*
by −
qed

lemma *register-function-at*[*simp*, *register*]: ‹*register* (*function-at x* :: ′*b update* ⇒ (′*a*⇒′*b*) *update*)› **for** *x* :: ′*a*
by (*auto simp add*: *function-at-def unitary-sandwich-register*)

lemma *function-at-comm*:
fixes *U V* :: ‹′*b ell2* ⇒$_{CL}$ ′*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*›
proof −
define *reorder* **where** ‹*reorder* = *classical-operator* (*Some o* (λ(*f* :: ′*a* ⇒ ′*b*, *a*, *b*). (*f*(*x*:=*a*, *y*:=*b*), *f x*, *f y*)))›

have *selfinv*: ‹(λ(*f*, *a*, *b*). (*f*(*x* := *a*, *y* := *b*), *f x*, *f y*)) *o* (λ(*f*, *a*, *b*). (*f*(*x* := *a*, *y* := *b*), *f x*, *f y*)) = *id*›
using *assms* **by** (*auto intro*!: *ext*)
have *bij*: ‹*bij* (λ(*f*, *a*, *b*). (*f*(*x* := *a*, *y* := *b*), *f x*, *f y*))›
using *o-bij selfinv* **by** *blast*
have *inv*: ‹*inv* (λ(*f*, *a*, *b*). (*f*(*x* := *a*, *y* := *b*), *f x*, *f y*)) = (λ(*f*, *a*, *b*). (*f*(*x* := *a*, *y* := *b*), *f x*, *f y*))›
using *inv-unique-comp selfinv* **by** *blast*
have *inj-map*: ‹*inj-map* (*Some o* (λ(*f*, *a*, *b*). (*f*(*x* := *a*, *y* := *b*), *f x*, *f y*)))›
by (*simp add*: *inj-map-total bij-is-inj*[*OF bij*])
have *inv*: ‹*inv-map* (*Some o* (λ(*f*, *a*, *b*). (*f*(*x* := *a*, *y* := *b*), *f x*, *f y*))) = (*Some o* (λ(*f*, *a*, *b*). (*f*(*x* := *a*, *y* := *b*), *f x*, *f y*)))›
by (*simp add*: *inv-map-total bij-is-surj bij inv*)
have *reorder-exists*: ‹*classical-operator-exists* (*Some o* (λ(*f*, *a*, *b*). (*f*(*x* := *a*, *y* := *b*), *f x*, *f y*)))›
using *inj-map* **by** (*rule classical-operator-exists-inj*)

have [*simp*]: ‹*reorder*∗ = *reorder*›
by (*simp add*: *reorder-def classical-operator-adjoint*[*OF inj-map*] *inv*)
have [*simp*]: ‹*reorder* (*ket f* ⊗$_s$ *ket a* ⊗$_s$ *ket b*) = *ket* (*f*(*x*:=*a*, *y*:=*b*), *f x*, *f y*)› **for** *f a b*
by (*simp add*: *reorder-def tensor-ell2-ket classical-operator-ket*[*OF reorder-exists*])
have [*simp*]: ‹*isometry reorder*›
using *inj-map-total isometry-classical-operator inj-map reorder-def* **by** *blast*

have *sandwichU*: ‹*sandwich reorder* (*function-at x U* ⊗$_o$ *id-cblinfun*) = *id-cblinfun* ⊗$_o$ (*U* ⊗$_o$ *id-cblinfun*)›
proof (*rule equal-ket*, *rule cinner-ket-eqI*, *rename-tac fab gcd*)
fix *fab gcd* :: ‹(′*a* ⇒ ′*b*) × ′*b* × ′*b*›
obtain *f a b* **where** [*simp*]: ‹*fab* = (*f*,*a*,*b*)›
by (*auto simp*: *prod-eq-iff*)
obtain *g c d* **where** [*simp*]: ‹*gcd* = (*g*,*c*,*d*)›
by (*auto simp*: *prod-eq-iff*)
have *fg-rewrite*: ‹*f* = *g* ∧ *b* = *d* ⟷
snd (*puncture-function x* (*f*(*x* := *a*, *y* := *b*))) = *snd* (*puncture-function x* (*g*(*x* := *c*, *y* := *d*))) ∧ *f x* = *g x* ∧ *f y* = *g y*›
using *assms*
by (*smt* (*verit*, *del-insts*) *array-rules*(*3*) *fun-upd-idem fun-upd-twist puncture-function-inverse puncture-function-recombine snd-puncture-function-upd*)
have ‹*ket gcd* •$_C$ ((*sandwich reorder* ∗$_V$ *function-at x U* ⊗$_o$ *id-cblinfun*) ∗$_V$ *ket fab*)
= *ket* (*g*(*x*:=*c*, *y*:=*d*), *g x*, *g y*) •$_C$ ((*function-at x U* ⊗$_o$ *id-cblinfun*) ∗$_V$ *ket* (*f*(*x*:=*a*, *y*:=*b*), *f x*, *f y*))›
by (*simp add*: *sandwich-apply flip*: *cinner-adj-left tensor-ell2-ket*)

also have ‹... = (*ket* (*g*(*x*:=*c*, *y*:=*d*)) $\cdot_C$ (*function-at x U* $*_V$ *ket* (*f*(*x*:=*a*, *y*:=*b*))))
$*$ *of-bool* (*f x* = *g x* ∧ *f y* = *g y*)›
by (*auto simp add*: *tensor-op-ell2 simp flip*: *tensor-ell2-ket*)
also have ‹... = *Rep-ell2* (*U* $*_V$ *ket a*) *c* $*$ *of-bool* (*f* = *g* ∧ *b* = *d*)›
using *assms* **by** (*auto simp add*: *cinner-ket-left Rep-ell2-function-at-ket fg-rewrite*)
also have ‹... = *ket gcd* $\cdot_C$ ((*id-cblinfun* $\otimes_o$ *U* $\otimes_o$ *id-cblinfun*) $*_V$ *ket fab*)›
by (*auto simp add*: *tensor-op-ell2 cinner-ket-left*[*of c*] *simp flip*: *tensor-ell2-ket*)
finally show ‹*ket gcd* $\cdot_C$ ((*sandwich reorder* $*_V$ *function-at x U* $\otimes_o$ *id-cblinfun*) $*_V$ *ket fab*) =
ket gcd $\cdot_C$ ((*id-cblinfun* $\otimes_o$ *U* $\otimes_o$ *id-cblinfun*) $*_V$ *ket fab*)›
by −
qed

have *sandwichV*: ‹*sandwich reorder* (*function-at y V* $\otimes_o$ *id-cblinfun*) = *id-cblinfun* $\otimes_o$ (*id-cblinfun* $\otimes_o$ *V*)›
proof (*rule equal-ket*, *rule cinner-ket-eqI*, *rename-tac fab gcd*)
fix *fab gcd* :: ‹('*a* ⇒ '*b*) × '*b* × '*b*›
obtain *f a b* **where** [*simp*]: ‹*fab* = (*f*,*a*,*b*)›
by (*auto simp*: *prod-eq-iff*)
obtain *g c d* **where** [*simp*]: ‹*gcd* = (*g*,*c*,*d*)›
by (*auto simp*: *prod-eq-iff*)
have *fg-rewrite*: ‹*f* = *g* ∧ *a* = *c* ⟷
snd (*puncture-function y* (*f*(*x* := *a*, *y* := *b*))) = *snd* (*puncture-function y* (*g*(*x* := *c*, *y* := *d*))) ∧ *f x* = *g x* ∧ *f y* = *g y*›
using *assms*
by (*metis array-rules*(*3*) *fun-upd-idem fun-upd-twist puncture-function-inverse puncture-function-recombine snd-puncture-function-upd*)
have ‹*ket gcd* $\cdot_C$ ((*sandwich reorder* $*_V$ *function-at y V* $\otimes_o$ *id-cblinfun*) $*_V$ *ket fab*)
= *ket* (*g*(*x*:=*c*, *y*:=*d*), *g x*, *g y*) $\cdot_C$ ((*function-at y V* $\otimes_o$ *id-cblinfun*) $*_V$ *ket* (*f*(*x*:=*a*, *y*:=*b*), *f x*, *f y*))›
by (*simp add*: *sandwich-apply flip*: *cinner-adj-left tensor-ell2-ket*)
also have ‹... = (*ket* (*g*(*x*:=*c*, *y*:=*d*)) $\cdot_C$ (*function-at y V* $*_V$ *ket* (*f*(*x*:=*a*, *y*:=*b*))))
$*$ *of-bool* (*f x* = *g x* ∧ *f y* = *g y*)›
by (*auto simp add*: *tensor-op-ell2 simp flip*: *tensor-ell2-ket*)
also have ‹... = *Rep-ell2* (*V* $*_V$ *ket b*) *d* $*$ *of-bool* (*f* = *g* ∧ *a* = *c*)›
using *assms* **by** (*auto simp add*: *cinner-ket-left Rep-ell2-function-at-ket fg-rewrite*)
also have ‹... = *ket gcd* $\cdot_C$ ((*id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *V*) $*_V$ *ket fab*)›
by (*auto simp add*: *tensor-op-ell2 cinner-ket-left*[*of d*] *simp flip*: *tensor-ell2-ket*)
finally show ‹*ket gcd* $\cdot_C$ ((*sandwich reorder* $*_V$ *function-at y V* $\otimes_o$ *id-cblinfun*) $*_V$ *ket fab*) =
ket gcd $\cdot_C$ ((*id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *V*) $*_V$ *ket fab*)›
by −
qed

have ‹*sandwich reorder* ((*function-at x U* $\otimes_o$ *id-cblinfun*) o_{CL} (*function-at y V* $\otimes_o$ *id-cblinfun*))
= *sandwich reorder* ((*function-at y V* $\otimes_o$ *id-cblinfun*) o_{CL} (*function-at x U* $\otimes_o$ *id-cblinfun*))›
apply (*simp add*: *sandwichU sandwichV flip*: *sandwich-arg-compose*)
by (*simp add*: *comp-tensor-op*)
then have ‹(*function-at x U* $\otimes_o$ (*id-cblinfun* :: ('*b* × '*b*) *ell2* $\Rightarrow_{CL}$ ('*b* × '*b*) *ell2*)) o_{CL} (*function-at y V* $\otimes_o$ *id-cblinfun*) = (*function-at y V* $\otimes_o$ *id-cblinfun*) o_{CL} (*function-at x U* $\otimes_o$ *id-cblinfun*)›
by (*smt* (*verit*, *best*) ‹*isometry reorder*› *cblinfun-compose-id-left cblinfun-compose-id-right compatible-ac-rules*(*2*) *isometryD sandwich-apply*)
then have ‹(*function-at x U* o_{CL} *function-at y V*) $\otimes_o$ (*id-cblinfun* :: ('*b* × '*b*) *ell2* $\Rightarrow_{CL}$ ('*b* × '*b*) *ell2*) = (*function-at y V* o_{CL} *function-at x U*) $\otimes_o$ *id-cblinfun*›
by (*simp add*: *comp-tensor-op*)
then show ‹*function-at x U* o_{CL} *function-at y V* = *function-at y V* o_{CL} *function-at x U*›
apply (*rule injD*[*OF inj-tensor-left*, *rotated*])

by *simp*
qed

lemma *compatible-function-at*[*simp*]:
assumes ‹$x \neq y$›
shows ‹*compatible* (*function-at x*) (*function-at y*)›
proof (*rule compatibleI*)
show ‹*register* (*function-at x*)›
by *simp*
show ‹*register* (*function-at y*)›
by *simp*
fix *a b* :: ‹*'b update*›
show ‹*function-at x a* o_{CL} *function-at y b* = *function-at y b* o_{CL} *function-at x a*›
using *assms* **by** (*rule function-at-comm*)
qed

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

lemma *bij-puncture-function*[*simp*]: ‹*bij* (*puncture-function x*)›
by (*metis bij-betw-inv-into bij-fix-punctured-function inv-fix-punctured-function*)

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

2.1 *apply-every*

Analogue to classical $\lambda M\ u\ f\ x.$ *if* $x \in M$ *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*›
by (*simp add*: *apply-every-def*)

interpretation *apply-every-aux*: *comp-fun-commute* ‹(λ*x*. (o_{CL}) (*function-at x* (*U x*)))›
apply *standard*
apply (*rule ext*)
apply (*case-tac* ‹*x*=*y*›)
by (*auto simp flip*: *cblinfun-compose-assoc swap-registers-left*)

lemma *apply-every-unitary*: ‹*unitary* (*apply-every M U*)› **if** ‹*finite M*› **and** [*simp*]: ‹$\bigwedge x.\ x{\in}M \Longrightarrow$ *unitary* (*U x*)›
proof −
show *?thesis*
using *that*
proof *induction*
case *empty*

then show *?case*
by *simp*
next
case (*insert x F*)
then have ∗: ‹*apply-every* (*insert x F*) *U* = *function-at x* (*U x*) o_{CL} *apply-every F U*›
by (*simp add*: *apply-every-def*)
show *?case*
by (*simp add*: ∗ *register-unitary insert*)
qed
qed

lemma *apply-every-comm*: ‹*apply-every M U* o_{CL} *V* = *V* o_{CL} *apply-every M U*›
if ‹*finite M*› **and** ‹$\bigwedge$*x*. *x*∈*M* ⟹ *function-at x* (*U x*) o_{CL} *V* = *V* o_{CL} *function-at x* (*U x*)›
unfolding *apply-every-def* **using** *that*
proof *induction*
case *empty*
show *?case*
by *simp*
next
case (*insert x F*)
then show *?case*
apply (*simp add*: *insert cblinfun-compose-assoc*)
by (*simp flip*: *cblinfun-compose-assoc insert.prems*)
qed

lemma *apply-every-infinite*: ‹*apply-every M U* = *0*› **if** ‹*infinite M*›
using *that* **by** (*simp add*: *apply-every-def*)

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*
proof −
wlog *finiteM*: ‹*finite M*›
using *negation*
by (*simp add*: *apply-every-infinite*)
wlog *finiteN*: ‹*finite N*› **keeping** *finiteM*
using *negation*
by (*simp add*: *apply-every-infinite*)
define *f* :: ‹*'a* ⇒ (*'a* ⇒ *'b*) *update* ⇒ (*'a* ⇒ *'b*) *update*› **where** ‹*f x* = (o_{CL}) (*function-at x* (*U x*))› **for** *x*
define *fM fN* **where** ‹*fM* = *Finite-Set.fold f id-cblinfun M*› **and** ‹*fN* = *Finite-Set.fold f id-cblinfun N*›
have ‹*apply-every* (*M* ∪ *N*) *U* = *apply-every* (*N* ∪ *M*) *U*›
by (*simp add*: *Un-commute*)
also have ‹. . . = *Finite-Set.fold f* (*Finite-Set.fold f id-cblinfun N*) *M*›
unfolding *apply-every-def*
apply (*subst apply-every-aux.fold-set-union-disj*)
using *finiteM finiteN that* **by** (*auto simp add*: *f-def*[*abs-def*])
also have ‹. . . = *fM* o_{CL} *fN*›
unfolding *fM-def fN-def*[*symmetric*]
using *finiteM*
apply (*induction M*)
by (*auto simp add*: *f-def*[*abs-def*] *cblinfun-compose-assoc*)
also have ‹. . . = *apply-every M U* o_{CL} *apply-every N U*›
by (*simp add*: *apply-every-def fN-def fM-def f-def*[*abs-def*] *finiteN finiteM*)

finally show *?thesis*
  by *simp*
qed

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

lemma *apply-every-insert*: ‹*apply-every* (*insert x M*) *U* = *function-at x* (*U x*) o_{CL} *apply-every M U*›
if ‹$x \notin M$› **and** ‹*finite M*›
  using *that* **by** (*simp add*: *apply-every-def*)

lemma *apply-every-mult*: ‹*apply-every M U* o_{CL} *apply-every M V* = *apply-every M* (λx. *U x* o_{CL} *V x*)›
proof (*induction rule*:*infinite-finite-induct*)
  case (*infinite M*)
  then show *?case*
    by (*simp add*: *apply-every-infinite*)
next
  case *empty*
  show *?case*
    by *simp*
next
  case (*insert x F*)
  have ‹*apply-every* (*insert x F*) *U* o_{CL} *apply-every* (*insert x F*) *V*
      = *function-at x* (*U x*) o_{CL} (*apply-every F U* o_{CL} *function-at x* (*V x*)) o_{CL} *apply-every F V*›
    using *insert* **by** (*simp add*: *apply-every-insert cblinfun-compose-assoc*)
  also have ‹... = (*function-at x* (*U x*) o_{CL} *function-at x* (*V x*)) o_{CL} (*apply-every F U* o_{CL} *apply-every F V*)›
    apply (*subst apply-every-comm*)
      apply (*fact insert*)
    using *insert* **apply** (*metis* (*no-types*, *lifting*) *compatible-function-at swap-registers*)
    by (*simp add*: *cblinfun-compose-assoc*)
  also have ‹... = (*function-at x* (*U x* o_{CL} *V x*)) o_{CL} (*apply-every F U* o_{CL} *apply-every F V*)›
    by (*simp add*: *register-mult*)
  also have ‹... = (*function-at x* (*U x* o_{CL} *V x*)) o_{CL} (*apply-every F* (λx. *U x* o_{CL} *V x*))›
    using *insert.IH* **by** *presburger*
  also have ‹... = (*apply-every* (*insert x F*) (λx. *U x* o_{CL} *V x*))›
    using *insert.hyps* **by** (*simp add*: *apply-every-insert*)
  finally show *?case*
    by −
qed

lemma *apply-every-id*[*simp*]: ‹*apply-every M* (λ-. *id-cblinfun*) = *id-cblinfun*› **if** ‹*finite M*›
  using *that* **apply** *induction*
  by (*auto simp*: *apply-every-insert*)

lemma *apply-every-function-at-comm*:
  assumes ‹$x \notin M$›
  shows ‹*function-at x U* o_{CL} *apply-every M f* = *apply-every M f* o_{CL} *function-at x U*›
  using *assms* **apply** (*induction rule*: *infinite-finite-induct*)
    apply (*simp add*: *apply-every-infinite*)
   apply *simp*
  apply (*simp add*: *apply-every-insert function-at-comm*[**where** *x*=*x*]
      flip: *cblinfun-compose-assoc*)
  by (*simp add*: *cblinfun-compose-assoc*)

lemma *apply-every-adj*: ‹(*apply-every M f*)∗ = *apply-every M* (λ*i*. (*f i*)∗)›
  apply (*induction rule*: *infinite-finite-induct*)
    apply (*simp add*: *apply-every-infinite*)
   apply *simp*
  by (*simp add*: *apply-every-insert apply-every-function-at-comm register-adjoint*)

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* ($\bot_1$)
unbundle *no m-inv-syntax*
unbundle *lattice-syntax*

3.1 Invariants

definition ‹*preserves U I J ε* ⟷ *ε* ≥ *0* ∧ (∀ *ψ*∈*space-as-set I*. *norm* ($U *_V \psi$ − *Proj J* $*_V$ $U *_V \psi$) ≤ *ε* ∗ *norm ψ*)›
  for *U* :: ‹'*a*::*chilbert-space* $\Rightarrow_{CL}$ '*b*::*chilbert-space*›

lemma *preserves-def-closure*:
  assumes ‹*space-as-set I* = *closure I*'›
  shows ‹*preserves U I J ε* ⟷ *ε* ≥ *0* ∧ (∀ *ψ*∈*I*'. *norm* ($U *_V \psi$ − *Proj J* $*_V$ $U *_V \psi$) ≤ *ε* ∗ *norm ψ*)›
proof (*rule iffI*; (*elim conjE*)?)
  show ‹*preserves U I J ε* ⟹ *0* ≤ *ε* ∧ (∀ *ψ*∈*I*'. *norm* ($U *_V \psi$ − *Proj J* $*_V$ $U *_V \psi$) ≤ *ε* ∗ *norm ψ*)›
    by (*metis assms closure-subset in-mono preserves-def*)
  show ‹*preserves U I J ε*›
    if ‹*0* ≤ *ε*› **and** *bound*: ‹(∀ *ψ*∈*I*'. *norm* ($U *_V \psi$ − *Proj J* $*_V$ $U *_V \psi$) ≤ *ε* ∗ *norm ψ*)›
  proof (*unfold preserves-def*, *intro conjI ballI*)
    from *that* **show** ‹*ε* ≥ *0*› **by** *simp*
    fix *ψ* **assume** ‹*ψ* ∈ *space-as-set I*›
    with *assms* **have** ‹*ψ* ∈ *closure I*'›
      by *simp*
    then obtain *φ* **where** ‹*φ* ⟶ *ψ*› **and** ‹*φ n* ∈ *I*'› **for** *n*
      using *closure-sequential* **by** *blast*
    define *f* **where** ‹*f ξ* = *ε* ∗ *norm ξ* − *norm* ($U *_V \xi$ − *Proj J* $*_V$ $U *_V \xi$)› **for** *ξ*
    with ‹*φ* - ∈ *I*'› *bound* **have** *bound*': ‹*f* (*φ n*) ≥ *0*› **for** *n*
      by *simp*
    have ‹*continuous-on UNIV f*›
      unfolding *f-def*
      by (*intro continuous-intros*)
    then have ‹(λ*n*. *f* (*φ n*)) ⟶ *f ψ*›
      using ‹*φ* ⟶ *ψ*› **apply** (*rule continuous-on-tendsto-compose*[**where** *s*=*UNIV* **and** *f*=*f*])
      by *auto*
    with *bound*' **have** ‹*f ψ* ≥ *0*›
      by (*simp add*: *Lim-bounded2*)
    then show ‹*norm* ($U *_V \psi$ − *Proj J* $*_V$ $U *_V \psi$) ≤ *ε* ∗ *norm ψ*›
      by (*simp add*: *f-def*)
  qed

```
qed

lemma preservesI-closure:
  assumes ‹ε ≥ 0›
  assumes closure: ‹space-as-set I ⊆ closure I′›
  assumes ‹csubspace I′›
  assumes bound: ‹⋀ψ. ψ ∈ I′ ⟹ norm ψ = 1 ⟹ norm (U *V ψ − Proj J *V U *V ψ) ≤ ε›
  shows ‹preserves U I J ε›
proof −
  have *: ‹space-as-set (ccspan I′) = closure I′›
    by (metis assms(3) ccspan.rep-eq complex-vector.span-eq-iff)
  have ‹preserves U (ccspan I′) J ε›
  proof (unfold preserves-def-closure[OF *], intro conjI ballI)
    from assms show ‹ε ≥ 0› by simp

    fix ψ assume ψI: ‹ψ ∈ I′›
    show ‹norm (U *V ψ − Proj J *V U *V ψ) ≤ ε * norm ψ›
    proof (cases ‹ψ = 0›)
      case True
      then show ?thesis by auto
    next
      case False
      then have ‹norm ψ > 0›
        by simp
      define φ where ‹φ = ψ /C norm ψ›
      from ψI have ‹φ ∈ I′›
        by (simp add: φ-def ‹csubspace I′› complex-vector.subspace-scale)
      moreover from False have ‹norm φ = 1›
        by (simp add: φ-def norm-inverse)
      ultimately have ‹norm (U *V φ − Proj J *V U *V φ) ≤ ε›
        by (rule bound)
      then have ‹norm (U *V ψ − Proj J *V U *V ψ) / norm ψ ≤ ε›
        unfolding φ-def
        by (auto simp flip: scaleC-diff-right
            simp add: norm-inverse divide-inverse-commute cblinfun.scaleC-right)
      with ‹norm ψ > 0› show ?thesis
        by (simp add: divide-le-eq)
    qed
  qed
  then show ‹preserves U I J ε›
    by (smt (verit) * closure in-mono preserves-def)
qed


lemma preservesI:
  assumes ‹ε ≥ 0›
  assumes ‹⋀ψ. ψ ∈ space-as-set I ⟹ norm ψ = 1 ⟹ norm (U *V ψ − Proj J *V U *V ψ) ≤ ε›
  shows ‹preserves U I J ε›
  apply (rule preservesI-closure[where I′=‹space-as-set I›])
  using assms by auto

lemma preservesI′:
  assumes ‹ε ≥ 0›
  assumes ‹⋀ψ. ψ ∈ space-as-set I ⟹ norm ψ = 1 ⟹ norm (Proj (−J) *V U *V ψ) ≤ ε›
  shows ‹preserves U I J ε›
```

using ‹$\varepsilon \geq 0$› **apply** (*rule preservesI*)
apply (*frule assms*(*2*))
by (*simp-all add*: *Proj-ortho-compl cblinfun.diff-left*)

lemma *preserves-onorm*: ‹*preserves U I J* $\varepsilon \longleftrightarrow$ *norm* ((*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*) $\leq \varepsilon$›
proof (*rule iffI*)
assume *pres*: ‹*preserves U I J* ε›
show ‹*norm* ((*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*) $\leq \varepsilon$›
proof (*rule norm-cblinfun-bound*)
from *pres* **show** ‹$\varepsilon \geq 0$›
by (*simp add*: *preserves-def*)
fix ψ
define φ **where** ‹φ = *Proj I* $*_V$ ψ›
have *norm*φ: ‹*norm* $\varphi \leq$ *norm* ψ›
unfolding φ*-def* **apply** (*rule is-Proj-reduces-norm*) **by** *simp*

have ‹*norm* (((*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*) $*_V$ ψ) = *norm* (*U* $*_V$ φ − *Proj J* $*_V$ *U* $*_V$ φ)›
unfolding φ*-def* **by** (*simp add*: *cblinfun.diff-left*)
also from *pres* **have** ‹. . . $\leq \varepsilon$ ∗ *norm* φ›
by (*metis Proj-range* φ*-def cblinfun-apply-in-image preserves-def*)
also have ‹. . . $\leq \varepsilon$ ∗ *norm* ψ›
by (*simp add*: ‹$0 \leq \varepsilon$› *mult-left-mono norm*φ)
finally show ‹*norm* (((*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*) $*_V$ ψ) $\leq \varepsilon$ ∗ *norm* ψ›
by −
qed
next
assume *norm*: ‹*norm* ((*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*) $\leq \varepsilon$›
show ‹*preserves U I J* ε›
proof (*rule preservesI*)
show ‹$\varepsilon \geq 0$›
using *norm norm-ge-zero order-trans* **by** *blast*
fix ψ **assume** [*simp*]: ‹$\psi \in$ *space-as-set I*› **and** [*simp*]: ‹*norm* ψ = *1*›
have ‹*norm* (*U* $*_V$ ψ − *Proj J* $*_V$ *U* $*_V$ ψ) = *norm* ((*id-cblinfun* − *Proj J*) $*_V$ *U* $*_V$ ψ)›
by (*simp add*: *cblinfun.diff-left*)
also have ‹. . . = *norm* ((*id-cblinfun* − *Proj J*) $*_V$ *U* $*_V$ *Proj I* $*_V$ ψ)›
by (*simp add*: *Proj-fixes-image*)
also have ‹. . . = *norm* (((*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*) $*_V$ ψ)›
by *simp*
also have ‹. . . $\leq$ *norm* ((*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*) ∗ *norm* ψ›
using *norm-cblinfun* **by** *blast*
also have ‹. . . $\leq \varepsilon$›
by (*simp add*: *norm*)
finally show ‹*norm* (*U* $*_V$ ψ − *Proj J* $*_V$ *U* $*_V$ ψ) $\leq \varepsilon$›
by −
qed
qed

lemma *preserves-cong*:
assumes ‹$\bigwedge\psi$. $\psi \in$ *space-as-set I* $\Longrightarrow$ *U* $*_V$ ψ = *U*′ $*_V$ ψ›
shows ‹*preserves U I J* $\varepsilon \longleftrightarrow$ *preserves U*′ *I J* ε›
by (*simp add*: *assms preserves-def*)

lemma *preserves-mono*:

assumes ‹*preserves U I J ε*›
assumes ‹$I \geq I'$›
assumes ‹$J \leq J'$›
assumes ‹$\varepsilon \leq \varepsilon'$›
shows ‹*preserves U I' J' ε'*›
proof (*rule preservesI*)
show ‹$\varepsilon' \geq 0$›
by (*smt* (*verit*) *assms*(*1*) *assms*(*4*) *preserves-def*)
fix ψ **assume** ‹$\psi \in$ *space-as-set* I'›
then have ‹$\psi \in$ *space-as-set* I›
using ‹$I \geq I'$› *less-eq-ccsubspace.rep-eq* **by** *blast*
assume [*simp*]: ‹*norm* $\psi = 1$›

have ‹*norm* $(U *_V \psi - $ *Proj* $J' *_V U *_V \psi) =$ *norm* $(($*id-cblinfun* $-$ *Proj* $J') *_V U *_V \psi)$›
by (*simp add*: *cblinfun.diff-left*)
also have ‹$\ldots \leq$ *norm* $(($*id-cblinfun* $-$ *Proj* $J) *_V U *_V \psi)$›
proof −
from ‹$J \leq J'$›
have ‹*id-cblinfun* $-$ *Proj* $J \geq$ *id-cblinfun* $-$ *Proj* J'›
by (*simp add*: *Proj-mono*)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *Proj-fixes-image Proj-ortho-compl Proj-range adj-Proj cblinfun-apply-in-image cdot-square-norm cinner-adj-right cnorm-ge-square less-eq-cblinfun-def*)
qed
also have ‹$\ldots =$ *norm* $(U *_V \psi - $ *Proj* $J *_V U *_V \psi)$›
by (*simp add*: *cblinfun.diff-left*)
also from ‹$\psi \in$ *space-as-set* I› ‹*preserves U I J ε*›
have ‹$\ldots \leq \varepsilon$›
by (*auto simp*: *preserves-def*)
also have ‹$\ldots \leq \varepsilon'$›
using ‹$\varepsilon \leq \varepsilon'$›
by (*simp add*: *mult-right-mono*)
finally show ‹*norm* $(U *_V \psi - $ *Proj* $J' *_V U *_V \psi) \leq \varepsilon'$›
by *simp*
qed

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* ⇒ *'a::chilbert-space ccsubspace*›
fixes J :: ‹*'i* ⇒ *'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 ≥ (∑x∈X. J x)›
  assumes ε0: ‹ε ≥ 0›
  assumes [iff]: ‹finite X›
  assumes pres: ‹⋀x. x∈X ⟹ preserves U (I x) (J x) ε›
  shows ‹preserves U II JJ ε›
proof -
  have ‹preserves U (∑x∈X. I x) (∑x∈X. J x) ε›
  proof (rule preservesI-closure[where I'=‹(∑x∈X. space-as-set (I x))›])
    from ε0 show ‹ε ≥ 0› by -

    show ‹csubspace (∑x∈X. space-as-set (I x))›
      by (simp add: csubspace-set-sum)
    show ‹space-as-set (sum I X) ⊆ closure (∑x∈X. space-as-set (I x))›
      apply (rule eq-refl)
      apply (use ‹finite X› in induction)
      by (auto simp: sup-ccsubspace.rep-eq simp flip: closed-sum-def)

    fix ψ assume ‹ψ ∈ (∑x∈X. space-as-set (I x))›
    then obtain ψ' where ψ'I: ‹ψ' x ∈ space-as-set (I x)› and ψ'sum: ‹ψ = (∑x∈X. ψ' x)› for x
    proof (atomize-elim, use ‹finite X› in ‹induction arbitrary: ψ›)
      case empty
      then show ?case
        by (auto intro!: exI[where x=‹λ-. 0›])
    next
      case (insert x X)
      have aux: ‹ψ ∈ space-as-set (I x) + (∑x∈X. space-as-set (I x)) ⟹
        ∃ψ0 ψ1. ψ = ψ0 + ψ1 ∧ ψ0 ∈ (∑x∈X. space-as-set (I x)) ∧ ψ1 ∈ space-as-set (I x)›
        by (metis add.commute set-plus-elim)
      from insert.prems
      obtain ψ0 ψ1 where ψ-decomp: ‹ψ = ψ0 + ψ1› and ψ0: ‹ψ0 ∈ (∑x∈X. space-as-set (I x))›
and ψ1: ‹ψ1 ∈ space-as-set (I x)›
        apply atomize-elim by (auto intro!: aux simp: insert.hyps)
      from insert.IH[OF ψ0]
      obtain ψ0' where ψ0'I: ‹ψ0' x ∈ space-as-set (I x)› and ψ0'sum: ‹ψ0 = sum ψ0' X› for x
        by auto
      define ψ' where ‹ψ' = ψ0'(x := ψ1)›
      have ‹ψ' x ∈ space-as-set (I x)› for x
        by (simp add: ψ'-def ψ0'I ψ1)
      moreover have ‹ψ = sum ψ' (insert x X)›
        by (metis ψ'-def ψ0'sum ψ-decomp add.commute fun-upd-apply insert.hyps(1) insert.hyps(2)
sum.cong sum.insert)
      ultimately show ?case
        by auto
    qed

    assume [simp]: ‹norm ψ = 1›

    define η' η where ‹η' x = U *V (ψ' x) - Proj (J x) *V U *V (ψ' x)› and ‹η = (∑x∈X. η' x)›
for x
    with pres have η'bound: ‹norm (η' x) ≤ ε * norm (ψ' x)› if ‹x∈X› for x
      using that by (simp add: ψ'I preserves-def)
    define US where ‹US x = U *S S x› for x

    have ‹ψ' x ∈ space-as-set (S x)› if ‹x∈X› for x
      using that ψ'I IS less-eq-ccsubspace.rep-eq by blast
```

then have $U\psi'S'$: ‹$U *_V \psi'\ x \in$ *space-as-set* $(S'\ x)$› **if** ‹$x{\in}X$› **for** x
using *USS'*[*OF that*] *that*
by (*metis cblinfun-image.rep-eq closure-subset imageI in-mono less-eq-ccsubspace.rep-eq*)

have $\eta'S'$: ‹$\eta'\ x \in$ *space-as-set* $(S'\ x)$› **if** ‹$x{\in}X$› **for** x
proof −
have ‹*Proj* $(J\ x) *_V U *_V (\psi'\ x) \in$ *space-as-set* $(J\ x)$›
by (*metis Proj-range cblinfun-apply-in-image*)
also have ‹$\ldots \subseteq$ *space-as-set* $(S'\ x)$›
unfolding *US-def less-eq-ccsubspace.rep-eq*[*symmetric*] **using** *JS' that* **by** *auto*
finally have ∗: ‹*Proj* $(J\ x) *_V U *_V (\psi'\ x) \in$ *space-as-set* $(S'\ x)$›
by −
with $U\psi'S'$[*OF that*]
show ‹$\eta'\ x \in$ *space-as-set* $(S'\ x)$›
unfolding η'*-def*
by (*metis Proj-fixes-image Proj-range cblinfun.diff-right cblinfun-apply-in-image*)
qed
from *ortho-S' USS'*
have *ortho-US*: ‹*orthogonal-spaces* (*US x*) (*US y*)› **if** ‹$x \neq y$› **and** ‹$x{\in}X$› **and** ‹$y{\in}X$› **for** $x\ y$
by (*metis US-def in-mono less-eq-ccsubspace.rep-eq orthogonal-spaces-def*
that(*1*,*2*,*3*))
have *ortho-I*: ‹*orthogonal-spaces* $(I\ x)\ (I\ y)$› **if** ‹$x \neq y$› **and** ‹$x{\in}X$› **and** ‹$y{\in}X$› **for** $x\ y$
by (*meson IS less-eq-ccsubspace.rep-eq ortho-S orthogonal-spaces-def subsetD that*)
have *ortho-J*: ‹*orthogonal-spaces* $(J\ x)\ (J\ y)$› **if** ‹$x \neq y$› **and** ‹$x{\in}X$› **and** ‹$y{\in}X$› **for** $x\ y$
using *JS' ortho-S' that*
by (*meson less-eq-ccsubspace.rep-eq orthogonal-spaces-def subsetD*)

from *ortho-S'* $\eta'S'$
have η'*ortho*: ‹*is-orthogonal* $(\eta'\ x)\ (\eta'\ y)$› **if** ‹$x \neq y$› **and** ‹$x{\in}X$› **and** ‹$y{\in}X$› **for** $x\ y$
by (*meson orthogonal-spaces-def that*)
have ψ'*ortho*: ‹*is-orthogonal* $(\psi'\ x)\ (\psi'\ y)$› **if** ‹$x \neq y$› **and** ‹$x{\in}X$› **and** ‹$y{\in}X$› **for** $x\ y$
using $\psi'I$ *ortho-I orthogonal-spaces-def that* **by** *blast*

have $\eta'2$: ‹$\eta'\ x = U *_V \psi'\ x -$ *Proj* $(\sum x{\in}X.\ (J\ x)) *_V U *_V \psi'\ x$› **if** ‹$x \in X$› **for** x
proof −
have ‹*Proj* $(J\ y) *_V U *_V \psi'\ x = 0$› **if** ‹$x \neq y$› **and** ‹$y \in X$› **for** y
proof −
have ‹$U *_V \psi'\ x \in$ *space-as-set* $(S'\ x)$›
using ‹$x \in X$› **by** (*rule* $U\psi'S'$)
moreover have ‹*orthogonal-spaces* $(S'\ x)\ (J\ y)$›
using *JS'*[*OF* ‹$y{\in}X$›] *ortho-S'*[*OF* ‹$x{\in}X$› ‹$y{\in}X$› ‹$x{\neq}y$›]
by (*meson less-eq-ccsubspace.rep-eq orthogonal-spaces-def subset-eq*)
ultimately show *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) *Proj-fixes-image Proj-ortho-compl Proj-range Set.basic-monos*(*7*)
cancel-comm-monoid-add-class.diff-cancel cblinfun.diff-left cblinfun.diff-right cblinfun-apply-in-image id-cblinfun.rep-eq less-eq-ccsubspace.rep-eq orthogonal-spaces-leq-compl)
qed
then have ‹$\eta'\ x = U *_V \psi'\ x -$ *Proj* $(J\ x) *_V U *_V \psi'\ x - (\sum y{\in}X{-}\{x\}.$ *Proj* $(J\ y) *_V U *_V$
$\psi'\ x)$›
unfolding η'*-def*
by (*metis* (*no-types*, *lifting*) *DiffE Diff-insert-absorb diff-0-right mk-disjoint-insert sum.not-neutral-contains-not-neut*
also have ‹$\ldots = U *_V \psi'\ x - (\sum y{\in}X.$ *Proj* $(J\ y) *_V U *_V \psi'\ x)$›
apply (*subst* (*2*) *asm-rl*[*of* ‹$X = \{x\} \cup (X{-}\{x\})$›])
apply (*simp add*: *insert-absorb* ‹$x \in X$›)
apply (*subst sum.union-disjoint*)

by *auto*
also have ‹... $= U *_V \psi'\ x - (\sum y{\in}X.\ Proj\ (J\ y)) *_V\ U *_V\ \psi'\ x$›
by (*simp add*: *cblinfun.sum-left*)
also have ‹... $= U *_V \psi'\ x - Proj\ (\sum y{\in}X.\ J\ y) *_V\ U *_V\ \psi'\ x$›
apply (*subst Proj-sum-spaces*)
using *ortho-J* **by** *auto*
finally show *?thesis*
by −
qed

have ‹$norm\ (U *_V \psi - Proj\ (sum\ J\ X) *_V\ U *_V\ \psi) = norm\ (\sum x{\in}X.\ U *_V\ \psi'\ x - Proj\ (sum\ J\ X) *_V\ U *_V\ \psi'\ x)$›
by (*simp add*: *ψ′sum sum-subtractf cblinfun.sum-right*)
also from *η′2* **have** ‹... $= norm\ (\sum x{\in}X.\ \eta'\ x)$›
by *simp*
also have ‹... $= norm\ \eta$›
using *η-def* **by** *blast*
also have ‹$(norm\ \eta)^2 = (\sum x{\in}X.\ (norm\ (\eta'\ x))^2)$›
unfolding *η-def*
apply (*rule pythagorean-theorem-sum*)
using *η′ortho* **by** *auto*
also have ‹... $\le (\sum x{\in}X.\ (\varepsilon * norm\ (\psi'\ x))^2)$›
apply (*rule sum-mono*)
by (*simp add*: *η′bound power-mono*)
also have ‹... $\le \varepsilon^2 * (\sum x{\in}X.\ (norm\ (\psi'\ x))^2)$›
by (*simp add*: *sum-distrib-left power-mult-distrib*)
also have ‹... $= \varepsilon^2 * (norm\ \psi)^2$›
proof −
have *aux*: ‹$a \in X \Longrightarrow a' \in X \Longrightarrow a \neq a' \Longrightarrow \psi = sum\ \psi'\ X \Longrightarrow is\text{-}orthogonal\ (\psi'\ a)\ (\psi'\ a')$› **for** *a a′*
by (*meson ψ′I IS less-eq-ccsubspace.rep-eq ortho-S orthogonal-spaces-def subset-iff*)
show *?thesis*
apply (*subst pythagorean-theorem-sum*[*symmetric*])
using *ψ′sum aux* **by** *auto*
qed
finally show ‹$norm\ (U *_V \psi - Proj\ (sum\ J\ X) *_V\ U *_V\ \psi) \le \varepsilon$›
using ‹$\varepsilon \ge 0$› ‹$norm\ \psi = 1$› **by** (*auto simp flip*: *power-mult-distrib*)
qed
then show *?thesis*
apply (*rule preserves-mono*)
using *assms* **by** *auto*
qed

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* = ⊤›
unfolding *ket-invariant-def* **by** *simp*

lemma *ket-invariant-empty*[*simp*]: ‹*ket-invariant* {} = ⊥›
unfolding *ket-invariant-def* **by** *simp*

lemma *ket-invariant-Rep-ell2*: ‹$\psi \in$ *space-as-set* (*ket-invariant* I) $\longleftrightarrow$ ($\forall i \in -I$. *Rep-ell2* ψ i = *0*)›
  by (*simp add*: *ket-invariant-def space-ccspan-ket*)

lemma *ket-invariant-compl*: ‹*ket-invariant* ($-M$) = $-$ *ket-invariant* M›
proof −
  have ‹*ket-invariant* ($-M$) $\leq$ $-$ *ket-invariant* M› **for** M :: ‹′*a set*›
    unfolding *ket-invariant-def*
    apply (*rule ccspan-leq-ortho-ccspan*)
    by *auto*
  moreover have ‹$-$ *ket-invariant* M $\leq$ *ket-invariant* ($-M$)›
  proof (*rule ccsubspace-leI-unit*)
    fix ψ
    assume ‹$\psi \in$ *space-as-set* ($-$ *ket-invariant* M)›
    then have ‹*is-orthogonal* ψ φ› **if** ‹$\varphi \in$ *space-as-set* (*ket-invariant* M)› **for** φ
      using *that*
      by (*auto simp*: *uminus-ccsubspace.rep-eq orthogonal-complement-def*)
    then have ‹*is-orthogonal* (*ket* m) ψ› **if** ‹$m \in M$› **for** m
      by (*simp add*: *ccspan-superset′ is-orthogonal-sym ket-invariant-def that*)
    then have ‹*Rep-ell2* ψ m = *0*› **if** ‹$m \in M$› **for** m
      by (*simp add*: *cinner-ket-left that*)
    then show ‹$\psi \in$ *space-as-set* (*ket-invariant* ($-$ M))›
      unfolding *ket-invariant-Rep-ell2*
      by *simp*
  qed
  ultimately show *?thesis*
    by (*rule order.antisym*)
qed

lemma *ket-invariant-tensor*: ‹*ket-invariant* I $\otimes_S$ *ket-invariant* J = *ket-invariant* ($I \times J$)›
proof −
  have ‹*ket-invariant* I $\otimes_S$ *ket-invariant* J = *ccspan* $\{x \otimes_s y \mid x\ y.\ x \in$ *ket* ‘ $I \wedge y \in$ *ket* ‘ $J\}$›
    by (*simp add*: *tensor-ccsubspace-ccspan ket-invariant-def*)
  also have ‹. . . = *ccspan* $\{$*ket* $(x, y) \mid x\ y.\ x \in I \wedge y \in J\}$›
    by (*auto intro*!: *arg-cong*[**where** *f*=*ccspan*] *simp flip*: *tensor-ell2-ket*)
  also have ‹. . . = *ccspan* (*ket* ‘ ($I \times J$))›
    by (*auto intro*!: *arg-cong*[**where** *f*=*ccspan*])
  also have ‹. . . = *ket-invariant* ($I \times J$)›
    by (*simp add*: *ket-invariant-def*)
  finally show *?thesis*
    by −
qed

abbreviation ‹*preserves-ket* U I J ε $\equiv$ *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
  apply *rule*
  apply (*simp add*: *ket-invariant-def orthogonal-spaces-def*)
  apply (*metis Int-emptyI ccspan-superset imageI inf-commute ket-invariant-def orthogonal-ket subset-iff*)
  apply (*simp add*: *orthogonal-spaces-leq-compl ket-invariant-def*)
  by (*smt* (*verit*, *best*) *ccspan-leq-ortho-ccspan disjoint-iff-not-equal imageE orthogonal-ket*)

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

```
proof -
  have ‹x ∈ N›
    if ‹x ∈ M› and *: ‹⋀ψ. (∀ y. y ∉ M ⟶ Rep-ell2 ψ y = 0) ⟶ (∀ y. y ∉ N ⟶ Rep-ell2 ψ y =
0)› for x
    using *[of ‹ket x›]
    using ‹x ∈ M › by (auto simp: ket.rep-eq)
  then show ?thesis
    by (auto simp add: less-eq-ccsubspace.rep-eq subset-eq Ball-def ket-invariant-Rep-ell2)
qed

lemma ket-invariant-mono:
  assumes ‹I ⊆ J›
  shows ‹ket-invariant I ≤ ket-invariant J›
  using [[simp-trace]]
  by (simp add: assms)

lemma ket-invariant-Inf: ‹ket-invariant (Inf M) = Inf (ket-invariant ‘ M)›
proof (rule order.antisym)
  show ‹ket-invariant (⋂ M) ≤ Inf (ket-invariant ‘ M)›
    by (simp add: Inf-lower le-Inf-iff)
  show ‹Inf (ket-invariant ‘ M) ≤ ket-invariant (⋂ M)›
  proof (rule ccsubspace-leI-unit)
    fix ψ
    assume ‹ψ ∈ space-as-set (Inf (ket-invariant ‘ M))›
    then have ‹ψ ∈ space-as-set (ket-invariant N)› if ‹N ∈ M› for N
      by (metis Inf-lower imageI in-mono less-eq-ccsubspace.rep-eq that)
    then have ‹Rep-ell2 ψ n = 0› if ‹n ∉ N› and ‹N ∈ M› for n N
      using that by (auto simp: ket-invariant-Rep-ell2)
    then have ‹Rep-ell2 ψ n = 0› if ‹n ∉ Inf M› for n
      using that by blast
    then show ‹ψ ∈ space-as-set (ket-invariant (⋂ M))›
      by (meson ComplD ket-invariant-Rep-ell2)
  qed
qed


lemma ket-invariant-INF: ‹ket-invariant (INF x∈M. f x) = (INF x∈M. ket-invariant (f x))›
  by (simp add: image-image ket-invariant-Inf)


lemma ket-invariant-Sup: ‹ket-invariant (Sup M) = Sup (ket-invariant ‘ M)›
proof -
  have ‹ket-invariant (Sup M) = ket-invariant (- (Inf (uminus ‘ M)))›
    by (subst uminus-Inf, simp)
  also have ‹. . . = - ket-invariant (Inf (uminus ‘ M))›
    using ket-invariant-compl by blast
  also have ‹. . . = - Inf (ket-invariant ‘ uminus ‘ M)›
    using ket-invariant-Inf by auto
  also have ‹. . . = - Inf (uminus ‘ ket-invariant ‘ M)›
    by (metis (no-types, lifting) INF-cong image-image ket-invariant-compl)
  also have ‹. . . = Sup (ket-invariant ‘ M)›
    apply (subst uminus-Inf)
    by (metis (no-types, lifting) SUP-cong image-comp image-image o-apply ortho-involution)
  finally show ?thesis
    by -
```

qed

lemma *ket-invariant-SUP*: ‹*ket-invariant* (*SUP x*∈*M*. *f x*) = (*SUP x*∈*M*. *ket-invariant* (*f x*))›
  by (*simp add*: *image-image ket-invariant-Sup*)

lemma *ket-invariant-inter*: ‹*ket-invariant M* ⊓ *ket-invariant N* = *ket-invariant* (*M* ∩ *N*)› **for** *M N*
  using *ket-invariant-INF*[**where** *M*=*UNIV* **and** *f*=‹λ*x*. *if x then M else N*›]
  by (*smt* (*verit*) *INF-UNIV-bool-expand*)

lemma *ket-invariant-union*: ‹*ket-invariant M* ⊔ *ket-invariant N* = *ket-invariant* (*M* ∪ *N*)› **for** *M N*
  using *ket-invariant-SUP*[**where** *M*=*UNIV* **and** *f*=‹λ*x*. *if x then M else N*›]
  by (*smt* (*verit*) *SUP-UNIV-bool-expand*)

lemma *sum-ket-invariant*[*simp*]:
  assumes ‹*finite X*›
  shows ‹($\sum$ *x*∈*X*. *ket-invariant* (*M x*)) = *ket-invariant* ($\bigcup$ *x*∈*X*. *M x*)›
  using *assms* **apply** *induction*
  apply *auto* **using** *ket-invariant-union* **by** *blast*

lemma *ket-invariant-inj*[*simp*]:
  ‹*ket-invariant M* = *ket-invariant N* ⟷ *M* = *N*› **for** *M N*
  by (*metis dual-order.eq-iff ket-invariant-le*)

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*) $*_S$ ⊤›

lemma *lift-invariant-comp*:
  assumes [*simp*]: ‹*register G*›
  shows ‹*lift-invariant* (*F o G*) = *lift-invariant F o lift-invariant G*›
  by (*auto intro*!: *ext simp*: *lift-invariant-def Proj-on-own-range register-projector*)

lemma *lift-invariant-top*[*simp*]: ‹*register F* ⟹ *lift-invariant F* ⊤ = ⊤›
  by (*metis Proj-on-own-range′ cblinfun-compose-id-right id-cblinfun-adjoint lift-invariant-def register-unitary unitary-id unitary-range*)

lemma *Proj-lift-invariant*: ‹*register F* ⟹ *Proj* (*lift-invariant F I*) = *F* (*Proj I*)›
  using [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
  unfolding *lift-invariant-def*
  by (*simp add*: *Proj-on-own-range register-projector*)

lemma *ket-invariant-image-assoc*:
  ‹*ket-invariant* ((λ((*a*, *b*), *c*). (*a*, *b*, *c*)) ‘ *X*) = *lift-invariant assoc* (*ket-invariant X*)›
proof −
  have ‹*ket-invariant* ((λ((*a*, *b*), *c*). (*a*, *b*, *c*)) ‘ *X*) = *assoc-ell2* $*_S$ *ket-invariant X*›
    by (*auto intro*!: *arg-cong*[**where** *f*=*ccspan*] *image-eqI simp add*: *ket-invariant-def image-image cblinfun-image-ccspan*)
  also have ‹. . . = *lift-invariant assoc* (*ket-invariant X*)›
    by (*simp add*: *lift-invariant-def assoc-ell2-sandwich Proj-sandwich*)
  finally show *?thesis*
    by −
qed

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

proof (*rule iffI*[*rotated*], *simp*)
  assume *asm*: ‹*lift-invariant F I* = *lift-invariant F J*›
  then have ‹F (*Proj I*) $*_S$ ⊤ = F (*Proj J*) $*_S$ ⊤›
    by (*simp add*: *lift-invariant-def*)
  then have ‹F (*Proj I*) = F (*Proj J*)›
    by (*metis Proj-lift-invariant asm that*)
  then have ‹*Proj I* = *Proj J*›
    by (*simp add*: *register-inj′*)
  then show ‹$I = J$›
    using *Proj-inj* **by** *blast*
qed

lemma *lift-invariant-decomp*:
  fixes U :: ‹- $\Rightarrow_{CL}$ -::*chilbert-space*›
  assumes ‹$\bigwedge\vartheta$. $F\ \vartheta$ = *sandwich U* $*_V$ ($\vartheta \otimes_o$ *id-cblinfun*)›
  assumes ‹*unitary U*›
  shows ‹*lift-invariant F I* = $U *_S$ ($I \otimes_S$ ⊤)›
  by (*simp add*: *lift-invariant-def assms Proj-tensor-Proj Proj-sandwich flip*: *Proj-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* ⟷ *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*›
proof −
  have ‹*is-Proj* (*Proj A* o_{CL} *Proj B*)›
    apply (*rule is-Proj-I*)
    apply (*metis Proj-idempotent cblinfun-assoc-left*(*1*) *compatible-invariants-def that*)
    by (*metis adj-Proj adj-cblinfun-compose compatible-invariants-def that*)

  have ‹(*Proj A* o_{CL} *Proj B*) $*_S$ ⊤ ≤ A›
    by (*simp add*: *Proj-image-leq cblinfun-compose-image*)
  moreover have ‹(*Proj A* o_{CL} *Proj B*) $*_S$ ⊤ ≤ B›
    using *that* **by** (*simp add*: *Proj-image-leq cblinfun-compose-image compatible-invariants-def*)
  ultimately have *leq1*: ‹(*Proj A* o_{CL} *Proj B*) $*_S$ ⊤ ≤ $A \sqcap B$›
    by *auto*

  have *leq2*: ‹$A \sqcap B$ ≤ (*Proj A* o_{CL} *Proj B*) $*_S$ ⊤›
  proof (*rule ccsubspace-leI*, *rule subsetI*)
    fix ψ **assume** ‹ψ ∈ *space-as-set* ($A \sqcap B$)›
    then have ‹*Proj A* $*_V\ \psi = \psi$› ‹*Proj B* $*_V\ \psi = \psi$›
      by (*simp-all add*: *Proj-fixes-image*)
    then have ‹ψ = (*Proj A* o_{CL} *Proj B*) $*_V\ \psi$›
      by *simp*
    also have ‹(*Proj A* o_{CL} *Proj B*) $*_V\ \psi$ ∈ *space-as-set* ((*Proj A* o_{CL} *Proj B*) $*_S$ ⊤)›
      using *cblinfun-apply-in-image* **by** *blast*
    finally show ‹ψ ∈ *space-as-set* ((*Proj A* o_{CL} *Proj B*) $*_S$ ⊤)›
      by −
  qed

  from *leq1 leq2* **have** ‹(*Proj A* o_{CL} *Proj B*) $*_S$ ⊤ = $A \sqcap B$›
    using *order-class.order-eq-iff* **by** *blast*

with ‹*is-Proj* (*Proj A* o_{CL} *Proj B*)› **show** ‹*Proj A* o_{CL} *Proj B* = *Proj* (*A* ⊓ *B*)›
  using *Proj-on-own-range* **by** *force*
qed

lemma *compatible-invariants-ket*[*iff*]: ‹*compatible-invariants* (*ket-invariant I*) (*ket-invariant J*)›
proof −
  have *I*: ‹*Proj* (*ket-invariant I*) = *Proj* (*ket-invariant* (*I*−*J*)) + *Proj* (*ket-invariant* (*I*∩*J*))›
    apply (*subst Proj-sup*[*symmetric*])
    by (*auto simp add*: *Un-Diff-Int ket-invariant-union*)
  have *J*: ‹*Proj* (*ket-invariant J*) = *Proj* (*ket-invariant* (*J*−*I*)) + *Proj* (*ket-invariant* (*I*∩*J*))›
    apply (*subst Proj-sup*[*symmetric*])
    by (*auto intro*!: *arg-cong*[**where** *f*=*Proj*] *simp add*: *Un-Diff-Int ket-invariant-union*)
  have ‹*Proj* (*ket-invariant I*) o_{CL} *Proj* (*ket-invariant J*) = *Proj* (*ket-invariant J*) o_{CL} *Proj* (*ket-invariant I*)›
    apply (*simp add*: *I J*)
    by (*smt* (*verit*) *Diff-disjoint I Int-Diff-disjoint Proj-bot adj-Proj adj-cblinfun-compose cblinfun-compose-add-left cblinfun-compose-add-right orthogonal-projectors-orthogonal-spaces orthogonal-spaces-ket*)
  then show *?thesis*
    by (*simp add*: *compatible-invariants-def*)
qed

lemma *preserves-intersect*:
  assumes ‹*compatible-invariants J1 J2*›
  assumes *pres1*: ‹*preserves U I J1 ε1*›
  assumes *pres2*: ‹*preserves U I J2 ε2*›
  shows ‹*preserves U I* (*J1* ⊓ *J2*) (*ε1* + *ε2*)›

proof (*rule preservesI*)
  show ‹*0* ≤ *ε1* + *ε2*›
    by (*meson add-nonneg-nonneg pres1 pres2 preserves-def*)

  fix *ψ* **assume** ‹*ψ* ∈ *space-as-set I*› **and** ‹*norm ψ* = *1*›
  define *φ J* **where** ‹*φ* = *U* $*_V$ *ψ*› **and** ‹*J* = *J1* ⊓ *J2*›

  note *norm-diff-triangle-le*[*trans*]

  from *pres1*
  have ‹*norm* (*φ* − *Proj J1* $*_V$ *φ*) ≤ *ε1*›
    by (*metis* ‹*ψ* ∈ *space-as-set I*› ‹*norm ψ* = *1*› *φ-def mult-cancel-left1 preserves-def*)
  also
  have ‹*norm* (*φ* − *Proj J2* $*_V$ *φ*) ≤ *ε2*›
    using ‹*ψ* ∈ *space-as-set I*› ‹*norm ψ* = *1*› *φ-def pres2 preserves-def* **by** *force*
  then have ‹*norm* (*Proj J1* $*_V$ (*φ* − *Proj J2* $*_V$ *φ*)) ≤ *ε2*›
    using *Proj-is-Proj is-Proj-reduces-norm order-trans* **by** *blast*
  then have ‹*norm* (*Proj J1* $*_V$ *φ* − *Proj J1* $*_V$ *Proj J2* $*_V$ *φ*) ≤ *ε2*›
    by (*simp add*: *cblinfun.diff-right*)
  also have ‹*Proj J1* $*_V$ *Proj J2* $*_V$ *φ* = *Proj J* $*_V$ *φ*›
    by (*metis J-def assms*(*1*) *cblinfun-apply-cblinfun-compose compatible-invariants-inter*)
  finally show ‹*norm* (*φ* − *Proj J* $*_V$ *φ*) ≤ *ε1* + *ε2*›
    by −
qed

lemma *preserves-intersect-ket*:
  assumes ‹*preserves-ket U I J1 ε1*›

assumes ‹*preserves-ket U I J2 ε2*›
shows ‹*preserves-ket U I* (*J1* ⊓ *J2*) (*ε1* + *ε2*)›
apply (*simp flip*: *ket-invariant-inter*)
using - *assms* **apply** (*rule preserves-intersect*)
by (*rule compatible-invariants-ket*)

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* ⟷ (∀ *A*. *Proj I* o_{CL} *F A* = *F A* o_{CL} *Proj I*)›
for *F* :: ‹*'a update* ⇒ *'b update*›

lemma *compatible-register-invariant-top*[*simp*]:
‹*compatible-register-invariant F* ⊤›
by (*simp add*: *compatible-register-invariant-def*)

lemma *compatible-register-invariant-bot*[*simp*]:
‹*compatible-register-invariant F* ⊥›
by (*simp add*: *compatible-register-invariant-def*)

lemma *compatible-register-invariant-id*:
assumes ‹⋀*y*. *I* = *UNIV* ∨ *I* = {}›
shows ‹*compatible-register-invariant id* (*ket-invariant I*)›
using *assms*
by (*metis compatible-register-invariant-bot compatible-register-invariant-top ket-invariant-UNIV ket-invariant-empty*)

lemma *compatible-register-invariant-compatible-register*:
assumes ‹*compatible F G*›
shows ‹*compatible-register-invariant F* (*lift-invariant G I*)›
unfolding *compatible-register-invariant-def lift-invariant-def*
by (*metis Proj-is-Proj Proj-on-own-range assms compatible-def register-projector*)

lemma *compatible-register-invariant-chain*[*simp*]:
‹*compatible-register-invariant* (*F o G*) (*lift-invariant F I*) ⟷ *compatible-register-invariant G I*› **if** [*simp*]: ‹*register F*›
by (*simp add*: *compatible-register-invariant-def Proj-lift-invariant register-mult register-inj*[*THEN inj-eq*])

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* ⇒ *'b update*›
assumes *pres*: ‹*preserves U' I' J' ε*›
assumes *reg*[*register*]: ‹*register F*›
assumes *compat*: ‹*compatible-register-invariant F K*›
assumes *FU'*: ‹∀ *ψ*∈*space-as-set I*. *F U'* $*_V$ *ψ* = *U* $*_V$ *ψ*›
assumes *FI'-I*: ‹*lift-invariant F I'* ≥ *I*›
assumes *KI*: ‹*K* ≥ *I*›
assumes *FJ'K-I*: ‹*lift-invariant F J'* ⊓ *K* ≤ *J*›
shows ‹*preserves U I J ε*›
proof −
define *PI' PJ'* **where** ‹*PI'* = *Proj I'*› **and** ‹*PJ'* = *Proj J'*›
have *1*: ‹*preserves* (*F U'*) (*lift-invariant F I'*) (*lift-invariant F J'*) *ε*›
proof (*unfold preserves-onorm*)
have ‹*norm* ((*id-cblinfun* − *Proj* (*lift-invariant F J'*)) o_{CL} *F U'* o_{CL} *Proj* (*lift-invariant F I'*))
= *norm* ((*id-cblinfun* − *PJ'*) o_{CL} *U'* o_{CL} *PI'*)› (**is** ‹*?lhs* = -›)

by (*smt* (*verit*, *best*) *PI′-def PJ′-def Proj-lift-invariant reg register-minus register-mult register-norm register-of-id*)
also from *pres* **have** ‹. . . ≤ *ε*›
by (*simp add*: *preserves-onorm PJ′-def PI′-def*)
finally show ‹*?lhs* ≤ *ε*›
by −
qed

from *compat*
have *2*: ‹*preserves* (*F U′*) *K K 0*›
by (*simp add*: *preserves-onorm cblinfun-compose-assoc cblinfun-compose-minus-left compatible-register-invariant-def*)

with *1 compat*
have ‹*preserves* (*F U′*) (*lift-invariant F I′* ⊓ *K*) (*lift-invariant F J′* ⊓ *K*) *ε*›
apply (*subst asm-rl*[*of* ‹*ε* = *ε* + *0*›], *simp*)
apply (*rule preserves-intersect*)
by (*auto simp add*: *compatible-invariants-def compatible-register-invariant-def preserves-mono Proj-lift-invariant*)

then have ‹*preserves* (*F U′*) *I J ε*›
apply (*rule preserves-mono*)
using *FI′-I FJ′K-I KI* **by** *auto*
then show *?thesis*
apply (*rule preserves-cong*[*THEN iffD1*, *rotated*])
using *FU′* **by** *auto*
qed

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

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

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

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* {$x \otimes_s y$ | *x y*. $x \in$ *space-as-set I* ∧ $y \in$ *space-as-set J*}›

lemma *tensor-invariant-via-Proj*: ‹*tensor-invariant I J* = (*Proj I* $\otimes_o$ *Proj J*) $*_S$ ⊤›
proof (*rule Proj-inj*, *rule tensor-ell2-extensionality*, *rename-tac ψ φ*)
fix *ψ φ*
define *ψ1 ψ2* **where** ‹*ψ1* = *Proj I ψ*› **and** ‹*ψ2* = *Proj* (−*I*) *ψ*›
have ‹*ψ* = *ψ1* + *ψ2*›
by (*simp add*: *ψ1-def ψ2-def Proj-ortho-compl minus-cblinfun.rep-eq*)
have *ψ1I*: ‹*ψ1* ∈ *space-as-set I*›
by (*metis Proj-idempotent ψ1-def cblinfun-apply-cblinfun-compose norm-Proj-apply*)

define *φ1 φ2* **where** ‹*φ1* = *Proj J φ*› **and** ‹*φ2* = *Proj* (−*J*) *φ*›
have ‹*φ* = *φ1* + *φ2*›
by (*simp add*: *φ1-def φ2-def Proj-ortho-compl minus-cblinfun.rep-eq*)
have *φ1J*: ‹*φ1* ∈ *space-as-set J*›
by (*metis Proj-idempotent φ1-def cblinfun-apply-cblinfun-compose norm-Proj-apply*)

have *aux*: ‹*xa* ∈ *space-as-set I* ⟹ *y* ∈ *space-as-set J* ⟹ $\varphi \bullet_C y \neq 0$ ⟹ *is-orthogonal ψ2 xa*› **for**

xa y
by (*metis Proj-fixes-image* ‹$\psi = \psi 1 + \psi 2$› *ψ1I ψ1-def add-left-imp-eq cblinfun.real.add-right kernel-Proj kernel-memberI orthogonal-complement-orthoI pth-d uminus-ccsubspace.rep-eq*)
have ‹$\psi 2 \otimes_s \varphi \in$ *space-as-set* $(-$ *tensor-invariant I J*$)$›
by (*auto intro*!: *aux orthogonal-complementI simp add*: *uminus-ccsubspace.rep-eq tensor-invariant-def ccspan.rep-eq*
simp flip: *orthogonal-complement-of-closure orthogonal-complement-of-cspan*)
then have *ψ2φ*: ‹*Proj* (*tensor-invariant I J*) $*_V$ $(\psi 2 \otimes_s \varphi) = 0$›
by (*simp add*: *kernel-memberD*)

have *aux*: ‹$xa \in$ *space-as-set I* $\Longrightarrow y \in$ *space-as-set J* $\Longrightarrow \varphi 2 \cdot_C y \neq 0 \Longrightarrow$ *is-orthogonal* $\psi 1$ *xa*› **for** *xa y*
by (*metis Proj-fixes-image* ‹$\varphi = \varphi 1 + \varphi 2$› *φ1J φ1-def add-left-imp-eq cblinfun.real.add-right kernel-Proj kernel-memberI orthogonal-complement-orthoI pth-d uminus-ccsubspace.rep-eq*)
have ‹$\psi 1 \otimes_s \varphi 2 \in$ *space-as-set* $(-$ *tensor-invariant I J*$)$›
by (*auto intro*!: *aux orthogonal-complementI simp add*: *uminus-ccsubspace.rep-eq tensor-invariant-def ccspan.rep-eq*
simp flip: *orthogonal-complement-of-closure orthogonal-complement-of-cspan*)
then have *ψ1φ2*: ‹*Proj* (*tensor-invariant I J*) $*_V$ $(\psi 1 \otimes_s \varphi 2) = 0$›
by (*simp add*: *kernel-memberD*)

have *ψ1φ1*: ‹*Proj* (*tensor-invariant I J*) $*_V$ $(\psi 1 \otimes_s \varphi 1) = \psi 1 \otimes_s \varphi 1$›
by (*auto intro*!: *Proj-fixes-image space-as-set-ccspan-memberI exI*[*of - ψ1*] *exI*[*of - φ1*]
simp: *tensor-invariant-def ψ1I φ1J*)

have *ProjProj*: ‹*Proj* ((*Proj I* $\otimes_o$ *Proj J*) $*_S$ ⊤) = *Proj I* $\otimes_o$ *Proj J*›
by (*simp add*: *Proj-on-own-range′ adj-Proj comp-tensor-op tensor-op-adjoint*)

show ‹*Proj* (*tensor-invariant I J*) $*_V$ $(\psi \otimes_s \varphi)$ = *Proj* ((*Proj I* $\otimes_o$ *Proj J*) $*_S$ ⊤) $*_V$ $(\psi \otimes_s \varphi)$›
apply (*simp add*: *ProjProj tensor-op-ell2 flip*: *ψ1-def φ1-def*)
apply (*simp add*: ‹$\psi = \psi 1 + \psi 2$› *tensor-ell2-add1 cblinfun.add-right ψ2φ*)
by (*simp add*: *ψ1φ1 ψ1φ2* ‹$\varphi = \varphi 1 + \varphi 2$› *tensor-ell2-add2 cblinfun.add-right*)
qed

lemma *tensor-invariant-mono-left*: ‹$I \leq I' \Longrightarrow$ *tensor-invariant I J* $\leq$ *tensor-invariant I′ J*›
by (*auto intro*!: *space-as-set-mono ccspan-mono simp add*: *tensor-invariant-def less-eq-ccsubspace.rep-eq*)

lemma *swap-tensor-invariant*[*simp*]: ‹*swap-ell2* $*_S$ *tensor-invariant I J* = *tensor-invariant J I*›
by (*force intro*!: *arg-cong*[**where** *f=ccspan*] *simp*: *cblinfun-image-ccspan tensor-invariant-def*)

lemma *tensor-invariant-SUP-left*: ‹*tensor-invariant* (*SUP x∈X. I x*) *J* = (*SUP x∈X. tensor-invariant* (*I x*) *J*)›
proof (*rule order.antisym*)
show ‹(*SUP x∈X. tensor-invariant* (*I x*) *J*) ≤ *tensor-invariant* (*SUP x∈X. I x*) *J*›
by (*auto intro*!: *SUP-least tensor-invariant-mono-left SUP-upper*)

have *tensor-left-apply*: ‹*CBlinfun* ($\lambda x.\ x \otimes_s y$) $*_V$ $x = x \otimes_s y$› **for** *x* :: ‹′*a ell2*› **and** *y* :: ‹′*b ell2*›
by (*simp add*: *bounded-clinear-tensor-ell22 bounded-clinear-CBlinfun-apply clinear-tensor-ell22*)

show ‹*tensor-invariant* (*SUP x∈X. I x*) *J* ≤ (*SUP x∈X. tensor-invariant* (*I x*) *J*)›
proof −
have ‹*tensor-invariant* (*SUP x∈X. I x*) *J* = *ccspan* {$x \otimes_s y$ |*x y. x* ∈ *space-as-set* (*SUP x∈X. I x*) ∧ *y* ∈ *space-as-set J*}›
by (*auto simp*: *tensor-invariant-def*)
also have ‹... = *ccspan* (⨆ *y∈space-as-set J*. {$x \otimes_s y$ |*x. x* ∈ *space-as-set* (*SUP x∈X. I x*)})›

```
      by (auto intro!: arg-cong[where f=ccspan])
    also have ‹... = (⨆y∈space-as-set J. ccspan {x ⊗s y |x. x ∈ space-as-set (SUP x∈X. I x)})›
      by (smt (verit) Sup.SUP-cong ccspan-Sup image-image)
    also have ‹... = (⨆y∈space-as-set J. ccspan (cblinfun-apply (CBlinfun (λx. x ⊗s y)) ‘ {x. x ∈
space-as-set (SUP x∈X. I x)}))›
      apply (rule SUP-cong, simp)
      apply (rule arg-cong[where f=ccspan])
      by (auto simp add: image-def tensor-left-apply)
    also have ‹... = (⨆y∈space-as-set J. CBlinfun (λx. x ⊗s y) *S (SUP x∈X. I x))›
      apply (subst cblinfun-image-ccspan[symmetric])
      by auto
    also have ‹... = (⨆y∈space-as-set J. (SUP x∈X. CBlinfun (λx. x ⊗s y) *S I x))›
      apply (subst cblinfun-image-SUP)
      by simp
    also have ‹... ≤ (⨆x∈X. tensor-invariant (I x) J)›
    proof (rule SUP-least)
      fix y
      assume ‹y ∈ space-as-set J›
      have ‹(CBlinfun (λx. x ⊗s y) *S I x) ≤ (tensor-invariant (I x) J)› for x
        apply (rule ccsubspace-leI)
        apply (simp add: tensor-invariant-def cblinfun-image.rep-eq ccspan.rep-eq image-def
            tensor-left-apply)
        apply (rule closure-mono)
        by (auto intro!: complex-vector.span-base ‹y ∈ space-as-set J›)
      then show ‹(SUP x∈X. CBlinfun (λx. x ⊗s y) *S I x) ≤ (SUP x∈X. tensor-invariant (I x) J)›
        by (auto intro!: SUP-mono)
    qed
    finally show ‹tensor-invariant (⨆ (I ‘ X)) J ≤ (⨆x∈X. tensor-invariant (I x) J)›
      by −
  qed
qed

lemma tensor-invariant-SUP-right: ‹tensor-invariant I (SUP x∈X. J x) = (SUP x∈X. tensor-invariant
I (J x))›
proof −
  have ‹tensor-invariant I (SUP x∈X. J x) = swap-ell2 *S tensor-invariant (SUP x∈X. J x) I›
    by simp
  also have ‹... = swap-ell2 *S (SUP x∈X. tensor-invariant (J x) I)›
    by (simp add: tensor-invariant-SUP-left)
  also have ‹... = (SUP x∈X. swap-ell2 *S tensor-invariant (J x) I)›
    using cblinfun-image-SUP by blast
  also have ‹... = (SUP x∈X. tensor-invariant I (J x))›
    by simp
  finally show ?thesis
    by −
qed

lemma tensor-invariant-bot-left[simp]: ‹tensor-invariant ⊥ J = ⊥›
  using tensor-invariant-SUP-left[where I=id and X=‹{}› and J=J]
  by simp

lemma tensor-invariant-bot-right[simp]: ‹tensor-invariant I ⊥ = ⊥›
  using tensor-invariant-SUP-right[where J=id and X=‹{}› and I=I]
  by simp
```

lemma *tensor-invariant-Sup-left*: ‹*tensor-invariant* (*Sup II*) *J* = (*SUP I*∈*II*. *tensor-invariant I J*)›
  using *tensor-invariant-SUP-left*[**where** *X*=*II* **and** *I*=*id* **and** *J*=*J*]
  by *simp*

lemma *tensor-invariant-Sup-right*: ‹*tensor-invariant I* (*Sup JJ*) = (*SUP J*∈*JJ*. *tensor-invariant I J*)›
  using *tensor-invariant-SUP-right*[**where** *X*=*JJ* **and** *I*=*I* **and** *J*=*id*]
  by *simp*

lemma *tensor-invariant-sup-left*: ‹*tensor-invariant* (*I1* ⊔ *I2*) *J* = *tensor-invariant I1 J* ⊔ *tensor-invariant I2 J*›
  using *tensor-invariant-Sup-left*[**where** *II*=‹{*I1*,*I2*}›]
  by *auto*

lemma *tensor-invariant-sup-right*: ‹*tensor-invariant I* (*J1* ⊔ *J2*) = *tensor-invariant I J1* ⊔ *tensor-invariant I J2*›
  using *tensor-invariant-Sup-right*[**where** *JJ*=‹{*J1*,*J2*}›]
  by *auto*

lemma *compatible-register-invariant-compl*: ‹*compatible-register-invariant F I* ⟹ *compatible-register-invariant F* (−*I*)›
  by (*simp add*: *compatible-register-invariant-def Proj-ortho-compl cblinfun-compose-minus-left cblinfun-compose-minus-right*)

lemma *compatible-register-invariant-SUP*:
  assumes [*simp*]: ‹*register F*›
  assumes *compat*: ‹⋀*x*. *x* ∈ *X* ⟹ *compatible-register-invariant F* (*I x*)›
  shows ‹*compatible-register-invariant F* (*SUP x*∈*X*. *I x*)›
proof −
  from *register-decomposition*[*OF* ‹*register F*›]
  have ‹*let* ′*d*::*type* = *register-decomposition-basis F in ?thesis*›
  proof *with-type-mp*
    case *with-type-mp*
    then obtain *U* :: ‹(′*a* × ′*d*) *ell2* $\Rightarrow_{CL}$ ′*b ell2*›
      where [*iff*]: ‹*unitary U*› **and** *FU*: ‹*F* ϑ = *sandwich U* $*_V$ (ϑ $\otimes_o$ *id-cblinfun*)› **for** ϑ
      by *auto*
    have ∗: ‹*Proj* (*I x*) o_{CL} *U* o_{CL} (*A* $\otimes_o$ *id-cblinfun*) o_{CL} *U*∗ = *U* o_{CL} (*A* $\otimes_o$ *id-cblinfun*) o_{CL} *U*∗ o_{CL} *Proj* (*I x*)› **if** ‹*x* ∈ *X*› **for** *x A*
      using *compat*[*OF that*]
      by (*simp add*: *compatible-register-invariant-def FU sandwich-apply cblinfun-compose-assoc*)
    have ‹(*U*∗ o_{CL} *Proj* (*I x*) o_{CL} *U*) o_{CL} (*A* $\otimes_o$ *id-cblinfun*) = (*A* $\otimes_o$ *id-cblinfun*) o_{CL} (*U*∗ o_{CL} *Proj* (*I x*) o_{CL} *U*)› **if** ‹*x* ∈ *X*› **for** *x A*
        using ∗[**where** *A*=*A*, *OF that*, *THEN arg-cong*, **where** *f*=‹λ*x*. *U*∗ o_{CL} *x*›, *THEN arg-cong*, **where** *f*=‹λ*x*. *x* o_{CL} *U*›]
      apply (*simp add*: *cblinfun-compose-assoc*)
      by (*simp flip*: *cblinfun-compose-assoc*)
    then have ‹*Proj* (*U*∗ $*_S$ *I x*) o_{CL} (*A* $\otimes_o$ *id-cblinfun*) = (*A* $\otimes_o$ *id-cblinfun*) o_{CL} *Proj* (*U*∗ $*_S$ *I x*)› **if** ‹*x* ∈ *X*› **for** *x A*
      using *that*
      by (*simp flip*: *Proj-sandwich add*: *sandwich-apply*)
    then have ‹*Proj* (*U*∗ $*_S$ *I x*) ∈ *commutant* (*range* (λ*A*. *A* $\otimes_o$ *id-cblinfun*))› **if** ‹*x* ∈ *X*› **for** *x*
      unfolding *commutant-def* **using** *that* **by** *auto*
    then have ‹*Proj* (*U*∗ $*_S$ *I x*) ∈ *range* (λ*B*. *id-cblinfun* $\otimes_o$ *B*)› **if** ‹*x* ∈ *X*› **for** *x*
      by (*simp add*: *commutant-tensor1 that*)
    then obtain π **where** ∗: ‹*Proj* (*U*∗ $*_S$ *I x*) = *id-cblinfun* $\otimes_o$ π *x*› **if** ‹*x* ∈ *X*› **for** *x*
      apply *atomize-elim*

apply (*rule choice*)
by (*simp add*: *image-iff*)
have π-*proj*: ‹*is-Proj* (π *x*)› **if** ‹$x \in X$› **for** *x*
proof −
have ‹*Proj* ($U* *_S I\ x$)$*$ = *Proj* ($U* *_S I\ x$)›
by (*simp add*: *adj-Proj*)
then have ‹(*id-cblinfun* :: *'a ell2* $\Rightarrow_{CL}$ -) $\otimes_o$ π *x* = *id-cblinfun* $\otimes_o$ π *x*$*$›
by (*simp add*: $*$[*OF that*] *tensor-op-adjoint*)
then have *1*: ‹π *x* = π *x*$*$›
using *inj-tensor-right*[*OF id-cblinfun-not-0*] *injD* **by** *fastforce*
have ‹*Proj* ($U* *_S I\ x$) o_{CL} *Proj* ($U* *_S I\ x$) = *Proj* ($U* *_S I\ x$)›
by *simp*
then have ‹(*id-cblinfun* :: *'a ell2* $\Rightarrow_{CL}$ -) $\otimes_o$ (π *x* o_{CL} π *x*) = *id-cblinfun* $\otimes_o$ π *x*›
by (*simp add*: $*$[*OF that*] *comp-tensor-op*)
then have *2*: ‹π *x* o_{CL} π *x* = π *x*›
using *inj-tensor-right*[*OF id-cblinfun-not-0*] *injD* **by** *fastforce*
from *1 2* **show** ‹*is-Proj* (π *x*)›
by (*simp add*: *is-Proj-I*)
qed
define σ **where** ‹σ *x* = π *x* $*_S$ $\top$› **for** *x*
have $**$: ‹$U* *_S I\ x$ = *tensor-invariant* $\top$ (σ *x*)› **if** ‹$x \in X$› **for** *x*
using $*$[*OF that*, *THEN arg-cong*, **where** *f*=‹$\lambda t.\ t *_S \top$›]
by (*simp add*: *tensor-invariant-via-Proj* σ-*def Proj-on-own-range* π-*proj that*)
have ‹*sandwich* ($U*$) (*Proj* (*SUP x*$\in$*X*. *I x*)) = *Proj* ($U* *_S$ (*SUP x*$\in$*X*. *I x*))›
by (*smt* (*verit*) *sandwich-apply Proj-lift-invariant Proj-range* ‹*unitary U*› *cblinfun-compose-image unitary-adj unitary-range unitary-sandwich-register*)
also have ‹... = *Proj* (*SUP x*$\in$*X*. $U* *_S I\ x$)›
by (*simp add*: *cblinfun-image-SUP*)
also have ‹... = *Proj* (*SUP x*$\in$*X*. *tensor-invariant* $\top$ (σ *x*))›
using $**$ **by** *auto*
also have ‹... = *Proj* (*tensor-invariant* $\top$ (*SUP x*$\in$*X*. σ *x*))›
by (*simp add*: *tensor-invariant-SUP-right*)
also have ‹... = *id-cblinfun* $\otimes_o$ *Proj* (*SUP x*$\in$*X*. σ *x*)›
by (*simp add*: *Proj-on-own-range' adj-Proj comp-tensor-op tensor-invariant-via-Proj tensor-op-adjoint*)
also have ‹... $\in$ *commutant* (*range* (λA. *A* $\otimes_o$ *id-cblinfun*))›
by (*simp add*: *commutant-tensor1*)
finally have ‹($U*$ o_{CL} *Proj* (*SUP x*$\in$*X*. *I x*) o_{CL} *U*) o_{CL} (*A* $\otimes_o$ *id-cblinfun*) = (*A* $\otimes_o$ *id-cblinfun*) o_{CL} ($U*$ o_{CL} *Proj* (*SUP x*$\in$*X*. *I x*) o_{CL} *U*)› **for** *A*
by (*simp add*: *sandwich-apply commutant-def*)
from *this*[*THEN arg-cong*, **where** *f*=‹λx. *U* o_{CL} *x*›, *THEN arg-cong*, **where** *f*=‹λx. *x* o_{CL} $U*$›]
have ‹*Proj* (*SUP x*$\in$*X*. *I x*) o_{CL} *U* o_{CL} (*A* $\otimes_o$ *id-cblinfun*) o_{CL} $U*$ = *U* o_{CL} (*A* $\otimes_o$ *id-cblinfun*) o_{CL} $U*$ o_{CL} *Proj* (*SUP x*$\in$*X*. *I x*)› **for** *A*
apply (*simp add*: *cblinfun-compose-assoc*)
by (*simp flip*: *cblinfun-compose-assoc*)
then have ‹*Proj* (*SUP x*$\in$*X*. *I x*) o_{CL} *F A* = *F A* o_{CL} *Proj* (*SUP x*$\in$*X*. *I x*)› **for** *A*
by (*simp add*: *FU sandwich-apply cblinfun-compose-assoc*)
then show ‹*compatible-register-invariant F* (*SUP x*$\in$*X*. *I x*)›
by (*simp add*: *compatible-register-invariant-def*)
qed
from *this*[*cancel-with-type*]
show *?thesis*
by −
qed

lemma *compatible-register-invariant-INF*:

assumes [*simp*]: ‹*register F*›
assumes *compat*: ‹$\bigwedge$*x*. *x* ∈ *X* ⟹ *compatible-register-invariant F* (*I x*)›
shows ‹*compatible-register-invariant F* (*INF x*∈*X*. *I x*)›
proof −
from *compat* **have** ‹*compatible-register-invariant F* (− *I x*)› **if** ‹*x* ∈ *X*› **for** *x*
by (*simp add*: *compatible-register-invariant-compl that*)
then have ‹*compatible-register-invariant F* (*SUP x*∈*X*. − *I x*)›
by (*simp add*: *compatible-register-invariant-SUP*)
then have ‹*compatible-register-invariant F* (− (*SUP x*∈*X*. − *I x*))›
by (*simp add*: *compatible-register-invariant-compl*)
then show ‹*compatible-register-invariant F* (*INF x*∈*X*. *I x*)›
by (*metis Extra-General.uminus-INF ortho-involution*)
qed

lemma *compatible-register-invariant-Sup*:
assumes ‹*register F*›
assumes ‹$\bigwedge$*I*. *I*∈*II* ⟹ *compatible-register-invariant F I*›
shows ‹*compatible-register-invariant F* (*Sup II*)›
using *compatible-register-invariant-SUP*[**where** *X*=*II* **and** *I*=*id* **and** *F*=*F*] *assms* **by** *simp*

lemma *compatible-register-invariant-Inf*:
assumes ‹*register F*›
assumes ‹$\bigwedge$*I*. *I*∈*II* ⟹ *compatible-register-invariant F I*›
shows ‹*compatible-register-invariant F* (*Inf II*)›
using *compatible-register-invariant-INF*[**where** *X*=*II* **and** *I*=*id* **and** *F*=*F*] *assms* **by** *simp*

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* ⊓ *J*)›
using *compatible-register-invariant-Inf*[**where** *II*=‹{*I*,*J*}›]
using *assms* **by** *auto*

lemma *compatible-register-invariant-pair*:
assumes ‹*compatible-register-invariant F I*›
assumes ‹*compatible-register-invariant G I*›
shows ‹*compatible-register-invariant* (*F*;*G*) *I*›
proof (*cases* ‹*compatible F G*›)
case *True*
note *this*[*simp*]

have ∗: ‹*Proj I* o_{CL} (*F*;*G*) (*a* $\otimes_o$ *b*) = (*F*;*G*) (*a* $\otimes_o$ *b*) o_{CL} *Proj I*› **for** *a b*
using *assms*
apply (*simp add*: *register-pair-apply compatible-register-invariant-def*)
by (*metis cblinfun-compose-assoc*)
have ‹*Proj I* o_{CL} (*F*;*G*) *A* = (*F*;*G*) *A* o_{CL} *Proj I*› **for** *A*
apply (*rule tensor-extensionality*[*THEN fun-cong*[**where** *x*=*A*]])
by (*auto intro*!: *comp-preregister*[*unfolded comp-def*, *OF* - *preregister-mult-left*]
comp-preregister[*unfolded comp-def*, *OF* - *preregister-mult-right*] ∗)
then show *?thesis*
using *assms* **by** (*auto simp*: *compatible-register-invariant-def*)
next
case *False*
then show *?thesis*

using [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
by (*auto simp*: *compatible-register-invariant-def register-pair-def compatible-def*)
qed

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* $(F \otimes_r G)$ $(I \otimes_S J)$›
proof −
have [*iff*]: ‹*preregister* $(\lambda ab.$ *Proj* $(I \otimes_S J)$ o_{CL} $(F \otimes_r G)$ $ab)$›
by (*auto intro*!: *comp-preregister*[*unfolded comp-def*, *OF - preregister-mult-left*])
have [*iff*]: ‹*preregister* $(\lambda ab.$ $(F \otimes_r G)$ ab o_{CL} *Proj* $(I \otimes_S J))$›
by (*auto intro*!: *comp-preregister*[*unfolded comp-def*, *OF - preregister-mult-right*])
have *IF*: ‹*Proj I* o_{CL} $F\ a = F\ a$ o_{CL} *Proj I*› **for** *a*
using *assms*(*3*) *compatible-register-invariant-def* **by** *blast*
have *JG*: ‹*Proj J* o_{CL} $G\ b = G\ b$ o_{CL} *Proj J*› **for** *b*
using *assms*(*4*) *compatible-register-invariant-def* **by** *blast*
have ‹*Proj* $(I \otimes_S J)$ o_{CL} $(F \otimes_r G)$ $(a \otimes_o b) = (F \otimes_r G)$ $(a \otimes_o b)$ o_{CL} *Proj* $(I \otimes_S J)$› **for** *a b*
by (*simp add*: *tensor-ccsubspace-via-Proj Proj-on-own-range is-Proj-tensor-op comp-tensor-op IF JG*)
then have ‹$(\lambda ab.$ *Proj* $(I \otimes_S J)$ o_{CL} $(F \otimes_r G)$ $ab) = (\lambda ab.$ $(F \otimes_r G)$ ab o_{CL} *Proj* $(I \otimes_S J))$›
apply (*rule-tac tensor-extensionality*)
by *auto*
then show *?thesis*
unfolding *compatible-register-invariant-def*
by *meson*
qed

lemma *compatible-register-invariant-image-shrinks*:
assumes ‹*compatible-register-invariant F I*›
shows ‹$F\ U *_S I \leq I$›
proof −
have ‹$F\ U *_S I = (F\ U$ o_{CL} *Proj* $I) *_S \top$›
by (*simp add*: *cblinfun-compose-image*)
also have ‹$\ldots = ($*Proj* I o_{CL} $F\ U) *_S \top$›
by (*metis assms compatible-register-invariant-def*)
also have ‹$\ldots \leq$ *Proj* $I *_S \top$›
by (*simp add*: *Proj-image-leq cblinfun-compose-image*)
also have ‹$\ldots = I$›
by *simp*
finally show *?thesis*
by −
qed

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

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* ⇒ *'y update* ⇒ *'m update*› — register on which the unitary operates
    and K :: ‹*'z* ⇒ *'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* $(U1\ z)\ (I1\ z)\ (J1\ z)\ \varepsilon$›
  assumes *I-leq*: ‹$I \le (SUP\ z{\in}M.\ K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (I1\ z))$›
  assumes *J-geq*: ‹$\bigwedge z.\ z{\in}M \Longrightarrow J \ge K\ z \sqcap$ *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*: ‹$\bigwedge z\ z'.\ z{\in}M \Longrightarrow z'{\in}M \Longrightarrow z \ne z' \Longrightarrow$ *orthogonal-spaces* $(K\ z)\ (K\ z')$›
  assumes ‹$\varepsilon \ge 0$›
  assumes [*iff*]: ‹*finite* M›
  shows ‹*preserves* $U\ I\ J\ \varepsilon$›
proof −
  show *?thesis*
  proof (*rule invariant-splitting*[**where** S=‹K› **and** S'=‹K› **and** I=‹$\lambda z.\ K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (I1$
$z)$›
        and J=‹$\lambda z.\ K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (J1\ z)$› **and** X=M])
    from *orthoK*
    show ‹*orthogonal-spaces* $(K\ z)\ (K\ z')$› **if** ‹$z{\in}M$› ‹$z'{\in}M$› ‹$z \ne z'$› **for** $z\ z'$
      using *that* **by** *simp*
    then show ‹*orthogonal-spaces* $(K\ z)\ (K\ z')$› **if** ‹$z{\in}M$› ‹$z'{\in}M$› ‹$z \ne z'$› **for** $z\ z'$
      using *that* **by** −
    show ‹$K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (I1\ z) \le K\ z$› **for** z
      by *auto*
    show ‹$K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (J1\ z) \le K\ z$› **for** z
      by *auto*
    show ‹$U *_S K\ z \le K\ z$› **if** ‹$z{\in}M$› **for** z
    proof −
      from *U1-U*[*OF that*]
      have ‹$U *_S K\ z = (Y\ z)\ (U1\ z) *_S K\ z$›
        apply (*rule-tac space-as-set-inject*[*THEN iffD1*])
        by (*simp add*: *cblinfun-image.rep-eq*)
      also from *YK*[*OF that*] **have** ‹$\ldots \le K\ z$›
        by (*simp add*: *compatible-register-invariant-image-shrinks*)
      finally show *?thesis*
        by −
    qed
    from *I-leq*
    show ‹$I \le (\sum z{\in}M.\ K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (I1\ z))$›
      apply (*subst sum-eq-SUP-ccsubspace*)
      by *auto*
    from *J-geq*
    show ‹$(\sum z{\in}M.\ K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (J1\ z)) \le J$›
      apply (*subst sum-eq-SUP-ccsubspace*)
      by (*auto simp*: *SUP-le-iff*)
    from *assms* **show** ‹$0 \le \varepsilon$›
      by −
    show ‹*preserves* $U\ (K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (I1\ z))$
        $(K\ z \sqcap$ *lift-invariant* $(Y\ z)\ (J1\ z))\ \varepsilon$› **if** ‹$z{\in}M$› **for** z
    proof −
      show *?thesis*

proof (*rule preserves-register*[**where** U'=‹*U1 z*› **and** I'=‹*I1 z*› **and** J'=‹*J1 z*› **and** F=‹*Y z*› **and** K=‹*K z*›])
show ‹*preserves* (*U1 z*) (*I1 z*) (*J1 z*) ε›
by (*simp add*: *pres-I1*[*OF that*])
show ‹*register* (*Y z*)›
using *regY*[*OF that*] **by** −
from *YK*[*OF that*] **show** ‹*compatible-register-invariant* (*Y z*) (*K z*)›
by −
from *U1-U*[*OF that*]
show ‹$\forall\psi\in$*space-as-set* (*K z* ⊓ *lift-invariant* (*Y z*) (*I1 z*)). (*Y z*) (*U1 z*) $*_V$ ψ = *U* $*_V$ ψ›
by *auto*
show ‹*K z* ⊓ *lift-invariant* (*Y z*) (*I1 z*) ≤ *lift-invariant* (*Y z*) (*I1 z*)›
by *auto*
show ‹*K z* ⊓ *lift-invariant* (*Y z*) (*I1 z*) ≤ *K z*›
by *simp*
show ‹*lift-invariant* (*Y z*) (*J1 z*) ⊓ *K z* ≤ *K z* ⊓ *lift-invariant* (*Y z*) (*J1 z*)›
using [[*simp-trace*]]
by *simp*
qed
qed
show ‹*finite M*›
by *simp*
qed
qed

lemma *Proj-ket-invariant-ket*: ‹*Proj* (*ket-invariant X*) $*_V$ *ket i* = (*if i*∈*X then ket i else 0*)›
proof (*cases* ‹*i*∈*X*›)
case *True*
then have ‹*ket i* ∈ *space-as-set* (*ket-invariant X*)›
by (*simp add*: *ccspan-superset′ ket-invariant-def*)
then have ‹*Proj* (*ket-invariant X*) $*_V$ *ket i* = *ket i*›
by (*rule Proj-fixes-image*)
also have ‹*ket i* = (*if i*∈*X then ket i else 0*)›
using *True* **by** *simp*
finally show *?thesis*
by −
next
case *False*
then have ∗: ‹*ket i* ∈ *space-as-set* (*ket-invariant* (−*X*))›
by (*simp add*: *ccspan-superset′ ket-invariant-def*)
have ‹*Proj* (*ket-invariant X*) $*_V$ *ket i* = (*id-cblinfun* − *Proj* (*ket-invariant* (−*X*))) $*_V$ *ket i*›
by (*simp add*: *Proj-ortho-compl ket-invariant-compl*)
also have ‹. . . = *ket i* − *Proj* (*ket-invariant* (−*X*)) $*_V$ *ket i*›
by (*simp add*: *minus-cblinfun.rep-eq*)
also from ∗ **have** ‹. . . = *ket i* − *ket i*›
by (*simp add*: *Proj-fixes-image*)
also have ‹. . . = (*if i*∈*X then ket i else 0*)›
using *False* **by** *simp*
finally show *?thesis*
by −
qed

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

proof −
  have ‹*Proj* (*lift-invariant* (*function-at x*) (*ket-invariant I*)) = *Proj* (*ket-invariant* {*f*. *f x* ∈ *I*})›
  proof (*rule equal-ket*)
    fix *f* :: ‹′*a* ⇒ ′*b*›
    have ‹*Proj* (*lift-invariant* (*function-at x*) (*ket-invariant I*)) (*ket f*) = *function-at x* (*Proj* (*ket-invariant I*)) (*ket f*)›
      by (*simp add*: *Proj-on-own-range lift-invariant-def register-projector*)
    also have ‹. . . = *function-at-U x* $*_V$ *Fst* (*Proj* (*ket-invariant I*)) $*_V$ (*function-at-U x*)∗ $*_V$ *ket f*›
      by (*simp add*: *function-at-def sandwich-apply comp-def*)
    also have ‹. . . = *function-at-U x* $*_V$ *Fst* (*Proj* (*ket-invariant I*)) $*_V$ *ket* (*f x*, *snd* (*puncture-function x f*))›
      by (*simp flip*: *puncture-function-split*)
    also have ‹. . . = (*if f x* ∈ *I then function-at-U x* $*_V$ (*ket* (*f x*) $\otimes_s$ *ket* (*snd* (*puncture-function x f*))) *else 0*)›
      by (*auto simp*: *Fst-def tensor-op-ell2 Proj-ket-invariant-ket simp flip*: *tensor-ell2-ket*)
    also have ‹. . . = (*if f x* ∈ *I then ket* (*fix-punctured-function x* (*f x*, *snd* (*puncture-function x f*))) *else 0*)›
      by (*simp add*: *tensor-ell2-ket*)
    also have ‹. . . = (*if f x* ∈ *I then ket f else 0*)›
      by (*simp flip*: *puncture-function-split*)
    also have ‹. . . = *Proj* (*ket-invariant* {*f*. *f x* ∈ *I*}) $*_V$ *ket f*›
      by (*simp add*: *Proj-ket-invariant-ket*)
    finally show ‹*Proj* (*lift-invariant* (*function-at x*) (*ket-invariant I*)) $*_V$ *ket f* = *Proj* (*ket-invariant* {*f*. *f x* ∈ *I*}) $*_V$ *ket f*›
      by −
  qed
  then show *?thesis*
    by (*rule Proj-inj*)
qed

lemma *ket-invariant-prod*: ‹*Proj* (*ket-invariant* (*A* × *B*)) = *Proj* (*ket-invariant A*) $\otimes_o$ *Proj* (*ket-invariant B*)›
  apply (*rule equal-ket*)
  by (*auto simp*: *Proj-ket-invariant-ket tensor-op-ell2 simp flip*: *tensor-ell2-ket*
      split: *if-split-asm*)

lemma *lift-Fst-inv*: ‹*lift-invariant Fst I* = *I* $\otimes_S$ ⊤›
  apply (*rule Proj-inj*)
  by (*simp add*: *lift-invariant-def Proj-on-own-range register-projector Fst-def tensor-ccsubspace-via-Proj*)
lemma *lift-Snd-inv*: ‹*lift-invariant Snd I* = ⊤ $\otimes_S$ *I*›
  apply (*rule Proj-inj*)
  by (*simp add*: *lift-invariant-def Proj-on-own-range register-projector Snd-def tensor-ccsubspace-via-Proj*)

lemma *lift-Snd-ket-inv*: ‹*lift-invariant Snd* (*ket-invariant I*) = *ket-invariant* (*UNIV* × *I*)›
  apply (*rule Proj-inj*)
  apply (*simp add*: *lift-invariant-def Proj-on-own-range register-projector ket-invariant-prod*)
  by (*simp add*: *Snd-def*)
lemma *lift-Fst-ket-inv*: ‹*lift-invariant Fst* (*ket-invariant I*) = *ket-invariant* (*I* × *UNIV*)›
  apply (*rule Proj-inj*)
  apply (*simp add*: *lift-invariant-def Proj-on-own-range register-projector ket-invariant-prod*)
  by (*simp add*: *Fst-def*)

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

lift-invariant F (*ket-invariant I*) ⊓ *lift-invariant G* (*ket-invariant J*)›
by (*simp add*: *compatible-proj-intersect lift-invariant-def register-pair-apply ket-invariant-prod*)

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*)›
by (*simp add*: *lift-invariant-def ket-invariant-prod tensor-ccsubspace-image*)

lemma *lift-invariant-sup*:
fixes *F* :: ‹($'a$ *ell2* $\Rightarrow_{CL}$ $'a$ *ell2*) ⇒ ($'b$ *ell2* $\Rightarrow_{CL}$ $'b$ *ell2*)›
assumes [*simp*]: ‹*register F*›
shows ‹*lift-invariant F* ($I \sqcup J$) = *lift-invariant F I* ⊔ *lift-invariant F J*›
proof −
from *register-decomposition*[*OF* ‹*register F*›]
have ‹*let* $'c$::*type* = *register-decomposition-basis F in ?thesis*›
proof *with-type-mp*
case *with-type-mp*
then obtain *U* :: ‹($'a \times 'c$) *ell2* $\Rightarrow_{CL}$ $'b$ *ell2*›
where ‹*unitary U*› **and** *FU*: ‹*F* ϑ = *sandwich U* $*_V$ ($\vartheta \otimes_o$ *id-cblinfun*)› **for** ϑ
by *auto*
have *lift-F*: ‹*lift-invariant F K* = *U* $*_S$ (*Proj* (*tensor-invariant K* ⊤)) $*_S$ ⊤› **for** *K*
using ‹*unitary U*›
by (*simp add*: *lift-invariant-def FU sandwich-apply cblinfun-compose-image tensor-invariant-via-Proj*)
show ‹*lift-invariant F* ($I \sqcup J$) = *lift-invariant F I* ⊔ *lift-invariant F J*›
by (*auto simp*: *lift-F tensor-invariant-sup-left*)
qed
from *this*[*cancel-with-type*]
show *?thesis*
by −
qed

lemma *lift-invariant-SUP*:
fixes *F* :: ‹($'a$ *ell2* $\Rightarrow_{CL}$ $'a$ *ell2*) ⇒ ($'b$ *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*))›
proof −
from *register-decomposition*[*OF* ‹*register F*›]
have ‹*let* $'d$::*type* = *register-decomposition-basis F in ?thesis*›
proof *with-type-mp*
case *with-type-mp*
then obtain *U* :: ‹($'a \times 'd$) *ell2* $\Rightarrow_{CL}$ $'b$ *ell2*›
where ‹*unitary U*› **and** *FU*: ‹*F* ϑ = *sandwich U* $*_V$ ($\vartheta \otimes_o$ *id-cblinfun*)› **for** ϑ
by *auto*
have *lift-F*: ‹*lift-invariant F K* = *U* $*_S$ (*Proj* (*tensor-invariant K* ⊤)) $*_S$ ⊤› **for** *K*
using ‹*unitary U*›
by (*simp add*: *lift-invariant-def FU sandwich-apply cblinfun-compose-image tensor-invariant-via-Proj*)
show ‹*lift-invariant F* (*SUP* $x \in X$. *I x*) = (*SUP* $x \in X$. *lift-invariant F* (*I x*))›
by (*auto simp*: *lift-F tensor-invariant-SUP-left cblinfun-image-SUP*)
qed
from *this*[*cancel-with-type*]
show *?thesis*
by −
qed

lemma *lift-invariant-compl*: ‹*lift-invariant R* (− *U*) = − *lift-invariant R U*› **if** ‹*register R*›
apply (*simp add*: *lift-invariant-def Proj-ortho-compl*)
by (*metis* (*no-types*, *lifting*) *Proj-is-Proj Proj-on-own-range Proj-ortho-compl Proj-range register-minus register-of-id*
register-projector that)

lemma *lift-invariant-INF*:
assumes ‹*register F*›
shows ‹*lift-invariant F* ($\bigsqcap$ *x*∈*A*. *I x*) = ($\bigsqcap$ *x*∈*A*. *lift-invariant F* (*I x*))›
using *lift-invariant-SUP*[*OF assms*, **where** *I*=‹λ*x*. − *I x*› **and** *X*=*A*]
by (*simp add*: *lift-invariant-compl assms flip*: *uminus-INF*)

lemma *lift-invariant-inf*:
assumes ‹*register F*›
shows ‹*lift-invariant F* (*I* ⊓ *J*) = *lift-invariant F I* ⊓ *lift-invariant F J*›
using *lift-invariant-INF*[**where** *A*=‹{*False*,*True*}› **and** *I*=‹λ*b*. *if b then J else I*›] *assms*
by *simp*

lemma *lift-invariant-mono*:
assumes ‹*register F*›
assumes ‹*I* ≤ *J*›
shows ‹*lift-invariant F I* ≤ *lift-invariant F J*›
by (*metis assms*(*1*,*2*) *inf*.*absorb-iff2 lift-invariant-inf*)

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

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

lemma *classical-operator-ket-invariant*:
assumes ‹*inj-map f*›
shows ‹*classical-operator f* $*_S$ *ket-invariant I* = *ket-invariant* (*Some* −‘ *f* ‘ *I*)›
proof −
have ‹*ccspan* ((λ*x*. *case f x of None* ⇒ *0* | *Some x* ⇒ *ket x*) ‘ *I*) = ($\bigsqcup$ *x*∈*I*. *ccspan* ((λ*x*. *case f x of None* ⇒ *0* | *Some x* ⇒ *ket x*) ‘ {*x*}))›
by (*auto intro*: *arg-cong*[**where** *f*=*ccspan*] *simp add*: *SUP-ccspan*)
also have ‹. . . = ($\bigsqcup$ *x*∈*I*. *ccspan* (*ket* ‘ *Some* −‘ *f* ‘ {*x*}))›
proof (*rule SUP-cong*[*OF refl*])
fix *x*
have [*simp*]: ‹*Some* −‘ {*None*} = {}›
by *fastforce*

have [*simp*]: ‹*Some* −‘ {*Some a*} = {*a*}› **for** *a*
by *fastforce*
show ‹*ccspan* ((λ*x*. *case f x of None* ⇒ *0* | *Some x* ⇒ *ket x*) ‘ {*x*}) = *ccspan* (*ket* ‘ *Some* −‘ *f* ‘ {*x*})›
apply (*cases* ‹*f x*›)
by *auto*
qed
also have ‹. . . = *ccspan* (*ket* ‘ *Some* −‘ *f* ‘ *I*)›
by (*auto intro*: *arg-cong*[**where** *f*=*ccspan*] *simp add*: *SUP-ccspan*)
finally show *?thesis*
by (*simp add*: *ket-invariant-def cblinfun-image-ccspan image-image classical-operator-ket assms classical-operator-exists-inj*)
qed

lemma *Proj-ket-invariant-singleton*: ‹*Proj* (*ket-invariant* {*x*}) = *selfbutter* (*ket x*)›
by (*simp add*: *ket-invariant-def butterfly-eq-proj*)

lemma *lift-inv-classical*:
fixes *F* :: ‹$'a$ *ell2* $\Rightarrow_{CL}$ $'a$ *ell2* ⇒ $'b$ *ell2* $\Rightarrow_{CL}$ $'b$ *ell2*› **and** *f* :: ‹$'a \times 'c \Rightarrow 'b$›
assumes [*register*]: ‹*register F*›
assumes ‹*inj f*›
assumes ‹⋀*x*::$'a$. *x* ∈ *I* ⟹ *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* × *UNIV*))›
proof −
have [*iff*]: ‹*isometry* (*classical-operator* (*Some* ∘ *f*))›
by (*auto intro*!: *isometry-classical-operator assms*)
have ‹*lift-invariant F* (*ket-invariant I*) = (*SUP x*∈*I*. *lift-invariant F* (*ket-invariant* {*x*}))›
by (*simp add*: *flip*: *lift-invariant-SUP ket-invariant-SUP*)
also have ‹. . . = (*SUP x*∈*I*. *F* (*selfbutter* (*ket x*)) $*_S$ ⊤)›
by (*simp add*: *lift-invariant-def Proj-ket-invariant-singleton*)
also have ‹. . . = (*SUP x*∈*I*. *sandwich* (*classical-operator* (*Some o f*)) (*selfbutter* (*ket x*) $\otimes_o$ *id-cblinfun*) $*_S$ ⊤)›
using *assms* **by** *force*
also have ‹. . . = (*SUP x*∈*I*. *sandwich* (*classical-operator* (*Some o f*)) (*Proj* (*ket-invariant* ({*x*} × *UNIV*))) $*_S$ ⊤)›
apply (*simp add*: *flip*: *ket-invariant-tensor*)
by (*metis* (*no-types*, *lifting*) *Proj-ket-invariant-singleton Proj-top ket-invariant-UNIV ket-invariant-prod ket-invariant-tensor*)
also have ‹. . . = (*SUP x*∈*I*. *Proj* (*classical-operator* (*Some o f*) $*_S$ *ket-invariant* ({*x*} × *UNIV*)) $*_S$ ⊤)›
using *Proj-sandwich* **by** *fastforce*
also have ‹. . . = (*SUP x*∈*I*. *classical-operator* (*Some o f*) $*_S$ *ket-invariant* ({*x*} × *UNIV*))›
by *auto*
also have ‹. . . = (*SUP x*∈*I*. *ket-invariant* (*f* ‘ ({*x*} × *UNIV*)))›
apply (*subst classical-operator-ket-invariant*)
apply (*simp add*: *assms*(*2*))
by (*simp add*: *inj-vimage-image-eq flip*: *image-image*)
also have ‹. . . = *ket-invariant* (⋃*x*∈*I*. *f* ‘ ({*x*} × *UNIV*))›
by (*simp add*: *ket-invariant-SUP*)
also have ‹. . . = *ket-invariant* (*f* ‘ (*I* × *UNIV*))›
by *auto*
finally show *?thesis*

by −
qed

lemma *register-image-lift-invariant*:
assumes ‹*register F*›
assumes ‹*isometry U*›
shows ‹$F\ U *_S$ *lift-invariant* $F\ I$ = *lift-invariant* $F\ (U *_S I)$›
proof −
have ‹$F\ U *_S$ *lift-invariant* $F\ I = F\ U *_S F$ (*Proj* I) $*_S \top$›
by (*simp add*: *lift-invariant-def*)
also have ‹$\ldots = F\ U *_S F$ (*Proj* I) $*_S (F\ U)* *_S \top$›
by (*simp add*: *assms*(*1*,*2*) *range-adjoint-isometry register-isometry*)
also have ‹$\ldots = F$ (*sandwich* U (*Proj* I)) $*_S \top$›
by (*smt* (*verit*, *best*) *Proj-lift-invariant Proj-range Proj-sandwich assms*(*1*,*2*) *range-adjoint-isometry register-isometry register-sandwich*)
also have ‹$\ldots = F$ (*Proj* $(U *_S I)$) $*_S \top$›
by (*simp add*: *Proj-sandwich assms*(*2*))
also have ‹$\ldots$ = *lift-invariant* $F\ (U *_S I)$›
by (*simp add*: *lift-invariant-def*)
finally show *?thesis*
by −
qed

lemma *ell2-sum-ket-ket-invariant*:
fixes ψ :: ‹*'a ell2*›
assumes ‹$\psi \in$ *space-as-set* (*ket-invariant* X)›
shows ‹$\psi = (\sum_\infty i{\in}X.$ *Rep-ell2* $\psi\ i *_C$ *ket* $i)$›
proof −
from *assms* **have** ‹ψ = *Proj* (*ket-invariant* X) $*_V \psi$›
by (*simp add*: *Proj-fixes-image*)
also have ‹$\ldots$ = *Proj* (*ket-invariant* X) $*_V (\sum_\infty i.$ *Rep-ell2* $\psi\ i *_C$ *ket* $i)$›
by (*simp flip*: *ell2-decompose-infsum*)
also have ‹$\ldots = (\sum_\infty i.$ *Rep-ell2* $\psi\ i *_C$ *Proj* (*ket-invariant* X) $*_V$ *ket* $i)$›
by (*simp flip*: *infsum-cblinfun-apply add*: *ell2-decompose-summable cblinfun.scaleC-right*)
also have ‹$\ldots = (\sum_\infty i.$ *Rep-ell2* $\psi\ i *_C$ (*if* $i{\in}X$ *then ket* i *else* 0))›
by (*simp add*: *Proj-ket-invariant-ket*)
also have ‹$\ldots = (\sum_\infty i{\in}X.$ *Rep-ell2* $\psi\ i *_C$ *ket* $i)$›
apply (*rule infsum-cong-neutral*)
by *auto*
finally show *?thesis*
by *simp*
qed

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* $((\lambda x.\ (x,y))$ −‘ I))›
shows ‹*compatible-register-invariant* (*Fst o F*) (*ket-invariant* I)›
apply (*subst asm-rl*[*of* ‹$I = (\bigcup y.\ ((\lambda x.\ (x,y))$ −‘ $I) \times \{y\})$›])
apply *fastforce*
apply (*simp add*: *ket-invariant-SUP*)
apply (*rule compatible-register-invariant-SUP*, *simp*)
apply (*simp add*: *compatible-register-invariant-def ket-invariant-prod Fst-def comp-tensor-op*)

by (*metis assms compatible-register-invariant-def*)

lemma *compatible-register-invariant-Fst*:
assumes ‹$\bigwedge y.$ $((\lambda x.\ (x,y)) -` I) = UNIV \vee ((\lambda x.\ (x,y)) -` I) = \{\}$›
shows ‹*compatible-register-invariant Fst* (*ket-invariant I*)›
apply (*subst asm-rl*[*of* ‹*Fst* = *Fst o id*›], *simp*)
apply (*rule compatible-register-invariant-Fst-comp*, *simp*)
using *assms* **by** (*rule compatible-register-invariant-id*)

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* $((\lambda y.\ (x,y)) -` I)$)›
shows ‹*compatible-register-invariant* (*Snd o F*) (*ket-invariant I*)›
apply (*subst asm-rl*[*of* ‹$I = (\bigcup x.\ \{x\} \times ((\lambda y.\ (x,y)) -` I))$›])
apply *fastforce*
apply (*simp add*: *ket-invariant-SUP*)
apply (*rule compatible-register-invariant-SUP*, *simp*)
apply (*simp add*: *compatible-register-invariant-def ket-invariant-prod Snd-def comp-tensor-op*)
by (*metis assms compatible-register-invariant-def*)

lemma *compatible-register-invariant-Snd*:
assumes ‹$\bigwedge x.$ $((\lambda y.\ (x,y)) -` I) = UNIV \vee ((\lambda y.\ (x,y)) -` I) = \{\}$›
shows ‹*compatible-register-invariant Snd* (*ket-invariant I*)›
apply (*subst asm-rl*[*of* ‹*Snd* = *Snd o id*›], *simp*)
apply (*rule compatible-register-invariant-Snd-comp*, *simp*)
using *assms* **by** (*rule compatible-register-invariant-id*)

lemma *compatible-register-invariant-Fst-tensor*[*simp*]:
shows ‹*compatible-register-invariant Fst* ($\top \otimes_S I$)›
by (*simp add*: *compatible-register-invariant-def Fst-def Proj-on-own-range comp-tensor-op is-Proj-tensor-op tensor-ccsubspace-via-Proj*)

lemma *compatible-register-invariant-Snd-tensor*[*simp*]:
shows ‹*compatible-register-invariant Snd* ($I \otimes_S \top$)›
by (*simp add*: *compatible-register-invariant-def Snd-def Proj-on-own-range comp-tensor-op is-Proj-tensor-op tensor-ccsubspace-via-Proj*)

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*›
apply (*subst asm-rl*[*of* ‹$I = U *_S U* *_S I$›])
apply (*simp add*: *cblinfun-assoc-left*(*2*))
using *assms*
by (*simp add*: *compatible-register-invariant-def unitary-sandwich-register register-mult flip*: *Proj-sandwich*[*of U*])

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

have ‹(λa. (a, snd (puncture-function x z))) −‘ Some −‘ inv-map (Some ∘ fix-punctured-function x) ‘ I
= (λa. (a, snd (puncture-function x z))) −‘ puncture-function x ‘ I› (**is** ‹?lhs = -›) **for** z
by (simp add: inv-map-total bij-fix-punctured-function bij-is-surj inj-vimage-image-eq
flip: image-image)
also have ‹... z = {f x | f. f∈I ∧ snd (puncture-function x z) = snd (puncture-function x f)}› **for** z
apply (transfer fixing: I x)
by auto
also have ‹... z = {f x | f. f∈I ∧ z(x:=undefined) = f(x:=undefined)}› **for** z
proof −
have aux: ‹f ∈ I ⟹
z(x := undefined) ∘ Transposition.transpose x undefined =
f(x := undefined) ∘ Transposition.transpose x undefined ⟹
∃fa. f x = fa x ∧ fa ∈ I ∧ z(x := undefined) = fa(x := undefined)› **for** f
by (metis swap-nilpotent)
show ?thesis
apply (transfer fixing: z x I)
using aux **by** (auto simp: fun-upd-comp-left)
qed
finally have ‹compatible-register-invariant F (ket-invariant ((λa. (a, snd (puncture-function x z))) −‘ Some −‘ inv-map (Some ∘ fix-punctured-function x) ‘ I))› **for** z
by (simp add: assms)
then have ∗: ‹compatible-register-invariant F (ket-invariant ((λa. (a, y)) −‘ Some −‘ inv-map (Some ∘ fix-punctured-function x) ‘ I))› **for** y
by (metis fix-punctured-function-inverse snd-conv)
show ?thesis
unfolding function-at-def function-at-U-def Let-def comp-assoc
apply (rule compatible-register-invariant-sandwich-comp)
apply (simp add: bij-fix-punctured-function)
apply (subst classical-operator-adjoint)
apply (simp add: bij-fix-punctured-function bij-is-inj)
apply (subst classical-operator-ket-invariant)
apply (simp add: bij-fix-punctured-function bij-is-inj)
apply (rule compatible-register-invariant-Fst-comp, simp)
using ∗ **by** simp
qed

lemma compatible-register-invariant-function-at:
assumes ‹⋀f y. f∈I ⟹ f(x:=y) ∈ I›
shows ‹compatible-register-invariant (function-at x) (ket-invariant I)›
apply (subst asm-rl[of ‹function-at x = function-at x o id›], simp)
apply (rule compatible-register-invariant-function-at-comp, simp)
apply (rule compatible-register-invariant-id)
using assms fun-upd-idem-iff **by** fastforce

The following lemma allows show that an invariant is preserved across several consecutive operations. Usually, *norm V* and *norm U* ≤ *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 ∗ ε + norm U ∗ δ)›
proof −
have ‹norm ((id-cblinfun − Proj K) o_{CL} (V o_{CL} U) o_{CL} Proj I)
= norm ((id-cblinfun − Proj K) o_{CL} V o_{CL} (Proj J + (id-cblinfun − Proj J)) o_{CL} U o_{CL} Proj I)›

by (*auto simp add*: *cblinfun-assoc-left*(*1*))
also have ‹. . . ≤ *norm* ((*id-cblinfun* − *Proj K*) o_{CL} *V* o_{CL} *Proj J* o_{CL} *U* o_{CL} *Proj I*)
+ *norm* ((*id-cblinfun* − *Proj K*) o_{CL} *V* o_{CL} (*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*)›
by (*smt* (*verit*) *cblinfun-compose-add-left cblinfun-compose-add-right norm-triangle-ineq*)
also have ‹. . . ≤ *norm* ((*id-cblinfun* − *Proj K*) o_{CL} *V* o_{CL} *Proj J* o_{CL} *U* o_{CL} *Proj I*) + *norm V* ∗ ε›
proof −
have ‹*norm* ((*id-cblinfun* − *Proj K*) o_{CL} *V* o_{CL} (*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*)
≤ *norm* (*id-cblinfun* − *Proj K*) ∗ *norm* (*V* o_{CL} (*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*)›
by (*metis cblinfun-assoc-left*(*1*) *norm-cblinfun-compose*)
also have ‹. . . ≤ *norm* (*V* o_{CL} (*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*)›
by (*metis Groups.mult-ac*(*2*) *Proj-ortho-compl mult.right-neutral mult-left-mono norm-Proj-leq1 norm-ge-zero*)
also have ‹. . . ≤ *norm V* ∗ *norm* ((*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I*)›
by (*metis cblinfun-assoc-left*(*1*) *norm-cblinfun-compose*)
also have ‹. . . ≤ *norm V* ∗ ε›
by (*meson norm-ge-zero ordered-comm-semiring-class.comm-mult-left-mono presU preserves-onorm*)
finally show *?thesis*
by (*rule add-left-mono*)
qed
also have ‹. . . ≤ *norm* ((*id-cblinfun* − *Proj K*) o_{CL} *V* o_{CL} *Proj J* o_{CL} *U*) ∗ *norm* (*Proj I*) + *norm V* ∗ ε›
by (*simp add*: *norm-cblinfun-compose*)
also have ‹. . . ≤ *norm* ((*id-cblinfun* − *Proj K*) o_{CL} *V* o_{CL} *Proj J* o_{CL} *U*) + *norm V* ∗ ε›
by (*simp add*: *norm-is-Proj mult.commute mult-left-le-one-le*)
also have ‹. . . ≤ *norm* ((*id-cblinfun* − *Proj K*) o_{CL} *V* o_{CL} *Proj J*) ∗ *norm U* + *norm V* ∗ ε›
by (*simp add*: *norm-cblinfun-compose*)
also have ‹. . . ≤ *norm U* ∗ δ + *norm V* ∗ ε›
by (*metis add.commute add-le-cancel-left mult.commute mult-left-mono norm-ge-zero presV preserves-onorm*)
finally show *?thesis*
by (*simp add*: *preserves-onorm*)
qed

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 ‹ε ≥ *0*›
shows ‹*preserves* (*F U*) *I I* ε›
proof (*rule preservesI′*)
from *assms* **show** ‹ε ≥ *0*› **by** −
fix ψ **assume** ‹ψ ∈ *space-as-set I*›
then have ψ*I*: ‹ψ = *Proj I* $*_V$ ψ›
using *Proj-fixes-image* **by** *force*
from *compat* **have** *FI*: ‹*F U* $*_V$ *Proj I* $*_V$ ψ = *Proj I* $*_V$ *F U* $*_V$ ψ›
by (*metis cblinfun-apply-cblinfun-compose compatible-register-invariant-def*)
have ‹*Proj* (− *I*) $*_V$ *F U* $*_V$ ψ = *0*›
apply (*subst* ψ*I*) **apply** (*subst FI*)
by (*metis FI Proj-ortho-compl* ψ*I cancel-comm-monoid-add-class.diff-cancel cblinfun.diff-left id-cblinfun-apply*)
with ‹ε ≥ *0*› **show** ‹*norm* (*Proj* (− *I*) $*_V$ *F U* $*_V$ ψ) ≤ ε›
by *simp*
qed

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

by (*simp add*: *butterfly-eq-proj ket-invariant-def*)

lemma *ket-in-ket-invariantI*: ‹*ket x* ∈ *space-as-set* (*ket-invariant I*)› **if** ‹*x* ∈ *I*›
by (*metis Proj-ket-invariant-ket Proj-range cblinfun-apply-in-image that*)

lemma *cblinfun-image-ket-invariant-leqI*:
assumes ‹$\bigwedge x.\ x \in I \Longrightarrow U *_V$ *ket x* ∈ *space-as-set J*›
shows ‹$U *_S$ *ket-invariant I* ≤ *J*›
by (*simp add*: *assms cblinfun-image-ccspan ccspan-leqI image-subset-iff ket-invariant-def*)

lemma *preserves0I*: ‹*preserves U I J 0* ⟷ $U *_S I \le J$›
proof
have ‹(*id-cblinfun* − *Proj J*) o_{CL} *U* o_{CL} *Proj I* = *0* ⟹ $U *_S I \le J$›
by (*metis* (*no-types*, *lifting*) *Proj-range add-diff-cancel-left′ cblinfun-assoc-left*(*2*) *cblinfun-compose-minus-left cblinfun-compose-id-left cblinfun-image-mono diff-add-cancel diff-zero top-greatest*)
then show ‹*preserves U I J 0* ⟹ $U *_S I \le J$›
by (*auto simp*: *preserves-onorm*)
next
assume *leq*: ‹$U *_S I \le J$›
show ‹*preserves U I J 0*›
proof (*rule preservesI*)
show ‹*0* ≤ (*0*::*real*)› **by** *simp*
fix ψ
assume ‹ψ ∈ *space-as-set I*›
with *leq* **have** ‹$U *_V \psi$ ∈ *space-as-set J*›
by (*metis* (*no-types*, *lifting*) *Proj-fixes-image Proj-range cblinfun-apply-cblinfun-compose cblinfun-apply-in-image cblinfun-compose-image less-eq-ccsubspace.rep-eq subset-iff*)
then have ‹*Proj J* $*_V$ $U *_V \psi = U *_V \psi$›
by (*simp add*: *Proj-fixes-image*)
then show ‹*norm* ($U *_V \psi$ − *Proj J* $*_V$ $U *_V \psi$) ≤ *0*›
by *simp*
qed
qed

lemma *lift-invariant-id*[*simp*]: ‹*lift-invariant id I* = *I*›
by (*simp add*: *lift-invariant-def*)

lemma *lift-invariant-pair-tensor*:
assumes ‹*compatible X Y*›
shows ‹*lift-invariant* (*X*;*Y*) ($I \otimes_S J$) = *lift-invariant X I* ⊓ *lift-invariant Y J*›
proof −
have ‹*lift-invariant* (*X*;*Y*) ($I \otimes_S J$) = (*X*;*Y*) (*Proj* ($I \otimes_S J$)) $*_S$ ⊤›
by (*simp add*: *lift-invariant-def*)
also have ‹. . . = (*X*;*Y*) (*Proj I* $\otimes_o$ *Proj J*) $*_S$ ⊤›
by (*simp add*: *Proj-on-own-range is-Proj-tensor-op tensor-ccsubspace-via-Proj*)
also have ‹. . . = (*X* (*Proj I*) o_{CL} *Y* (*Proj J*)) $*_S$ ⊤›
by (*simp add*: *Laws-Quantum.register-pair-apply assms*)
also have ‹. . . = *lift-invariant X I* ⊓ *lift-invariant Y J*›
by (*simp add*: *assms compatible-proj-intersect lift-invariant-def*)
finally show *?thesis*
by −
qed

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›
proof −
have ‹*lift-invariant* $(X \otimes_r Y)$ $(I \otimes_S J)$ = $(X \otimes_r Y)$ (*Proj* $(I \otimes_S J)$) $*_S$ ⊤›
by (*simp add*: *lift-invariant-def*)
also have ‹. . . = $(X \otimes_r Y)$ (*Proj* I $\otimes_o$ *Proj* J) $*_S$ ⊤›
by (*simp add*: *Proj-on-own-range is-Proj-tensor-op tensor-ccsubspace-via-Proj*)
also have ‹. . . = (X (*Proj* I) $\otimes_o$ Y (*Proj* J)) $*_S$ ⊤›
by (*simp add*: *Laws-Quantum.register-pair-apply assms register-tensor-apply*)
also have ‹. . . = *lift-invariant* X I $\otimes_S$ *lift-invariant* Y J›
by (*simp add*: *lift-invariant-def tensor-ccsubspace-image*)
finally show *?thesis*
by −
qed

lemma *orthogonal-spaces-lift-invariant*[*simp*]:
assumes ‹*register Q*›
shows ‹*orthogonal-spaces* (*lift-invariant Q S*) (*lift-invariant Q T*) ⟷ *orthogonal-spaces S T*›
proof −
have ‹*orthogonal-spaces* (*lift-invariant Q S*) (*lift-invariant Q T*) ⟷ Q (*Proj S*) o_{CL} Q (*Proj T*) = *0*›
by (*simp add*: *orthogonal-projectors-orthogonal-spaces lift-invariant-def Proj-on-own-range assms register-projector*)
also have ‹. . . ⟷ *Proj S* o_{CL} *Proj T* = *0*›
by (*metis* (*no-types*, *lifting*) *assms norm-eq-zero register-mult register-norm*)
also have ‹. . . ⟷ *orthogonal-spaces S T*›
by (*simp add*: *orthogonal-projectors-orthogonal-spaces*)
finally show *?thesis*
by −
qed

3.2 Distance from invariants

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

lemma *dist-inv-pos*[*iff*]: ‹*dist-inv R I* $\psi \geq 0$›
by (*simp add*: *dist-inv-def*)
lemma *dist-inv-avg-pos*[*iff*]: ‹*dist-inv-avg R I* $\psi \geq 0$›
by (*simp add*: *dist-inv-avg-def sum-nonneg*)

lemma *dist-inv-0-iff*:
assumes ‹*register R*›
shows ‹*dist-inv R I* ψ = *0* ⟷ $\psi \in$ *space-as-set* (*lift-invariant R I*)›
proof −
have ‹*dist-inv R I* ψ = *0* ⟷ R (*Proj* (− I)) $*_V$ ψ = *0*›
by (*simp add*: *dist-inv-def*)
also have ‹. . . ⟷ *Proj* (R (*Proj* (− I)) $*_S$ ⊤) ψ = *0*›
by (*simp add*: *Proj-on-own-range assms register-projector*)
also have ‹. . . ⟷ $\psi \in$ *space-as-set* (− (R (*Proj* (− I)) $*_S$ ⊤))›
using *Proj-0-compl kernel-memberI* **by** *fastforce*
also have ‹. . . ⟷ $\psi \in$ *space-as-set* (− *lift-invariant R* $(-I)$)›
by (*simp add*: *lift-invariant-def*)
also have ‹. . . ⟷ $\psi \in$ *space-as-set* (*lift-invariant R I*)›
by (*metis* (*no-types*, *lifting*) *Proj-lift-invariant Proj-ortho-compl Proj-range assms*

ortho-involution register-minus register-of-id)
finally show *?thesis*
by −
qed

lemma *dist-inv-avg-0-iff*:
assumes ‹*register R*›
shows ‹*dist-inv-avg R I* ψ = *0* ⟷ (∀ *h*. ψ *h* ∈ *space-as-set* (*lift-invariant R* (*I h*)))›
proof −
have ‹*dist-inv-avg R I* ψ = *0* ⟷ (∀ *h*. (*dist-inv R* (*I h*) (ψ *h*))^2 = *0*)›
by (*simp add*: *dist-inv-avg-def sum-nonneg-eq-0-iff*)
also have ‹. . . ⟷ (∀ *h*. ψ *h* ∈ *space-as-set* (*lift-invariant R* (*I h*)))›
by (*simp add*: *assms dist-inv-0-iff*)
finally show *?thesis*
by −
qed

lemma *dist-inv-mono*:
assumes ‹*I* ≤ *J*›
assumes [*register*]: ‹*register Q*›
shows ‹*dist-inv Q J* ψ ≤ *dist-inv Q I* ψ›
proof −
from *assms*
have *ProjJI*: ‹*Proj* (−*J*) = *Proj* (−*J*) o_{CL} *Proj* (−*I*)›
by (*simp add*: *Proj-o-Proj-subspace-left*)
have ‹*norm* (*Q* (*Proj* (− *J*) o_{CL} *Proj* (− *I*)) $*_V$ ψ) ≤ *norm* (*Q* (*Proj* (− *I*)) $*_V$ ψ)›
by (*metis Proj-is-Proj assms*(*2*) *is-Proj-reduces-norm register-mult′*
register-projector)
then show *?thesis*
by (*simp add*: *dist-inv-def flip*: *ProjJI*)
qed

lemma *dist-inv-avg-mono*:
assumes ‹⋀*h*. *I h* ≤ *J h*›
assumes [*register*]: ‹*register Q*›
shows ‹*dist-inv-avg Q J* ψ ≤ *dist-inv-avg Q I* ψ›
by (*auto intro*!: *sum-mono divide-right-mono dist-inv-mono assms*
simp: *dist-inv-avg-def*)

lemma *dist-inv-Fst-tensor*:
assumes ‹*norm* φ = *1*›
shows ‹*dist-inv* (*Fst o R*) *I* ($\psi \otimes_s \varphi$) = *dist-inv R I* ψ›
proof −
have ‹(*norm* (*Fst* (*R* (*Proj* (− *I*))) $*_V$ $\psi \otimes_s \varphi$))^2 = (*norm* (*R* (*Proj* (− *I*)) $*_V$ ψ))^2›
by (*simp add*: *Fst-def tensor-op-ell2 norm-tensor-ell2 assms*)
then show *?thesis*
by (*simp add*: *dist-inv-def*)
qed

lemma *dist-inv-avg-Fst-tensor*:
assumes ‹⋀*h*. *norm* (φ *h*) = *1*›
shows ‹*dist-inv-avg* (*Fst o R*) *I* (λ*h*. ψ *h* $\otimes_s$ φ *h*) = *dist-inv-avg R I* ψ›
by (*simp add*: *assms dist-inv-avg-def dist-inv-Fst-tensor*)

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*›
proof −
  from *assms*
  have ‹*lift-invariant Q* $(-I)$ = *lift-invariant R* $(-J)$›
    by (*simp add*: *lift-invariant-compl*)
  then have ‹*Proj* (*Q* (*Proj* $(-I)$) $*_S$ ⊤) = *Proj* (*R* (*Proj* $(-J)$) $*_S$ ⊤)›
    by (*simp add*: *lift-invariant-def*)
  then have ‹*R* (*Proj* (− *J*)) = *Q* (*Proj* (− *I*))›
    by (*metis Proj-lift-invariant assms lift-invariant-def*)
  with *assms*
  show *?thesis*
    by (*simp add*: *dist-inv-def*)
qed

lemma *dist-inv-avg-register-rewrite*:
  assumes ‹*register Q*› **and** ‹*register R*›
  assumes ‹⋀*h. lift-invariant Q* (*I h*) = *lift-invariant R* (*J h*)›
  shows ‹*dist-inv-avg Q I ψ = dist-inv-avg R J ψ*›
  using *assms* **by** (*auto intro*!: *dist-inv-register-rewrite sum.cong simp add*: *dist-inv-avg-def*)

lemma *distance-from-inv-avg0I*:
  ‹*dist-inv-avg Q I ψ = 0* ⟷ (∀ *h. dist-inv Q* (*I h*) (*ψ h*) = *0*)› **for** *h* :: ‹*'h::finite*› **and** *ψ* :: ‹*'h* ⇒ -›
  by (*simp add*: *dist-inv-avg-def sum-nonneg-eq-0-iff*)

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*›
proof −
  have ‹*norm* (*Q* (*Proj* (− *I*)) $*_V$ *R U* $*_V$ *ψ*) = *norm* (*Q* (*Proj* (− (*S U*∗ $*_S$ *I*))) $*_V$ *ψ*)›
  proof −
    have ‹*norm* (*Q* (*Proj* (− *I*)) $*_V$ *R U* $*_V$ *ψ*) = *norm* (*Q* (*S U*)∗ $*_V$ *Q* (*Proj* (− *I*)) $*_V$ *Q* (*S U*) $*_V$
ψ)›
      by (*metis assms*(*1*,*2*,*3*,*4*) *isometry-preserves-norm o-def register-unitary unitary-twosided-isometry*)
    also have ‹. . . = *norm* (*sandwich* (*Q* (*S U*)∗) (*Q* (*Proj* $(-I)$)) $*_V$ *ψ*)›
      by (*simp add*: *sandwich-apply*)
    also have ‹. . . = *norm* (*Q* (*sandwich* (*S U*∗) $*_V$ *Proj* (− *I*)) $*_V$ *ψ*)›
      by (*simp add*: *flip*: *register-sandwich register-adj*)
    also have ‹. . . = *norm* (*Q* (*Proj* (*S U*∗ $*_S$ − *I*)) $*_V$ *ψ*)›
      by (*simp add*: *Proj-sandwich register-coisometry*)
    also have ‹. . . = *norm* (*Q* (*Proj* (− (*S U*∗ $*_S$ *I*))) $*_V$ *ψ*)›
      by (*simp add*: *unitary-image-ortho-compl register-unitary*)
    finally show *?thesis*
      by −
  qed
  then show *?thesis*
    by (*simp add*: *dist-inv-def*)
qed

lemma *dist-inv-apply-iff*:
assumes [*register*]: ‹*register Q*›
assumes [*iff*]: ‹*unitary U*›
shows ‹*dist-inv Q I* ($Q\ U *_V \psi$) = *dist-inv Q* ($U* *_S I$) ψ›
apply (*subst dist-inv-apply*[**where** *S*=*id*])
by *auto*

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* (λh. *R* (*U h*) $*_V \psi$ *h*) = *dist-inv-avg Q* (λh. *S* (*U h*)$* *_S$ *I h*) ψ›
using *assms* **by** (*auto intro*!: *sum.cong simp*: *dist-inv-avg-def dist-inv-apply*[**where** *S*=*S*])

lemma *dist-inv-avg-apply-iff*:
assumes [*register*]: ‹*register Q*›
assumes [*iff*]: ‹$\bigwedge h$. *unitary* (*U h*)›
shows ‹*dist-inv-avg Q I* (λh. *Q* (*U h*) $*_V \psi$ *h*) = *dist-inv-avg Q* (λh. *U h*$* *_S$ *I h*) ψ›
by (*auto intro*!: *sum.cong dist-inv-apply-iff simp*: *dist-inv-avg-def*)

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›
proof −
have *inside*: ‹$\psi \in$ *space-as-set* (*lift-invariant Q I*)›
using *assms*(*2*,*3*) *dist-inv-0-iff* **by** *blast*
have ‹*norm* (*Q* (*Proj* (− ($J \sqcap I$))) $*_V \psi$) = *norm* (ψ − *Q* (*Proj* ($J \sqcap I$)) $*_V \psi$)›
by (*metis* (*no-types*, *lifting*) *Proj-ortho-compl assms*(*2*) *cblinfun.diff-left id-cblinfun.rep-eq register-minus*
register-of-id)
also have ‹... = *norm* (ψ − *Q* (*Proj* (*J*) o_{CL} *Proj* (*I*)) $*_V \psi$)›
by (*metis assms compatible-invariants-def compatible-invariants-inter*)
also have ‹... = *norm* (ψ − *Q* (*Proj* (*J*)) $*_V$ *Q* (*Proj* (*I*)) $*_V \psi$)›
by (*simp add*: *assms register-mult'*)
also have ‹... = *norm* (ψ − *Q* (*Proj* (*J*)) $*_V \psi$)›
by (*metis Proj-fixes-image Proj-lift-invariant assms inside*)
also have ‹... = *norm* (*Q* (*Proj* (− *J*)) $*_V \psi$)›
by (*simp add*: *Proj-ortho-compl assms cblinfun.diff-left register-minus*)
finally have ‹*norm* (*Q* (*Proj* (− ($J \sqcap I$))) $*_V \psi$) = *norm* (*Q* (*Proj* (− *J*)) $*_V \psi$)›
by −
then show *?thesis*
by (*simp add*: *dist-inv-def*)
qed

lemma *dist-inv-avg-intersect*:

assumes ‹$\bigwedge h$. *compatible-invariants* (*I h*) (*J h*)›
assumes ‹*register Q*›
assumes ‹*dist-inv-avg Q I* ψ = *0*›
shows ‹*dist-inv-avg Q* (λh. *J h* ⊓ *I h*) ψ = *dist-inv-avg Q J* ψ›
proof −
have ‹*dist-inv Q* (*I h*) (ψ *h*) = *0* › **for** *h*
using *assms*(*3*) *distance-from-inv-avg0I* **by** *blast*
then show *?thesis*
by (*auto intro*!: *sum.cong dist-inv-intersect-onesided assms simp*: *dist-inv-avg-def*)
qed

lemma *dist-inv-avg-const*: ‹*dist-inv-avg Q* (λ-. *I*) (λ-. ψ) = *dist-inv Q I* ψ›
by (*simp add*: *dist-inv-avg-def dist-inv-def*)

lemma *register-plus*:
assumes ‹*register Q*›
shows ‹*Q* (*a* + *b*) = *Q a* + *Q b*›
by (*simp add*: *assms clinear-register complex-vector.linear-add*)

lemma *compatible-invariants-uminus-left*[*simp*]: ‹*compatible-invariants* (−*I*) *J* ⟷ *compatible-invariants I J*›
by (*simp add*: *Proj-ortho-compl cblinfun-compose-minus-left cblinfun-compose-minus-right compatible-invariants-def*)

lemma *compatible-invariants-uminus-right*[*simp*]: ‹*compatible-invariants I* (−*J*) ⟷ *compatible-invariants I J*›
by (*simp add*: *Proj-ortho-compl cblinfun-compose-minus-left cblinfun-compose-minus-right compatible-invariants-def*)

lemma *compatible-invariants-sup*: ‹*Proj* (*A* ⊔ *B*) = *Proj A* + *Proj B* − *Proj A* o_{CL} *Proj B*› **if** ‹*compatible-invariants A B*›
apply (*rewrite at* ‹*A* ⊔ *B*› *to* ‹− (−*A* ⊓ −*B*)› *DEADID.rel-mono-strong*)
apply *simp*
apply (*subst Proj-ortho-compl*)
by (*simp add*: *that Proj-ortho-compl cblinfun-compose-minus-left cblinfun-compose-minus-right flip*: *compatible-invariants-inter*)

lemma *compatible-invariants-sym*: ‹*compatible-invariants S T* ⟷ *compatible-invariants T S*›
by (*metis compatible-invariants-def*)

lemma *compatible-invariants-refl*[*iff*]: ‹*compatible-invariants S S*›
by (*metis compatible-invariants-def*)

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* ⊓ *U*)›
by (*smt* (*verit*, *del-insts*) *assms*(*1*,*2*,*3*) *cblinfun-compose-assoc compatible-invariants-def compatible-invariants-inter*)

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* ⊔ *U*)›
apply (*rewrite at* ‹*T* ⊔ *U*› *to* ‹− (−*T* ⊓ −*U*)› *DEADID.rel-mono-strong*)
apply *simp*
by (*auto intro*!: *compatible-invariants-infI simp del*: *compl-inf*)

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* ⊓ (*T* ⊔ *U*) = (*S* ⊓ *T*) ⊔ (*S* ⊓ *U*)›
proof −
have [*iff*]: ‹*compatible-invariants* (*S* ⊓ *T*) (*S* ⊓ *U*)›
using *assms* **by** (*auto intro*!: *compatible-invariants-infI simp*: *compatible-invariants-sym*)
have ‹*Proj* (*S* ⊓ (*T* ⊔ *U*)) = *Proj* ((*S* ⊓ *T*) ⊔ (*S* ⊓ *U*))›
apply (*simp add*: *assms compatible-invariants-sup compatible-invariants-supI flip*: *compatible-invariants-inter*)
by (*metis* (*no-types*, *lifting*) *Proj-idempotent assms*(*2*) *cblinfun-compose-add-right cblinfun-compose-assoc cblinfun-compose-minus-right*
compatible-invariants-def)
then show *?thesis*
using *Proj-inj* **by** *blast*
qed

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* ⊔ *U*) ⊓ *S* = (*T* ⊓ *S*) ⊔ (*U* ⊓ *S*)›
by (*simp add*: *compatible-invariants-inf-sup-distrib1 inf-commute*)

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* ⊔ (*T* ⊓ *U*) = (*S* ⊔ *T*) ⊓ (*S* ⊔ *U*)›
by (*smt* (*verit*, *ccfv-SIG*) *Groups.add-ac*(*1*) *assms*(*1*,*2*,*3*) *compatible-invariants-def compatible-invariants-inf-sup-distrib*
compatible-invariants-supI inf-commute inf-sup-absorb plus-ccsubspace-def)

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* ⊓ *U*) ⊔ *S* = (*T* ⊔ *S*) ⊓ (*U* ⊔ *S*)›
by (*metis Groups.add-ac*(*2*) *assms*(*1*,*2*,*3*) *compatible-invariants-sup-inf-distrib1 plus-ccsubspace-def*)

lemma *is-orthogonal-Proj-orthogonal-spaces*:
assumes ‹*orthogonal-spaces S T*›

shows ‹*is-orthogonal* (*Proj* $S *_V \psi$) (*Proj* $T *_V \psi$)›
by (*metis Proj-range assms cblinfun-apply-in-image orthogonal-spaces-def*)

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* $\psi)^2$)›
proof −
define *PInJ PJnI PnInJ PnI PnJ PnIJ* **where** ‹*PInJ* = *Q* (*Proj* $(I \sqcap - J)$)›
and ‹*PJnI* = *Q* (*Proj* $(-I \sqcap J)$)› **and** ‹*PnInJ* = *Q* (*Proj* $(-I \sqcap -J)$)›
and ‹*PnI* = *Q* (*Proj* $(-I)$)› **and** ‹*PnJ* = *Q* (*Proj* $(-J)$)›
and ‹*PnIJ* = *Q* (*Proj* $(- (I \sqcap J))$)›

have *compat1*: ‹*compatible-invariants* $(I \sqcap - J)$ *J*›
by (*metis Proj-o-Proj-subspace-left Proj-o-Proj-subspace-right compatible-invariants-def compatible-invariants-uminus-right inf-le2*)
have *compat2*: ‹*compatible-invariants* $(I \sqcap - J)$ *I*›
by (*simp add*: *Proj-o-Proj-subspace-left Proj-o-Proj-subspace-right compatible-invariants-def*)

have *ortho1*: ‹*orthogonal-spaces* $(I \sqcap - J)$ $(- I \sqcap J)$›
by (*simp add*: *le-infI2 orthogonal-spaces-leq-compl*)
have *ortho2*: ‹*orthogonal-spaces* $(I \sqcap - J \sqcup - I \sqcap J)$ $(- I \sqcap - J)$›
by (*metis inf-le1 inf-le2 ortho-involution orthocomplemented-lattice-class.compl-sup orthogonal-spaces-leq-compl sup.mono*)
have *ortho3*: ‹*orthogonal-spaces* $(- I \sqcap J)$ $(- I \sqcap - J)$›
by (*simp add*: *le-infI2 orthogonal-spaces-leq-compl*)
have *ortho4*: ‹*orthogonal-spaces* $(I \sqcap - J)$ $(- I \sqcap - J)$›
by (*metis inf-sup-absorb le-infI2 ortho2 orthogonal-spaces-leq-compl*)
have *ortho5*: ‹*is-orthogonal* (*PInJ* ψ) (*PnInJ* ψ)›
using *ortho4* **by** (*auto intro*!: *is-orthogonal-Proj-orthogonal-spaces simp*: *PInJ-def PnInJ-def simp flip*: *Proj-lift-invariant*)
have *ortho6*: ‹*is-orthogonal* (*PJnI* ψ) (*PnInJ* ψ)›
using *ortho3* **by** (*auto intro*!: *is-orthogonal-Proj-orthogonal-spaces simp*: *PJnI-def PnInJ-def simp flip*: *Proj-lift-invariant*)
have *ortho7*: ‹*is-orthogonal* (*PInJ* ψ) (*PJnI* ψ)›
using *ortho1* **by** (*auto intro*!: *is-orthogonal-Proj-orthogonal-spaces simp*: *PJnI-def PInJ-def simp flip*: *Proj-lift-invariant*)

have *nI*: ‹$-I \sqcap J \sqcup -I \sqcap -J = -I$›
by (*simp flip*: *compatible-invariants-inf-sup-distrib1*)
then have *PnI-decomp*: ‹*PnI* = *PJnI* + *PnInJ*›
by (*simp add*: *PnI-def PJnI-def PnInJ-def register-inj′ ortho3*
flip: *register-plus Proj-sup*)

have *nJ*: ‹$I \sqcap -J \sqcup -I \sqcap -J = -J$›
by (*metis* (*no-types*, *lifting*) *assms*(*2*) *compatible-invariants-inf-sup-distrib1 compatible-invariants-refl compatible-invariants-sym*
compatible-invariants-uminus-left complemented-lattice-class.sup-compl-top inf-aci(*1*) *inf-top.comm-neutral*)
then have *PnJ-decomp*: ‹*PnJ* = *PInJ* + *PnInJ*›
by (*simp add*: *PnJ-def PInJ-def PnInJ-def register-inj′ ortho4*
flip: *register-plus Proj-sup*)

have ‹$I \sqcap - J \sqcup - I \sqcap J \sqcup - I \sqcap - J = - I \sqcup - J$›
by (*metis* (*no-types*, *lifting*) *Groups.add-ac*(*1*) *boolean-algebra-cancel.sup2 nI nJ plus-ccsubspace-def sup-inf-absorb*)

then have *PnIJ-decomp*: ‹$PnIJ = PInJ + PJnI + PnInJ$›
  by (*simp add*: *PnIJ-def PInJ-def PJnI-def PnInJ-def register-inj′ ortho1 ortho2*
    flip: *register-plus Proj-sup*)

have ‹$(\mathit{dist\text{-}inv}\ Q\ (I \sqcap J)\ \psi)^2 = (\mathit{norm}\ (\mathit{PnIJ}\ \psi))^2$›
  by (*simp add*: *PnIJ-def dist-inv-def*)
also have ‹$\ldots = (\mathit{norm}\ (\mathit{PInJ} *_V \psi))^2 + (\mathit{norm}\ (\mathit{PJnI} *_V \psi))^2 + (\mathit{norm}\ (\mathit{PnInJ} *_V \psi))^2$›
  by (*simp add*: *PnIJ-decomp cblinfun.add-left pythagorean-theorem cinner-add-left ortho5 ortho6 ortho7*)
also have ‹$\ldots \le ((\mathit{norm}\ (\mathit{PJnI} *_V \psi))^2 + (\mathit{norm}\ (\mathit{PnInJ} *_V \psi))^2) + ((\mathit{norm}\ (\mathit{PInJ} *_V \psi))^2 + (\mathit{norm}\ (\mathit{PnInJ} *_V \psi))^2)$›
  by *simp*
also have ‹$\ldots = (\mathit{norm}\ (\mathit{PnI}\ \psi))^2 + (\mathit{norm}\ (\mathit{PnJ}\ \psi))^2$›
  by (*simp add*: *ortho5 ortho6 PnI-decomp PnJ-decomp cblinfun.add-left pythagorean-theorem*)
also have ‹$\ldots \le (\mathit{dist\text{-}inv}\ Q\ I\ \psi)^2 + (\mathit{dist\text{-}inv}\ Q\ J\ \psi)^2$›
  apply (*rule add-mono*)
  using *assms*
  by (*simp-all add*: *PnI-def PnJ-def dist-inv-def*)

finally show *?thesis*
  using *real-le-rsqrt* **by** *presburger*
qed

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›
  using *register-minus*[*OF assms*, *of id-cblinfun*, *symmetric*]
  by (*simp add*: *preserves-onorm Proj-lift-invariant register-mult register-norm*)

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 ‹$\mathit{dist\text{-}inv}\ R\ I\ (U *_V \psi) \le \mathit{norm}\ U * \mathit{dist\text{-}inv}\ S\ J\ \psi + \gamma * \mathit{norm}\ \psi$›
proof −
  note [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
  define ψ*good* ψ*bad* **where** ‹$\psi good = S\ (\mathit{Proj}\ J) *_V \psi$› **and** ‹$\psi bad = S\ (\mathit{Proj}\ (-\ J)) *_V \psi$›
  define ψ' ψ'*good* ψ'*bad* **where** ‹$\psi' = U\ \psi good$› **and** ‹$\psi' good = R\ (\mathit{Proj}\ I)\ \psi'$› **and** ‹$\psi' bad = R\ (\mathit{Proj}\ (-I))\ \psi'$›
  from *pres* **have** ‹$\gamma \ge 0$›
    using *preserves-def* **by** *blast*
  have ψ*-decomp*: ‹$\psi = \psi good + \psi bad$›
    by (*simp add*: ψ*good-def* ψ*bad-def Proj-ortho-compl register-minus flip*: *cblinfun.add-left*)
  have ψ'*-decomp*: ‹$\psi' = \psi' good + \psi' bad$›
    by (*simp add*: ψ'*good-def* ψ'*bad-def Proj-ortho-compl register-minus flip*: *cblinfun.add-left*)
  define δ **where** ‹$\delta = \mathit{dist\text{-}inv}\ S\ J\ \psi$›
  then have ψ*bad-bound*: ‹$\mathit{norm}\ \psi bad \le \delta$›
    unfolding *dist-inv-def* ψ*bad-def* **by** *blast*
  have ‹$\psi good \in$ *space-as-set* (*lift-invariant S J*)›
    by (*simp add*: ψ*good-def lift-invariant-def*)
  with *pres* **have** ‹$\mathit{norm}\ (\psi' - \mathit{Proj}\ (\mathit{lift\text{-}invariant}\ R\ I) *_V \psi') \le \gamma * \mathit{norm}\ \psi good$›
    by (*simp add*: *preserves-def* ψ'*-def*)
  then have ‹$\mathit{norm}\ \psi' bad \le \gamma * \mathit{norm}\ \psi good$›
    by (*simp add*: ψ'*bad-def Proj-ortho-compl register-minus cblinfun.diff-left Proj-lift-invariant*)

also have ‹$\gamma * norm\ \psi good \leq \gamma * norm\ \psi$›
  by (*auto intro*!: *mult-left-mono is-Proj-reduces-norm* ‹$\gamma \geq 0$› *intro*: *register-projector*
    simp add: *ψgood-def*)
  finally have *ψ′bad-bound*: ‹$norm\ \psi' bad \leq \gamma * norm\ \psi$›
    by *meson*
  have *Uψ-decomp*: ‹$U\ \psi = \psi' good + \psi' bad + U\ \psi bad$›
    by (*simp add*: *ψ-decomp ψ′-decomp cblinfun.add-right flip*: *ψ′-def*)
  have *mIψ′good0*: ‹$R\ (Proj\ (-\ I))\ \psi' good = 0$›
    by (*metis Proj-fixes-image Proj-lift-invariant ψ′-decomp ψ′-def ψ′bad-def add-diff-cancel-right′ assms*
      cancel-comm-monoid-add-class.diff-cancel cblinfun.diff-right cblinfun-apply-in-image lift-invariant-def)
  have *mIψ′bad*: ‹$norm\ (R\ (Proj\ (-\ I))\ \psi' bad) \leq \gamma * norm\ \psi$›
    by (*metis ψ′bad-bound ψ′-decomp ψ′bad-def add-diff-cancel-left′ cblinfun.diff-right diff-zero*
      mIψ′good0)
  from *ψbad-bound*
  have ‹$norm\ (U\ \psi bad) \leq norm\ U * \delta$›
    apply (*rule-tac order-trans*[*OF norm-cblinfun*[*of U ψbad*]])
    by (*simp add*: *mult-left-mono*)
  then have ‹$norm\ (R\ (Proj\ (-\ I)) *_V\ U\ \psi bad) \leq norm\ U * \delta$›
    apply (*rule-tac order-trans*[*OF norm-cblinfun*])
    apply (*subgoal-tac* ‹$norm\ (R\ (Proj\ (-\ I))) \leq 1$›)
     apply (*smt* (*verit*, *best*) *mult-left-le-one-le norm-ge-zero*)
    by (*simp add*: *norm-Proj-leq1 register-norm*)
  with *mIψ′bad* **have** ‹$dist\text{-}inv\ R\ I\ (U *_V\ \psi) \leq norm\ U * \delta + \gamma * norm\ \psi$›
    apply (*simp add*: *dist-inv-def Uψ-decomp cblinfun.add-right mIψ′good0*)
    by (*smt* (*verit*, *del-insts*) *norm-triangle-ineq*)
  then show *?thesis*
    by (*simp add*: *δ-def*)
qed

lemma *dist-inv-preservesI*:
  assumes ‹$dist\text{-}inv\ S\ J\ \psi \leq \varepsilon$›
  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 ‹$dist\text{-}inv\ R\ I\ (U *_V\ \psi) \leq \delta$›
proof −
  have ‹$\gamma \geq 0$›
    using *pres preserves-def* **by** *blast*
  with *assms* **have** ‹$norm\ U * dist\text{-}inv\ S\ J\ \psi + \gamma * norm\ \psi \leq \delta$›
    by (*smt* (*verit*, *ccfv-SIG*) *dist-inv-def mult-left-le mult-left-le-one-le norm-ge-zero*)
  then show *?thesis*
    apply (*rule order-trans*[*rotated*])
    by (*rule dist-inv-leq-if-preserves*[*OF pres* ‹*register S*› ‹*register R*›])
qed

lemma *dist-inv-apply-compatible*:
  assumes ‹*compatible Q R*›
  shows ‹$dist\text{-}inv\ Q\ I\ (R\ U *_V\ \psi) \leq norm\ U * dist\text{-}inv\ Q\ I\ \psi$›
proof −
  have [*register*]: ‹*register Q*›
    using *assms compatible-register1* **by** *blast*
  have [*register*]: ‹*register R*›

using *assms compatible-register2* **by** *blast*
have ‹*preserves* (*R U*) (*lift-invariant Q I*) (*lift-invariant Q I*) *0*›
apply (*rule preserves-compatible*[*of R*])
by (*simp-all add*: *assms compatible-register-invariant-compatible-register compatible-sym*)
then have ‹*dist-inv Q I* (*R U* $*_V$ ψ) $\leq$ *norm* (*R U*) $*$ *dist-inv Q I* ψ + *0* $*$ *norm* ψ›
apply (*rule dist-inv-leq-if-preserves*)
by *simp-all*
also have ‹. . . $\leq$ *norm U* $*$ *dist-inv Q I* ψ›
by (*simp add*: *register-norm*)
finally show *?thesis*
by −
qed

lemma *dist-inv-avg-apply-compatible*:
assumes ‹$\bigwedge h$. *compatible Q* (*R h*)›
shows ‹*dist-inv-avg Q I* (λh. *R h* (*U h*) $*_V$ ψ *h*) $\leq$ (*MAX h*. *norm* (*U h*)) $*$ *dist-inv-avg Q I* ψ›
proof −
have [*iff*]: ‹(*MAX h*$\in$*UNIV*. *norm* (*U h*)) $\geq$ *0*›
by (*simp add*: *Max-ge-iff*)
have ‹*dist-inv-avg Q I* (λh. *R h* (*U h*) $*_V$ ψ *h*)
= *sqrt* (($\sum$ *h*$\in$*UNIV*. (*dist-inv Q* (*I h*) (*R h* (*U h*) $*_V$ ψ *h*))^2) / *real CARD*('*a*))›
by (*simp add*: *dist-inv-avg-def*)
also have ‹. . . $\leq$ *sqrt* (($\sum$ *h*$\in$*UNIV*. (*norm* (*U h*) $*$ *dist-inv Q* (*I h*) (ψ *h*))^2) / *real CARD*('*a*))›
by (*auto intro*!: *divide-right-mono sum-mono dist-inv-apply-compatible assms*)
also have ‹. . . $\leq$ *sqrt* (($\sum$ *h*$\in$*UNIV*. ((*MAX h*. *norm* (*U h*)) $*$ *dist-inv Q* (*I h*) (ψ *h*))^2) / *real CARD*('*a*))›
by (*auto intro*!: *divide-right-mono power-mono sum-mono mult-right-mono*)
also have ‹. . . = (*MAX h*. *norm* (*U h*)) $*$ *sqrt* (($\sum$ *h*$\in$*UNIV*. (*dist-inv Q* (*I h*) (ψ *h*))^2) / *real CARD*('*a*))›
by (*simp add*: *power-mult-distrib real-sqrt-mult real-sqrt-abs abs-of-nonneg flip*: *sum-distrib-left times-divide-eq-right*)
also have ‹. . . = (*MAX h*. *norm* (*U h*)) $*$ *dist-inv-avg Q I* ψ›
by (*simp add*: *dist-inv-avg-def*)
finally show *?thesis*
by −
qed

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)›
unfolding *function-oracle-def*
apply (*subst classical-operator-ket*)
by (*auto intro*!: *classical-operator-exists-inj injI simp*: *inj-map-total*[*unfolded o-def*] *case-prod-unfold*)

lemma *function-oracle-adj-apply*: ‹*function-oracle h*∗ $*_V$ *ket* (x, y) = *ket* (x, y − h x)›
proof −
define *f* **where** ‹f = (λ(x,y). (x, y + h x))›
define *g* **where** ‹g = (λ(x,y). (x, y − h x))›
have *gf*: ‹g ∘ f = id› **and** *fg*: ‹f ∘ g = id›
by (*auto simp*: *f-def g-def*)
have [*iff*]: ‹*inj f*›
by (*metis fg gf injI isomorphism-expand*)
have ‹*inv f* = g›
using *fg gf inv-unique-comp* **by** *blast*
have *inv-map-f*: ‹*inv-map* (*Some o f*) = (*Some o g*)›
by (*metis* ‹*inj f*› ‹*inv f* = g› *fg fun.set-map inj-imp-surj-inv inv-map-total surj-id*)
have ‹*function-oracle h*∗ = *classical-operator* (*Some o f*)∗›
by (*simp add*: *function-oracle-def f-def case-prod-unfold o-def*)
also have ‹. . . = *classical-operator* (*Some o g*)›
using *inv-map-f* **by** (*simp add*: *classical-operator-adjoint function-oracle-def*)
also have ‹. . . $*_V$ *ket* (x,y) = *ket* (x, y − h x)›
apply (*subst classical-operator-ket*)
apply (*metis classical-operator-exists-inj inj-map-inv-map inv-map-f*)
by (*simp add*: *g-def*)
finally show *?thesis*
by −
qed

lemma *unitary-function-oracle*[*iff*]: ‹*unitary* (*function-oracle h*)›
proof −
have ‹*bij* (λx. (*fst* x, *snd* x + h (*fst* x)))›
apply (*rule o-bij*[**where** g=‹(λx. (*fst* x, *snd* x − h (*fst* x)))›])
by *auto*
then show *?thesis*
by (*auto intro*!: *unitary-classical-operator*[*unfolded o-def*]
simp add: *function-oracle-def case-prod-unfold*)
qed

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

lemma *function-oracle-adj*[*simp*]: ‹*function-oracle h*∗ = *function-oracle* (λx. − h x)› **for** h :: ‹'x ⇒ 'y::ab-group-add›
apply (*rule equal-ket*)
by (*auto simp*: *function-oracle-apply function-oracle-adj-apply*)

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*›
proof −
  wlog ‹*CARD*('a) ≥ *2*›
  proof −
    have ‹*CARD*('a) ≠ *0*›
      by *simp*
    with *negation* **have** ‹*CARD*('a) = *1*›
      by *presburger*
    then have [*simp*]: ‹*UNIV* = {*0*::'a}›
      by (*metis UNIV-I card-1-singletonE singletonD*)
    have ‹*uniform-superpos UNIV* = *ket* (*0*::'a)›
      by (*simp add*: *uniform-superpos-def2*)
    then show *?thesis*
      by (*auto intro*!: *exI*[**where** *x*=*id-cblinfun*])
  qed

  let *?uniform* = ‹*uniform-superpos* (*UNIV* :: 'a set)›
  define α **where** ‹α = *complex-of-real* (*1* / *sqrt* (*of-nat CARD*('a)))›
  define *p p2 p4 a c* **where** ‹*p* = *cinner ?uniform* (*ket* (*0*::'a))› **and** ‹*p2* = *1* − *p* ∗ *p*›
    and ‹*p4* = *p2* ∗ *p2*› **and** ‹*a* = (*1*+*p*) / *p2*› **and** ‹*c* = (−*1*−*p*) / *p2*›
  define *T* :: ‹'a update› **where**
    ‹*T* = *a* $*_C$ *butterfly* (*ket 0*) *?uniform* + *a* $*_C$ *butterfly ?uniform* (*ket 0*)
      + *c* $*_C$ *selfbutter* (*ket 0*) + *c* $*_C$ *selfbutter ?uniform* + *id-cblinfun*›
  have *p*α: ‹*p* = α›
    apply (*simp add*: *p-def cinner-ket-right* α*-def*)
    apply *transfer*
    by *simp*
  have *p20*: ‹*p2* ≠ *0*›
    unfolding α*-def p2-def p*α **using** ‹*CARD*('a) ≥ *2*› **apply** *auto*
     by (*smt* (*verit*) *complex-of-real-leq-1-iff numeral-nat-le-iff of-real-1 of-real-power power2-eq-square*
real-sqrt-pow2
       rel-simps(*26*) *semiring-norm*(*69*))
  have *h1*: ‹*a* ∗ *p* + *c* + *1* = *0*›
    using *p20* **apply** (*simp add*: *a-def c-def*)
   by (*metis add.assoc add-divide-distrib add-neg-numeral-special*(*8*) *diff-add-cancel divide-eq-minus-1-iff*
minus-diff-eq mult.commute mult-1 p2-def ring-class.ring-distribs(*2*) *uminus-add-conv-diff*)
  have *h2*: ‹*a* + *c* ∗ *p* = *1*›
    using *p20* **apply** (*simp add*: *c-def*)
   by (*metis a-def ab-group-add-class.ab-diff-conv-add-uminus add.commute add.inverse-inverse add-neg-numeral-special*(
add-right-cancel c-def divide-minus-left h1 minus-add-distrib mult-minus-left times-divide-eq-left)

  have [*simp*]: ‹*?uniform* $\cdot_C$ *?uniform* = *1*›
    by (*simp add*: *cdot-square-norm norm-uniform-superpos*)
  have [*simp*]: ‹*ket* (*0*::'a) $\cdot_C$ *?uniform* = *cnj p*›
    by (*simp add*: *p-def*)

  have ‹*T* $*_V$ *ket 0* = (*a* ∗ *p* + *c* + *1*) $*_C$ *ket 0* + (*a* + *c* ∗ *p*) $*_C$ *?uniform*›
    unfolding *T-def*
    by (*auto simp*: *cblinfun.add-left scaleC-add-left simp flip*: *p-def*)
  also have ‹. . . = *?uniform*›
    by (*simp add*: *h1 h2*)

finally have *1*: ‹*T* $*_V$ *ket 0* = *?uniform*›
  by −

have *scaleC-add-left′*: ‹*v* + *scaleC x w* + *scaleC y w* = *v* + *scaleC* (*x*+*y*) *w*› **for** *x y* **and** *v w* :: ‹′*a update*›
  by (*simp add*: *scaleC-add-left*)
have *sort*:
  ‹*v* + *x* $*_C$ *butterfly ?uniform* (*ket 0*) + *y* $*_C$ *selfbutter* (*ket 0*) = *v* + *y* $*_C$ *selfbutter* (*ket 0*) + *x* $*_C$ *butterfly ?uniform* (*ket 0*)›
  ‹*v* + *x* $*_C$ *selfbutter ?uniform* + *y* $*_C$ *butterfly* (*ket 0*) *?uniform* = *v* + *y* $*_C$ *butterfly* (*ket 0*) *?uniform* + *x* $*_C$ *selfbutter ?uniform*›
  ‹*v* + *x* $*_C$ *selfbutter ?uniform* + *y* $*_C$ *butterfly ?uniform* (*ket 0*) = *v* + *y* $*_C$ *butterfly ?uniform* (*ket 0*) + *x* $*_C$ *selfbutter ?uniform*›
  ‹*v* + *x* $*_C$ *selfbutter ?uniform* + *y* $*_C$ *selfbutter* (*ket 0*) = *v* + *y* $*_C$ *selfbutter* (*ket 0*) + *x* $*_C$ *selfbutter ?uniform*›
  ‹*v* + *x* $*_C$ *butterfly* (*ket 0*) *?uniform* + *y* $*_C$ *selfbutter* (*ket 0*) = *v* + *y* $*_C$ *selfbutter* (*ket 0*) + *x* $*_C$ *butterfly* (*ket 0*) *?uniform*›
  ‹*v* + *x* $*_C$ *butterfly* (*ket 0*) *?uniform* + *y* $*_C$ *butterfly ?uniform* (*ket 0*) = *v* + *y* $*_C$ *butterfly ?uniform* (*ket 0*) + *x* $*_C$ *butterfly* (*ket 0*) *?uniform*›
  for *v* :: ‹′*a update*› **and** *x y*
  by *auto*

have *aux*: ‹*x* = *0* ⟷ *x* ∗ *p4* = *0*› **for** *x*
  by (*simp add*: *p20 p4-def*)

have [*simp*]: ‹*cnj p* = *p*›
  by (*simp add*: *α-def pα*)
have [*simp*]: ‹*cnj c* = *c*›
  by (*simp add*: *c-def p2-def*)
have [*simp*]: ‹*cnj a* = *a*›
  by (*simp add*: *a-def p2-def*)
have [*simp*]: ‹*p4* ≠ *0*›
  by (*simp add*: *p20 p4-def*)
have [*simp*]: ‹*x* ∗ *p4* / *p2* = *x* ∗ *p2*› **for** *x*
  by (*simp add*: *p4-def*)

have *h3*: ‹*2* ∗ *c* + (*2* ∗ (*a* ∗ (*c* ∗ *p*)) + (*a* ∗ *a* + *c* ∗ *c*)) = *0*›
  apply (*subst aux*)
  apply (*simp add*: *a-def c-def distrib-right distrib-left p20 add-divide-distrib*
    right-diff-distrib left-diff-distrib diff-divide-distrib
    flip: *p4-def add.assoc*
    del: *mult-eq-0-iff vector-space-over-itself.scale-eq-0-iff*)
  by (*simp add*: *p4-def p2-def right-diff-distrib left-diff-distrib flip*: *add.assoc*)
have *h4*: ‹*2* ∗ *a* + (*2* ∗ (*a* ∗ *c*) + (*a* ∗ *a* ∗ *p* + *c* ∗ *c* ∗ *p*)) = *0*›
  apply (*subst aux*)
  apply (*simp add*: *a-def c-def distrib-right distrib-left p20 add-divide-distrib*
    right-diff-distrib left-diff-distrib diff-divide-distrib
    flip: *p4-def add.assoc*
    del: *mult-eq-0-iff vector-space-over-itself.scale-eq-0-iff*)
  by (*simp add*: *p4-def p2-def right-diff-distrib left-diff-distrib flip*: *add.assoc*)

have *2*: ‹*T* o_{CL} *T*∗ = *id-cblinfun*›
  unfolding *T-def*
  apply (*simp add*: *cblinfun-compose-add-left cblinfun-compose-add-right adj-plus*
    scaleC-add-right flip: *p-def add.assoc mult.assoc*)

apply (*simp add*: *sort scaleC-add-left′ flip*: *scaleC-add-left*)
by (*simp add*: *h3 h4*)

have *3*: ‹*T*∗ = *T*›
unfolding *T-def*
by (*auto simp*: *adj-plus*)

from *2 3* **have** *4*: ‹*unitary T*›
by (*simp add*: *unitary-def*)

from *1 4* **show** *?thesis*
by *auto*
qed

Set of total functions

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

lemma *total-functions-def2*: ‹*total-functions* = (*comp Some*) ‘ *UNIV*›
proof −
have ‹*x* ∈ *range* ((∘) *Some*)› **if** ‹*None* ∉ *range x*› **for** *x* :: ‹′*x* ⇒ ′*y option*›
by (*metis function-factors-right option.collapse range-eqI that*)
then show *?thesis*
unfolding *total-functions-def* **by** *auto*
qed

lemma *total-functions-def3*: ‹*total-functions* = {*f*. *dom f* = *UNIV*}›
by (*force simp add*: *total-functions-def*)

lemma *card-total-functions*: ‹*card* (*total-functions* :: (′*x* ⇒ ′*y option*) *set*) = *CARD*(′*y*) ^ *CARD*(′*x*::*finite*)›
proof −
have ‹*card* (*total-functions* :: (′*x* ⇒ ′*y option*) *set*) = *CARD* (′*x* ⇒ ′*y*)›
unfolding *total-functions-def2*
by (*simp add*: *card-image fun.inj-map*)
also have ‹. . . = *CARD*(′*y*) ^ *CARD*(′*x*)›
by (*simp add*: *card-fun*)
finally show *?thesis*
by −
qed

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)*›
by (*simp add*: *dummy2-def N-def0*)

lemma *Nneq0*[*iff*]: ‹$N \neq 0$›
by (*simp add*: *N-def*)

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*
by (*metis add.right-inverse group-cancel.add1 group-cancel.rule0 y-cancel*)

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*)›
unfolding *switch0-def classical-operator-ket-finite*
by *auto*

lemma *switch0-0*[*simp*]: ‹*switch0* $*_V$ *ket* (*Some 0*) = *ket None*›
unfolding *switch0-def classical-operator-ket-finite*
by *auto*

lemma *switch0-other*: ‹*switch0* $*_V$ *ket* (*Some x*) = *ket* (*Some x*)› **if** ‹$x \neq 0$›
unfolding *switch0-def classical-operator-ket-finite*
using *that* **by** *auto*

lemma *unitary-switch0*[*simp*]: ‹*unitary switch0*›
unfolding *switch0-def*
apply (*rule unitary-classical-operator*)
by *auto*

lemma *switch0-adj*[*simp*]: ‹*switch0*$*$ = *switch0*›
unfolding *switch0-def*
apply (*subst classical-operator-adjoint*)
apply *simp*
by (*simp add*: *inv-map-total*)

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-superpos UNIV* = ($\sum$ *d*∈*UNIV*. α $*_C$ *ket* (*d*::′*y*))›
  apply (*subst ell2-sum-ket*)
  by (*simp add*: *uniform-superpos.rep-eq* α*-def N-def*)

lemma *compress1-None*[*simp*]: ‹*compress1* $*_V$ *ket None* = ($\sum$ *d*∈*UNIV*. α $*_C$ *ket* (*Some d*))›
  by (*auto simp*: *cblinfun.sum-right compress1-def lift-op-adj trafo-0 uniform-superpos-y-sum cblinfun.scaleC-right*)

lemma *compress1-Some*[*simp*]: ‹*compress1* $*_V$ *ket* (*Some d*) =
    ket (*Some d*) − ($\sum$ *d*∈*UNIV*. α^2 $*_C$ *ket* (*Some d*)) + α $*_C$ *ket None*›
proof −
  define *c* **where** ‹*c e* = *cinner* (*ket e*) (*trafo*∗ $*_V$ *ket d*)› **for** *e*
  have *c0*: ‹*c 0* = α›
    apply (*simp add*: *c-def cinner-adj-right trafo-0*)
    by (*simp add*: α*-def N-def cinner-ket-right uniform-superpos.rep-eq*)

  have ‹*compress1* $*_V$ *ket* (*Some d*) = *lift-op trafo* $*_V$ *switch0* $*_V$ *lift-ell2* $*_V$ *trafo*∗ $*_V$ *ket d*›
    by (*auto simp*: *compress1-def lift-op-adj*)
  also have ‹. . . = *lift-op trafo* $*_V$ *switch0* $*_V$ *lift-ell2* $*_V$ ($\sum$ *e*∈*UNIV*. *c e* $*_C$ *ket e*)›
    by (*simp add*: *c-def cinner-ket-left flip*: *ell2-sum-ket*)
  also have ‹. . . = *lift-op trafo* $*_V$ *switch0* $*_V$ ($\sum$ *e*∈*UNIV*. *c e* $*_C$ *ket* (*Some e*))›
    by (*auto simp*: *cblinfun.sum-right cblinfun.scaleC-right*)
  also have ‹. . . = *lift-op trafo* $*_V$ *switch0* $*_V$ (($\sum$ *e*∈−{*0*}. *c e* $*_C$ *ket* (*Some e*)) + *c 0* $*_C$ *ket* (*Some 0*))›
    apply (*subst asm-rl*[*of* ‹*UNIV* = *insert 0* (−{*0*})›])
    by (*auto simp add*: *add.commute*)
  also have ‹. . . = *lift-op trafo* $*_V$ (($\sum$ *e*∈−{*0*}. *c e* $*_C$ (*switch0* $*_V$ *ket* (*Some e*))) + *c 0* $*_C$ *switch0* $*_V$ *ket* (*Some 0*))›
    by (*simp add*: *cblinfun.add-right cblinfun.scaleC-right cblinfun.sum-right*)
  also have ‹. . . = *lift-op trafo* $*_V$ (($\sum$ *e*∈−{*0*}. *c e* $*_C$ *ket* (*Some e*)) + *c 0* $*_C$ *ket None*)›
    by (*simp add*: *switch0-other*)
  also have ‹. . . = *lift-op trafo* $*_V$ (($\sum$ *e*∈*UNIV*. *c e* $*_C$ *ket* (*Some e*)) − *c 0* $*_C$ *ket* (*Some 0*) + *c 0* $*_C$ *ket None*)›
    by (*simp add*: *Compl-eq-Diff-UNIV sum-diff*)
  also have ‹. . . = ($\sum$ *e*∈*UNIV*. *c e* $*_C$ *lift-ell2* $*_V$ *trafo* $*_V$ *ket e*) − *c 0* $*_C$ *lift-ell2* $*_V$ *trafo* $*_V$ *ket 0* + *c 0* $*_C$ *ket None*›
    by (*simp add*: *cblinfun.add-right cblinfun.diff-right cblinfun.scaleC-right cblinfun.sum-right*)
  also have ‹. . . = *lift-ell2* $*_V$ *trafo* $*_V$ ($\sum$ *e*∈*UNIV*. *c e* $*_C$ *ket e*) − *c 0* $*_C$ *lift-ell2* $*_V$ *uniform-superpos UNIV* + *c 0* $*_C$ *ket None*›
    by (*simp add*: *trafo-0 cblinfun.scaleC-right cblinfun.sum-right*)
  also have ‹. . . = *lift-ell2* $*_V$ *trafo* $*_V$ *trafo*∗ $*_V$ *ket d* − *c 0* $*_C$ *lift-ell2* $*_V$ *uniform-superpos UNIV* + *c 0* $*_C$ *ket None*›
    by (*simp add*: *c-def cinner-ket-left flip*: *ell2-sum-ket*)
  also have ‹. . . = *lift-ell2* $*_V$ (*trafo* o_{CL} *trafo*∗) $*_V$ *ket d* − *c 0* $*_C$ *lift-ell2* $*_V$ *uniform-superpos UNIV* + *c 0* $*_C$ *ket None*›
    by (*metis cblinfun-apply-cblinfun-compose*)
  also have ‹. . . = *lift-ell2* $*_V$ *ket d* − *c 0* $*_C$ *lift-ell2* $*_V$ *uniform-superpos UNIV* + *c 0* $*_C$ *ket None*›
    by *auto*
  also have ‹. . . = *ket* (*Some d*) − *c 0* $*_C$ ($\sum$ *d*∈*UNIV*. α $*_C$ *ket* (*Some d*)) + *c 0* $*_C$ *ket None*›
    by (*auto simp*: *uniform-superpos-y-sum mult.commute scaleC-sum-right cblinfun.scaleC-right cblinfun.sum-right*)
  also have ‹. . . = *ket* (*Some d*) − ($\sum$ *d*∈*UNIV*. α^2 $*_C$ *ket* (*Some d*)) + α $*_C$ *ket None*›
    by (*simp add*: *c0 power2-eq-square scaleC-sum-right*)

finally show *?thesis*
  by −
qed

lemma *unitary-compress1*[*simp*]: ‹*unitary compress1*›
  by (*simp add*: *compress1-def*)

lemma *compress1-adj*[*simp*]: ‹*compress1*∗ = *compress1*›
  by (*simp add*: *compress1-def cblinfun-compose-assoc*)

lemma *compress1-square*: ‹*compress1* o_{CL} *compress1* = *id-cblinfun*›
  by (*metis compress1-adj unitary-compress1 unitary-def*)

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* ⇀ ′*y*) *update*›
  where ‹*compress* = *apply-every UNIV* (λ-. *compress1*)›

lemma *unitary-compress*[*simp*]: ‹*unitary compress*›
  by (*simp add*: *compress-def apply-every-unitary*)

lemma *compress-selfinverse*: ‹*compress* o_{CL} *compress* = *id-cblinfun*›
  by (*simp add*: *compress-def apply-every-mult compress1-square*)

lemma *compress-adj*: ‹*compress*∗ = *compress*›
  by (*simp add*: *compress-def apply-every-adj*)

lemma *compress-empty*: ‹*compress* $*_V$ *ket Map.empty* = *superpos-total*›
proof −
  have ∗: ‹*apply-every M* (λ-. *compress1*) $*_V$ *ket Map.empty* =
      ($\sum$*f*|*dom f* = *M*. *ket f* $/_R$ *sqrt* (*CARD*(′*y*) ^ *card M*))› **for** *M* :: ‹′*x set*›
  proof (*use finite*[*of M*] **in** *induction*)
    case *empty*
    then show *?case*
      by *simp*
  next
    case (*insert x F*)
    have ‹*apply-every* (*insert x F*) (λ-. *compress1*) $*_V$ *ket Map.empty*
      = *function-at x compress1* $*_V$ *apply-every F* (λ-. *compress1*) $*_V$ *ket Map.empty*›
      using *insert.hyps* **by** (*simp add*: *apply-every-insert*)
    also have ‹. . . = *function-at x compress1* $*_V$ ($\sum$*f* | *dom f* = *F*. *ket f* $/_R$ *sqrt* (*real* (*CARD*(′*y*) ^ *card F*)))›
      by (*simp add*: *insert.IH*)
    also have ‹. . . = ($\sum$*f* | *dom f* = *F*. (*function-at x compress1* $*_V$ *ket f*) $/_R$ *sqrt* (*real* (*CARD*(′*y*) ^ *card F*)))›
      by (*simp add*: *cblinfun.real.scaleR-right cblinfun.sum-right*)
    also have ‹. . . = ($\sum$*f* | *dom f* = *F*. ($\sum$*y*∈*UNIV*. *Rep-ell2* (*compress1* $*_V$ *ket* (*f x*)) *y* $*_C$ *ket* (*f*(*x* := *y*))) $/_R$ *sqrt* (*real* (*CARD*(′*y*) ^ *card F*)))›
      by (*simp add*: *function-at-ket*)
    also have ‹. . . = ($\sum$*f* | *dom f* = *F*. ($\sum$*y*∈*UNIV*. *Rep-ell2* (*compress1* $*_V$ *ket None*) *y* $*_C$ *ket* (*f*(*x*

:= y))) $/_R$ sqrt (real (CARD('y) ^ card F)))›
by (smt (verit) Finite-Cartesian-Product.sum-cong-aux domIff local.insert(2) mem-Collect-eq)
also have ‹... = (∑f | dom f = F. (∑y∈UNIV. Rep-ell2 (∑d∈UNIV. α $*_C$ ket (Some d)) y $*_C$ ket (f(x := y))) $/_R$ sqrt (real (CARD('y) ^ card F)))›
by simp
also have ‹... = (∑f | dom f = F. (∑y∈UNIV. (∑d∈UNIV. α $*_C$ Rep-ell2 (ket (Some d)) y) $*_C$ ket (f(x := y))) $/_R$ sqrt (real (CARD('y) ^ card F)))›
apply (subst complex-vector.linear-sum[**where** f=‹λx. Rep-ell2 x -›])
apply (simp add: clinearI plus-ell2.rep-eq scaleC-ell2.rep-eq)
apply (subst clinear.scaleC[**where** f=‹λx. Rep-ell2 x -›])
by (simp-all add: clinearI plus-ell2.rep-eq scaleC-ell2.rep-eq)
also have ‹... = (∑f | dom f = F. (∑y∈UNIV. (if y = None then 0 else α) $*_C$ ket (f(x := y))) $/_R$ sqrt (real (CARD('y) ^ card F)))›
apply (rule sum.cong, simp)
subgoal for f
apply (rule arg-cong[**where** f=‹λx. x $/_R$ -›])
apply (rule sum.cong, simp)
subgoal for y
apply (subst sum-single[**where** i=‹the y›])
by (auto simp: ket.rep-eq)
by −
by −
also have ‹... = (∑f | dom f = F. (∑y∈range Some. α $*_C$ ket (f(x := y))) $/_R$ sqrt (real (CARD('y) ^ card F)))›
apply (rule sum.cong, simp)
apply (subst sum.mono-neutral-cong-right[**where** S=‹range Some› **and** h=‹λy. α $*_C$ ket (-(x := y))›])
by auto
also have ‹... = (∑f | dom f = F. ∑y∈range Some. α $*_C$ ket (f(x := y)) $/_R$ sqrt (real (CARD('y) ^ card F)))›
by (simp add: scaleR-right.sum)
also have ‹... = (∑(f, y)∈{f. dom f = F} × range Some.
α $*_C$ ket (f(x := y)) $/_R$ sqrt (real (CARD('y) ^ card F)))›
by (simp add: sum.cartesian-product)
also have ‹... = (∑(f, y)∈(λf. (f(x:=None), f x)) ‘ {f. dom f = insert x F}.
α $*_C$ ket (f(x := y))) $/_R$ sqrt (real (CARD('y) ^ card F))›
proof −
have 1: ‹{f. dom f = F} × range Some = (λf. (f(x := None), f x)) ‘ {f. dom f = insert x F}›
proof (rule Set.set-eqI, rule iffI)
fix z :: ‹('x ⇒ 'a option) × 'a option›
assume asm: ‹z ∈ {f. dom f = F} × range Some›
define f **where** ‹f = (fst z)(x := snd z)›
have ‹f ∈ {f. dom f = insert x F}›
using asm **by** (auto simp: f-def)
moreover have ‹(λf. (f(x := None), f x)) f = z›
using asm insert.hyps **by** (auto simp add: f-def)
ultimately show ‹z ∈ (λf. (f(x := None), f x)) ‘ {f. dom f = insert x F}›
by auto
next
fix z :: ‹('x ⇒ 'a option) × 'a option›
assume ‹z ∈ (λf. (f(x := None), f x)) ‘ {f. dom f = insert x F}›
then obtain f **where** ‹dom f = insert x F› **and** ‹z = (λf. (f(x := None), f x)) f›
by auto
then show ‹z ∈ {f. dom f = F} × range Some›
using insert.hyps **by** auto

qed
show *?thesis*
apply (*subst scaleR-right.sum*)
apply (*rule sum.cong*)
using *1* **by** *auto*
qed
also have ‹... $= (\sum f|$ *dom f* $=$ *insert x F*. $\alpha *_C$ *ket f*) $/_R$ *sqrt* (*real* (*CARD*(*'y*) ^ *card F*))›
apply (*subst sum.reindex*)
apply *auto*
by (*smt* (*verit*) *fun-upd-idem-iff fun-upd-upd inj-on-def prod.simps*(*1*))
also have ‹... $= (\sum f|$ *dom f* $=$ *insert x F*. *ket f* $/_R$ *sqrt* (*real* (*CARD*(*'y*) ^ *card* (*insert x F*))))›
by (*simp add*: *card-insert-disjoint insert.hyps real-sqrt-mult* α*-def N-def scaleR-scaleC divide-inverse-commute flip*: *scaleC-sum-right*)
finally show *?case*
by −
qed

have ‹($\sum f|$*dom f* $=$ *UNIV*. *ket f* $/_R$ *sqrt* (*CARD*(*'y*) ^ *CARD*(*'x*))) = (*superpos-total* :: (*'x* ⇒ *'y option*) *ell2*)›
unfolding *uniform-superpos-def2*
apply (*rule sum.cong*)
apply (*simp add*: *total-functions-def3*)
by (*simp add*: *card-total-functions scaleR-scaleC*)

with ∗[*of UNIV*]
show *?thesis*
by (*simp flip*: *compress-def*)
qed

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-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* × *'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 *stan-*

dard-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*)›
by (*simp add*: *standard-query1-def classical-operator-ket-finite*)

lemma *standard-query1-None*[*simp*]: ‹*standard-query1* $*_V$ *ket* (y, *None*) = *ket* (y, *None*)›
by (*simp add*: *standard-query1-def classical-operator-ket-finite*)

lemma *standard-query1′-Some*[*simp*]: ‹*standard-query1′* $*_V$ *ket* (y, *Some z*) = *ket* (y + z, *Some z*)›
by (*simp add*: *standard-query1′-def classical-operator-ket-finite*)

lemma *standard-query1′-None*[*simp*]: ‹*standard-query1′* $*_V$ *ket* (y, *None*) = *0*›
by (*simp add*: *standard-query1′-def classical-operator-ket-finite*)

lemma *unitary-standard-query1*[*simp*]: ‹*unitary standard-query1*›
unfolding *standard-query1-def*
apply (*rule unitary-classical-operator*)
apply (*rule o-bij*[**where** g=‹λ(y,z). *case z of None* ⇒ (y,*None*) | *Some z′* ⇒ (y − z′, z)›])
by (*auto intro*!: *ext simp*: *case-prod-beta cong del*: *option.case-cong split*!: *option.split option.split-asm*)

lemma *norm-standard-query1′*[*simp*]: ‹*norm standard-query1′* = *1*›
proof (*rule order.antisym*)
show ‹*norm standard-query1′* ≤ *1*›
unfolding *standard-query1′-def*
apply (*rule classical-operator-norm-inj*)
by (*auto simp*: *inj-map-def split*!: *option.split-asm*)
show ‹*norm standard-query1′* ≥ *1*›
apply (*rule cblinfun-norm-geqI*[**where** x=‹*ket* (*undefined*, *Some undefined*)›])
by *simp*
qed

lemma *standard-query1-selfinverse*[*simp*]: ‹*standard-query1* o_{CL} *standard-query1* = *id-cblinfun*›
proof −
have ∗: ‹(*Some* ∘ (λ(y::′y, z). *case z of None* ⇒ (y, *None*) | *Some z′* ⇒ (y + z′, z)) $\circ_m$
(*Some* ∘ (λ(y, z). *case z of None* ⇒ (y, *None*) | *Some z′* ⇒ (y + z′, z)))) = *Some*›
by (*auto intro*!: *ext*, *rename-tac a b*, *case-tac b*, *auto*)
show *?thesis*
by (*auto simp*: *standard-query1-def classical-operator-mult* ∗)
qed

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 ⇒$_{CL}$ ('x × 'y × ('x ⇀ 'y)) ell2› **where**
‹*standard-query = controlled-op* (λx. (*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 ⇒$_{CL}$ ('x × 'y × ('x ⇀ 'y)) ell2› **where**
‹*standard-query' = controlled-op* (λx. (*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)›
by (*auto simp*: *standard-query-def*)

lemma *standard-query-ket-full-Some*:
assumes ‹*H x = Some z*›
shows ‹*standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket* (*x*, *y* + *z*, *H*)›
proof −
obtain *H'* **where** *pf-xH*: ‹*puncture-function x H* = (*H x*, *H'*)›
by (*metis fst-puncture-function prod.collapse*)
have ‹*standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket x* $\otimes_s$ *sandwich* (*id-cblinfun* $\otimes_o$ *function-at-U x*) ((*id* $\otimes_r$ *Fst*) *standard-query1*) $*_V$ *ket y* $\otimes_s$ *ket H*›
by (*simp add*: *standard-query-ket function-at-def pair-o-tensor-right pair-Fst-Snd pair-o-tensor-right unitary-sandwich-register pair-o-tensor-right register-tensor-distrib-right id-tensor-sandwich flip*: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ (*id-cblinfun* $\otimes_o$ *function-at-U x*) $*_V$ (*id* $\otimes_r$ *Fst*) *standard-query1* $*_V$ (*ket y* $\otimes_s$ *ket* (*H x*) $\otimes_s$ *ket H'*)›
(**is** ‹- = - $\otimes_s$ - $*_V$ *?R standard-query1* $*_V$ -›)
by (*simp add*: *sandwich-apply' tensor-op-adjoint tensor-op-ell2 pf-xH assms flip*: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ (*id-cblinfun* $\otimes_o$ *function-at-U x*) $*_V$ (*ket* (*y* + *z*) $\otimes_s$ *ket* (*H x*) $\otimes_s$ *ket H'*)›
apply (*subst asm-rl*[*of* ‹(*id* $\otimes_r$ *Fst*) = *assoc o Fst*›])
subgoal by (*auto intro*!: *tensor-extensionality simp add*: *register-tensor-is-register Fst-def*)
apply (*simp add*: *Fst-def assoc-ell2-sandwich sandwich-apply' assoc-ell2'-tensor tensor-op-ell2 assms*)
apply (*simp add*: *tensor-ell2-ket del*: *function-at-U-ket*)
by (*simp add*: *assoc-ell2-tensor tensor-op-ell2 flip*: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ *ket* (*y* + *z*) $\otimes_s$ *ket H*›
apply (*simp add*: *tensor-op-ell2 flip*: *tensor-ell2-ket*)
by (*simp flip*: *pf-xH add*: *tensor-ell2-ket*)
finally show *?thesis*
by (*simp add*: *tensor-ell2-ket*)
qed

lemma *standard-query-ket-full-None*:
assumes ‹*H x = None*›
shows ‹*standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket* (*x*, *y*, *H*)›
proof −
obtain *H'* **where** *pf-xH*: ‹*puncture-function x H* = (*H x*, *H'*)›
by (*metis fst-puncture-function prod.collapse*)
have ‹*standard-query* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket x* $\otimes_s$ *sandwich* (*id-cblinfun* $\otimes_o$ *function-at-U x*) ((*id* $\otimes_r$ *Fst*) *standard-query1*) $*_V$ *ket y* $\otimes_s$ *ket H*›
by (*simp add*: *standard-query-ket function-at-def pair-o-tensor-right pair-Fst-Snd pair-o-tensor-right unitary-sandwich-register pair-o-tensor-right*

register-tensor-distrib-right id-tensor-sandwich
flip: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ (*id-cblinfun* $\otimes_o$ *function-at-U x*) $*_V$ (*id* $\otimes_r$ *Fst*) *standard-query1* $*_V$ *ket y* $\otimes_s$ *ket* (*H x*) $\otimes_s$ *ket H'*›
by (*simp add*: *sandwich-apply' tensor-op-adjoint tensor-op-ell2 pf-xH assms flip*: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ (*id-cblinfun* $\otimes_o$ *function-at-U x*) $*_V$ *ket y* $\otimes_s$ *ket* (*H x*) $\otimes_s$ *ket H'*›
apply (*subst asm-rl*[*of* ‹(*id* $\otimes_r$ *Fst*) = *assoc o Fst*›])
subgoal by (*auto intro*!: *tensor-extensionality simp add*: *register-tensor-is-register Fst-def*)
apply (*simp add*: *Fst-def assoc-ell2-sandwich sandwich-apply' assoc-ell2'-tensor tensor-op-ell2 assms*)
apply (*simp add*: *tensor-ell2-ket del*: *function-at-U-ket*)
by (*simp add*: *assoc-ell2-tensor tensor-op-ell2 flip*: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ *ket y* $\otimes_s$ *ket H*›
apply (*simp add*: *tensor-op-ell2 flip*: *tensor-ell2-ket*)
by (*simp flip*: *pf-xH add*: *tensor-ell2-ket*)
finally show *?thesis*
by (*simp add*: *tensor-ell2-ket*)
qed

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)›
by (*auto simp*: *standard-query'-def*)

lemma *standard-query'-ket-full-Some*:
assumes ‹*H x* = *Some z*›
shows ‹*standard-query'* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket* (*x*, *y* + *z*, *H*)›
proof −
obtain *H'* **where** *pf-xH*: ‹*puncture-function x H* = (*H x*, *H'*)›
by (*metis fst-puncture-function prod.collapse*)
have ‹*standard-query'* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket x* $\otimes_s$ *sandwich* (*id-cblinfun* $\otimes_o$ *function-at-U x*) ((*id* $\otimes_r$ *Fst*) *standard-query1'*) $*_V$ *ket y* $\otimes_s$ *ket H*›
by (*simp add*: *standard-query'-ket function-at-def pair-o-tensor-right pair-Fst-Snd*
pair-o-tensor-right unitary-sandwich-register pair-o-tensor-right
register-tensor-distrib-right id-tensor-sandwich
flip: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ (*id-cblinfun* $\otimes_o$ *function-at-U x*) $*_V$ (*id* $\otimes_r$ *Fst*) *standard-query1'* $*_V$ (*ket y* $\otimes_s$ *ket* (*H x*) $\otimes_s$ *ket H'*)›
(**is** ‹- = - $\otimes_s$ - $*_V$ *?R standard-query1'* $*_V$ -›)
by (*simp add*: *sandwich-apply' tensor-op-adjoint tensor-op-ell2 pf-xH assms flip*: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ (*id-cblinfun* $\otimes_o$ *function-at-U x*) $*_V$ (*ket* (*y* + *z*) $\otimes_s$ *ket* (*H x*) $\otimes_s$ *ket H'*)›
apply (*subst asm-rl*[*of* ‹(*id* $\otimes_r$ *Fst*) = *assoc o Fst*›])
subgoal by (*auto intro*!: *tensor-extensionality simp add*: *register-tensor-is-register Fst-def*)
apply (*simp add*: *Fst-def assoc-ell2-sandwich sandwich-apply' assoc-ell2'-tensor tensor-op-ell2 assms*)
apply (*simp add*: *tensor-ell2-ket del*: *function-at-U-ket*)
by (*simp add*: *assoc-ell2-tensor tensor-op-ell2 flip*: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ *ket* (*y* + *z*) $\otimes_s$ *ket H*›
apply (*simp add*: *tensor-op-ell2 flip*: *tensor-ell2-ket*)
by (*simp flip*: *pf-xH add*: *tensor-ell2-ket*)
finally show *?thesis*
by (*simp add*: *tensor-ell2-ket*)
qed

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

obtain H' **where** *pf-xH*: ‹*puncture-function x H* = (*H x*, *H'*)›
  by (*metis fst-puncture-function prod.collapse*)
have ‹*standard-query'* $*_V$ (*ket* (*x*,*y*,*H*)) = *ket x* $\otimes_s$ *sandwich* (*id-cblinfun* $\otimes_o$ *function-at-U x*) ((*id* $\otimes_r$ *Fst*) *standard-query1'*) $*_V$ *ket y* $\otimes_s$ *ket H*›
  by (*simp add*: *standard-query'-ket function-at-def pair-o-tensor-right pair-Fst-Snd pair-o-tensor-right unitary-sandwich-register pair-o-tensor-right register-tensor-distrib-right id-tensor-sandwich flip*: *tensor-ell2-ket*)
also have ‹... = *ket x* $\otimes_s$ (*id-cblinfun* $\otimes_o$ *function-at-U x*) $*_V$ (*id* $\otimes_r$ *Fst*) *standard-query1'* $*_V$ *ket y* $\otimes_s$ *ket* (*H x*) $\otimes_s$ *ket H'*›
  by (*simp add*: *sandwich-apply' tensor-op-adjoint tensor-op-ell2 pf-xH assms flip*: *tensor-ell2-ket*)
also have ‹... = *0*›
  apply (*subst asm-rl*[*of* ‹(*id* $\otimes_r$ *Fst*) = *assoc o Fst*›])
  subgoal by (*auto intro*!: *tensor-extensionality simp add*: *register-tensor-is-register Fst-def*)
  apply (*simp add*: *Fst-def assoc-ell2-sandwich sandwich-apply' assoc-ell2'-tensor tensor-op-ell2 assms*)
  by (*simp add*: *tensor-ell2-ket del*: *function-at-U-ket*)
finally show *?thesis*
  by −
qed

lemma *standard-query-selfinverse*[*simp*]: ‹*standard-query* o_{CL} *standard-query* = *id-cblinfun*›
  by (*simp add*: *standard-query-def controlled-op-compose register-mult*)

lemma *unitary-standard-query*[*simp*]: ‹*unitary standard-query*›
  by (*auto simp*: *standard-query-def intro*!: *controlled-op-unitary register-unitary*[*of* ‹(-;-)›])

lemma *contracting-standard'-query*[*simp*]: ‹*norm standard-query'* = *1*›
proof (*rule antisym*)
  show ‹*norm standard-query'* ≤ *1*›
  unfolding *standard-query'-def*
  apply (*rule controlled-op-norm-leq*)
  by (*smt* (*verit*) *norm-standard-query1' norm-zero register-norm register-pair-def register-pair-is-register*)
  show ‹*norm standard-query'* ≥ *1*›
  apply (*rule cblinfun-norm-geqI*[**where** *x*=‹*ket* (*undefined*, *undefined*, λ-. *Some undefined*)›])
  apply (*subst standard-query'-ket-full-Some*)
  by *auto*
qed

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** ‹$query1' = Snd\ compress1\ o_{CL}\ standard\text{-}query1'\ o_{CL}\ Snd\ compress1$›

lemma *unitary-query1*[*simp*]: ‹*unitary query1*›
  by (*auto simp*: *query1-def register-unitary intro*!: *unitary-cblinfun-compose*)

lemma *norm-query1'*[*simp*]: ‹*norm query1'* = *1*›
  unfolding *query1'-def*
  apply (*subst norm-isometry-compose'*)
   apply (*simp add*: *Snd-def comp-tensor-op compress1-square isometry-def tensor-op-adjoint*)
  apply (*subst norm-isometry-compose*)
   apply (*simp add*: *Snd-def comp-tensor-op compress1-square isometry-def tensor-op-adjoint*)
  by *simp*

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›)
proof −
  have [*simp*]: ‹$\alpha * \alpha = \alpha^2$› ‹$\alpha * \alpha^2 = \alpha\hat{}3$›
    by (*simp-all add*: *power2-eq-square numeral-2-eq-2 numeral-3-eq-3*)

  have *aux*: ‹$a = a' \Longrightarrow b = b' \Longrightarrow c = c' \Longrightarrow a - b + c = a' - b' + c'$› **for** $a\ b\ c\ a'\ b'\ c'$ :: ‹*'z*::*group-add*›
    by *simp*

  have ‹$Snd\ compress1 *_V ket\ (y,\ None) = (\sum d{\in}UNIV.\ \alpha *_C ket\ (y,\ Some\ d))$›
    by (*simp add*: *query1-def tensor-ell2-scaleC2 tensor-ell2-sum-right flip*: *tensor-ell2-ket*)
  also have ‹$standard\text{-}query1 *_V \ldots = (\sum d{\in}UNIV.\ \alpha *_C ket\ (y + d,\ Some\ d))$›
    by (*simp add*: *cblinfun.scaleC-right cblinfun.sum-right*)
  also have ‹$Snd\ compress1 *_V \ldots =$
    $\alpha *_C (\sum d{\in}UNIV.\ (ket\ (y + d) \otimes_s ket\ (Some\ d)))$
    $-\ \alpha\hat{}3 *_C (\sum z{\in}UNIV.\ \sum d{\in}UNIV.\ (ket\ (y + z) \otimes_s ket\ (Some\ d)))$
    $+\ \alpha^2 *_C (\sum z{\in}UNIV.\ (ket\ (y + z) \otimes_s ket\ None))$›
    by (*simp add*: *tensor-ell2-diff2 tensor-ell2-add2 scaleC-add-right sum.distrib tensor-ell2-sum-right tensor-ell2-scaleC2 sum-subtractf scaleC-diff-right scaleC-sum-right cblinfun.scaleC-right cblinfun.sum-right*
      flip: *tensor-ell2-ket*)
  also have ‹… = *?rhs*›
    apply (*rule aux*)
    subgoal
      by (*simp add*: *tensor-ell2-ket*)
    subgoal
      apply (*subst sum.reindex-bij-betw*[**where** *h*=‹$\lambda d.\ y + d$› **and** *T*=*UNIV*])
      by (*simp-all add*: *tensor-ell2-ket*)
    subgoal
      apply *simp*
      apply (*subst sum.reindex-bij-betw*[**where** *h*=‹$\lambda d.\ y + d$› **and** *T*=*UNIV*])
      by (*simp-all add*: *tensor-ell2-ket*)

```
    by −
  finally show ?thesis
    unfolding query1-def by simp
qed

lemma query1-Some: ‹query1 *V ket (y, Some d) =
        ket (y + d, Some d)
        + α *C ket (y + d, None)
        − α^3 *C (∑ y'∈UNIV. ket (y', None))
        − α^2 *C (∑ d'∈UNIV. ket (y + d', Some d'))
        − α^2 *C (∑ d'∈UNIV. ket (y + d, Some d'))
        + α^2 *C (∑ d'∈UNIV. ket (y, Some d'))
        + α^4 *C (∑ y'∈UNIV. ∑ d'∈UNIV. ket (y', Some d'))›
    (is ‹- = ?rhs›)
proof −
  have [simp]: ‹α * α = α^2› ‹α^2 * α = α^3›
    by (simp-all add: power2-eq-square numeral-2-eq-2 numeral-3-eq-3)

  have aux: ‹a=a' ⟹ b=b' ⟹ c=c' ⟹ d=d' ⟹ e=e' ⟹ f=f' ⟹ g=g'
        ⟹  a' − e' + b' + g' − d' − c' + f' = a + b − c − d − e + f + g›
    for a b c d e f g a' b' c' d' e' f' g' :: ‹'z::ab-group-add›
    by simp

  have ‹Snd compress1 *V ket (y, Some d) =
            ket (y, Some d) − α^2 *C (∑ d'∈UNIV. ket (y, Some d')) + α *C ket (y, None)›
    by (simp add: query1-def tensor-ell2-scaleC2 tensor-ell2-diff2 tensor-ell2-add2 tensor-ell2-sum-right
          flip: tensor-ell2-ket scaleC-sum-right)
  also have ‹standard-query1 *V ... = ket (y + d, Some d) − α^2 *C (∑ d'∈UNIV. ket (y + d', Some
d')) + α *C ket (y, None)›
    by (simp add: cblinfun.add-right cblinfun.diff-right cblinfun.scaleC-right cblinfun.sum-right)
  also have ‹Snd compress1 *V ... =
              ket (y + d, Some d)
              − α^2 *C (∑ d'∈UNIV. ket (y + d, Some d'))
              + α *C ket (y + d, None)
              + α^4 *C (∑ z∈UNIV. ∑ d'∈UNIV. ket (y + z, Some d'))
              − α^2 *C (∑ d'∈UNIV. ket (y + d', Some d'))
              − α^3 *C (∑ z∈UNIV. ket (y + z, None))
              + α^2 *C (∑ d'∈UNIV. ket (y, Some d'))›
    by (simp add: tensor-ell2-diff2 tensor-ell2-add2 scaleC-add-right sum.distrib
        tensor-ell2-scaleC2 sum-subtractf scaleC-diff-right scaleC-sum-right tensor-ell2-sum-right
        cblinfun.add-right cblinfun.diff-right diff-diff-eq2 cblinfun.scaleC-right cblinfun.sum-right
        flip: tensor-ell2-ket diff-diff-eq scaleC-sum-right)
  also have ‹... = ?rhs›
    apply (rule aux)
    subgoal by rule
    subgoal by rule
    subgoal
      apply (subst sum.reindex-bij-betw[where h=‹λd. y + d› and T=UNIV])
      by simp-all
    subgoal by rule
    subgoal by rule
    subgoal by rule
    subgoal
      apply (subst sum.reindex-bij-betw[where h=‹λd. y + d› and T=UNIV])
      by simp-all
```

by −
finally show *?thesis*
unfolding *query1-def* **by** *simp*
qed

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

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*›)
proof −
have [*simp*]: ‹$\alpha * \alpha = \alpha^2$› ‹$\alpha * \alpha^2 = \alpha \hat{~} 3$›
by (*simp-all add*: *power2-eq-square numeral-2-eq-2 numeral-3-eq-3*)

have *aux*: ‹$a = a' \Longrightarrow b = b' \Longrightarrow c = c' \Longrightarrow a - b + c = a' - b' + c'$› **for** $a\ b\ c\ a'\ b'\ c'$ :: ‹$'z$::*group-add*›
by *simp*

have ‹$Snd\ compress1 *_V ket\ (y, None) = (\sum d{\in}UNIV.\ \alpha *_C ket\ (y, Some\ d))$›
by (*simp add*: *query1-def tensor-ell2-scaleC2 tensor-ell2-sum-right flip*: *tensor-ell2-ket*)
also have ‹$standard\text{-}query1' *_V \ldots = (\sum d{\in}UNIV.\ \alpha *_C ket\ (y + d, Some\ d))$›
by (*simp add*: *cblinfun.scaleC-right cblinfun.sum-right*)
also have ‹$Snd\ compress1 *_V \ldots =$
$\alpha *_C (\sum d{\in}UNIV.\ (ket\ (y + d) \otimes_s ket\ (Some\ d)))$
$- \alpha \hat{~} 3 *_C (\sum z{\in}UNIV.\ \sum d{\in}UNIV.\ (ket\ (y + z) \otimes_s ket\ (Some\ d)))$
$+ \alpha^2 *_C (\sum z{\in}UNIV.\ (ket\ (y + z) \otimes_s ket\ None))$›
by (*simp add*: *tensor-ell2-diff2 tensor-ell2-add2 scaleC-add-right sum.distrib tensor-ell2-sum-right tensor-ell2-scaleC2 sum-subtractf scaleC-diff-right scaleC-sum-right cblinfun.scaleC-right cblinfun.sum-right*
flip: *tensor-ell2-ket*)
also have ‹$\ldots$ = *?rhs*›
apply (*rule aux*)
subgoal
by (*simp add*: *tensor-ell2-ket*)

subgoal
apply (*subst sum.reindex-bij-betw*[**where** h=‹λd. y + d› **and** T=*UNIV*])
by (*simp-all add*: *tensor-ell2-ket*)
subgoal
apply *simp*
apply (*subst sum.reindex-bij-betw*[**where** h=‹λd. y + d› **and** T=*UNIV*])
by (*simp-all add*: *tensor-ell2-ket*)
by −
finally show *?thesis*
unfolding *query1′-def* **by** *simp*
qed

lemma *query1′-Some*: ‹*query1′* $*_V$ *ket* (y, *Some* d) =
ket (y + d, *Some* d)
+ α $*_C$ *ket* (y + d, *None*)
− α^*3* $*_C$ ($\sum$ y'∈*UNIV*. *ket* (y', *None*))
− α^2 $*_C$ ($\sum$ d'∈*UNIV*. *ket* (y + d', *Some* d'))
− α^2 $*_C$ ($\sum$ d'∈*UNIV*. *ket* (y + d, *Some* d'))
+ α^*4* $*_C$ ($\sum$ y'∈*UNIV*. $\sum$ d'∈*UNIV*. *ket* (y', *Some* d'))›
(**is** ‹- = *?rhs*›)
proof −
have [*simp*]: ‹$\alpha * \alpha = \alpha^2$› ‹$\alpha^2 * \alpha$ = α^*3*›
by (*simp-all add*: *power2-eq-square numeral-2-eq-2 numeral-3-eq-3*)

have *aux*: ‹a=a' ⟹ b=b' ⟹ c=c' ⟹ d=d' ⟹ e=e' ⟹ g=g'
⟹ $a' - e' + b' + g' - d' - c' = a + b - c - d - e + g$›
for a b c d e f g a' b' c' d' e' f' g' :: ‹′*z*::*ab-group-add*›
by *simp*

have ‹*Snd compress1* $*_V$ *ket* (y, *Some* d) =
ket (y, *Some* d) − α^2 $*_C$ ($\sum$ d'∈*UNIV*. *ket* (y, *Some* d')) + α $*_C$ *ket* (y, *None*)›
by (*simp add*: *query1-def tensor-ell2-scaleC2 tensor-ell2-diff2 tensor-ell2-add2 tensor-ell2-sum-right*
flip: *tensor-ell2-ket scaleC-sum-right*)
also have ‹*standard-query1′* $*_V$ ... = *ket* (y + d, *Some* d) − α^2 $*_C$ ($\sum$ d'∈*UNIV*. *ket* (y + d', *Some* d'))›
by (*simp add*: *cblinfun.add-right cblinfun.diff-right cblinfun.scaleC-right cblinfun.sum-right*)
also have ‹*Snd compress1* $*_V$ ... =
ket (y + d, *Some* d)
− α^2 $*_C$ ($\sum$ d'∈*UNIV*. *ket* (y + d, *Some* d'))
+ α $*_C$ *ket* (y + d, *None*)
+ α^*4* $*_C$ ($\sum$ z∈*UNIV*. $\sum$ d'∈*UNIV*. *ket* (y + z, *Some* d'))
− α^2 $*_C$ ($\sum$ d'∈*UNIV*. *ket* (y + d', *Some* d'))
− α^*3* $*_C$ ($\sum$ z∈*UNIV*. *ket* (y + z, *None*))›
by (*simp add*: *tensor-ell2-diff2 tensor-ell2-add2 scaleC-add-right sum.distrib tensor-ell2-sum-right*
tensor-ell2-scaleC2 sum-subtractf scaleC-diff-right scaleC-sum-right cblinfun.sum-right
cblinfun.add-right cblinfun.diff-right diff-diff-eq2 cblinfun.scaleC-right
flip: *tensor-ell2-ket diff-diff-eq scaleC-sum-right*)
also have ‹... = *?rhs*›
apply (*rule aux*)
subgoal by *rule*
subgoal by *rule*
subgoal
apply (*subst sum.reindex-bij-betw*[**where** h=‹λd. y + d› **and** T=*UNIV*])
by *simp-all*
subgoal by *rule*

subgoal by *rule*
subgoal
apply (*subst sum.reindex-bij-betw*[**where** *h*=‹λ*d*. *y* + *d*› **and** *T*=*UNIV*])
by *simp-all*
by −
finally show *?thesis*
unfolding *query1′-def* **by** *simp*
qed

lemma *query1′*:
shows ‹*query1′* $*_V$ (*ket yd*) = (*case yd of*
(*y*, *None*) ⇒
$\alpha *_C (\sum d{\in}UNIV.\ ket\ (y + d,\ Some\ d))$
$- \alpha\hat{\ }3 *_C (\sum y'{\in}UNIV.\ \sum d{\in}UNIV.\ ket\ (y',\ Some\ d))$
$+ \alpha^2 *_C (\sum d{\in}UNIV.\ ket\ (d,\ None))$
| (*y*, *Some d*) ⇒
$ket\ (y + d,\ Some\ d)$
$+ \alpha *_C ket\ (y + d,\ None)$
$- \alpha\hat{\ }3 *_C (\sum y'{\in}UNIV.\ ket\ (y',\ None))$
$- \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y + d',\ Some\ d'))$
$- \alpha^2 *_C (\sum d'{\in}UNIV.\ ket\ (y + d,\ Some\ d'))$
$+ \alpha\hat{\ }4 *_C (\sum y'{\in}UNIV.\ \sum d'{\in}UNIV.\ ket\ (y',\ Some\ d')))$›
apply (*cases yd*, *rename-tac y d*) **apply** (*case-tac d*)
apply (*simp-all add*:)
apply (*subst query1′-None*)
apply *simp*
apply (*subst query1′-Some*)
by *simp*

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*›

by (*auto simp*: *query-def register-unitary intro*!: *unitary-cblinfun-compose*)

lemma *norm-query*[*simp*]: ‹*norm query* = *1*›
using *norm-isometry unitary-isometry unitary-query* **by** *blast*

lemma *norm-query'*[*simp*]: ‹*norm query'* = *1*›
unfolding *query'-def*
apply (*subst norm-isometry-compose'*)
apply (*subst register-adjoint*[*OF register-3-3*, *symmetric*])
apply (*rule register-isometry*[*OF register-3-3*])
apply *simp*
apply (*subst norm-isometry-compose*)
apply (*rule register-isometry*[*OF register-3-3*])
apply *simp*
by *simp*

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* $\otimes_s$ ψ) = *ket x* $\otimes_s$ ((*Fst*; *Snd o function-at x*) *standard-query1* $*_V$ ψ))›
shows ‹*query* = *controlled-op* (λx. (*Fst*; *Snd o function-at x*) *query1*)›
proof −
have ‹*query* $*_V$ *ket x* $\otimes_s$ ψ = *controlled-op* (λx. (*Fst*;*Snd* ∘ *function-at x*) *query1*) $*_V$ *ket x* $\otimes_s$ ψ› **for** *x* ψ
proof −
have *aux*: ‹(*Snd* ((*Fst*;*Snd* ∘ *function-at x*) *Q*) o_{CL} *reg-3-3* (*apply-every M R*) :: ($'x \times 'y \times ('x \rightharpoonup 'y)$) *update*)
= *reg-3-3* (*apply-every M R*) o_{CL} *Snd* ((*Fst*;*Snd* ∘ *function-at x*) *Q*)›
if ‹$x \notin M$› **for** *M* **and** *Q* :: ‹($'y \times 'y$ *option*) *update*› **and** *R*
using *finite*[*of M*] *that*
proof *induction*
case *empty*
show *?case*
by *simp*
next
case (*insert y F*)
have ‹(*Snd* ((*Fst*;*Snd* ∘ *function-at x*) *Q*) o_{CL} *reg-3-3* (*apply-every* (*insert y F*) *R*) :: ($'x \times 'y \times ('x \rightharpoonup 'y)$) *update*) =
((*Snd o* (*Fst*;*Snd* ∘ *function-at x*)) *Q* o_{CL} (*reg-3-3 o function-at y*) (*R y*)) o_{CL} *reg-3-3* (*apply-every F R*)›
by (*simp add*: *apply-every-insert insert register-mult*[*of reg-3-3*, *symmetric*] *cblinfun-compose-assoc*)
also have ‹. . . = (*reg-3-3* ∘ *function-at y*) (*R y*) o_{CL} ((*Snd* ((*Fst*;*Snd* ∘ *function-at x*) *Q*)) o_{CL} *reg-3-3* (*apply-every F R*))›
apply (*subst swap-registers*[*of* ‹*Snd o* -› ‹*reg-3-3 o* -›])
using *insert* **apply** (*simp add*: *reg-3-3-def add*: *comp-assoc*)
by (*simp add*: *cblinfun-compose-assoc*)
also have ‹. . . = ((*reg-3-3* ∘ *function-at y*) (*R y*) o_{CL} *reg-3-3* (*apply-every F R*)) o_{CL} *Snd* ((*Fst*;*Snd* ∘ *function-at x*) *Q*)›
apply (*subst insert.IH*)
using *insert* **by** (*auto simp*: *cblinfun-compose-assoc*)

also have ‹... = (*reg-3-3* (*apply-every* (*insert y F*) *R*)) o_{CL} *Snd* ((*Fst*;*Snd* ∘ *function-at x*) *Q*)›
by (*simp add*: *apply-every-insert insert register-mult*[*of reg-3-3*, *symmetric*] *cblinfun-compose-assoc*)
finally show *?case*
by −
qed

have ‹*query* $*_V$ (*ket x* $\otimes_s$ ψ) = *reg-3-3 compress* $*_V$ *standard-query* $*_V$ *reg-3-3 compress* $*_V$ (*ket x* $\otimes_s$ ψ)›
by (*simp add*: *query-def*)
also have ‹... = *reg-3-3 compress* $*_V$
standard-query $*_V$ (*ket x* $\otimes_s$ *Snd compress* $*_V$ ψ)›
apply (*rule arg-cong*[**where** *f*=‹λ*x*. - $*_V$ - $*_V$ *x*›])
by (*auto simp*: *reg-3-3-def*)
also have ‹... = *reg-3-3 compress* $*_V$
(*ket x* $\otimes_s$ (((*Fst*; *Snd o function-at x*) *standard-query1* $*_V$ *Snd compress* $*_V$ ψ)))›
by (*simp add*: *standard-query-ket*)
also have ‹... = *reg-3-3 compress* $*_V$
(*Snd* ((*Fst*; *Snd o function-at x*) *standard-query1*)) $*_V$ (*ket x* $\otimes_s$ *Snd compress* $*_V$ ψ)›
by *auto*
also have ‹... = *reg-3-3 compress* $*_V$
(*Snd* ((*Fst*; *Snd o function-at x*) *standard-query1*)) $*_V$ *reg-3-3 compress* $*_V$ (*ket x* $\otimes_s$ ψ)›
apply (*rule arg-cong*[**where** *f*=‹λ*x*. - $*_V$ - $*_V$ *x*›])
by (*auto simp*: *reg-3-3-def*)
also have ‹... = (*reg-3-3 compress* o_{CL} (*Snd* ((*Fst*; *Snd o function-at x*) *standard-query1*)) o_{CL} *reg-3-3 compress*) $*_V$ (*ket x* $\otimes_s$ ψ)›
by *auto*
also have ‹... = (*reg-3-3* (*function-at x compress1*) o_{CL} (*Snd* ((*Fst*; *Snd o function-at x*) *standard-query1*)) o_{CL} *reg-3-3* (*function-at x compress1*)) $*_V$ (*ket x* $\otimes_s$ ψ)›
(**is** ‹*?lhs* $*_V$ - = *?rhs* $*_V$ -›)
proof −
have [*simp*]: ‹*insert x* (− {*x*}) = *UNIV*› **for** *x* :: ′*x*
by *auto*
have ‹*?lhs* = *reg-3-3* (*apply-every* ({*x*} ∪ −{*x*}) (λ-. *compress1*))
o_{CL} *Snd* ((*Fst*;*Snd* ∘ *function-at x*) *standard-query1*)
o_{CL} *reg-3-3* (*apply-every* (−{*x*} ∪ {*x*}) (λ-. *compress1*))›
by (*simp add*: *compress-def*)
also have ‹... = *reg-3-3* (*function-at x compress1*) o_{CL} *reg-3-3* (*apply-every* (− {*x*}) (λ-. *compress1*))
o_{CL} (*Snd* ((*Fst*;*Snd* ∘ *function-at x*) *standard-query1*) o_{CL} *reg-3-3* (*apply-every* (− {*x*}) (λ-. *compress1*)))
o_{CL} *reg-3-3* (*function-at x compress1*)›
apply (*subst apply-every-split*[*symmetric*], *simp*)
apply (*subst apply-every-split*[*symmetric*], *simp*)
by (*simp add*: *register-mult cblinfun-compose-assoc*)
also have ‹... = *reg-3-3* (*function-at x compress1*)
o_{CL} (*reg-3-3* (*apply-every* (− {*x*}) (λ-. *compress1*)) o_{CL} *reg-3-3* (*apply-every* (− {*x*}) (λ-. *compress1*)))
o_{CL} *Snd* ((*Fst*;*Snd* ∘ *function-at x*) *standard-query1*)
o_{CL} *reg-3-3* (*function-at x compress1*)›
apply (*subst aux*)
by (*auto simp add*: *cblinfun-compose-assoc*)
also have ‹... = *reg-3-3* (*function-at x compress1*)
o_{CL} (*reg-3-3* (*apply-every* (− {*x*}) (λ-. *compress1* o_{CL} *compress1*)))
o_{CL} *Snd* ((*Fst*;*Snd* ∘ *function-at x*) *standard-query1*)
o_{CL} *reg-3-3* (*function-at x compress1*)›

by (*simp add*: *register-mult*[*of reg-3-3*] *apply-every-mult*)
also have ‹. . . = *reg-3-3* (*function-at x compress1*)
o_{CL} *Snd* ((*Fst*;*Snd* ∘ *function-at x*) *standard-query1*)
o_{CL} *reg-3-3* (*function-at x compress1*)›
by (*simp add*: *compress1-square*)
finally show *?thesis*
by *auto*
qed
also have ‹. . . = *ket x* $\otimes_s$ ((*Snd* (*function-at x compress1*) o_{CL} ((*Fst*; *Snd o function-at x*) *standard-query1*) o_{CL} *Snd* (*function-at x compress1*)) $*_V$ ψ)›
by (*simp add*: *reg-3-3-def*)
also have ‹. . . = *controlled-op* (λ*x*. *Snd* (*function-at x compress1*) o_{CL} ((*Fst*; *Snd o function-at x*) *standard-query1*) o_{CL} *Snd* (*function-at x compress1*)) $*_V$
(*ket x* $\otimes_s$ ψ)›
by *simp*
also have ‹. . . = *controlled-op* (λ*x*. (*Fst*; *Snd o function-at x*) *query1*) (*ket x* $\otimes_s$ ψ)›
by (*auto simp*: *query1-def register-mult*[*symmetric*] *register-pair-Snd*[*unfolded o-def*, *THEN fun-cong*])
finally show *?thesis*
by −
qed

from *this*[*of* - ‹*ket* -›]
show *?thesis*
by (*auto intro*!: *equal-ket simp*: *tensor-ell2-ket*)
qed

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* (λ*x*. (*Fst*; *Snd o function-at x*) *query1*)›
using *query-def query1-def standard-query-ket* **by** (*rule query-local-generic*)

lemma *query'-local*: ‹*query'* = *controlled-op* (λ*x*. (*Fst*; *Snd o function-at x*) *query1'*)›
using *query'-def query1'-def standard-query'-ket* **by** (*rule query-local-generic*)

lemma (**in** *compressed-oracle*) *standard-query-compress*: ‹*standard-query* o_{CL} *reg-3-3 compress* = *reg-3-3 compress* o_{CL} *query*›
by (*simp add*: *query-def register-mult compress-selfinverse flip*: *cblinfun-compose-assoc*)

lemma (**in** *compressed-oracle*) *standard-query'-compress*: ‹*standard-query'* o_{CL} *reg-3-3 compress* = *reg-3-3 compress* o_{CL} *query'*›
by (*simp add*: *query'-def register-mult compress-selfinverse flip*: *cblinfun-compose-assoc*)

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* ((λ(*x*,*y*). (*x*,*y* + *h x*)) ‘ *I*)›
  by (*auto intro*!: *arg-cong*[**where** *f*=‹λ*x*. *ccspan* (*x* ‘ *I*)›] *simp add*: *ket-invariant-def cblinfun-image-ccspan image-image function-oracle-apply*)

lemma *function-oracle-Snd-ket-invariant*: ‹*Snd* (*function-oracle h*) $*_S$ *ket-invariant I* = *ket-invariant* ((λ(*w*,*x*,*y*). (*w*,*x*,*y*+*h x*)) ‘ *I*)›
  by (*auto intro*!: *ext arg-cong*[**where** *f*=‹λ*x*. *ccspan* (*x* ‘ *I*)›]
    simp add: *Snd-def ket-invariant-def cblinfun-image-ccspan image-image function-oracle-apply tensor-op-ket tensor-ell2-ket*)

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* (λ*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*: ‹⋀*z*. *z*∈*M* ⟹ *K z* ≤ *lift-invariant X* (*ket-invariant* {*x z*})›
  assumes *pres-I1*: ‹⋀*z*. *z*∈*M* ⟹ *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*›
proof (*rule inv-split-reg*[**where** *?U1.0*=‹λ-. *query1*› **and** *?I1.0*=*I1* **and** *?J1.0*=*J1*
    and *Y*=‹λ*z*. (*Y*;*H o function-at* (*x z*))› **and** *K*=*K*])
  show ‹(*Y*;*H* ∘ *function-at* (*x z*)) *query1* $*_V$ *ψ* = *U* $*_V$ *ψ*›
    if ‹*z*∈*M*› **and** ‹*ψ* ∈ *space-as-set* (*K z*)› **for** *ψ z*

proof −
  from *that*(*2*) *XK*[*OF* ‹*z*∈*M*›] **have** ‹$\psi \in$ *space-as-set* (*lift-invariant X* (*ket-invariant* {*x z*}))›
    using *less-eq-ccsubspace.rep-eq* **by** *blast*
  then have ψx: ‹ψ = *X* (*Proj* (*ket-invariant* {*x z*})) $*_V$ ψ›
    by (*metis Proj-lift-invariant Proj-fixes-image* ‹*compatible X Y*› *compatible-register1*)
  have ‹*U* $*_V$ ψ = (*X*;(*Y*;*H*)) *query* $*_V$ ψ›
    by (*simp add*: *U*)
  also have ‹. . . = (*X*;(*Y*;*H*)) (*controlled-op* (λ*x*. (*Fst*;*Snd* ∘ *function-at x*) *query1*)) $*_V$ ψ›
    by (*simp add*: *query-local*)
  also have ‹. . . = (*X*;(*Y*;*H*)) (*controlled-op* (λ*x*. (*Fst*;*Snd* ∘ *function-at x*) *query1*) o_{CL} *Fst* (*selfbutter* (*ket* (*x z*)))) $*_V$ ψ›
    by (*simp add*: *register-pair-apply Fst-def flip*: *register-mult Proj-ket-invariant-butterfly* ψx)
  also have ‹. . . = (*X*;(*Y*;*H*)) (*Snd* ((*Fst*;*Snd* ∘ *function-at* (*x z*)) *query1*) o_{CL} *Fst* (*selfbutter* (*ket* (*x z*)))) $*_V$ ψ›
    by (*simp add*: *controlled-op-comp-butter*)
  also have ‹. . . = (*X*;(*Y*;*H*)) (*Snd* ((*Fst*;*Snd* ∘ *function-at* (*x z*)) *query1*)) $*_V$ ψ›
    by (*simp add*: *register-pair-apply Fst-def flip*: *register-mult Proj-ket-invariant-butterfly* ψx)
  also have ‹. . . = (((*X*;(*Y*;*H*)) *o Snd o* (*Fst*;*Snd* ∘ *function-at* (*x z*))) *query1*) $*_V$ ψ›
    by *auto*
  also have ‹. . . = (*Y*;*H* ∘ *function-at* (*x z*)) *query1* $*_V$ ψ›
    by (*simp add*: *register-pair-Snd register-pair-Fst flip*: *register-comp-pair comp-assoc*)
  finally show *?thesis*
    by *simp*
qed
from *pres-I1* **show** ‹*preserves query1* (*I1 z*) (*J1 z*) ε› **if** ‹*z*∈*M*› **for** *z*
  using *that* **by** −
from *I-leq*
show ‹*I* ≤ (*SUP z*∈*M*. *K z* ⊓ *lift-invariant* (*Y*;*H o function-at* (*x z*)) (*I1 z*))›
  by −
from *J-geq*
show ‹*J* ≥ *K z* ⊓ *lift-invariant* (*Y*;*H o function-at* (*x z*)) (*J1 z*)› **if** ‹*z*∈*M*› **for** *z*
  using *that* **by** −
show ‹*compatible-register-invariant* (*Y*;*H* ∘ *function-at* (*x z*)) (*K z*)› **if** ‹*z*∈*M*› **for** *z*
  using *YK*[*OF that*] *HK*[*OF that*] **by** (*rule compatible-register-invariant-pair*)
from *orthoK*
show ‹*orthogonal-spaces* (*K z*) (*K z′*)› **if** ‹*z*∈*M*› ‹*z′*∈*M*› ‹*z* ≠ *z′*› **for** *z z′*
  using *that* **by** −
show ‹*register* (*Y*;*H* ∘ *function-at* (*x z*))› **for** *z*
  by *simp*
from *assms* **show** ‹ε ≥ *0*›
  by −
from *assms* **show** ‹*finite M*›
  by *metis*
qed

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*)›
  by (*auto intro*!: *simp*: *num-queries-def num-queries′-def*)

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

(*num-queries′ q*)›
  by (*auto simp*: *ket-invariant-tensor num-queries-num-queries′ simp flip*: *ket-invariant-UNIV*)

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*›
proof −
  define *K* **where** ‹*K x* = *ket-invariant* {(*x*::′*x*,*y*::′*y*,*D*::′*x*⇀′*y*) | *y D*. *card* (*dom D* − {*x*}) ≤ *q*}› **for** *x*
  define *Kd* **where** ‹*Kd x D0* = *ket-invariant* {(*x*::′*x*,*y*::′*y*,*D*::′*x*⇀′*y*) | *y D*. (∀ *x*′≠*x*. *D x*′ = *D0 x*′)}› **for** *x D0*
  have *K*: ‹*K x* = (*SUP D0*∈{*D0*. *D0 x* = *None* ∧ *card* (*dom D0* − {*x*}) ≤ *q*}. *Kd x D0*)› **for** *x*
  proof −
    have *aux1*: ‹*card* (*dom D* − {*x*}) ≤ *q* ⟹
        ∃ *D*′. *D*′ *x* = *None* ∧ *card* (*dom D*′ − {*x*}) ≤ *q* ∧ (∀ *x*′. *x*′ ≠ *x* ⟶ *D x*′ = *D*′ *x*′)› **for** *D*
      apply (*rule exI*[*of* - ‹*D*(*x*:=*None*)›])
      by *auto*
    have *aux2*: ‹*D*′ *x* = *None* ⟹
        card (*dom D*′ − {*x*}) ≤ *q* ⟹ ∀ *x*′. *x*′ ≠ *x* ⟶ *D x*′ = *D*′ *x*′ ⟹ *card* (*dom D* − {*x*}) ≤ *q*› **for** *D*′ *D*
      by (*smt* (*verit*) *DiffE Diff-empty card-mono domIff dom-minus dual-order.trans finite-class.finite-code singleton-iff subsetI*)
    show *?thesis*
      by (*auto intro*!: *aux1 aux2 simp add*: *K-def Kd-def simp flip*: *ket-invariant-SUP*)
  qed
  define *Kdx* **where** ‹*Kdx x D0 x*′ = *ket-invariant* {(*x*::′*x*,*y*::′*y*,*D*::′*x*⇀′*y*) | *y D*. *D x*′ = *D0 x*′}› **for** *D0 x*′ *x*
  have *Kd*: ‹*Kd x D0* = (*INF x*′∈−{*x*}. *Kdx x D0 x*′)› **for** *x D0*
    unfolding *Kd-def Kdx-def*
    apply (*subst ket-invariant-INF*[*symmetric*])
    apply (*rule arg-cong*[**where** *f*=*ket-invariant*])
    by *auto*
  have *Kdx*: ‹*Kdx x D0 x*′ = *lift-invariant reg-1-3* (*ket-invariant* {*x*}) ⊓ *lift-invariant* (*reg-3-3 o function-at x*′) (*ket-invariant* {*D0 x*′})› **for** *D0 x*′ *x*
    unfolding *Kdx-def reg-3-3-def reg-1-3-def*
    apply (*simp add*: *lift-invariant-comp*)
    apply (*subst lift-invariant-function-at-ket-inv*)
    apply (*subst lift-Snd-ket-inv*)
    apply (*subst lift-Snd-ket-inv*)
    apply (*subst lift-Fst-ket-inv*)
    apply (*subst ket-invariant-inter*)
    apply (*rule arg-cong*[**where** *f*=*ket-invariant*])
    by *auto*

  show *?thesis*
  proof (*rule inv-split-reg-query-generic*[**where** *X*=‹*reg-1-3*› **and** *Y*=‹*reg-2-3*› **and** *H*=‹*reg-3-3*› **and** *K*=*K*
      and *x*=‹λ*x*. *x*› **and** *?I1.0*=‹λ-. ⊤› **and** *?J1.0*=‹λ-. ⊤›, *OF query-local*])
    show ‹*query* = (*reg-1-3*;(*reg-2-3*;*reg-3-3*)) *query*›
      by (*auto simp add*: *pair-Fst-Snd reg-1-3-def reg-2-3-def reg-3-3-def*)
    show ‹*compatible reg-1-3 reg-2-3*› ‹*compatible reg-1-3 reg-3-3*› ‹*compatible reg-2-3 reg-3-3*›
      by *simp-all*

    show ‹*compatible-register-invariant reg-2-3* (*K x*)› **for** *x*
      unfolding *K Kd Kdx*
      apply (*rule compatible-register-invariant-SUP*, *simp*)
      apply (*rule compatible-register-invariant-INF*, *simp*)
      apply (*rule compatible-register-invariant-inter*, *simp*)
       apply (*rule compatible-register-invariant-compatible-register*)
       apply *simp*
      apply (*rule compatible-register-invariant-compatible-register*)
      by *simp*
    show ‹*compatible-register-invariant* (*reg-3-3 o function-at x*) (*K x*)› **for** *x*
      unfolding *K Kd Kdx*
      apply (*rule compatible-register-invariant-SUP*, *simp*)
      apply (*rule compatible-register-invariant-INF*, *simp*)
      apply (*rule compatible-register-invariant-inter*, *simp*)
       apply (*rule compatible-register-invariant-compatible-register*)
       apply *simp*
      apply (*rule compatible-register-invariant-compatible-register*)
      by *simp*
    show ‹*ket-invariant* (*num-queries q*)
      $\leq$ (*SUP x*. *K x* $\sqcap$ *lift-invariant* (*reg-2-3*;*reg-3-3* $\circ$ *function-at x*) $\top$)›
      by (*auto intro*!: *card-Diff1-le*[*THEN order-trans*]
          simp: *K-def lift-Fst-ket-inv reg-1-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*]
          num-queries-def)
    have $*$: ‹*card* (*dom D*) $\leq$ *card* (*dom D* $-$ $\{x\}$) + *1*› **for** *D x*
     by (*metis One-nat-def card.empty card.insert diff-card-le-card-Diff empty-not-insert finite.intros*(*1*)
          finite-insert insert-absorb le-diff-conv)
    show ‹*K x* $\sqcap$ *lift-invariant* (*reg-2-3*;*reg-3-3* $\circ$ *function-at x*) $\top$
      $\leq$ *ket-invariant* (*num-queries* (*q* + *1*))› **for** *x*
      by (*auto intro*!: $*$[*THEN order-trans*]
        simp add: *num-queries-def K-def lift-Fst-ket-inv reg-1-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*])
    show ‹*preserves query1* $\top$ $\top$ *0*›
      by *simp*
    show ‹*orthogonal-spaces* (*K x*) (*K x'*)› **if** ‹$x \neq x'$› **for** *x x'*
      unfolding *K-def orthogonal-spaces-ket* **using** *that* **by** *auto*
    show ‹*K x* $\leq$ *lift-invariant reg-1-3* (*ket-invariant* $\{x\}$)› **for** *x*
      by (*auto simp add*: *K Kd-def reg-1-3-def lift-inv-prod' lift-Fst-ket-inv*
          simp flip: *ket-invariant-SUP*)
    show ‹*0* $\leq$ (*0*::*real*)›
      by *auto*
    show ‹*finite* (*UNIV*::*'x set*)›
      by *simp*
  qed
qed

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›
proof (*rule preservesI′*)
  show ‹ε ≥ *0*›
    using - ε **apply** (*rule order.trans*)
    by (*simp add*: *preserve-query1-bound-def*)
  fix ψ :: ‹(′*y* × ′*y option*) *ell2*›
  assume ψ′: ‹ψ ∈ *space-as-set* (*ket-invariant* (*UNIV* × *I*))›
  assume ‹*norm* ψ = *1*›

  define *I′ J′* **where** ‹*I′* = *Some* −‘ *I*› **and** ‹*J′* = *Some* −‘ *J*›
  from ψ′ **have** ψ: ‹ψ ∈ *space-as-set* (*ket-invariant* (*UNIV* × ((*Some* ‘ *I′* ∪ {*None*}))))›
    using *I′-def less-eq-ccsubspace.rep-eq* **by** *fastforce*
  have [*simp*]: ‹*I′* ⊆ *J′*›
    using *I′-def J′-def IJ* **by** *blast*

  define β **where** ‹β = *Rep-ell2* ψ›
  then have β: ‹ψ = ($\sum$ *yd*∈*UNIV*×(*Some* ‘ *I′* ∪ {*None*}). β *yd* $*_C$ *ket yd*)›
    using *ell2-sum-ket-ket-invariant*[*OF* ψ] **by** *auto*
  have β*bound*: ‹($\sum$ *yd*∈*UNIV*×(*Some* ‘ *I′* ∪ {*None*}). (*cmod* (β *yd*))^2) ≤ *1*› (**is** ‹*?lhs* ≤ *1*›)
    apply (*subgoal-tac* ‹(*norm* ψ)^2 = *?lhs*›)
     apply (*simp add*: ‹*norm* ψ = *1*›)
    by (*simp add*: β *pythagorean-theorem-sum*)
  have β*None0*: ‹β (*y*,*None*) = *0*› **if** ‹*None* ∉ *I*› **for** *y*
    using ψ′ *that* **by** (*simp add*: β*-def ket-invariant-Rep-ell2*)

  have [*simp*]: ‹*Some* −‘ *insert None X* = *Some* −‘ *X*› **for** *X* :: ‹′*z option set*›
    by *auto*
  have [*simp*]: ‹*Some* −‘ *Some* ‘ *X* = *X*› **for** *X* :: ‹′*z set*›
    by *auto*
  have [*simp*]: ‹*Some x* ∈ *J* ⟷ *x* ∈ *J′*› **for** *x*
    by (*simp add*: *J′-def*)
  have [*simp*]: ‹*x* ∈ *I′* ⟹ *x* ∈ *J′*› **for** *x*
    using ‹*I′* ⊆ *J′*› **by** *blast*
  have [*simp*]: ‹($\sum$ *x*∈*X*. *if x* ∉ *Y then f x else 0*) = ($\sum$ *x*∈*X*−*Y*. *f x*)› **if** ‹*finite X*› **for** *f* :: ‹′*y* ⇒ ′*z*::*ab-group-add*› **and** *X Y*
    apply (*rule sum.mono-neutral-cong-right*)
    using *that* **by** *auto*
  have [*simp*]: ‹($\sum$ *x*∈*X*. $\sum$ *y*∈*Y*. *if x* ∉ *X′ then f x y else 0*) = ($\sum$ *x*∈*X*−*X′*. $\sum$ *y*∈*Y*. *f x y*)› **if** ‹*finite X*›
    for *f* :: ‹′*x* ⇒ ′*y* ⇒ ′*z*::*ab-group-add*› **and** *X X′ Y*
    apply (*rule sum.mono-neutral-cong-right*)
    using *that* **by** *auto*
  have [*simp*]: ‹β *yd* $*_C$ *a* $*_C$ *b* = *a* $*_C$ β *yd* $*_C$ *b*› **for** *yd a* **and** *b* :: ‹′*z*::*complex-vector*›
    by *auto*

have [*simp*]: ‹*cmod* α = *inverse* (*sqrt* *N*)› ‹*cmod* (α^2) = *inverse* *N*› ‹*cmod* (α^*3*) = *inverse* (*N* ∗ *sqrt* *N*)› ‹*cmod* (α^*4*) = *inverse* (N^2)›
by (*auto simp*: *power4-eq-xxxx power2-eq-square norm-mult numeral-3-eq-3* α*-def inverse-eq-divide norm-divide norm-power power-one-over*)
have [*simp*]: ‹*card* (*Some* ‘ *I*′) ≤ b_i›
by (*metis I*′*-def* b_i *card-image inj-Some*)
have *bound-J*′[*simp*]: ‹*card* (*Some* ‘ (− *J*′)) ≤ b_{j0}›
using b_{j0} **unfolding** *J*′*-def* **by** (*simp add*: *card-image*)

define φ **and** *PJ* :: ‹(′*y* × ′*y option*) *update*› **where**
‹φ = *query1* $*_V$ ψ› **and** ‹*PJ* = *Proj* (*ket-invariant* (*UNIV* × −*J*))›
have [*simp*]: ‹*PJ* $*_V$ *ket* (*x*,*y*) = (*if y*∈−*J then ket* (*x*,*y*) *else 0*)› **for** *x y*
by (*simp add*: *Proj-ket-invariant-ket PJ-def*)
have *P0*φ: ‹*PJ* $*_V$ φ =
α^*4* $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d*∈*I*′. $\sum$ *y*′∈*UNIV*. $\sum$ *d*′∈− *J*′. β (*y*, *Some d*) $*_C$ *ket* (*y*′, *Some d*′)) −
α^2 $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d*∈*I*′. $\sum$ *d*′∈− *J*′. β (*y*, *Some d*) $*_C$ *ket* (*y* + *d*, *Some d*′)) −
α^2 $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d*∈*I*′. $\sum$ *d*′∈− *J*′. β (*y*, *Some d*) $*_C$ *ket* (*y* + *d*′, *Some d*′)) +
α^2 $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d*∈*I*′. $\sum$ *d*′∈− *J*′. β (*y*, *Some d*) $*_C$ *ket* (*y*, *Some d*′)) +
α $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d*′∈− *J*′. β (*y*, *None*) $*_C$ *ket* (*y* + *d*′, *Some d*′)) −
α^*3* $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *y*′∈*UNIV*. $\sum$ *d*′∈− *J*′. β (*y*, *None*) $*_C$ *ket* (*y*′, *Some d*′))›
(**is** ‹- = *?t1* − *?t2* − *?t3* + *?t4* + *?t5* − *?t6*›)
by (*simp add*: φ*-def* β *query1 option-sum-split vimage-Compl*
cblinfun.add-right cblinfun.diff-right if-distrib Compl-eq-Diff-UNIV
vimage-singleton-inj sum-sum-if-eq sum.distrib scaleC-diff-right scaleC-sum-right
sum-subtractf case-prod-beta sum.cartesian-product′ *scaleC-add-right add-diff-eq*
cblinfun.scaleC-right cblinfun.sum-right
flip: *sum.Sigma add.assoc scaleC-scaleC*
cong del: *option.case-cong if-cong*)

have *norm-t1*: ‹*norm ?t1* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
proof −
have ∗: ‹*norm* ($\sum$ *yd*∈*UNIV* × *Some* ‘ *I*′. β *yd* $*_C$ *ket y*′*d*′) ≤ *sqrt* (*N* ∗ b_i)› **for** *y*′*d*′ :: ‹′*y* × ′*y option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
by (*auto simp*: *N-def*)

have ‹*norm ?t1* = *inverse* (N^2) ∗ *norm* ($\sum$ *y*′*d*′ ∈ (*UNIV*::′*y set*) × *Some* ‘ (−*J*′). $\sum$ *yd*∈*UNIV* × *Some* ‘ *I*′. β *yd* $*_C$ *ket y*′*d*′)›
apply (*simp add*: *sum.cartesian-product*′ *sum.reindex N-def*)
apply (*subst* (*2*) *sum.swap*) **apply** (*subst* (*3*) *sum.swap*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*)
by (*rule refl*)
also have ‹… ≤ *inverse* (N^2) ∗ (*N* ∗ *sqrt* (*card* (*Some* ‘ (− *J*′))) ∗ *sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def real-sqrt-mult algebra-simps real-sqrt-mult algebra-simps*
sqrt-sqrt[*THEN extend-mult-rule*])
also have ‹… ≤ *inverse* (N^2) ∗ (*N* ∗ *sqrt* b_{j0} ∗ *sqrt* b_i)›
by (*metis bound-J*′ *linordered-field-class.inverse-nonnegative-iff-nonnegative mult.commute mult-right-mono of-nat-0-le-iff of-nat-mono real-sqrt-ge-zero real-sqrt-le-iff*)
also have ‹… ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by (*smt* (*verit*) *ab-semigroup-mult-class.mult-ac*(*1*) *divide-inverse-commute of-nat-power power2-eq-square real-divide-square-eq*)
finally show ‹*norm ?t1* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›

```
      by −
  qed

  have norm-t2: ‹norm ?t2 ≤ sqrt b_j0 ∗ sqrt b_i / N›
  proof −
    have ∗: ‹card {y. δ = fst y + the (snd y) ∧ y ∈ UNIV × Some ‘ I'} ≤ card I'› for δ
      apply (rule card-inj-on-le[where f=‹λy. the (snd y)›])
      by (auto intro!: inj-onI)
    have ∗: ‹norm (∑ yd∈UNIV × Some ‘ I'. β yd ∗_C ket (fst yd + the (snd yd), d')) ≤ sqrt b_i› for
d' :: ‹'y option›
      using - - βbound apply (rule bound-coeff-sum2)
      using ∗ I'-def b_i order.trans by auto

    have ‹norm ?t2 = inverse (real N) ∗ norm (∑ d' ∈ Some ‘ (−J'). ∑ yd∈UNIV × Some ‘ I'. β yd
∗_C ket (fst yd + the (snd yd), d'))›
      apply (simp add: sum.cartesian-product' sum.reindex N-def)
      apply (subst sum.swap) apply (subst (2) sum.swap) apply (subst (3) sum.swap)
      apply (subst sum.swap)
      by (rule refl)
    also have ‹. . . ≤ inverse (real N) ∗ (sqrt (card (Some ‘ (− J'))) ∗ sqrt b_i)›
      apply (rule mult-left-mono)
      using ∗ apply (rule norm-ortho-sum-bound)
      by (auto simp add: cinner-sum-right cinner-sum-left)
    also have ‹. . . ≤ inverse N ∗ (sqrt b_j0 ∗ sqrt b_i)›
      by (simp add: mult-right-mono N-def)
    also have ‹. . . ≤ sqrt b_j0 ∗ sqrt b_i / N›
      by (simp add: divide-inverse-commute)
    finally show ‹norm ?t2 ≤ sqrt b_j0 ∗ sqrt b_i / N›
      by −
  qed

  have norm-t3: ‹norm ?t3 ≤ sqrt b_j0 ∗ sqrt b_i / N›
  proof −
    have ∗: ‹card {y. a = fst y ∧ y ∈ UNIV × (I ∩ range Some)} ≤ card I'› for a :: 'y
      apply (rule card-inj-on-le[where f=‹λy. the (snd y)›])
      by (auto intro!: inj-onI simp: I'-def)
    have ∗: ‹norm (∑ yd∈UNIV × Some ‘ I'. β yd ∗_C ket (fst yd + the d', d')) ≤ sqrt b_i› for d' :: ‹'y
option›
      using - - βbound apply (rule bound-coeff-sum2)
      using ∗ I'-def b_i order.trans by auto

    have ‹norm ?t3 = inverse (real N) ∗ norm (∑ d' ∈ Some ‘ (−J'). ∑ yd∈UNIV × Some ‘ I'.
                        β yd ∗_C ket (fst yd + the d', d'))›
      apply (simp add: sum.cartesian-product' sum.reindex N-def)
      apply (subst sum.swap) apply (subst (2) sum.swap) apply (subst (3) sum.swap)
      apply (subst sum.swap)
      by (rule refl)
    also have ‹. . . ≤ inverse (real N) ∗ (sqrt (card (Some ‘ (− J'))) ∗ sqrt b_i)›
      apply (rule mult-left-mono)
      using ∗ apply (rule norm-ortho-sum-bound)
      by (auto simp add: cinner-sum-right cinner-sum-left)
    also have ‹. . . ≤ inverse N ∗ (sqrt b_j0 ∗ sqrt b_i)›
      by (simp add: mult-right-mono N-def)
    also have ‹. . . ≤ sqrt b_j0 ∗ sqrt b_i / N›
      by (simp add: divide-inverse-commute)
```

```
    finally show ‹norm ?t3 ≤ sqrt b_j0 * sqrt b_i / N›
      by –
  qed

  have norm-t4: ‹norm ?t4 ≤ sqrt b_j0 * sqrt b_i / N›
  proof –
    have *: ‹card {y. a = fst y ∧ y ∈ UNIV × (I ∩ range Some)} ≤ card I'› for a :: 'y
      apply (rule card-inj-on-le[where f=‹λy. the (snd y)›])
      by (auto intro!: inj-onI simp: I'-def)
    have *: ‹norm (∑yd∈UNIV × Some ‘ I'. β yd *C ket (fst yd, d')) ≤ sqrt b_i› for d' :: ‹'y option›
      using - - βbound apply (rule bound-coeff-sum2)
      using * I'-def b_i order.trans by auto

    have ‹norm ?t4 = inverse (real N) * norm (∑d' ∈ Some ‘ (−J'). ∑yd∈UNIV × Some ‘ I'.
                      β yd *C ket (fst yd, d'))›
      apply (simp add: sum.cartesian-product' sum.reindex N-def)
      apply (subst sum.swap) apply (subst (2) sum.swap) apply (subst (3) sum.swap)
      apply (subst sum.swap)
      by (rule refl)
    also have ‹... ≤ inverse (real N) * (sqrt (card (Some ‘ (− J'))) * sqrt b_i)›
      apply (rule mult-left-mono)
      using * apply (rule norm-ortho-sum-bound)
      by (auto simp add: cinner-sum-right cinner-sum-left)
    also have ‹... ≤ inverse N * (sqrt b_j0 * sqrt b_i)›
      by (simp add: mult-right-mono N-def)
    also have ‹... ≤ sqrt b_j0 * sqrt b_i / N›
      by (simp add: divide-inverse-commute)
    finally show ‹norm ?t4 ≤ sqrt b_j0 * sqrt b_i / N›
      by –
  qed

  have norm-t5: ‹norm ?t5 ≤ of-bool (None∈I) * sqrt b_j0 / sqrt N›
  proof (cases ‹None∈I›)
    case True
    have *: ‹card {y. a = fst y ∧ y ∈ UNIV × {None :: 'y option}} ≤ card {()}› for a :: 'y
      apply (rule card-inj-on-le[where f=‹λ-. undefined›])
      by (auto intro!: inj-onI)
    have *: ‹norm (∑yd∈UNIV × {None}. β yd *C ket (fst yd + the d', d')) ≤ sqrt (1::nat)› for d'
:: ‹'y option›
      using - - βbound apply (rule bound-coeff-sum2)
      using * by auto

    have ‹norm ?t5 = inverse (sqrt N) * norm (∑d' ∈ Some ‘ (−J'). ∑yd∈UNIV × {None}.
                      β yd *C ket (fst yd + the d', d'))›
      apply (simp add: sum.cartesian-product' sum.reindex N-def)
      apply (subst sum.swap)
      by (rule refl)
    also have ‹... ≤ inverse (sqrt N) * (sqrt (card (Some ‘ (− J'))))›
      apply (rule mult-left-mono)
      using * apply (rule norm-ortho-sum-bound)
      by (auto simp add: cinner-sum-right cinner-sum-left)
    also have ‹... ≤ inverse (sqrt N) * sqrt b_j0›
      by (simp add: mult-right-mono N-def)
    also have ‹... ≤ of-bool (None∈I) * sqrt b_j0 / sqrt N›
      by (simp add: True divide-inverse-commute)
```

finally show *?thesis*
by −
next
case *False*
then show *?thesis*
using β*None0* **by** *auto*
qed

have *norm-t6*: ‹*norm ?t6* $\leq$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
proof (*cases* ‹*None*$\in$*I*›)
case *True*
have $*$: ‹*norm* ($\sum$ *yd*$\in$*UNIV* $\times$ {*None*}. β *yd* $*_C$ *ket y'd'*) $\leq$ *sqrt N*› **for** *y'd'* :: ‹*'y* $\times$ *'y option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
by (*auto simp*: *N-def*)

have ‹*norm ?t6* = *inverse* (*N* $*$ *sqrt N*) $*$ *norm* ($\sum$ *y'd'* $\in$ (*UNIV*::*'y set*) $\times$ *Some* ‘ (−*J'*). $\sum$ *yd*$\in$*UNIV* $\times$ {*None*}. β *yd* $*_C$ *ket y'd'*)›
apply (*simp add*: *sum.cartesian-product' sum.reindex N-def*)
apply (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹... $\leq$ *inverse* (*N* $*$ *sqrt N*) $*$ (*N* $*$ *sqrt* (*card* (*Some* ‘ (− *J'*))))›
apply (*rule mult-left-mono*)
using $*$ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def mult.commute real-sqrt-mult vector-space-over-itself.scale-scale*)
also have ‹... $\leq$ *inverse* (*N* $*$ *sqrt N*) $*$ (*N* $*$ *sqrt* b_{j0})›
by (*simp add*: *N-def*)
also have ‹... $\leq$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
using *True* **by** (*simp add*: *divide-inverse-commute less-eq-real-def N-def*)
finally show *?thesis*
by −
next
case *False*
then show *?thesis*
using β*None0* **by** *auto*
qed

have ‹*norm* (*PJ* $*_V$ φ) $\leq$ *sqrt* b_{j0} $*$ *sqrt* b_i / *N* + *sqrt* b_{j0} $*$ *sqrt* b_i / *N* + *sqrt* b_{j0} $*$ *sqrt* b_i / *N*
+ *sqrt* b_{j0} $*$ *sqrt* b_i / *N* + *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N* + *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
unfolding *P0*φ
apply (*rule norm-triangle-le-diff norm-triangle-le*, *rule add-mono*)+
apply (*rule norm-t1*)
apply (*rule norm-t2*)
apply (*rule norm-t3*)
apply (*rule norm-t4*)
apply (*rule norm-t5*)
by (*rule norm-t6*)

also have ‹... $\leq$ *4* $*$ *sqrt* b_{j0} $*$ *sqrt* b_i / *N* + *2* $*$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
by (*simp add*: *mult.commute vector-space-over-itself.scale-left-commute*)
also have ‹... $\leq$ ε›
using ε **by** (*auto intro*!: *simp add*: *preserve-query1-bound-def*)
finally show ‹*norm* (*Proj* (− *ket-invariant* (*UNIV* $\times$ *J*)) $*_V$ φ) $\leq$ ε›

unfolding *PJ-def*
apply (*subst ket-invariant-compl*[*symmetric*])
by *simp*
qed

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*: ‹$I \subseteq J$›
assumes [*simp*]: ‹*None* ∈ *J*›
assumes b_i: ‹*card* (*Some* −‘ *I*) ≤ b_i›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) ≤ b_{j0}›
assumes ε: ‹$\varepsilon \geq$ *preserve-query1′-bound* (*None*∈*I*) b_i b_{j0}›
shows ‹*preserves-ket query1′* (*UNIV* × *I*) (*UNIV* × *J*) ε›
proof (*rule preservesI′*)
show ‹$\varepsilon \geq 0$›
using - ε **apply** (*rule order.trans*)
by (*simp add*: *preserve-query1′-bound-def*)
fix ψ :: ‹(′*y* × ′*y option*) *ell2*›
assume ψ': ‹ψ ∈ *space-as-set* (*ket-invariant* (*UNIV* × *I*))›
assume ‹*norm* ψ = *1*›

define *I′ J′* **where** ‹*I′* = *Some* −‘ *I*› **and** ‹*J′* = *Some* −‘ *J*›
from ψ' **have** ψ: ‹ψ ∈ *space-as-set* (*ket-invariant* (*UNIV* × ((*Some* ‘ *I′* ∪ {*None*}))))›
using *I′-def less-eq-ccsubspace.rep-eq* **by** *fastforce*
have [*simp*]: ‹*I′* ⊆ *J′*›
using *I′-def J′-def IJ* **by** *blast*

define β **where** ‹β = *Rep-ell2* ψ›
then have β: ‹ψ = ($\sum$ *yd*∈*UNIV*×(*Some* ‘ *I′* ∪ {*None*}). β *yd* $*_C$ *ket yd*)›
using *ell2-sum-ket-ket-invariant*[*OF* ψ] **by** *auto*
have β*bound*: ‹($\sum$ *yd*∈*UNIV*×(*Some* ‘ *I′* ∪ {*None*}). (*cmod* (β *yd*))^2) ≤ *1*› (**is** ‹*?lhs* ≤ *1*›)
apply (*subgoal-tac* ‹(*norm* ψ)^2 = *?lhs*›)
apply (*simp add*: ‹*norm* ψ = *1*›)
by (*simp add*: β *pythagorean-theorem-sum*)
have β*None0*: ‹β (*y*,*None*) = *0*› **if** ‹*None* ∉ *I*› **for** *y*
using ψ' *that* **by** (*simp add*: β*-def ket-invariant-Rep-ell2*)

have [*simp*]: ‹*Some* −‘ *insert None X* = *Some* −‘ *X*› **for** *X* :: ‹′*z option set*›
by *auto*
have [*simp*]: ‹*Some* −‘ *Some* ‘ *X* = *X*› **for** *X* :: ‹′*z set*›
by *auto*
have [*simp*]: ‹*Some x* ∈ *J* ⟷ *x* ∈ *J′*› **for** *x*
by (*simp add*: *J′-def*)
have [*simp*]: ‹*x* ∈ *I′* ⟹ *x* ∈ *J′*› **for** *x*
using ‹*I′* ⊆ *J′*› **by** *blast*
have [*simp*]: ‹($\sum$ *x*∈*X*. *if x* ∉ *Y then f x else 0*) = ($\sum$ *x*∈*X*−*Y*. *f x*)› **if** ‹*finite X*› **for** *f* :: ‹′*y* ⇒ ′*z*::*ab-group-add*› **and** *X Y*
apply (*rule sum.mono-neutral-cong-right*)
using *that* **by** *auto*
have [*simp*]: ‹($\sum$ *x*∈*X*. $\sum$ *y*∈*Y*. *if x* ∉ *X′ then f x y else 0*) = ($\sum$ *x*∈*X*−*X′*. $\sum$ *y*∈*Y*. *f x y*)› **if** ‹*finite X*›
for *f* :: ‹′*x* ⇒ ′*y* ⇒ ′*z*::*ab-group-add*› **and** *X X′ Y*
apply (*rule sum.mono-neutral-cong-right*)
using *that* **by** *auto*

have [*simp*]: ‹β *yd* $*_C$ *a* $*_C$ *b* = *a* $*_C$ β *yd* $*_C$ *b*› **for** *yd a* **and** *b* :: ‹*'z::complex-vector*›
by *auto*
have [*simp*]: ‹*cmod* α = *inverse* (*sqrt N*)› ‹*cmod* (α^2) = *inverse N*› ‹*cmod* (α^*3*) = *inverse* (*N* ∗ *sqrt N*)› ‹*cmod* (α^*4*) = *inverse* (N^2)›
by (*auto simp*: *norm-mult numeral-3-eq-3 α-def inverse-eq-divide norm-divide norm-power power-one-over power4-eq-xxxx power2-eq-square*)
have [*simp*]: ‹*card* (*Some* ‘ *I'*) ≤ b_i›
by (*metis I'-def* b_i *card-image inj-Some*)
have *bound-J'*[*simp*]: ‹*card* (*Some* ‘ (− *J'*)) ≤ b_{j0}›
using b_{j0} **unfolding** *J'-def* **by** (*simp add*: *card-image*)

define φ **and** *PJ* :: ‹(*'y* ∗ *'y option*) *update*› **where**
‹φ = *query1'* $*_V$ ψ› **and** ‹*PJ* = *Proj* (*ket-invariant* (*UNIV* × −*J*))›
have [*simp*]: ‹*PJ* $*_V$ *ket* (*x*,*y*) = (*if y*∈−*J then ket* (*x*,*y*) *else 0*)› **for** *x y*
by (*simp add*: *Proj-ket-invariant-ket PJ-def*)
have *P0φ*: ‹*PJ* $*_V$ φ =
α^*4* $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d*∈*I'*. $\sum$ *y'*∈*UNIV*. $\sum$ *d'*∈− *J'*. β (*y*, *Some d*) $*_C$ *ket* (*y'*, *Some d'*)) −
α^2 $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d*∈*I'*. $\sum$ *d'*∈− *J'*. β (*y*, *Some d*) $*_C$ *ket* (*y* + *d*, *Some d'*)) −
α^2 $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d*∈*I'*. $\sum$ *d'*∈− *J'*. β (*y*, *Some d*) $*_C$ *ket* (*y* + *d'*, *Some d'*)) +
α $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d'*∈− *J'*. β (*y*, *None*) $*_C$ *ket* (*y* + *d'*, *Some d'*)) −
α^*3* $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *y'*∈*UNIV*. $\sum$ *d'*∈− *J'*. β (*y*, *None*) $*_C$ *ket* (*y'*, *Some d'*))›
(**is** ‹- = *?t1* − *?t2* − *?t3* + *?t5* − *?t6*›)
by (*simp add*: *φ-def β query1' option-sum-split vimage-Compl cblinfun.scaleC-right cblinfun.add-right cblinfun.diff-right if-distrib Compl-eq-Diff-UNIV vimage-singleton-inj sum-sum-if-eq sum.distrib scaleC-diff-right scaleC-sum-right sum-subtractf case-prod-beta sum.cartesian-product' scaleC-add-right add-diff-eq cblinfun.sum-right*
flip: *sum.Sigma add.assoc scaleC-scaleC*
cong del: *option.case-cong if-cong*)

have *norm-t1*: ‹*norm ?t1* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
proof −
have ∗: ‹*norm* ($\sum$ *yd*∈*UNIV* × *Some* ‘ *I'*. β *yd* $*_C$ *ket y'd'*) ≤ *sqrt* (*N* ∗ b_i)› **for** *y'd'* :: ‹*'y* × *'y option*›
using - - *βbound* **apply** (*rule bound-coeff-sum2*)
by (*auto simp*: *N-def*)

have ‹*norm ?t1* = *inverse* (N^2) ∗ *norm* ($\sum$ *y'd'* ∈ (*UNIV*::*'y set*) × *Some* ‘ (−*J'*). $\sum$ *yd*∈*UNIV* × *Some* ‘ *I'*. β *yd* $*_C$ *ket y'd'*)›
apply (*simp add*: *sum.cartesian-product' sum.reindex N-def*)
apply (*subst* (*2*) *sum.swap*) **apply** (*subst* (*3*) *sum.swap*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (N^2) ∗ (*N* ∗ *sqrt* (*card* (*Some* ‘ (− *J'*))) ∗ *sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def real-sqrt-mult algebra-simps sqrt-sqrt*[*THEN extend-mult-rule*])
also have ‹. . . ≤ *inverse* (N^2) ∗ (*N* ∗ *sqrt* b_{j0} ∗ *sqrt* b_i)›
by (*metis bound-J' linordered-field-class.inverse-nonnegative-iff-nonnegative mult.commute mult-right-mono of-nat-0-le-iff of-nat-mono real-sqrt-ge-zero real-sqrt-le-iff*)
also have ‹. . . ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by (*smt* (*verit*) *ab-semigroup-mult-class.mult-ac*(*1*) *divide-inverse-commute of-nat-power power2-eq-square real-divide-square-eq*)
finally show ‹*norm ?t1* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›

by −
qed

have *norm-t2*: ‹*norm ?t2* $\leq$ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
proof −
have ∗: ‹*card* {*y*. δ = *fst y* + *the* (*snd y*) ∧ *y* ∈ *UNIV* × *Some* ‘ *I*′} ≤ *card I*′› **for** δ
apply (*rule card-inj-on-le*[**where** *f*=‹λ*y*. *the* (*snd y*)›])
by (*auto intro*!: *inj-onI*)
have ∗: ‹*norm* (∑ *yd*∈*UNIV* × *Some* ‘ *I*′. β *yd* $*_C$ *ket* (*fst yd* + *the* (*snd yd*), *d*′)) ≤ *sqrt* b_i› **for** *d*′ :: ‹′*y option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using ∗ *I*′*-def* b_i *order.trans* **by** *auto*

have ‹*norm ?t2* = *inverse* (*real N*) ∗ *norm* (∑ *d*′ ∈ *Some* ‘ (−*J*′). ∑ *yd*∈*UNIV* × *Some* ‘ *I*′. β *yd* $*_C$ *ket* (*fst yd* + *the* (*snd yd*), *d*′))›
apply (*simp add*: *sum.cartesian-product*′ *sum.reindex N-def*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*) **apply** (*subst* (*3*) *sum.swap*)
apply (*subst sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (*real N*) ∗ (*sqrt* (*card* (*Some* ‘ (− *J*′))) ∗ *sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹. . . ≤ *inverse N* ∗ (*sqrt* b_{j0} ∗ *sqrt* b_i)›
by (*simp add*: *mult-right-mono N-def*)
also have ‹. . . ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by (*simp add*: *divide-inverse-commute*)
finally show ‹*norm ?t2* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by −
qed

have *norm-t3*: ‹*norm ?t3* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
proof −
have ∗: ‹*card* {*y*. *a* = *fst y* ∧ *y* ∈ *UNIV* × (*I* ∩ *range Some*)} ≤ *card I*′› **for** *a* :: ′*y*
apply (*rule card-inj-on-le*[**where** *f*=‹λ*y*. *the* (*snd y*)›])
by (*auto intro*!: *inj-onI simp*: *I*′*-def*)
have ∗: ‹*norm* (∑ *yd*∈*UNIV* × *Some* ‘ *I*′. β *yd* $*_C$ *ket* (*fst yd* + *the d*′, *d*′)) ≤ *sqrt* b_i› **for** *d*′ :: ‹′*y option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using ∗ *I*′*-def* b_i *order.trans* **by** *auto*

have ‹*norm ?t3* = *inverse* (*real N*) ∗ *norm* (∑ *d*′ ∈ *Some* ‘ (−*J*′). ∑ *yd*∈*UNIV* × *Some* ‘ *I*′.
β *yd* $*_C$ *ket* (*fst yd* + *the d*′, *d*′))›
apply (*simp add*: *sum.cartesian-product*′ *sum.reindex N-def*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*) **apply** (*subst* (*3*) *sum.swap*)
apply (*subst sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (*real N*) ∗ (*sqrt* (*card* (*Some* ‘ (− *J*′))) ∗ *sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹. . . ≤ *inverse N* ∗ (*sqrt* b_{j0} ∗ *sqrt* b_i)›
by (*simp add*: *mult-right-mono N-def*)
also have ‹. . . ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by (*simp add*: *divide-inverse-commute*)

finally show ‹*norm ?t3* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by −
qed

have *norm-t5*: ‹*norm ?t5* ≤ *of-bool* (*None*∈*I*) ∗ *sqrt* b_{j0} / *sqrt N*›
proof (*cases* ‹*None*∈*I*›)
case *True*
have ∗: ‹*card* {*y*. *a* = *fst y* ∧ *y* ∈ *UNIV* × {*None* :: ′*y option*}} ≤ *card* {()}› **for** *a* :: ′*y*
apply (*rule card-inj-on-le*[**where** *f*=‹λ-. *undefined*›])
by (*auto intro*!: *inj-onI*)
have ∗: ‹*norm* (∑ *yd*∈*UNIV* × {*None*}. β *yd* $*_C$ *ket* (*fst yd* + *the d*′, *d*′)) ≤ *sqrt* (*1*::*nat*)› **for** *d*′ :: ‹′*y option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using ∗ **by** *auto*

have ‹*norm ?t5* = *inverse* (*sqrt N*) ∗ *norm* (∑ *d*′ ∈ *Some* ‘ (−*J*′). ∑ *yd*∈*UNIV* × {*None*}.
β *yd* $*_C$ *ket* (*fst yd* + *the d*′, *d*′))›
apply (*simp add*: *sum.cartesian-product*′ *sum.reindex N-def*)
apply (*subst sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (*sqrt N*) ∗ (*sqrt* (*card* (*Some* ‘ (− *J*′))))›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹. . . ≤ *inverse* (*sqrt N*) ∗ *sqrt* b_{j0}›
by (*simp add*: *mult-right-mono N-def*)
also have ‹. . . ≤ *of-bool* (*None*∈*I*) ∗ *sqrt* b_{j0} / *sqrt N*›
using *True* **by** (*simp add*: *divide-inverse-commute*)
finally show *?thesis*
by −
next
case *False*
then show *?thesis*
using β*None0* **by** *auto*
qed

have *norm-t6*: ‹*norm ?t6* ≤ *of-bool* (*None*∈*I*) ∗ *sqrt* b_{j0} / *sqrt N*›
proof (*cases* ‹*None*∈*I*›)
case *True*
have ∗: ‹*norm* (∑ *yd*∈*UNIV* × {*None*}. β *yd* $*_C$ *ket y*′*d*′) ≤ *sqrt N*› **for** *y*′*d*′ :: ‹′*y* × ′*y option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
by (*auto simp*: *N-def*)

have ‹*norm ?t6* = *inverse* (*N* ∗ *sqrt N*) ∗ *norm* (∑ *y*′*d*′ ∈ (*UNIV*::′*y set*) × *Some* ‘ (−*J*′). ∑ *yd*∈*UNIV* × {*None*}. β *yd* $*_C$ *ket y*′*d*′)›
apply (*simp add*: *sum.cartesian-product*′ *sum.reindex N-def*)
apply (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (*N* ∗ *sqrt N*) ∗ (*N* ∗ *sqrt* (*card* (*Some* ‘ (− *J*′))))›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def real-sqrt-mult algebra-simps sqrt-sqrt*[*THEN extend-mult-rule*])
also have ‹. . . ≤ *inverse* (*N* ∗ *sqrt N*) ∗ (*N* ∗ *sqrt* b_{j0})›
by (*simp add*: *N-def*)

also have ‹... $\leq$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
using *True* **by** (*simp add*: *divide-inverse-commute less-eq-real-def N-def*)
finally show *?thesis*
by −
next
case *False*
then show *?thesis*
using *βNone0* **by** *auto*
qed

have ‹*norm* (*PJ* $*_V$ φ) $\leq$ *sqrt* b_{j0} $*$ *sqrt* b_i / *N* + *sqrt* b_{j0} $*$ *sqrt* b_i / *N* + *sqrt* b_{j0} $*$ *sqrt* b_i / *N*
+ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N* + *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
unfolding *P0φ*
apply (*rule norm-triangle-le-diff norm-triangle-le*, *rule add-mono*)+
apply (*rule norm-t1*)
apply (*rule norm-t2*)
apply (*rule norm-t3*)
apply (*rule norm-t5*)
by (*rule norm-t6*)
also have ‹... $\leq$ *3* $*$ *sqrt* b_{j0} $*$ *sqrt* b_i / *N* + *2* $*$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
by (*simp add*: *mult.commute vector-space-over-itself.scale-left-commute*)
also have ‹... $\leq$ ε›
using ε **by** (*auto intro*!: *simp add*: *preserve-query1′-bound-def*)
finally show ‹*norm* (*Proj* (− *ket-invariant* (*UNIV* × *J*)) $*_V$ φ) $\leq$ ε›
unfolding *PJ-def*
apply (*subst ket-invariant-compl*[*symmetric*])
by *simp*
qed

lemma *preserve-query1-simplified*:
assumes ‹*I* $\subseteq$ *J*›
assumes ‹*None* $\in$ *J*›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) $\leq$ b_{j0}›
shows ‹*preserves-ket query1* (*UNIV* × *I*) (*UNIV* × *J*) (*6* $*$ *sqrt* b_{j0} / *sqrt N*)›
apply (*rule preserve-query1*[**where** b_{j0}=b_{j0} **and** b_i=*N*])
using *assms* **by** (*auto intro*!: *divide-right-mono simp*: *preserve-query1-bound-def card-mono N-def*)

lemma *preserve-query1′-simplified*:
assumes ‹*I* $\subseteq$ *J*›
assumes ‹*None* $\in$ *J*›
assumes b_{j0}: ‹*card* (− *Some* −‘ *J*) $\leq$ b_{j0}›
shows ‹*preserves-ket query1′* (*UNIV* × *I*) (*UNIV* × *J*) (*5* $*$ *sqrt* b_{j0} / *sqrt N*)›
apply (*rule preserve-query1′*[**where** b_{j0}=b_{j0} **and** b_i=*N*])
using *assms* **by** (*auto intro*!: *divide-right-mono simp*: *preserve-query1′-bound-def card-mono N-def*)

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* $(NoneI \wedge NoneJ)\ /\ sqrt\ N$›

lemma *preserve-query1-fixY*:
assumes *IJ*: ‹$I \subseteq J$›
assumes b_i: ‹*card* $(Some\ -`\ I) \leq b_i$›
assumes b_{j0}: ‹*card* $(-\ Some\ -`\ J) \leq b_{j0}$›
assumes ε: ‹$\varepsilon \geq$ *preserve-query1-fixY-bound* $(None \in I)\ (None \notin J)\ b_i\ b_{j0}$›
shows ‹*preserves-ket query1* $(\{y_0\} \times I)\ (UNIV \times J)\ \varepsilon$›
proof (*rule preservesI'*)
show ‹$\varepsilon \geq 0$›
using - ε **apply** (*rule order.trans*)
by (*simp add*: *preserve-query1-fixY-bound-def*)
fix ψ :: ‹$('y \times 'y\ option)\ ell2$›
assume ψ: ‹$\psi \in$ *space-as-set* (*ket-invariant* $(\{y_0\} \times I)$)›
assume ‹*norm* $\psi = 1$›

define $I'\ J'$ **where** ‹$I' = Some\ -`\ I$› **and** ‹$J' = Some\ -`\ J$›
then have ‹$\{y_0\} \times I \subseteq \{y_0\} \times (Some\ `\ I' \cup \{None\})$›
apply (*rule-tac Sigma-mono*)
by *auto*
with ψ **have** ψ': ‹$\psi \in$ *space-as-set* (*ket-invariant* $(\{y_0\} \times ((Some\ `\ I' \cup \{None\})))$)›
using *less-eq-ccsubspace.rep-eq ket-invariant-le* **by** *fastforce*
have [*simp*]: ‹$I' \subseteq J'$›
using *I'-def J'-def IJ* **by** *blast*

define β **where** ‹$\beta\ d =$ *Rep-ell2* $\psi\ (y_0,d)$› **for** d
then have β: ‹$\psi = (\sum d \in Some\ `\ I' \cup \{None\}.\ \beta\ d *_C ket\ (y_0,d))$›
using *ell2-sum-ket-ket-invariant*[*OF* ψ']
by (*auto simp*: *sum.cartesian-product'*)
have *βbound*: ‹$(\sum d \in (Some\ `\ I' \cup \{None\}).\ (cmod\ (\beta\ d))^2) \leq 1$› (**is** ‹*?lhs* ≤ 1›)
apply (*subgoal-tac* ‹$(norm\ \psi)^2 = ?lhs$›)
apply (*simp add*: ‹*norm* $\psi = 1$›)
by (*simp add*: β *pythagorean-theorem-sum del*: *sum.insert*)
have *βNone*: ‹*cmod* $(\beta\ None) \leq 1$›
proof −
have ‹$(cmod\ (\beta\ None))^2 = (\sum d \in \{None\}.\ (cmod\ (\beta\ d))^2)$›
by *simp*
also have ‹$\ldots \leq (\sum d \in (Some\ `\ I' \cup \{None\}).\ (cmod\ (\beta\ d))^2)$›
apply (*rule sum-mono2*) **by** *auto*
also have ‹$\ldots \leq 1$›
by (*rule βbound*)
finally show *?thesis*
by (*simp add*: *power-le-one-iff*)
qed
have *βNone0*: ‹$\beta\ None = 0$› **if** ‹$None \notin I$›
using ψ *that* **by** (*simp add*: *β-def ket-invariant-Rep-ell2*)

have [*simp*]: ‹$Some\ -`\ insert\ None\ X = Some\ -`\ X$› **for** X :: ‹$'z\ option\ set$›
by *auto*
have [*simp*]: ‹$Some\ -`\ Some\ `\ X = X$› **for** X :: ‹$'z\ set$›
by *auto*
have [*simp*]: ‹$Some\ x \in J \longleftrightarrow x \in J'$› **for** x
by (*simp add*: *J'-def*)
have [*simp*]: ‹$x \in I' \Longrightarrow x \in J'$› **for** x
using ‹$I' \subseteq J'$› **by** *blast*

have [*simp*]: ‹($\sum x{\in}X.$ *if* $x \notin Y$ *then* $f\ x$ *else* *0*) = ($\sum x{\in}X-Y.\ f\ x$)› **if** ‹*finite X*› **for** f :: ‹'*y* ⇒ '*z*::*ab-group-add*› **and** X Y
  apply (*rule sum.mono-neutral-cong-right*)
  using *that* **by** *auto*
have [*simp*]: ‹$\beta\ yd *_C a *_C b = a *_C \beta\ yd *_C b$› **for** *yd a* **and** *b* :: ‹'*z*::*complex-vector*›
  by *auto*
have [*simp*]: ‹*cmod* α = *inverse* (*sqrt N*)› ‹*cmod* (α^2) = *inverse N*› ‹*cmod* (α^*3*) = *inverse* (*N* ∗ *sqrt N*)› ‹*cmod* (α^*4*) = *inverse* (N^2)›
  by (*auto simp*: *norm-mult numeral-3-eq-3* α*-def inverse-eq-divide norm-divide norm-power power-one-over power4-eq-xxxx power2-eq-square*)
have [*simp*]: ‹*card* (*Some* ‘ *I*′) ≤ b_i›
  by (*metis I*′*-def* b_i *card-image inj-Some*)
have *bound-J*′[*simp*]: ‹*card* (*Some* ‘ (− *J*′)) ≤ b_{j0}›
  using b_{j0} **unfolding** *J*′*-def* **by** (*simp add*: *card-image*)

define φ **and** *PJ* :: ‹('*y* ∗ '*y option*) *update*› **where**
  ‹φ = *query1* $*_V$ ψ› **and** ‹*PJ* = *Proj* (*ket-invariant* (*UNIV* × −*J*))›
have [*simp*]: ‹*PJ* $*_V$ *ket* (*x*,*y*) = (*if y*∈−*J then ket* (*x*,*y*) *else 0*)› **for** *x y*
  by (*simp add*: *Proj-ket-invariant-ket PJ-def*)
have *P0*φ: ‹*PJ* $*_V$ φ =
    α^*4* $*_C$ ($\sum d{\in}I'.\ \sum y'{\in}UNIV.\ \sum d'{\in}-\ J'.\ \beta\ (Some\ d) *_C ket\ (y', Some\ d')$)
    − $\alpha^2 *_C$ ($\sum d{\in}I'.\ \sum d'{\in}-\ J'.\ \beta\ (Some\ d) *_C ket\ (y_0 + d, Some\ d')$)
    − $\alpha^2 *_C$ ($\sum d{\in}I'.\ \sum d'{\in}-\ J'.\ \beta\ (Some\ d) *_C ket\ (y_0 + d', Some\ d')$)
    + $\alpha^2 *_C$ ($\sum d{\in}I'.\ \sum d'{\in}-\ J'.\ \beta\ (Some\ d) *_C ket\ (y_0, Some\ d')$)
    + $\alpha *_C$ ($\sum d'{\in}-\ J'.\ \beta\ (None) *_C ket\ (y_0 + d', Some\ d')$)
    − α^*3* $*_C$ ($\sum y'{\in}UNIV.\ \sum d'{\in}-\ J'.\ \beta\ (None) *_C ket\ (y', Some\ d')$)
    + (*of-bool* (*None* ∉ *J*) ∗ α) $*_C$ ($\sum d{\in}I'.\ \beta\ (Some\ d) *_C ket\ (y_0 + d, None)$)
    − (*of-bool* (*None* ∉ *J*) ∗ α^*3*) $*_C$ ($\sum d{\in}I'.\ \sum y'{\in}UNIV.\ \beta\ (Some\ d) *_C ket\ (y', None)$)
    + (*of-bool* (*None* ∉ *J*) ∗ α^2) $*_C$ ($\sum y'{\in}UNIV.\ \beta\ None *_C ket\ (y', None)$)
    ›
  (**is** ‹- = *?t1* − *?t2* − *?t3* + *?t4* + *?t5* − *?t6* + *?t7* − *?t8* + *?t9*›)
  by (*simp add*: φ*-def* β *query1 option-sum-split vimage-Compl*
    cblinfun.add-right cblinfun.diff-right if-distrib Compl-eq-Diff-UNIV
    vimage-singleton-inj sum-sum-if-eq sum.distrib scaleC-diff-right scaleC-sum-right
    sum-subtractf case-prod-beta sum.cartesian-product′ *scaleC-add-right add-diff-eq*
    cblinfun.scaleC-right cblinfun.sum-right
    flip: *sum.Sigma add.assoc scaleC-scaleC*
    cong del: *option.case-cong if-cong*)

have *norm-t1*: ‹*norm ?t1* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*)›
proof −
  have ∗: ‹*norm* ($\sum d{\in}Some$ ‘ $I'.\ \beta\ d *_C ket\ y'd'$) ≤ *sqrt* b_i› **for** *y*′*d*′ :: ‹'*y* × '*y option*›
    using - - β*bound* **apply** (*rule bound-coeff-sum2*)
    by *auto*

  have ‹*norm ?t1* = *inverse* (N^2) ∗ *norm* ($\sum y'd' \in$ (*UNIV*::'*y set*) × *Some* ‘ (−*J*′). $\sum d{\in}Some$ ‘ *I*′. $\beta\ d *_C ket\ y'd'$)›
    apply (*simp add*: *sum.cartesian-product*′ *sum.reindex N-def*)
    apply (*subst* (*2*) *sum.swap*) **apply** (*subst* (*3*) *sum.swap*)
    by (*rule refl*)
  also have ‹. . . ≤ *inverse* (N^2) ∗ (*sqrt N* ∗ *sqrt* (*card* (*Some* ‘ (− *J*′))) ∗ *sqrt* b_i)›
    apply (*rule mult-left-mono*)
    using ∗ **apply** (*rule norm-ortho-sum-bound*)
    by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def real-sqrt-mult*)
  also have ‹. . . ≤ *inverse* (N^2) ∗ (*sqrt N* ∗ *sqrt* b_{j0} ∗ *sqrt* b_i)›

by (*metis bound-J′ linordered-field-class.inverse-nonnegative-iff-nonnegative mult.commute mult-right-mono of-nat-0-le-iff of-nat-mono real-sqrt-ge-zero real-sqrt-le-iff*)
also have ‹... $\leq$ *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*)›
by (*smt* (*verit, del-insts*) *divide-divide-eq-left divide-inverse mult.commute of-nat-0-le-iff of-nat-power power2-eq-square real-divide-square-eq real-sqrt-pow2 times-divide-eq-left*)
finally show ‹*norm ?t1* $\leq$ *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*)›
by −
qed

have *norm-t2*: ‹*norm ?t2* $\leq$ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
proof −
have ∗: ‹*card* {*d*. δ = *the d* ∧ *d* ∈ *Some* ‘ *I′*} $\leq$ *card I′*› **for** δ
apply (*rule card-inj-on-le*[**where** *f*=*the*])
by (*auto intro*!: *inj-onI*)
have ∗: ‹*norm* (∑ *d*∈*Some* ‘ *I′*. β *d* $*_C$ *ket* (y_0 + *the d*, *d′*)) $\leq$ *sqrt* b_i› **for** *d′* :: ‹′*y option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using ∗ *I′-def* b_i *order.trans* **by** *auto*

have ‹*norm ?t2* = *inverse* (*real N*) ∗ *norm* (∑ *d′* ∈ *Some* ‘ (−*J′*). ∑ *d*∈*Some* ‘ *I′*. β *d* $*_C$ *ket* (y_0 + *the d*, *d′*))›
apply (*simp add*: *sum.cartesian-product′ sum.reindex N-def*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹... $\leq$ *inverse* (*real N*) ∗ (*sqrt* (*card* (*Some* ‘ (− *J′*))) ∗ *sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹... $\leq$ *inverse N* ∗ (*sqrt* b_{j0} ∗ *sqrt* b_i)›
by (*simp add*: *mult-right-mono N-def*)
also have ‹... $\leq$ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by (*simp add*: *divide-inverse-commute*)
finally show ‹*norm ?t2* $\leq$ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by −
qed

have *norm-t3*: ‹*norm ?t3* $\leq$ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
proof −
have *aux*: ‹*I′* = *Some* −‘ *I* ⟹ *card* (*Some* −‘ *I*) $\leq$ b_i ⟹ *Some x* ∈ *I* ⟹ *card* {*y* ∈ *I*. *y* ∈ *range Some*} $\leq$ b_i› **for** *x*
by (*smt* (*verit, ccfv-SIG*) *Collect-cong Int-iff card-image image-vimage-eq inf-set-def inj-Some mem-Collect-eq*)
have ∗: ‹*norm* (∑ *d*∈*Some* ‘ *I′*. β *d* $*_C$ *ket* (y_0 + *the d′*, *d′*)) $\leq$ *sqrt* b_i› **for** *d′* :: ‹′*y option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using *I′-def* b_i *aux* **by** *auto*
have ‹*norm ?t3* = *inverse* (*real N*) ∗ *norm* (∑ *d′* ∈ *Some* ‘ (−*J′*). ∑ *d*∈*Some* ‘ *I′*.
β *d* $*_C$ *ket* (y_0 + *the d′*, *d′*))›
apply (*simp add*: *sum.reindex N-def*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹... $\leq$ *inverse* (*real N*) ∗ (*sqrt* (*card* (*Some* ‘ (− *J′*))) ∗ *sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹... $\leq$ *inverse N* ∗ (*sqrt* b_{j0} ∗ *sqrt* b_i)›
by (*simp add*: *mult-right-mono N-def*)

also have ‹... $\leq$ *sqrt* b_{j0} $*$ *sqrt* b_i / *N*›
by (*simp add*: *divide-inverse-commute*)
finally show ‹*norm ?t3* $\leq$ *sqrt* b_{j0} $*$ *sqrt* b_i / *N*›
by −
qed

have *norm-t4*: ‹*norm ?t4* $\leq$ *sqrt* b_{j0} $*$ *sqrt* b_i / *N*›
proof −
have *aux*: ‹$I' = Some -` I \Longrightarrow$
card (*Some* −‘ *I*) $\leq b_i \Longrightarrow$ *Some* $x \in I \Longrightarrow$ *card* $\{y \in I.\ y \in$ *range Some*$\} \leq b_i$› **for** *x*
by (*smt* (*verit*) *Collect-cong Int-iff card-image image-vimage-eq inf-set-def inj-Some mem-Collect-eq*)
have $*$: ‹*norm* ($\sum d{\in}Some$ ‘ I'. β *d* $*_C$ *ket* $(y_0,\ d')$) $\leq$ *sqrt* b_i› **for** d' :: ‹$'y$ *option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using I'*-def* b_i *aux* **by** *auto*

have ‹*norm ?t4* = *inverse* (*real N*) $*$ *norm* ($\sum d' \in$ *Some* ‘ $(-J')$. $\sum d{\in}Some$ ‘ I'.
β *d* $*_C$ *ket* $(y_0,\ d')$)›
apply (*simp add*: *sum.cartesian-product*$'$ *sum.reindex N-def*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹... $\leq$ *inverse* (*real N*) $*$ (*sqrt* (*card* (*Some* ‘ $(-\ J')$)) $*$ *sqrt* b_i)›
apply (*rule mult-left-mono*)
using $*$ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹... $\leq$ *inverse N* $*$ (*sqrt* b_{j0} $*$ *sqrt* b_i)›
by (*simp add*: *mult-right-mono N-def*)
also have ‹... $\leq$ *sqrt* b_{j0} $*$ *sqrt* b_i / *N*›
by (*simp add*: *divide-inverse-commute*)
finally show ‹*norm ?t4* $\leq$ *sqrt* b_{j0} $*$ *sqrt* b_i / *N*›
by −
qed

have *norm-t5*: ‹*norm ?t5* $\leq$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
proof (*cases* ‹*None*$\in$*I*›)
case *True*
have $*$: ‹*norm* (β *None* $*_C$ *ket* $(y_0 +$ *the* $d',\ d')$) $\leq$ *sqrt* (*1*::*nat*)› **for** d' :: ‹$'y$ *option*›
using β*None* **by** *simp*

have ‹*norm ?t5* = *inverse* (*sqrt N*) $*$ *norm* ($\sum d' \in$ *Some* ‘ $(-J')$.
β *None* $*_C$ *ket* $(y_0 +$ *the* $d',\ d')$)›
by (*simp add*: *sum.cartesian-product*$'$ *sum.reindex*)
also have ‹... $\leq$ *inverse* (*sqrt N*) $*$ (*sqrt* (*card* (*Some* ‘ $(-\ J')$)))›
apply (*rule mult-left-mono*)
using $*$ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹... $\leq$ *inverse* (*sqrt N*) $*$ *sqrt* b_{j0}›
by (*simp add*: *mult-right-mono N-def*)
also have ‹... $\leq$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
by (*simp add*: *divide-inverse-commute True*)
finally show ‹*norm ?t5* $\leq$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *sqrt N*›
by −
next
case *False*
then show *?thesis* **by** (*simp add*: β*None0*)
qed

have *norm-t6*: ‹*norm ?t6* $\leq$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *N*›
proof (*cases* ‹*None*$\in$*I*›)
case *True*
have $*$: ‹*norm* (β *None* $*_C$ *ket* $y'd'$) $\leq$ *1*› **for** $y'd'$ :: ‹$'y \times 'y$ *option*›
using β*None* **by** *simp*

have ‹*norm ?t6* = *inverse* (*N* $*$ *sqrt N*) $*$ *norm* ($\sum$ $y'd' \in$ (*UNIV*::$'y$ *set*) $\times$ *Some* ‘ ($-J'$). β *None* $*_C$ *ket* $y'd'$)›
by (*simp add*: *sum.cartesian-product′ sum.reindex*)
also have ‹... $\leq$ *inverse* (*N* $*$ *sqrt N*) $*$ (*sqrt N* $*$ *sqrt* (*card* (*Some* ‘ ($-$ J'))))›
apply (*rule mult-left-mono*)
using $*$ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def real-sqrt-mult*)
also have ‹... $\leq$ *inverse* (*N* $*$ *sqrt N*) $*$ (*sqrt N* $*$ *sqrt* b_{j0})›
by (*simp add*: *N-def*)
also have ‹... $\leq$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *N*›
by (*simp add*: *divide-inverse-commute less-eq-real-def True N-def*)
finally show ‹*norm ?t6* $\leq$ *of-bool* (*None*$\in$*I*) $*$ *sqrt* b_{j0} / *N*›
by $-$
next
case *False*
then show *?thesis* **by** (*simp add*: β*None0*)
qed

have *norm-t7*: ‹*norm ?t7* $\leq$ *of-bool* (*None*$\notin$*J*) / *sqrt N*›
proof (*cases* ‹*None*$\in$*J*›)
assume ‹*None* $\notin$ *J*›

have ‹*norm ?t7* = *inverse* (*sqrt N*) $*$ *norm* ($\sum$ *d*$\in I'$. β (*Some d*) $*_C$ *ket* (y_0 + *d*, *None* :: $'y$ *option*))›
using ‹*None* $\notin$ *J*› **by** *simp*
also have ‹... = *inverse* (*sqrt N*) $*$ *norm* ($\sum$ *d*$\in$*Some* ‘ I'. β *d* $*_C$ *ket* (y_0 + *the d*, *None* :: $'y$ *option*))›
apply (*subst sum.reindex*)
by *auto*
also have ‹... $\leq$ *inverse* (*sqrt N*) $*$ (*sqrt* (*real 1*))›
proof $-$
have *aux*: ‹$x \in I' \Longrightarrow$ *card* {*y*. *x* = *the y* $\wedge$ *y* $\in$ *Some* ‘ I'} $\leq$ *Suc 0*› **for** *x*
by (*smt* (*verit*, *del-insts*) *card-eq-Suc-0-ex1 dual-order.refl imageE imageI mem-Collect-eq option.sel*)
show *?thesis*
apply (*rule mult-left-mono*)
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using *aux* **by** *auto*
qed
also have ‹... = *of-bool* (*None*$\notin$*J*) / *sqrt N*›
using ‹*None* $\notin$ *J*› *inverse-eq-divide* **by** *auto*
finally show *?thesis*
by $-$
qed *simp*

have *norm-t8*: ‹*norm ?t8* $\leq$ *of-bool* (*None*$\notin$*J*) $*$ *sqrt* b_i / *N*›
proof (*cases* ‹*None*$\in$*J*›)
assume ‹*None* $\notin$ *J*›
have *aux*: ‹I' = *Some* $-$‘ *I* $\Longrightarrow$

card (*Some* −' *I*) ≤ b_i ⟹ *Some x* ∈ *I* ⟹ *card* {*y* ∈ *I*. *y* ∈ *range Some*} ≤ b_i› **for** *x*
by (*smt* (*verit*, *ccfv-SIG*) *Collect-cong Int-iff card-image image-vimage-eq inf-set-def inj-Some mem-Collect-eq*)
have ∗: ‹*norm* (∑ *d*∈*Some* ' *I'*. β *d* ∗$_C$ *ket* (*y'*, *None* :: '*y option*)) ≤ *sqrt* (*real* b_i)› **for** *y'* :: '*y*
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using *I'-def* b_i *aux* **by** *auto*

have ‹*norm ?t8* = *inverse* (*N* ∗ *sqrt N*) ∗ *norm* (∑ *y'*::'*y*∈*UNIV*. ∑ *d*∈*Some* ' *I'*. β *d* ∗$_C$ *ket* (*y'*, *None* :: '*y option*))›
apply (*simp add*: *sum.reindex* ‹*None* ∉ *J*› *N-def*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (*N* ∗ *sqrt N*) ∗ (*sqrt N* ∗ *sqrt* (*real* b_i))›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def*)
also have ‹. . . = *of-bool* (*None*∉*J*) ∗ *sqrt* b_i / *N*›
using ‹*None* ∉ *J*› *inverse-eq-divide*
by (*simp add*: *divide-inverse-commute N-def*)
finally show *?thesis*
by −
qed *simp*

have *norm-t9*: ‹*norm ?t9* ≤ *of-bool* (*None*∈*I* ∧ *None*∉*J*) / *sqrt N*›
proof (*cases* ‹*None*∈*I* ∧ *None*∉*J*›)
case *True*

have ‹*norm ?t9* = *inverse N* ∗ *norm* (∑ *y'*::'*y*∈*UNIV*. β *None* ∗$_C$ *ket* (*y'*, *None* :: '*y option*))›
by (*simp add*: *sum.reindex True*)
also have ‹. . . ≤ *inverse N* ∗ (*sqrt N* ∗ *sqrt 1*)›
apply (*rule mult-left-mono*)
apply (*rule norm-ortho-sum-bound*[**where** *b'*=*1*])
using β*None* **by** (*auto simp*: *N-def*)
also have ‹. . . = *of-bool* (*None*∈*I* ∧ *None*∉*J*) / *sqrt N*›
using *True* **apply** *simp*
by (*metis divide-inverse-commute inverse-eq-divide of-nat-0-le-iff sqrt-divide-self-eq*)
finally show *?thesis*
by −
next
case *False* **with** β*None0*
show *?thesis* **by** *auto*
qed

have ‹*norm* (*PJ* ∗$_V$ φ) ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*) + *sqrt* b_{j0} ∗ *sqrt* b_i / *N* + *sqrt* b_{j0} ∗ *sqrt* b_i / *N*
+ *sqrt* b_{j0} ∗ *sqrt* b_i / *N* + *of-bool* (*None*∈*I*) ∗ *sqrt* b_{j0} / *sqrt N* + *of-bool* (*None*∈*I*) ∗ *sqrt* b_{j0} / *N*
+ *of-bool* (*None*∉*J*) / *sqrt N* + *of-bool* (*None*∉*J*) ∗ *sqrt* b_i / *N* + *of-bool* (*None*∈*I* ∧ *None*∉*J*) / *sqrt N*›
unfolding *P0*φ
apply (*rule norm-triangle-le-diff norm-triangle-le*, *rule add-mono*)+
apply (*rule norm-t1*)
apply (*rule norm-t2*)
apply (*rule norm-t3*)

apply (*rule norm-t4*)
apply (*rule norm-t5*)
apply (*rule norm-t6*)
apply (*rule norm-t7*)
apply (*rule norm-t8*)
apply (*rule norm-t9*)
by −
also have ‹... ≤ *preserve-query1-fixY-bound* (*None*∈*I*) (*None*∉*J*) b_i b_{j0}›
by (*auto simp*: *preserve-query1-fixY-bound-def mult.commute mult.left-commute*)
also have ‹... ≤ ε›
by (*simp add*: ε)
finally show ‹*norm* (*Proj* (− *ket-invariant* (*UNIV* × *J*)) $*_V$ φ) ≤ ε›
unfolding *PJ-def*
apply (*subst ket-invariant-compl*[*symmetric*])
by *simp*
qed

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›
proof (*rule preservesI′*)
show ‹ε ≥ *0*›
using - ε **apply** (*rule order.trans*)
by (*simp add*: *preserve-query1′-fixY-bound-def*)
fix ψ :: ‹(′*y* × ′*y option*) *ell2*›
assume ψ: ‹ψ ∈ *space-as-set* (*ket-invariant* ({y_0} × *I*))›
assume ‹*norm* ψ = *1*›

define *I′ J′* **where** ‹*I′* = *Some* −‘ *I*› **and** ‹*J′* = *Some* −‘ *J*›
then have ‹{y_0} × *I* ⊆ {y_0} × (*Some* ‘ *I′* ∪ {*None*})›
apply (*rule-tac Sigma-mono*)
by *auto*
with ψ **have** ψ′: ‹ψ ∈ *space-as-set* (*ket-invariant* ({y_0} × ((*Some* ‘ *I′* ∪ {*None*}))))›
using *less-eq-ccsubspace.rep-eq ket-invariant-le* **by** *fastforce*
have [*simp*]: ‹*I′* ⊆ *J′*›
using *I′-def J′-def IJ* **by** *blast*

define β **where** ‹β *d* = *Rep-ell2* ψ (y_0,*d*)› **for** *d*
then have β: ‹ψ = ($\sum$ *d*∈*Some* ‘ *I′* ∪ {*None*}. β *d* $*_C$ *ket* (y_0,*d*))›
using *ell2-sum-ket-ket-invariant*[*OF* ψ′]
by (*auto simp*: *sum.cartesian-product′*)
have β*bound*: ‹($\sum$ *d*∈(*Some* ‘ *I′* ∪ {*None*}). (*cmod* (β *d*))^2) ≤ *1*› (**is** ‹*?lhs* ≤ *1*›)
apply (*subgoal-tac* ‹(*norm* ψ)^2 = *?lhs*›)
apply (*simp add*: ‹*norm* ψ = *1*›)
by (*simp add*: β *pythagorean-theorem-sum del*: *sum.insert*)
have β*None*: ‹*cmod* (β *None*) ≤ *1*›
proof −

have ‹$(cmod\ (\beta\ None))^2 = (\sum d{\in}\{None\}.\ (cmod\ (\beta\ d))^2)$›
by *simp*
also have ‹$\ldots \le (\sum d{\in}(Some\ `\ I' \cup \{None\}).\ (cmod\ (\beta\ d))^2)$›
apply (*rule sum-mono2*) **by** *auto*
also have ‹$\ldots \le 1$›
by (*rule βbound*)
finally show *?thesis*
by (*simp add*: *power-le-one-iff*)
qed
have *βNone0*: ‹$\beta\ None = 0$› **if** ‹$None \notin I$›
using ψ *that* **by** (*simp add*: *β-def ket-invariant-Rep-ell2*)

have [*simp*]: ‹$Some\ -`\ insert\ None\ X = Some\ -`\ X$› **for** X :: ‹*'z option set*›
by *auto*
have [*simp*]: ‹$Some\ -`\ Some\ `\ X = X$› **for** X :: ‹*'z set*›
by *auto*
have [*simp*]: ‹$Some\ x \in J \longleftrightarrow x \in J'$› **for** x
by (*simp add*: *J'-def*)
have [*simp*]: ‹$x \in I' \Longrightarrow x \in J'$› **for** x
using ‹$I' \subseteq J'$› **by** *blast*
have [*simp*]: ‹$(\sum x{\in}X.\ if\ x \notin Y\ then\ f\ x\ else\ 0) = (\sum x{\in}X{-}Y.\ f\ x)$› **if** ‹*finite X*› **for** f :: ‹$'y \Rightarrow$ *'z::ab-group-add*› **and** $X\ Y$
apply (*rule sum.mono-neutral-cong-right*)
using *that* **by** *auto*
have [*simp*]: ‹$\beta\ yd *_C a *_C b = a *_C \beta\ yd *_C b$› **for** *yd a* **and** b :: ‹*'z::complex-vector*›
by *auto*
have [*simp*]: ‹$cmod\ \alpha = inverse\ (sqrt\ N)$› ‹$cmod\ (\alpha^2) = inverse\ N$› ‹$cmod\ (\alpha\hat{}3) = inverse\ (N * sqrt\ N)$› ‹$cmod\ (\alpha\hat{}4) = inverse\ (N^2)$›
by (*auto simp*: *norm-mult numeral-3-eq-3 α-def inverse-eq-divide norm-divide norm-power power-one-over power2-eq-square power4-eq-xxxx*)
have [*simp*]: ‹$card\ (Some\ `\ I') \le b_i$›
by (*metis I'-def* b_i *card-image inj-Some*)
have *bound-J'*[*simp*]: ‹$card\ (Some\ `\ (-\ J')) \le b_{j0}$›
using b_{j0} **unfolding** *J'-def* **by** (*simp add*: *card-image*)

define φ **and** *PJ* :: ‹*('y * 'y option) update*› **where**
‹$\varphi = query1' *_V \psi$› **and** ‹$PJ = Proj\ (ket\text{-}invariant\ (UNIV \times -J))$›
have [*simp*]: ‹$PJ *_V ket\ (x,y) = (if\ y{\in}{-}J\ then\ ket\ (x,y)\ else\ 0)$› **for** $x\ y$
by (*simp add*: *Proj-ket-invariant-ket PJ-def*)
have *P0φ*: ‹$PJ *_V \varphi =$
$\alpha\hat{}4 *_C (\sum d{\in}I'.\ \sum y'{\in}UNIV.\ \sum d'{\in}-\ J'.\ \beta\ (Some\ d) *_C ket\ (y',\ Some\ d'))$
$-\ \alpha^2 *_C (\sum d{\in}I'.\ \sum d'{\in}-\ J'.\ \beta\ (Some\ d) *_C ket\ (y_0 + d,\ Some\ d'))$
$-\ \alpha^2 *_C (\sum d{\in}I'.\ \sum d'{\in}-\ J'.\ \beta\ (Some\ d) *_C ket\ (y_0 + d',\ Some\ d'))$
$+\ \alpha *_C (\sum d'{\in}-\ J'.\ \beta\ (None) *_C ket\ (y_0 + d',\ Some\ d'))$
$-\ \alpha\hat{}3 *_C (\sum y'{\in}UNIV.\ \sum d'{\in}-\ J'.\ \beta\ (None) *_C ket\ (y',\ Some\ d'))$
$+\ (of\text{-}bool\ (None \notin J) * \alpha) *_C (\sum d{\in}I'.\ \beta\ (Some\ d) *_C ket\ (y_0 + d,\ None))$
$-\ (of\text{-}bool\ (None \notin J) * \alpha\hat{}3) *_C (\sum d{\in}I'.\ \sum y'{\in}UNIV.\ \beta\ (Some\ d) *_C ket\ (y',\ None))$
$+\ (of\text{-}bool\ (None \notin J) * \alpha^2) *_C (\sum y'{\in}UNIV.\ \beta\ None *_C ket\ (y',\ None))$
›
(**is** ‹$- = ?t1 - ?t2 - ?t3 + ?t5 - ?t6 + ?t7 - ?t8 + ?t9$›)
by (*simp add*: *φ-def β query1' option-sum-split vimage-Compl cblinfun.add-right cblinfun.diff-right if-distrib Compl-eq-Diff-UNIV vimage-singleton-inj sum-sum-if-eq sum.distrib scaleC-diff-right scaleC-sum-right sum-subtractf case-prod-beta sum.cartesian-product' scaleC-add-right add-diff-eq cblinfun.scaleC-right cblinfun.sum-right*

flip: *sum.Sigma add.assoc scaleC-scaleC*
cong del: *option.case-cong if-cong*)

have *norm-t1*: ‹*norm ?t1* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*)›
proof −
have ∗: ‹*norm* ($\sum$ *d*∈*Some* ‘ *I*′. *β d* $*_C$ *ket y*′*d*′) ≤ *sqrt* b_i› **for** *y*′*d*′ :: ‹′*y* × ′*y option*›
using - - *βbound* **apply** (*rule bound-coeff-sum2*)
by *auto*

have ‹*norm ?t1* = *inverse* (N^2) ∗ *norm* ($\sum$ *y*′*d*′ ∈ (*UNIV*::′*y set*) × *Some* ‘ (−*J*′). $\sum$ *d*∈*Some* ‘ *I*′. *β d* $*_C$ *ket y*′*d*′)›
apply (*simp add*: *sum.cartesian-product*′ *sum.reindex N-def*)
apply (*subst* (*2*) *sum.swap*) **apply** (*subst* (*3*) *sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (N^2) ∗ (*sqrt N* ∗ *sqrt* (*card* (*Some* ‘ (− *J*′))) ∗ *sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def real-sqrt-mult*)
also have ‹. . . ≤ *inverse* (N^2) ∗ (*sqrt N* ∗ *sqrt* b_{j0} ∗ *sqrt* b_i)›
by (*metis bound-J*′ *linordered-field-class.inverse-nonnegative-iff-nonnegative mult.commute mult-right-mono of-nat-0-le-iff of-nat-mono real-sqrt-ge-zero real-sqrt-le-iff*)
also have ‹. . . ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*)›
by (*smt* (*verit*, *del-insts*) *divide-divide-eq-left divide-inverse mult.commute of-nat-0-le-iff of-nat-power power2-eq-square real-divide-square-eq real-sqrt-pow2 times-divide-eq-left*)
finally show ‹*norm ?t1* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*)›
by −
qed

have *norm-t2*: ‹*norm ?t2* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
proof −
have ∗: ‹*card* {*d*. *δ* = *the d* ∧ *d* ∈ *Some* ‘ *I*′} ≤ *card I*′› **for** *δ*
apply (*rule card-inj-on-le*[**where** *f*=*the*])
by (*auto intro*!: *inj-onI*)
have ∗: ‹*norm* ($\sum$ *d*∈*Some* ‘ *I*′. *β d* $*_C$ *ket* (y_0 + *the d*, *d*′)) ≤ *sqrt* b_i› **for** *d*′ :: ‹′*y option*›
using - - *βbound* **apply** (*rule bound-coeff-sum2*)
using ∗ *I*′*-def* b_i *order.trans* **by** *auto*

have ‹*norm ?t2* = *inverse* (*real N*) ∗ *norm* ($\sum$ *d*′ ∈ *Some* ‘ (−*J*′). $\sum$ *d*∈*Some* ‘ *I*′. *β d* $*_C$ *ket* (y_0 + *the d*, *d*′))›
apply (*simp add*: *sum.cartesian-product*′ *sum.reindex N-def*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (*real N*) ∗ (*sqrt* (*card* (*Some* ‘ (− *J*′))) ∗ *sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹. . . ≤ *inverse N* ∗ (*sqrt* b_{j0} ∗ *sqrt* b_i)›
by (*simp add*: *mult-right-mono N-def*)
also have ‹. . . ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by (*simp add*: *divide-inverse-commute*)
finally show ‹*norm ?t2* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›
by −
qed

have *norm-t3*: ‹*norm ?t3* ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / *N*›

proof −
have *aux*: ‹$I' = Some$ −‘ $I \Longrightarrow card\ (Some$ −‘ $I) \le b_i \Longrightarrow card\ \{y \in I.\ y \in range\ Some\} \le b_i$›
by (*smt* (*verit*, *ccfv-SIG*) *Collect-cong Int-iff card-image image-vimage-eq inf-set-def inj-Some mem-Collect-eq*)
have ∗: ‹$norm\ (\sum d{\in}Some\ `\ I'.\ \beta\ d *_C ket\ (y_0 + the\ d',\ d')) \le sqrt\ b_i$› **for** d' :: ‹$'y$ *option*›
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using I'*-def* b_i *aux* **by** *auto*

have ‹$norm\ ?t3 = inverse\ (real\ N) * norm\ (\sum d' \in Some\ `\ (-J').\ \sum d{\in}Some\ `\ I'.$
$\beta\ d *_C ket\ (y_0 + the\ d',\ d'))$›
apply (*simp add*: *sum.reindex N-def*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹$\ldots \le inverse\ (real\ N) * (sqrt\ (card\ (Some\ `\ (-\ J'))) * sqrt\ b_i)$›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹$\ldots \le inverse\ N * (sqrt\ b_{j0} * sqrt\ b_i)$›
by (*simp add*: *mult-right-mono N-def*)
also have ‹$\ldots \le sqrt\ b_{j0} * sqrt\ b_i\ /\ N$›
by (*simp add*: *divide-inverse-commute*)
finally show ‹$norm\ ?t3 \le sqrt\ b_{j0} * sqrt\ b_i\ /\ N$›
by −
qed

have *norm-t5*: ‹$norm\ ?t5 \le of\text{-}bool\ (None{\in}I) * sqrt\ b_{j0}\ /\ sqrt\ N$›
proof (*cases* ‹$None{\in}I$›)
case *True*
have ∗: ‹$norm\ (\beta\ None *_C ket\ (y_0 + the\ d',\ d')) \le sqrt\ (1{::}nat)$› **for** d' :: ‹$'y$ *option*›
using β*None* **by** *simp*

have ‹$norm\ ?t5 = inverse\ (sqrt\ N) * norm\ (\sum d' \in Some\ `\ (-J').$
$\beta\ None *_C ket\ (y_0 + the\ d',\ d'))$›
by (*simp add*: *sum.cartesian-product′ sum.reindex*)
also have ‹$\ldots \le inverse\ (sqrt\ N) * (sqrt\ (card\ (Some\ `\ (-\ J'))))$›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹$\ldots \le inverse\ (sqrt\ N) * sqrt\ b_{j0}$›
by (*simp add*: *mult-right-mono N-def*)
also have ‹$\ldots \le of\text{-}bool\ (None{\in}I) * sqrt\ b_{j0}\ /\ sqrt\ N$›
by (*simp add*: *divide-inverse-commute True*)
finally show ‹$norm\ ?t5 \le of\text{-}bool\ (None{\in}I) * sqrt\ b_{j0}\ /\ sqrt\ N$›
by −
next
case *False*
then show *?thesis* **by** (*simp add*: β*None0*)
qed

have *norm-t6*: ‹$norm\ ?t6 \le of\text{-}bool\ (None{\in}I) * sqrt\ b_{j0}\ /\ N$›
proof (*cases* ‹$None{\in}I$›)
case *True*
have ∗: ‹$norm\ (\beta\ None *_C ket\ y'd') \le 1$› **for** $y'd'$ :: ‹$'y \times 'y$ *option*›
using β*None* **by** *simp*

have ‹*norm ?t6 = inverse (N ∗ sqrt N) ∗ norm* ($\sum$ *y'd'* ∈ (*UNIV*::'*y set*) × *Some* ' (−*J'*). *β None* $*_C$ *ket y'd'*)›
by (*simp add*: *sum.cartesian-product' sum.reindex*)
also have ‹. . . ≤ *inverse* (*N* ∗ *sqrt N*) ∗ (*sqrt N* ∗ *sqrt* (*card* (*Some* ' (− *J'*))))›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left real-sqrt-mult N-def*)
also have ‹. . . ≤ *inverse* (*N* ∗ *sqrt N*) ∗ (*sqrt N* ∗ *sqrt* b_{j0})›
by (*simp add*: *N-def*)
also have ‹. . . ≤ *of-bool* (*None*∈*I*) ∗ *sqrt* b_{j0} / *N*›
by (*simp add*: *divide-inverse-commute less-eq-real-def True N-def*)
finally show ‹*norm ?t6* ≤ *of-bool* (*None*∈*I*) ∗ *sqrt* b_{j0} / *N*›
by −
next
case *False*
then show *?thesis* **by** (*simp add*: *βNone0*)
qed

have *norm-t7*: ‹*norm ?t7* ≤ *of-bool* (*None*∉*J*) / *sqrt N*›
proof (*cases* ‹*None*∈*J*›)
assume ‹*None* ∉ *J*›

have ‹*norm ?t7* = *inverse* (*sqrt N*) ∗ *norm* ($\sum$ *d*∈*I'*. *β* (*Some d*) $*_C$ *ket* (y_0 + *d*, *None* :: '*y option*))›
using ‹*None* ∉ *J*› **by** *simp*
also have ‹. . . = *inverse* (*sqrt N*) ∗ *norm* ($\sum$ *d*∈*Some* ' *I'*. *β d* $*_C$ *ket* (y_0 + *the d*, *None* :: '*y option*))›
apply (*subst sum.reindex*)
by *auto*
also have ‹. . . ≤ *inverse* (*sqrt N*) ∗ (*sqrt* (*real 1*))›
proof −
have *aux*: ‹*x* ∈ *I'* ⟹ *card* {*y*. *x* = *the y* ∧ *y* ∈ *Some* ' *I'*} ≤ *Suc 0*› **for** *x*
by (*smt* (*verit*, *del-insts*) *card-eq-Suc-0-ex1 dual-order.refl imageE imageI mem-Collect-eq option.sel*)
show *?thesis*
apply (*rule mult-left-mono*)
using - - *βbound* **apply** (*rule bound-coeff-sum2*)
using *aux* **by** *auto*
qed
also have ‹. . . = *of-bool* (*None*∉*J*) / *sqrt N*›
using ‹*None* ∉ *J*› *inverse-eq-divide* **by** *auto*
finally show *?thesis*
by −
qed *simp*

have *norm-t8*: ‹*norm ?t8* ≤ *of-bool* (*None*∉*J*) ∗ *sqrt* b_i / *N*›
proof (*cases* ‹*None*∈*J*›)
assume ‹*None* ∉ *J*›

have *aux*: ‹*card* (*Some* −' *I*) ≤ b_i ⟹ *card* {*y* ∈ *I*. *y* ∈ *range Some*} ≤ b_i›
by (*smt* (*verit*, *ccfv-SIG*) *Collect-cong Int-iff card-image image-vimage-eq inf-set-def inj-Some mem-Collect-eq*)
have ∗: ‹*norm* ($\sum$ *d*∈*Some* ' *I'*. *β d* $*_C$ *ket* (*y'*, *None* :: '*y option*)) ≤ *sqrt* (*real* b_i)› **for** *y'* :: '*y*
using - - *βbound* **apply** (*rule bound-coeff-sum2*)
using *I'-def* b_i *aux* **by** *auto*

have ‹*norm ?t8* = *inverse* (*N* ∗ *sqrt N*) ∗ *norm* ($\sum$ *y′*::*′y*∈*UNIV*. $\sum$ *d*∈*Some* ‘ *I′*. *β d* $*_C$ *ket* (*y′*, *None* :: *′y option*))›
apply (*simp add*: *sum.reindex* ‹*None* ∉ *J*› *N-def*)
apply (*subst sum.swap*) **apply** (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (*N* ∗ *sqrt N*) ∗ (*sqrt N* ∗ *sqrt* (*real* b_i))›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def*)
also have ‹. . . = *of-bool* (*None*∉*J*) ∗ *sqrt* b_i / *N*›
using ‹*None* ∉ *J*› *inverse-eq-divide*
by (*simp add*: *divide-inverse-commute N-def*)
finally show *?thesis*
by −
qed *simp*

have *norm-t9*: ‹*norm ?t9* ≤ *of-bool* (*None*∈*I* ∧ *None*∉*J*) / *sqrt N*›
proof (*cases* ‹*None*∈*I* ∧ *None*∉*J*›)
case *True*

have ‹*norm ?t9* = *inverse N* ∗ *norm* ($\sum$ *y′*::*′y*∈*UNIV*. *β None* $*_C$ *ket* (*y′*, *None* :: *′y option*))›
by (*simp add*: *sum.reindex True*)
also have ‹. . . ≤ *inverse N* ∗ (*sqrt N* ∗ *sqrt 1*)›
apply (*rule mult-left-mono*)
apply (*rule norm-ortho-sum-bound*[**where** *b′*=*1*])
using *βNone* **by** (*auto simp*: *N-def*)
also have ‹. . . = *of-bool* (*None*∈*I* ∧ *None*∉*J*) / *sqrt N*›
using *True* **apply** *simp*
by (*metis divide-inverse-commute inverse-eq-divide of-nat-0-le-iff sqrt-divide-self-eq*)
finally show *?thesis*
by −
next
case *False* **with** *βNone0*
show *?thesis* **by** *auto*
qed

have ‹*norm* (*PJ* $*_V$ *φ*) ≤ *sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*) + *sqrt* b_{j0} ∗ *sqrt* b_i / *N* + *sqrt* b_{j0} ∗ *sqrt* b_i / *N*
+ *of-bool* (*None*∈*I*) ∗ *sqrt* b_{j0} / *sqrt N* + *of-bool* (*None*∈*I*) ∗ *sqrt* b_{j0} / *N*
+ *of-bool* (*None*∉*J*) / *sqrt N* + *of-bool* (*None*∉*J*) ∗ *sqrt* b_i / *N* + *of-bool* (*None*∈*I* ∧ *None*∉*J*) / *sqrt N*›
unfolding *P0φ*
apply (*rule norm-triangle-le-diff norm-triangle-le*, *rule add-mono*)+
apply (*rule norm-t1*)
apply (*rule norm-t2*)
apply (*rule norm-t3*)
apply (*rule norm-t5*)
apply (*rule norm-t6*)
apply (*rule norm-t7*)
apply (*rule norm-t8*)
apply (*rule norm-t9*)
by −
also have ‹. . . ≤ *preserve-query1′-fixY-bound* (*None*∈*I*) (*None*∉*J*) b_i b_{j0}›
by (*auto simp*: *preserve-query1′-fixY-bound-def mult.commute mult.left-commute*)

also have ‹... $\leq \varepsilon$›
  by (*simp add*: ε)
finally show ‹*norm* (*Proj* (− *ket-invariant* (*UNIV* × *J*)) $*_V$ φ) $\leq \varepsilon$›
  unfolding *PJ-def*
  apply (*subst ket-invariant-compl*[*symmetric*])
  by *simp*
qed

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} × *I*) {(y_0+*d*, *Some d*)| *d*. *True*} ε›
proof (*rule preservesI'*)
  show ‹$\varepsilon \geq 0$›
    using - ε **apply** (*rule order.trans*)
    by (*simp add*: *preserve-query1'-fixY-bound-output-def*)
  fix ψ :: ‹(*'y* × *'y option*) *ell2*›
  assume ψ: ‹$\psi \in$ *space-as-set* (*ket-invariant* ({y_0} × *I*))›
  assume ‹*norm* ψ = *1*›

  define *I'* **where** ‹*I'* = *Some* −‘ *I*›
  then have ‹{y_0} × *I* ⊆ {y_0} × (*Some* ‘ *I'* ∪ {*None*})›
    apply (*rule-tac Sigma-mono*)
    by *auto*
  with ψ **have** ψ': ‹$\psi \in$ *space-as-set* (*ket-invariant* ({y_0} × ((*Some* ‘ *I'* ∪ {*None*}))))›
    using *less-eq-ccsubspace.rep-eq ket-invariant-le* **by** *fastforce*

  define β **where** ‹β *d* = *Rep-ell2* ψ (y_0,*d*)› **for** *d*
  then have β: ‹ψ = ($\sum$ *d*∈*Some* ‘ *I'* ∪ {*None*}. β *d* $*_C$ *ket* (y_0,*d*))›
    using *ell2-sum-ket-ket-invariant*[*OF* ψ']
    by (*auto simp*: *sum.cartesian-product'*)
  have β*bound*: ‹($\sum$ *d*∈(*Some* ‘ *I'* ∪ {*None*}). (*cmod* (β *d*))^2) $\leq$ *1*› (**is** ‹*?lhs* $\leq$ *1*›)
    apply (*subgoal-tac* ‹(*norm* ψ)^2 = *?lhs*›)
     apply (*simp add*: ‹*norm* ψ = *1*›)
    by (*simp add*: β *pythagorean-theorem-sum del*: *sum.insert*)
  have β*bound1*[*simp*]: ‹*cmod* (β *x*) $\leq$ *1*› **for** *x*
    using *norm-point-bound-ell2*[*of* ψ] ‹*norm* ψ = *1*› **unfolding** β*-def* **by** *auto*

  have [*simp*]: ‹*Some* −‘ *insert None X* = *Some* −‘ *X*› **for** *X* :: ‹*'z option set*›
    by *auto*
  have [*simp*]: ‹*Some* −‘ *Some* ‘ *X* = *X*› **for** *X* :: ‹*'z set*›
    by *auto*
  have [*simp*]: ‹β *yd* $*_C$ *a* $*_C$ *b* = *a* $*_C$ β *yd* $*_C$ *b*› **for** *yd a* **and** *b* :: ‹*'z*::*complex-vector*›
    by *auto*
  have [*simp*]: ‹*cmod* α = *inverse* (*sqrt N*)› ‹*cmod* (α^2) = *inverse N*› ‹*cmod* (α^*3*) = *inverse* (*N* ∗ *sqrt* *N*)› ‹*cmod* (α^*4*) = *inverse* (N^2)›
   by (*auto simp*: *norm-mult numeral-3-eq-3* α*-def inverse-eq-divide norm-divide norm-power power-one-over power4-eq-xxxx power2-eq-square*)
  have [*simp*]: ‹*card* (*Some* ‘ *I'*) $\leq b_i$›
    by (*metis I'-def* b_i *card-image inj-Some*)

define φ **and** *PJ* :: ‹('*y* ∗ '*y option*) *update*› **where**
‹φ = *query1'* $*_V$ ψ› **and** ‹*PJ* = *Proj* (*ket-invariant* (− {(y_0+*d*, *Some d*)| *d*. *True*}))›
have *aux*: ‹∀ *d*. $y \neq y_0 + d \Longrightarrow d \neq$ *Some* (y_0 + *y*)› **for** *d y*
by (*metis add.right-neutral y-cancel ordered-field-class.sign-simps*(*1*))
then have [*simp*]: ‹*PJ* $*_V$ *ket* (*y*,*d*) = (*if Some* (y_0 + *y*) = *d then 0 else ket* (*y*,*d*))› **for** *y d*
by (*auto simp add*: *Proj-ket-invariant-ket PJ-def*)
have *P0*φ: ‹*PJ* $*_V$ φ =
α $*_C$ ($\sum$ *d*∈*I'*. β (*Some d*) $*_C$ *ket* (y_0 + *d*, *None*))
− α^*3* $*_C$ ($\sum$ *d*∈*I'*. $\sum$ *y*∈*UNIV*. β (*Some d*) $*_C$ *ket* (*y*, *None*))
− α^2 $*_C$ ($\sum$ *d*∈*I'*. $\sum$ *d'*∈*UNIV*. *if d*=*d' then 0 else* β (*Some d*) $*_C$ *ket* (y_0 + *d*, *Some d'*))
+ α^*4* $*_C$ ($\sum$ *d*∈*I'*. $\sum$ *y*∈*UNIV*. $\sum$ *d'*∈*UNIV*. *if* y_0+*y*=*d' then 0 else* β (*Some d*) $*_C$ *ket* (*y*, *Some d'*))
− α^*3* $*_C$ ($\sum$ *y*∈*UNIV*. $\sum$ *d'*∈*UNIV*. *if* y_0+*y*=*d' then 0 else* β *None* $*_C$ *ket* (*y*, *Some d'*))
+ α^2 $*_C$ ($\sum$ *y*∈*UNIV*. β *None* $*_C$ *ket* (*y*, *None*))›
(**is** ‹- = *?t1* − *?t2* − *?t3* + *?t4* − *?t5* + *?t6*›)

by (*simp add*: φ*-def* β *query1' option-sum-split vimage-Compl of-bool-def cblinfun.sum-right cblinfun.add-right cblinfun.diff-right if-distrib Compl-eq-Diff-UNIV cblinfun.scaleC-right vimage-singleton-inj sum-sum-if-eq sum.distrib scaleC-diff-right scaleC-sum-right sum-subtractf case-prod-beta sum.cartesian-product' scaleC-add-right add-diff-eq flip*: *sum.Sigma add.assoc scaleC-scaleC cong del*: *option.case-cong if-cong*)

have *norm-t1*: ‹*norm ?t1* ≤ *1* / *sqrt N*›
proof −
have ‹*norm ?t1* = *inverse* (*sqrt N*) ∗ *norm* ($\sum$ *d*∈*Some* ‘ *I'*. β *d* $*_C$ *ket* (y_0 + *the d*, *None* :: '*y option*))›
by (*simp add*: *sum.reindex*)
also have ‹... ≤ *inverse* (*sqrt N*) ∗ *sqrt* (*1*::*nat*)›
proof −
have *aux*: ‹*x* ∈ *I'* ⟹ *card* {*y*. *x* = *the y* ∧ *y* ∈ *Some* ‘ *I'*} ≤ *Suc 0*› **for** *x*
by (*smt* (*verit*, *del-insts*) *card-eq-Suc-0-ex1 imageE imageI le-refl mem-Collect-eq option.sel*)
show *?thesis*
apply (*rule mult-left-mono*)
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using *aux* **by** *auto*
qed
also have ‹... = *1* / *sqrt N*›
apply *simp*
using *inverse-eq-divide* **by** *blast*
finally show ‹*norm ?t1* ≤ *1* / *sqrt N*›
by −
qed

have *norm-t2*: ‹*norm ?t2* ≤ *sqrt* b_i / *N*›
proof −
have ∗: ‹*norm* ($\sum$ *d*∈*Some* ‘ *I'*. β *d* $*_C$ *ket* (*y*, *None* :: '*y option*)) ≤ *sqrt* b_i› **for** *y* :: '*y*
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
by *auto*

have ‹*norm ?t2* = *inverse* (*N* ∗ *sqrt N*) ∗ *norm* ($\sum$ *y*∈*UNIV*. $\sum$ *d*∈*Some* ‘ *I'*. β *d* $*_C$ *ket* (*y* :: '*y*, *None* :: '*y option*))›
apply (*simp add*: *sum.cartesian-product' sum.reindex N-def*)
apply (*subst sum.swap*)

by (*rule refl*)
also have ‹... $\leq$ *inverse* (N $*$ *sqrt* N) $*$ (*sqrt* (*real* N) $*$ *sqrt* (*real* b_i))›
apply (*rule mult-left-mono*)
using $*$ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def*)
also have ‹... $=$ *sqrt* b_i / N›
by (*simp add*: *divide-inverse-commute N-def*)
finally show *?thesis*
by *auto*
qed

have *norm-t3*: ‹*norm ?t3* $\leq$ *1* / *sqrt* N›
proof $-$
have *aux*: ‹*card* $\{y.\ x = the\ y \wedge y \in Some$ ‘ $(I' - \{d'\})\} \leq Suc\ 0$› **for** x d'
by (*smt* (*verit*, *best*) *Collect-empty-eq bot-nat-0.not-eq-extremum card.empty card-eq-Suc-0-ex1 imageE le-simps*(*3*) *mem-Collect-eq nat-le-linear option.sel*)
have $*$: ‹*norm* ($\sum d{\in}Some$ ‘ $(I' - \{d'\})$. β d $*_C$ *ket* (y_0 + *the* d, *Some* d')) $\leq$ *sqrt* (*1*::*nat*)› **for** d'
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
using *aux*[*of* - d'] **by** *auto*

have ‹*norm ?t3* $=$ *inverse* N $*$ *norm* ($\sum d'{\in}UNIV$. $\sum d{\in}Some$ ‘ $(I'{-}\{d'\})$. β d $*_C$ *ket* (y_0 + *the* d, *Some* d'))›
apply (*simp add*: *sum.cartesian-product′ sum.reindex*)
apply (*subst sum.swap*)
apply (*simp add*: *sum-if-eq-else*)
by $-$
also have ‹... $\leq$ *inverse* N $*$ *sqrt* N›
apply (*rule mult-left-mono*)
using $*$ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def*)
also have ‹... $=$ *1* / *sqrt* N›
by (*simp add*: *inverse-eq-divide sqrt-divide-self-eq*)
finally show *?thesis*
by $-$
qed

have *norm-t4*: ‹*norm ?t4* $\leq$ *sqrt* (*real* b_i) / N›
proof $-$
have $*$: ‹*norm* ($\sum d{\in}Some$ ‘ I'. *if* y_0 + *fst* yd' $=$ *snd* yd' *then 0 else* β d $*_C$ *ket* (*fst* yd', *Some* (*snd* yd'))) $\leq$ *sqrt* b_i› **for** yd'
apply (*cases* ‹y_0 + *fst* yd' $=$ *snd* yd'›)
apply *simp*
apply *simp*
using - - β*bound* **apply** (*rule bound-coeff-sum2*)
by *auto*

note *if-cong*[*cong del*]

have ‹*norm ?t4* $=$ *inverse* (N^2) $*$ *norm* ($\sum yd'{\in}UNIV$. $\sum d{\in}Some$ ‘ I'. *if* y_0 + *fst* yd' $=$ *snd* yd' *then 0 else* β d $*_C$ *ket* (*fst* yd', *Some* (*snd* yd')))›
apply (*simp add*: *sum.cartesian-product′ sum.reindex N-def flip*: *UNIV-Times-UNIV*)
apply (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹... $\leq$ *inverse* (N^2) $*$ (*real* N $*$ *sqrt* (*real* b_i))›
apply (*rule mult-left-mono*)

using $*$ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left cinner-ket N-def*
if-distrib[**where** f=‹λx. *cinner - x*›] *if-distrib*[**where** f=‹λx. *cinner x -*›])
also have ‹... $\leq$ *sqrt* (*real* b_i) / *N*›
by (*metis divide-inverse-commute dual-order.refl of-nat-mult power2-eq-square real-divide-square-eq*)
finally show *?thesis*
by −
qed

have *norm-t5*: ‹*norm ?t5* $\leq$ *1* / *sqrt N*›
proof −
have ‹*norm ?t5* = *inverse* (*N* $*$ *sqrt N*) $*$ *norm* ($\sum$ *yd*$\in$*UNIV*. *if* y_0 + *fst yd* = *snd yd then 0 else* β *None* $*_C$ *ket* (*fst yd*, *Some* (*snd yd*)))›
by (*simp add*: *sum.cartesian-product′ sum.reindex flip*: *UNIV-Times-UNIV cong del*: *if-cong*)
also have ‹... $\leq$ *inverse* (*N* $*$ *sqrt N*) $*$ *N*›
apply (*rule mult-left-mono*)
apply (*rule norm-ortho-sum-bound*[**where** *b′*=*1*])
by (*auto simp*: *N-def*)
also have ‹... = *1* / *sqrt N*›
by (*simp add*: *divide-inverse-commute N-def*)
finally show *?thesis*
by −
qed

have *norm-t6*: ‹*norm ?t6* $\leq$ *1* / *sqrt N*›
proof −
have ‹*norm ?t6* = *inverse N* $*$ *norm* ($\sum$ *y*$\in$*UNIV*. β *None* $*_C$ *ket* (*y* :: *′y*, *None* :: *′y option*))›
by *simp*
also have ‹... $\leq$ *inverse N* $*$ *sqrt N*›
apply (*rule mult-left-mono*)
apply (*rule norm-ortho-sum-bound*[**where** *b′*=*1*])
by (*auto simp*: *N-def*)
also have ‹... = *1* / *sqrt N*›
by (*simp add*: *inverse-eq-divide sqrt-divide-self-eq*)
finally show *?thesis*
by −
qed

have ‹*norm* (*PJ* $*_V$ φ) $\leq$ *1* / *sqrt N* + *sqrt* b_i / *N* + *1* / *sqrt N*
+ *sqrt* (*real* b_i) / *N* + *1* / *sqrt N* + *1* / *sqrt N*›
unfolding *P0*φ
apply (*rule norm-triangle-le-diff norm-triangle-le*, *rule add-mono*)+
apply (*rule norm-t1*)
apply (*rule norm-t2*)
apply (*rule norm-t3*)
apply (*rule norm-t4*)
apply (*rule norm-t5*)
apply (*rule norm-t6*)
by −
also have ‹... $\leq$ *preserve-query1′-fixY-bound-output* b_i›
by (*auto simp*: *preserve-query1′-fixY-bound-output-def mult.commute mult.left-commute*)
also have ‹... $\leq$ ε›
by (*simp add*: ε)
finally show ‹*norm* (*Proj* (− *ket-invariant* {(y_0 + *d*, *Some d*) |*d*. *True*}) $*_V$ φ) $\leq$ ε›
unfolding *PJ-def*

apply (*subst ket-invariant-compl*[*symmetric*])
by *simp*
qed

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*)›
proof −
define *K* :: ‹*'x* ⇒ (*'x* × *'y* × (*'x* ⇒ *'y option*)) *ell2 ccsubspace*› **where** ‹*K x* = *lift-invariant reg-1-3* (*ket-invariant* {*x*})› **for** *x*

show *?thesis*
proof (*rule inv-split-reg-query'*[**where** *X*=‹*reg-1-3*› **and** *Y*=‹*reg-2-3*› **and** *H*=‹*reg-3-3*› **and** *K*=*K* **and** *?I1.0*=‹λ-. *ket-invariant* ({*0*} × *UNIV*)› **and** *?J1.0*=‹λ-. *ket-invariant* {(*0*+*d*, *Some d*)| *d*. *True*}›])
show ‹*query'* = (*reg-1-3*;(*reg-2-3*;*reg-3-3*)) *query'*›
by (*auto simp add*: *pair-Fst-Snd reg-1-3-def reg-2-3-def reg-3-3-def*)
show ‹*compatible reg-1-3 reg-2-3*› ‹*compatible reg-1-3 reg-3-3*› ‹*compatible reg-2-3 reg-3-3*›
by *simp-all*
show ‹*compatible-register-invariant reg-2-3* (*K x*)› **for** *x*
by (*auto intro*!: *compatible-register-invariant-compatible-register simp add*: *K-def*)
show ‹*compatible-register-invariant* (*reg-3-3 o function-at x*) (*K x*)› **for** *x*
by (*auto intro*!: *compatible-register-invariant-compatible-register simp add*: *K-def*)
show ‹*ket-invariant* {(*x*, *y*, *D*). *y* = *0*}
≤ (*SUP x*. *K x* ⊓ *lift-invariant* (*reg-2-3*;*reg-3-3* ∘ *function-at x*) (*ket-invariant* ({*0*} × *UNIV*)))›
apply (*simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*] *lift-inv-prod' lift-invariant-comp lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv case-prod-unfold*)
by *force*
show ‹*K x* ⊓ *lift-invariant* (*reg-2-3*;*reg-3-3* ∘ *function-at x*) (*ket-invariant* {(*0*+*d*, *Some d*)| *d*. *True*})
≤ *ket-invariant* {(*x*, *y*, *D*). *D x* = *Some y*}› **for** *x*
apply (*simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*]
lift-inv-prod' lift-invariant-comp lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv
case-prod-unfold)
by *fastforce*
show ‹*orthogonal-spaces* (*K x*) (*K x'*)› **if** ‹*x* ≠ *x'*› **for** *x x'*
using *that* **by** (*auto simp add*: *K-def orthogonal-spaces-lift-invariant*)
show ‹*preserves-ket query1'* ({*0*} × *UNIV*) {(*0*+*d*, *Some d*)| *d*. *True*} (*6* / *sqrt N*)›
apply (*rule preserve-query1'-fixY-output*[**where** b_i=*N*])
by (*auto intro*!: *simp*: *preserve-query1'-fixY-bound-output-def simp flip*: *N-def*)
show ‹*K x* ≤ *lift-invariant reg-1-3* (*ket-invariant* {*x*})› **for** *x*
by (*simp add*: *K-def*)
show ‹*6* / *sqrt N* ≥ *0*›
by *simp*
qed *simp*
qed

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* ⇒ *'mem update*›
and P :: ‹$'w$::*finite* ⇒ $'x$ ⇒ $('x \rightharpoonup 'y)$ ⇒ *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-invariant G* (*ket-invariant* $\{(w, x, y, D).\ y = 0 \wedge P\ w\ x\ D\}$))
(*lift-invariant G* (*ket-invariant* $\{(w, x, y, D).\ D\ x = Some\ y \wedge P\ w\ x\ D\}$))
(9 / *sqrt* N + 2 ∗ *sqrt* b / N)›
proof −
define K :: ‹$'w\times'x\times('x \rightharpoonup 'y)$ ⇒ *'mem ell2 ccsubspace*› **where**
‹$K = (\lambda(w,x,D).$ *lift-invariant G* (*ket-invariant* $\{(w, x, y, D') \mid y\ D'.\ D'(x{:=}None) = D\}$))›
define M :: ‹$('w\times'x\times('x \rightharpoonup 'y))$ *set*› **where**
‹$M = \{(w,x,D).\ D\ x = None\}$›
define *I1* :: ‹$'w\times'x\times('x \rightharpoonup 'y)$ ⇒ ($'y \times 'y$ *option*) *ell2 ccsubspace*› **where**
‹*I1* = $(\lambda(w,x,D).$ *ket-invariant* $\{(0, y) \mid y.\ P\ w\ x\ (D(x{:=}y))\}$)›
define *J1* :: ‹$'w\times'x\times('x \rightharpoonup 'y)$ ⇒ ($'y \times 'y$ *option*) *ell2 ccsubspace*› **where**
‹*J1* = $(\lambda(w,x,D).$ *ket-invariant* $\{(y, Some\ y) \mid y.\ P\ w\ x\ (D(x{:=}Some\ y))\}$)›

show *?thesis*
proof (*rule inv-split-reg-query′*[**where** X=‹*G o Snd o reg-1-3*› **and** Y=‹*G o Snd o reg-2-3*› **and** H=‹*G o Snd o reg-3-3*›
and K=K **and** *?I1.0*=*I1* **and** *?J1.0*=*J1* **and** M=M])
show ‹*F query′* = (*G* ∘ *Snd* ∘ *reg-1-3*;(*G* ∘ *Snd* ∘ *reg-2-3*;*G* ∘ *Snd* ∘ *reg-3-3*)) *query′*›
unfolding *reg-1-3-def reg-2-3-def reg-3-3-def assms*
by (*simp flip*: *comp-assoc*)
show ‹*compatible* (*G* ∘ *Snd* ∘ *reg-1-3*) (*G* ∘ *Snd* ∘ *reg-2-3*)› ‹*compatible* (*G* ∘ *Snd* ∘ *reg-1-3*) (*G* ∘ *Snd* ∘ *reg-3-3*)› ‹*compatible* (*G* ∘ *Snd* ∘ *reg-2-3*) (*G* ∘ *Snd* ∘ *reg-3-3*)›
by *simp-all*
show ‹*compatible-register-invariant* (*G* ∘ *Snd* ∘ *reg-2-3*) (*K wxD*)› **if** ‹$wxD \in M$› **for** *wxD*
by (*auto intro*!: *compatible-register-invariant-Snd-comp compatible-register-invariant-Fst*
simp add: *K-def assms compatible-register-invariant-chain reg-2-3-def*
comp-assoc M-def split!: *prod.split*)
show ‹*compatible-register-invariant* ((*G* ∘ *Snd o reg-3-3*) *o function-at* (*let* (*w*,*x*,*D*) = *wxD in x*)) (*K wxD*)› **if** ‹$wxD \in M$› **for** *wxD*
by (*auto intro*!: *compatible-register-invariant-Snd-comp compatible-register-invariant-function-at*
simp add: *K-def compatible-register-invariant-chain comp-assoc reg-3-3-def*
split!: *prod.split*)
show ‹*lift-invariant G* (*ket-invariant* $\{(w, x, y, D).\ y = 0 \wedge P\ w\ x\ D\}$)
≤ ($\bigsqcup wxD{\in}M.$ *K wxD* ⊓ *lift-invariant* (*G* ∘ *Snd* ∘ *reg-2-3*;*G* ∘ *Snd* ∘ *reg-3-3* ∘ *function-at* (*let* (*w*, *x*, *D*) = *wxD in x*)) (*I1 wxD*))›
by (*auto intro*!: *lift-invariant-mono*
simp add: *K-def M-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*] *register-comp-pair*
comp-assoc I1-def
lift-inv-prod′ lift-invariant-comp lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv case-prod-unfold
simp flip: *lift-invariant-inf lift-invariant-SUP*
split!: *prod.split*)
have *aux*: ‹D'(*fst* (*snd wxD*) := *None*) = *snd* (*snd wxD*) ⟹
D' (*fst* (*snd wxD*)) = *Some ya* ⟹
P (*fst wxD*) (*fst* (*snd wxD*)) ((*snd* (*snd wxD*))(*fst* (*snd wxD*) ↦ *ya*)) ⟹

P (*fst wxD*) (*fst* (*snd wxD*)) *D′*› **for** *wxD D′ ya*
by (*metis fun-upd-triv fun-upd-upd*)
show ‹*K wxD* ⊓ *lift-invariant* (*G* ∘ *Snd* ∘ *reg-2-3*;*G* ∘ *Snd* ∘ *reg-3-3* ∘ *function-at* (*let* (*w*, *x*, *D*) = *wxD in x*)) (*J1 wxD*)
≤ *lift-invariant G* (*ket-invariant* {(*w*, *x*, *y*, *D*). *D x* = *Some y* ∧ *P w x D*})› **if** ‹*wxD* ∈ *M*› **for** *wxD*
using *that*
by (*auto intro*!: *aux lift-invariant-mono*
simp add: *K-def J1-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*]
lift-inv-prod′ Times-Int-Times lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv
lift-invariant-comp register-comp-pair lift-Snd-inv
comp-assoc case-prod-unfold ket-invariant-tensor
simp flip: *lift-invariant-inf ket-invariant-SUP ket-invariant-UNIV*
split!: *prod.split*)
show ‹*orthogonal-spaces* (*K wxD*) (*K wxD′*)› **if** ‹*wxD* ∈ *M*› **and** ‹*wxD′* ∈ *M*› **and** ‹*wxD* ≠ *wxD′*› **for** *wxD wxD′*
using *that*
by (*auto simp add*: *K-def orthogonal-spaces-lift-invariant M-def split*!: *prod.split*)
show ‹*preserves query1′* (*I1 wxD*) (*J1 wxD*) (*9* / *sqrt N* + *2* ∗ *sqrt b* / *N*)› **if** ‹*wxD* ∈ *M*› **for** *wxD*
proof −
obtain *w x D* **where** *wxD*[*simp*]: ‹*wxD* = (*w*,*x*,*D*)›
by (*simp add*: *prod-eq-iff*)
from *that*
have *Dx*: ‹*D x* = *None*›
by (*simp add*: *M-def*)
have *I1*: ‹*I1* (*w*,*x*,*D*) = *ket-invariant* ({*0*} × {*y*. *P w x* (*D*(*x* := *y*))})›
by (*auto simp add*: *I1-def*)
have *presY*: ‹*preserves query1′* (*I1 wxD*) (*ket-invariant* {(*0*+*d*, *Some d*)| *d*. *True*}) (*6* / *sqrt* (*real N*))›
apply (*simp only*: *wxD I1*)
apply (*rule preserve-query1′-fixY-output*[**where** b_i=*N*])
apply (*simp add*: *N-def card-mono*)
using *sqrt-divide-self-eq*
by (*simp add*: *preserve-query1′-fixY-bound-output-def divide-inverse flip*: *N-def*)
have *presP1*: ‹*preserves query1′* (*I1 wxD*) (*ket-invariant* (*UNIV* × {*y*. *P w x* (*D*(*x* := *y*))})) (*3* / *sqrt N* + *2* ∗ *sqrt b* / *N*)›
if ‹¬ *P w x D*›
proof −
from *that Dx PNone* **have** *NoneI*: ‹(*None* ∈ {*y*. *P w x* (*D*(*x* := *y*))}) = *False*›
by *auto*
from *that Dx PNone* **have** *NoneJ*: ‹(*None* ∉ {*y*. *P w x* (*D*(*x* := *y*))}) = *True*›
by *auto*
define b_i **where** ‹b_i = *card* (*Some* −‘ {*y*. *P w x* (*D*(*x* := *y*))})›
define b_{j0} **where** ‹b_{j0} = *card* (− *Some* −‘ {*y*. *P w x* (*D*(*x* := *y*))})›
have ‹*sqrt* b_{j0} ∗ *sqrt* b_i / (*N* ∗ *sqrt N*) + *2* ∗ *sqrt* b_{j0} ∗ *sqrt* b_i / *N* + *1* / *sqrt N* + *sqrt* b_i / *N*
≤ *3* / *sqrt N* + *2* ∗ *sqrt b* / *N*›
proof −
have ‹b_{j0} = *N* − b_i›
by (*simp add*: *N-def* b_i*-def* b_{j0}*-def card-complement*)
then have ‹b_i ∗ b_{j0} ≤ *b*›
using *PSome*[*of D x w*] *that*
by (*auto intro*!: *simp*: b_i*-def Let-def Dx*)
have ‹b_i ≤ *N*›
apply (*simp add*: b_i*-def*)
by (*metis N-def card-complement diff-le-self double-complement*)

```
        have bbN: ‹sqrt b_j0 * sqrt b_i ≤ N›
          using ‹b_i ≤ N› ‹b_j0 = N − b_i›
          by (smt (verit, best) Extra-Ordered-Fields.sign-simps(5) of-nat-0-le-iff of-nat-diff
              ordered-comm-semiring-class.comm-mult-left-mono real-sqrt-ge-0-iff sqrt-sqrt)
        have bbb: ‹sqrt b_j0 * sqrt b_i ≤ sqrt b›
          using ‹b_i * b_j0 ≤ b›
       by (smt (verit) Num.of-nat-simps(5) cross3-simps(11) of-nat-mono real-sqrt-le-iff real-sqrt-mult)
        have sqrtNN: ‹sqrt N / N = 1 / sqrt N›
          by (metis div-by-1 inverse-divide of-nat-0-le-iff real-div-sqrt)
       have ‹sqrt b_j0 * sqrt b_i / (N * sqrt N) + 2 * sqrt b_j0 * sqrt b_i / N + 1 / sqrt N + sqrt b_i / N
          ≤ N / (N * sqrt N) + 2 * sqrt b / N + 1 / sqrt N + sqrt N / N›
          apply (intro add-mono divide-right-mono)
          by (auto intro!: ‹b_i ≤ N› bbN bbb)
        also have ‹... = 3 / sqrt N + 2 * sqrt b / N›
          by (simp add: nonzero-divide-mult-cancel-left sqrtNN)
        finally show ?thesis
          by −
      qed
      then show ?thesis
        apply (simp only: wxD I1)
        apply (rule preserve-query1'-fixY[where b_i=b_i and b_j0=b_j0])
        unfolding NoneI
        by (simp-all add: b_i-def b_j0-def preserve-query1'-fixY-bound-def)
    qed
    have presP2: ‹preserves query1' (I1 wxD) (ket-invariant (UNIV × {y. P w x (D(x := y))})) (3 /
sqrt N + 2 * sqrt b / N)›
      if ‹P w x D›
      apply (rewrite at ‹{y. P w x (D(x := y))}› to UNIV DEADID.rel-mono-strong)
      using that PNone Dx apply (metis UNIV-eq-I array-rules(5) fun-upd-triv mem-Collect-eq)
      by auto
    from presP1 presP2
    have presP: ‹preserves query1' (I1 wxD) (ket-invariant (UNIV × {y. P w x (D(x := y))})) (3 /
sqrt N + 2 * sqrt b / N)›
      by auto
    from preserves-intersect[OF - presY presP]
   have ‹preserves query1' (I1 wxD) (ket-invariant {(0 + d, Some d) |d. True} ⊓ ket-invariant (UNIV
× {y. P w x (D(x := y))}))
          ((6 / sqrt N) + (3 / sqrt N + 2 * sqrt b / N))›
      by auto
    then show ?thesis
      apply (rule arg-cong4[where f=preserves, THEN iffD1, rotated −1])
      by (auto intro!: simp: ket-invariant-inter J1-def)
  qed
  show ‹K wxD ≤ lift-invariant (G ∘ Snd ∘ reg-1-3) (ket-invariant {let (w, x, D) = wxD in x})› for
wxD
    by (auto intro!: lift-invariant-mono
        simp add: K-def lift-invariant-comp reg-1-3-def lift-Fst-ket-inv lift-Snd-ket-inv
        split!: prod.split)
  show ‹9 / sqrt N + 2 * sqrt b / N ≥ 0›
    by simp
  show ‹finite M›
    by simp
 qed
qed
```

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* ≡ (*X1*;(*Y1*;*H*)) *query′* $*_V$ *state1*›
  defines ‹*state3* ≡ (*X2*;(*Y2*;*H*)) *query′* $*_V$ *state2*›
  assumes [*register*]: ‹*mutually compatible* (*X1*,*X2*,*Y1*,*Y2*,*H*)›
  assumes [*iff*]: ‹*norm state1* ≤ *1*›
  assumes *dist1*: ‹*dist-inv* ((*X1*;*X2*);((*Y1*;*Y2*);*H*)) (*ket-invariant* {((*x1*,*x2*),(*y1*,*y2*),*D*). *y1* = *0* ∧ *y2* = *0*}) *state1* ≤ ε›
  shows ‹*dist-inv* ((*X1*;*X2*);((*Y1*;*Y2*);*H*)) (*ket-invariant* {((*x1*,*x2*),(*y1*,*y2*),*D*). *D x1* = *Some y1* ∧ *D x2* = *Some y2*}) *state3* ≤ ε + *20* / *sqrt N*›
proof −
  have [*iff*]: ‹*norm state2* ≤ *1*›
    by (*auto intro*!: *norm-cblinfun-apply-leq1I simp add*: *state2-def register-norm*)
  have *bound*: ‹*let c* = *card* {*y2*. (*x′* = *x2* ⟶ *y2* = *y′*) ∧ *x′* = *x2*} *in c* ∗ (*N* − *c*) ≤ *N*› **for** *x′ x2 y′*
    by (*cases* ‹*x′* = *x2*›, *auto*)
  from *dist1* **have** ‹*dist-inv* ((*X2*; *Y2*); (*X1*;(*Y1*;*H*)))
      (*ket-invariant* {(*x2y2*,(*x1*,*y1*,*D*)). *y1* = *0* ∧ *snd x2y2* = *0*}) *state1* ≤ ε›
    apply (*rule le-back-subst*)
    apply (*rule dist-inv-register-rewrite*, *simp*, *simp*)
    apply (*rewrite at* ‹((*X2*;*Y2*);(*X1*;(*Y1*;*H*)))› *to* ‹((*X1*;*X2*);((*Y1*;*Y2*);*H*)) *o* ((*reg-1-3 o Snd*; *reg-2-3 o Snd*); (*reg-1-3 o Fst*; (*reg-2-3 o Fst*; *reg-3-3*)))› *DEADID.rel-mono-strong*)
     apply (*simp add*: *register-pair-Snd register-pair-Fst flip*: *register-comp-pair comp-assoc*)
    apply (*subst lift-invariant-comp*, *simp*)
    apply *simp*
   by (*auto intro*!: *simp*: *lift-inv-prod′ reg-1-3-def reg-3-3-def reg-2-3-def lift-invariant-comp lift-Snd-ket-inv lift-Fst-ket-inv*
      ket-invariant-inter case-prod-unfold
      simp flip: *ket-invariant-SUP*)
  then have ‹*dist-inv* ((*X2*; *Y2*); (*X1*;(*Y1*;*H*)))
      (*ket-invariant* {(*x2y2*,(*x1*,*y1*,*D*)). *D x1* = *Some y1* ∧ *snd x2y2* = *0*}) *state2* ≤ ε + *9* / *sqrt* (*real N*)›
    unfolding *state2-def*
    apply (*rule dist-inv-preservesI*)
      apply (*rule preserves-ket-query′-output*[**where** *b*=*0*])
    by (*auto intro*!: *simp*: *register-pair-Snd register-norm simp del*: *o-apply*)
  then have ‹*dist-inv* ((*X1*; *Y1*); (*X2*;(*Y2*;*H*)))
      (*ket-invariant* {(*x1y1*,(*x2*,*y2*,*D*)). *y2* = *0* ∧ *D* (*fst x1y1*) = *Some* (*snd x1y1*)}) *state2* ≤ ε + *9* / *sqrt* (*real N*)›
    apply (*rule le-back-subst*)
    apply (*rule dist-inv-register-rewrite*, *simp*, *simp*)
    apply (*rewrite at* ‹((*X1*; *Y1*); (*X2*;(*Y2*;*H*)))› *to* ‹((*X2*; *Y2*); (*X1*;(*Y1*;*H*))) *o* ((*Snd o reg-1-3*; *Snd o reg-2-3*); (*Fst o Fst*; (*Fst o Snd*; *Snd o reg-3-3*)))› *DEADID.rel-mono-strong*)
     apply (*simp add*: *register-pair-Snd register-pair-Fst flip*: *register-comp-pair comp-assoc*)
    apply (*subst lift-invariant-comp*, *simp*)
    apply *simp*
   by (*auto intro*!: *simp*: *lift-inv-prod′ reg-1-3-def reg-3-3-def reg-2-3-def lift-invariant-comp lift-Snd-ket-inv lift-Fst-ket-inv*
      ket-invariant-inter case-prod-unfold
      simp flip: *ket-invariant-SUP*)
  then have ‹*dist-inv* ((*X1*; *Y1*); (*X2*;(*Y2*;*H*)))
      (*ket-invariant* {(*x1y1*,(*x2*,*y2*,*D*)). *D x2* = *Some y2* ∧ *D* (*fst x1y1*) = *Some* (*snd x1y1*)}) *state3* ≤ ε + *20* / *sqrt N*›

unfolding *state3-def*
apply (*rule dist-inv-preservesI*)
apply (*rule preserves-ket-query′-output*[**where** *b*=*N*])
by (*auto intro*!: *bound simp*: *register-pair-Snd register-norm simp del*: *o-apply split*!: *if-split-asm*)
then show ‹*dist-inv* ((*X1*;*X2*);((*Y1*;*Y2*);*H*)) (*ket-invariant* {((*x1*,*x2*),(*y1*,*y2*),*D*). *D x1* = *Some y1* ∧ *D x2* = *Some y2*}) *state3* ≤ *ε* + *20* / *sqrt N*›
apply (*rule le-back-subst*)
apply (*rule dist-inv-register-rewrite*, *simp*, *simp*)
apply (*rewrite at* ‹((*X1*; *Y1*); (*X2*;(*Y2*;*H*)))› *to* ‹((*X1*;*X2*);((*Y1*;*Y2*);*H*)) *o* ((*reg-1-3 o Fst*; *reg-2-3 o Fst*); (*reg-1-3 o Snd*; (*reg-2-3 o Snd*; *reg-3-3*)))› *DEADID.rel-mono-strong*)
apply (*simp add*: *register-pair-Snd register-pair-Fst flip*: *register-comp-pair comp-assoc*)
apply (*subst lift-invariant-comp*, *simp*)
apply *simp*
by (*auto intro*!: *simp*: *lift-inv-prod′ reg-1-3-def reg-3-3-def reg-2-3-def lift-invariant-comp lift-Snd-ket-inv lift-Fst-ket-inv*
ket-invariant-inter case-prod-unfold
simp flip: *ket-invariant-SUP*)
qed

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* ⊆ *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*›
proof (*rule preservesI′*)
show ‹*ε* ≥ *0*›
using - *ε* **apply** (*rule order.trans*)
by (*simp add*: *preserve-query1′-uncompute-bound-def*)
fix *ψ* :: ‹(′*y* × ′*y option*) *ell2*›
assume *ψ*: ‹*ψ* ∈ *space-as-set* (*ket-invariant* ((*UNIV* × *I*) ∩ {(*d*, *Some d*)| *d*. *True*}))›
assume ‹*norm ψ* = *1*›

define *I′ J′* **where** ‹*I′* = *Some* −‘ *I*› **and** ‹*J′* = *Some* −‘ *J*›
then have ‹((*UNIV* × *I*) ∩ {(*d*, *Some d*)| *d*. *True*}) = (*λd*. (*d*, *Some d*)) ‘ *I′*›
by *auto*
with *ψ* **have** *ψ′*: ‹*ψ* ∈ *space-as-set* (*ket-invariant* ((*λd*. (*d*, *Some d*)) ‘ *I′*))›
by *fastforce*
have [*simp*]: ‹*I′* ⊆ *J′*›
using *I′-def J′-def IJ* **by** *blast*
have *card-minus-J′*: ‹*card* (− *J′*) ≤ b_{j0}›
using *J′-def* b_{j0} **by** *force*

define *β* **where** ‹*β d* = *Rep-ell2 ψ* (*d*, *Some d*)› **for** *d*
have *β*: ‹*ψ* = (∑ *d*∈*I′*. *β d* $*_C$ *ket* (*d*, *Some d*))›
using *ell2-sum-ket-ket-invariant*[*OF ψ′*]
apply (*subst* (*asm*) *infsum-reindex*)

apply (*simp add*: *inj-on-convol-ident*)
by (*auto simp*: *β-def*)
have *βbound*: ‹($\sum$ *d*∈*I′*. (*cmod* (*β d*))^2) ≤ *1*› (**is** ‹*?lhs* ≤ *1*›)
apply (*subgoal-tac* ‹(*norm ψ*)^2 = *?lhs*›)
apply (*simp add*: ‹*norm ψ* = *1*›)
by (*simp add*: *β pythagorean-theorem-sum del*: *sum.insert*)

have [*simp*]: ‹*Some x* ∈ *J* ⟷ *x* ∈ *J′*› **for** *x*
by (*simp add*: *J′-def*)
have [*simp*]: ‹*x* ∈ *I′* ⟹ *x* ∈ *J′*› **for** *x*
using ‹*I′* ⊆ *J′*› **by** *blast*
have [*simp*]: ‹($\sum$ *x*∈*X*. *if x* ∉ *Y then f x else 0*) = ($\sum$ *x*∈*X*−*Y*. *f x*)› **if** ‹*finite X*› **for** *f* :: ‹*′y* ⇒ *′z*::*ab-group-add*› **and** *X Y*
apply (*rule sum.mono-neutral-cong-right*)
using *that* **by** *auto*
have [*simp*]: ‹*β yd* $*_C$ *a* $*_C$ *b* = *a* $*_C$ *β yd* $*_C$ *b*› **for** *yd a* **and** *b* :: ‹*′z*::*complex-vector*›
by *auto*
have [*simp*]: ‹*cmod α* = *inverse* (*sqrt N*)› ‹*cmod* ($α^2$) = *inverse N*› ‹*cmod* (*α*^*3*) = *inverse* (*N* ∗ *sqrt N*)› ‹*cmod* (*α*^*4*) = *inverse* (N^2)›
by (*auto simp*: *norm-mult numeral-3-eq-3 α-def inverse-eq-divide norm-divide norm-power power-one-over power2-eq-square power4-eq-xxxx*)
have [*simp*]: ‹*card I′* ≤ b_i›
by (*metis I′-def* b_i)

define *φ* **and** *PJ* :: ‹(*′y* ∗ *′y option*) *update*› **where**
‹*φ* = *query1′* $*_V$ *ψ*› **and** ‹*PJ* = *Proj* (*ket-invariant* (*UNIV* × −*J*))›
have [*simp*]: ‹*PJ* $*_V$ *ket* (*x*,*y*) = (*if y*∈−*J then ket* (*x*,*y*) *else 0*)› **for** *x y*
by (*simp add*: *Proj-ket-invariant-ket PJ-def*)
have *P0φ*: ‹*PJ* $*_V$ *φ* =
(*of-bool* (*None*∉*J*) ∗ *α*) $*_C$ ($\sum$ *d*∈*I′*. *β d* $*_C$ *ket* (*0*, *None*))
− (*of-bool* (*None*∉*J*) ∗ *α*^*3*) $*_C$ ($\sum$ *d*∈*I′*. $\sum$ *y*∈*UNIV*. *β d* $*_C$ *ket* (*y*, *None*))
− $α^2$ $*_C$ ($\sum$ *d*∈*I′*. $\sum$ *d′*∈−*J′*. *β d* $*_C$ *ket* (*d* + *d′*, *Some d′*))
− $α^2$ $*_C$ ($\sum$ *d*∈*I′*. $\sum$ *d′*∈−*J′*. *β d* $*_C$ *ket* (*0*, *Some d′*))
+ *α*^*4* $*_C$ ($\sum$ *d*∈*I′*. $\sum$ *y*∈*UNIV*. $\sum$ *d′′*∈−*J′*. *β d* $*_C$ *ket* (*y*, *Some d′′*))
›
(**is** ‹- = *?t1* − *?t2* − *?t3* − *?t4* + *?t5*›)
by (*simp add*: *φ-def β query1′ option-sum-split vimage-Compl*
cblinfun.add-right cblinfun.diff-right if-distrib Compl-eq-Diff-UNIV
vimage-singleton-inj sum-sum-if-eq sum.distrib scaleC-diff-right scaleC-sum-right
sum-subtractf case-prod-beta sum.cartesian-product′ scaleC-add-right add-diff-eq
cblinfun.scaleC-right cblinfun.sum-right
flip: *sum.Sigma add.assoc scaleC-scaleC*
cong del: *option.case-cong if-cong*)

have *norm-t1*: ‹*norm ?t1* ≤ *of-bool* (*None*∉*J*) ∗ *sqrt* b_i / *sqrt N*›
proof (*cases* ‹*None* ∈ *J*›)
case *True*
then show *?thesis*
by *simp*
next
case *False*
then have ‹*norm ?t1* = *inverse* (*sqrt N*) ∗ *norm* ($\sum$ *d*∈*I′*. *β d* $*_C$ *ket* (*0* :: *′y*, *None* :: *′y option*))›
by *simp*
also have ‹. . . ≤ *inverse* (*sqrt N*) ∗ *sqrt* b_i›
apply (*rule mult-left-mono*)

using - - *βbound* **apply** (*rule bound-coeff-sum2*)
by *auto*
also have ‹... = *of-bool* (*None*∉*J*) ∗ *sqrt* b_i / *sqrt N*›
using *False* **by** (*simp add*: *divide-inverse-commute*)
finally show *?thesis*
by −
qed

have *norm-t2*: ‹*norm ?t2* ≤ *of-bool* (*None*∉*J*) ∗ *sqrt* b_i / *N*›
proof (*cases* ‹*None* ∈ *J*›)
case *True*
then show *?thesis*
by *simp*
next
case *False*
have ∗: ‹*norm* (∑ *d*∈*I*′. β *d* $*_C$ *ket* (*y*, *None* :: ′*y option*)) ≤ *sqrt* b_i› **for** *y* :: ′*y*
using - - *βbound* **apply** (*rule bound-coeff-sum2*)
by *auto*

have ‹*norm ?t2* = *inverse* (*N* ∗ *sqrt N*) ∗ *norm* (∑ *y*∈*UNIV*. ∑ *d*∈*I*′. β *d* $*_C$ *ket* (*y* :: ′*y*, *None* :: ′*y option*))›
apply (*subst sum.swap*) **by** (*simp add*: *False*)
also have ‹... ≤ *inverse* (*N* ∗ *sqrt N*) ∗ (*sqrt* (*real N*) ∗ *sqrt* (*real* b_i))›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto simp add*: *cinner-sum-right cinner-sum-left N-def*)
also have ‹... = *of-bool* (*None*∉*J*) ∗ *sqrt* b_i / *N*›
using *False* **by** (*simp add*: *divide-inverse-commute N-def*)
finally show *?thesis*
by −
qed

have *norm-t3*: ‹*norm ?t3* ≤ *sqrt* b_{j0} / *N*›
proof −
have ∗: ‹*norm* (∑ *d*∈*I*′. β *d* $*_C$ *ket* (*d* + *d*′, *Some d*′)) ≤ *sqrt* (*1*::*nat*)› **for** *d*′ :: ′*y*
using - - *βbound* **apply** (*rule bound-coeff-sum2*)
by (*auto simp add*: *card-le-Suc0-iff-eq*)

have ‹*norm ?t3* = *inverse N* ∗ *norm* (∑ *d*′∈− *J*′. ∑ *d*∈*I*′. β *d* $*_C$ *ket* (*d* + *d*′, *Some d*′))›
apply (*subst sum.swap*) **by** *simp*
also have ‹... ≤ *inverse N* ∗ *sqrt* b_{j0}›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
using *card-minus-J*′ **by** (*auto simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹... = *sqrt* b_{j0} / *N*›
by (*simp add*: *divide-inverse-commute*)
finally show *?thesis*
by −
qed

have *norm-t4*: ‹*norm ?t4* ≤ *sqrt* b_i ∗ *sqrt* b_{j0} / *N*›
proof −
have ∗: ‹*norm* (∑ *d*∈*I*′. β *d* $*_C$ *ket* (*0*, *Some d*′)) ≤ *sqrt* b_i› **for** *d*′ :: ′*y*
using - - *βbound* **apply** (*rule bound-coeff-sum2*)
by *auto*

have ‹*norm ?t4 = inverse N ∗ norm* ($\sum$ *d′∈− J′*. $\sum$ *d∈I′*. *β d ∗*$_C$ *ket* (*0* :: *′y*, *Some d′* :: *′y option*))›
apply (*subst sum.swap*) **by** *simp*
also have ‹. . . ≤ *inverse N ∗* (*sqrt* b_{j0} *∗ sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
by (*auto intro*!: *card-minus-J′ mult-right-mono simp add*: *cinner-sum-right cinner-sum-left*)
also have ‹. . . = *sqrt* b_i *∗ sqrt* b_{j0} / *N*›
by (*simp add*: *divide-inverse-commute*)
finally show *?thesis*
by −
qed

have *norm-t5*: ‹*norm ?t5* ≤ *sqrt* b_i *∗ sqrt* b_{j0} / (*N ∗ sqrt N*)›
proof −
have ∗: ‹*norm* ($\sum$ *d∈I′*. *β d ∗*$_C$ *ket* (*fst yd′′*, *Some* (*snd yd′′*))) ≤ *sqrt* b_i› **for** *yd′′* :: ‹*′y × ′y*›
using - - *βbound* **apply** (*rule bound-coeff-sum2*)
by *auto*

have ‹*norm ?t5 = inverse* (N^2) *∗ norm* ($\sum$ *yd′′∈UNIV×−J′*. $\sum$ *d∈I′*. *β d ∗*$_C$ *ket* (*fst yd′′* :: *′y*, *Some* (*snd yd′′*)))›
apply (*simp add*: *sum.cartesian-product′ sum.reindex N-def*)
apply (*subst* (*2*) *sum.swap*) **apply** (*subst sum.swap*)
by (*rule refl*)
also have ‹. . . ≤ *inverse* (N^2) *∗* (*sqrt N ∗ sqrt* b_{j0} *∗ sqrt* b_i)›
apply (*rule mult-left-mono*)
using ∗ **apply** (*rule norm-ortho-sum-bound*)
using *card-minus-J′* **by** (*auto intro*!: *mult-right-mono simp add*: *cinner-sum-right cinner-sum-left cinner-ket real-sqrt-mult N-def*)
also have ‹. . . = *sqrt* b_i *∗ sqrt* b_{j0} / (*N ∗ sqrt N*)›
by (*smt* (*verit*, *ccfv-threshold*) *field-class.field-divide-inverse mult.commute of-nat-0-le-iff of-nat-power power2-eq-square real-divide-square-eq real-sqrt-mult-self times-divide-times-eq*)
finally show *?thesis*
by −
qed

have ‹*norm* (*PJ ∗*$_V$ *φ*) ≤ *of-bool* (*None∉J*) *∗ sqrt* b_i / *sqrt N* + *of-bool* (*None∉J*) *∗ sqrt* b_i / *N* + *sqrt* b_{j0} / *N* + *sqrt* b_i *∗ sqrt* b_{j0} / *N* + *sqrt* b_i *∗ sqrt* b_{j0} / (*N ∗ sqrt N*)›
unfolding *P0φ*
apply (*rule norm-triangle-le-diff norm-triangle-le*, *rule add-mono*)+
apply (*rule norm-t1*)
apply (*rule norm-t2*)
apply (*rule norm-t3*)
apply (*rule norm-t4*)
by (*rule norm-t5*)
also have ‹. . . = *preserve-query1′-uncompute-bound* (*None∉J*) b_i b_{j0}›
by (*auto simp*: *preserve-query1′-uncompute-bound-def mult.commute mult.left-commute*)
also have ‹. . . ≤ *ε*›
by (*simp add*: *ε*)
finally show ‹*norm* (*Proj* (− *ket-invariant* (*UNIV* × *J*)) *∗*$_V$ *φ*) ≤ *ε*›
unfolding *PJ-def*
apply (*subst ket-invariant-compl*[*symmetric*])
by *simp*
qed

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*)))›
proof −
note [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
let *?goal* = *?thesis*
have [*register*]: ‹*register* (*Fst o X*; (*Fst o Y*; *Snd* :: - ⇒ (*'mem* × (*'x* ⇀ *'y*)) *update*))›
by *simp*
from *register-decomposition*[*OF this*]
have ‹*let 'd*::*type* = *register-decomposition-basis* (*Fst o X*; (*Fst o Y*; *Snd* :: - ⇒ (*'mem* × (*'x* ⇀ *'y*)) *update*)) *in ?thesis*›
proof *with-type-mp*
case *with-type-mp*
then obtain *U* :: ‹(((*'x* × *'y* × (*'x* ⇒ *'y option*)) × *'d*) *ell2* $\Rightarrow_{CL}$ (*'mem* × (*'x* ⇒ *'y option*)) *ell2*›
where ‹*unitary U*› **and** *unwrap*: ‹(*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *ϑ* = *sandwich U* $*_V$ (*ϑ* $\otimes_o$ *id-cblinfun*)› **for** *ϑ*
by *blast*
have *unwrap-Snd*: ‹*Snd a* = *sandwich U* $*_V$ ((*id-cblinfun* $\otimes_o$ (*id-cblinfun* $\otimes_o$ *a*)) $\otimes_o$ *id-cblinfun*)› **for** *a*
apply (*rewrite at Snd to* ‹(*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *o Snd o Snd*› *DEADID.rel-mono-strong*)
apply (*simp add*: *register-pair-Snd*)
by (*simp add*: *unwrap Snd-def*)
have *unwrap-Fst-XY*: ‹(*Fst o* (*X*;*Y*)) *a* = *sandwich U* $*_V$ *assoc* (*a* $\otimes_o$ *id-cblinfun*) $\otimes_o$ *id-cblinfun*›
for *a*
apply (*rewrite at* ‹*Fst o* (*X*;*Y*)› *to* ‹(*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *o assoc o Fst*› *DEADID.rel-mono-strong*)
apply (*simp add*: *register-pair-Fst register-comp-pair*)
by (*simp add*: *only*: *o-apply unwrap Fst-def*)

have ‹*standard-query* $\otimes_o$ *id-cblinfun* o_{CL}
(*id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *proj* (*ket* (*Some* ∘ *h*))) $\otimes_o$ *id-cblinfun* =
assoc (*function-oracle h* $\otimes_o$ *id-cblinfun*) $\otimes_o$ *id-cblinfun* o_{CL}
(*id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *proj* (*ket* (*Some* ∘ *h*))) $\otimes_o$ *id-cblinfun*›
by (*auto intro*!: *equal-ket*
simp: *tensor-op-ket tensor-ell2-ket proj-ket-x-y-ofbool std-query-Some assoc-ell2-sandwich sandwich-apply function-oracle-apply*)
then show *?goal*
by (*auto intro*!: *arg-cong*[**where** *f*=‹*sandwich U*›]

simp add: *unwrap unwrap-Snd unwrap-Fst-XY*[*unfolded o-def*] *sandwich-arg-compose* ‹*unitary U*›)
qed
from *this*[*cancel-with-type*]
show *?goal*
by −
qed

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 ‹$\bigwedge H\ x\ y\ z.\ H\ x = Some\ z \Longrightarrow standard\text{-}query *_V (ket\ (x,y,H)) = ket\ (x,\ y + z,\ H)$›
assumes ‹*compatible X Y*›
shows ‹$(Fst\ o\ X;\ (Fst\ o\ Y;\ Snd))\ standard\text{-}query *_V (\psi \otimes_s ket\ (Some \circ h))$
$= Fst\ ((X;Y)\ (function\text{-}oracle\ h)) *_V (\psi \otimes_s ket\ (Some \circ h))$›
proof −
have ‹$(Fst\ o\ X;\ (Fst\ o\ Y;\ Snd))\ standard\text{-}query *_V (\psi \otimes_s ket\ (Some \circ h))$
$= (Fst\ o\ X;\ (Fst\ o\ Y;\ Snd))\ standard\text{-}query *_V Snd\ (proj\ (ket\ (Some\ o\ h))) *_V (\psi \otimes_s ket\ (Some \circ h))$›
by (*simp add*: *proj-ket-x-y-ofbool*)
also have ‹$\ldots = Fst\ ((X;Y)\ (function\text{-}oracle\ h)) *_V Snd\ (proj\ (ket\ (Some\ o\ h))) *_V (\psi \otimes_s ket\ (Some \circ h))$›
apply (*subst cblinfun-apply-cblinfun-compose*[*symmetric*])+
by (*simp-all add*: *assms swap-function-oracle-measure-generic*)
also have ‹$\ldots = Fst\ ((X;Y)\ (function\text{-}oracle\ h)) *_V (\psi \otimes_s ket\ (Some \circ h))$›
by (*simp add*: *Proj-fixes-image ccspan.rep-eq complex-vector.span-base flip*: *cblinfun-apply-cblinfun-compose*)
finally show *?thesis*
by −
qed

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*)›
  using *is-QueryStep.cases* **by** *blast*

lemma *is-ProgramStep-QueryStep*[*iff*]: ‹¬ *is-ProgramStep* (*QueryStep X Y*)›
  by (*simp add*: *is-ProgramStep.simps*)

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*›
  by (*simp add*: *valid-program-def*)

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

lemma *valid-program-empty*[*iff*]: ‹*valid-program* []›
  by (*simp add*: *valid-program-def*)

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* $*_V$ ψ›
| ‹*exec-program-step h* (*QueryStep X Y*) ψ = (*X*;*Y*) (*function-oracle h*) $*_V$ ψ›

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* $*_V$ ψ›
| ‹*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*)›
  apply (*cases step*)
  by (*auto intro*!: *cblinfun.bounded-clinear-right simp add*: *exec-program-step-with.simps*[*abs-def*])

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

by (*simp add*: *exec-program-def*)
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)›
by (*simp add*: *exec-program-with-def*)

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*›
proof (*cases program-step*)
case (*ProgramStep U*)
with *that* **have** ‹*isometry U*›
by *simp*
then have ‹*norm* (*Fst U*) = *1*›
by (*simp add*: *register-norm norm-isometry*)
then show *?thesis*
apply (*simp add*: *ProgramStep*)
by (*smt* (*verit*, *del-insts*) ‹*norm* (*Fst U*) = *1*› *mult-cancel-right1 norm-cblinfun*)
next
case (*QueryStep X Y*)
with *that*
have [*register*]: ‹*compatible X Y*›
using *valid-program-step.simps* **by** *blast*
have [*register*]: ‹*register* (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*))›
by *simp*
have ‹*norm* ((*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *oracle* $*_V$ ψ) $\leq$ *norm* ((*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *oracle*) $*$ *norm* ψ›
using *norm-cblinfun* **by** *blast*
also have ‹. . . = *norm oracle* $*$ *norm* ψ›
by (*simp add*: *register-norm*)
also have ‹. . . $\leq$ *norm* ψ›
by (*simp add*: *mult-left-le-one-le that*(*2*))
finally show *?thesis*
by (*simp add*: *QueryStep*)
qed

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*
proof (*insert that*(*2*), *induction program rule*: *rev-induct*)
case *Nil*
then show *?case*
by *simp*
next
case (*snoc program-step program*)
then have ‹*valid-program-step program-step*› **and** ‹*valid-program program*›
by (*auto simp*: *valid-program-append*)
have ‹*norm* (*exec-program-step-with oracle program-step* (*exec-program-with oracle program* ψ)) $\leq$ *norm* (*exec-program-with oracle program* ψ)›
by (*smt* (*verit*, *del-insts*) ‹*valid-program-step program-step*› *mult-left-le-one-le norm-exec-program-step-with norm-ge-zero that*(*1*))
also have ‹. . . $\leq$ *norm* ψ›
using ‹*valid-program program*› *snoc.IH* **by** *force*
finally show *?case*
by (*simp add*: *exec-program-with-append*)
qed

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›
proof (*cases program-step*)
case (*ProgramStep U*)
with *assms* **have** ‹*isometry U*›
by *simp*
with *ProgramStep* **show** *?thesis*
by (*simp add*: *isometry-preserves-norm register-isometry*)
next
case (*QueryStep X Y*)
with *assms* **have** [*register*]: ‹*compatible X Y*›
by *simp*
have ‹*register* (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*))›
by *simp*
with *assms* **have** ‹*isometry* ((*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *query*)›
using *register-isometry* **by** *blast*
then show *?thesis*
by (*simp add*: *QueryStep isometry-preserves-norm*)
qed

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*)›
proof (*cases p*)
case (*ProgramStep U*)
then have ‹*isometry* (*Q U*)›
using *assms register-isometry valid-program-step.simps*(*2*) **by** *blast*
then show *?thesis*
using *ProgramStep* **by** *auto*
next
case (*QueryStep X Y*)
with *assms* **show** *?thesis*
by *simp*
qed

lemma *valid-program-lift*:
assumes ‹*register Q*› **and** ‹*valid-program p*›
shows ‹*valid-program* (*lift-program Q p*)›
using *assms*(*2*)
by (*auto simp add*: *valid-program-def lift-program-def list.pred-map list-all-length valid-program-step-lift assms*(*1*))

lemma *lift-program-empty*[*simp*]: ‹*lift-program Q* [] = []›
by (*simp add*: *lift-program-def*)

lemma *lift-program-cons*: ‹*lift-program Q* (*program-step* # *program*) = *lift-program-step Q program-step* # *lift-program Q program*›
by (*simp add*: *lift-program-def*)

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

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

lemma *filter-is-QueryStep-lift-program*: ‹*filter is-QueryStep* (*lift-program Q program*) = *lift-program Q* (*filter is-QueryStep program*)›
apply (*induction program*)
by (*auto simp*: *lift-program-def*)

lemma *length-lift-program*[*simp*]: ‹*length* (*lift-program Q program*) = *length program*›
apply (*induction program*)
by (*auto simp*: *lift-program-def*)

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

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

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* $\psi \otimes_s \varphi$›
proof (*cases program-step*)
case (*ProgramStep U*)
then show *?thesis*
by (*simp add*: *Fst-def tensor-op-ell2*)
next
case (*QueryStep X Y*)
with *assms* **have** [*register*]: ‹*compatible X Y*›
using *valid-program-step.simps*(*1*) **by** *blast*
have ‹($Fst \circ (X;Y)$) (*function-oracle h*) $*_V$ $\psi \otimes_s \varphi$ = ($(X;Y)$ (*function-oracle h*) $*_V$ ψ) $\otimes_s \varphi$›
by (*simp add*: *Fst-def tensor-op-ell2*)
then show *?thesis*
by (*simp add*: *QueryStep register-comp-pair*)
qed

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* $\psi \otimes_s \varphi$›
apply (*insert assms*, *induction program rule*:*rev-induct*)
by (*simp-all add*: *lift-program-append exec-program-append lift-program-cons valid-program-append exec-lift-program-step-Fst*)

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* $\geq$ *0*›
by (*simp add*: *measurement-probability-def*)

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›
proof −
have *ortho1*: ‹*orthogonal-spaces* (*ket-invariant A*) (*ket-invariant B*)›
using *assms*(*2*) **by** *force*
have *ortho2*: ‹*is-orthogonal* (*Q* (*Proj* (*ket-invariant A*)) $*_V$ ψ) (*Q* (*Proj* (*ket-invariant B*)) $*_V$ ψ)›
proof −
from *ortho1* **have** ‹*orthogonal-spaces* (*lift-invariant Q* (*ket-invariant A*)) (*lift-invariant Q* (*ket-invariant B*))›
by (*simp add*: *orthogonal-spaces-lift-invariant assms*)
then have ‹*is-orthogonal* (*Proj* (*lift-invariant Q* (*ket-invariant A*)) ψ) (*Proj* (*lift-invariant Q* (*ket-invariant B*)) ψ)›
by (*metis Proj-lift-invariant assms*(*1*) *cblinfun-apply-in-image lift-invariant-def orthogonal-spaces-def*)
moreover have ‹*Proj* (*lift-invariant Q* (*ket-invariant A*)) = *Q* (*Proj* (*ket-invariant A*))›
by (*simp add*: *Proj-ket-invariant-butterfly Proj-lift-invariant assms*(*1*) *butterfly-eq-proj*)
moreover have ‹*Proj* (*lift-invariant Q* (*ket-invariant B*)) = *Q* (*Proj* (*ket-invariant B*))›
by (*simp add*: *Proj-lift-invariant assms*(*1*) *ket-invariant-def*)
ultimately show *?thesis*
by *simp*
qed
have ‹(*norm* (*Q* (*Proj* (*ket-invariant* ($A \cup B$))) ψ))^2 = (*norm* (*Q* (*Proj* (*ket-invariant A* ⊔ *ket-invariant B*)) ψ))^2›
by (*simp add*: *ket-invariant-union*)
also have ‹... = (*norm* (*Q* (*Proj* (*ket-invariant A*) + (*Proj* (*ket-invariant B*))) ψ))^2›
by (*metis Proj-sup ortho1*)
also have ‹... = (*norm* (*Q* (*Proj* (*ket-invariant A*)) ψ + *Q* (*Proj* (*ket-invariant B*)) ψ))^2›
by (*simp add*: *complex-vector.linear-add clinear-register* ‹*register Q*› *cblinfun.add-left*)
also have ‹... = (*norm* (*Q* (*Proj* (*ket-invariant A*)) ψ))^2 + (*norm* (*Q* (*Proj* (*ket-invariant B*)) ψ))^2›
by (*simp add*: *pythagorean-theorem ortho2*)
finally show *?thesis*
by −
qed

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›
proof (*use* ‹*finite F*› **in** *induction*)
case *empty*
show *?case*
apply *simp*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *cancel-comm-monoid-add-class.diff-cancel cblinfun.zero-left register-minus*)
next
case (*insert x F*)
have ‹(*norm* (*Q* (*Proj* (*ket-invariant* (*insert x F*))) $*_V$ *ψ*))^2 = (*norm* (*Q* (*Proj* (*ket-invariant F*)) $*_V$ *ψ*))^2 + (*norm* (*Q* (*Proj* (*ket-invariant* {*x*})) $*_V$ *ψ*))^2›
apply (*rewrite at* ‹*insert x F*› *to* ‹*F* ∪ {*x*}› *DEADID.rel-mono-strong*)
apply *simp*
apply (*subst norm-register-Proj-ket-invariant-union*)
by (*simp-all add*: *assms insert.hyps*)
also have ‹. . . = (*norm* (*Q* (*proj* (*ket x*)) *ψ*))^2 + (*norm* (*Q* (*Proj* (*ccspan* (*ket* ‘ *F*))) *ψ*))^2›
by (*simp add*: *ket-invariant-def*)
also have ‹. . . = (*norm* (*Q* (*proj* (*ket x*)) *ψ*))^2 + *sum* (*measurement-probability Q ψ*) *F*›
by (*simp add*: *insert.IH ket-invariant-def*)
also have ‹. . . = *sum* (*measurement-probability Q ψ*) (*insert x F*)›
by (*simp add*: *insert.hyps measurement-probability-def*)
finally show *?case*
by *simp*
qed

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›
proof −
define *m s* **where** ‹*m x* = *measurement-probability Q ψ x*› **and** ‹*s A* = (*norm* (*Q* (*Proj* (*ket-invariant A*)) *ψ*))^2› **for** *A x*
have *sum-m-fin*: ‹*sum m F* = *s F*› **if** ‹*finite F*› **for** *F*
by (*simp add*: *measurement-probability-sum m-def s-def that assms*)
have *s-mono*: ‹*s A* ≤ *s B*› **if** ‹*A* ⊆ *B*› **for** *A B*
proof −
have [*simp*]: ‹*A* ∪ *B* = *B*›
using *that* **by** *blast*
have ‹*s A* ≤ *s A* + *s* (*B*−*A*)›
by (*simp add*: *s-def*)
also have ‹. . . = *s B*›
apply (*simp add*: *s-def*)
apply (*subst norm-register-Proj-ket-invariant-union*[*symmetric*])
using *that*
by (*auto simp*: *assms*)
finally show *?thesis*
by −
qed
have *m-pos*: ‹*m x* ≥ *0*› **for** *x*
by (*simp add*: *m-def measurement-probability-nonneg*)
show *summable*: ‹*m summable-on A*› **for** *A*
apply (*rule nonneg-bounded-partial-sums-imp-summable-on*[**where** *C*=‹*s UNIV*›])
using *s-mono*[*of - UNIV*] *sum-m-fin*
by (*auto intro*!: *eventually-finite-subsets-at-top-weakI simp*: *m-pos*)

then show ‹*infsum m A* ≤ *s A*›
apply (*rule infsum-le-finite-sums*)
using *s-mono*[*of* - *A*] *sum-m-fin*
by *auto*
qed

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* $(-I)$) $\psi)^2$›
proof −
have ‹$(\sum x{\in}I.$ *measurement-probability Q* ψ *x*) = (*norm* (*Q* (*Proj* (*ket-invariant I*)) $*_V$ $\psi))^2$›
by (*simp add*: *measurement-probability-sum*)
then show *?thesis*
by (*simp add*: *dist-inv-def ket-invariant-compl*)
qed

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*)) $\psi)^2$›
by (*simp add*: *dist-inv-avg-def real-sqrt-pow2 divide-nonneg-pos*
sum-nonneg dist-inv-measurement-probability)

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*))
≤ *dist-inv-avg Q I* ψ›
proof (*insert* ‹*valid-program prog*›, *induction prog rule*:*rev-induct*)
case *Nil*
with *assms* **show** *?case*
by *simp*
next
case (*snoc program-step prog*)
show *?case*
proof (*cases program-step*)
case (*ProgramStep U*)
have ‹*dist-inv-avg Q I* (λh. *exec-program h* (*lift-program R* (*prog* @ [*program-step*])) (ψ *h*))
≤ (*MAX h*::*'x*⇒*'y*. *norm U*) ∗ *dist-inv-avg Q I* (λh. *exec-program h* (*lift-program R prog*) (ψ *h*))›
apply (*simp add*: *lift-program-append lift-program-cons exec-program-append ProgramStep del*: *range-constant Max-const*)
apply (*rule dist-inv-avg-apply-compatible*[**where** *R*=‹λ-. *R*›])
by *simp*
also have ‹. . . ≤ *dist-inv-avg Q I* ψ›
using *snoc*
by (*simp-all add*: *ProgramStep norm-isometry valid-program-append*)
finally show *?thesis*
by −
next
case (*QueryStep X Y*)

```
    with snoc have [register]: ‹compatible X Y› by (simp add: valid-program-append)
    have ‹dist-inv-avg Q I (λh. exec-program h (lift-program R (prog @ [program-step])) (ψ h))
      ≤ (MAX h::'x⇒'y. norm (function-oracle h)) ∗ dist-inv-avg Q I (λh. exec-program h (lift-program
R prog) (ψ h))›
      apply (simp add: lift-program-append lift-program-cons exec-program-append QueryStep
          del: range-constant Max-const norm-function-oracle)
       apply (rule dist-inv-avg-apply-compatible[where R=‹λ-. (R ∘ X;R ∘ Y)›])
      by simp
    also have ‹. . . ≤ dist-inv-avg Q I ψ›
      using snoc
      by (simp-all add: QueryStep valid-program-append)
    finally show ?thesis
      by −
  qed
qed

lemma dist-inv-exec'-compatible:
  fixes prog
  assumes ‹valid-program prog›
  assumes normU: ‹norm U ≤ 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) ≤ dist-inv Q I ψ›
proof (insert ‹valid-program prog›, induction prog rule:rev-induct)
  case Nil
  with assms show ?case
    by simp
next
  case (snoc program-step prog)
  show ?case
  proof (cases program-step)
    case (ProgramStep V)
    have ‹dist-inv Q I (exec-program-with U (lift-program R (prog @ [program-step])) ψ)
      ≤ norm V ∗ dist-inv Q I (exec-program-with U (lift-program R prog) ψ)›
      apply (simp add: lift-program-append exec-program-with-append lift-program-cons ProgramStep)
      using dist-inv-apply-compatible[OF compatQ1]
      by simp
    also have ‹. . . ≤ dist-inv Q I ψ›
     using ProgramStep norm-isometry snoc(1) snoc.prems valid-program-append exec-program-with-append
by fastforce
    finally show ?thesis
      by −
  next
    case (QueryStep X Y)
    with snoc have [register]: ‹compatible X Y› by (simp add: valid-program-append)
    then have compat[register]: ‹compatible Q (Fst ∘ (R ∘ X);(Fst ∘ (R ∘ Y);Snd))›
      by (auto intro!: compatible3' compatible-comp-inner simp flip: comp-assoc)
    have ‹dist-inv Q I (exec-program-with U (lift-program R (prog @ [program-step])) ψ)
      ≤ norm U ∗ dist-inv Q I (exec-program-with U (lift-program R prog) ψ)›
      apply (simp add: lift-program-append lift-program-cons QueryStep exec-program-with-append)
      by (rule dist-inv-apply-compatible[OF compat])
    also have ‹. . . ≤ dist-inv Q I ψ›
      by (smt (verit, ccfv-SIG) assms(2) dist-inv-pos mult-cancel-right1
          mult-left-le-one-le snoc(1) snoc.prems valid-program-append zero-le-mult-iff)
```

finally show *?thesis*
by −
qed
qed

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) ≤ ε›
assumes *init*: ‹ψ0 ∈ space-as-set (lift-invariant R (J 0))›
assumes ‹J (query-count program) ≤ I›
assumes ‹valid-program program›
assumes ‹$\bigwedge$X Y i. compatible X Y ⟹ 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 ≤ 1›
assumes ‹norm ψ0 ≤ 1›
shows ‹dist-inv R I (exec-program-with oracle program ψ0) ≤ ε›
proof −
note [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
define *f* **where** ‹f n = ($\sum$ i<n. g i)› **for** *n*
from ‹compatible R Fst› **have** [*register*]: ‹register R›
using *compatible-register1* **by** *blast*
have ‹dist-inv R (J (query-count program)) (exec-program-with oracle program ψ0) ≤ f (query-count program)›
proof (*insert* ‹valid-program program›, *induction program rule*:*rev-induct*)
case *Nil*
from *init*
have ‹dist-inv R (J 0) ψ0 = 0›
by (*simp add*: *dist-inv-0-iff*)
then show *?case*
by (*simp add*: *assms*(*2*) *query-count-def f-def*)
next
case (*snoc program-step program*)
from *snoc.prems* **have** ‹valid-program program›
using *valid-program-append* **by** *blast*
from *snoc.prems* **have** ‹valid-program-step program-step›
by (*simp add*: *valid-program-append*)
define *i* **where** ‹i = query-count program›
show *?case*
proof (*cases program-step*)
case (*ProgramStep U*)
with ‹valid-program-step program-step›
have [*iff*]: ‹isometry U›
by *simp*
have ‹preserves (Fst U) (lift-invariant R (J i)) (lift-invariant R (J i)) 0›
apply (*rule-tac preserves-compatible*[**where** *F*=*Fst*])
using ‹compatible R Fst› *compatible-register-invariant-compatible-register compatible-sym* **apply** *blast*
by *simp*
then have ‹dist-inv R (J i) (Fst U $*_V$ exec-program-with oracle program ψ0) ≤ f i›
apply (*rule dist-inv-leq-if-preserves*[*THEN order-trans*])
using *snoc.IH*[*OF* ‹valid-program program›]
by (*simp-all add*: *norm-isometry register-isometry query-count-def i-def*)

```
      then show ?thesis
        by (simp add: ProgramStep exec-program-with-append i-def query-count-def)
    next
      case (QueryStep X Y)
      with ‹valid-program-step program-step›
      have [register]: ‹compatible X Y›
        by simp
      with assms have pres: ‹preserves ((Fst ∘ X;(Fst ∘ Y;Snd)) oracle) (lift-invariant R (J i))
          (lift-invariant R (J (Suc i))) (g i)›
        by fast
      have fg′: ‹norm ((Fst ∘ X;(Fst ∘ Y;Snd)) oracle) ∗ f i + g i ∗ norm (exec-program-with oracle
program ψ0) ≤ f (Suc i)›
      proof −
        have ‹norm ((Fst ∘ X;(Fst ∘ Y;Snd)) oracle) ≤ 1›
          apply (subst register-norm[where a=oracle])
          by (simp-all add: assms)
        moreover have ‹norm (exec-program-with oracle program ψ0) ≤ 1›
          apply (rule norm-exec-program-with[THEN order-trans])
          using ‹valid-program program› assms by simp-all
        moreover have ‹f i + g i ≤ f (Suc i)›
          by (simp add: f-def)
        moreover have gpos: ‹g i ≥ 0› for i
          using ‹compatible X Y› assms(6) preserves-def by blast
        moreover have ‹f i ≥ 0›
          by (auto intro!: sum-nonneg gpos simp: f-def)
        ultimately
        show ?thesis
          by (smt (verit) mult-left-le mult-left-le-one-le norm-ge-zero)
      qed
      show ?thesis
        apply (simp add: QueryStep exec-program-with-append flip: i-def)
        using pres apply (rule dist-inv-leq-if-preserves[THEN order-trans])
          apply (simp, simp)
        using snoc.IH[OF ‹valid-program program›, folded i-def]
        by (smt (verit, ccfv-SIG) fg′ mult-left-mono norm-ge-zero)
    qed
  qed
  with assms show ?thesis
    using ‹register R›
    by (smt (verit, best) dist-inv-mono f-def)
qed
```

7.6 Random Oracles

```
lemma standard-query-for-fixed-function-generic:
  fixes standard-query
  assumes ‹⋀H x y z. H x = Some z ⟹ standard-query ∗V (ket (x,y,H)) = ket (x, y + z, H)›
  assumes ‹valid-program program›
  shows ‹exec-program h program initial-state ⊗s ket (Some o h)
      = exec-program-with standard-query program (initial-state ⊗s ket (Some ∘ h))›
proof (insert ‹valid-program program›, induction program rule: rev-induct)
  case Nil
  then show ?case
    by simp
next
  case (snoc program-step prog)
```

then have [*simp*]: ‹*valid-program prog*›
  using *list-all-append valid-program-def* **by** *blast*
show *?case*
proof (*cases program-step*)
  case (*ProgramStep U*)
  have ‹*exec-program h* (*prog* @ [*program-step*]) *initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*) = (*U* $*_V$ *exec-program h prog initial-state*) $\otimes_s$ *ket* (*Some* ∘ *h*)›
    by (*simp add*: *ProgramStep exec-program-append*)
  also have ‹. . . = *Fst U* $*_V$ *exec-program h prog initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*)›
    by (*simp add*: *Fst-def tensor-op-ell2*)
  also have ‹. . . = *Fst U* $*_V$ *exec-program-with standard-query prog* (*initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*))›
    by (*subst snoc.IH*, *simp-all*)
  also have ‹. . . = *exec-program-with standard-query* (*prog* @ [*program-step*]) (*initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*))›
    by (*simp add*: *ProgramStep exec-program-with-append*)
  finally show *?thesis*
    by −
next
  case (*QueryStep X Y*)
  then have [*register*]: ‹*compatible X Y*›
    using *snoc.prems valid-program-def* **by** *force*
  have ‹*exec-program h* (*prog* @ [*program-step*]) *initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*) = ((*X*;*Y*) (*function-oracle h*) $*_V$ *exec-program h prog initial-state*) $\otimes_s$ *ket* (*Some* ∘ *h*)›
    by (*simp add*: *QueryStep exec-program-append*)
  also have ‹. . . = *Fst* ((*X*;*Y*) (*function-oracle h*)) $*_V$ (*exec-program h prog initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*))›
    by (*simp add*: *Fst-def tensor-op-ell2*)
  also have ‹. . . = (*Fst o X*; (*Fst o Y*; *Snd*)) *standard-query* $*_V$ (*exec-program h prog initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*))›
    by (*simp add*: *standard-query-for-fixed-func-generic assms*)
  also have ‹. . . = (*Fst o X*; (*Fst o Y*; *Snd*)) *standard-query* $*_V$ *exec-program-with standard-query prog* (*initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*))›
    by (*subst snoc.IH*, *simp-all*)
  also have ‹. . . = *exec-program-with standard-query* (*prog* @ [*program-step*]) (*initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*))›
    by (*simp add*: *QueryStep exec-program-with-append*)
  finally show *?thesis*
    by −
qed
qed

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* (⊤ $\otimes_S$ *ccspan* {*ket* (*Some* ∘ *h*)}) *J*›
assumes *IJ*: ‹*J* ⊓ (⊤ $\otimes_S$ *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*)))›
proof −
define *e1 e2* **where** ‹*e1* = *exec-program h program initial-state*› **and** ‹*e2* = *exec-program-with standard-query program* (*initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*))›
define *keth* **where** ‹*keth* = *ket* (*Some o h*)›
have *e2e1*: ‹*e2* = *e1* $\otimes_s$ *keth*›

unfolding *e1-def e2-def keth-def*
using *standard-query-for-fixed-function-generic assms*
by *fastforce*
have ‹*dist-inv Q I e1* = *norm* ((*id-cblinfun* − *Q* (*Proj I*)) $*_V$ *e1*)›
by (*simp add*: *dist-inv-def Proj-ortho-compl register-minus*)
also have ‹. . . = *norm* (*e1* $\otimes_s$ *keth* − (*Q* (*Proj I*) $*_V$ *e1*) $\otimes_s$ *keth*)›
by (*simp add*: *norm-tensor-ell2 keth-def cblinfun.diff-left flip*: *tensor-ell2-diff1*)
also have ‹. . . = *norm* ((*id-cblinfun* − *Fst* (*Q* (*Proj I*)) o_{CL} *Snd* (*proj keth*)) $*_V$ *e1* $\otimes_s$ *keth*)›
by (*simp add*: *Fst-def Snd-def comp-tensor-op tensor-op-ell2 cblinfun.diff-left*)
also have ‹. . . = *dist-inv* (*Fst o Q*; *Snd*) (*I* $\otimes_S$ *ccspan*{*keth*}) (*e1* $\otimes_s$ *keth*)›
by (*simp add*: *dist-inv-def Proj-ortho-compl register-minus tensor-ccsubspace-via-Proj Proj-on-own-range is-Proj-tensor-op register-pair-apply*)
also have ‹. . . = *dist-inv* (*Fst o Q*; *Snd*) (*J* ⊓ (⊤ $\otimes_S$ *ccspan*{*ket* (*Some o h*)})) (*e1* $\otimes_s$ *keth*)›
by (*simp add*: *IJ keth-def*)
also have ‹. . . = *dist-inv* (*Fst o Q*; *Snd*) *J* (*e1* $\otimes_s$ *keth*)›
using *compat* **apply** (*rule dist-inv-intersect-onesided*)
apply *simp*
by (*simp add*: *dist-inv-def Proj-ortho-compl register-minus tensor-ccsubspace-via-Proj Proj-on-own-range is-Proj-tensor-op register-pair-apply cblinfun.diff-left keth-def*)
also have ‹. . . = *dist-inv* (*Fst o Q*; *Snd*) *J e2*›
by (*simp add*: *e2e1*)
finally show ‹*dist-inv Q I e1* = . . .›
by −
qed

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*))›
proof (*insert assms*(*2*), *induction program rule*: *rev-induct*)
case *Nil*
have ‹*sum ket* (*total-functions* :: (*'x* ⇒ *'y option*) *set*) = ($\sum$ *h*∈*UNIV*. *ket* (λ*a*. *Some* (*h a*)))›
apply (*simp add*: *total-functions-def2 sum.reindex fun.inj-map*)
by (*simp add*: *o-def*)
then show *?case*
by (*simp add*: *uniform-superpos-def2 scaleR-scaleC card-fun card-total-functions tensor-ell2-scaleC2 flip*: *scaleC-sum-right tensor-ell2-sum-right*)
next
case (*snoc step prog*)
then have ‹*valid-program-step step*› **and** [*iff*]: ‹*valid-program prog*›
by (*simp-all add*: *valid-program-append*)
have ‹*exec-program-step-with standard-query step* (ψ $\otimes_s$ *ket* (*Some o h*)) = *exec-program-step h step* ψ $\otimes_s$ *ket* (*Some o h*)› **for** *h* ψ
proof (*cases step*)
case (*ProgramStep U*)
then show *?thesis*
by (*simp add*: *Fst-def tensor-op-ell2*)
next
case (*QueryStep X Y*)

with ‹*valid-program-step step*› **have** [*register*]: ‹*compatible X Y*›
by *simp*
have ‹*exec-program-step-with standard-query step* ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))
= (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *standard-query* $*_V$ ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))›
by (*simp add*: *QueryStep*)
also have ‹. . . = ((*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *standard-query* o_{CL} *Snd* (*selfbutter* (*ket* (*Some o h*)))) $*_V$ ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))›
by *simp*
also have ‹. . . = (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) (*standard-query* o_{CL} (*Snd o Snd*) (*selfbutter* (*ket* (*Some o h*)))) $*_V$ ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))›
by (*simp add*: *register-mult*[*symmetric*, **where** *F*=‹(-;-)›] *register-pair-Snd*[*unfolded o-def*, *THEN fun-cong*])
also have ‹. . . = (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) ((*Fst*; *Snd o Fst*) (*function-oracle h*) o_{CL} (*Snd o Snd*) (*selfbutter* (*ket* (*Some o h*)))) $*_V$ ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))›
apply (*rewrite at* ‹(*Fst*;*Snd* ∘ *Fst*) (*function-oracle h*)› *to* ‹*assoc* (*function-oracle h* $\otimes_o$ *id-cblinfun*)› *DEADID.rel-mono-strong*)
apply (*simp add*: *assoc-def register-pair-Fst*[*unfolded o-def*, *THEN fun-cong*] *flip*: *Fst-def*)
apply (*rule arg-cong*[**where** *f*=‹λ*x*. (-;-) *x* $*_V$ -›])
by (*auto intro*!: *equal-ket simp*: *Snd-def tensor-op-ket cinner-ket tensor-ell2-ket assms assoc-ell2-sandwich sandwich-apply function-oracle-apply*)
also have ‹. . . = ((*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) ((*Fst*; *Snd o Fst*) (*function-oracle h*)) o_{CL} *Snd* (*selfbutter* (*ket* (*Some o h*)))) $*_V$ ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))›
by (*simp add*: *register-mult*[*symmetric*, **where** *F*=‹(-;-)›] *register-pair-Snd*[*unfolded o-def*, *THEN fun-cong*])
also have ‹. . . = (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) ((*Fst*; *Snd o Fst*) (*function-oracle h*)) $*_V$ ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))›
by *simp*
also have ‹. . . = *Fst* ((*X*;*Y*) (*function-oracle h*)) $*_V$ ($\psi \otimes_s$ *ket* (*Some* ∘ *h*))›
apply (*rewrite at* ‹(*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) ((*Fst*;*Snd* ∘ *Fst*) -)› *to* ‹((*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *o* (*Fst*;*Snd* ∘ *Fst*)) -› *DEADID.rel-mono-strong*)
apply *simp*
apply (*subst register-comp-pair*[*symmetric*])
apply (*simp*, *simp*)
by (*simp add*: *register-pair-Snd register-pair-Fst register-comp-pair flip*: *comp-assoc*)
also have ‹. . . = ((*X*;*Y*) (*function-oracle h*) $*_V$ ψ) $\otimes_s$ *ket* (*Some* ∘ *h*)›
by (*simp add*: *Fst-def tensor-op-ell2*)
also have ‹. . . = *exec-program-step h step* $\psi \otimes_s$ *ket* (*Some* ∘ *h*)›
by (*simp add*: *QueryStep*)
finally show *?thesis*
by −
qed
then show *?case*
by (*simp add*: *exec-program-with-append exec-program-append snoc.IH o-def complex-vector.linear-sum*[**where** *f*=‹*exec-program-step-with standard-query step*›] *bounded-clinear.clinear bounded-clinear-exec-program-step-with scaleR-scaleC clinear.scaleC*)
qed

lemma *standard-query-is-ro-dist-inv-generic*:
fixes *standard-query* :: ‹('*x*::*finite* × '*y*::{*finite*,*ab-group-add*} × ('*x* ⇀ '*y*)) *ell2* $\Rightarrow_{CL}$ -›
assumes ‹⋀*H x y z*. *H x* = *Some z* ⟹ *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*›)
proof −
have ‹$?rhs^2$ = (*dist-inv* (*Fst* ∘ *Q*) *I* ($\sum$ *h*∈*UNIV*. (*exec-program h program initial-state* $\otimes_s$ *ket* (*Some* ∘ *h*)) $/_R$ *sqrt* (*real CARD*('*x* ⇒ '*y*))))^2›
apply (*subst standard-query-is-ro-generic*)
using *assms* **by** *simp-all*
also have ‹. . . = (*norm* ($\sum$ *i*∈*UNIV*. ((*Q* (*Proj* (− *I*)) $*_V$ *exec-program i program initial-state*) $\otimes_s$ *ket* (*Some* ∘ *i*)) $/_R$ *sqrt* (*real CARD*('*x* ⇒ '*y*))))^2›
by (*simp add*: *dist-inv-def cblinfun.sum-right Fst-def tensor-op-ell2 cblinfun.scaleR-right*)
also have ‹. . . = ($\sum$ *a*∈*UNIV*. (*norm* (((*Q* (*Proj* (− *I*)) $*_V$ *exec-program a program initial-state*) $\otimes_s$ *ket* (*Some* ∘ *a*)) $/_R$ *sqrt* (*real CARD*('*x* ⇒ '*y*))))^2)›
apply (*subst pythagorean-theorem-sum*)
apply (*simp*, *metis fun.inj-map-strong option.inject*)
apply *simp*
by *simp*
also have ‹. . . = ($\sum$ *a*∈*UNIV*. (*dist-inv* (*Fst o Q*) *I* (*exec-program a program initial-state* $\otimes_s$ *ket* (*Some* ∘ *a*)))^2 $/_R$ *real CARD*('*x* ⇒ '*y*))›
by (*auto intro*!: *sum.cong simp*: *dist-inv-def Fst-def tensor-op-ell2 power-mult-distrib real-inv-sqrt-pow2*)
also have ‹. . . = ($\sum$ *x*∈*UNIV*. (*dist-inv Q I* (*exec-program x program initial-state*))^2) $/_R$ *real CARD*('*x* ⇒ '*y*)›
by (*metis* (*no-types*, *lifting*) *dist-inv-Fst-tensor norm-ket scaleR-right.sum sum.cong*)
also have ‹. . . = $?lhs^2$›
by (*simp add*: *dist-inv-avg-def real-sqrt-pow2 sum-nonneg divide-inverse flip*: *sum-distrib-left*)
finally show *?thesis*
by *simp*
qed

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*›)
using *standard-query-ket-full-Some assms* **by** (*rule standard-query-is-ro-dist-inv-generic*)

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*›)
using *standard-query'-ket-full-Some assms* **by** (*rule standard-query-is-ro-dist-inv-generic*)

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*))›
proof (*insert* ‹*valid-program program*›, *induction program rule*: *rev-induct*)
case *Nil*

then show *?case*
by (*simp add*: *compress-empty*)
next
case (*snoc program-step prog*)
then have [*simp*]: ‹*valid-program prog*›
by (*simp add*: *valid-program-def*)
show *?case*
proof (*cases program-step*)
case (*ProgramStep U*)
have ‹*exec-program-with standard-query* (*prog* @ [*program-step*]) (*initial-state* $\otimes_s$ *superpos-total*)
= *Fst U* $*_V$ *Snd compress* $*_V$ *exec-program-with query prog* (*initial-state* $\otimes_s$ *ket Map.empty*)›
by (*simp add*: *ProgramStep snoc.IH exec-program-with-append*)
also have ‹... = *Snd compress* $*_V$ *Fst U* $*_V$ *exec-program-with query prog* (*initial-state* $\otimes_s$ *ket Map.empty*)›
by (*simp flip*: *cblinfun-apply-cblinfun-compose swap-registers*)
also have ‹... = *Snd compress* $*_V$ *exec-program-with query* (*prog* @ [*program-step*]) (*initial-state* $\otimes_s$ *ket Map.empty*)›
by (*simp add*: *ProgramStep exec-program-with-append*)
finally show *?thesis*
by −
next
case (*QueryStep X Y*)
with *snoc.prems* **have** [*register*]: ‹*compatible X Y*›
by (*simp add*: *valid-program-def*)
have *aux*: ‹(*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) (*reg-3-3 compress*) = *Snd compress*›
by (*simp add*: *reg-3-3-def register-pair-Snd*[*unfolded o-def*, *THEN fun-cong*])
have ‹*exec-program-with standard-query* (*prog* @ [*program-step*]) (*initial-state* $\otimes_s$ *superpos-total*)
= (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *standard-query* $*_V$ *Snd compress* $*_V$ *exec-program-with query prog* (*initial-state* $\otimes_s$ *ket Map.empty*)›
by (*simp add*: *QueryStep snoc.IH exec-program-with-append*)
also have ‹... = (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) (*standard-query* o_{CL} *reg-3-3 compress*) $*_V$ *exec-program-with query prog* (*initial-state* $\otimes_s$ *ket Map.empty*)›
by (*simp-all add*: *aux flip*: *register-mult*)
also have ‹... = (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) (*reg-3-3 compress* o_{CL} *query*) $*_V$ *exec-program-with query prog* (*initial-state* $\otimes_s$ *ket Map.empty*)›
by (*simp add*: *assms*)
also have ‹... = *Snd compress* $*_V$ (*Fst* ∘ *X*;(*Fst* ∘ *Y*;*Snd*)) *query* $*_V$ (*exec-program-with query*) *prog* (*initial-state* $\otimes_s$ *ket Map.empty*)›
by (*simp-all add*: *aux flip*: *register-mult*)
also have ‹... = *Snd compress* $*_V$ (*exec-program-with query*) (*prog* @ [*program-step*]) (*initial-state* $\otimes_s$ *ket* (λ*x*. *None*))›
by (*simp add*: *QueryStep Cons exec-program-with-append*)
finally show *?thesis*
by −
qed
qed

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*)))›

proof −
have ‹*dist-inv Fst I* (*exec-program-with standard-query program* (*initial-state* $\otimes_s$ *superpos-total*)) = *norm* (*Fst* (*Proj* (− *I*)) $*_V$ *Snd compress* $*_V$ *exec-program-with query program* (*initial-state* $\otimes_s$ *ket* (λ*x*. *None*)))›
by (*simp add*: *compress-query-is-standard-query-generic assms dist-inv-def Proj-on-own-range register-projector*)
also have ‹. . . = *norm* (*Snd compress* $*_V$ *Fst* (*Proj* (− *I*)) $*_V$ *exec-program-with query program* (*initial-state* $\otimes_s$ *ket* (λ*x*. *None*)))›
by (*simp flip*: *cblinfun-apply-cblinfun-compose swap-registers*)
also have ‹. . . = *norm* (*Fst* (*Proj* (− *I*)) $*_V$ *exec-program-with query program* (*initial-state* $\otimes_s$ *ket* (λ*x*. *None*)))›
by (*simp add*: *isometry-preserves-norm register-isometry*[**where** *F*=*Snd*])
also have ‹. . . = *dist-inv Fst I* (*exec-program-with query program* (*initial-state* $\otimes_s$ *ket* (λ*x*. *None*)))›
by (*simp add*: *dist-inv-def Proj-on-own-range register-projector*)
finally show *?thesis*
by −
qed

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* (λ*x*. *None*)))›
using *query-is-standard-query-generic standard-query-compress assms* **by** *blast*

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* (λ*x*. *None*)))›
using *query-is-standard-query-generic standard-query′-compress assms* **by** *blast*

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

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

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* ⇒ ′*mem ell2* $\Rightarrow_{CL}$ ′*mem ell2*›
assumes ‹*valid-program program*›
assumes [*register*]: ‹*register Q*›
shows ‹*dist-inv* (*Fst* ∘ *Q*) *I* (*exec-program-with query program* (ψ $\otimes_s$ *ket* (λ-. *None*)))
= *dist-inv-avg Q* (λ-. *I*) (λ*h*. *exec-program h program* ψ)›

proof −
  have ‹*dist-inv* (*Fst* ∘ *Q*) *I* (*exec-program-with query program* ($\psi \otimes_s$ *ket* (λ-. *None*)))
      = *dist-inv Fst* (*lift-invariant Q I*) (*exec-program-with query program* ($\psi \otimes_s$ *ket* (λ-. *None*)))›
    by (*metis* (*no-types*, *lifting*) *Proj-lift-invariant assms*(*2*) *dist-inv-def lift-invariant-compl o-apply*)
  also have ‹. . . = *dist-inv-avg id* (λ*h*. *lift-invariant Q I*) (λ*h*. *exec-program h program* ψ)›
    by (*simp add*: *query-is-random-oracle assms*)
  also have ‹. . . = *dist-inv-avg Q* (λ-. *I*) (λ*h*. *exec-program h program* ψ)›
    by (*simp add*: *dist-inv-avg-register-rewrite*)
  finally show *?thesis*
    by −
qed

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* ⇒ *′mem ell2* $\Rightarrow_{CL}$ *′mem ell2*›
  assumes ‹*valid-program program*›
  assumes [*register*]: ‹*register Q*›
  shows ‹*dist-inv* (*Fst* ∘ *Q*) *I* (*exec-program-with query′ program* ($\psi \otimes_s$ *ket* (λ-. *None*)))
      = *dist-inv-avg Q* (λ-. *I*) (λ*h*. *exec-program h program* ψ)›
proof −
  have ‹*dist-inv* (*Fst* ∘ *Q*) *I* (*exec-program-with query′ program* ($\psi \otimes_s$ *ket* (λ-. *None*)))
      = *dist-inv Fst* (*lift-invariant Q I*) (*exec-program-with query′ program* ($\psi \otimes_s$ *ket* (λ-. *None*)))›
    by (*metis* (*no-types*, *lifting*) *Proj-lift-invariant assms*(*2*) *dist-inv-def lift-invariant-compl o-apply*)
  also have ‹. . . = *dist-inv-avg id* (λ*h*. *lift-invariant Q I*) (λ*h*. *exec-program h program* ψ)›
    by (*simp add*: *query′-is-random-oracle assms*)
  also have ‹. . . = *dist-inv-avg Q* (λ-. *I*) (λ*h*. *exec-program h program* ψ)›
    by (*simp add*: *dist-inv-avg-register-rewrite*)
  finally show *?thesis*
    by −
qed

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′*›
  by (*auto intro*!: *simp*: *no-zero-def no-zero′-def*)

lemma *ket-invariant-no-zero-no-zero′*: ‹*ket-invariant no-zero* = $\top \otimes_S \top \otimes_S$ *ket-invariant no-zero′*›
  by (*auto simp*: *ket-invariant-tensor no-zero-no-zero′ simp flip*: *ket-invariant-UNIV*)

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*)›
proof −

define K **where** ‹$K\ x = ket\text{-}invariant\ \{(x,y::'y,D::'x \rightharpoonup 'y) \mid y\ D.\ Some\ 0 \notin D\ `\ (-\{x\})\}$› **for** x
define Kd **where** ‹$Kd\ x\ D0 = ket\text{-}invariant\ \{(x,y::'y,D::'x \rightharpoonup 'y) \mid y\ D.\ (\forall x' \neq x.\ D\ x' = D0\ x')\}$› **for** $x\ D0$
have *aux*: ‹$Some\ 0 \notin D\ `\ (-\ \{x\}) \Longrightarrow$
$\exists xa.\ xa\ x = None \land Some\ 0 \notin range\ xa \land (\forall x'.\ x' \neq x \longrightarrow D\ x' = xa\ x')$› **for** D::‹$'x \rightharpoonup 'y$› **and** x
apply (*rule exI*[*of* - ‹$D(x:=None)$›])
by *force*
have K: ‹$K\ x = (SUP\ D0 \in \{D0.\ D0\ x = None \land Some\ 0 \notin range\ D0\}.\ Kd\ x\ D0)$› **for** x
using *aux*[*of* - x] **by** (*auto intro*!: *simp*: *K-def Kd-def simp flip*: *ket-invariant-SUP*)
define Kdx **where** ‹$Kdx\ x\ D0\ x' = ket\text{-}invariant\ \{(x::'x,y::'y,D::'x \rightharpoonup 'y) \mid y\ D.\ D\ x' = D0\ x'\}$› **for** $x\ D0\ x'$
have Kd: ‹$Kd\ x\ D0 = (INF\ x' \in -\{x\}.\ Kdx\ x\ D0\ x')$› **for** $x\ D0$
unfolding *Kd-def Kdx-def*
apply (*subst ket-invariant-INF*[*symmetric*])
apply (*rule arg-cong*[**where** f=*ket-invariant*])
by *auto*
have Kdx: ‹$Kdx\ x\ D0\ x' = lift\text{-}invariant\ reg\text{-}1\text{-}3\ (ket\text{-}invariant\ \{x\}) \sqcap lift\text{-}invariant\ (reg\text{-}3\text{-}3\ o\ function\text{-}at\ x')\ (ket\text{-}invariant\ \{D0\ x'\})$› **for** $x\ D0\ x'$
unfolding *Kdx-def reg-3-3-def reg-1-3-def*
apply (*simp add*: *lift-invariant-comp*)
apply (*subst lift-invariant-function-at-ket-inv*)
apply (*subst lift-Snd-ket-inv*)
apply (*subst lift-Snd-ket-inv*)
apply (*subst lift-Fst-ket-inv*)
apply (*subst ket-invariant-inter*)
apply (*rule arg-cong*[**where** f=*ket-invariant*])
by *auto*

show *?thesis*
proof (*rule inv-split-reg-query*[**where** X=‹*reg-1-3*› **and** Y=‹*reg-2-3*› **and** H=‹*reg-3-3*› **and** K=K **and** $?I1.0$=‹λ-. *ket-invariant* ($UNIV \times -\{Some\ 0\}$)› **and** $?J1.0$=‹λ-. *ket-invariant* ($UNIV \times -\{Some\ 0\}$)›])
show ‹*query* = (*reg-1-3*;(*reg-2-3*;*reg-3-3*)) *query*›
by (*simp add*: *pair-Fst-Snd reg-1-3-def reg-2-3-def reg-3-3-def*)
show ‹*compatible reg-1-3 reg-2-3*› ‹*compatible reg-1-3 reg-3-3*› ‹*compatible reg-2-3 reg-3-3*›
by *simp-all*
show ‹*compatible-register-invariant reg-2-3* ($K\ x$)› **for** x
unfolding *K Kd Kdx*
apply (*rule compatible-register-invariant-SUP*, *simp*)
apply (*rule compatible-register-invariant-INF*, *simp*)
apply (*rule compatible-register-invariant-inter*, *simp*)
apply (*rule compatible-register-invariant-compatible-register*)
apply *simp*
apply (*rule compatible-register-invariant-compatible-register*)
by *simp*
show ‹*compatible-register-invariant* (*reg-3-3 o function-at* x) ($K\ x$)› **for** x
unfolding *K Kd Kdx*
apply (*rule compatible-register-invariant-SUP*, *simp*)
apply (*rule compatible-register-invariant-INF*, *simp*)
apply (*rule compatible-register-invariant-inter*, *simp*)
apply (*rule compatible-register-invariant-compatible-register*)
apply *simp*
apply (*rule compatible-register-invariant-compatible-register*)
by *simp*

show ‹*ket-invariant no-zero*
$\leq$ (*SUP x. K x* ⊓
lift-invariant (*reg-2-3*;*reg-3-3* ∘ *function-at x*) (*ket-invariant* (*UNIV* × − {*Some 0*})))›
apply (*simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*]
lift-inv-prod lift-invariant-comp lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv)
unfolding *no-zero-def*
by (*auto simp add*: *ranI*)
have *aux*: ‹⋀*D*::′*x*⇀′*y*. *Some 0* ∉ *D* ‘ (− {*x*}) ⟹ *D x* ≠ *Some 0* ⟹ *0* ∈ *ran D* ⟹ *False*› **for** *x*
by (*smt* (*verit*, *del-insts*) *ComplI image-iff mem-Collect-eq ran-def singletonD*)
show ‹*K x* ⊓ *lift-invariant* (*reg-2-3*;*reg-3-3* ∘ *function-at x*) (*ket-invariant* (*UNIV* × − {*Some 0*}))
$\leq$ *ket-invariant no-zero*› **for** *x*
by (*auto intro*: *aux simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*]
lift-inv-prod lift-invariant-comp lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv no-zero-def)
show ‹*orthogonal-spaces* (*K x*) (*K x*′)› **if** ‹*x* ≠ *x*′› **for** *x x*′
using *that* **by** (*auto simp add*: *K-def*)
show ‹*preserves-ket query1* (*UNIV* × − {*Some 0*}) (*UNIV* × − {*Some 0*}) (*6* / *sqrt N*)›
apply (*subst asm-rl*[*of* ‹*6* / *sqrt N* = *6* ∗ *sqrt* (*1*::*nat*) / *sqrt N*›], *simp*)
apply (*rule preserve-query1-simplified*)
by (*auto simp add*: *card-le-Suc0-iff-eq*)

show ‹*K x* ≤ *lift-invariant reg-1-3* (*ket-invariant* {*x*})› **for** *x*
by (*auto simp add*: *K-def reg-1-3-def lift-Fst-ket-inv*)
show ‹*6* / *sqrt N* ≥ *0*›
by *simp*
qed *simp*
qed

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*)›
proof −
define *K* **where** ‹*K x* = *ket-invariant* {(*x*,*y*::′*y*,*D*::′*x*⇀′*y*) | *y D*. *Some 0* ∉ *D* ‘ (−{*x*})}› **for** *x*
define *Kd* **where** ‹*Kd x D0* = *ket-invariant* {(*x*,*y*::′*y*,*D*::′*x*⇀′*y*) | *y D*. (∀ *x*′≠*x*. *D x*′ = *D0 x*′)}› **for** *x D0*
have *aux*: ‹*Some 0* ∉ *D* ‘ (− {*x*}) ⟹
∃ *xa*. *xa x* = *None* ∧ *Some 0* ∉ *range xa* ∧ (∀ *x*′. *x*′ ≠ *x* ⟶ *D x*′ = *xa x*′)› **for** *D*::‹′*x*⇀′*y*› **and** *x*
apply (*rule exI*[*of* - ‹*D*(*x*:=*None*)›])
by *force*
have *K*: ‹*K x* = (*SUP D0*∈{*D0*. *D0 x* = *None* ∧ *Some 0* ∉ *range D0*}. *Kd x D0*)› **for** *x*
using *aux*[*of* - *x*] **by** (*auto intro*!: *simp*: *K-def Kd-def simp flip*: *ket-invariant-SUP*)
define *Kdx* **where** ‹*Kdx x D0 x*′ = *ket-invariant* {(*x*::′*x*,*y*::′*y*,*D*::′*x*⇀′*y*) | *y D*. *D x*′ = *D0 x*′}› **for** *x D0 x*′
have *Kd*: ‹*Kd x D0* = (*INF x*′∈−{*x*}. *Kdx x D0 x*′)› **for** *x D0*
unfolding *Kd-def Kdx-def*
apply (*subst ket-invariant-INF*[*symmetric*])
apply (*rule arg-cong*[**where** *f*=*ket-invariant*])
by *auto*
have *Kdx*: ‹*Kdx x D0 x*′ = *lift-invariant reg-1-3* (*ket-invariant* {*x*}) ⊓ *lift-invariant* (*reg-3-3 o function-at x*′) (*ket-invariant* {*D0 x*′})› **for** *x D0 x*′
unfolding *Kdx-def reg-3-3-def reg-1-3-def*
apply (*simp add*: *lift-invariant-comp*)
apply (*subst lift-invariant-function-at-ket-inv*)
apply (*subst lift-Snd-ket-inv*)

apply (*subst lift-Snd-ket-inv*)
apply (*subst lift-Fst-ket-inv*)
apply (*subst ket-invariant-inter*)
apply (*rule arg-cong*[**where** *f*=*ket-invariant*])
by *auto*

show *?thesis*
proof (*rule inv-split-reg-query'*[**where** *X*=‹*reg-1-3*› **and** *Y*=‹*reg-2-3*› **and** *H*=‹*reg-3-3*› **and** *K*=*K* **and** *?I1.0*=‹λ-. *ket-invariant* (*UNIV* × −{*Some 0*})› **and** *?J1.0*=‹λ-. *ket-invariant* (*UNIV* × −{*Some 0*})›])
show ‹*query'* = (*reg-1-3*;(*reg-2-3*;*reg-3-3*)) *query'*›
by (*simp add*: *pair-Fst-Snd reg-1-3-def reg-2-3-def reg-3-3-def*)
show ‹*compatible reg-1-3 reg-2-3*› ‹*compatible reg-1-3 reg-3-3*› ‹*compatible reg-2-3 reg-3-3*›
by *simp-all*
show ‹*compatible-register-invariant reg-2-3* (*K x*)› **for** *x*
unfolding *K Kd Kdx*
apply (*rule compatible-register-invariant-SUP*, *simp*)
apply (*rule compatible-register-invariant-INF*, *simp*)
apply (*rule compatible-register-invariant-inter*, *simp*)
apply (*rule compatible-register-invariant-compatible-register*)
apply *simp*
apply (*rule compatible-register-invariant-compatible-register*)
by *simp*
show ‹*compatible-register-invariant* (*reg-3-3 o function-at x*) (*K x*)› **for** *x*
unfolding *K Kd Kdx*
apply (*rule compatible-register-invariant-SUP*, *simp*)
apply (*rule compatible-register-invariant-INF*, *simp*)
apply (*rule compatible-register-invariant-inter*, *simp*)
apply (*rule compatible-register-invariant-compatible-register*)
apply *simp*
apply (*rule compatible-register-invariant-compatible-register*)
by *simp*
show ‹*ket-invariant no-zero*
≤ (*SUP x*. *K x* ⊓
lift-invariant (*reg-2-3*;*reg-3-3* ∘ *function-at x*) (*ket-invariant* (*UNIV* × − {*Some 0*})))›
apply (*simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*] *lift-inv-prod lift-invariant-comp lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv*)
unfolding *no-zero-def*
by (*auto simp add*: *ranI*)
have *aux*: ‹*Some 0* ∉ *D* ‘ (− {*x*}) ⟹ *D x* ≠ *Some 0* ⟹ *0* ∉ *ran D*› **for** *D x*
by (*smt* (*verit*, *del-insts*) *ComplI image-iff mem-Collect-eq ran-def singletonD*)
show ‹*K x* ⊓ *lift-invariant* (*reg-2-3*;*reg-3-3* ∘ *function-at x*) (*ket-invariant* (*UNIV* × − {*Some 0*}))
≤ *ket-invariant no-zero*› **for** *x*
using *aux*[*of - x*]
by (*auto simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*] *lift-inv-prod lift-invariant-comp lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv no-zero-def*)
show ‹*orthogonal-spaces* (*K x*) (*K x'*)› **if** ‹*x* ≠ *x'*› **for** *x x'*
using *that* **by** (*auto simp add*: *K-def*)
show ‹*preserves-ket query1'* (*UNIV* × − {*Some 0*}) (*UNIV* × − {*Some 0*}) (*5* / *sqrt N*)›
apply (*subst asm-rl*[*of* ‹*5* / *sqrt N* = *5* ∗ *sqrt* (*1*::*nat*) / *sqrt N*›], *simp*)
apply (*rule preserve-query1'-simplified*)
by (*auto simp add*: *card-le-Suc0-iff-eq*)
show ‹*K x* ≤ *lift-invariant reg-1-3* (*ket-invariant* {*x*})› **for** *x*
by (*auto simp add*: *K-def reg-1-3-def lift-Fst-ket-inv*)

show ‹*5 / sqrt N ≥ 0*›
by *simp*
qed *simp*
qed

lemma *preserves-no-zero-num*: ‹*preserves-ket query* (*no-zero* ∩ *num-queries q*) (*no-zero* ∩ *num-queries* (*q+1*)) (*6 / sqrt N*)›
apply (*subst add-0-right*[*of* ‹*6/sqrt N*›, *symmetric*])
apply (*rule preserves-intersect-ket*)
apply (*simp add*: *preserves-mono*[*OF preserves-no-zero*])
apply (*rule preserves-mono*[*OF preserves-num*])
by *auto*

lemma *preserves-no-zero-num'*: ‹*preserves-ket query'* (*no-zero* ∩ *num-queries q*) (*no-zero* ∩ *num-queries* (*q+1*)) (*5 / sqrt N*)›
apply (*subst add-0-right*[*of* ‹*5/sqrt N*›, *symmetric*])
apply (*rule preserves-intersect-ket*)
apply (*simp add*: *preserves-mono*[*OF preserves-no-zero'*])
apply (*rule preserves-mono*[*OF preserves-num'*])
by *auto*

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*∈*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*›
proof −
note [[*simproc del*: *Laws-Quantum.compatibility-warn*]]

In this game based proof, we consider three different quantum memory models:

- The one from the statement of the lemma, where the overall quantum state lives in *'mem*, and the adversary output register is described by *adv-output*, and the initial state in *initial-state*. The program *program* assumes this memory model.
- The "extra output" (short XO) memory model, where there is an extra auxiliary register *aux* of type *'y*. The type of the memory is then *'mem × 'y*. (I.e., the extra register is in addition to the content of *'mem*.)
- The "compressed oracle" (short CO) memory model, where additionally to XO, we have an oracle register that can holds the content of the compressed oracle (or the standard oracle).

Since the register *adv-output* is defined w.r.t. a specific memory, we define convenience definitions for the same register as it would be accessed in the other memories:

define *adv-output-in-xo* :: ‹*'x update* ⇒ (*'mem*×*'y*) *update*› **where** ‹*adv-output-in-xo* = *Fst o adv-output*›
define *adv-output-in-co* :: ‹*'x update* ⇒ ((*'mem*×*'y*) × (*'x*⇀*'y*)) *update*› **where** ‹*adv-output-in-co* = *Fst o adv-output-in-xo*›

Analogously, we defined the *aux*-register and the oracle register in the applicable memories:

define *aux-in-xo* :: ‹*'y update* ⇒ (*'mem*×*'y*) *update*› **where** ‹*aux-in-xo* = *Snd*›
define *aux-in-co* :: ‹*'y update* ⇒ ((*'mem*×*'y*) × (*'x*⇀*'y*)) *update*› **where** ‹*aux-in-co* = *Fst o aux-in-xo*›
define *oracle-in-co* :: ‹(*'x*⇀*'y*) *update* ⇒ ((*'mem*×*'y*) × (*'x*⇀*'y*)) *update*› **where** ‹*oracle-in-co* = *Snd*›
define *aao-in-co* **where** ‹*aao-in-co* = (*adv-output-in-co*; (*aux-in-co*; *oracle-in-co*))›
— Abbreviation since we use this combination often.

have [*register*]: ‹*compatible aux-in-co oracle-in-co*›
by (*simp add*: *adv-output-in-co-def aux-in-co-def oracle-in-co-def adv-output-in-xo-def aux-in-xo-def*)
have [*register*]: ‹*compatible adv-output-in-xo aux-in-xo*›
by (*simp add*: *adv-output-in-xo-def aux-in-xo-def*)
have [*register*]: ‹*compatible adv-output-in-co aux-in-co*›
by (*simp add*: *adv-output-in-co-def aux-in-co-def*)
have [*register*]: ‹*compatible adv-output-in-co oracle-in-co*›
by (*simp add*: *adv-output-in-co-def oracle-in-co-def*)
have [*register*]: ‹*compatible aux-in-xo Fst*›
by (*simp add*: *aux-in-xo-def*)
have [*register*]: ‹*compatible aux-in-co* (*Fst o Fst*)›
by (*simp add*: *aux-in-co-def*)
have [*register*]: ‹*compatible aux-in-co Snd*›
by (*simp add*: *aux-in-co-def*)
have [*register*]: ‹*register aao-in-co*›
by (*simp add*: *aao-in-co-def*)

The initial states in XO/CO are like the original initial state, but with *ket 0* in *aux* and *ket* ($\lambda x.$ *None*) (the fully undefined function) in the oracle register.

define *initial-state-in-xo* **where** ‹*initial-state-in-xo* = *initial-state* $\otimes_s$ *ket* (*0* :: *'y*)›
define *initial-state-in-co* :: ‹((*'mem*×*'y*) × (*'x*⇀*'y*)) *ell2*› **where** ‹*initial-state-in-co* = *initial-state-in-xo* $\otimes_s$ *ket Map.empty*›

We define an extended program *ext-program* that executes *program*, followed by one additional query to the oracle. Input register is the adversary output register. Output register is the additional register *aux*. Hence *ext-program* is only meaningful in the models XO and CO. (Our definition is for XO.)

define *ext-program* **where** ‹*ext-program* = *lift-program Fst program* @ [*QueryStep adv-output-in-xo aux-in-xo*]›
have [*iff*]: ‹*valid-program ext-program*›
by (*auto intro*!: *valid-program-lift simp add*: *valid-program-append adv-output-in-xo-def aux-in-xo-def ext-program-def*)

We define the final states of the programs *program* and *ext-program*, in the original model, and in XO, and CO.

define *final* :: ‹(*'x* ⇒ *'y*) ⇒ *'mem ell2*› **where** ‹*final h* = *exec-program h program initial-state*› **for** *h*
define *xo-ext-final* :: ‹(*'x* ⇒ *'y*) ⇒ (*'mem*×*'y*) *ell2*› **where** ‹*xo-ext-final h* = *exec-program h ext-program initial-state-in-xo*› **for** *h*
define *xo-final* :: ‹(*'x* ⇒ *'y*) ⇒ (*'mem*×*'y*) *ell2*› **where** ‹*xo-final h* = *exec-program h* (*lift-program Fst program*) *initial-state-in-xo*› **for** *h*
define *co-ext-final* :: ‹((*'mem*×*'y*) × (*'x*⇀*'y*)) *ell2*› **where** ‹*co-ext-final* = *exec-program-with query' ext-program initial-state-in-co*›

define *co-final* :: ‹((′*mem*×′*y*) × (′*x*⇀′*y*)) *ell2*› **where** ‹*co-final* = *exec-program-with query′* (*lift-program Fst program*) *initial-state-in-co*›

have [*simp*]: ‹*norm initial-state-in-xo* = *1*›
  by (*simp add*: *initial-state-in-xo-def norm-tensor-ell2 assms*)
have *norm-initial-state-in-co*[*simp*]: ‹*norm initial-state-in-co* = *1*›
  by (*simp add*: *initial-state-in-co-def norm-tensor-ell2*)

have *norm-co-final*[*simp*]: ‹*norm co-final* ≤ *1*›
  unfolding *co-final-def*
  using *norm-exec-program-with valid-program-lift* ‹*valid-program program*›
    norm-query′ register-Fst norm-initial-state-in-co
  by *smt*

We derive the relationships between the various final states:

have *co-ext-final-prefinal*: ‹*co-ext-final* = *aao-in-co query′* $*_V$ *co-final*›
  by (*simp add*: *co-ext-final-def ext-program-def exec-program-with-append aao-in-co-def*
    flip: *initial-state-in-co-def co-final-def adv-output-in-co-def aux-in-co-def oracle-in-co-def*)

have *xo-final-final*: ‹*xo-final h* = *final h* $\otimes_s$ *ket 0*› **for** *h*
  by (*simp add*: *xo-final-def final-def initial-state-in-xo-def exec-lift-program-Fst*)

have *xo-ext-final-xo-final*: ‹*xo-ext-final h* = (*adv-output-in-xo*;*aux-in-xo*) (*function-oracle h*) $*_V$ *xo-final h*› **for** *h*
  by (*simp add*: *xo-ext-final-def xo-final-def ext-program-def exec-program-def*)

After executing *program* (in XO), the *aux*-register is in state *ket 0*:

have *xo-final-has-y0*: ‹*dist-inv-avg* (*adv-output-in-xo*;*aux-in-xo*) (λ-. *ket-invariant* {(*x*,*y*). *y* = *0*}) *xo-final* = *0*›
proof −
  have ‹*dist-inv-avg aux-in-xo* (λ-::′*x*⇒′*y*. *ket-invariant* {*0*}) *xo-final*
    ≤ *dist-inv-avg aux-in-xo* (λ-::′*x*⇒′*y*. *ket-invariant* {*0*}) (λ*h*. *initial-state-in-xo*)›
    unfolding *xo-final-def*
    apply (*subst dist-inv-avg-exec-compatible*)
    using *dist-inv-avg-exec-compatible*
    by *auto*
  also have ‹. . . = *0*›
    by (*auto intro*!: *tensor-ell2-in-tensor-ccsubspace ket-in-ket-invariantI*
      simp add: *initial-state-in-xo-def dist-inv-0-iff distance-from-inv-avg0I aux-in-xo-def lift-Snd-inv*)
  finally have ‹*dist-inv-avg aux-in-xo* (λ-. *ket-invariant* {*0*}) *xo-final* = *0*›
    by (*smt* (*verit*, *ccfv-SIG*) *dist-inv-avg-pos*)
  then show *?thesis*
    apply (*rewrite at* ‹{(*x*, *y*). *y* = *0*}› *to* ‹*UNIV* × {*0*}› *DEADID.rel-mono-strong*, *blast*)
    apply (*subst dist-inv-avg-register-rewrite*)
    by (*simp-all add*: *lift-inv-prod*)
qed

Same as *xo-final-has-y0*, but in CO:

have *co-final-has-y0*: ‹*dist-inv aao-in-co* (*ket-invariant* {(*x*,*y*,*D*). *y* = *0*}) *co-final* = *0*›
proof −
  have ‹*dist-inv aux-in-co* (*ket-invariant* {*0*}) *co-final*
    ≤ *dist-inv aux-in-co* (*ket-invariant* {*0*}) *initial-state-in-co*›
    unfolding *co-final-def*
    apply (*rule dist-inv-exec′-compatible*)
    by *simp-all*

also have ‹. . . = *0*›
  by (*auto intro*!: *tensor-ell2-in-tensor-ccsubspace ket-in-ket-invariantI*
    simp add: *initial-state-in-co-def initial-state-in-xo-def dist-inv-0-iff*
    aux-in-co-def aux-in-xo-def lift-Fst-inv lift-Snd-inv lift-invariant-comp)
finally have ‹*dist-inv aux-in-co* (*ket-invariant* {*0*}) *co-final* = *0*›
  by (*smt* (*verit*, *best*) *dist-inv-pos*)
then show *?thesis*
  apply (*rewrite at* ‹{(*x*, *y*, *D*). *y* = *0*}› *to* ‹*UNIV* × {*0*} × *UNIV*› *DEADID.rel-mono-strong*, *blast*)
  apply (*subst dist-inv-register-rewrite*)
  by (*simp-all add*: *lift-inv-prod aao-in-co-def*)
qed

define *q* **where** ‹*q* = *query-count program*›

The following term occurs a lot (it's how much the *no-zero* invariant is preserved after running *ext-program*). So we abbreviate it as *d*.

define *d* :: *real* **where** ‹*d* = (*5* ∗ *q* + *11*) / *sqrt N*›

have [*iff*]: ‹*d* ≥ *0*›
  by (*simp add*: *d-def*)

have ‹*dist-inv oracle-in-co* (*ket-invariant no-zero′*) *co-ext-final* ≤ *5* ∗ (*q*+*1*) / *sqrt N*›
  — In CO-execution, before the adversary's final query, the oracle register has no 0 in its range
proof (*unfold co-ext-final-def*, *rule dist-inv-induct*[**where** *g*=‹λ-. *5* / *sqrt N*› **and** *J*=‹λ-. *ket-invariant no-zero′*›])
  show ‹*compatible oracle-in-co Fst*›
    using *oracle-in-co-def* **by** *simp*
  show ‹*initial-state-in-co* ∈ *space-as-set* (*lift-invariant oracle-in-co* (*ket-invariant no-zero′*))›
    by (*auto intro*!: *tensor-ell2-in-tensor-ccsubspace*
      simp add: *initial-state-in-co-def oracle-in-co-def lift-Snd-ket-inv*
      initial-state-in-xo-def tensor-ell2-ket ket-in-ket-invariantI no-zero′-def
      simp flip: *ket-invariant-tensor*)
  show ‹*ket-invariant no-zero′* ≤ *ket-invariant no-zero′*›
    by *simp*
  show ‹*valid-program ext-program*›
    by (*simp add*: *valid-program-lift*)
  show ‹*preserves* ((*Fst* ∘ *X-in-xo*;(*Fst* ∘ *Y-in-xo*;*Snd*)) *query′*) (*lift-invariant oracle-in-co* (*ket-invariant no-zero′*))
    (*lift-invariant oracle-in-co* (*ket-invariant no-zero′*)) (*5* / *sqrt N*)› **if** [*register*]: ‹*compatible X-in-xo Y-in-xo*› **for** *X-in-xo Y-in-xo*
  proof −
    from *preserves-no-zero′*
    have ‹*preserves* ((*Fst* ∘ *X-in-xo*;(*Fst* ∘ *Y-in-xo*;*Snd*)) *query′*)
        (*lift-invariant* (*Fst* ∘ *X-in-xo*;(*Fst* ∘ *Y-in-xo*;*Snd*)) (*ket-invariant no-zero*))
        (*lift-invariant* (*Fst* ∘ *X-in-xo*;(*Fst* ∘ *Y-in-xo*;*Snd*)) (*ket-invariant no-zero*))
        (*5* / *sqrt* (*real N*))›
      unfolding *N-def*
      apply (*rule preserves-lift-invariant*[*THEN iffD2*, *rotated*])
      by *simp*
    moreover have ‹*lift-invariant* (*Fst* ∘ *X-in-xo*;(*Fst* ∘ *Y-in-xo*;*Snd*)) (*ket-invariant no-zero*)
          = *lift-invariant oracle-in-co* (*ket-invariant no-zero′*)›
      by (*simp add*: *oracle-in-co-def no-zero-no-zero′ lift-inv-prod*)
    finally show *?thesis*
      by −

```
    qed
    show ‹norm query' ≤ 1›
      by simp
    show ‹norm initial-state-in-co ≤ 1›
      by simp
    show ‹(∑ i<query-count ext-program. 5 / sqrt N) ≤ real (5 ∗ (q+1)) / sqrt N›
      apply (simp add: query-count-lift-program ext-program-def flip: q-def)
      by argo
  qed

  then have dist-zero: ‹dist-inv aao-in-co (ket-invariant no-zero) co-ext-final ≤ 5 ∗ (q+1) / sqrt N›
      — Same thing, but expressed w.r.t. different register
    apply (rule le-back-subst)
    apply (rule dist-inv-register-rewrite)
    by (auto intro!: simp: aao-in-co-def no-zero-no-zero' lift-inv-prod)

  have dist-Dxy: ‹dist-inv aao-in-co (ket-invariant {(x,y,D). D x = Some y}) co-ext-final ≤ 6 / sqrt N›
(is ‹?lhs ≤ -›)
    unfolding co-ext-final-prefinal
    apply (rule dist-inv-leq-if-preserves[THEN order-trans])
       apply (subst preserves-lift-invariant)
        apply (auto intro!: preserves-ket-query'-output-simple simp: register-norm)[4]
    using norm-co-final
    by (simp add: N-def co-final-has-y0 field-class.field-divide-inverse)

  have ‹dist-inv aao-in-co
          (ket-invariant {(x, y, D::'x ⇀ 'y). 0 ∉ ran D ∧ D x = Some y}) co-ext-final ≤ d› (is ‹?lhs ≤
d›)
     — In CO-execution, after the adversary's final query, the oracle register has no 0 in its range, and
the aux register contains the output of the oracle function evaluated on the adversary output register.
  proof −
    have ‹?lhs = dist-inv aao-in-co (ket-invariant no-zero ⊓ ket-invariant {(x, y, D). D x = Some y})
co-ext-final›
      apply (rule arg-cong3[where f=dist-inv])
      by (auto intro!: simp: no-zero-def ket-invariant-inter)
    also have ‹... ≤ sqrt ((dist-inv aao-in-co (ket-invariant no-zero) co-ext-final)^2
                   + (dist-inv aao-in-co (ket-invariant {(x, y, D). D x = Some y}) co-ext-final)^2)›
      apply (rule dist-inv-intersect)
      by auto
    also have ‹... ≤ sqrt ((5 ∗ (q+1) / sqrt N)^2 + (6 / sqrt N)^2)›
      apply (rule real-sqrt-le-mono)
      apply (rule add-mono)
      using dist-zero dist-Dxy
      by auto
    also have ‹... ≤ (5 ∗ q + 11) / sqrt N›
      apply (rule sqrt-sum-squares-le-sum[THEN order-trans])
      by (auto, argo)
    finally show ?thesis
      by (simp add: d-def)
  qed
  then have ‹dist-inv aao-in-co (ket-invariant {(x, y, D::'x ⇀ 'y). y ≠ 0}) co-ext-final ≤ d›
    — In CO-execution, after the adversary's final query, the adversary output register is not 0.
    apply (rule le-back-subst-le)
    apply (rule dist-inv-mono)
    by (auto intro!: ranI)
```

then have ‹*dist-inv* (*adv-output-in-co*; *aux-in-co*) (*ket-invariant* $\{(x,\ y).\ y \neq 0\}$) *co-ext-final* $\leq$ *d*›
— As before, but with respect to a different register (without the oracle register that doesn't exist in XO).
apply (*rule le-back-subst*)
apply (*rule dist-inv-register-rewrite*)
apply (*simp*, *simp*)
apply (*rewrite at* ‹$\{(x,\ y,\ D).\ y \neq 0\}$›
to ‹$(\lambda((a,b),c).\ (a,b,c))$ ‘ $(\{(x,\ y) \mid x\ y.\ y \neq 0\} \times UNIV)$› *DEADID.rel-mono-strong*)
apply *force*
by (*simp add*: *ket-invariant-image-assoc pair-o-assoc pair-o-assoc*[*unfolded o-def*] *lift-inv-prod aao-in-co-def flip*: *lift-invariant-comp*[*unfolded o-def*, *THEN fun-cong*])
then have ‹*dist-inv-avg* (*adv-output-in-xo*; *aux-in-xo*) (λh. *ket-invariant* $\{(x,\ y).\ y \neq 0\}$) *xo-ext-final* $\leq$ *d*›
— In XO-execution, after the adversary's final query, the auxiliary register is not 0.
apply (*rule le-back-subst*)
unfolding *co-ext-final-def xo-ext-final-def*
apply (*rewrite at* ‹(*adv-output-in-co*;*aux-in-co*)› *to* ‹*Fst o* (*adv-output-in-xo*;*aux-in-xo*)› *DEADID.rel-mono-strong*)
apply (*simp add*: *adv-output-in-co-def aux-in-co-def register-comp-pair*)
by (*simp add*: *initial-state-in-co-def dist-inv-exec-query'-exec-fixed*)
then have ‹*dist-inv-avg* (*adv-output-in-xo*; *aux-in-xo*)
(λh. *ket-invariant* $\{(x,\ y).\ h\ x \neq 0 \lor y \neq 0\}$) *xo-final* $\leq$ *d*›
— In XO-execution, before the adversary's final query, $h\ x \neq 0$ or $y \neq 0$.
apply (*rule le-back-subst-le*)
unfolding *xo-ext-final-xo-final*[*abs-def*]
apply (*subst dist-inv-avg-apply-iff*)
by (*auto intro*!: *ext dist-inv-avg-mono simp*: *function-oracle-ket-invariant*)
then have $*$: ‹*dist-inv-avg* (*adv-output-in-xo*; *aux-in-xo*)
(λh. *ket-invariant* $\{(x,\ y).\ h\ x \neq 0\}$) *xo-final* $\leq$ *d*›
— In XO-execution, before the adversary's final query, $h\ x \neq 0$.
apply (*rule le-back-subst-le*)
apply (*rule ord-le-eq-trans*)
apply (*rule dist-inv-avg-mono*[**where** *I*=‹λh. *ket-invariant* $\{(x,\ y).\ h\ x \neq 0 \lor y \neq 0\}$ $\sqcap$ *ket-invariant* $\{(x,y).\ y=0\}$›])
apply (*auto simp*: *ket-invariant-inter*)[*2*]
apply (*rule dist-inv-avg-intersect*)
apply *simp-all*[*2*]
by (*fact xo-final-has-y0*)
then have ‹*dist-inv-avg adv-output-in-xo*
(λh. *ket-invariant* $\{x.\ h\ x \neq 0\}$) *xo-final* $\leq$ *d*›
apply (*subst dist-inv-avg-register-rewrite*[**where** *R*=‹(*adv-output-in-xo*; *aux-in-xo*)› **and** *J*=‹λh. *ket-invariant* $\{(x,\ y).\ h\ x \neq 0\}$›])
apply (*simp*, *simp*)
apply (*rewrite at* ‹$\{(x,\ y).\ h\ x \neq 0\}$› **in for** (*h*) *to* ‹$\{x.\ h\ x \neq 0\} \times UNIV$› *DEADID.rel-mono-strong*)
apply *fastforce*
by (*simp add*: *lift-inv-prod*)
then have ‹*dist-inv-avg adv-output* (λh. *ket-invariant* $\{x.\ h\ x \neq 0\}$) *final* $\leq$ *d*›
by (*simp add*: *xo-final-final*[*abs-def*] *adv-output-in-xo-def dist-inv-avg-Fst-tensor*)
then have ‹$(\sum h{\in}UNIV.\ \sum x|h\ x = 0.$ *measurement-probability adv-output* (*final h*) *x*) / *CARD*('*x* $\Rightarrow$ '*y*) $\leq d^2$›
apply (*subst dist-inv-avg-measurement-probability*)
apply *simp*
apply (*rewrite at* ‹$-\ \{x.\ h\ x = 0\}$› **in** ‹λh. ¤› *to* ‹$\{x.\ h\ x \neq 0\}$› *DEADID.rel-mono-strong*)
apply *blast*
by *auto*
also have ‹$d^2 = (5 * q + 11)^2\ /\ N$›

by (*simp add*: *d-def power2-eq-square*)
finally show *?thesis*
by (*simp add*: *final-def q-def*)
qed

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*: ‹$12 * (r^2+154)\hat{}3 - (10/3 * (r+2)\hat{}3 + 20)^2 \neq 0$› **if** ‹$r \geq 0$› **for** r :: *real*
by *sturm*

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-collision* = {(x,y,D::'x⇀'y). *inj-map* D}›
definition ‹*no-collision'* = {D::'x⇀'y. *inj-map* D}›

lemma *no-collision-no-collision'*: ‹*no-collision* = *UNIV* × *UNIV* × *no-collision'*›
by (*auto intro*!: *simp*: *no-collision-def no-collision'-def*)

lemma *ket-invariant-no-collision-no-collision'*: ‹*ket-invariant no-collision* = $\top \otimes_S \top \otimes_S$ *ket-invariant no-collision'*›
by (*auto simp*: *ket-invariant-tensor no-collision-no-collision' simp flip*: *ket-invariant-UNIV*)

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

lemma *preserves-no-collision*: ‹*preserves-ket query* (*no-collision* ∩ *num-queries* q) *no-collision* (6 ∗ *sqrt* q / *sqrt* N)›
proof –
define K **where** ‹K = (λ(x::'x,D0::'x⇀'y). *ket-invariant* {(x, y, D0(x:=d)) | (y::'y) d. D0 x = *None* ∧ *card* (*dom* D0) ≤ q ∧ *inj-map* D0})›

define *I1 J1* :: ‹('x⇀'y) ⇒ ('y × 'y *option*) *set*›
where ‹*I1* D0 = *UNIV* × (*if card* (*dom* D0) ≤ q *then* – *Some* ‘ *ran* D0 *else* {})›
and ‹*J1* D0 = (*UNIV* × – *Some* ‘ *ran* D0)›

for *D0* :: ‹*'x* ⇀ *'y*›

show *?thesis*
proof (*rule inv-split-reg-query*[**where** *X*=‹*reg-1-3*› **and** *Y*=‹*reg-2-3*› **and** *H*=‹*reg-3-3*› **and** *K*=*K*
and *?I1.0*=‹λ(*x*,*D0*). *ket-invariant* (*I1 D0*)› **and** *?J1.0*=‹λ(*x*,*D0*). *ket-invariant* (*J1 D0*)›])
show ‹*query* = (*reg-1-3*;(*reg-2-3*;*reg-3-3*)) *query*›
by (*auto simp*: *reg-1-3-def reg-2-3-def reg-3-3-def pair-Fst-Snd*)
show ‹*compatible reg-1-3 reg-2-3*› ‹*compatible reg-1-3 reg-3-3*› ‹*compatible reg-2-3 reg-3-3*›
by *simp-all*
show ‹*compatible-register-invariant reg-2-3* (*K xD0*)› **for** *xD0*
apply (*cases xD0*)
by (*auto simp add*: *K-def reg-2-3-def*
intro!: *compatible-register-invariant-Snd-comp compatible-register-invariant-Fst*)
show ‹*compatible-register-invariant* (*reg-3-3 o function-at* (*fst xD0*)) (*K xD0*)› **for** *xD0*
apply (*cases xD0*)
by (*auto simp add*: *K-def reg-3-3-def comp-assoc*
intro!: *compatible-register-invariant-Snd-comp compatible-register-invariant-Fst*
compatible-register-invariant-function-at)

have *aux*: ‹*inj-map b* ⟹
card (*dom b*) ≤ *q* ⟹
∃ *ba*. (*card* (*dom ba*) ≤ *q* ⟶
(∃ *d*. *b* = *ba*(*a* := *d*)) ∧ *ba a* = *None* ∧ *inj-map ba* ∧ *b a* ∉ *Some* ‘ *ran ba*) ∧
card (*dom ba*) ≤ *q*› **for** *b a*
apply (*intro exI*[*of* - ‹*b*(*a*:=*None*)›] *exI*[*of* - ‹*b a*›] *impI conjI*)
apply *fastforce*
apply *force*
apply (*smt* (*verit*, *ccfv-SIG*) *array-rules*(*2*) *inj-map-def*)
apply (*auto simp*: *ran-def inj-map-def*)[*1*]
by (*simp add*: *dom-fun-upd card-Diff1-le*[*THEN order-trans*])
show ‹*ket-invariant* (*no-collision* ∩ *num-queries q*)
≤ (*SUP xD0*∈*UNIV*. *K xD0* ⊓ *lift-invariant* (*reg-2-3*;*reg-3-3* ∘ *function-at* (*fst xD0*)) (*case xD0 of* (*x*, *D0*) ⇒ *ket-invariant* (*I1 D0*)))›
by (*auto intro*!: *aux simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*] *I1-def*
lift-inv-prod lift-invariant-comp lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv case-prod-beta
no-collision-def num-queries-def)
have *aux*: ‹*d* ∉ *Some* ‘ *ran* (*snd xD0*) ⟹ *inj-map* (*snd xD0*) ⟹ *inj-map* ((*snd xD0*)(*fst xD0* := *d*))› **for** *d xD0*
by (*smt* (*verit*, *del-insts*) *fun-upd-other fun-upd-same image-iff inj-map-def not-Some-eq ranI*)
show ‹*K xD0* ⊓ *lift-invariant* (*reg-2-3*;*reg-3-3* ∘ *function-at* (*fst xD0*)) (*case xD0 of* (*x*, *D0*) ⇒ *ket-invariant* (*J1 D0*))
≤ *ket-invariant no-collision*› **for** *xD0*
apply (*simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter*
ket-invariant-SUP[*symmetric*] *J1-def lift-inv-prod lift-invariant-comp*
lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv case-prod-beta)
unfolding *no-collision-def*
using *aux*[*of* - *xD0*] **by** *auto*
have *aux*: ‹*b aa* = *None* ⟹
ba aa = *None* ⟹
b ≠ *ba* ⟹
card (*dom b*) ≤ *q* ⟹
inj-map b ⟹ *card* (*dom ba*) ≤ *q* ⟹ *inj-map ba* ⟹ *b*(*aa* := *d*) ≠ *ba*(*aa* := *da*)›
for *aa b ba d da*

by (*metis fun-upd-triv fun-upd-upd*)
have *aux*: ‹⋀*b aa ba d da*.
b(*aa* := *d*) = *ba*(*aa* := *da*) ⟹
b aa = *None* ⟹
ba aa = *None* ⟹
b ≠ *ba* ⟹
card (*dom b*) ≤ *q* ⟹
inj-map b ⟹ *card* (*dom ba*) ≤ *q* ⟹ *inj-map ba* ⟹ *False*›
by (*metis fun-upd-triv fun-upd-upd*)
show ‹*orthogonal-spaces* (*K xD0*) (*K xD0′*)› **if** ‹*xD0* ≠ *xD0′*› **for** *xD0 xD0′*
apply (*cases xD0*; *cases xD0′*)
unfolding *K-def* **using** *that* **by** (*auto elim*!: *aux*)
have ‹*preserves-ket query1* (*I1 D0*) (*J1 D0*) (*6* ∗ *sqrt q* / *sqrt N*)› **for** *D0* :: ‹′*x*⇀′*y*›
proof (*cases* ‹*card* (*dom D0*) ≤ *q*›)
case *True*
have [*simp*]: ‹*card* (*ran D0*) ≤ *q*›
using *True ran-smaller-dom*[*of D0*] **by** *simp*
show *?thesis*
apply (*simp add*: *I1-def J1-def True*)
apply (*rule preserve-query1-simplified*)
by (*auto simp add*: *inj-vimage-image-eq vimage-Compl*)
next
case *False*
then show *?thesis*
unfolding *I1-def* **by** *simp*
qed
then show ‹*preserves query1* (*case xD0 of* (*x*, *D0*) ⇒ *ket-invariant* (*I1 D0*)) (*case xD0 of* (*x*::′*x*, *D0*) ⇒ *ket-invariant* (*J1 D0*)) (*6* ∗ *sqrt q* / *sqrt N*)› **for** *xD0*
apply (*cases xD0*) **by** *auto*
show ‹*6* ∗ *sqrt q* / *sqrt N* ≥ *0*›
by *auto*
show ‹*K xD0* ≤ *lift-invariant reg-1-3* (*ket-invariant* {*fst xD0*})› **for** *xD0*
apply (*cases xD0*)
by (*auto simp add*: *K-def reg-1-3-def lift-Fst-ket-inv*)
qed *simp*
qed

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

lemma *preserves-no-collision′*: ‹*preserves-ket query′* (*no-collision* ∩ *num-queries q*) *no-collision* (*5* ∗ *sqrt q* / *sqrt N*)›
proof −
define *K* **where** ‹*K* = (λ(*x*::′*x*,*D0*::′*x*⇀′*y*). *ket-invariant* {(*x*, *y*, *D0*(*x*:=*d*)) | (*y*::′*y*) *d*.
D0 x = *None* ∧ *card* (*dom D0*) ≤ *q* ∧ *inj-map D0*})›

define *I1 J1* :: ‹(′*x*⇀′*y*) ⇒ (′*y* × ′*y option*) *set*›
where ‹*I1 D0* = *UNIV* × (*if card* (*dom D0*) ≤ *q then* − *Some* ‘ *ran D0 else* {})›
and ‹*J1 D0* = (*UNIV* × − *Some* ‘ *ran D0*)›
for *D0* :: ‹′*x* ⇀ ′*y*›

show *?thesis*
proof (*rule inv-split-reg-query′*[**where** *X*=‹*reg-1-3*› **and** *Y*=‹*reg-2-3*› **and** *H*=‹*reg-3-3*› **and** *K*=*K*
and *?I1.0*=‹λ(*x*,*D0*). *ket-invariant* (*I1 D0*)› **and** *?J1.0*=‹λ(*x*,*D0*). *ket-invariant* (*J1 D0*)›])
show ‹*query′* = (*reg-1-3*;(*reg-2-3*;*reg-3-3*)) *query′*›
by (*simp add*: *reg-1-3-def reg-2-3-def reg-3-3-def pair-Fst-Snd*)
show ‹*compatible reg-1-3 reg-2-3*› ‹*compatible reg-1-3 reg-3-3*› ‹*compatible reg-2-3 reg-3-3*›

by *simp-all*
show ‹*compatible-register-invariant reg-2-3* (*K xD0*)› **for** *xD0*
apply (*cases xD0*)
by (*auto simp add*: *K-def reg-2-3-def*
intro!: *compatible-register-invariant-Snd-comp compatible-register-invariant-Fst*)
show ‹*compatible-register-invariant* (*reg-3-3 o function-at* (*fst xD0*)) (*K xD0*)› **for** *xD0*
apply (*cases xD0*)
by (*auto simp add*: *K-def reg-3-3-def comp-assoc*
intro!: *compatible-register-invariant-Snd-comp compatible-register-invariant-Fst*
compatible-register-invariant-function-at)
have *aux*: ‹*inj-map b* ⟹
card (*dom b*) ≤ *q* ⟹
∃ *ba*. (*card* (*dom ba*) ≤ *q* ⟶
(∃ *d*. *b* = *ba*(*a* := *d*)) ∧ *ba a* = *None* ∧ *inj-map ba* ∧ *b a* ∉ *Some* ‘ *ran ba*) ∧
card (*dom ba*) ≤ *q*› **for** *a b*
apply (*intro exI*[*of* - ‹*b*(*a*:=*None*)›] *exI*[*of* - ‹*b a*›] *impI conjI*)
apply *fastforce*
apply *force*
apply (*smt* (*verit*, *ccfv-SIG*) *array-rules*(*2*) *inj-map-def*)
apply (*auto simp*: *ran-def inj-map-def*)[*1*]
by (*simp add*: *dom-fun-upd card-Diff1-le*[*THEN order-trans*])
show ‹*ket-invariant* (*no-collision* ∩ *num-queries q*)
≤ (*SUP xD0*∈*UNIV*. *K xD0* ⊓ *lift-invariant* (*reg-2-3*;*reg-3-3* ∘ *function-at* (*fst xD0*)) (*case xD0 of* (*x*, *D0*) ⇒ *ket-invariant* (*I1 D0*)))›
by (*auto intro*!: *aux simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter ket-invariant-SUP*[*symmetric*] *I1-def*
lift-inv-prod lift-invariant-comp lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv case-prod-beta
no-collision-def num-queries-def)
show ‹*K xD0* ⊓ *lift-invariant* (*reg-2-3*;*reg-3-3* ∘ *function-at* (*fst xD0*)) (*case xD0 of* (*x*, *D0*) ⇒ *ket-invariant* (*J1 D0*))
≤ *ket-invariant no-collision*› **for** *xD0*
proof −
have *aux*: ‹*d* ∉ *Some* ‘ *ran* (*snd xD0*) ⟹
snd xD0 (*fst xD0*) = *None* ⟹
card (*dom* (*snd xD0*)) ≤ *q* ⟹ *inj-map* (*snd xD0*) ⟹ *inj-map* ((*snd xD0*)(*fst xD0* := *d*))› **for** *d*
by (*smt* (*verit*, *del-insts*) *fun-upd-other fun-upd-same image-iff inj-map-def not-Some-eq ranI*)
show *?thesis*
apply (*simp add*: *K-def lift-Fst-ket-inv reg-1-3-def reg-2-3-def ket-invariant-inter*
ket-invariant-SUP[*symmetric*] *J1-def lift-inv-prod lift-invariant-comp*
lift-invariant-function-at-ket-inv reg-3-3-def lift-Snd-ket-inv case-prod-beta no-collision-def)
using *aux* **by** *auto*
qed
have *aux*: ‹*b aa* = *None* ⟹ *ba aa* = *None* ⟹ *b* ≠ *ba* ⟹ *b*(*aa* := *d*) = *ba*(*aa* := *da*) ⟹ *False*› **for** *aa b ba d da*
by (*metis fun-upd-triv fun-upd-upd*)
show ‹*orthogonal-spaces* (*K xD0*) (*K xD0'*)› **if** ‹*xD0* ≠ *xD0'*› **for** *xD0 xD0'*
apply (*cases xD0*; *cases xD0'*)
unfolding *K-def* **using** *that aux* **by** *auto*
have ‹*preserves-ket query1'* (*I1 D0*) (*J1 D0*) (*5* ∗ *sqrt q* / *sqrt N*)› **for** *D0* :: ‹'*x*⇀'*y*›
proof (*cases* ‹*card* (*dom D0*) ≤ *q*›)
case *True*
have [*simp*]: ‹*card* (*ran D0*) ≤ *q*›
using *True ran-smaller-dom*[*of D0*] **by** *simp*

show *?thesis*
apply (*simp add*: *I1-def J1-def True*)
apply (*rule preserve-query1′-simplified*)
by (*auto simp add*: *inj-vimage-image-eq vimage-Compl*)
next
case *False*
then show *?thesis*
unfolding *I1-def* **by** *simp*
qed
then show ‹*preserves query1′* (*case xD0 of* (*x*, *D0*) ⇒ *ket-invariant* (*I1 D0*)) (*case xD0 of* (*x*::′*x*, *D0*) ⇒ *ket-invariant* (*J1 D0*)) (*5* ∗ *sqrt q* / *sqrt N*)› **for** *xD0*
apply (*cases xD0*) **by** *auto*
show ‹*5* ∗ *sqrt q* / *sqrt N* ≥ *0*›
by *auto*
show ‹*K xD0* ≤ *lift-invariant reg-1-3* (*ket-invariant* {*fst xD0*})› **for** *xD0*
apply (*cases xD0*)
by (*auto simp add*: *K-def reg-1-3-def lift-Fst-ket-inv*)
qed *simp*
qed

lemma *preserves-no-collision-num*: ‹*preserves-ket query* (*no-collision* ∩ *num-queries q*) (*no-collision* ∩ *num-queries* (*q*+*1*)) (*6* ∗ *sqrt q* / *sqrt N*)›
apply (*subst add-0-right*[*of* ‹*6* ∗ *sqrt q* / *sqrt N*›, *symmetric*])
apply (*rule preserves-intersect-ket*)
apply (*rule preserves-no-collision*)
apply (*rule preserves-mono*[*OF preserves-num*])
by *auto*

lemma *preserves-no-collision′-num*: ‹*preserves-ket query′* (*no-collision* ∩ *num-queries q*) (*no-collision* ∩ *num-queries* (*q*+*1*)) (*5* ∗ *sqrt q* / *sqrt N*)›
apply (*subst add-0-right*[*of* ‹*5* ∗ *sqrt q* / *sqrt N*›, *symmetric*])
apply (*rule preserves-intersect-ket*)
apply (*rule preserves-no-collision′*)
apply (*rule preserves-mono*[*OF preserves-num′*])
by *auto*

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* × ′*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*∈*UNIV*. $\sum$ (*x1*,*x2*)|*x1* ≠ *x2* ∧ *h x1* = *h x2*. *measurement-probability adv-output* (*exec-program h program initial-state*) (*x1*,*x2*)) / *CARD*(′*x* ⇒ ′*y*)
≤ *12* ∗ (*query-count program* + *154*)^*3* / *N*›
proof −
note [[*simproc del*: *Laws-Quantum.compatibility-warn*]]

In this game based proof, we consider three different quantum memory models:

- The one from the statement of the lemma, where the overall quantum state lives in ′*mem*, and the adversary output register is described by *adv-output*, and the initial state in *initial-state*. The program *program* assumes this memory model.

- The "extra output" (short XO) memory model, where there is an extra auxiliary register *aux* of type $'y \times 'y$. The type of the memory is then $'mem \times 'y \times 'y$. (I.e., the extra register is in addition to the content of $'mem$.)
- The "compressed oracle" (short CO) memory model, where additionally to XO, we have an oracle register that can holds the content of the compressed oracle (or the standard oracle).

Since the register *adv-output* is defined w.r.t. a specific memory, we define convenience definitions for the same register as it would be accessed in the other memories:

define *adv-output-in-xo* :: ‹$('x\times'x)$ *update* ⇒ $('mem\times'y\times'y)$ *update*› **where** ‹*adv-output-in-xo* = *Fst o adv-output*›
define *adv-output-in-co* :: ‹$('x\times'x)$ *update* ⇒ $(('mem\times'y\times'y) \times ('x \rightharpoonup 'y))$ *update*› **where** ‹*adv-output-in-co* = *Fst o adv-output-in-xo*›

Analogously, we defined the *aux*-register and the oracle register in the applicable memories:

define *aux-in-xo* :: ‹$('y\times'y)$ *update* ⇒ $('mem\times'y\times'y)$ *update*› **where** ‹*aux-in-xo* = *Snd*›
define *aux-in-co* :: ‹$('y\times'y)$ *update* ⇒ $(('mem\times'y\times'y) \times ('x \rightharpoonup 'y))$ *update*› **where** ‹*aux-in-co* = *Fst o aux-in-xo*›
define *oracle-in-co* :: ‹$('x \rightharpoonup 'y)$ *update* ⇒ $(('mem\times'y\times'y) \times ('x \rightharpoonup 'y))$ *update*› **where** ‹*oracle-in-co* = *Snd*›
define *aao-in-co* **where** ‹*aao-in-co* = (*adv-output-in-co*; (*aux-in-co*; *oracle-in-co*))›
— Abbreviation since we use this combination often.

have [*register*]: ‹*compatible aux-in-co oracle-in-co*›
by (*simp add*: *adv-output-in-co-def aux-in-co-def oracle-in-co-def adv-output-in-xo-def aux-in-xo-def*)
have [*register*]: ‹*compatible adv-output-in-xo aux-in-xo*›
by (*simp add*: *adv-output-in-xo-def aux-in-xo-def*)
have [*register*]: ‹*compatible adv-output-in-co aux-in-co*›
by (*simp add*: *adv-output-in-co-def aux-in-co-def*)
have [*register*]: ‹*compatible adv-output-in-co oracle-in-co*›
by (*simp add*: *adv-output-in-co-def oracle-in-co-def*)
have [*register*]: ‹*compatible aux-in-xo Fst*›
by (*simp add*: *aux-in-xo-def*)
have [*register*]: ‹*compatible aux-in-co* (*Fst o Fst*)›
by (*simp add*: *aux-in-co-def*)
have [*register*]: ‹*compatible aux-in-co Snd*›
by (*simp add*: *aux-in-co-def*)
have [*register*]: ‹*register aao-in-co*›
by (*simp add*: *aao-in-co-def*)

The initial states in XO/CO are like the original initial state, but with *ket* $(0,\ 0)$ in *aux* and *ket* $(\lambda x.\ None)$ (the fully undefined function) in the oracle register.

define *initial-state-in-xo* **where** ‹*initial-state-in-xo* = *initial-state* $\otimes_s$ *ket* $((0,0) :: 'y\times'y)$›
define *initial-state-in-co* :: ‹$(('mem\times'y\times'y) \times ('x \rightharpoonup 'y))$ *ell2*› **where** ‹*initial-state-in-co* = *initial-state-in-xo* $\otimes_s$ *ket Map.empty*›

We define an extended program *ext-program* that executes *program*, followed by two additional queries to the oracle. Input register is the adversary output register. Output register is the additional register *aux*. Hence *ext-program* is only meaningful in the models XO and CO. (Our definition is for XO.)

define *ext-program* **where** ‹*ext-program* = *lift-program Fst program*

@ [*QueryStep* (*adv-output-in-xo o Fst*) (*aux-in-xo o Fst*), *QueryStep* (*adv-output-in-xo o Snd*) (*aux-in-xo o Snd*)]›
have [*iff*]: ‹*valid-program ext-program*›
by (*auto intro*!: *valid-program-lift simp add*: *valid-program-append adv-output-in-xo-def aux-in-xo-def ext-program-def*)

We define the final states of the programs *program* and *ext-program*, in the original model, and in XO, and CO.

define *final* :: ‹($'x \Rightarrow 'y$) ⇒ $'mem$ *ell2*› **where** ‹*final h* = *exec-program h program initial-state*› **for** *h*
define *xo-ext-final* :: ‹($'x \Rightarrow 'y$) ⇒ ($'mem \times 'y \times 'y$) *ell2*› **where** ‹*xo-ext-final h* = *exec-program h ext-program initial-state-in-xo*› **for** *h*
define *xo-final* :: ‹($'x \Rightarrow 'y$) ⇒ ($'mem \times 'y \times 'y$) *ell2*› **where** ‹*xo-final h* = *exec-program h* (*lift-program Fst program*) *initial-state-in-xo*› **for** *h*
define *co-ext-final* :: ‹(($'mem \times 'y \times 'y$) × ($'x \rightharpoonup 'y$)) *ell2*› **where** ‹*co-ext-final* = *exec-program-with query′ ext-program initial-state-in-co*›
define *co-final* :: ‹(($'mem \times 'y \times 'y$) × ($'x \rightharpoonup 'y$)) *ell2*› **where** ‹*co-final* = *exec-program-with query′* (*lift-program Fst program*) *initial-state-in-co*›

have [*simp*]: ‹*norm initial-state-in-xo* = *1*›
by (*simp add*: *initial-state-in-xo-def norm-tensor-ell2 assms*)
have *norm-initial-state-in-co*[*simp*]: ‹*norm initial-state-in-co* = *1*›
by (*simp add*: *initial-state-in-co-def norm-tensor-ell2*)

have *norm-co-final*[*simp*]: ‹*norm co-final* ≤ *1*›
unfolding *co-final-def*
using *norm-exec-program-with valid-program-lift* ‹*valid-program program*›
norm-query′ register-Fst norm-initial-state-in-co
by *smt*

We derive the relationships between the various final states:

have *co-ext-final-prefinal*:
‹*co-ext-final* = (*adv-output-in-co o Snd*; (*aux-in-co o Snd*; *oracle-in-co*)) *query′* $*_V$
(*adv-output-in-co o Fst*; (*aux-in-co o Fst*; *oracle-in-co*)) *query′* $*_V$ *co-final*›
by (*simp add*: *co-ext-final-def ext-program-def exec-program-with-append adv-output-in-co-def aux-in-co-def oracle-in-co-def comp-assoc*
flip: *initial-state-in-co-def co-final-def*)

have *xo-final-final*: ‹*xo-final h* = *final h* $\otimes_s$ *ket* (*0*,*0*)› **for** *h*
by (*simp add*: *xo-final-def final-def initial-state-in-xo-def exec-lift-program-Fst*)

have *xo-ext-final-xo-final*: ‹*xo-ext-final h* = (*adv-output-in-xo o Snd*; *aux-in-xo o Snd*) (*function-oracle h*) $*_V$
(*adv-output-in-xo o Fst*; *aux-in-xo o Fst*) (*function-oracle h*) $*_V$ *xo-final h*› **for** *h*
by (*simp add*: *xo-ext-final-def xo-final-def ext-program-def exec-program-def*)

After executing *program* (in XO), the *aux*-register is in state *ket* (*0*, *0*):

have *xo-final-has-y0*: ‹*dist-inv-avg* (*adv-output-in-xo*;*aux-in-xo*) (λ-. *ket-invariant* {(*xx*,*yy*). *yy* = (*0*,*0*)}) *xo-final* = *0*›
proof −
have ‹*dist-inv-avg aux-in-xo* (λ-::$'x \Rightarrow 'y$. *ket-invariant* {(*0*,*0*)}) *xo-final*
≤ *dist-inv-avg aux-in-xo* (λ-::$'x \Rightarrow 'y$. *ket-invariant* {(*0*,*0*)}) (λ*h*. *initial-state-in-xo*)›
unfolding *xo-final-def*
apply (*subst dist-inv-avg-exec-compatible*)
using *dist-inv-avg-exec-compatible*
by *auto*

also have ‹. . . = *0*›
  by (*auto intro*!: *tensor-ell2-in-tensor-ccsubspace ket-in-ket-invariantI*
    simp add: *initial-state-in-xo-def dist-inv-0-iff distance-from-inv-avg0I aux-in-xo-def lift-Snd-inv*)
finally have ‹*dist-inv-avg aux-in-xo* (λ-. *ket-invariant* {(*0*,*0*)}) *xo-final* = *0*›
  by (*smt* (*verit*, *ccfv-SIG*) *dist-inv-avg-pos*)
then show *?thesis*
  apply (*rewrite at* ‹{(*xx*, *yy*). *yy* = (*0*,*0*)}› *to* ‹*UNIV* × {(*0*,*0*)}› *DEADID.rel-mono-strong*, *blast*)
  apply (*subst dist-inv-avg-register-rewrite*)
  by (*simp-all add*: *lift-inv-prod*)
qed

Same as *xo-final-has-y0*, but in CO:

have *co-final-has-y0*: ‹*dist-inv aao-in-co* (*ket-invariant* {(*x*,*y*,*D*). *y* = (*0*,*0*)}) *co-final* = *0*›
proof −
  have ‹*dist-inv aux-in-co* (*ket-invariant* {(*0*,*0*)}) *co-final*
    ≤ *dist-inv aux-in-co* (*ket-invariant* {(*0*,*0*)}) *initial-state-in-co*›
  unfolding *co-final-def*
  apply (*rule dist-inv-exec′-compatible*)
  by *simp-all*
  also have ‹. . . = *0*›
  by (*auto intro*!: *tensor-ell2-in-tensor-ccsubspace ket-in-ket-invariantI*
    simp add: *initial-state-in-co-def initial-state-in-xo-def dist-inv-0-iff*
    aux-in-co-def aux-in-xo-def lift-Fst-inv lift-Snd-inv lift-invariant-comp)
  finally have ‹*dist-inv aux-in-co* (*ket-invariant* {(*0*,*0*)}) *co-final* = *0*›
  by (*smt* (*verit*, *best*) *dist-inv-pos*)
  then show *?thesis*
  apply (*rewrite at* ‹{(*xx*, *yy*, *D*). *yy* = (*0*,*0*)}› *to* ‹*UNIV* × {(*0*,*0*)} × *UNIV*› *DEADID.rel-mono-strong*, *blast*)
  apply (*subst dist-inv-register-rewrite*)
  by (*simp-all add*: *lift-inv-prod aao-in-co-def*)
qed

define *q* **where** ‹*q* = *query-count program*›

The following term occurs a lot (it's how much the *no-collision* invariant is preserved after running *ext-program*). So we abbreviate it as *d*.

define *d* :: *real* **where** ‹*d* = (*10*/*3* ∗ *sqrt* (*q*+*2*)ˆ*3* + *20*) / *sqrt N*›

have [*iff*]: ‹*d* ≥ *0*›
  by (*simp add*: *d-def*)

have ‹*dist-inv oracle-in-co* (*ket-invariant* (*no-collision′* ∩ *num-queries′* (*q*+*2*))) *co-ext-final* ≤ *10*/*3* ∗ *sqrt* (*q*+*2*)ˆ*3* / *sqrt N*›
  — In CO-execution, before the adversary's final query, the oracle register has no collision in its range (and we also track the number of queries to make the induction go through)
  unfolding *co-ext-final-def*
proof (*rule dist-inv-induct*[**where** *g*=‹λ*i*::*nat*. *5* ∗ *sqrt i* / *sqrt N*›
    and *J*=‹λ*i*. *ket-invariant* (*no-collision′* ∩ *num-queries′ i*)›])
  show ‹*compatible oracle-in-co Fst*›
    using *oracle-in-co-def* **by** *simp*
  show ‹*initial-state-in-co* ∈ *space-as-set* (*lift-invariant oracle-in-co* (*ket-invariant* (*no-collision′* ∩ *num-queries′ 0*)))›
    by (*auto intro*!: *tensor-ell2-in-tensor-ccsubspace ket-in-ket-invariantI*
      simp add: *initial-state-in-co-def oracle-in-co-def lift-Snd-ket-inv inj-map-def num-queries′-def*
      initial-state-in-xo-def tensor-ell2-ket ket-in-ket-invariantI no-collision′-def

simp flip: ket-invariant-tensor)
show ‹ket-invariant (no-collision′ ∩ num-queries′ (query-count ext-program)) ≤ ket-invariant (no-collision′ ∩ num-queries′ (q+2))›
by (simp add: q-def ext-program-def)
show ‹valid-program ext-program›
by simp
show ‹preserves ((Fst ∘ X-in-xo;(Fst ∘ Y-in-xo;Snd)) query′) (lift-invariant oracle-in-co (ket-invariant (no-collision′ ∩ num-queries′ i)))
(lift-invariant oracle-in-co (ket-invariant (no-collision′ ∩ num-queries′ (Suc i)))) (5 ∗ sqrt i / sqrt N)›
if [register]: ‹compatible X-in-xo Y-in-xo› **for** X-in-xo Y-in-xo i
proof −
from preserves-no-collision′-num
have ‹preserves ((Fst ∘ X-in-xo;(Fst ∘ Y-in-xo;Snd)) query′)
(lift-invariant (Fst ∘ X-in-xo;(Fst ∘ Y-in-xo;Snd)) (ket-invariant (no-collision ∩ num-queries i)))
(lift-invariant (Fst ∘ X-in-xo;(Fst ∘ Y-in-xo;Snd)) (ket-invariant (no-collision ∩ num-queries (i+1))))
(5 ∗ sqrt (real i) / sqrt N)›
apply (rule preserves-lift-invariant[THEN iffD2, rotated])
by simp
moreover have ‹lift-invariant (Fst ∘ X-in-xo;(Fst ∘ Y-in-xo;Snd)) (ket-invariant (no-collision ∩ num-queries i))
= lift-invariant oracle-in-co (ket-invariant (no-collision′ ∩ num-queries′ i))› **for** i
by (simp add: oracle-in-co-def no-collision-no-collision′ num-queries-num-queries′ lift-inv-prod Times-Int-Times)
ultimately show ?thesis
by simp
qed
show ‹norm query′ ≤ 1›
by simp
show ‹norm initial-state-in-co ≤ 1›
by simp
show ‹(∑ i<query-count ext-program. 5 ∗ sqrt i / sqrt N) ≤ 10/3 ∗ sqrt (q+2)^3 / sqrt N›
proof −
have ‹(∑ i<q+2. sqrt i) ≤ 2/3 ∗ sqrt (q+2) ^ 3›
by (rule sum-sqrt)
then have ‹(∑ i<q+2. 5 ∗ sqrt i / sqrt N) ≤ 5 ∗ (2/3 ∗ sqrt (q+2) ^ 3) / sqrt N›
by (auto intro!: divide-right-mono real-sqrt-ge-zero simp only: simp flip: sum-distrib-left sum-divide-distrib)
also have ‹... = 10/3 ∗ sqrt (q+2)^3 / sqrt N›
by simp
finally
show ‹(∑ i<query-count ext-program. 5 ∗ sqrt i / sqrt N) ≤ 10/3 ∗ sqrt (q+2)^3 / sqrt N›
by (simp add: q-def ext-program-def)
qed
qed

then have ‹dist-inv oracle-in-co (ket-invariant no-collision′) co-ext-final ≤ 10/3 ∗ sqrt (q+2)^3 / sqrt N›
— Like the previous but without the number of queries)
apply (rule le-back-subst-le)
apply (rule dist-inv-mono)
by auto

then have dist-collision: ‹dist-inv aao-in-co (ket-invariant no-collision) co-ext-final ≤ 10/3 ∗ sqrt

$(q+2)$^3 / *sqrt N*›
— Same thing, but expressed w.r.t. different register
apply (*rule le-back-subst*)
apply (*rule dist-inv-register-rewrite*)
by (*auto intro*!: *simp*: *aao-in-co-def no-collision-no-collision′ lift-inv-prod*)

have *dist-Dxy*: ‹*dist-inv aao-in-co* (*ket-invariant* $\{((x1,x2),(y1,y2),D).\ D\ x1 = Some\ y1 \wedge D\ x2 = Some\ y2\}$) *co-ext-final* $\leq$ *20 / sqrt N*›
proof –
have *aao-in-co-decomp*: ‹*aao-in-co* = ((*adv-output-in-co o Fst*; *adv-output-in-co o Snd*); ((*aux-in-co o Fst*; *aux-in-co o Snd*); *oracle-in-co*))›
by (*simp add*: *register-pair-Snd register-pair-Fst aao-in-co-def flip*: *register-comp-pair comp-assoc*)
have ‹*dist-inv* ((*adv-output-in-co* ∘ *Fst*;*adv-output-in-co* ∘ *Snd*);((*aux-in-co* ∘ *Fst*;*aux-in-co* ∘ *Snd*);*oracle-in-co*))
(*ket-invariant* $\{((x1,\ x2),\ (y1,\ y2),\ D).\ y1 = 0 \wedge y2 = 0\}$) *co-final* = *0*›
using *co-final-has-y0*
by (*simp add*: *aao-in-co-decomp case-prod-unfold prod-eq-iff*)
then show *?thesis*
apply (*rewrite at* ‹*20 / sqrt N*› *to* ‹*0 + 20 / sqrt N*› *DEADID.rel-mono-strong*, *simp*)
unfolding *co-ext-final-prefinal aao-in-co-decomp*
apply (*rule dist-inv-double-query′*)
by (*simp-all add*: *aao-in-co-decomp*)
qed

have ‹*dist-inv aao-in-co*
(*ket-invariant* $\{((x1,x2),(y1,y2),D).\ inj\text{-}map\ D \wedge D\ x1 = Some\ y1 \wedge D\ x2 = Some\ y2\}$) *co-ext-final* $\leq$ *d*› (**is** ‹*?lhs* $\leq$ *d*›)
— In CO-execution, after the adversary's final query, the oracle register has no collision, and the aux register contains the outputs of the oracle function evaluated on the adversary output registers.
proof –
have ‹*?lhs* = *dist-inv aao-in-co* (*ket-invariant no-collision* ⊓ *ket-invariant* $\{((x1,x2),(y1,y2),D).\ D\ x1 = Some\ y1 \wedge D\ x2 = Some\ y2\}$) *co-ext-final*›
apply (*rule arg-cong3*[**where** *f*=*dist-inv*])
by (*auto intro*!: *simp*: *no-collision-def ket-invariant-inter*)
also have ‹... $\leq$ *sqrt* ((*dist-inv aao-in-co* (*ket-invariant no-collision*) *co-ext-final*)^2
+ (*dist-inv aao-in-co* (*ket-invariant* $\{((x1,x2),(y1,y2),D).\ D\ x1 = Some\ y1 \wedge D\ x2 = Some\ y2\}$) *co-ext-final*)^2)›
apply (*rule dist-inv-intersect*)
by *auto*
also have ‹... $\leq$ *sqrt* ((*10/3 ∗ sqrt* $(q+2)$^3 / *sqrt N*)^2 + (*20 / sqrt N*)^2)›
apply (*rule real-sqrt-le-mono*)
apply (*rule add-mono*)
using *dist-collision dist-Dxy*
by *auto*
also have ‹... $\leq$ (*10/3 ∗ sqrt* $(q+2)$^3 + *20*) / *sqrt N*›
apply (*rule sqrt-sum-squares-le-sum*[*THEN order-trans*])
by (*auto*, *argo*)
finally show *?thesis*
by (*simp add*: *d-def*)
qed
then have ‹*dist-inv aao-in-co* (*ket-invariant* $\{((x1,x2),(y1,y2),D).\ x1 \neq x2 \longrightarrow y1 \neq y2\}$) *co-ext-final* $\leq$ *d*›
— In CO-execution, after the adversary's final query, the auxiliary registers are non-equal (if the adversary registers are).
apply (*rule le-back-subst-le*)
apply (*rule dist-inv-mono*)

by (*auto simp*: *inj-map-def*)
then have ‹*dist-inv* (*adv-output-in-co*; *aux-in-co*) (*ket-invariant* $\{((x1,x2), (y1,y2)).\ x1 \neq x2 \longrightarrow y1 \neq y2\}$) *co-ext-final* $\leq$ *d*›
— As before, but with respect to a different register (without the oracle register that doesn't exist in XO).
apply (*rule le-back-subst*)
apply (*rule dist-inv-register-rewrite*)
apply (*simp*, *simp*)
apply (*rewrite at* ‹(*adv-output-in-co*;*aux-in-co*)› *to* ‹*aao-in-co o* (*reg-1-3*; *reg-2-3*)› *DEADID.rel-mono-strong*)
apply (*simp add*: *aao-in-co-def flip*: *register-comp-pair*)
apply (*subst lift-invariant-comp*, *simp*)
by (*auto intro*!: *simp*: *lift-inv-prod′ reg-1-3-def reg-3-3-def reg-2-3-def lift-invariant-comp lift-Snd-ket-inv lift-Fst-ket-inv*
ket-invariant-inter case-prod-unfold
simp flip: *ket-invariant-SUP*)
then have $*$: ‹*dist-inv-avg* (*adv-output-in-xo*; *aux-in-xo*) ($\lambda h.$ *ket-invariant* $\{((x1,x2), (y1,y2)).\ x1 \neq x2 \longrightarrow y1 \neq y2\}$) *xo-ext-final* $\leq$ *d*›
— In XO-execution, after the adversary's final query, the adversary output register is not 0.
apply (*rule le-back-subst*)
unfolding *co-ext-final-def xo-ext-final-def*
apply (*rewrite at* ‹(*adv-output-in-co*;*aux-in-co*)› *to* ‹*Fst o* (*adv-output-in-xo*;*aux-in-xo*)› *DEADID.rel-mono-strong*)
apply (*simp add*: *adv-output-in-co-def aux-in-co-def register-comp-pair*)
by (*simp add*: *initial-state-in-co-def dist-inv-exec-query′-exec-fixed*)
have ‹*dist-inv-avg* (*adv-output-in-xo*; *aux-in-xo*)
($\lambda h.$ *ket-invariant* $\{((x1,x2), yy).\ (x1 \neq x2 \longrightarrow h\ x1 \neq h\ x2) \vee yy \neq (0,0)\}$) *xo-final* $\leq$ *d*›
— In XO-execution, before the adversary's final query, x1,x2 are a collision, or the aux register is nonzero.
proof −
define *state2* **where** ‹*state2 h* = (*adv-output-in-xo o Fst*; *aux-in-xo o Fst*) (*function-oracle h*) $*_V$ *xo-final h*› **for** *h*
have *xo-ext-final-state2*: ‹*xo-ext-final h* = (*adv-output-in-xo* ∘ *Snd*;*aux-in-xo* ∘ *Snd*) (*function-oracle h*) $*_V$ *state2 h*› **for** *h*
using *state2-def xo-ext-final-xo-final* **by** *presburger*
have *fo-apply2*: ‹(*Snd* $\otimes_r$ *Snd*) (*function-oracle h*)$*$ $*_S$ *ket-invariant* $\{((x1, x2), y1, y2).\ x1 \neq x2 \longrightarrow y1 \neq y2\}$
$\leq$ *ket-invariant* $\{((x1,x2), (y1,y2)).\ (x1 \neq x2 \longrightarrow y1 \neq h\ x2) \vee y2 \neq 0\}$› **for** *h* :: ‹′*x* ⇒ ′*y*›
proof −
have ‹(*Snd* $\otimes_r$ *Snd*) (*function-oracle h*)$*$ $*_S$ *ket-invariant* $\{((x1, x2), y1, y2).\ x1 \neq x2 \longrightarrow y1 \neq y2\}$
= (*Snd* $\otimes_r$ *Snd*) (*function-oracle h*) $*_S$ *ket-invariant* $\{((x1, x2), y1, y2).\ x1 \neq x2 \longrightarrow y1 \neq y2\}$›
by (*simp add*: *uminus-y flip*: *register-adj*)
also have ‹. . . = *lift-invariant* (*Fst* $\otimes_r$ *Fst*;*Snd* $\otimes_r$ *Snd*) (*Snd* (*function-oracle h*) $*_S$ *ket-invariant* $\{((x1, y1), x2, y2).\ x1 \neq x2 \longrightarrow y1 \neq y2\}$)›
apply (*rewrite at* ‹(*Snd* $\otimes_r$ *Snd*)› *to* ‹(*Fst* $\otimes_r$ *Fst*; *Snd* $\otimes_r$ *Snd*) *o Snd*› *DEADID.rel-mono-strong*)
apply (*simp add*: *register-pair-Snd compatible-register-tensor*)
apply (*rewrite at* ‹*ket-invariant* $\{((x1, x2), y1, y2).\ x1 \neq x2 \longrightarrow y1 \neq y2\}$›
to ‹*lift-invariant* (*Fst* $\otimes_r$ *Fst*; *Snd* $\otimes_r$ *Snd*) (*ket-invariant* $\{((x1, y1), x2, y2).\ x1 \neq x2 \longrightarrow y1 \neq y2\}$)› *DEADID.rel-mono-strong*)
apply (*auto intro*!: *simp*: *lift-inv-prod′ compatible-register-tensor lift-inv-tensor′ lift-Fst-ket-inv lift-Snd-ket-inv*
ket-invariant-tensor case-prod-unfold ket-invariant-inter
simp flip: *ket-invariant-SUP*)[1]
by (*simp add*: *o-apply register-image-lift-invariant compatible-register-tensor register-isometry*)
also have ‹. . . = *lift-invariant* (*Fst* $\otimes_r$ *Fst*; *Snd* $\otimes_r$ *Snd*) (*ket-invariant* $\{((x1, y1), (x2, y2 + h$

x2)) | *x1 y1 x2 y2*. *x1* ≠ *x2* ⟶ *y1* ≠ *y2*})›
  apply (*simp add*: *function-oracle-Snd-ket-invariant*)
  apply (*rule arg-cong*[**where** *f*=‹λ*x*. *lift-invariant* - (*ket-invariant x*)›])
  by (*auto simp add*: *image-iff*)
 also have ‹. . . ≤ *lift-invariant* (*Fst* $\otimes_r$ *Fst*; *Snd* $\otimes_r$ *Snd*) (*ket-invariant* {((*x1*, *y1*), (*x2*, *y2*)). (*x1* ≠ *x2* ⟶ *y1* ≠ *h x2*) ∨ *y2* ≠ *0*})›
 proof −
  have *aux*: ‹*x1* ≠ *x2* ⟹ *h x2* ≠ *y2* ⟹ *y2* + *h x2* ≠ *0*› **for** *x1 x2 y2*
   by (*metis add-right-cancel y-cancel*)
  show *?thesis*
   apply (*rule lift-invariant-mono*, *simp add*: *compatible-register-tensor*)
   apply (*rule ket-invariant-mono*)
   using *aux* **by** *auto*
 qed
 also have ‹. . . = *ket-invariant* {((*x1*, *x2*), (*y1*, *y2*)). (*x1* ≠ *x2* ⟶ *y1* ≠ *h x2*) ∨ *y2* ≠ *0*}›
  by (*auto intro*!: *simp*: *lift-inv-prod′ compatible-register-tensor lift-inv-tensor′ lift-Fst-ket-inv lift-Snd-ket-inv*
   ket-invariant-tensor case-prod-unfold ket-invariant-inter
   simp flip: *ket-invariant-SUP*)[*1*]
 finally show *?thesis*
  by −
qed
have *fo-apply1*: ‹(*Fst* $\otimes_r$ *Fst*) (*function-oracle h*)∗ $*_S$ *ket-invariant* {((*x1*, *x2*), (*y1*, *y2*)). *x1* ≠ *x2* ⟶ *y1* = *h x2* ⟶ *y2* ≠ *0*}
  ≤ *ket-invariant* {((*x1*,*x2*), *yy*). (*x1* ≠ *x2* ⟶ *h x1* ≠ *h x2*) ∨ *yy* ≠ (*0*,*0*)}› **for** *h* :: ‹′*x* ⇒ ′*y*›
proof −
 have ‹(*Fst* $\otimes_r$ *Fst*) (*function-oracle h*)∗ $*_S$ *ket-invariant* {((*x1*, *x2*), (*y1*, *y2*)). *x1* ≠ *x2* ⟶ *y1* = *h x2* ⟶ *y2* ≠ *0*}
  = (*Fst* $\otimes_r$ *Fst*) (*function-oracle h*) $*_S$ *ket-invariant* {((*x1*, *x2*), (*y1*, *y2*)). *x1* ≠ *x2* ⟶ *y1* = *h x2* ⟶ *y2* ≠ *0*}›
  by (*simp add*: *uminus-y flip*: *register-adj*)
 also have ‹. . . = *lift-invariant* (*Snd* $\otimes_r$ *Snd*;*Fst* $\otimes_r$ *Fst*) (*Snd* (*function-oracle h*) $*_S$ *ket-invariant* {((*x2*, *y2*), (*x1*, *y1*)). *x1* ≠ *x2* ⟶ *y1* = *h x2* ⟶ *y2* ≠ *0*})›
  apply (*rewrite at* ‹(*Fst* $\otimes_r$ *Fst*)› *to* ‹(*Snd* $\otimes_r$ *Snd*; *Fst* $\otimes_r$ *Fst*) *o Snd*› *DEADID.rel-mono-strong*)
   apply (*simp add*: *register-pair-Snd compatible-register-tensor*)
  apply (*rewrite at* ‹*ket-invariant* {((*x1*, *x2*), (*y1*, *y2*)). *x1* ≠ *x2* ⟶ *y1* = *h x2* ⟶ *y2* ≠ *0*}›
   to ‹*lift-invariant* (*Snd* $\otimes_r$ *Snd*; *Fst* $\otimes_r$ *Fst*) (*ket-invariant* {((*x2*, *y2*), (*x1*, *y1*)). *x1* ≠ *x2* ⟶ *y1* = *h x2* ⟶ *y2* ≠ *0*})› *DEADID.rel-mono-strong*)
   apply (*auto intro*!: *simp*: *lift-inv-prod′ compatible-register-tensor lift-inv-tensor′ lift-Snd-ket-inv lift-Fst-ket-inv*
   ket-invariant-tensor case-prod-unfold ket-invariant-inter
   simp flip: *ket-invariant-SUP*)[*1*]
  by (*simp-all add*: *o-apply register-image-lift-invariant compatible-register-tensor register-isometry*)
 also have ‹. . . = *lift-invariant* (*Snd* $\otimes_r$ *Snd*; *Fst* $\otimes_r$ *Fst*) (*ket-invariant* {((*x2*, *y2*), (*x1*, *y1* + *h x1*)) | *x1 y1 x2 y2*. *x1* ≠ *x2* ⟶ *y1* = *h x2* ⟶ *y2* ≠ *0*})›
  apply (*simp add*: *function-oracle-Snd-ket-invariant*)
  apply (*rule arg-cong*[**where** *f*=‹λ*x*. *lift-invariant* - (*ket-invariant x*)›])
  by (*auto simp add*: *image-iff*)
 also have ‹. . . ≤ *lift-invariant* (*Snd* $\otimes_r$ *Snd*; *Fst* $\otimes_r$ *Fst*) (*ket-invariant* {((*x2*, *y2*), (*x1*, *y1*)). (*x1* ≠ *x2* ⟶ *h x1* ≠ *h x2*) ∨ (*y1*,*y2*) ≠ (*0*,*0*)})›
 proof −
  have *aux*: ‹*y1* + *h x2* = *0* ⟹ *x1* ≠ *x2* ⟹ *h x1* = *h x2* ⟹ *y1* = *h x2*› **for** *y1 x2 x1*
   by (*metis add-right-cancel y-cancel*)
  show *?thesis*
  apply (*rule lift-invariant-mono*, *simp add*: *compatible-register-tensor*)

apply (*rule ket-invariant-mono*)
using *aux* **by** *auto*
qed
also have ‹... = *ket-invariant* $\{((x1,x2),\ yy).\ (x1 \neq x2 \longrightarrow h\ x1 \neq h\ x2) \vee yy \neq (0,0)\}$›
by (*auto intro*!: *simp*: *lift-inv-prod′ compatible-register-tensor lift-inv-tensor′ lift-Snd-ket-inv lift-Fst-ket-inv*
ket-invariant-tensor case-prod-unfold ket-invariant-inter
simp flip: *ket-invariant-SUP*)[*1*]
finally show *?thesis*
by −
qed
from ∗ **have** ‹*dist-inv-avg* (*adv-output-in-xo*; *aux-in-xo*)
(λh. *ket-invariant* $\{((x1,x2),\ (y1,y2)).\ (x1 \neq x2 \longrightarrow y1 \neq h\ x2) \vee y2 \neq 0\}$) *state2* $\leq$ *d*›
apply (*rule le-back-subst-le*)
unfolding *xo-ext-final-state2*[*abs-def*]
apply (*subst dist-inv-avg-apply*[**where** *U*=‹λh. *function-oracle h*› **and** *S*=‹*Snd* $\otimes_r$ *Snd*›])
using *fo-apply2* **by** (*auto intro*!: *dist-inv-avg-mono simp*: *function-oracle-ket-invariant pair-o-tensor simp del*: *o-apply*)
then show *?thesis*
apply (*rule le-back-subst-le*)
unfolding *state2-def*[*abs-def*]
apply (*subst dist-inv-avg-apply*[**where** *U*=‹λh. *function-oracle h*› **and** *S*=‹*Fst* $\otimes_r$ *Fst*›])
using *fo-apply1* **by** (*auto intro*!: *dist-inv-avg-mono simp*: *function-oracle-ket-invariant pair-o-tensor simp del*: *o-apply*)
qed
then have ∗: ‹*dist-inv-avg* (*adv-output-in-xo*; *aux-in-xo*)
(λh. *ket-invariant* $\{((x1,x2),\ yy).\ x1 \neq x2 \longrightarrow h\ x1 \neq h\ x2\}$) *xo-final* $\leq$ *d*›
— In XO-execution, before the adversary's final query, x1,x2 are a collision.
apply (*rule le-back-subst-le*)
apply (*rule ord-le-eq-trans*)
apply (*rule dist-inv-avg-mono*[**where** *I*=‹λh. *ket-invariant* $\{((x1,x2),\ yy).\ (x1 \neq x2 \longrightarrow h\ x1 \neq h\ x2) \vee yy \neq (0,0)\} \sqcap$ *ket-invariant* $\{(xx,yy).\ yy=(0,0)\}$›])
apply (*auto simp*: *ket-invariant-inter*)[*2*]
apply (*rule dist-inv-avg-intersect*)
apply *simp-all*[*2*]
by (*fact xo-final-has-y0*)
then have ‹*dist-inv-avg adv-output-in-xo*
(λh. *ket-invariant* $\{(x1,x2).\ x1 \neq x2 \longrightarrow h\ x1 \neq h\ x2\}$) *xo-final* $\leq$ *d*›
— As before, but with respect to only the adversary output register.
apply (*subst dist-inv-avg-register-rewrite*[**where** *R*=‹(*adv-output-in-xo*; *aux-in-xo*)› **and** *J*=‹λh. *ket-invariant* $\{((x1,x2),yy).\ x1 \neq x2 \longrightarrow h\ x1 \neq h\ x2\}$›])
apply (*simp*, *simp*)
apply (*rewrite at* ‹$\{((x1,x2),yy).\ x1 \neq x2 \longrightarrow h\ x1 \neq h\ x2\}$› **in for** (*h*) *to* ‹$\{(x1,x2).\ x1 \neq x2 \longrightarrow h\ x1 \neq h\ x2\} \times$ *UNIV*› *DEADID.rel-mono-strong*)
apply *fastforce*
by (*simp add*: *lift-inv-prod*)
then have ‹*dist-inv-avg adv-output* (λh. *ket-invariant* $\{(x1,x2).\ x1 \neq x2 \longrightarrow h\ x1 \neq h\ x2\}$) *final* $\leq$ *d*›
— As before, but in the original execution.
by (*simp add*: *xo-final-final*[*abs-def*] *adv-output-in-xo-def dist-inv-avg-Fst-tensor*)
then have ‹$(\sum h \in UNIV.\ \sum (x1,x2)|x1 \neq x2 \wedge h\ x1 = h\ x2.$ *measurement-probability adv-output* (*final h*) $(x1,x2))\ /\ CARD('x \Rightarrow 'y) \leq d^2$›
unfolding *case-prod-unfold prod.collapse*
apply (*subst dist-inv-avg-measurement-probability*)
apply *simp*

```
      apply (rewrite at ‹− {p. fst p ≠ snd p ∧ h (fst p) = h (snd p)}› in ‹λh. ¤› to ‹{p. fst p ≠ snd p
⟶ h (fst p) ≠ h (snd p)}› DEADID.rel-mono-strong)
       apply blast
      by auto
    also have ‹d² ≤ 12 ∗ (q+154)^3 / N›
    proof −
      define r where ‹r = sqrt q›
      have [iff]: ‹r ≥ 0›
        using r-def by force
      have 1: ‹sqrt (r² + 2) ≤ r + 2›
        apply (rule real-le-lsqrt)
        by (simp-all add: power2-sum)
      have ‹N ∗ d² = (10/3 ∗ sqrt (r²+2)^3 + 20)²›
        apply (simp add: d-def power-divide of-nat-add r-def) by argo
      also have ‹. . . ≤ (10/3 ∗ (r+2)^3 + 20)²›
        using 1 by (auto intro!: power-mono add-right-mono mult-left-mono)
      also have ‹. . . ≤ 12 ∗ (r²+154)^3›
      proof −
        define f where ‹f r = 12 ∗ (r²+154)^3 − (10/3 ∗ (r+2)^3 + 20)²› for r :: real
        have fr: ‹f r ≠ 0› if ‹r ≥ 0› for r :: real
          unfolding f-def using that by (rule sturm-calculation)
        have f0: ‹f 0 ≥ 0›
          by (simp add: f-def power2-eq-square)
        have ‹isCont f r› for r
          unfolding f-def
          by (intro continuous-intros)
        have ‹f r ≥ 0› if ‹r ≥ 0› for r :: real
        proof (rule ccontr)
          assume ‹¬ 0 ≤ f r›
          then have ‹∃x≥0. x ≤ r ∧ f x = 0›
            apply (rule-tac IVT2[where f=f and a=0 and b=r and y=0])
            by (auto intro!: ‹isCont f -› simp: f0 that)
          then show False
            using fr by blast
        qed
        then show ?thesis
          by (simp add: f-def)
      qed
      finally show ?thesis
        apply (rule-tac mult-left-le-imp-le[where c=‹real N›])
        using Nneq0 r-def by force+
    qed
    finally show ?thesis
      by (simp add: final-def q-def)
qed

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.