Quantum and Classical Registers*

Dominique Unruh

March 17, 2025

Abstract

A formalization of the theory of quantum and classical registers as developed by Unruh [Unr24]. In a nutshell, a register refers to a part of a larger memory or system that can be accessed independently. Registers can be constructed from other registers and several (compatible) registers can be composed. For more details, see [Unr24]. This formalization develops both the generic theory of registers as well as specific instantiations for classical and quantum registers.

Note: This document assumes familiarity with the theoretical background developed in [Unr24]. [Unr24] also describes this formalization and mentions some of the design choices and challenges.

Some of the theories are autogenerated (*Laws_Classical*, *Laws_Quantum*, *Laws_Complement_Quantum*). Use the Python script *instantiate_laws.py* to recreate them after changing any of the theories starting with *Laws* or *Axioms*. See [Unr24] for an explanation of this mechanism and the reasons for it.

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 Centre of Exellence in IT (EXCITE) funded by ERDF.

1 Axioms of registers

theory *Axioms*
imports *Main*
begin

class *domain*
instance *prod* :: (*domain*,*domain*) *domain*
by *intro-classes*

typedecl *'a update*
axiomatization *comp-update* :: *'a*::*domain update* ⇒ *'a update* ⇒ *'a update* **where**
comp-update-assoc: *comp-update* (*comp-update a b*) *c* = *comp-update a* (*comp-update b c*)
axiomatization *id-update* :: *'a*::*domain update* **where**
id-update-left: *comp-update id-update a* = *a* **and**
id-update-right: *comp-update a id-update* = *a*

axiomatization *preregister* :: ‹(*'a*::*domain update* ⇒ *'b*::*domain update*) ⇒ *bool*›
axiomatization where
comp-preregister: *preregister F* ⟹ *preregister G* ⟹ *preregister* (*G* ∘ *F*) **and**
id-preregister: ‹*preregister id*›
for *F* :: ‹*'a*::*domain update* ⇒ *'b*::*domain update*› **and** *G* :: ‹*'b update* ⇒ *'c*::*domain update*›

axiomatization where
preregister-mult-right: ‹*preregister* (λ*a*. *comp-update a z*)› **and**
preregister-mult-left: ‹*preregister* (λ*a*. *comp-update z a*)›
for *z* :: *'a*::*domain update*

axiomatization *tensor-update* :: ‹*'a*::*domain update* ⇒ *'b*::*domain update* ⇒ (*'a*×*'b*) *update*›
where *tensor-extensionality*: *preregister F* ⟹ *preregister G* ⟹ (⋀*a b*. *F* (*tensor-update a b*) = *G* (*tensor-update a b*)) ⟹ *F* = *G*
for *F G* :: ‹(*'a*×*'b*) *update* ⇒ *'c*::*domain update*›

axiomatization where *tensor-update-mult*: ‹*comp-update* (*tensor-update a c*) (*tensor-update b d*) = *tensor-update* (*comp-update a b*) (*comp-update c d*)›
for *a b* :: ‹*'a*::*domain update*› **and** *c d* :: ‹*'b*::*domain update*›

axiomatization *register* :: ‹(*'a update* ⇒ *'b update*) ⇒ *bool*›
axiomatization where
register-preregister: *register F* ⟹ *preregister F* **and**
register-comp: *register F* ⟹ *register G* ⟹ *register* (*G* ∘ *F*) **and**
register-mult: *register F* ⟹ *comp-update* (*F a*) (*F b*) = *F* (*comp-update a b*) **and**
register-of-id: ‹*register F* ⟹ *F id-update* = *id-update*› **and**
register-id: ‹*register* (*id* :: *'a update* ⇒ *'a update*)›
for *F* :: *'a*::*domain update* ⇒ *'b*::*domain update* **and** *G* :: *'b update* ⇒ *'c*::*domain update*

axiomatization where *register-tensor-left*: ‹*register* (λ*a*. *tensor-update a id-update*)›
axiomatization where *register-tensor-right*: ‹*register* (λ*a*. *tensor-update id-update a*)›

axiomatization *register-pair* ::
‹(*'a*::*domain update* ⇒ *'c*::*domain update*) ⇒ (*'b*::*domain update* ⇒ *'c update*)
⇒ ((*'a*×*'b*) *update* ⇒ *'c update*)› **where**
register-pair-is-register: ‹*register F* ⟹ *register G* ⟹ (⋀*a b*. *comp-update* (*F a*) (*G b*) = *comp-update* (*G b*) (*F a*))
⟹ *register* (*register-pair F G*)› **and**
register-pair-apply: ‹*register F* ⟹ *register G* ⟹ (⋀*a b*. *comp-update* (*F a*) (*G b*) = *comp-update* (*G b*) (*F a*))
⟹ (*register-pair F G*) (*tensor-update a b*) = *comp-update* (*F a*) (*G b*)›

end

2 Generic laws about registers

theory *Laws*
imports *Axioms*
begin

This notation is only used inside this file

notation *comp-update* (**infixl** $*_u$ *55*)
notation *tensor-update* (**infixr** $\otimes_u$ *70*)
notation *register-pair* ('(-;-'))

2.1 Elementary facts

declare *id-preregister*[*simp*]
declare *id-update-left*[*simp*]
declare *id-update-right*[*simp*]
declare *register-preregister*[*simp*]
declare *register-comp*[*simp*]
declare *register-of-id*[*simp*]
declare *register-tensor-left*[*simp*]
declare *register-tensor-right*[*simp*]
declare *preregister-mult-right*[*simp*]
declare *preregister-mult-left*[*simp*]
declare *register-id*[*simp*]

2.2 Preregisters

lemma *preregister-tensor-left*[*simp*]: ‹*preregister* (λb::'*b*::*domain update*. *tensor-update a b*)›
for *a* :: ‹'*a*::*domain update*›
proof −
have ‹*preregister* (($\lambda b1$::('*a*×'*b*) *update*. (*a* $\otimes_u$ *id-update*) $*_u$ *b1*) *o* (λb. *tensor-update id-update b*))›
by (*rule comp-preregister*; *simp*)
then show *?thesis*
by (*simp add*: *o-def tensor-update-mult*)
qed

lemma *preregister-tensor-right*[*simp*]: ‹*preregister* (λa::'*a*::*domain update*. *tensor-update a b*)›
for *b* :: ‹'*b*::*domain update*›
proof −
have ‹*preregister* (($\lambda a1$::('*a*×'*b*) *update*. (*id-update* $\otimes_u$ *b*) $*_u$ *a1*) *o* (λa. *tensor-update a id-update*))›
by (*rule comp-preregister*, *simp-all*)
then show *?thesis*
by (*simp add*: *o-def tensor-update-mult*)
qed

2.3 Registers

lemma *id-update-tensor-register*[*simp*]:
assumes ‹*register F*›
shows ‹*register* (λa::'*a*::*domain update*. *id-update* $\otimes_u$ *F a*)›
using *assms* **apply** (*rule register-comp*[*unfolded o-def*])
by *simp*

lemma *register-tensor-id-update*[*simp*]:
assumes ‹*register F*›
shows ‹*register* (λa::'*a*::*domain update*. *F a* $\otimes_u$ *id-update*)›
using *assms* **apply** (*rule register-comp*[*unfolded o-def*])
by *simp*

2.4 Tensor product of registers

definition *register-tensor* (**infixr** $\otimes_r$ *70*) **where**
register-tensor F G = *register-pair* (λa. *tensor-update* (*F a*) *id-update*) (λb. *tensor-update id-update* (*G b*))

lemma *register-tensor-is-register*:
fixes $F :: 'a::domain\ update \Rightarrow 'b::domain\ update$ **and** $G :: 'c::domain\ update \Rightarrow 'd::domain\ update$
shows *register* $F \Longrightarrow$ *register* $G \Longrightarrow$ *register* $(F \otimes_r G)$
unfolding *register-tensor-def*
apply (*rule register-pair-is-register*)
by (*simp-all add*: *tensor-update-mult*)

lemma *register-tensor-apply*[*simp*]:
fixes $F :: 'a::domain\ update \Rightarrow 'b::domain\ update$ **and** $G :: 'c::domain\ update \Rightarrow 'd::domain\ update$
assumes ‹*register F*› **and** ‹*register G*›
shows $(F \otimes_r G)\ (a \otimes_u b) = F\ a \otimes_u G\ b$
unfolding *register-tensor-def*
apply (*subst register-pair-apply*)
unfolding *register-tensor-def*
by (*simp-all add*: *assms tensor-update-mult*)

definition *separating* (-::'*b*::*domain itself*) *A* ⟷
$(\forall F\ G :: 'a::domain\ update \Rightarrow 'b\ update.\ preregister\ F \longrightarrow preregister\ G \longrightarrow (\forall x{\in}A.\ F\ x = G\ x) \longrightarrow F = G)$

lemma *separating-UNIV*[*simp*]: ‹*separating TYPE*(-) *UNIV*›
unfolding *separating-def* **by** *auto*

lemma *separating-mono*: ‹$A \subseteq B \Longrightarrow$ *separating TYPE*('*a*::*domain*) *A* $\Longrightarrow$ *separating TYPE*('*a*) *B*›
unfolding *separating-def* **by** (*meson in-mono*)

lemma *register-eqI*: ‹*separating TYPE*('*b*::*domain*) *A* $\Longrightarrow$ *preregister F* $\Longrightarrow$ *preregister G* $\Longrightarrow$ $(\bigwedge x.\ x{\in}A \Longrightarrow F\ x = G\ x) \Longrightarrow F = (G::- \Rightarrow 'b\ update)$›
unfolding *separating-def* **by** *auto*

lemma *separating-tensor*:
fixes *A* :: ‹'*a*::*domain update set*› **and** *B* :: ‹'*b*::*domain update set*›
assumes [*simp*]: ‹*separating TYPE*('*c*::*domain*) *A*›
assumes [*simp*]: ‹*separating TYPE*('*c*) *B*›
shows ‹*separating TYPE*('*c*) $\{a \otimes_u b \mid a\ b.\ a{\in}A \wedge b{\in}B\}$›
proof (*unfold separating-def*, *intro allI impI*)
fix *F G* :: ‹('*a*×'*b*) *update* ⇒ '*c update*›
assume [*simp*]: ‹*preregister F*› ‹*preregister G*›
have [*simp*]: ‹*preregister* $(\lambda x.\ F\ (a \otimes_u x))$› **for** *a*
using - ‹*preregister F*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*preregister* $(\lambda x.\ G\ (a \otimes_u x))$› **for** *a*
using - ‹*preregister G*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*preregister* $(\lambda x.\ F\ (x \otimes_u b))$› **for** *b*
using - ‹*preregister F*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*preregister* $(\lambda x.\ G\ (x \otimes_u b))$› **for** *b*
using - ‹*preregister G*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*

assume ‹$\forall x{\in}\{a \otimes_u b \mid a\ b.\ a{\in}A \wedge b{\in}B\}.\ F\ x = G\ x$›
then have *EQ*: ‹$F\ (a \otimes_u b) = G\ (a \otimes_u b)$› **if** ‹$a \in A$› **and** ‹$b \in B$› **for** *a b*
using *that* **by** *auto*
then have ‹$F\ (a \otimes_u b) = G\ (a \otimes_u b)$› **if** ‹$a \in A$› **for** *a b*
apply (*rule register-eqI*[**where** *A*=*B*, *THEN fun-cong*, **where** *x*=*b*, *rotated* −*1*])
using *that* **by** *auto*
then have ‹$F\ (a \otimes_u b) = G\ (a \otimes_u b)$› **for** *a b*
apply (*rule register-eqI*[**where** *A*=*A*, *THEN fun-cong*, **where** *x*=*a*, *rotated* −*1*])
by *auto*
then show *F* = *G*
apply (*rule tensor-extensionality*[*rotated* −*1*])
by *auto*
qed

lemma *register-tensor-distrib*:
assumes [*simp*]: ‹*register F*› ‹*register G*› ‹*register H*› ‹*register L*›
shows ‹(*F* ⊗*r* *G*) *o* (*H* ⊗*r* *L*) = (*F o H*) ⊗*r* (*G o L*)›
apply (*rule tensor-extensionality*)
by (*auto intro*!: *register-comp register-preregister register-tensor-is-register*)

The following is easier to apply using the *rule*-method than *separating-tensor*

lemma *separating-tensor′*:
fixes *A* :: ‹'*a*::*domain update set*› **and** *B* :: ‹'*b*::*domain update set*›
assumes ‹*separating TYPE*('*c*::*domain*) *A*›
assumes ‹*separating TYPE*('*c*) *B*›
assumes ‹*C* = {*a* ⊗*u* *b* | *a b*. *a*∈*A* ∧ *b*∈*B*}›
shows ‹*separating TYPE*('*c*) *C*›
using *assms*
by (*simp add*: *separating-tensor*)

lemma *tensor-extensionality3*:
fixes *F G* :: ‹('*a*::*domain*×'*b*::*domain*×'*c*::*domain*) *update* ⇒ '*d*::*domain update*›
assumes [*simp*]: ‹*register F*› ‹*register G*›
assumes ⋀*f g h*. *F* (*f* ⊗*u* *g* ⊗*u* *h*) = *G* (*f* ⊗*u* *g* ⊗*u* *h*)
shows *F* = *G*
proof (*rule register-eqI*[**where** *A*=‹{*a*⊗*u b*⊗*u c*| *a b c*. *True*}›])
have ‹*separating TYPE*('*d*) {*b* ⊗*u* *c* |*b c*. *True*}›
apply (*rule separating-tensor′*[**where** *A*=*UNIV* **and** *B*=*UNIV*])
by *auto*
then show ‹*separating TYPE*('*d*) {*a* ⊗*u* *b* ⊗*u* *c* |*a b c*. *True*}›
apply (*rule-tac separating-tensor′*[**where** *A*=*UNIV* **and** *B*=‹{*b*⊗*u c*| *b c*. *True*}›])
by *auto*
show ‹*preregister F*› ‹*preregister G*› **by** *auto*
show ‹*x* ∈ {*a* ⊗*u* *b* ⊗*u* *c* |*a b c*. *True*} ⟹ *F x* = *G x*› **for** *x*
using *assms*(*3*) **by** *auto*
qed

lemma *tensor-extensionality3′*:
fixes *F G* :: ‹(('*a*::*domain*×'*b*::*domain*)×'*c*::*domain*) *update* ⇒ '*d*::*domain update*›
assumes [*simp*]: ‹*register F*› ‹*register G*›
assumes ⋀*f g h*. *F* ((*f* ⊗*u* *g*) ⊗*u* *h*) = *G* ((*f* ⊗*u* *g*) ⊗*u* *h*)
shows *F* = *G*
proof (*rule register-eqI*[**where** *A*=‹{(*a*⊗*u b*)⊗*u c*| *a b c*. *True*}›])
have ‹*separating TYPE*('*d*) {*a* ⊗*u* *b* | *a b*. *True*}›
apply (*rule separating-tensor′*[**where** *A*=*UNIV* **and** *B*=*UNIV*])
by *auto*
then show ‹*separating TYPE*('*d*) {(*a* ⊗*u* *b*) ⊗*u* *c* |*a b c*. *True*}›
apply (*rule-tac separating-tensor′*[**where** *B*=*UNIV* **and** *A*=‹{*a*⊗*u b*| *a b*. *True*}›])
by *auto*
show ‹*preregister F*› ‹*preregister G*› **by** *auto*
show ‹*x* ∈ {(*a* ⊗*u* *b*) ⊗*u* *c* |*a b c*. *True*} ⟹ *F x* = *G x*› **for** *x*
using *assms*(*3*) **by** *auto*
qed

lemma *register-tensor-id*[*simp*]: ‹*id* ⊗*r* *id* = *id*›
apply (*rule tensor-extensionality*)
by (*auto simp add*: *register-tensor-is-register*)

2.5 Pairs and compatibility

definition *compatible* :: ‹('*a*::*domain update* ⇒ '*c*::*domain update*)
⇒ ('*b*::*domain update* ⇒ '*c update*) ⇒ *bool*› **where**
‹*compatible F G* ⟷ *register F* ∧ *register G* ∧ (∀ *a b*. *F a* ∗*u* *G b* = *G b* ∗*u* *F a*)›

lemma *compatibleI*:
assumes *register F* **and** *register G*

assumes ‹$\bigwedge a\ b.\ (F\ a) *_u (G\ b) = (G\ b) *_u (F\ a)$›
shows *compatible F G*
using *assms* **unfolding** *compatible-def* **by** *simp*

lemma *swap-registers*:
assumes *compatible R S*
shows $R\ a *_u S\ b = S\ b *_u R\ a$
using *assms* **unfolding** *compatible-def* **by** *metis*

lemma *compatible-sym*: *compatible x y* $\Longrightarrow$ *compatible y x*
by (*simp add*: *compatible-def*)

lemma *pair-is-register*[*simp*]:
assumes *compatible F G*
shows *register* (*F*; *G*)
by (*metis assms compatible-def register-pair-is-register*)

lemma *register-pair-apply*:
assumes ‹*compatible F G*›
shows ‹(*F*; *G*) ($a \otimes_u b$) = (*F a*) $*_u$ (*G b*)›
apply (*rule register-pair-apply*)
using *assms* **unfolding** *compatible-def* **by** *metis*+

lemma *register-pair-apply′*:
assumes ‹*compatible F G*›
shows ‹(*F*; *G*) ($a \otimes_u b$) = (*G b*) $*_u$ (*F a*)›
apply (*subst register-pair-apply*)
using *assms* **by** (*auto simp*: *compatible-def intro*: *register-preregister*)

lemma *compatible-comp-left*[*simp*]: *compatible F G* $\Longrightarrow$ *register H* $\Longrightarrow$ *compatible* (*F* $\circ$ *H*) *G*
by (*simp add*: *compatible-def*)

lemma *compatible-comp-right*[*simp*]: *compatible F G* $\Longrightarrow$ *register H* $\Longrightarrow$ *compatible F* (*G* $\circ$ *H*)
by (*simp add*: *compatible-def*)

lemma *compatible-comp-inner*[*simp*]:
compatible F G $\Longrightarrow$ *register H* $\Longrightarrow$ *compatible* (*H* $\circ$ *F*) (*H* $\circ$ *G*)
by (*smt* (*verit*, *best*) *comp-apply compatible-def register-comp register-mult*)

lemma *compatible-register1*: ‹*compatible F G* $\Longrightarrow$ *register F*›
by (*simp add*: *compatible-def*)
lemma *compatible-register2*: ‹*compatible F G* $\Longrightarrow$ *register G*›
by (*simp add*: *compatible-def*)

lemma *pair-o-tensor*:
assumes *compatible A B* **and** [*simp*]: ‹*register C*› **and** [*simp*]: ‹*register D*›
shows (*A*; *B*) *o* (*C* $\otimes_r$ *D*) = (*A o C*; *B o D*)
apply (*rule tensor-extensionality*)
using *assms* **by** (*simp-all add*: *register-tensor-is-register register-pair-apply comp-preregister*)

lemma *compatible-tensor-id-update-left*[*simp*]:
fixes *F* :: *′a*::*domain update* $\Rightarrow$ *′c*::*domain update* **and** *G* :: *′b*::*domain update* $\Rightarrow$ *′c*::*domain update*
assumes *compatible F G*
shows *compatible* ($\lambda a.$ *id-update* $\otimes_u$ *F a*) ($\lambda a.$ *id-update* $\otimes_u$ *G a*)
using *assms* **apply** (*rule compatible-comp-inner*[*unfolded o-def*])
by *simp*

lemma *compatible-tensor-id-update-right*[*simp*]:
fixes *F* :: *′a*::*domain update* $\Rightarrow$ *′c*::*domain update* **and** *G* :: *′b*::*domain update* $\Rightarrow$ *′c*::*domain update*
assumes *compatible F G*
shows *compatible* ($\lambda a.$ *F a* $\otimes_u$ *id-update*) ($\lambda a.$ *G a* $\otimes_u$ *id-update*)

using *assms* **apply** (*rule compatible-comp-inner*[*unfolded o-def*])
by *simp*

lemma *compatible-tensor-id-update-rl*[*simp*]:
assumes *register F* **and** *register G*
shows *compatible* (λ*a*. *F a* ⊗*u id-update*) (λ*a*. *id-update* ⊗*u G a*)
apply (*rule compatibleI*)
using *assms* **by** (*auto simp*: *tensor-update-mult*)

lemma *compatible-tensor-id-update-lr*[*simp*]:
assumes *register F* **and** *register G*
shows *compatible* (λ*a*. *id-update* ⊗*u F a*) (λ*a*. *G a* ⊗*u id-update*)
apply (*rule compatibleI*)
using *assms* **by** (*auto simp*: *tensor-update-mult*)

lemma *register-comp-pair*:
assumes [*simp*]: ‹*register F*› **and** [*simp*]: ‹*compatible G H*›
shows (*F o G*; *F o H*) = *F o* (*G*; *H*)
proof (*rule tensor-extensionality*)
show ‹*preregister* (*F* ∘ *G*;*F* ∘ *H*)› **and** ‹*preregister* (*F* ∘ (*G*;*H*))›
by *simp-all*

have [*simp*]: ‹*compatible* (*F o G*) (*F o H*)›
apply (*rule compatible-comp-inner*, *simp*)
by *simp*
then have [*simp*]: ‹*register* (*F* ∘ *G*)› ‹*register* (*F* ∘ *H*)›
unfolding *compatible-def* **by** *auto*
from *assms* **have** [*simp*]: ‹*register G*› ‹*register H*›
unfolding *compatible-def* **by** *auto*
fix *a b*
show ‹(*F* ∘ *G*;*F* ∘ *H*) (*a* ⊗*u b*) = (*F* ∘ (*G*;*H*)) (*a* ⊗*u b*)›
by (*auto simp*: *register-pair-apply register-mult tensor-update-mult*)
qed

lemma *swap-registers-left*:
assumes *compatible R S*
shows *R a* ∗*u S b* ∗*u c* = *S b* ∗*u R a* ∗*u c*
using *assms* **unfolding** *compatible-def* **by** *metis*

lemma *swap-registers-right*:
assumes *compatible R S*
shows *c* ∗*u R a* ∗*u S b* = *c* ∗*u S b* ∗*u R a*
by (*metis assms comp-update-assoc compatible-def*)

lemmas *compatible-ac-rules* = *swap-registers comp-update-assoc*[*symmetric*] *swap-registers-right*

2.6 Fst and Snd

definition *Fst* **where** ‹*Fst a* = *a* ⊗*u id-update*›
definition *Snd* **where** ‹*Snd a* = *id-update* ⊗*u a*›

lemma *register-Fst*[*simp*]: ‹*register Fst*›
unfolding *Fst-def* **by** (*rule register-tensor-left*)

lemma *register-Snd*[*simp*]: ‹*register Snd*›
unfolding *Snd-def* **by** (*rule register-tensor-right*)

lemma *compatible-Fst-Snd*[*simp*]: ‹*compatible Fst Snd*›
apply (*rule compatibleI*, *simp*, *simp*)
by (*simp add*: *Fst-def Snd-def tensor-update-mult*)

lemmas *compatible-Snd-Fst*[*simp*] = *compatible-Fst-Snd*[*THEN compatible-sym*]

definition ‹*swap* = (*Snd*; *Fst*)›

lemma *swap-apply*[*simp*]: *swap* (*a* ⊗*u* *b*) = (*b* ⊗*u* *a*)
unfolding *swap-def*
by (*simp add*: *Axioms.register-pair-apply Fst-def Snd-def tensor-update-mult*)

lemma *swap-o-Fst*: *swap o Fst* = *Snd*
by (*auto simp add*: *Fst-def Snd-def*)
lemma *swap-o-Snd*: *swap o Snd* = *Fst*
by (*auto simp add*: *Fst-def Snd-def*)

lemma *register-swap*[*simp*]: ‹*register swap*›
by (*simp add*: *swap-def*)

lemma *pair-Fst-Snd*: ‹(*Fst*; *Snd*) = *id*›
apply (*rule tensor-extensionality*)
by (*simp-all add*: *register-pair-apply Fst-def Snd-def tensor-update-mult*)

lemma *swap-o-swap*[*simp*]: ‹*swap o swap* = *id*›
by (*metis swap-def compatible-Snd-Fst pair-Fst-Snd register-comp-pair register-swap swap-o-Fst swap-o-Snd*)

lemma *swap-swap*[*simp*]: ‹*swap* (*swap x*) = *x*›
by (*simp add*: *pointfree-idE*)

lemma *inv-swap*[*simp*]: ‹*inv swap* = *swap*›
by (*meson inv-unique-comp swap-o-swap*)

lemma *register-pair-Fst*:
assumes ‹*compatible F G*›
shows ‹(*F*;*G*) *o Fst* = *F*›
using *assms* **by** (*auto intro*!: *ext simp*: *Fst-def register-pair-apply compatible-register2*)

lemma *register-pair-Snd*:
assumes ‹*compatible F G*›
shows ‹(*F*;*G*) *o Snd* = *G*›
using *assms* **by** (*auto intro*!: *ext simp*: *Snd-def register-pair-apply compatible-register1*)

lemma *register-Fst-register-Snd*[*simp*]:
assumes ‹*register F*›
shows ‹(*F o Fst*; *F o Snd*) = *F*›
apply (*rule tensor-extensionality*)
using *assms* **by** (*auto simp*: *register-pair-apply Fst-def Snd-def register-mult tensor-update-mult*)

lemma *register-Snd-register-Fst*[*simp*]:
assumes ‹*register F*›
shows ‹(*F o Snd*; *F o Fst*) = *F o swap*›
apply (*rule tensor-extensionality*)
using *assms* **by** (*auto simp*: *register-pair-apply Fst-def Snd-def register-mult tensor-update-mult*)

lemma *compatible3*[*simp*]:
assumes [*simp*]: *compatible F G* **and** *compatible G H* **and** *compatible F H*
shows *compatible* (*F*; *G*) *H*
proof (*rule compatibleI*)
have [*simp*]: ‹*register F*› ‹*register G*› ‹*register H*›
using *assms compatible-def* **by** *auto*
then have [*simp*]: ‹*preregister F*› ‹*preregister G*› ‹*preregister H*›
using *register-preregister* **by** *blast*+
have [*simp*]: ‹*preregister* (λ*a*. (*F*;*G*) *a* ∗*u* *z*)› **for** *z*
apply (*rule comp-preregister*[*unfolded o-def*, *of* ‹(*F*;*G*)›])
by *simp-all*
have [*simp*]: ‹*preregister* (λ*a*. *z* ∗*u* (*F*;*G*) *a*)› **for** *z*
apply (*rule comp-preregister*[*unfolded o-def*, *of* ‹(*F*;*G*)›])

```
    by simp-all
  have (F; G) (f ⊗u g) ∗u H h = H h ∗u (F; G) (f ⊗u g) for f g h
  proof −
    have FH: F f ∗u H h = H h ∗u F f
      using assms compatible-def by metis
    have GH: G g ∗u H h = H h ∗u G g
      using assms compatible-def by metis
    have ‹(F; G) (f ⊗u g) ∗u (H h) = F f ∗u G g ∗u H h›
      using ‹compatible F G› by (subst register-pair-apply, auto)
    also have ‹... = H h ∗u F f ∗u G g›
      using FH GH by (metis comp-update-assoc)
    also have ‹... = H h ∗u (F; G) (f ⊗u g)›
      using ‹compatible F G› by (subst register-pair-apply, auto simp: comp-update-assoc)
    finally show ?thesis
      by −
  qed
  then show (F; G) fg ∗u (H h) = (H h) ∗u (F; G) fg for fg h
    apply (rule-tac tensor-extensionality[THEN fun-cong])
    by auto
  show register H and  register (F; G)
    by simp-all
qed

lemma compatible3'[simp]:
  assumes compatible F G and compatible G H and compatible F H
  shows compatible F (G; H)
  apply (rule compatible-sym)
  apply (rule compatible3)
  using assms by (auto simp: compatible-sym)

lemma pair-o-swap[simp]:
  assumes [simp]: compatible A B
  shows (A; B) o swap = (B; A)
proof (rule tensor-extensionality)
  have [simp]: preregister A preregister B
    apply (metis (no-types, opaque-lifting) assms compatible-register1 register-preregister)
    by (metis (full-types) assms compatible-register2 register-preregister)
  then show ‹preregister ((A; B) ∘ swap)›
    by simp
  show ‹preregister (B; A)›
    by (metis (no-types, lifting) assms compatible-sym register-preregister pair-is-register)
  show ‹((A; B) ∘ swap) (a ⊗u b) = (B; A) (a ⊗u b)› for a b

    apply (simp only: o-def swap-apply)
    apply (subst register-pair-apply, simp)
    apply (subst register-pair-apply, simp add: compatible-sym)
    by (metis (no-types, lifting) assms compatible-def)
qed
```

2.7 Compatibility of register tensor products

```
lemma compatible-register-tensor:
  fixes F :: ‹'a::domain update ⇒ 'e::domain update› and G :: ‹'b::domain update ⇒ 'f::domain update›
    and F' :: ‹'c::domain update ⇒ 'e update› and G' :: ‹'d::domain update ⇒ 'f update›
  assumes [simp]: ‹compatible F F'›
  assumes [simp]: ‹compatible G G'›
  shows ‹compatible (F ⊗r G) (F' ⊗r G')›
proof −
  note [intro!] =
    comp-preregister[OF - preregister-mult-right, unfolded o-def]
    comp-preregister[OF - preregister-mult-left, unfolded o-def]
    comp-preregister
    register-tensor-is-register
```

have [*simp*]: ‹*register F*› ‹*register G*› ‹*register F'*› ‹*register G'*›
using *assms compatible-def* **by** *blast+*
have [*simp*]: ‹*register* (*F* ⊗*r* *G*)› ‹*register* (*F'* ⊗*r* *G'*)›
by (*auto simp add*: *register-tensor-def*)
have [*simp*]: ‹*register* (*F*;*F'*)› ‹*register* (*G*;*G'*)›
by *auto*
define *reorder* :: ‹(('*a*×'*b*) × ('*c*×'*d*)) *update* ⇒ (('*a*×'*c*) × ('*b*×'*d*)) *update*›
where ‹*reorder* = ((*Fst o Fst*; *Snd o Fst*); (*Fst o Snd*; *Snd o Snd*))›
have [*simp*]: ‹*preregister reorder*›
by (*auto simp*: *reorder-def*)
have [*simp*]: ‹*reorder* ((*a* ⊗*u* *b*) ⊗*u* (*c* ⊗*u* *d*)) = ((*a* ⊗*u* *c*) ⊗*u* (*b* ⊗*u* *d*))› **for** *a b c d*
apply (*simp add*: *reorder-def register-pair-apply*)
by (*simp add*: *Fst-def Snd-def tensor-update-mult*)
define Φ **where** ‹Φ *c d* = ((*F*;*F'*) ⊗*r* (*G*;*G'*)) *o reorder o* (λσ. σ ⊗*u* (*c* ⊗*u* *d*))› **for** *c d*
have [*simp*]: ‹*preregister* (Φ *c d*)› **for** *c d*
unfolding Φ-*def*
by (*auto intro*: *register-preregister*)
have ‹Φ *c d* (*a* ⊗*u* *b*) = (*F* ⊗*r* *G*) (*a* ⊗*u* *b*) ∗*u* (*F'* ⊗*r* *G'*) (*c* ⊗*u* *d*)› **for** *a b c d*
unfolding Φ-*def* **by** (*auto simp*: *register-pair-apply tensor-update-mult*)
then have Φ*1*: ‹Φ *c d* σ = (*F* ⊗*r* *G*) σ ∗*u* (*F'* ⊗*r* *G'*) (*c* ⊗*u* *d*)› **for** *c d* σ
apply (*rule-tac fun-cong*[*of - - σ*])
apply (*rule tensor-extensionality*)
by *auto*
have ‹Φ *c d* (*a* ⊗*u* *b*) = (*F'* ⊗*r* *G'*) (*c* ⊗*u* *d*) ∗*u* (*F* ⊗*r* *G*) (*a* ⊗*u* *b*)› **for** *a b c d*
using *assms*
unfolding Φ-*def compatible-def* **by** (*auto simp*: *register-pair-apply tensor-update-mult*)
then have Φ*2*: ‹Φ *c d* σ = (*F'* ⊗*r* *G'*) (*c* ⊗*u* *d*) ∗*u* (*F* ⊗*r* *G*) σ› **for** *c d* σ
apply (*rule-tac fun-cong*[*of - - σ*])
apply (*rule tensor-extensionality*)
by *auto*
from Φ*1* Φ*2* **have** ‹(*F* ⊗*r* *G*) σ ∗*u* (*F'* ⊗*r* *G'*) τ = (*F'* ⊗*r* *G'*) τ ∗*u* (*F* ⊗*r* *G*) σ› **for** τ σ
apply (*rule-tac fun-cong*[*of - - τ*])
apply (*rule tensor-extensionality*)
by *auto*
then show *?thesis*
apply (*rule compatibleI*[*rotated −1*])
by *auto*
qed

2.8 Associativity of the tensor product

definition *assoc* :: ‹(('*a*::*domain*×'*b*::*domain*)×'*c*::*domain*) *update* ⇒ ('*a*×('*b*×'*c*)) *update*› **where**
‹*assoc* = ((*Fst*; *Snd o Fst*); *Snd o Snd*)›

lemma *assoc-is-hom*[*simp*]: ‹*preregister assoc*›
by (*auto simp*: *assoc-def*)

lemma *assoc-apply*[*simp*]: ‹*assoc* ((*a* ⊗*u* *b*) ⊗*u* *c*) = (*a* ⊗*u* (*b* ⊗*u* *c*))›
by (*auto simp*: *assoc-def register-pair-apply Fst-def Snd-def tensor-update-mult*)

definition *assoc'* :: ‹('*a*×('*b*×'*c*)) *update* ⇒ (('*a*::*domain*×'*b*::*domain*)×'*c*::*domain*) *update*› **where**
‹*assoc'* = (*Fst o Fst*; (*Fst o Snd*; *Snd*))›

lemma *assoc'-is-hom*[*simp*]: ‹*preregister assoc'*›
by (*auto simp*: *assoc'-def*)

lemma *assoc'-apply*[*simp*]: ‹*assoc'* (*a* ⊗*u* (*b* ⊗*u* *c*)) = ((*a* ⊗*u* *b*) ⊗*u* *c*)›
by (*auto simp*: *assoc'-def register-pair-apply Fst-def Snd-def tensor-update-mult*)

lemma *register-assoc*[*simp*]: ‹*register assoc*›
unfolding *assoc-def*
by *force*

lemma *register-assoc'*[*simp*]: ‹*register assoc'*›
unfolding *assoc'-def*
by *force*

lemma *pair-o-assoc*[*simp*]:
assumes [*simp*]: ‹*compatible F G*› ‹*compatible G H*› ‹*compatible F H*›
shows ‹(*F*; (*G*; *H*)) ∘ *assoc* = ((*F*; *G*); *H*)›
proof (*rule tensor-extensionality3'*)
show ‹*register* ((*F*; (*G*; *H*)) ∘ *assoc*)›
by *simp*
show ‹*register* ((*F*; *G*); *H*)›
by *simp*
show ‹((*F*; (*G*; *H*)) ∘ *assoc*) ((*f* ⊗*u* *g*) ⊗*u* *h*) = ((*F*; *G*); *H*) ((*f* ⊗*u* *g*) ⊗*u* *h*)› **for** *f g h*
by (*simp add*: *register-pair-apply assoc-apply comp-update-assoc*)
qed

lemma *pair-o-assoc'*[*simp*]:
assumes [*simp*]: ‹*compatible F G*› ‹*compatible G H*› ‹*compatible F H*›
shows ‹((*F*; *G*); *H*) ∘ *assoc'* = (*F*; (*G*; *H*))›
proof (*rule tensor-extensionality3*)
show ‹*register* (((*F*; *G*); *H*) ∘ *assoc'*)›
by *simp*
show ‹*register* (*F*; (*G*; *H*))›
by *simp*
show ‹(((*F*; *G*); *H*) ∘ *assoc'*) (*f* ⊗*u* *g* ⊗*u* *h*) = (*F*; (*G*; *H*)) (*f* ⊗*u* *g* ⊗*u* *h*)› **for** *f g h*
by (*simp add*: *register-pair-apply assoc'-apply comp-update-assoc*)
qed

lemma *assoc'-o-assoc*[*simp*]: ‹*assoc' o assoc* = *id*›
apply (*rule tensor-extensionality3'*)
by *auto*

lemma *assoc'-assoc*[*simp*]: ‹*assoc'* (*assoc x*) = *x*›
by (*simp add*: *pointfree-idE*)

lemma *assoc-o-assoc'*[*simp*]: ‹*assoc o assoc'* = *id*›
apply (*rule tensor-extensionality3*)
by *auto*

lemma *assoc-assoc'*[*simp*]: ‹*assoc* (*assoc'* *x*) = *x*›
by (*simp add*: *pointfree-idE*)

lemma *inv-assoc*[*simp*]: ‹*inv assoc* = *assoc'*›
using *assoc'-o-assoc assoc-o-assoc' inv-unique-comp* **by** *blast*

lemma *inv-assoc'*[*simp*]: ‹*inv assoc'* = *assoc*›
by (*simp add*: *inv-equality*)

lemma *bij-assoc*[*simp*]: ‹*bij assoc*›
using *assoc'-o-assoc assoc-o-assoc' o-bij* **by** *blast*

lemma *bij-assoc'*[*simp*]: ‹*bij assoc'*›
using *assoc'-o-assoc assoc-o-assoc' o-bij* **by** *blast*

2.9 Iso-registers

definition ‹*iso-register F* ⟷ *register F* ∧ (∃ *G*. *register G* ∧ *F o G* = *id* ∧ *G o F* = *id*)›
for *F* :: ‹-::*domain update* ⇒ -::*domain update*›

lemma *iso-registerI*:
assumes ‹*register F*› ‹*register G*› ‹*F o G* = *id*› ‹*G o F* = *id*›
shows ‹*iso-register F*›
using *assms(1) assms(2) assms(3) assms(4) iso-register-def* **by** *blast*

lemma *iso-register-inv*: ‹*iso-register F* ⟹ *iso-register* (*inv F*)›
by (*metis inv-unique-comp iso-register-def*)

lemma *iso-register-inv-comp1*: ‹*iso-register F* ⟹ *inv F o F* = *id*›
using *inv-unique-comp iso-register-def* **by** *blast*

lemma *iso-register-inv-comp2*: ‹*iso-register F* ⟹ *F o inv F* = *id*›
using *inv-unique-comp iso-register-def* **by** *blast*

lemma *iso-register-id*[*simp*]: ‹*iso-register id*›
by (*simp add*: *iso-register-def*)

lemma *iso-register-is-register*: ‹*iso-register F* ⟹ *register F*›
using *iso-register-def* **by** *blast*

lemma *iso-register-comp*[*simp*]:
assumes [*simp*]: ‹*iso-register F*› ‹*iso-register G*›
shows ‹*iso-register* (*F o G*)›
proof −
from *assms* **obtain** *F′ G′* **where** [*simp*]: ‹*register F′*› ‹*register G′*› ‹*F o F′* = *id*› ‹*F′ o F* = *id*›
‹*G o G′* = *id*› ‹*G′ o G* = *id*›
by (*meson iso-register-def*)
have *1*: ‹*F* ∘ *G* ∘ (*G′* ∘ *F′*) = *id*›
by (*metis* ‹*F* ∘ *F′* = *id*› ‹*G* ∘ *G′* = *id*› *fcomp-assoc fcomp-comp id-fcomp*)
have *2*: ‹*G′* ∘ *F′* ∘ (*F* ∘ *G*) = *id*›
by (*metis* (*no-types*, *lifting*) ‹*F* ∘ *F′* = *id*› ‹*F′* ∘ *F* = *id*› ‹*G′* ∘ *G* = *id*› *fun.map-comp inj-iff inv-unique-comp o-inv-o-cancel*)

show *?thesis*
apply (*rule iso-registerI*[**where** *G*=‹*G′ o F′*›])
using *1 2* **by** (*auto simp*: *register-tensor-is-register iso-register-is-register register-tensor-distrib*)
qed

lemma *iso-register-tensor-is-iso-register*[*simp*]:
assumes [*simp*]: ‹*iso-register F*› ‹*iso-register G*›
shows ‹*iso-register* (*F* ⊗*r G*)›
proof −
from *assms* **obtain** *F′ G′* **where** [*simp*]: ‹*register F′*› ‹*register G′*› ‹*F o F′* = *id*› ‹*F′ o F* = *id*›
‹*G o G′* = *id*› ‹*G′ o G* = *id*›
by (*meson iso-register-def*)
show *?thesis*
apply (*rule iso-registerI*[**where** *G*=‹*F′* ⊗*r G′*›])
by (*auto simp*: *register-tensor-is-register iso-register-is-register register-tensor-distrib*)
qed

lemma *iso-register-bij*: ‹*iso-register F* ⟹ *bij F*›
using *iso-register-def o-bij* **by** *auto*

lemma *inv-register-tensor*[*simp*]:
assumes [*simp*]: ‹*iso-register F*› ‹*iso-register G*›
shows ‹*inv* (*F* ⊗*r G*) = *inv F* ⊗*r inv G*›
apply (*auto intro*!: *inj-imp-inv-eq bij-is-inj iso-register-bij*
simp: *register-tensor-distrib*[*unfolded o-def*, *THEN fun-cong*] *iso-register-is-register*
iso-register-inv bij-is-surj iso-register-bij surj-f-inv-f)
by (*metis eq-id-iff register-tensor-id*)

lemma *iso-register-swap*[*simp*]: ‹*iso-register swap*›
apply (*rule iso-registerI*[*of - swap*])
by *auto*

lemma *iso-register-assoc*[*simp*]: ‹*iso-register assoc*›
apply (*rule iso-registerI*[*of - assoc'*])
by *auto*

lemma *iso-register-assoc'*[*simp*]: ‹*iso-register assoc'*›
apply (*rule iso-registerI*[*of - assoc*])
by *auto*

definition ‹*equivalent-registers F G* ⟷ (*register F* ∧ (∃ *I*. *iso-register I* ∧ *F o I* = *G*))›
for *F G* :: ‹-::*domain update* ⇒ -::*domain update*›

lemma *iso-register-equivalent-id*[*simp*]: ‹*equivalent-registers id F* ⟷ *iso-register F*›
by (*simp add*: *equivalent-registers-def*)

lemma *equivalent-registersI*:
assumes ‹*register F*›
assumes ‹*iso-register I*›
assumes ‹*F o I* = *G*›
shows ‹*equivalent-registers F G*›
using *assms* **unfolding** *equivalent-registers-def* **by** *blast*

lemma *equivalent-registers-refl*: ‹*equivalent-registers F F*› **if** ‹*register F*›
using *that* **by** (*auto intro*!: *exI*[*of - id*] *simp*: *equivalent-registers-def*)

lemma *equivalent-registers-register-left*: ‹*equivalent-registers F G* ⟹ *register F*›
using *equivalent-registers-def* **by** *auto*

lemma *equivalent-registers-register-right*: ‹*register G*› **if** ‹*equivalent-registers F G*›
by (*metis equivalent-registers-def iso-register-def register-comp that*)

lemma *equivalent-registers-sym*:
assumes ‹*equivalent-registers F G*›
shows ‹*equivalent-registers G F*›
by (*smt* (*verit*) *assms comp-id equivalent-registers-def equivalent-registers-register-right fun.map-comp iso-register-def*)

lemma *equivalent-registers-trans*[*trans*]:
assumes ‹*equivalent-registers F G*› **and** ‹*equivalent-registers G H*›
shows ‹*equivalent-registers F H*›
proof −
from *assms* **have** [*simp*]: ‹*register F*› ‹*register G*›
by (*auto simp*: *equivalent-registers-def*)
from *assms*(*1*) **obtain** *I* **where** [*simp*]: ‹*iso-register I*› **and** ‹*F o I* = *G*›
using *equivalent-registers-def* **by** *blast*
from *assms*(*2*) **obtain** *J* **where** [*simp*]: ‹*iso-register J*› **and** ‹*G o J* = *H*›
using *equivalent-registers-def* **by** *blast*
have ‹*register F*›
by (*auto simp*: *equivalent-registers-def*)
moreover have ‹*iso-register* (*I o J*)›
using ‹*iso-register I*› ‹*iso-register J*› *iso-register-comp* **by** *blast*
moreover have ‹*F o* (*I o J*) = *H*›
by (*simp add*: ‹*F* ∘ *I* = *G*› ‹*G* ∘ *J* = *H*› *o-assoc*)
ultimately show *?thesis*
by (*rule equivalent-registersI*)
qed

lemma *equivalent-registers-assoc*[*simp*]:
assumes [*simp*]: ‹*compatible F G*› ‹*compatible F H*› ‹*compatible G H*›
shows ‹*equivalent-registers* (*F*;(*G*;*H*)) ((*F*;*G*);*H*)›
apply (*rule equivalent-registersI*[**where** *I*=*assoc*])
by *auto*

lemma *equivalent-registers-pair-right*:
assumes [*simp*]: ‹*compatible F G*›

assumes *eq*: ‹*equivalent-registers G H*›
shows ‹*equivalent-registers* (*F*;*G*) (*F*;*H*)›
proof −
from *eq* **obtain** *I* **where** [*simp*]: ‹*iso-register I*› **and** ‹*G o I* = *H*›
by (*metis equivalent-registers-def*)
then have ∗: ‹(*F*;*G*) ∘ (*id* ⊗$_r$ *I*) = (*F*;*H*)›
by (*auto intro*!: *tensor-extensionality register-comp register-preregister register-tensor-is-register*
simp: *register-pair-apply iso-register-is-register*)
show *?thesis*
apply (*rule equivalent-registersI*[**where** *I*=‹*id* ⊗$_r$ *I*›])
using ∗ **by** (*auto intro*!: *iso-register-tensor-is-iso-register*)
qed

lemma *equivalent-registers-pair-left*:
assumes [*simp*]: ‹*compatible F G*›
assumes *eq*: ‹*equivalent-registers F H*›
shows ‹*equivalent-registers* (*F*;*G*) (*H*;*G*)›
proof −
from *eq* **obtain** *I* **where** [*simp*]: ‹*iso-register I*› **and** ‹*F o I* = *H*›
by (*metis equivalent-registers-def*)
then have ∗: ‹(*F*;*G*) ∘ (*I* ⊗$_r$ *id*) = (*H*;*G*)›
by (*auto intro*!: *tensor-extensionality register-comp register-preregister register-tensor-is-register*
simp: *register-pair-apply iso-register-is-register*)
show *?thesis*
apply (*rule equivalent-registersI*[**where** *I*=‹*I* ⊗$_r$ *id*›])
using ∗ **by** (*auto intro*!: *iso-register-tensor-is-iso-register*)
qed

lemma *equivalent-registers-comp*:
assumes ‹*register H*›
assumes ‹*equivalent-registers F G*›
shows ‹*equivalent-registers* (*H o F*) (*H o G*)›
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *comp-assoc equivalent-registers-def register-comp*)

lemma *equivalent-registers-compatible1*:
assumes ‹*compatible F G*›
assumes ‹*equivalent-registers F F'*›
shows ‹*compatible F' G*›
by (*metis assms*(*1*) *assms*(*2*) *compatible-comp-left equivalent-registers-def iso-register-is-register*)

lemma *equivalent-registers-compatible2*:
assumes ‹*compatible F G*›
assumes ‹*equivalent-registers G G'*›
shows ‹*compatible F G'*›
by (*metis assms*(*1*) *assms*(*2*) *compatible-comp-right equivalent-registers-def iso-register-is-register*)

2.10 Compatibility simplification

The simproc *compatibility-warn* produces helpful warnings for subgoals of the form *compatible x y* that are probably unsolvable due to missing declarations of variable compatibility facts. Same for subgoals of the form *register x*.

simproc-setup *compatibility-warn* (*compatible x y* | *register x*) = ‹
let val thy-string = *Markup.markup* (*Theory.get-markup* ***theory***) (*Context.theory-base-name* ***theory***)
in
fn m => *fn ctxt* => *fn ct* => *let*
val (*x*,*y*) = *case Thm.term-of ct of*
Const(***const-name***‹*compatible*›,-) $ *x* $ *y* => (*x*, *SOME y*)
| *Const*(***const-name***‹*register*›,-) $ *x* => (*x*, *NONE*)
val str : *string lazy* = *Lazy.lazy* (*fn* () => *Syntax.string-of-term ctxt* (*Thm.term-of ct*))
fun w msg = *warning* (*msg* ^\n(*Disable these warnings with*: *using* [[*simproc del*: ^*thy-string*^.*compatibility-warn*]]))
val - = *case* (*x*,*y*) *of*
(*Free*(*n*,*T*), *SOME* (*Free*(*n'*,*T'*))) =>
if String.isPrefix : *n orelse String.isPrefix* : *n' then*

```
                w (Simplification subgoal ' ^ Lazy.force str ^ ' contains a bound variable.\n ^
                Try to add some assumptions that makes this goal solvable by the simplifier)
          else if n=n' then (if T=T' then ()
                  else w (In simplification subgoal ' ^ Lazy.force str ^
                      ', variables have same name and different types.\n ^
                      Probably something is wrong.))
                else w (Simplification subgoal ' ^ Lazy.force str ^
                    ' occurred but cannot be solved.\n ^
                    Please add assumption/fact [simp]: ‹ ^ Lazy.force str ^
                    › somewhere.)
    | (Free(n,T), NONE) =>
        if String.isPrefix : n then
                w (Simplification subgoal ' ^ Lazy.force str ^ ' contains a bound variable.\n ^
                Try to add some assumptions that makes this goal solvable by the simplifier)
          else w (Simplification subgoal ' ^ Lazy.force str ^ ' occurred but cannot be solved.\n ^
                Please add assumption/fact [simp]: ‹ ^ Lazy.force str ^ › somewhere.)
    | - => ()
  in NONE end
end›
```

named-theorems *register-attribute-rule-immediate*
named-theorems *register-attribute-rule*

lemmas [*register-attribute-rule*] = *conjunct1 conjunct2 iso-register-is-register iso-register-is-register*[*OF iso-register-inv*]
lemmas [*register-attribute-rule-immediate*] = *compatible-sym compatible-register1 compatible-register2*
asm-rl[*of* ‹*compatible* - ->] *asm-rl*[*of* ‹*iso-register* ->] *asm-rl*[*of* ‹*register* ->] *iso-register-inv*

The following declares an attribute [*register*]. When the attribute is applied to a fact of the form *register F*, *iso-register F*, *compatible F G* or a conjunction of these, then those facts are added to the simplifier together with some derived theorems (e.g., *compatible F G* also adds *register F*).

In theory *Laws-Complement*, support for *is-unit-register F* and *complements F G* is added to this attribute.

```
setup ‹
let
fun add thm results =
  Net.insert-term (K true) (Thm.concl-of thm, thm) results
  handle Net.INSERT => results
fun try-rule f thm rule state = case SOME (rule OF [thm]) handle THM - => NONE  of
  NONE => state | SOME th => f th state
fun collect (rules,rules-immediate) thm results =
  results |> fold (try-rule add thm) rules-immediate |> fold (try-rule (collect (rules,rules-immediate)) thm) rules
fun declare thm context = let
  val ctxt = Context.proof-of context
  val rules = Named-Theorems.get ctxt @{named-theorems register-attribute-rule}
  val rules-immediate = Named-Theorems.get ctxt @{named-theorems register-attribute-rule-immediate}
  val thms = collect (rules,rules-immediate) thm Net.empty |> Net.entries
  (* val - = print thms *)
  in Simplifier.map-ss (fn ctxt => ctxt addsimps thms) context end
in
Attrib.setup binding‹register›
 (Scan.succeed (Thm.declaration-attribute declare))
  Add register−related rules to the simplifier
end
›
```

2.11 Notation

no-notation *comp-update* (**infixl** $*_u$ *55*)
no-notation *tensor-update* (**infixr** $\otimes_u$ *70*)

bundle *register-syntax* **begin**
notation *register-tensor* (**infixr** $\otimes_r$ *70*)

notation *register-pair* (*'*(-;-*'*))
end

end

3 Axioms of complements

theory *Axioms-Complement*
imports *Laws With-Type.With-Type*
begin

typedecl (*'a*, *'b*) *complement-domain*
instance *complement-domain* :: (*domain*, *domain*) *domain***..**
typedecl (*'a*, *'b*) *complement-domain-simple*
instance *complement-domain-simple* :: (*domain*, *domain*) *domain***..**

setup ‹ (∗ *Supporting with−type for the dummy class 'domain'* ∗)
With-Type.add-with-type-info-global {
class = ***class*** ‹*domain*›,
rep-class = ***const-name*** ‹*WITH-TYPE-CLASS-type*›,
rep-rel = ***const-name*** ‹*WITH-TYPE-REL-type*›,
with-type-wellformed = @{*thm with-type-wellformed-type*},
param-names = [],
transfer = *NONE*,
rep-rel-itself = *NONE*
}›

class *domain-with-simple-complement* = *domain*

— We need that there is at least one object in our category. We call is *some-domain*.
typedecl *some-domain*
instance *some-domain* :: *domain-with-simple-complement* **..**

axiomatization where
complement-exists-simple: ‹*register F* ⟹ ∃ *G* :: (*'a*, *'b*) *complement-domain-simple update* ⇒ *'b update*. *compatible F G* ∧ *iso-register* (*F*;*G*)›
for *F* :: ‹*'a*::*domain update* ⇒ *'b*::*domain-with-simple-complement update*›

axiomatization *cdc* :: ‹(*'a*::*domain update* ⇒ *'b*::*domain update*) ⇒ (*'a*,*'b*) *complement-domain set*› **where**
complement-exists: ‹*register F* ⟹ *let 'c*::*domain* = *cdc F in*
∃ *G* :: *'c update* ⇒ *'b update*. *compatible F G* ∧ *iso-register* (*F*;*G*)›
for *F* :: ‹*'a*::*domain update* ⇒ *'b*::*domain update*›

axiomatization where *complement-unique*: ‹*compatible F G* ⟹ *iso-register* (*F*;*G*) ⟹ *compatible F H* ⟹ *iso-register* (*F*;*H*)
⟹ *equivalent-registers G H*›
for *F* :: ‹*'a*::*domain update* ⇒ *'b*::*domain update*› **and** *G* :: ‹*'g*::*domain update* ⇒ *'b update*› **and** *H* :: ‹*'h*::*domain update* ⇒ *'b update*›

end

4 Generic laws about complements

theory *Laws-Complement*
imports *Laws Axioms-Complement*
begin

unbundle *register-syntax*
notation *comp-update* (**infixl** ∗*u* *55*)
notation *tensor-update* (**infixr** ⊗*u* *70*)

definition ‹*complements F G* ⟷ *compatible F G* ∧ *iso-register* (*F*;*G*)›

lemma *complementsI*: ‹*compatible F G* ⟹ *iso-register* (*F*;*G*) ⟹ *complements F G*›
using *complements-def* **by** *blast*

lemma *complement-exists*:
fixes *F* :: ‹'*a*::*domain update* ⇒ '*b*::*domain update*›
assumes ‹*register F*›
shows ‹*let* '*c*::*domain* = *cdc F in*
∃ *G* :: '*c update* ⇒ '*b update*. *complements F G*›
by (*simp add*: *assms complement-exists complements-def*)

lemma *complements-sym*: ‹*complements G F*› **if** ‹*complements F G*›
proof (*rule complementsI*)
show [*simp*]: ‹*compatible G F*›
using *compatible-sym complements-def that* **by** *blast*
from *that* **have** ‹*iso-register* (*F*;*G*)›
by (*meson complements-def*)
then obtain *I* **where** [*simp*]: ‹*register I*› **and** ‹(*F*;*G*) *o I* = *id*› **and** ‹*I o* (*F*;*G*) = *id*›
using *iso-register-def* **by** *blast*
have ‹*register* (*swap o I*)›
using ‹*register I*› *register-comp register-swap* **by** *blast*
moreover have ‹(*G*;*F*) *o* (*swap o I*) = *id*›
by (*simp add*: ‹(*F*;*G*) ∘ *I* = *id*› *rewriteL-comp-comp*)
moreover have ‹(*swap o I*) *o* (*G*;*F*) = *id*›
by (*metis* (*no-types*, *opaque-lifting*) *swap-swap* ‹*I* ∘ (*F*;*G*) = *id*› *calculation*(*2*) *comp-def eq-id-iff*)
ultimately show ‹*iso-register* (*G*;*F*)›
using ‹*compatible G F*› *iso-register-def pair-is-register* **by** *blast*
qed

definition *complement* :: ‹('*a*::*domain update* ⇒ '*b*::*domain-with-simple-complement update*) ⇒ (('*a*,'*b*) *complement-domain-simple update* ⇒ '*b update*)› **where**
‹*complement F* = (*SOME G* :: ('*a*, '*b*) *complement-domain-simple update* ⇒ '*b update*. *compatible F G* ∧ *iso-register* (*F*;*G*))›

lemma *register-complement*[*simp*]: ‹*register* (*complement F*)› **if** ‹*register F*›
using *complement-exists-simple*[*OF that*]
by (*metis* (*no-types*, *lifting*) *compatible-def complement-def some-eq-imp*)

lemma *complement-is-complement*[*simp*]:
assumes ‹*register F*›
shows ‹*complements F* (*complement F*)›
using *complement-exists-simple*[*OF assms*] **unfolding** *complements-def*
by (*metis* (*mono-tags*, *lifting*) *complement-def some-eq-imp*)

lemma *complement-unique*:
assumes ‹*complements F G*›
assumes ‹*complements F G'*›
shows ‹*equivalent-registers G G'*›
apply (*rule complement-unique*[**where** *F*=*F*])
using *assms* **unfolding** *complements-def* **by** *auto*

lemma *complement-unique'*:
assumes ‹*complements F G*›
shows ‹*equivalent-registers G* (*complement F*)›
apply (*rule complement-unique*[**where** *F*=*F*])
using *assms* **unfolding** *complements-def* **using** *compatible-register1 complement-is-complement complements-def*
by *blast*+

lemma *compatible-complement*[*simp*]: ‹*register F* ⟹ *compatible F* (*complement F*)›
using *complement-is-complement complements-def* **by** *blast*

lemma *complements-register-tensor*:

```
  assumes [simp]: ‹register F› ‹register G›
  shows ‹complements (F ⊗r G) (complement F ⊗r complement G)›
proof (rule complementsI)
  have [iff]: ‹iso-register (F;complement F)›  ‹iso-register (G;complement G)›
    using complements-def by fastforce+

  have sep4: ‹separating TYPE('z::domain) {(a ⊗u b) ⊗u (c ⊗u d) |a b c d. True}›
    apply (rule separating-tensor'[where A=‹{(a ⊗u b) |a b. True}› and B=‹{(c ⊗u d) |c d. True}›])
      apply (rule separating-tensor'[where A=UNIV and B=UNIV]) apply auto[3]
     apply (rule separating-tensor'[where A=UNIV and B=UNIV]) apply auto[3]
    by auto
  show compat: ‹compatible (F ⊗r G) (complement F ⊗r complement G)›
    by (metis assms(1) assms(2) compatible-register-tensor complement-is-complement complements-def)
  let ?reorder = ‹((Fst o Fst; Snd o Fst); (Fst o Snd; Snd o Snd))›
  have [simp]: ‹register ?reorder›
    by auto
  have [simp]: ‹?reorder ((a ⊗u b) ⊗u (c ⊗u d)) = ((a ⊗u c) ⊗u (b ⊗u d))›
   for a::‹'t::domain update› and b::‹'u::domain update› and c::‹'v::domain update› and d::‹'w::domain update›
    by (simp add: register-pair-apply Fst-def Snd-def tensor-update-mult)
  have [simp]: ‹iso-register ?reorder›
    apply (rule iso-registerI[of - ?reorder]) apply auto[2]
     apply (rule register-eqI[OF sep4]) apply auto[3]
    apply (rule register-eqI[OF sep4]) by auto
  have ‹(F ⊗r G; complement F ⊗r complement G) = ((F; complement F) ⊗r (G; complement G)) o ?reorder›
    apply (rule register-eqI[OF sep4])
    by (auto intro!: register-preregister register-comp register-tensor-is-register pair-is-register
        simp: compat register-pair-apply tensor-update-mult)
  moreover have ‹iso-register ...›
    using assms complement-is-complement complements-def
    by (auto intro!: iso-register-comp iso-register-tensor-is-iso-register)
  ultimately show ‹iso-register (F ⊗r G;complement F ⊗r complement G)›
    by simp
qed

definition is-unit-register where
  ‹is-unit-register U ⟷ complements U id›

lemma register-unit-register[simp]: ‹is-unit-register U ⟹ register U›
  by (simp add: compatible-def complements-def is-unit-register-def)

lemma unit-register-compatible[simp]: ‹compatible U X› if ‹is-unit-register U› ‹register X›
  by (metis compatible-comp-right complements-def id-comp is-unit-register-def that(1) that(2))

lemma unit-register-compatible'[simp]: ‹compatible X U› if ‹is-unit-register U› ‹register X›
  using compatible-sym that(1) that(2) unit-register-compatible by blast

lemma compatible-complement-left[simp]: ‹register X ⟹ compatible (complement X) X›
  using compatible-sym complement-is-complement complements-def by blast

lemma compatible-complement-right[simp]: ‹register X ⟹ compatible X (complement X)›
  using complement-is-complement complements-def by blast

lemma unit-register-pair[simp]: ‹equivalent-registers X (U; X)› if [simp]: ‹is-unit-register U› ‹register X›
proof −
  from complement-exists[OF ‹register X›]
  have ‹let 'x::domain = cdc X in equivalent-registers X (U; X)›
  proof with-type-mp
    note [[simproc del: compatibility-warn]]

    with-type-case
    then obtain compX :: ‹'x update ⇒ 'b update› where compX: ‹complements X compX›
      by blast
    then have [simp]: ‹register compX› ‹compatible X compX›
```

```
    by (auto simp add: compatible-def complements-def)
  have [iff]: ‹iso-register (X;compX)›
    using compX complements-def by blast

  have ‹equivalent-registers id (U; id)›
    using complements-def is-unit-register-def iso-register-equivalent-id that(1) by blast
  also have ‹equivalent-registers ... (U; (X; compX))›
    apply (rule equivalent-registers-pair-right)
    by (auto intro!: unit-register-compatible)
  also have ‹equivalent-registers ... ((U; X); compX)›
    apply (rule equivalent-registers-assoc)
    by auto
  finally have ‹complements (U; X) compX›
    by (auto simp: equivalent-registers-def complements-def)
  moreover have ‹equivalent-registers (X; compX) id›
    using compX complements-def equivalent-registers-sym iso-register-equivalent-id by blast
  ultimately show ‹equivalent-registers X (U; X)›
    by (meson complement-unique compX complements-sym)
 qed
 from this[cancel-with-type]
 show ‹equivalent-registers X (U; X)›
  by –
qed


lemma unit-register-compose-left:
 assumes [simp]: ‹is-unit-register U›
 assumes [simp]: ‹register A›
 shows ‹is-unit-register (A o U)›
proof –
 from complement-exists[OF ‹register A›]
 have ‹let 'x::domain = cdc A in is-unit-register (A o U)›
 proof with-type-mp
  note [[simproc del: compatibility-warn]]
  with-type-case
  then obtain compA :: ‹'x update ⇒ 'c update› where compX: ‹complements A compA›
    by blast
  then have [simp]: ‹register compA› ‹compatible A compA›
    by (auto simp add: compatible-def complements-def)
  have [iff]: ‹iso-register (A;compA)›
    using compX complements-def by blast

  have ‹compatible (A ∘ U) A›
    by (metis assms(1) assms(2) comp-id compatible-comp-inner complements-def is-unit-register-def)
  then have compat'[simp]: ‹compatible (A o U) (A; compA)›
    by (auto intro!: compatible3')
  moreover have ‹equivalent-registers (A; compA) id›
    using compX complements-def equivalent-registers-sym iso-register-equivalent-id by blast
  ultimately have compat[simp]: ‹compatible (A o U) id›
    using equivalent-registers-compatible2 by blast

  have aux: ‹equivalent-registers (U;id) id›
    using assms(1) equivalent-registers-sym register-id unit-register-pair by blast

  have ‹equivalent-registers (A o U; id) (A o U; (A; compA))›
    by (auto intro!: equivalent-registers-pair-right)
  also have ‹equivalent-registers ... (A o U; (A o id; compA))›
    by (auto intro!: equivalent-registers-refl pair-is-register)
  also have ‹equivalent-registers ... ((A o U; A o id); compA)›
    apply (intro equivalent-registers-assoc compatible-comp-inner)
    by auto
  also have ‹equivalent-registers ... (A o (U; id); compA)›
   by (metis (no-types, opaque-lifting) assms(1) assms(2) calculation complements-def equivalent-registers-sym
```

equivalent-registers-trans is-unit-register-def register-comp-pair)
  also have ‹*equivalent-registers* ... (*A o id*; *compA*)›
    apply (*intro equivalent-registers-pair-left equivalent-registers-comp*)
    using *aux* **by** (*auto simp*: *assms*)
  also have ‹*equivalent-registers* ... *id*›
    by (*simp add*: ‹*equivalent-registers* (*A*;*compA*) *id*›)
  finally show ‹*is-unit-register* (*A o U*)›
    using *compat complementsI equivalent-registers-sym is-unit-register-def iso-register-equivalent-id* **by** *blast*
 qed
 from *this*[*cancel-with-type*]
 show *?thesis*
  by −
qed

lemma *unit-register-compose-right*:
 assumes [*simp*]: ‹*is-unit-register U*›
 assumes [*simp*]: ‹*iso-register A*›
 shows ‹*is-unit-register* (*U o A*)›
proof (*unfold is-unit-register-def*, *rule complementsI*)
 show ‹*compatible* (*U* ∘ *A*) *id*›
  by (*simp add*: *iso-register-is-register*)
 have *1*: ‹*iso-register* ((*U*;*id*) ∘ *A* ⊗$_r$ *id*)›
  by (*meson assms*(*1*) *assms*(*2*) *complements-def is-unit-register-def iso-register-comp iso-register-id iso-register-tensor-is-iso-regist*
 have *2*: ‹*id* ∘ ((*U*;*id*) ∘ *A* ⊗$_r$ *id*) = (*U* ∘ *A*;*id*)›
  by (*metis assms*(*1*) *assms*(*2*) *complements-def fun.map-id is-unit-register-def iso-register-id iso-register-is-register*
pair-o-tensor)
 show ‹*iso-register* (*U* ∘ *A*;*id*)›
  using *1 2* **by** *auto*
qed

lemma *unit-register-unique*:
 assumes ‹*is-unit-register F*›
 assumes ‹*is-unit-register G*›
 shows ‹*equivalent-registers F G*›
proof −
 have ‹*complements F id*› ‹*complements G id*›
  using *assms* **by** (*metis complements-def equivalent-registers-def id-comp is-unit-register-def*)+
 then show *?thesis*
  by (*meson complement-unique complements-sym equivalent-registers-sym equivalent-registers-trans*)
qed

lemma *unit-register-domains-isomorphic*:
 fixes *F* :: ‹'*a*::*domain update* ⇒ '*c*::*domain update*›
 fixes *G* :: ‹'*b*::*domain update* ⇒ '*d*::*domain update*›
 assumes ‹*is-unit-register F*›
 assumes ‹*is-unit-register G*›
 shows ‹∃ *I* :: '*a update* ⇒ '*b update*. *iso-register I*›
proof −
 have ‹*is-unit-register* ((λ*d*. *tensor-update id-update d*) *o G*)›
  by (*simp add*: *assms*(*2*) *unit-register-compose-left*)
 moreover have ‹*is-unit-register* ((λ*c*. *tensor-update c id-update*) *o F*)›
  using *assms*(*1*) *register-tensor-left unit-register-compose-left* **by** *blast*
 ultimately have ‹*equivalent-registers* ((λ*d*. *tensor-update id-update d*) *o G*) ((λ*c*. *tensor-update c id-update*)
o F)›
  using *unit-register-unique* **by** *blast*
 then show *?thesis*
  unfolding *equivalent-registers-def* **by** *auto*
qed

lemma *id-complement-is-unit-register*[*simp*]: ‹*is-unit-register* (*complement id*)›
 by (*metis is-unit-register-def complement-is-complement complements-def complements-sym equivalent-registers-def*
id-comp register-id)

type-synonym *unit-register-domain* = ‹(*some-domain*, *some-domain*) *complement-domain-simple*›
definition *unit-register* :: ‹*unit-register-domain update* ⇒ *'a*::*domain update*› **where** ‹*unit-register* = (*SOME U*. *is-unit-register U*)›

lemma *unit-register-is-unit-register*[*simp*]: ‹*is-unit-register* (*unit-register* :: *unit-register-domain update* ⇒ *'a*::*domain update*)›
proof −
  note [[*simproc del*: *compatibility-warn*]]
  let *?U* = ‹*unit-register* :: *unit-register-domain update* ⇒ *'a*::*domain update*›
  let *?U1* = ‹*complement id* :: *unit-register-domain update* ⇒ *some-domain update*›
  from *complement-exists*[*OF register-id*[**where** *'a*=*'a*]]
  have ‹*let 'x*::*domain* = *cdc* (*id*::*'a update* ⇒ -) *in is-unit-register ?U*›
  proof *with-type-mp*
    with-type-case
    then obtain *U2* :: ‹*'x update* ⇒ *'a update*› **where** *comp1*: ‹*complements id U2*›
      by *blast*
    then have [*simp*]: ‹*register U2*› ‹*compatible id U2*› ‹*compatible id U2*›
      by (*auto simp add*: *compatible-def complements-def*)

    have ‹*is-unit-register ?U1*› ‹*is-unit-register U2*›
       by (*auto simp*: *comp1 complements-sym is-unit-register-def*)

    then obtain *I* :: ‹*unit-register-domain update* ⇒ *'x update*› **where** ‹*iso-register I*›
      apply *atomize-elim* **by** (*rule unit-register-domains-isomorphic*)
    with ‹*is-unit-register U2*› **have** ‹*is-unit-register* (*U2 o I*)›
      by (*rule unit-register-compose-right*)
    then show ‹*is-unit-register ?U*›
      by (*metis someI-ex unit-register-def*)
  qed
  from *this*[*cancel-with-type*]
  show *?thesis*
    by −
qed

lemma *unit-register-domain-tensor-unit*:
  fixes *U* :: ‹*'a*::*domain update* ⇒ -›
  assumes ‹*is-unit-register U*›
  shows ‹∃ *I* :: *'b*::*domain update* ⇒ (*'a*∗*'b*) *update*. *iso-register I*›

proof −
  from *complement-exists*[*OF register-id*[**where** *'a*=*'b*]]
  have ‹*let 'x*::*domain* = *cdc* (*id* :: *'b update* ⇒ -) *in*
        ∃ *I* :: *'b*::*domain update* ⇒ (*'a*∗*'b*) *update*. *iso-register I*›
  proof *with-type-mp*
    note [[*simproc del*: *compatibility-warn*]]
    with-type-case
    assume ‹∃ *G* :: *'x update* ⇒ *'b update*. *complements id G*›
    then obtain *U'* :: ‹*'x update* ⇒ *'b update*› **where** *comp*: ‹*complements id U'*›
      by *blast*
    then have [*simp*]: ‹*register U'*› ‹*compatible id U'*› ‹*compatible U' id*›
      by (*auto simp add*: *compatible-def complements-def*)
    have ‹*is-unit-register U'*›
      by (*simp add*: *comp complements-sym is-unit-register-def*)

    have ‹*equivalent-registers* (*id* :: *'b update* ⇒ -) (*U'*; *id*)›
      using *comp complements-def complements-sym iso-register-equivalent-id* **by** *blast*
    then obtain *J* :: ‹*'b update* ⇒ ((*'x* ∗ *'b*) *update*)› **where** ‹*iso-register J*›
      using *equivalent-registers-def iso-register-inv* **by** *blast*
    moreover obtain *K* :: ‹*'x update* ⇒ *'a update*› **where** ‹*iso-register K*›
      apply *atomize-elim*
      using ‹*is-unit-register U'*› *assms*
      by (*rule unit-register-domains-isomorphic*)

ultimately have ‹*iso-register* ((*K* ⊗$_r$ *id*) *o J*)›
by *auto*
then show ‹∃ *I* :: *'b*::*domain update* ⇒ (*'a*∗*'b*) *update*. *iso-register I*›
by *auto*
qed
from *this*[*cancel-with-type*]
show *?thesis*
by−
qed

lemma *compatible-complement-pair1*:
assumes ‹*compatible F G*›
shows ‹*compatible F* (*complement* (*F*;*G*))›
by (*metis assms compatible-comp-left compatible-complement-right pair-is-register register-Fst register-pair-Fst*)

lemma *compatible-complement-pair2*:
assumes [*simp*]: ‹*compatible F G*›
shows ‹*compatible G* (*complement* (*F*;*G*))›
proof −
have ‹*compatible* (*F*;*G*) (*complement* (*F*;*G*))›
by *simp*
then have ‹*compatible* ((*F*;*G*) *o Snd*) (*complement* (*F*;*G*))›
by *auto*
then show *?thesis*
by (*auto simp*: *register-pair-Snd*)
qed

lemma *equivalent-complements*:
assumes ‹*complements F G*›
assumes ‹*equivalent-registers G G'*›
shows ‹*complements F G'*›
apply (*rule complementsI*)
apply (*metis assms*(*1*) *assms*(*2*) *compatible-comp-right complements-def equivalent-registers-def iso-register-is-register*)
by (*metis assms*(*1*) *assms*(*2*) *complements-def equivalent-registers-def equivalent-registers-pair-right iso-register-comp*)

lemma *complements-complement-pair*:
assumes [*simp*]: ‹*compatible F G*›
assumes *FG'*: ‹*complements* (*F*;*G*) *FG'*›
shows ‹*complements F* (*G*; *FG'*)›
proof (*rule complementsI*)
note [[*simproc del*: *compatibility-warn*]]
have ‹*compatible* (*F*;*G*) *FG'*›
using *FG' complements-def* **by** *auto*
then have [*simp*]: ‹*compatible F FG'*›
by (*smt* (*verit*)*assms*(*1*) *compatibleI compatible-register1 compatible-register2 id-update-right register-of-id register-pair-apply' swap-registers*)
have [*simp*]: ‹*compatible G FG'*›
by (*smt* (*verit*) *register-pair-apply* ‹*compatible* (*F*;*G*) *FG'*› *assms*(*1*) *compatibleI compatible-register1 compatible-register2 id-update-right register-of-id swap-registers*)

have ‹*equivalent-registers* (*F*; (*G*; *FG'*)) ((*F*;*G*); *FG'*)›
apply (*rule equivalent-registers-assoc*)
apply *simp*
apply (*smt* (*verit*) ‹*compatible* (*F*;*G*) *FG'*› *assms*(*1*) *compatibleI compatible-register1 compatible-register2 id-update-right register-of-id register-pair-apply' swap-registers*)
by (*smt* (*verit*) *register-pair-apply* ‹*compatible* (*F*;*G*) *FG'*› *assms*(*1*) *compatibleI compatible-register1 compatible-register2 id-update-right register-of-id swap-registers*)
also have ‹*equivalent-registers* ... *id*›
by (*meson assms complement-is-complement complements-def equivalent-registers-sym iso-register-equivalent-id pair-is-register*)
finally show ‹*iso-register* (*F*;(*G*;*FG'*))›
using *equivalent-registers-sym iso-register-equivalent-id* **by** *blast*
show ‹*compatible F* (*G*;*FG'*)›

by (*auto intro*!: *compatible3'*)
qed

lemma *equivalent-registers-complement*:
assumes ‹*equivalent-registers F G*›
assumes ‹*complements F F'*›
assumes ‹*complements G G'*›
shows ‹*equivalent-registers F' G'*›
by (*meson complement-unique assms*(*1*) *assms*(*2*) *assms*(*3*) *complements-sym equivalent-complements*)

lemma *equivalent-registers-complement'*:
assumes ‹*equivalent-registers F G*›
shows ‹*equivalent-registers* (*complement F*) (*complement G*)›
using *assms* **apply** (*rule equivalent-registers-complement*)
using *assms complement-is-complement equivalent-registers-register-left equivalent-registers-register-right*
by *blast*+

lemma *complements-complement-pair'*:
assumes [*simp*]: ‹*compatible F G*›
assumes *FG'*: ‹*complements* (*F*;*G*) *FG'*›
shows ‹*complements G* (*F*; *FG'*)›
proof −
have ‹*equivalent-registers* (*F*;*G*) (*G*;*F*)›
using *assms*(*1*) *equivalent-registers-def iso-register-swap pair-is-register pair-o-swap* **by** *blast*
with *FG'* **have** ∗: ‹*complements* (*G*;*F*) *FG'*›
by (*meson complements-sym equivalent-complements*)
show *?thesis*
apply (*rule complements-complement-pair*)
using ∗ **by** (*simp-all add*: *compatible-sym*)
qed

lemma *complements-chain*:
assumes [*simp*]: ‹*register F*› ‹*register G*›
shows ‹*complements* (*F o G*) (*complement F*; *F o complement G*)›
proof (*rule complementsI*)
show ‹*compatible* (*F o G*) (*complement F*; *F o complement G*)›
by *auto*
have ‹*equivalent-registers* (*F* ∘ *G*;(*complement F*;*F* ∘ *complement G*)) (*F* ∘ *G*;(*F* ∘ *complement G*;*complement F*))›
apply (*rule equivalent-registersI*[**where** *I*=‹*id* ⊗*r swap*›])
by (*auto intro*!: *iso-register-tensor-is-iso-register simp*: *pair-o-tensor*)
also have ‹*equivalent-registers* ... ((*F* ∘ *G*;*F* ∘ *complement G*);*complement F*)›
apply (*rule equivalent-registersI*[**where** *I*=*assoc*])
by (*auto intro*!: *iso-register-tensor-is-iso-register simp*: *pair-o-tensor*)
also have ‹*equivalent-registers* ... (*F o* (*G*; *complement G*);*complement F*)›
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *calculation compatible-complement-right equivalent-registers-sym equivalent-registers-trans register-comp-pair*)
also have ‹*equivalent-registers* ... (*F o id*;*complement F*)›
apply (*rule equivalent-registers-pair-left*, *simp*)
apply (*rule equivalent-registers-comp*, *simp*)
by (*metis assms*(*2*) *complement-is-complement complements-def equivalent-registers-def iso-register-def*)
also have ‹*equivalent-registers* ... *id*›
by (*metis assms*(*1*) *comp-id complement-is-complement complements-def equivalent-registers-def iso-register-def*)
finally show ‹*iso-register* (*F* ∘ *G*;(*complement F*;*F* ∘ *complement G*))›
using *equivalent-registers-sym iso-register-equivalent-id* **by** *blast*
qed

lemma *complements-Fst-Snd*[*simp*]: ‹*complements Fst Snd*›
by (*auto intro*!: *complementsI simp*: *pair-Fst-Snd*)

lemma *complements-Snd-Fst*[*simp*]: ‹*complements Snd Fst*›
by (*auto intro*!: *complementsI simp flip*: *swap-def*)

lemma *compatible-unit-register*[*simp*]: ‹*register F* ⟹ *compatible F unit-register*›
using *compatible-sym unit-register-compatible unit-register-is-unit-register* **by** *blast*

lemma *complements-id-unit-register*[*simp*]: ‹*complements id unit-register*›
using *complements-sym is-unit-register-def unit-register-is-unit-register* **by** *blast*

lemma *complements-iso-unit-register*: ‹*iso-register I* ⟹ *is-unit-register U* ⟹ *complements I U*›
using *complements-sym equivalent-complements is-unit-register-def iso-register-equivalent-id* **by** *blast*

lemma *iso-register-complement-is-unit-register*[*simp*]:
assumes ‹*iso-register F*›
shows ‹*is-unit-register* (*complement F*)›
by (*meson assms complement-is-complement complements-sym equivalent-complements equivalent-registers-sym is-unit-register-def iso-register-equivalent-id iso-register-is-register*)

Adding support for *is-unit-register F* and *complements F G* to the [*register*] attribute

lemmas [*register-attribute-rule*] = *is-unit-register-def*[*THEN iffD1*] *complements-def*[*THEN iffD1*]
lemmas [*register-attribute-rule-immediate*] = *asm-rl*[*of* ‹*is-unit-register* -›]

no-notation *comp-update* (**infixl** $*_u$ *55*)
no-notation *tensor-update* (**infixr** $\otimes_u$ *70*)
unbundle *no register-syntax*

end

5 Classical instantiation of registers

theory *Axioms-Classical*
imports *Main*
begin

type-synonym *'a update* = ‹*'a* ⇀ *'a*›

lemma *id-update-left*: *Some* $\circ_m$ *a* = *a*
by (*auto intro*!: *ext simp add*: *map-comp-def option.case-eq-if*)
lemma *id-update-right*: *a* $\circ_m$ *Some* = *a*
by *auto*

lemma *comp-update-assoc*: (*a* $\circ_m$ *b*) $\circ_m$ *c* = *a* $\circ_m$ (*b* $\circ_m$ *c*)
by (*auto intro*!: *ext simp add*: *map-comp-def option.case-eq-if*)

type-synonym (*'a*,*'b*) *preregister* = ‹*'a update* ⇒ *'b update*›
definition *preregister* :: ‹(*'a*,*'b*) *preregister* ⇒ *bool*› **where**
‹*preregister F* ⟷ (∃ *g s*. ∀ *a m*. *F a m* = (*case a* (*g m*) *of None* ⇒ *None* | *Some x* ⇒ *s x m*))›

lemma *id-preregister*: ‹*preregister id*›
unfolding *preregister-def*
apply (*rule exI*[*of* - ‹λ*m*. *m*›])
apply (*rule exI*[*of* - ‹λ*a m*. *Some a*›])
by (*simp add*: *option.case-eq-if*)

lemma *preregister-mult-right*: ‹*preregister* (λ*a*. *a* $\circ_m$ *z*)›
unfolding *preregister-def*
apply (*rule exI*[*of* - ‹λ*m*. *the* (*z m*)›])
apply (*rule exI*[*of* - ‹λ*x m*. *case z m of None* ⇒ *None* | - ⇒ *Some x*›])
by (*auto simp add*: *option.case-eq-if*)

lemma *preregister-mult-left*: ‹*preregister* (λ*a*. *z* $\circ_m$ *a*)›
unfolding *preregister-def*
apply (*rule exI*[*of* - ‹λ*m*. *m*›])
apply (*rule exI*[*of* - ‹λ*x m*. *z x*›])
by (*auto simp add*: *option.case-eq-if*)

lemma *comp-preregister*: *preregister* $(G \circ F)$ **if** *preregister* F **and** ‹*preregister* G›
proof −
from ‹*preregister* F›
obtain *sF gF* **where** *F*: ‹$F\ a\ m = (case\ a\ (gF\ m)\ of\ None \Rightarrow None \mid Some\ x \Rightarrow sF\ x\ m)$› **for** $a\ m$
using *preregister-def* **by** *blast*
from ‹*preregister* G›
obtain *sG gG* **where** *G*: ‹$G\ a\ m = (case\ a\ (gG\ m)\ of\ None \Rightarrow None \mid Some\ x \Rightarrow sG\ x\ m)$› **for** $a\ m$
using *preregister-def* **by** *blast*
define *s g* **where** ‹$s\ a\ m = (case\ sF\ a\ (gG\ m)\ of\ None \Rightarrow None \mid Some\ x \Rightarrow sG\ x\ m)$›
and ‹$g\ m = gF\ (gG\ m)$› **for** $a\ m$
have ‹$(G \circ F)\ a\ m = (case\ a\ (g\ m)\ of\ None \Rightarrow None \mid Some\ x \Rightarrow s\ x\ m)$› **for** $a\ m$
unfolding *F G s-def g-def*
by (*auto simp add*: *option.case-eq-if*)
then show *preregister* $(G \circ F)$
using *preregister-def* **by** *blast*
qed

definition *tensor-update* :: ‹$'a\ update \Rightarrow 'b\ update \Rightarrow ('a \times 'b)\ update$› **where**
‹*tensor-update a b m* = (*case a* (*fst m*) *of None* ⇒ *None* | *Some x* ⇒ (*case b* (*snd m*) *of None* ⇒ *None* | *Some y* ⇒ *Some* (*x*,*y*)))›

lemma *tensor-update-mult*: ‹*tensor-update a c* $\circ_m$ *tensor-update b d* = *tensor-update* (*a* $\circ_m$ *b*) (*c* $\circ_m$ *d*)›
by (*auto intro*!: *ext simp add*: *map-comp-def option.case-eq-if tensor-update-def*)

definition *update1* :: ‹$'a \Rightarrow 'a \Rightarrow 'a\ update$› **where**
‹*update1 x y m* = (*if m*=*x then Some y else None*)›

lemma *update1-extensionality*:
assumes ‹*preregister F*›
assumes ‹*preregister G*›
assumes *FGeq*: ‹$\bigwedge x\ y.\ F\ (update1\ x\ y) = G\ (update1\ x\ y)$›
shows $F = G$
proof (*rule ccontr*)
assume *neq*: ‹$F \neq G$›
then obtain *z m* **where** *neq'*: ‹$F\ z\ m \neq G\ z\ m$›
apply *atomize-elim* **by** *auto*
obtain *gF sF* **where** *gsF*: ‹$F\ z\ m = (case\ z\ (gF\ m)\ of\ None \Rightarrow None \mid Some\ x \Rightarrow sF\ x\ m)$› **for** *z m*
using ‹*preregister F*› *preregister-def* **by** *blast*
obtain *gG sG* **where** *gsG*: ‹$G\ z\ m = (case\ z\ (gG\ m)\ of\ None \Rightarrow None \mid Some\ x \Rightarrow sG\ x\ m)$› **for** *z m*
using ‹*preregister G*› *preregister-def* **by** *blast*
consider (*abeq*) *x* **where** ‹$z\ (gF\ m) = Some\ x$› ‹$z\ (gG\ m) = Some\ x$› ‹$gF\ m = gG\ m$›
| (*abnone*) ‹$z\ (gG\ m) = None$› ‹$z\ (gF\ m) = None$›
| (*neqF*) *x* **where** ‹$gF\ m \neq gG\ m$› ‹$F\ z\ m = Some\ x$›
| (*neqG*) *y* **where** ‹$gF\ m \neq gG\ m$› ‹$G\ z\ m = Some\ y$›
| (*neqNone*) ‹$gF\ m \neq gG\ m$› ‹$F\ z\ m = None$› ‹$G\ z\ m = None$›
apply *atomize-elim* **by** (*metis option.exhaust-sel*)
then show *False*
proof *cases*
case (*abeq x*)
then have ‹$F\ z\ m = sF\ x\ m$› **and** ‹$G\ z\ m = sG\ x\ m$›
by (*simp-all add*: *gsF gsG*)
moreover have ‹$F\ (update1\ (gF\ m)\ x)\ m = sF\ x\ m$›
by (*simp add*: *gsF update1-def*)
moreover have ‹$G\ (update1\ (gF\ m)\ x)\ m = sG\ x\ m$›
by (*simp add*: *abeq gsG update1-def*)
ultimately show *False*
using *FGeq neq'* **by** *force*
next
case *abnone*
then show *False*
using *gsF gsG neq'* **by** *force*
next

```
    case neqF
    moreover
    have ‹F (update1 (gF m) (the (z (gF m)))) m = F z m›
      by (metis gsF neqF(2) option.case-eq-if option.simps(3) option.simps(5) update1-def)
    moreover have ‹G (update1 (gF m) (the (z (gF m)))) m = None›
      by (metis gsG neqF(1) option.case-eq-if update1-def)
    ultimately show False
      using FGeq by force
  next
    case neqG
    moreover
    have ‹G (update1 (gG m) (the (z (gG m)))) m = G z m›
      by (metis gsG neqG(2) option.case-eq-if option.distinct(1) option.simps(5) update1-def)
    moreover have ‹F (update1 (gG m) (the (z (gG m)))) m = None›
      by (simp add: gsF neqG(1) update1-def)
    ultimately show False
      using FGeq by force
  next
    case neqNone
    with neq' show False
      by fastforce
  qed
qed

lemma tensor-extensionality:
  assumes ‹preregister F›
  assumes ‹preregister G›
  assumes FGeq: ‹⋀a b. F (tensor-update a b) = G (tensor-update a b)›
  shows F = G
proof −
  have aux1: ‹(case (if a then b else c) of Some x ⇒ f x | None ⇒ g) =
          (if a then (case b of Some x ⇒ f x | None ⇒ g) else (case c of Some x ⇒ f x | None ⇒ g))› for a b c
f g
    by simp
  have aux2: ‹F (λm. if m = x then Some y else None) z = G (λm. if m = x then Some y else None) z›
    if ‹F (λm. if fst m = fst x then if snd m = snd x then Some y else None else None) =
        G (λm. if fst m = fst x then if snd m = snd x then Some y else None else None)› for x y z
    using assms that
    by (smt (z3) preregister-def prod.collapse)
  have ‹F (update1 x y) = G (update1 x y)› for x y
    using FGeq[of ‹update1 (fst x) (fst y)› ‹update1 (snd x) (snd y)›]
    by (auto intro!:ext simp: tensor-update-def[abs-def] update1-def[abs-def] aux1 aux2
        cong: if-cong)
  with assms(1,2) show F = G
    by (rule update1-extensionality)
qed

definition valid-getter-setter g s ⟷
  (∀b. b = s (g b) b) ∧ (∀a b. g (s a b) = a) ∧ (∀a a' b. s a (s a' b) = s a b)

definition ‹register-from-getter-setter g s a m = (case a (g m) of None ⇒ None | Some x ⇒ Some (s x m))›

definition ‹register-apply F a = the o F (Some o a)›
definition ‹setter F a m = register-apply F (λ-. a) m› for F :: ‹'a update ⇒ 'b update›
definition ‹getter F m = (THE x. setter F x m = m)› for F :: ‹'a update ⇒ 'b update›

lemma
  assumes ‹valid-getter-setter g s›
  shows getter-of-register-from-getter-setter[simp]: ‹getter (register-from-getter-setter g s) = g›
    and setter-of-register-from-getter-setter[simp]: ‹setter (register-from-getter-setter g s) = s›
proof −
  define g' s' where ‹g' = getter (register-from-getter-setter g s)›
    and ‹s' = setter (register-from-getter-setter g s)›
```

```
  show ‹s' = s›
    by (auto intro!:ext simp: s'-def setter-def register-apply-def register-from-getter-setter-def)
  moreover show ‹g' = g›
  proof (rule ext, rename-tac m)
    fix m
    have ‹g' m = (THE x. s x m = m)›
     by (auto intro!:ext simp: g'-def s'-def[symmetric] ‹s'=s› getter-def register-apply-def register-from-getter-setter-def)
    moreover have ‹s (g m) m = m›
      by (metis assms valid-getter-setter-def)
    moreover have ‹x = x'› if ‹s x m = m› ‹s x' m = m› for x x'
      by (metis assms that(1) that(2) valid-getter-setter-def)
    ultimately show ‹g' m = g m›
      by (simp add: Uniq-def the1-equality')
  qed
qed

definition register :: ‹('a,'b) preregister ⇒ bool› where
  ‹register F ⟷ (∃ g s. F = register-from-getter-setter g s ∧ valid-getter-setter g s)›

lemma register-of-id: ‹register F ⟹ F Some = Some›
  by (auto simp add: register-def valid-getter-setter-def register-from-getter-setter-def)

lemma register-id: ‹register id›
  unfolding register-def
  apply (rule exI[of - id], rule exI[of - ‹λa m. a›])
  by (auto intro!: ext simp: option.case-eq-if register-from-getter-setter-def valid-getter-setter-def)

lemma register-tensor-left: ‹register (λa. tensor-update a Some)›
  by (auto simp: register-def intro!: exI[of - fst] exI[of - ‹λx' (x,y). (x',y)›]
      intro!: ext simp add: tensor-update-def valid-getter-setter-def register-from-getter-setter-def option.case-eq-if)

lemma register-tensor-right: ‹register (λa. tensor-update Some a)›
  by (auto simp: register-def intro!: exI[of - snd] exI[of - ‹λy' (x,y). (x,y')›]
      intro!: ext simp add: tensor-update-def valid-getter-setter-def register-from-getter-setter-def option.case-eq-if)

lemma register-preregister: preregister F if ‹register F›
proof −
  from ‹register F›
  obtain s g where F: ‹F a m = (case a (g m) of None ⇒ None | Some x ⇒ Some (s x m))› for a m
    unfolding register-from-getter-setter-def register-def by blast
  show ?thesis
    unfolding preregister-def
    apply (rule exI[of - g])
    apply (rule exI[of - ‹λx m. Some (s x m)›])
    using F by simp
qed

lemma register-comp: register (G ∘ F) if ‹register F› and ‹register G›
  for F :: ('a,'b) preregister and G :: ('b,'c) preregister
proof −
  from ‹register F›
  obtain sF gF where F: ‹F a m = (case a (gF m) of None ⇒ None | Some x ⇒ Some (sF x m))›
    and validF: ‹valid-getter-setter gF sF› for a m
    unfolding register-def register-from-getter-setter-def by blast
  from ‹register G›
  obtain sG gG where G: ‹G a m = (case a (gG m) of None ⇒ None | Some x ⇒ Some (sG x m))›
    and validG: ‹valid-getter-setter gG sG› for a m
    unfolding register-def register-from-getter-setter-def by blast
  define s g where ‹s a m = sG (sF a (gG m)) m› and ‹g m = gF (gG m)› for a m
  have ‹(G ∘ F) a m = (case a (g m) of None ⇒ None | Some x ⇒ Some (s x m))› for a m
    by (auto simp add: option.case-eq-if F G s-def g-def)
  moreover have ‹valid-getter-setter g s›
    using validF validG by (auto simp: valid-getter-setter-def s-def g-def)
```

ultimately show *register* $(G \circ F)$
unfolding *register-def register-from-getter-setter-def* **by** *blast*
qed

lemma *register-mult*: *register* $F \Longrightarrow F\ a \circ_m F\ b = F\ (a \circ_m b)$
by (*auto intro*!: *ext simp*: *register-def register-from-getter-setter-def*[*abs-def*] *valid-getter-setter-def map-comp-def option.case-eq-if*)

definition *register-pair* ::
‹('*a update* ⇒ '*c update*) ⇒ ('*b update* ⇒ '*c update*) ⇒ (('*a*×'*b*) *update* ⇒ '*c update*)› **where**
‹*register-pair F G* =
register-from-getter-setter (λ*m*. (*getter F m*, *getter G m*)) (λ(*a*,*b*) *m*. *setter F a* (*setter G b m*))›

lemma *compatible-setter*:
assumes [*simp*]: ‹*register F*› ‹*register G*›
assumes *compat*: ‹⋀*a b*. *F a* ∘$_m$ *G b* = *G b* ∘$_m$ *F a*›
shows ‹*setter F x o setter G y* = *setter G y o setter F x*›
proof −
have ∗: ‹*F* (λ*xa*. *Some x*) (*the* (*G* (λ*x*. *Some y*) *xa*)) = *G* (λ*x*. *Some y*) (*the* (*F* (λ*xa*. *Some x*) *xa*))›
if ‹⋀*a b*. (λ*k*. *case G b k of None* ⇒ *None* | *Some v* ⇒ *F a v*) =
(λ*k*. *case F a k of None* ⇒ *None* | *Some v* ⇒ *G b v*)› **for** *xa*
using *that assms* **by** (*smt* (*verit*, *best*) *option.case-eq-if option.distinct*(*1*) *register-def register-from-getter-setter-def*)
then show *?thesis*
using *compat* **by** (*auto intro*!: *ext simp*: *setter-def register-apply-def o-def map-comp-def*)
qed

lemma *register-pair-apply*:
assumes [*simp*]: ‹*register F*› ‹*register G*›
assumes ‹⋀*a b*. *F a* ∘$_m$ *G b* = *G b* ∘$_m$ *F a*›
shows ‹(*register-pair F G*) (*tensor-update a b*) = *F a* ∘$_m$ *G b*›
proof −
have *validF*: ‹*valid-getter-setter* (*getter F*) (*setter F*)› **and** *validG*: ‹*valid-getter-setter* (*getter G*) (*setter G*)›
by (*metis assms getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter*)+
then have *F*: ‹*F* = *register-from-getter-setter* (*getter F*) (*setter F*)› **and** *G*: ‹*G* = *register-from-getter-setter* (*getter G*) (*setter G*)›
by (*metis assms getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter*)+
have *gFsG*: ‹*getter F* (*setter G y m*) = *getter F m*› **for** *y m*
proof −
have ‹*getter F* (*setter G y m*) = *getter F* (*setter G y* (*setter F* (*getter F m*) *m*))›
using *validF* **by** (*metis valid-getter-setter-def*)
also have ‹... = *getter F* (*setter F* (*getter F m*) (*setter G y m*))›
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *comp-eq-dest-lhs compatible-setter*)
also have ‹... = *getter F m*›
by (*metis validF valid-getter-setter-def*)
finally show *?thesis* **by** −
qed

show *?thesis*
apply (*subst* (*2*) *F*, *subst* (*2*) *G*)
by (*auto intro*!:*ext simp*: *register-pair-def tensor-update-def map-comp-def option.case-eq-if*
register-from-getter-setter-def gFsG)
qed

lemma *register-pair-is-register*:
fixes *F* :: ‹'*a update* ⇒ '*c update*› **and** *G*
assumes [*simp*]: ‹*register F*› **and** [*simp*]: ‹*register G*›
assumes *compat*: ‹⋀*a b*. *F a* ∘$_m$ *G b* = *G b* ∘$_m$ *F a*›
shows ‹*register* (*register-pair F G*)›
proof −
have *validF*: ‹*valid-getter-setter* (*getter F*) (*setter F*)› **and** *validG*: ‹*valid-getter-setter* (*getter G*) (*setter G*)›
by (*metis assms getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter*)+
then have ‹*valid-getter-setter* (λ*m*. (*getter F m*, *getter G m*)) (λ(*a*, *b*) *m*. *setter F a* (*setter G b m*))›
apply (*simp add*: *valid-getter-setter-def*)

by (*metis* (*mono-tags*, *lifting*) *assms comp-eq-dest-lhs compat compatible-setter*)
then show *?thesis*
by (*auto simp*: *register-pair-def register-def*)
qed

end

6 Generic laws about registers, instantiated classically

theory *Laws-Classical*
imports *Axioms-Classical*
begin

This notation is only used inside this file

notation *map-comp* (**infixl** $*_u$ *55*)
notation *tensor-update* (**infixr** $\otimes_u$ *70*)
notation *register-pair* (‹(-;-)›)

6.1 Elementary facts

declare *id-preregister*[*simp*]
declare *id-update-left*[*simp*]
declare *id-update-right*[*simp*]
declare *register-preregister*[*simp*]
declare *register-comp*[*simp*]
declare *register-of-id*[*simp*]
declare *register-tensor-left*[*simp*]
declare *register-tensor-right*[*simp*]
declare *preregister-mult-right*[*simp*]
declare *preregister-mult-left*[*simp*]
declare *register-id*[*simp*]

6.2 Preregisters

lemma *preregister-tensor-left*[*simp*]: ‹*preregister* (λ*b*::'*b*::*type update*. *tensor-update a b*)›
for *a* :: ‹'*a*::*type update*›
proof −
have ‹*preregister* ((λ*b1*::('*a*×'*b*) *update*. (*a* $\otimes_u$ *Some*) $*_u$ *b1*) *o* (λ*b*. *tensor-update Some b*))›
by (*rule comp-preregister*; *simp*)
then show *?thesis*
by (*simp add*: *o-def tensor-update-mult*)
qed

lemma *preregister-tensor-right*[*simp*]: ‹*preregister* (λ*a*::'*a*::*type update*. *tensor-update a b*)›
for *b* :: ‹'*b*::*type update*›
proof −
have ‹*preregister* ((λ*a1*::('*a*×'*b*) *update*. (*Some* $\otimes_u$ *b*) $*_u$ *a1*) *o* (λ*a*. *tensor-update a Some*))›
by (*rule comp-preregister*, *simp-all*)
then show *?thesis*
by (*simp add*: *o-def tensor-update-mult*)
qed

6.3 Registers

lemma *id-update-tensor-register*[*simp*]:
assumes ‹*register F*›
shows ‹*register* (λ*a*::'*a*::*type update*. *Some* $\otimes_u$ *F a*)›
using *assms* **apply** (*rule register-comp*[*unfolded o-def*])
by *simp*

lemma *register-tensor-id-update*[*simp*]:
assumes ‹*register F*›
shows ‹*register* (λ*a*::'*a*::*type update*. *F a* $\otimes_u$ *Some*)›

using *assms* **apply** (*rule register-comp*[*unfolded o-def*])
by *simp*

6.4 Tensor product of registers

definition *register-tensor* (**infixr** $\otimes_r$ *70*) **where**
register-tensor F G = *register-pair* (λ*a*. *tensor-update* (*F a*) *Some*) (λ*b*. *tensor-update Some* (*G b*))

lemma *register-tensor-is-register*:
fixes *F* :: *'a*::*type update* ⇒ *'b*::*type update* **and** *G* :: *'c*::*type update* ⇒ *'d*::*type update*
shows *register F* ⟹ *register G* ⟹ *register* (*F* ⊗$_r$ *G*)
unfolding *register-tensor-def*
apply (*rule register-pair-is-register*)
by (*simp-all add*: *tensor-update-mult*)

lemma *register-tensor-apply*[*simp*]:
fixes *F* :: *'a*::*type update* ⇒ *'b*::*type update* **and** *G* :: *'c*::*type update* ⇒ *'d*::*type update*
assumes ‹*register F*› **and** ‹*register G*›
shows (*F* ⊗$_r$ *G*) (*a* ⊗$_u$ *b*) = *F a* ⊗$_u$ *G b*
unfolding *register-tensor-def*
apply (*subst register-pair-apply*)
unfolding *register-tensor-def*
by (*simp-all add*: *assms tensor-update-mult*)

definition *separating* (-::*'b*::*type itself*) *A* ⟷
(∀ *F G* :: *'a*::*type update* ⇒ *'b update*. *preregister F* ⟶ *preregister G* ⟶ (∀ *x*∈*A*. *F x* = *G x*) ⟶ *F* = *G*)

lemma *separating-UNIV*[*simp*]: ‹*separating TYPE*(-) *UNIV*›
unfolding *separating-def* **by** *auto*

lemma *separating-mono*: ‹*A* ⊆ *B* ⟹ *separating TYPE*(*'a*::*type*) *A* ⟹ *separating TYPE*(*'a*) *B*›
unfolding *separating-def* **by** (*meson in-mono*)

lemma *register-eqI*: ‹*separating TYPE*(*'b*::*type*) *A* ⟹ *preregister F* ⟹ *preregister G* ⟹ (⋀*x*. *x*∈*A* ⟹ *F x* = *G x*) ⟹ *F* = (*G*::- ⇒ *'b update*)›
unfolding *separating-def* **by** *auto*

lemma *separating-tensor*:
fixes *A* :: ‹*'a*::*type update set*› **and** *B* :: ‹*'b*::*type update set*›
assumes [*simp*]: ‹*separating TYPE*(*'c*::*type*) *A*›
assumes [*simp*]: ‹*separating TYPE*(*'c*) *B*›
shows ‹*separating TYPE*(*'c*) {*a* ⊗$_u$ *b* | *a b*. *a*∈*A* ∧ *b*∈*B*}›
proof (*unfold separating-def*, *intro allI impI*)
fix *F G* :: ‹(*'a*×*'b*) *update* ⇒ *'c update*›
assume [*simp*]: ‹*preregister F*› ‹*preregister G*›
have [*simp*]: ‹*preregister* (λ*x*. *F* (*a* ⊗$_u$ *x*))› **for** *a*
using - ‹*preregister F*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*preregister* (λ*x*. *G* (*a* ⊗$_u$ *x*))› **for** *a*
using - ‹*preregister G*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*preregister* (λ*x*. *F* (*x* ⊗$_u$ *b*))› **for** *b*
using - ‹*preregister F*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*preregister* (λ*x*. *G* (*x* ⊗$_u$ *b*))› **for** *b*
using - ‹*preregister G*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*

assume ‹∀ *x*∈{*a* ⊗$_u$ *b* |*a b*. *a*∈*A* ∧ *b*∈*B*}. *F x* = *G x*›
then have *EQ*: ‹*F* (*a* ⊗$_u$ *b*) = *G* (*a* ⊗$_u$ *b*)› **if** ‹*a* ∈ *A*› **and** ‹*b* ∈ *B*› **for** *a b*
using *that* **by** *auto*
then have ‹*F* (*a* ⊗$_u$ *b*) = *G* (*a* ⊗$_u$ *b*)› **if** ‹*a* ∈ *A*› **for** *a b*
apply (*rule register-eqI*[**where** *A*=*B*, *THEN fun-cong*, **where** *x*=*b*, *rotated* −*1*])

```
    using that by auto
  then have ‹F (a ⊗u b) = G (a ⊗u b)› for a b
    apply (rule register-eqI[where A=A, THEN fun-cong, where x=a, rotated −1])
    by auto
  then show F = G
    apply (rule tensor-extensionality[rotated −1])
    by auto
qed

lemma register-tensor-distrib:
  assumes [simp]: ‹register F› ‹register G› ‹register H› ‹register L›
  shows ‹(F ⊗r G) o (H ⊗r L) = (F o H) ⊗r (G o L)›
  apply (rule tensor-extensionality)
  by (auto intro!: register-comp register-preregister register-tensor-is-register)
```

The following is easier to apply using the *rule*-method than *separating-tensor*

```
lemma separating-tensor':
  fixes A :: ‹'a::type update set› and B :: ‹'b::type update set›
  assumes ‹separating TYPE('c::type) A›
  assumes ‹separating TYPE('c) B›
  assumes ‹C = {a ⊗u b | a b. a∈A ∧ b∈B}›
  shows ‹separating TYPE('c) C›
  using assms
  by (simp add: separating-tensor)

lemma tensor-extensionality3:
  fixes F G :: ‹('a::type×'b::type×'c::type) update ⇒ 'd::type update›
  assumes [simp]: ‹register F› ‹register G›
  assumes ⋀f g h. F (f ⊗u g ⊗u h) = G (f ⊗u g ⊗u h)
  shows F = G
proof (rule register-eqI[where A=‹{a⊗ub⊗uc| a b c. True}›])
  have ‹separating TYPE('d) {b ⊗u c |b c. True}›
    apply (rule separating-tensor'[where A=UNIV and B=UNIV])
    by auto
  then show ‹separating TYPE('d) {a ⊗u b ⊗u c |a b c. True}›
    apply (rule-tac separating-tensor'[where A=UNIV and B=‹{b⊗uc| b c. True}›])
    by auto
  show ‹preregister F› ‹preregister G› by auto
  show ‹x ∈ {a ⊗u b ⊗u c |a b c. True} ⟹ F x = G x› for x
    using assms(3) by auto
qed

lemma tensor-extensionality3':
  fixes F G :: ‹(('a::type×'b::type)×'c::type) update ⇒ 'd::type update›
  assumes [simp]: ‹register F› ‹register G›
  assumes ⋀f g h. F ((f ⊗u g) ⊗u h) = G ((f ⊗u g) ⊗u h)
  shows F = G
proof (rule register-eqI[where A=‹{(a⊗ub)⊗uc| a b c. True}›])
  have ‹separating TYPE('d) {a ⊗u b | a b. True}›
    apply (rule separating-tensor'[where A=UNIV and B=UNIV])
    by auto
  then show ‹separating TYPE('d) {(a ⊗u b) ⊗u c |a b c. True}›
    apply (rule-tac separating-tensor'[where B=UNIV and A=‹{a⊗ub| a b. True}›])
    by auto
  show ‹preregister F› ‹preregister G› by auto
  show ‹x ∈ {(a ⊗u b) ⊗u c |a b c. True} ⟹ F x = G x› for x
    using assms(3) by auto
qed

lemma register-tensor-id[simp]: ‹id ⊗r id = id›
  apply (rule tensor-extensionality)
  by (auto simp add: register-tensor-is-register)
```

6.5 Pairs and compatibility

definition *compatible* :: ‹('a::type update ⇒ 'c::type update)
⇒ ('b::type update ⇒ 'c update) ⇒ bool› **where**
‹*compatible F G* ⟷ *register F* ∧ *register G* ∧ (∀ *a b*. *F a* ∗$_u$ *G b* = *G b* ∗$_u$ *F a*)›

lemma *compatibleI*:
assumes *register F* **and** *register G*
assumes ‹⋀*a b*. (*F a*) ∗$_u$ (*G b*) = (*G b*) ∗$_u$ (*F a*)›
shows *compatible F G*
using *assms* **unfolding** *compatible-def* **by** *simp*

lemma *swap-registers*:
assumes *compatible R S*
shows *R a* ∗$_u$ *S b* = *S b* ∗$_u$ *R a*
using *assms* **unfolding** *compatible-def* **by** *metis*

lemma *compatible-sym*: *compatible x y* ⟹ *compatible y x*
by (*simp add*: *compatible-def*)

lemma *pair-is-register*[*simp*]:
assumes *compatible F G*
shows *register* (*F*; *G*)
by (*metis assms compatible-def register-pair-is-register*)

lemma *register-pair-apply*:
assumes ‹*compatible F G*›
shows ‹(*F*; *G*) (*a* ⊗$_u$ *b*) = (*F a*) ∗$_u$ (*G b*)›
apply (*rule register-pair-apply*)
using *assms* **unfolding** *compatible-def* **by** *metis*+

lemma *register-pair-apply'*:
assumes ‹*compatible F G*›
shows ‹(*F*; *G*) (*a* ⊗$_u$ *b*) = (*G b*) ∗$_u$ (*F a*)›
apply (*subst register-pair-apply*)
using *assms* **by** (*auto simp*: *compatible-def intro*: *register-preregister*)

lemma *compatible-comp-left*[*simp*]: *compatible F G* ⟹ *register H* ⟹ *compatible* (*F* ∘ *H*) *G*
by (*simp add*: *compatible-def*)

lemma *compatible-comp-right*[*simp*]: *compatible F G* ⟹ *register H* ⟹ *compatible F* (*G* ∘ *H*)
by (*simp add*: *compatible-def*)

lemma *compatible-comp-inner*[*simp*]:
compatible F G ⟹ *register H* ⟹ *compatible* (*H* ∘ *F*) (*H* ∘ *G*)
by (*smt* (*verit*, *best*) *comp-apply compatible-def register-comp register-mult*)

lemma *compatible-register1*: ‹*compatible F G* ⟹ *register F*›
by (*simp add*: *compatible-def*)
lemma *compatible-register2*: ‹*compatible F G* ⟹ *register G*›
by (*simp add*: *compatible-def*)

lemma *pair-o-tensor*:
assumes *compatible A B* **and** [*simp*]: ‹*register C*› **and** [*simp*]: ‹*register D*›
shows (*A*; *B*) *o* (*C* ⊗$_r$ *D*) = (*A o C*; *B o D*)
apply (*rule tensor-extensionality*)
using *assms* **by** (*simp-all add*: *register-tensor-is-register register-pair-apply comp-preregister*)

lemma *compatible-tensor-id-update-left*[*simp*]:
fixes *F* :: 'a::type update ⇒ 'c::type update **and** *G* :: 'b::type update ⇒ 'c::type update
assumes *compatible F G*
shows *compatible* (λ*a*. *Some* ⊗$_u$ *F a*) (λ*a*. *Some* ⊗$_u$ *G a*)

using *assms* **apply** (*rule compatible-comp-inner*[*unfolded o-def*])
by *simp*

lemma *compatible-tensor-id-update-right*[*simp*]:
fixes $F :: 'a::type\ update \Rightarrow 'c::type\ update$ **and** $G :: 'b::type\ update \Rightarrow 'c::type\ update$
assumes *compatible F G*
shows *compatible* $(\lambda a.\ F\ a \otimes_u Some)$ $(\lambda a.\ G\ a \otimes_u Some)$
using *assms* **apply** (*rule compatible-comp-inner*[*unfolded o-def*])
by *simp*

lemma *compatible-tensor-id-update-rl*[*simp*]:
assumes *register F* **and** *register G*
shows *compatible* $(\lambda a.\ F\ a \otimes_u Some)$ $(\lambda a.\ Some \otimes_u G\ a)$
apply (*rule compatibleI*)
using *assms* **by** (*auto simp*: *tensor-update-mult*)

lemma *compatible-tensor-id-update-lr*[*simp*]:
assumes *register F* **and** *register G*
shows *compatible* $(\lambda a.\ Some \otimes_u F\ a)$ $(\lambda a.\ G\ a \otimes_u Some)$
apply (*rule compatibleI*)
using *assms* **by** (*auto simp*: *tensor-update-mult*)

lemma *register-comp-pair*:
assumes [*simp*]: ‹*register F*› **and** [*simp*]: ‹*compatible G H*›
shows $(F\ o\ G;\ F\ o\ H) = F\ o\ (G;\ H)$
proof (*rule tensor-extensionality*)
show ‹*preregister* $(F \circ G; F \circ H)$› **and** ‹*preregister* $(F \circ (G;H))$›
by *simp-all*

have [*simp*]: ‹*compatible* $(F\ o\ G)$ $(F\ o\ H)$›
apply (*rule compatible-comp-inner*, *simp*)
by *simp*
then have [*simp*]: ‹*register* $(F \circ G)$› ‹*register* $(F \circ H)$›
unfolding *compatible-def* **by** *auto*
from *assms* **have** [*simp*]: ‹*register G*› ‹*register H*›
unfolding *compatible-def* **by** *auto*
fix *a b*
show ‹$(F \circ G; F \circ H)\ (a \otimes_u b) = (F \circ (G;H))\ (a \otimes_u b)$›
by (*auto simp*: *register-pair-apply register-mult tensor-update-mult*)
qed

lemma *swap-registers-left*:
assumes *compatible R S*
shows $R\ a *_u S\ b *_u c = S\ b *_u R\ a *_u c$
using *assms* **unfolding** *compatible-def* **by** *metis*

lemma *swap-registers-right*:
assumes *compatible R S*
shows $c *_u R\ a *_u S\ b = c *_u S\ b *_u R\ a$
by (*metis assms comp-update-assoc compatible-def*)

lemmas *compatible-ac-rules* = *swap-registers comp-update-assoc*[*symmetric*] *swap-registers-right*

6.6 Fst and Snd

definition *Fst* **where** ‹$Fst\ a = a \otimes_u Some$›
definition *Snd* **where** ‹$Snd\ a = Some \otimes_u a$›

lemma *register-Fst*[*simp*]: ‹*register Fst*›
unfolding *Fst-def* **by** (*rule register-tensor-left*)

lemma *register-Snd*[*simp*]: ‹*register Snd*›
unfolding *Snd-def* **by** (*rule register-tensor-right*)

lemma *compatible-Fst-Snd*[*simp*]: ‹*compatible Fst Snd*›
apply (*rule compatibleI*, *simp*, *simp*)
by (*simp add*: *Fst-def Snd-def tensor-update-mult*)

lemmas *compatible-Snd-Fst*[*simp*] = *compatible-Fst-Snd*[*THEN compatible-sym*]

definition ‹*swap* = (*Snd*; *Fst*)›

lemma *swap-apply*[*simp*]: *swap* (*a* ⊗*u* *b*) = (*b* ⊗*u* *a*)
unfolding *swap-def*
by (*simp add*: *Axioms-Classical.register-pair-apply Fst-def Snd-def tensor-update-mult*)

lemma *swap-o-Fst*: *swap o Fst* = *Snd*
by (*auto simp add*: *Fst-def Snd-def*)
lemma *swap-o-Snd*: *swap o Snd* = *Fst*
by (*auto simp add*: *Fst-def Snd-def*)

lemma *register-swap*[*simp*]: ‹*register swap*›
by (*simp add*: *swap-def*)

lemma *pair-Fst-Snd*: ‹(*Fst*; *Snd*) = *id*›
apply (*rule tensor-extensionality*)
by (*simp-all add*: *register-pair-apply Fst-def Snd-def tensor-update-mult*)

lemma *swap-o-swap*[*simp*]: ‹*swap o swap* = *id*›
by (*metis swap-def compatible-Snd-Fst pair-Fst-Snd register-comp-pair register-swap swap-o-Fst swap-o-Snd*)

lemma *swap-swap*[*simp*]: ‹*swap* (*swap x*) = *x*›
by (*simp add*: *pointfree-idE*)

lemma *inv-swap*[*simp*]: ‹*inv swap* = *swap*›
by (*meson inv-unique-comp swap-o-swap*)

lemma *register-pair-Fst*:
assumes ‹*compatible F G*›
shows ‹(*F*;*G*) *o Fst* = *F*›
using *assms* **by** (*auto intro*!: *ext simp*: *Fst-def register-pair-apply compatible-register2*)

lemma *register-pair-Snd*:
assumes ‹*compatible F G*›
shows ‹(*F*;*G*) *o Snd* = *G*›
using *assms* **by** (*auto intro*!: *ext simp*: *Snd-def register-pair-apply compatible-register1*)

lemma *register-Fst-register-Snd*[*simp*]:
assumes ‹*register F*›
shows ‹(*F o Fst*; *F o Snd*) = *F*›
apply (*rule tensor-extensionality*)
using *assms* **by** (*auto simp*: *register-pair-apply Fst-def Snd-def register-mult tensor-update-mult*)

lemma *register-Snd-register-Fst*[*simp*]:
assumes ‹*register F*›
shows ‹(*F o Snd*; *F o Fst*) = *F o swap*›
apply (*rule tensor-extensionality*)
using *assms* **by** (*auto simp*: *register-pair-apply Fst-def Snd-def register-mult tensor-update-mult*)

lemma *compatible3*[*simp*]:
assumes [*simp*]: *compatible F G* **and** *compatible G H* **and** *compatible F H*
shows *compatible* (*F*; *G*) *H*
proof (*rule compatibleI*)
have [*simp*]: ‹*register F*› ‹*register G*› ‹*register H*›
using *assms compatible-def* **by** *auto*

```
  then have [simp]: ‹preregister F› ‹preregister G› ‹preregister H›
    using register-preregister by blast+
  have [simp]: ‹preregister (λa. (F;G) a *u z)› for z
    apply (rule comp-preregister[unfolded o-def, of ‹(F;G)›])
    by simp-all
  have [simp]: ‹preregister (λa. z *u (F;G) a)› for z
    apply (rule comp-preregister[unfolded o-def, of ‹(F;G)›])
    by simp-all
  have (F; G) (f ⊗u g) *u H h = H h *u (F; G) (f ⊗u g) for f g h
  proof −
    have FH: F f *u H h = H h *u F f
      using assms compatible-def by metis
    have GH: G g *u H h = H h *u G g
      using assms compatible-def by metis
    have ‹(F; G) (f ⊗u g) *u (H h) = F f *u G g *u H h›
      using ‹compatible F G› by (subst register-pair-apply, auto)
    also have ‹... = H h *u F f *u G g›
      using FH GH by (metis comp-update-assoc)
    also have ‹... = H h *u (F; G) (f ⊗u g)›
      using ‹compatible F G› by (subst register-pair-apply, auto simp: comp-update-assoc)
    finally show ?thesis
      by −
  qed
  then show (F; G) fg *u (H h) = (H h) *u (F; G) fg for fg h
    apply (rule-tac tensor-extensionality[THEN fun-cong])
    by auto
  show register H and  register (F; G)
    by simp-all
qed

lemma compatible3'[simp]:
  assumes compatible F G and compatible G H and compatible F H
  shows compatible F (G; H)
  apply (rule compatible-sym)
  apply (rule compatible3)
  using assms by (auto simp: compatible-sym)

lemma pair-o-swap[simp]:
  assumes [simp]: compatible A B
  shows (A; B) o swap = (B; A)
proof (rule tensor-extensionality)
  have [simp]: preregister A preregister B
     apply (metis (no-types, opaque-lifting) assms compatible-register1 register-preregister)
    by (metis (full-types) assms compatible-register2 register-preregister)
  then show ‹preregister ((A; B) ∘ swap)›
    by simp
  show ‹preregister (B; A)›
    by (metis (no-types, lifting) assms compatible-sym register-preregister pair-is-register)
  show ‹((A; B) ∘ swap) (a ⊗u b) = (B; A) (a ⊗u b)› for a b

    apply (simp only: o-def swap-apply)
    apply (subst register-pair-apply, simp)
    apply (subst register-pair-apply, simp add: compatible-sym)
    by (metis (no-types, lifting) assms compatible-def)
qed
```

6.7 Compatibility of register tensor products

```
lemma compatible-register-tensor:
  fixes F :: ‹'a::type update ⇒ 'e::type update› and G :: ‹'b::type update ⇒ 'f::type update›
    and F' :: ‹'c::type update ⇒ 'e update› and G' :: ‹'d::type update ⇒ 'f update›
  assumes [simp]: ‹compatible F F'›
  assumes [simp]: ‹compatible G G'›
```

shows ‹*compatible* (*F* ⊗*r* *G*) (*F'* ⊗*r* *G'*)›
proof −
note [*intro*!] =
comp-preregister[*OF* - *preregister-mult-right*, *unfolded o-def*]
comp-preregister[*OF* - *preregister-mult-left*, *unfolded o-def*]
comp-preregister
register-tensor-is-register
have [*simp*]: ‹*register F*› ‹*register G*› ‹*register F'*› ‹*register G'*›
using *assms compatible-def* **by** *blast+*
have [*simp*]: ‹*register* (*F* ⊗*r* *G*)› ‹*register* (*F'* ⊗*r* *G'*)›
by (*auto simp add*: *register-tensor-def*)
have [*simp*]: ‹*register* (*F*;*F'*)› ‹*register* (*G*;*G'*)›
by *auto*
define *reorder* :: ‹((*'a*×*'b*) × (*'c*×*'d*)) *update* ⇒ ((*'a*×*'c*) × (*'b*×*'d*)) *update*›
where ‹*reorder* = ((*Fst o Fst*; *Snd o Fst*); (*Fst o Snd*; *Snd o Snd*))›
have [*simp*]: ‹*preregister reorder*›
by (*auto simp*: *reorder-def*)
have [*simp*]: ‹*reorder* ((*a* ⊗*u* *b*) ⊗*u* (*c* ⊗*u* *d*)) = ((*a* ⊗*u* *c*) ⊗*u* (*b* ⊗*u* *d*))› **for** *a b c d*
apply (*simp add*: *reorder-def register-pair-apply*)
by (*simp add*: *Fst-def Snd-def tensor-update-mult*)
define Φ **where** ‹Φ *c d* = ((*F*;*F'*) ⊗*r* (*G*;*G'*)) *o reorder o* (λσ. σ ⊗*u* (*c* ⊗*u* *d*))› **for** *c d*
have [*simp*]: ‹*preregister* (Φ *c d*)› **for** *c d*
unfolding Φ-*def*
by (*auto intro*: *register-preregister*)
have ‹Φ *c d* (*a* ⊗*u* *b*) = (*F* ⊗*r* *G*) (*a* ⊗*u* *b*) ∗*u* (*F'* ⊗*r* *G'*) (*c* ⊗*u* *d*)› **for** *a b c d*
unfolding Φ-*def* **by** (*auto simp*: *register-pair-apply tensor-update-mult*)
then have Φ*1*: ‹Φ *c d* σ = (*F* ⊗*r* *G*) σ ∗*u* (*F'* ⊗*r* *G'*) (*c* ⊗*u* *d*)› **for** *c d* σ
apply (*rule-tac fun-cong*[*of* - - σ])
apply (*rule tensor-extensionality*)
by *auto*
have ‹Φ *c d* (*a* ⊗*u* *b*) = (*F'* ⊗*r* *G'*) (*c* ⊗*u* *d*) ∗*u* (*F* ⊗*r* *G*) (*a* ⊗*u* *b*)› **for** *a b c d*
using *assms*
unfolding Φ-*def compatible-def* **by** (*auto simp*: *register-pair-apply tensor-update-mult*)
then have Φ*2*: ‹Φ *c d* σ = (*F'* ⊗*r* *G'*) (*c* ⊗*u* *d*) ∗*u* (*F* ⊗*r* *G*) σ› **for** *c d* σ
apply (*rule-tac fun-cong*[*of* - - σ])
apply (*rule tensor-extensionality*)
by *auto*
from Φ*1* Φ*2* **have** ‹(*F* ⊗*r* *G*) σ ∗*u* (*F'* ⊗*r* *G'*) τ = (*F'* ⊗*r* *G'*) τ ∗*u* (*F* ⊗*r* *G*) σ› **for** τ σ
apply (*rule-tac fun-cong*[*of* - - τ])
apply (*rule tensor-extensionality*)
by *auto*
then show *?thesis*
apply (*rule compatibleI*[*rotated* −*1*])
by *auto*
qed

6.8 Associativity of the tensor product

definition *assoc* :: ‹((*'a*::*type*×*'b*::*type*)×*'c*::*type*) *update* ⇒ (*'a*×(*'b*×*'c*)) *update*› **where**
‹*assoc* = ((*Fst*; *Snd o Fst*); *Snd o Snd*)›

lemma *assoc-is-hom*[*simp*]: ‹*preregister assoc*›
by (*auto simp*: *assoc-def*)

lemma *assoc-apply*[*simp*]: ‹*assoc* ((*a* ⊗*u* *b*) ⊗*u* *c*) = (*a* ⊗*u* (*b* ⊗*u* *c*))›
by (*auto simp*: *assoc-def register-pair-apply Fst-def Snd-def tensor-update-mult*)

definition *assoc'* :: ‹(*'a*×(*'b*×*'c*)) *update* ⇒ ((*'a*::*type*×*'b*::*type*)×*'c*::*type*) *update*› **where**
‹*assoc'* = (*Fst o Fst*; (*Fst o Snd*; *Snd*))›

lemma *assoc'-is-hom*[*simp*]: ‹*preregister assoc'*›
by (*auto simp*: *assoc'-def*)

lemma *assoc′-apply*[*simp*]: ‹*assoc′* (*a* ⊗*u* (*b* ⊗*u* *c*)) = ((*a* ⊗*u* *b*) ⊗*u* *c*)›
by (*auto simp*: *assoc′-def register-pair-apply Fst-def Snd-def tensor-update-mult*)

lemma *register-assoc*[*simp*]: ‹*register assoc*›
unfolding *assoc-def*
by *force*

lemma *register-assoc′*[*simp*]: ‹*register assoc′*›
unfolding *assoc′-def*
by *force*

lemma *pair-o-assoc*[*simp*]:
assumes [*simp*]: ‹*compatible F G*› ‹*compatible G H*› ‹*compatible F H*›
shows ‹(*F*; (*G*; *H*)) ∘ *assoc* = ((*F*; *G*); *H*)›
proof (*rule tensor-extensionality3′*)
show ‹*register* ((*F*; (*G*; *H*)) ∘ *assoc*)›
by *simp*
show ‹*register* ((*F*; *G*); *H*)›
by *simp*
show ‹((*F*; (*G*; *H*)) ∘ *assoc*) ((*f* ⊗*u* *g*) ⊗*u* *h*) = ((*F*; *G*); *H*) ((*f* ⊗*u* *g*) ⊗*u* *h*)› **for** *f g h*
by (*simp add*: *register-pair-apply assoc-apply comp-update-assoc*)
qed

lemma *pair-o-assoc′*[*simp*]:
assumes [*simp*]: ‹*compatible F G*› ‹*compatible G H*› ‹*compatible F H*›
shows ‹((*F*; *G*); *H*) ∘ *assoc′* = (*F*; (*G*; *H*))›
proof (*rule tensor-extensionality3*)
show ‹*register* (((*F*; *G*); *H*) ∘ *assoc′*)›
by *simp*
show ‹*register* (*F*; (*G*; *H*))›
by *simp*
show ‹(((*F*; *G*); *H*) ∘ *assoc′*) (*f* ⊗*u* *g* ⊗*u* *h*) = (*F*; (*G*; *H*)) (*f* ⊗*u* *g* ⊗*u* *h*)› **for** *f g h*
by (*simp add*: *register-pair-apply assoc′-apply comp-update-assoc*)
qed

lemma *assoc′-o-assoc*[*simp*]: ‹*assoc′ o assoc* = *id*›
apply (*rule tensor-extensionality3′*)
by *auto*

lemma *assoc′-assoc*[*simp*]: ‹*assoc′* (*assoc x*) = *x*›
by (*simp add*: *pointfree-idE*)

lemma *assoc-o-assoc′*[*simp*]: ‹*assoc o assoc′* = *id*›
apply (*rule tensor-extensionality3*)
by *auto*

lemma *assoc-assoc′*[*simp*]: ‹*assoc* (*assoc′ x*) = *x*›
by (*simp add*: *pointfree-idE*)

lemma *inv-assoc*[*simp*]: ‹*inv assoc* = *assoc′*›
using *assoc′-o-assoc assoc-o-assoc′ inv-unique-comp* **by** *blast*

lemma *inv-assoc′*[*simp*]: ‹*inv assoc′* = *assoc*›
by (*simp add*: *inv-equality*)

lemma *bij-assoc*[*simp*]: ‹*bij assoc*›
using *assoc′-o-assoc assoc-o-assoc′ o-bij* **by** *blast*

lemma *bij-assoc′*[*simp*]: ‹*bij assoc′*›
using *assoc′-o-assoc assoc-o-assoc′ o-bij* **by** *blast*

6.9 Iso-registers

definition ‹*iso-register F* ⟷ *register F* ∧ (∃ *G*. *register G* ∧ *F o G* = *id* ∧ *G o F* = *id*)›
for *F* :: ‹-::*type update* ⇒ -::*type update*›

lemma *iso-registerI*:
assumes ‹*register F*› ‹*register G*› ‹*F o G* = *id*› ‹*G o F* = *id*›
shows ‹*iso-register F*›
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *iso-register-def* **by** *blast*

lemma *iso-register-inv*: ‹*iso-register F* ⟹ *iso-register* (*inv F*)›
by (*metis inv-unique-comp iso-register-def*)

lemma *iso-register-inv-comp1*: ‹*iso-register F* ⟹ *inv F o F* = *id*›
using *inv-unique-comp iso-register-def* **by** *blast*

lemma *iso-register-inv-comp2*: ‹*iso-register F* ⟹ *F o inv F* = *id*›
using *inv-unique-comp iso-register-def* **by** *blast*

lemma *iso-register-id*[*simp*]: ‹*iso-register id*›
by (*simp add*: *iso-register-def*)

lemma *iso-register-is-register*: ‹*iso-register F* ⟹ *register F*›
using *iso-register-def* **by** *blast*

lemma *iso-register-comp*[*simp*]:
assumes [*simp*]: ‹*iso-register F*› ‹*iso-register G*›
shows ‹*iso-register* (*F o G*)›
proof −
from *assms* **obtain** *F′ G′* **where** [*simp*]: ‹*register F′*› ‹*register G′*› ‹*F o F′* = *id*› ‹*F′ o F* = *id*›
‹*G o G′* = *id*› ‹*G′ o G* = *id*›
by (*meson iso-register-def*)
have *1*: ‹*F* ∘ *G* ∘ (*G′* ∘ *F′*) = *id*›
by (*metis* ‹*F* ∘ *F′* = *id*› ‹*G* ∘ *G′* = *id*› *fcomp-assoc fcomp-comp id-fcomp*)
have *2*: ‹*G′* ∘ *F′* ∘ (*F* ∘ *G*) = *id*›
by (*metis* (*no-types*, *lifting*) ‹*F* ∘ *F′* = *id*› ‹*F′* ∘ *F* = *id*› ‹*G′* ∘ *G* = *id*› *fun.map-comp inj-iff inv-unique-comp o-inv-o-cancel*)

show *?thesis*
apply (*rule iso-registerI*[**where** *G*=‹*G′ o F′*›])
using *1 2* **by** (*auto simp*: *register-tensor-is-register iso-register-is-register register-tensor-distrib*)
qed

lemma *iso-register-tensor-is-iso-register*[*simp*]:
assumes [*simp*]: ‹*iso-register F*› ‹*iso-register G*›
shows ‹*iso-register* (*F* ⊗*r G*)›
proof −
from *assms* **obtain** *F′ G′* **where** [*simp*]: ‹*register F′*› ‹*register G′*› ‹*F o F′* = *id*› ‹*F′ o F* = *id*›
‹*G o G′* = *id*› ‹*G′ o G* = *id*›
by (*meson iso-register-def*)
show *?thesis*
apply (*rule iso-registerI*[**where** *G*=‹*F′* ⊗*r G′*›])
by (*auto simp*: *register-tensor-is-register iso-register-is-register register-tensor-distrib*)
qed

lemma *iso-register-bij*: ‹*iso-register F* ⟹ *bij F*›
using *iso-register-def o-bij* **by** *auto*

lemma *inv-register-tensor*[*simp*]:
assumes [*simp*]: ‹*iso-register F*› ‹*iso-register G*›
shows ‹*inv* (*F* ⊗*r G*) = *inv F* ⊗*r inv G*›
apply (*auto intro*!: *inj-imp-inv-eq bij-is-inj iso-register-bij*

```
    simp: register-tensor-distrib[unfolded o-def, THEN fun-cong] iso-register-is-register
      iso-register-inv bij-is-surj iso-register-bij surj-f-inv-f)
  by (metis eq-id-iff register-tensor-id)

lemma iso-register-swap[simp]: ‹iso-register swap›
  apply (rule iso-registerI[of - swap])
  by auto

lemma iso-register-assoc[simp]: ‹iso-register assoc›
  apply (rule iso-registerI[of - assoc'])
  by auto

lemma iso-register-assoc'[simp]: ‹iso-register assoc'›
  apply (rule iso-registerI[of - assoc])
  by auto

definition ‹equivalent-registers F G ⟷ (register F ∧ (∃ I. iso-register I ∧ F o I = G))›
  for F G :: ‹-::type update ⇒ -::type update›

lemma iso-register-equivalent-id[simp]: ‹equivalent-registers id F ⟷ iso-register F›
  by (simp add: equivalent-registers-def)

lemma equivalent-registersI:
  assumes ‹register F›
  assumes ‹iso-register I›
  assumes ‹F o I = G›
  shows ‹equivalent-registers F G›
  using assms unfolding equivalent-registers-def by blast

lemma equivalent-registers-refl: ‹equivalent-registers F F› if ‹register F›
  using that by (auto intro!: exI[of - id] simp: equivalent-registers-def)

lemma equivalent-registers-register-left: ‹equivalent-registers F G ⟹ register F›
  using equivalent-registers-def by auto

lemma equivalent-registers-register-right: ‹register G› if ‹equivalent-registers F G›
  by (metis equivalent-registers-def iso-register-def register-comp that)

lemma equivalent-registers-sym:
  assumes ‹equivalent-registers F G›
  shows ‹equivalent-registers G F›
 by (smt (verit) assms comp-id equivalent-registers-def equivalent-registers-register-right fun.map-comp iso-register-def)

lemma equivalent-registers-trans[trans]:
  assumes ‹equivalent-registers F G› and ‹equivalent-registers G H›
  shows ‹equivalent-registers F H›
proof −
  from assms have [simp]: ‹register F› ‹register G›
    by (auto simp: equivalent-registers-def)
  from assms(1) obtain I where [simp]: ‹iso-register I› and ‹F o I = G›
    using equivalent-registers-def by blast
  from assms(2) obtain J where [simp]: ‹iso-register J› and ‹G o J = H›
    using equivalent-registers-def by blast
  have ‹register F›
    by (auto simp: equivalent-registers-def)
  moreover have ‹iso-register (I o J)›
    using ‹iso-register I› ‹iso-register J› iso-register-comp by blast
  moreover have ‹F o (I o J) = H›
    by (simp add: ‹F ∘ I = G› ‹G ∘ J = H› o-assoc)
  ultimately show ?thesis
    by (rule equivalent-registersI)
qed
```

lemma *equivalent-registers-assoc*[*simp*]:
assumes [*simp*]: ‹*compatible F G*› ‹*compatible F H*› ‹*compatible G H*›
shows ‹*equivalent-registers* (*F*;(*G*;*H*)) ((*F*;*G*);*H*)›
apply (*rule equivalent-registersI*[**where** *I*=*assoc*])
by *auto*

lemma *equivalent-registers-pair-right*:
assumes [*simp*]: ‹*compatible F G*›
assumes *eq*: ‹*equivalent-registers G H*›
shows ‹*equivalent-registers* (*F*;*G*) (*F*;*H*)›
proof −
from *eq* **obtain** *I* **where** [*simp*]: ‹*iso-register I*› **and** ‹*G o I* = *H*›
by (*metis equivalent-registers-def*)
then have ∗: ‹(*F*;*G*) ∘ (*id* ⊗*r I*) = (*F*;*H*)›
by (*auto intro*!: *tensor-extensionality register-comp register-preregister register-tensor-is-register*
simp: *register-pair-apply iso-register-is-register*)
show *?thesis*
apply (*rule equivalent-registersI*[**where** *I*=‹*id* ⊗*r I*›])
using ∗ **by** (*auto intro*!: *iso-register-tensor-is-iso-register*)
qed

lemma *equivalent-registers-pair-left*:
assumes [*simp*]: ‹*compatible F G*›
assumes *eq*: ‹*equivalent-registers F H*›
shows ‹*equivalent-registers* (*F*;*G*) (*H*;*G*)›
proof −
from *eq* **obtain** *I* **where** [*simp*]: ‹*iso-register I*› **and** ‹*F o I* = *H*›
by (*metis equivalent-registers-def*)
then have ∗: ‹(*F*;*G*) ∘ (*I* ⊗*r id*) = (*H*;*G*)›
by (*auto intro*!: *tensor-extensionality register-comp register-preregister register-tensor-is-register*
simp: *register-pair-apply iso-register-is-register*)
show *?thesis*
apply (*rule equivalent-registersI*[**where** *I*=‹*I* ⊗*r id*›])
using ∗ **by** (*auto intro*!: *iso-register-tensor-is-iso-register*)
qed

lemma *equivalent-registers-comp*:
assumes ‹*register H*›
assumes ‹*equivalent-registers F G*›
shows ‹*equivalent-registers* (*H o F*) (*H o G*)›
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *comp-assoc equivalent-registers-def register-comp*)

lemma *equivalent-registers-compatible1*:
assumes ‹*compatible F G*›
assumes ‹*equivalent-registers F F'*›
shows ‹*compatible F' G*›
by (*metis assms*(*1*) *assms*(*2*) *compatible-comp-left equivalent-registers-def iso-register-is-register*)

lemma *equivalent-registers-compatible2*:
assumes ‹*compatible F G*›
assumes ‹*equivalent-registers G G'*›
shows ‹*compatible F G'*›
by (*metis assms*(*1*) *assms*(*2*) *compatible-comp-right equivalent-registers-def iso-register-is-register*)

6.10 Compatibility simplification

The simproc *compatibility-warn* produces helpful warnings for subgoals of the form *compatible x y* that are probably unsolvable due to missing declarations of variable compatibility facts. Same for subgoals of the form *register x*.

simproc-setup *compatibility-warn* (*compatible x y* | *register x*) = ‹
let val thy-string = *Markup.markup* (*Theory.get-markup* ***theory***) (*Context.theory-base-name* ***theory***)
in
fn m => fn ctxt => fn ct => let

```
  val (x,y) = case Thm.term-of ct of
                Const(const-name‹compatible›,- ) $ x $ y => (x, SOME y)
              | Const(const-name‹register›,- ) $ x => (x, NONE)
  val str : string lazy = Lazy.lazy (fn () => Syntax.string-of-term ctxt (Thm.term-of ct))
 fun w msg = warning (msg ^\n(Disable these warnings with: using [[simproc del: ^thy-string^.compatibility-warn]]))
  val - = case (x,y) of
       (Free(n,T), SOME (Free(n',T'))) =>
          if String.isPrefix : n orelse String.isPrefix : n' then
                  w (Simplification subgoal  ^ Lazy.force str ^  contains a bound variable.\n ^
                   Try to add some assumptions that makes this goal solvable by the simplifier)
          else if n=n' then (if T=T' then ()
                       else w (In simplification subgoal  ^ Lazy.force str ^
                            , variables have same name and different types.\n ^
                            Probably something is wrong.))
                  else w (Simplification subgoal  ^ Lazy.force str ^
                           occurred but cannot be solved.\n ^
                          Please add assumption/fact  [simp]: ‹ ^ Lazy.force str ^
                          ›  somewhere.)
     | (Free(n,T), NONE) =>
          if String.isPrefix : n then
                  w (Simplification subgoal ' ^ Lazy.force str ^ ' contains a bound variable.\n ^
                   Try to add some assumptions that makes this goal solvable by the simplifier)
          else w (Simplification subgoal  ^ Lazy.force str ^  occurred but cannot be solved.\n ^
                 Please add assumption/fact  [simp]: ‹ ^ Lazy.force str ^ ›  somewhere.)
     | - => ()
  in NONE end
end›


named-theorems register-attribute-rule-immediate
named-theorems register-attribute-rule

lemmas [register-attribute-rule] = conjunct1 conjunct2 iso-register-is-register iso-register-is-register[OF iso-register-inv]
lemmas [register-attribute-rule-immediate] = compatible-sym compatible-register1 compatible-register2
  asm-rl[of ‹compatible - ›] asm-rl[of ‹iso-register ›] asm-rl[of ‹register ›] iso-register-inv
```

The following declares an attribute [*register*]. When the attribute is applied to a fact of the form *register F*, *iso-register F*, *compatible F G* or a conjunction of these, then those facts are added to the simplifier together with some derived theorems (e.g., *compatible F G* also adds *register F*).

In theory *Laws-Complement*, support for *is-unit-register F* and *complements F G* is added to this attribute.

```
setup ‹
let
fun add thm results =
  Net.insert-term (K true) (Thm.concl-of thm, thm) results
  handle Net.INSERT => results
fun try-rule f thm rule state = case SOME (rule OF [thm]) handle THM - => NONE  of
  NONE => state | SOME th => f th state
fun collect (rules,rules-immediate) thm results =
  results |> fold (try-rule add thm) rules-immediate |> fold (try-rule (collect (rules,rules-immediate)) thm) rules
fun declare thm context = let
  val ctxt = Context.proof-of context
  val rules = Named-Theorems.get ctxt @{named-theorems register-attribute-rule}
  val rules-immediate = Named-Theorems.get ctxt @{named-theorems register-attribute-rule-immediate}
  val thms = collect (rules,rules-immediate) thm Net.empty |> Net.entries
  (* val - = print thms *)
  in Simplifier.map-ss (fn ctxt => ctxt addsimps thms) context end
in
Attrib.setup binding‹register›
 (Scan.succeed (Thm.declaration-attribute declare))
  Add register−related rules to the simplifier
end
›
```

6.11 Notation

no-notation *map-comp* (**infixl** $*_u$ *55*)
no-notation *tensor-update* (**infixr** $\otimes_u$ *70*)

bundle *register-syntax* **begin**
notation *register-tensor* (**infixr** $\otimes_r$ *70*)
notation *register-pair* ('(-;-'))
end

end

7 Miscellaneous facts

This theory proves various facts that are not directly related to this developments but do not occur in the imported theories.

theory *Misc*
  imports
    Complex-Bounded-Operators.Cblinfun-Code
    HOL−Library.Z2
    Jordan-Normal-Form.Matrix
    Hilbert-Space-Tensor-Product.Weak-Star-Topology
begin

— Remove notation that collides with the notation we use
no-notation *Order.top* ($\top_1$)
unbundle *no m-inv-syntax*
unbundle *no vec-syntax*
unbundle *no inner-syntax*

— Import notation from Bounded Operator and Jordan Normal Form libraries
unbundle *cblinfun-syntax*
unbundle *jnf-syntax*

The following declares the ML antiquotation `fact`. In ML code, `@{fact f}` for a theorem/fact name f is replaced by an ML string containing a printable(!) representation of fact. (I.e., if you print that string using writeln, the user can ctrl-click on it.)

This is useful when constructing diagnostic messages in ML code, e.g., `"Use the theorem " ^ @{fact thmname} ^ "here."`

setup ‹*ML-Antiquotation.inline-embedded* ***binding*** ‹*fact*›
((*Args.context* −− *Scan.lift Args.name-position*) >> (*fn* (*ctxt*,*namepos*) => *let*
  val facts = *Proof-Context.facts-of ctxt*
  val fullname = *Facts.check* (*Context.Proof ctxt*) *facts namepos*
  val (*markup*, *shortname*) = *Proof-Context.markup-extern-fact ctxt fullname*
  val string = *Markup.markups markup shortname*
  in ML-Syntax.print-string string end
))
›

instantiation *bit* :: *enum* **begin**
definition *enum-bit* = [*0*::*bit*,*1*]
definition *enum-all-bit P* ⟷ *P* (*0*::*bit*) ∧ *P 1*
definition *enum-ex-bit P* ⟷ *P* (*0*::*bit*) ∨ *P 1*
instance
proof *intro-classes*
  show ‹(*UNIV* :: *bit set*) = *set enum-class.enum*›
    by (*auto simp*: *enum-bit-def*)
  show ‹*distinct* (*enum-class.enum* :: *bit list*)›
    by (*auto simp*: *enum-bit-def*)
  show ‹*enum-class.enum-all P* = *Ball UNIV P*› **for** *P* :: ‹*bit* ⇒ *bool*›
    apply (*simp add*: *enum-bit-def enum-all-bit-def enum-ex-bit-def*)

```
    by (metis bit.exhaust)
  show ‹enum-class.enum-ex P = Bex UNIV P› for P :: ‹bit ⇒ bool›
    apply (simp add: enum-bit-def enum-all-bit-def enum-ex-bit-def)
    by (metis bit.exhaust)
qed
end

lemma card-bit[simp]: CARD(bit) = 2
  using card-2-iff' by force

instantiation bit :: card-UNIV begin
definition finite-UNIV = Phantom(bit) True
definition card-UNIV = Phantom(bit) 2
instance
  apply intro-classes
  by (simp-all add: finite-UNIV-bit-def card-UNIV-bit-def)
end

lemma mat-of-rows-list-carrier[simp]:
  mat-of-rows-list n vs ∈ carrier-mat (length vs) n
  dim-row (mat-of-rows-list n vs) = length vs
  dim-col (mat-of-rows-list n vs) = n
  unfolding mat-of-rows-list-def by auto

lemma mat-of-rows-list-carrier2xn[iff]:
  mat-of-rows-list n [a,b] ∈ carrier-mat 2 n
  by auto

lemma mat-of-rows-list-carrier4xn[iff]:
  mat-of-rows-list n [a,b,c,d] ∈ carrier-mat 4 n
  by auto
```

```
lemma prod-cases3' [cases type]:
  obtains (fields) a b c where y = ((a, b), c)
  by (cases y, case-tac a) blast
```

We define the following abbreviations:

- *mutually f* (x_1,x_2,...,x_n) expands to the conjuction of all *f* x_i x_j with $i \neq j$.
- *each f* (x_1,x_2,...,x_n) expands to the conjuction of all *f* x_i.

```
syntax -mutually :: 'a ⇒ args ⇒ 'b (mutually - '(-'))
syntax -mutually2 :: 'a ⇒ 'b ⇒ args ⇒ args ⇒ 'c

translations mutually f (x) => CONST True
translations mutually f (-args x y) => f x y ∧ f y x
translations mutually f (-args x (-args x' xs)) => -mutually2 f x (-args x' xs) (-args x' xs)
translations -mutually2 f x y zs => f x y ∧ f y x ∧ -mutually f zs
translations -mutually2 f x (-args y ys) zs => f x y ∧ f y x ∧ -mutually2 f x ys zs

syntax -each :: 'a ⇒ args ⇒ 'b (each - '(-'))
translations each f (x) => f x
translations -each f (-args x xs) => f x ∧ -each f xs
```

lemma *dim-col-mat-adjoint*[*simp*]: *dim-col* (*mat-adjoint m*) = *dim-row m*
unfolding *mat-adjoint-def* **by** *simp*
lemma *dim-row-mat-adjoint*[*simp*]: *dim-row* (*mat-adjoint m*) = *dim-col m*
unfolding *mat-adjoint-def* **by** *simp*

lemma *invI*:
assumes ‹*inj f*›
assumes ‹*x* = *f y*›
shows ‹*inv f x* = *y*›
by (*simp add*: *assms*(*1*) *assms*(*2*))

instantiation *prod* :: (*default*,*default*) *default* **begin**
definition ‹*default-prod* = (*default*, *default*)›
instance..
end

instance *bit* :: *default***..**

lemma *surj-from-comp*:
assumes ‹*surj* (*g o f*)›
assumes ‹*inj g*›
shows ‹*surj f*›
by (*metis assms*(*1*) *assms*(*2*) *f-inv-into-f fun.set-map inj-image-mem-iff iso-tuple-UNIV-I surj-iff-all*)

lemma *double-exists*: ‹(∃ *x y*. *Q x y*) ⟷ (∃ *z*. *Q* (*fst z*) (*snd z*))›
by *simp*

lemma *Ex-iffI*:
assumes ‹⋀*x*. *P x* ⟹ *Q* (*f x*)›
assumes ‹⋀*x*. *Q x* ⟹ *P* (*g x*)›
shows ‹*Ex P* ⟷ *Ex Q*›
using *assms*(*1*) *assms*(*2*) **by** *auto*

lemma *cspan-space-as-set*[*simp*]: ‹*cspan* (*space-as-set X*) = *space-as-set X*›
by *auto*

thm *lift-cblinfun-comp*
lemma *lift-cblinfun-comp2*:
assumes ‹*a* o_{CL} *b* = *c*›
shows ‹(*d* o_{CL} *a*) o_{CL} *b* = *d* o_{CL} *c*›
by (*simp add*: *assms cblinfun-assoc-right*)
lemmas *lift-cblinfun-comp* = *lift-cblinfun-comp lift-cblinfun-comp2*

unbundle *no cblinfun-syntax*
unbundle *no jnf-syntax*

end

8 Derived facts about classical registers

theory *Classical-Extra*
imports *Laws-Classical Misc*
begin

lemma *register-from-getter-setter-of-getter-setter*[*simp*]: ‹*register-from-getter-setter* (*getter F*) (*setter F*) = *F*›
if ‹*register F*›
by (*metis getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter that*)

lemma *valid-getter-setter-getter-setter*[*simp*]: ‹*valid-getter-setter* (*getter F*) (*setter F*)› **if** ‹*register F*›
by (*metis getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter that*)

lemma *register-register-from-getter-setter*[*simp*]: ‹*register* (*register-from-getter-setter g s*)› **if** ‹*valid-getter-setter g s*›
using *register-def that* **by** *blast*

definition ‹*total-fun f* = (∀ *x*. *f x* ≠ *None*)›

lemma *register-total*:
assumes ‹*register F*›
assumes ‹*total-fun a*›
shows ‹*total-fun* (*F a*)›
using *assms*
by (*auto simp*: *register-def total-fun-def register-from-getter-setter-def option.case-eq-if*)

lemma *register-apply*:
assumes ‹*register F*›
shows ‹*Some o register-apply F a* = *F* (*Some o a*)›
proof −
have ‹*total-fun* (*F* (*Some o a*))›
using *assms* **apply** (*rule register-total*)
by (*auto simp*: *total-fun-def*)
then show *?thesis*
by (*auto simp*: *register-apply-def dom-def total-fun-def*)
qed

lemma *register-empty*:
assumes ‹*preregister F*›
shows ‹*F Map.empty* = *Map.empty*›
using *assms* **unfolding** *preregister-def* **by** *auto*

lemma *compatible-setter*:
fixes *F* :: ‹('*a*,'*c*) *preregister*› **and** *G* :: ‹('*b*,'*c*) *preregister*›
assumes [*simp*]: ‹*register F*› ‹*register G*›
shows ‹*compatible F G* ⟷ (∀ *a b*. *setter F a o setter G b* = *setter G b o setter F a*)›
proof (*intro allI iffI*)
fix *a b*
assume ‹*compatible F G*›
then show ‹*setter F a o setter G b* = *setter G b o setter F a*›
apply (*rule-tac compatible-setter*)
unfolding *compatible-def* **by** *auto*
next
assume *commute*[*rule-format*, *THEN fun-cong*, *unfolded o-def*]: ‹∀ *a b*. *setter F a* ∘ *setter G b* = *setter G b* ∘ *setter F a*›

have ‹*valid-getter-setter* (*getter F*) (*setter F*)›
by *auto*
then have ‹*register-from-getter-setter* (*getter F*) (*setter F*) *a* ∘$_m$ *register-from-getter-setter* (*getter G*) (*setter G*) *b* =
register-from-getter-setter (*getter G*) (*setter G*) *b* ∘$_m$ *register-from-getter-setter* (*getter F*) (*setter F*) *a*› **for** *a b*
unfolding *register-from-getter-setter-def valid-getter-setter-def*
apply (*rule-tac ext*)
by (*smt* (*verit*) *assms*(*2*) *commute map-comp-def option.case*(*2*) *option.case-eq-if valid-getter-setter-def valid-getter-setter-getter-setter*)
then have ‹*F a* ∘$_m$ *G b* = *G b* ∘$_m$ *F a*› **for** *a b*
apply (*subst* (*2*) *register-from-getter-setter-of-getter-setter*[*symmetric*, *of F*], *simp*)
apply (*subst* (*1*) *register-from-getter-setter-of-getter-setter*[*symmetric*, *of F*], *simp*)
apply (*subst* (*2*) *register-from-getter-setter-of-getter-setter*[*symmetric*, *of G*], *simp*)
apply (*subst* (*1*) *register-from-getter-setter-of-getter-setter*[*symmetric*, *of G*], *simp*)
by *simp*
then show ‹*compatible F G*›

unfolding *compatible-def* **by** *auto*
qed

lemma *register-from-getter-setter-compatibleI*[*intro*]:
assumes [*simp*]: ‹*valid-getter-setter g s*› ‹*valid-getter-setter g′ s′*›
assumes ‹⋀*x y m*. *s x* (*s′ y m*) = *s′ y* (*s x m*)›
shows ‹*compatible* (*register-from-getter-setter g s*) (*register-from-getter-setter g′ s′*)›
apply (*subst compatible-setter*)
using *assms* **by** *auto*

lemma *separating-update1*:
‹*separating TYPE*(-) {*update1 x y* | *x y*. *True*}›
by (*smt* (*verit*) *mem-Collect-eq separating-def update1-extensionality*)

definition *permutation-register* (*p*::*′b*⇒*′a*) = *register-from-getter-setter p* (λ*a* -. *inv p a*)

lemma *permutation-register-register*[*simp*]:
fixes *p* :: *′b* ⇒ *′a*
assumes [*simp*]: *bij p*
shows *register* (*permutation-register p*)
using *assms*
by (*auto intro*!: *register-register-from-getter-setter surj-f-inv-f*[*of p*] *bij-betw-imp-surj-on*
simp: *permutation-register-def valid-getter-setter-def bij-inv-eq-iff*)

lemma *getter-permutation-register*: ‹*bij p* ⟹ *getter* (*permutation-register p*) = *p*›
by (*smt* (*verit*, *ccfv-threshold*) *bij-inv-eq-iff getter-of-register-from-getter-setter permutation-register-def valid-getter-setter-def*)

lemma *setter-permutation-register*: ‹*bij p* ⟹ *setter* (*permutation-register p*) *a m* = *inv p a*›
by (*metis bij-inv-eq-iff getter-permutation-register permutation-register-register valid-getter-setter-def valid-getter-setter-getter-sett*

definition *empty-var* :: ‹*′a*::{*CARD-1*} *update* ⇒ *′b update*› **where**
empty-var = *register-from-getter-setter* (λ-. *undefined*) (λ- *m*. *m*)

lemma *valid-empty-var*[*simp*]: ‹*valid-getter-setter* (λ-. (*undefined*::-::*CARD-1*)) (λ- *m*. *m*)›
by (*simp add*: *valid-getter-setter-def*)

lemma *register-empty-var*[*simp*]: ‹*register empty-var*›
using *empty-var-def register-def valid-empty-var* **by** *blast*

lemma *getter-empty-var*[*simp*]: ‹*getter empty-var m* = *undefined*›
by (*rule everything-the-same*)

lemma *setter-empty-var*[*simp*]: ‹*setter empty-var a m* = *m*›
by (*simp add*: *empty-var-def setter-of-register-from-getter-setter*)

lemma *empty-var-compatible*[*simp*]: ‹*compatible empty-var X*› **if** [*simp*]: ‹*register X*›
apply (*subst compatible-setter*) **by** *auto*

lemma *empty-var-compatible′*[*simp*]: ‹*register X* ⟹ *compatible X empty-var*›
using *compatible-sym empty-var-compatible* **by** *blast*

Example **record** *memory* =
x :: *int*∗*int*
y :: *nat*

definition *X* = *register-from-getter-setter x* (λ*a b*. *b*(|*x*:=*a*|))
definition *Y* = *register-from-getter-setter y* (λ*a b*. *b*(|*y*:=*a*|))

lemma *validX*[*simp*]: ‹*valid-getter-setter x* (λ*a b*. *b*(|*x*:=*a*|))›
unfolding *valid-getter-setter-def* **by** *auto*

lemma *registerX*[*simp*]: ‹*register X*›
using *X-def register-def validX* **by** *blast*

lemma *validY*[*simp*]: ‹*valid-getter-setter y* (λ*a b*. *b*(|*y*:=*a*|))›
unfolding *valid-getter-setter-def* **by** *auto*

lemma *registerY*[*simp*]: ‹*register Y*›
using *Y-def register-def validY* **by** *blast*

lemma *compatibleXY*[*simp*]: ‹*compatible X Y*›
unfolding *X-def Y-def* **by** *auto*

hide-const (**open**) *x y x-update y-update X Y*

end

9 Quantum instantiation of registers

theory *Axioms-Quantum*
imports *Jordan-Normal-Form.Matrix-Impl HOL−Library.Rewrite*
Complex-Bounded-Operators.Complex-L2
Hilbert-Space-Tensor-Product.Hilbert-Space-Tensor-Product
Hilbert-Space-Tensor-Product.Weak-Star-Topology
Hilbert-Space-Tensor-Product.Partial-Trace
Hilbert-Space-Tensor-Product.Von-Neumann-Algebras
With-Type.With-Type
Misc
begin

unbundle *cblinfun-syntax*
unbundle *no m-inv-syntax*

type-synonym *'a update* = ‹(*'a ell2*, *'a ell2*) *cblinfun*›

definition *preregister* :: ‹(*'a update* ⇒ *'b update*) ⇒ *bool*› **where**
‹*preregister F* ⟷ *bounded-clinear F* ∧ *continuous-map weak-star-topology weak-star-topology F*›

lemma *preregister-mult-right*: ‹*preregister* (λ*a*. *a* o_{CL} *z*)›
by (*auto simp add*: *bounded-cbilinear.bounded-clinear-left bounded-cbilinear-cblinfun-compose*
preregister-def continuous-map-right-comp-weak-star)

lemma *preregister-mult-left*: ‹*preregister* (λ*a*. *z* o_{CL} *a*)›
by (*auto simp add*: *bounded-cbilinear.bounded-clinear-right bounded-cbilinear-cblinfun-compose*
preregister-def continuous-map-left-comp-weak-star)

lemma *comp-preregister*: *preregister F* ⟹ *preregister G* ⟹ *preregister* (*G* ∘ *F*)
by (*auto simp add*: *preregister-def continuous-map-compose comp-bounded-clinear*)

lemma *id-preregister*: ‹*preregister id*›
unfolding *preregister-def* **by** *auto*

lemma *tensor-extensionality*:
‹*preregister F* ⟹ *preregister G* ⟹ (⋀*a b*. *F* (*tensor-op a b*) = *G* (*tensor-op a b*)) ⟹ *F* = *G*›
apply (*rule weak-star-clinear-eq-butterfly-ketI*)
by (*auto intro*!: *bounded-clinear.clinear simp*: *preregister-def simp flip*: *tensor-ell2-ket tensor-butterfly*)

definition *register* :: ‹(*'a update* ⇒ *'b update*) ⇒ *bool*› **where**
register F ⟷
bounded-clinear F
∧ *continuous-map weak-star-topology weak-star-topology F*
∧ *F id-cblinfun* = *id-cblinfun*
∧ (∀ *a b*. *F*(*a* o_{CL} *b*) = *F a* o_{CL} *F b*)

$\wedge$ ($\forall a.\ F\ (a*) = (F\ a)*$)

lemma *register-of-id*: ‹*register F* $\Longrightarrow$ *F id-cblinfun = id-cblinfun*›
by (*simp add*: *register-def*)

lemma *register-id*: ‹*register id*›
by (*simp add*: *register-def complex-vector.module-hom-id*)

lemma *register-preregister*: *register F* $\Longrightarrow$ *preregister F*
unfolding *register-def preregister-def* **by** *auto*

lemma *register-comp*: *register F* $\Longrightarrow$ *register G* $\Longrightarrow$ *register* ($G \circ F$)
using *bounded-clinear-compose continuous-map-compose*
apply (*simp add*: *o-def register-def*)
by *blast*

lemma *register-mult*: *register F* $\Longrightarrow$ *cblinfun-compose* (*F a*) (*F b*) = *F* (*cblinfun-compose a b*)
unfolding *register-def*
by *auto*

lemma *register-tensor-left*: ‹*register* ($\lambda a.$ *tensor-op a id-cblinfun*)›
by (*auto simp add*: *comp-tensor-op register-def tensor-op-cbilinear tensor-op-adjoint*
intro!: *tensor-op-cbilinear.bounded-clinear-left*)

lemma *register-tensor-right*: ‹*register* ($\lambda a.$ *tensor-op id-cblinfun a*)›
by (*auto simp add*: *comp-tensor-op register-def tensor-op-cbilinear tensor-op-adjoint*
bounded-cbilinear-apply-bounded-clinear tensor-op-bounded-cbilinear)

definition *register-pair* ::
‹('*a update* $\Rightarrow$ '*c update*) $\Rightarrow$ ('*b update* $\Rightarrow$ '*c update*)
$\Rightarrow$ (('*a*×'*b*) *update* $\Rightarrow$ '*c update*)› **where**
‹*register-pair F G* = (*if register F* $\wedge$ *register G* $\wedge$ ($\forall a\ b.\ F\ a\ o_{CL}\ G\ b = G\ b\ o_{CL}\ F\ a$) *then*
SOME R. ($\forall a\ b.$ *register R* $\wedge$ *R* (*a* $\otimes_o$ *b*) = *F a* o_{CL} *G b*) *else* (λ-. *0*))›

lemma *cbilinear-F-comp-G*[*simp*]: ‹*clinear F* $\Longrightarrow$ *clinear G* $\Longrightarrow$ *cbilinear* ($\lambda a\ b.$ *F a* o_{CL} *G b*)›
unfolding *cbilinear-def*
by (*auto simp add*: *clinear-iff bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose bounded-cbilinear.add-right*)

lemma *register-projector*:
assumes *register F*
assumes *is-Proj a*
shows *is-Proj* (*F a*)
using *assms* **unfolding** *register-def is-Proj-algebraic* **by** *metis*

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

definition ‹*register-decomposition-basis* Φ = (*SOME B. is-ortho-set B* $\wedge$ ($\forall b{\in}B.$ *norm b = 1*) $\wedge$ *ccspan B* = Φ
(*butterfly* (*ket undefined*) (*ket undefined*)) $*_S$ $\top$)›
for Φ :: ‹'*a update* $\Rightarrow$ '*b update*›

lemma *register-decomposition*:
fixes Φ :: ‹'*a update* $\Rightarrow$ '*b update*›
assumes [*simp*]: ‹*register* Φ›
shows ‹*let* '*c*::*type* = *register-decomposition-basis* Φ *in*
($\exists U$ :: ('*a* × '*c*) *ell2* $\Rightarrow_{CL}$ '*b ell2*. *unitary U* $\wedge$
($\forall \vartheta.\ \Phi\ \vartheta =$ *sandwich U* ($\vartheta \otimes_o$ *id-cblinfun*)))›
— Proof based on [Daw21]
proof *with-type-intro*

define $\xi 0 :: 'a$ **where** ‹$\xi 0 = undefined$›

have ‹$bounded\text{-}clinear\ \Phi$›
using *assms register-def* **by** *blast*
then have [*simp*]: ‹$clinear\ \Phi$›
by (*simp add*: *bounded-clinear.clinear*)

define P **where** ‹$P\ i = butterfly\ (ket\ i)\ (ket\ i)$› **for** $i :: 'a$

note *blinfun-cblinfun-eq-bi-unique*[*transfer-rule del*]
note *cblinfun.bi-unique*[*transfer-rule del*]
note *cblinfun.left-unique*[*transfer-rule del*]
note *cblinfun.right-unique*[*transfer-rule del*]
note *cblinfun.right-total*[*transfer-rule del*]
note *id-cblinfun.transfer*[*transfer-rule del*]

define P' **where** ‹$P'\ i = \Phi\ (P\ i)$› **for** $i :: 'a$
have *proj-P'*: ‹*is-Proj* $(P'\ i)$› **for** i
by (*simp add*: *P-def P'-def butterfly-is-Proj register-projector*)
have *sumP'id2*: ‹*has-sum-in weak-star-topology* $(\lambda i.\ P'\ i)$ *UNIV id-cblinfun*›
proof −
from *has-sum-butterfly-ket*
have ‹*has-sum-in weak-star-topology* $(\Phi\ o\ (\lambda x.\ butterfly\ (ket\ x)\ (ket\ x)))$ *UNIV* $(\Phi\ id\text{-}cblinfun)$›
apply (*rule has-sum-in-comm-additive*[*rotated* −*1*])
using *assms*
by (*auto simp*: *complex-vector.linear-add register-def Modules.additive-def*
intro!: *continuous-map-is-continuous-at-point complex-vector.linear-0* ‹*clinear* Φ›)
then show *?thesis*
by (*simp add*: *P'-def P-def o-def register-of-id*)
qed
define S **where** ‹$S\ i = P'\ i *_S \top$› **for** $i :: 'a$
have *P'id*: ‹$P'\ i *_V \psi = \psi$› **if** ‹$\psi \in space\text{-}as\text{-}set\ (S\ i)$› **for** $i\ \psi$
using *S-def that proj-P'*
by (*metis cblinfun-fixes-range is-Proj-algebraic*)

define *S-iso'* :: ‹$'a \Rightarrow 'a \Rightarrow 'a\ update$› **where** ‹*S-iso' i j* = *classical-operator* (*Some o Transposition.transpose i j*)› **for** $i\ j :: 'a$
have *S-iso'-apply*: ‹*S-iso' i j* $*_V$ *ket i* = *ket j*› **for** *i j*
by (*simp add*: *S-iso'-def classical-operator-ket classical-operator-exists-inj*)
have *S-iso'-unitary*: ‹*unitary* (*S-iso' i j*)› **for** *i j*
by (*simp add*: *S-iso'-def unitary-classical-operator*)
have *S-iso'-id*: ‹*S-iso' i i* = *id-cblinfun*› **for** *i*
by (*auto intro*!: *equal-ket simp*: *S-iso'-def classical-operator-ket classical-operator-exists-inj*)
have *S-iso'-adj-apply*: ‹(*S-iso' i j*)* $*_V$ *ket j* = *ket i*› **for** *i j*
by (*metis S-iso'-apply S-iso'-unitary cblinfun-apply-cblinfun-compose id-cblinfun-apply unitaryD1*)
define *S-iso* **where** ‹*S-iso i j* = Φ (*S-iso' i j*)› **for** *i j*
have *uni-S-iso*: ‹*unitary* (*S-iso i j*)› **for** *i j*
by (*simp add*: *S-iso-def S-iso'-unitary register-unitary*)
have *S-iso-S*: ‹*S-iso i j* $*_S$ *S i* = *S j*› **for** *i j*
proof −
have ‹*S-iso i j* $*_S$ *S i* = *S-iso i j* $*_S$ *P' i* $*_S$ *S-iso j i* $*_S \top$›
by (*simp add*: *S-def uni-S-iso*)
also have ‹... = *S j*›
by (*simp add*: *S-def P'-def S-iso-def P-def register-mult butterfly-comp-cblinfun cblinfun-comp-butterfly S-iso'-apply S-iso'-adj-apply*
flip: *cblinfun-compose-image*)
finally show *?thesis*
by −
qed
have *S-iso-id*[*simp*]: ‹*S-iso i i* = *id-cblinfun*› **for** *i*
by (*simp add*: *S-iso'-id S-iso-def register-of-id*)

obtain B_0 **where** B_0: ‹*is-ortho-set* B_0› ‹$\bigwedge b.\ b \in B_0 \Longrightarrow norm\ b = 1$› ‹*ccspan* B_0 = *S* $\xi 0$›

using *orthonormal-subspace-basis-exists*[**where** S={} **and** V=‹S $\xi 0$›]
apply *atomize-elim* **by** *auto*

have *register-decomposition-basis-Φ*: ‹*is-ortho-set* (*register-decomposition-basis* Φ) ∧
(∀ b∈*register-decomposition-basis* Φ. *norm* b = 1) ∧
ccspan (*register-decomposition-basis* Φ) = S $\xi 0$›
unfolding *register-decomposition-basis-def*
apply (*rule someI2*[**where** a=B_0])
using B_0 **by** (*auto simp*: *S-def P'-def P-def ξ0-def*)

define B **where** ‹B i = *S-iso* $\xi 0$ i ‘ *register-decomposition-basis* Φ› **for** i
have *Bξ0*: ‹B $\xi 0$ = *register-decomposition-basis* Φ›
using *B-def* **by** *force*
have *orthoB*: ‹*is-ortho-set* (B i)› **for** i
proof −
have *1*: ‹x ∈ *register-decomposition-basis* Φ ⟹
y ∈ *register-decomposition-basis* Φ ⟹
S-iso $\xi 0$ i *∗V* x ≠ *S-iso* $\xi 0$ i *∗V* y ⟹ *is-orthogonal* (*S-iso* $\xi 0$ i *∗V* x) (*S-iso* $\xi 0$ i *∗V* y)› **for** x y
by (*metis* (*no-types*, *lifting*) *register-decomposition-basis-Φ UNIV-I cblinfun-apply-cblinfun-compose cblinfun-fixes-range cinner-adj-left id-cblinfun-adjoint is-ortho-set-def top-ccsubspace.rep-eq uni-S-iso unitaryD1 unitary-id unitary-range*)
have *2*: ‹x ∈ *register-decomposition-basis* Φ ⟹ *S-iso* $\xi 0$ i *∗V* x ≠ 0› **for** x
by (*metis register-decomposition-basis-Φ cinner-ket-same cinner-zero-left cnorm-eq-1 isometry-preserves-norm orthogonal-ket uni-S-iso unitary-isometry*)
from *1 2* **show** *?thesis*
by (*auto simp add*: *B-def is-ortho-set-def*)
qed
have *normalB*: ‹⋀b. b ∈ B i ⟹ *norm* b = 1› **for** i
by (*metis* (*no-types*, *lifting*) *register-decomposition-basis-Φ B-def imageE isometry-preserves-norm uni-S-iso unitary-twosided-isometry*)
have *cspanB*: ‹*ccspan* (B i) = S i› **for** i
by (*simp add*: *B-def register-decomposition-basis-Φ Bξ0 S-iso-S flip*: *cblinfun-image-ccspan*)

from *orthoB* **have** *indepB*: ‹*cindependent* (B i)› **for** i
by (*simp add*: *Complex-Inner-Product.is-ortho-set-cindependent*)

have *orthoBiBj*: ‹*is-orthogonal* x y› **if** ‹x ∈ B i› **and** ‹y ∈ B j› **and** ‹i ≠ j› **for** x y i j
proof −
have ‹P' i *oCL* P' j = 0›
using ‹i ≠ j›
by (*simp add*: *P'-def P-def register-mult butterfly-comp-butterfly cinner-ket*
‹*clinear* Φ› *complex-vector.linear-0*)
then have ∗: ‹*Proj* (*ccspan* (B i)) *oCL* *Proj* (*ccspan* (B j)) = 0›
by (*simp add*: *Proj-on-own-range S-def cspanB proj-P'*)
then show ‹*is-orthogonal* x y›
by (*meson orthogonal-projectors-orthogonal-spaces orthogonal-spaces-ccspan that*(*1*) *that*(*2*))
qed

define B' **where** ‹B' = (⋃i∈*UNIV*. B i)›

have *P'B*: ‹P' i = *Proj* (*ccspan* (B i))› **for** i
unfolding *cspanB S-def*
using *proj-P' Proj-on-own-range'*[*symmetric*] *is-Proj-algebraic* **by** *blast*

show ‹*register-decomposition-basis* Φ ≠ {}›
proof (*rule ccontr*)
assume ‹¬ *register-decomposition-basis* Φ ≠ {}›
then have ‹B i = {}› **for** i
by (*simp add*: *B-def*)
then have ‹S i = 0› **for** i
using *cspanB* **by** *force*
then have ‹P' i = 0› **for** i
by (*simp add*: *P'B cspanB*)

```
    with sumP'id2
    have ‹has-sum-in weak-star-topology (λi. 0) UNIV id-cblinfun›
      by (metis (no-types, lifting) UNIV-I has-sum-in-0 has-sum-in-cong has-sum-in-unique hausdorff-weak-star
id-cblinfun-not-0 weak-star-topology-topspace)
    then have ‹id-cblinfun = 0›
     using has-sum-in-0 has-sum-in-unique hausdorff-weak-star id-cblinfun-not-0 weak-star-topology-topspace by
fast
    then show False
      using id-cblinfun-not-0 by blast
  qed

  from orthoBiBj orthoB have orthoB': ‹is-ortho-set B'›
    unfolding B'-def is-ortho-set-def by blast
  then have indepB': ‹cindependent B'›
    using is-ortho-set-cindependent by blast

  from orthoBiBj orthoB
  have Bdisj: ‹B i ∩ B j = {}› if ‹i ≠ j› for i j
    unfolding is-ortho-set-def
    using cinner-eq-zero-iff that
    by fastforce

  fix rep-c :: ‹'c ⇒ 'b ell2›
  assume bij-rep-c: ‹bij-betw rep-c UNIV (register-decomposition-basis Φ)›
  then interpret type-definition rep-c ‹inv rep-c› ‹register-decomposition-basis Φ›
    by (simp add: type-definition-bij-betw-iff)

  from bij-rep-c have bij-rep-c: ‹bij-betw rep-c (UNIV :: 'c set) (B ξ0)›
    unfolding Bξ0 by simp

  define u where ‹u = (λ(ξ,α). Φ (butterfly (ket ξ) (ket ξ0)) *V rep-c α)› for ξ :: 'a and α :: ‹'c›

  have cinner-u: ‹cinner (u ξα) (u ξ'α') = of-bool (ξα = ξ'α')› for ξα ξ'α'
  proof −
    obtain ξ α ξ' α' where ξα: ‹ξα = (ξ,α)› and ξ'α': ‹ξ'α' = (ξ',α')›
      apply atomize-elim by auto
    have ‹cinner (u (ξ,α)) (u (ξ', α')) = cinner (Φ (butterfly (ket ξ) (ket ξ0)) *V rep-c α) (Φ (butterfly (ket ξ')
(ket ξ0)) *V rep-c α')›
      unfolding u-def by simp
    also have ‹... = cinner ((Φ (butterfly (ket ξ') (ket ξ0)))* *V Φ (butterfly (ket ξ) (ket ξ0)) *V rep-c α)
(rep-c α')›
      by (simp add: cinner-adj-left)
    also have ‹... = cinner (Φ (butterfly (ket ξ') (ket ξ0) *) *V Φ (butterfly (ket ξ) (ket ξ0)) *V rep-c α) (rep-c
α')›
      by (metis (no-types, lifting) assms register-def)
    also have ‹... = cinner (Φ (butterfly (ket ξ0) (ket ξ') oC∙L butterfly (ket ξ) (ket ξ0)) *V rep-c α) (rep-c
α')›
      by (simp add: register-mult cblinfun-apply-cblinfun-compose[symmetric])
    also have ‹... = cinner (Φ (of-bool (ξ'=ξ) *C butterfly (ket ξ0) (ket ξ0)) *V rep-c α) (rep-c α')›
      by (simp add: cinner-ket-left ket.rep-eq)
    also have ‹... = of-bool (ξ'=ξ) * cinner (Φ (butterfly (ket ξ0) (ket ξ0)) *V rep-c α) (rep-c α')›
      by (simp add: complex-vector.linear-0)
    also have ‹... = of-bool (ξ'=ξ) * cinner (P' ξ0 *V rep-c α) (rep-c α')›
      using P-def P'-def by simp
    also have ‹... = of-bool (ξ'=ξ) * cinner (rep-c α) (rep-c α')›
      apply (subst P'id)
      apply (metis Bξ0 Rep ccspan-superset cspanB in-mono)
      by simp
    also have ‹... = of-bool (ξ'=ξ) * of-bool (α=α')›
      using bij-rep-c orthoB normalB unfolding is-ortho-set-def
      by (smt (verit, best) Bξ0 Rep Rep-inject cnorm-eq-1 of-bool-eq(1) of-bool-eq(2))
    finally show ?thesis
      by (simp add: ξ'α' ξα)
```

qed
define U **where** ‹U = *cblinfun-extension* (*range ket*) (*u o inv ket*)›
have *Uapply*: ‹U $*_V$ *ket* $\xi\alpha$ = *u* $\xi\alpha$› **for** $\xi\alpha$
proof −
have *aux*: ‹($\bigwedge$*a b aa ba*. *u* (*a*, *b*) $\cdot_C$ *u* (*aa*, *ba*) = *of-bool* (*a* = *aa* ∧ *b* = *ba*)) ⟹ *norm* (*u* (*a*, *b*)) ≤ *1*› **for** *a b*
by (*metis* (*full-types*) *cinner-u cnorm-eq-1 of-bool-eq-1-iff order-refl*)
then show *?thesis*
unfolding *U-def*
apply (*subst cblinfun-extension-apply*)
using *cinner-u* **by** (*auto intro*!: *cblinfun-extension-exists-ortho*[**where** *B*=*1*])
qed
have ‹*isometry U*›
apply (*rule-tac orthogonal-on-basis-is-isometry*[**where** *B*=‹*range ket*›])
by (*auto simp*: *Uapply cinner-u*)

have *1*: ‹$U*$ o_{CL} Φ ϑ o_{CL} U = ϑ $\otimes_o$ *id-cblinfun*› **for** ϑ
proof −
have $*$: ‹$U*$ o_{CL} Φ (*butterfly* (*ket* ξ) (*ket* η)) o_{CL} U = *butterfly* (*ket* ξ) (*ket* η) $\otimes_o$ *id-cblinfun*› **for** ξ η
proof (*rule equal-ket*, *rename-tac* $\xi 1\alpha$)
fix $\xi 1\alpha$ **obtain** $\xi 1$ :: $'a$ **and** α :: ‹$'c$› **where** $\xi 1\alpha$: ‹$\xi 1\alpha$ = ($\xi 1$,α)›
apply *atomize-elim* **by** *auto*
have ‹($U*$ o_{CL} Φ (*butterfly* (*ket* ξ) (*ket* η)) o_{CL} U) $*_V$ *ket* $\xi 1\alpha$ = $U*$ $*_V$ Φ (*butterfly* (*ket* ξ) (*ket* η)) $*_V$ Φ (*butterfly* (*ket* $\xi 1$) (*ket* $\xi 0$)) $*_V$ *rep-c* α›
unfolding *cblinfun-apply-cblinfun-compose* $\xi 1\alpha$ *Uapply u-def* **by** *simp*
also have ‹... = $U*$ $*_V$ Φ (*butterfly* (*ket* ξ) (*ket* η) o_{CL} *butterfly* (*ket* $\xi 1$) (*ket* $\xi 0$)) $*_V$ *rep-c* α›
by (*metis assms register-mult simp-a-oCL-b'*)
also have ‹... = $U*$ $*_V$ Φ (*of-bool* (η=$\xi 1$) $*_C$ *butterfly* (*ket* ξ) (*ket* $\xi 0$)) $*_V$ *rep-c* α›
by (*simp add*: *cinner-ket*)
also have ‹... = *of-bool* (η=$\xi 1$) $*_C$ $U*$ $*_V$ Φ (*butterfly* (*ket* ξ) (*ket* $\xi 0$)) $*_V$ *rep-c* α›
by (*simp add*: *complex-vector.linear-scale*)
also have ‹... = *of-bool* (η=$\xi 1$) $*_C$ $U*$ $*_V$ U $*_V$ *ket* (ξ, α)›
unfolding *Uapply u-def* **by** *simp*
also from ‹*isometry U*› **have** ‹... = *of-bool* (η=$\xi 1$) $*_C$ *ket* (ξ, α)›
unfolding *cblinfun-apply-cblinfun-compose*[*symmetric*] **by** *simp*
also have ‹... = (*butterfly* (*ket* ξ) (*ket* η) $*_V$ *ket* $\xi 1$) $\otimes_s$ *ket* α›
by (*simp add*: *tensor-ell2-scaleC1 tensor-ell2-ket*)
also have ‹... = (*butterfly* (*ket* ξ) (*ket* η) $\otimes_o$ *id-cblinfun*) $*_V$ *ket* $\xi 1\alpha$›
by (*simp add*: $\xi 1\alpha$ *tensor-op-ket*)
finally show ‹($U*$ o_{CL} Φ (*butterfly* (*ket* ξ) (*ket* η)) o_{CL} U) $*_V$ *ket* $\xi 1\alpha$ = (*butterfly* (*ket* ξ) (*ket* η) $\otimes_o$ *id-cblinfun*) $*_V$ *ket* $\xi 1\alpha$›
by −
qed

have *cont1*: ‹*continuous-map weak-star-topology weak-star-topology* (λa. $U*$ o_{CL} Φ *a* o_{CL} U)›
apply (*subst asm-rl*[*of* ‹(λa. $U*$ o_{CL} Φ *a* o_{CL} U) = (λx. *x* o_{CL} U) *o* (λx. $U*$ o_{CL} *x*) *o* Φ›])
apply *force*
apply (*intro continuous-map-compose*[**where** *X'*=*weak-star-topology*])
using *assms register-def continuous-map-left-comp-weak-star continuous-map-right-comp-weak-star* **by** *blast*+

have $*$: ‹$U*$ o_{CL} Φ ϑ o_{CL} U = ϑ $\otimes_o$ *id-cblinfun*› **if** ‹ϑ ∈ *cspan* (*range* ($\lambda(\xi, \eta)$. *butterfly* (*ket* ξ) (*ket* η)))› **for** ϑ
apply (*rule complex-vector.linear-eq-on*[**where** *x*=ϑ, *OF - - that*])
apply (*intro* ‹*clinear* Φ›
clinear-compose[*OF - clinear-cblinfun-compose-left*, *unfolded o-def*]
clinear-compose[*OF - clinear-cblinfun-compose-right*, *unfolded o-def*])
apply *simp*
using $*$ **by** *fast*
have ‹$U*$ o_{CL} Φ ϑ o_{CL} U = ϑ $\otimes_o$ *id-cblinfun*›
if ‹ϑ ∈ (*weak-star-topology closure-of* (*cspan* (*range* ($\lambda(\xi, \eta)$. *butterfly* (*ket* ξ) (*ket* η)))))› **for** ϑ
apply (*rule closure-of-eqI*[*OF - - that*])
using $*$ *cont1 left-amplification-weak-star-cont* **by** *auto*

```
    with butterkets-weak-star-dense show ?thesis
      by auto
  qed
  have ‹unitary U›
  proof −
    have ‹Φ (butterfly (ket ξ) (ket ξ1)) *S ⊤ ≤ U *S ⊤› for ξ ξ1
    proof −
      have *: ‹Φ (butterfly (ket ξ) (ket ξ0)) *V b ∈ space-as-set (U *S ⊤)› if ‹b ∈ B ξ0› for b
        apply (subst asm-rl[of ‹Φ (butterfly (ket ξ) (ket ξ0)) *V b = u (ξ, inv rep-c b)›])
         apply (simp add: u-def, metis bij-betw-inv-into-right bij-rep-c that)
        by (metis Uapply cblinfun-apply-in-image)

     have ‹Φ (butterfly (ket ξ) (ket ξ1)) *S ⊤ = Φ (butterfly (ket ξ) (ket ξ0)) *S Φ (butterfly (ket ξ0) (ket ξ0))
*S Φ (butterfly (ket ξ0) (ket ξ1)) *S ⊤›
        unfolding cblinfun-compose-image[symmetric] register-mult[OF assms]
        by simp
      also have ‹... ≤ Φ (butterfly (ket ξ) (ket ξ0)) *S Φ (butterfly (ket ξ0) (ket ξ0)) *S ⊤›
        by (meson cblinfun-image-mono top-greatest)
      also have ‹... = Φ (butterfly (ket ξ) (ket ξ0)) *S S ξ0›
        by (simp add: S-def P'-def P-def)
      also have ‹... = Φ (butterfly (ket ξ) (ket ξ0)) *S ccspan (B ξ0)›
        by (simp add: cspanB)
      also have ‹... = ccspan (Φ (butterfly (ket ξ) (ket ξ0)) ‘ B ξ0)›
        by (meson cblinfun-image-ccspan)
      also have ‹... ≤ U *S ⊤›
        by (rule ccspan-leqI, use * in auto)
      finally show ?thesis by −
    qed
    then have ‹ccspan {Φ (butterfly (ket ξ) (ket ξ)) *V α |ξ α. True} ≤ U *S ⊤›
      apply (rule-tac ccspan-leqI)
      using cblinfun-apply-in-image less-eq-ccsubspace.rep-eq by blast
    moreover have ‹ccspan {Φ (butterfly (ket ξ) (ket ξ)) *V α |ξ α. True} = ⊤›
    proof −
      define Q where ‹Q = Proj (− ccspan {Φ (butterfly (ket ξ) (ket ξ)) *V α |ξ α. True})›
      have ‹has-sum-in weak-star-topology (λξ. Q oCL Φ (butterfly (ket ξ) (ket ξ))) UNIV (Q oCL id-cblinfun)›
      apply (rule has-sum-in-comm-additive[where g=‹cblinfun-compose Q› and T=weak-star-topology, unfolded
o-def])
        using sumP'id2
        by (auto simp add: continuous-map-left-comp-weak-star P'-def P-def cblinfun-compose-add-right Mod-
ules.additive-def)
      moreover have ‹Q oCL Φ (butterfly (ket ξ) (ket ξ)) = 0› for ξ
        apply (rule equal-ket)
        apply (simp add: Q-def Proj-ortho-compl cblinfun.diff-left)
        apply (subst Proj-fixes-image)
        by (auto intro!: ccspan-superset[THEN set-mp])
      ultimately have ‹Q = 0›
        apply (rule-tac has-sum-in-unique)
        by auto
      then show ?thesis
        by (smt (verit, del-insts) Q-def Proj-ortho-compl Proj-range cblinfun-image-id right-minus-eq)
    qed
    ultimately have ‹U *S ⊤ = ⊤›
      by (simp add: top.extremum-unique)
    with ‹isometry U› show ‹unitary U›
      by (rule surj-isometry-is-unitary)
  qed

  have ‹Φ ϑ = U oCL (ϑ ⊗o id-cblinfun) oCL U*› for ϑ
  proof −
    from ‹unitary U›
    have ‹Φ ϑ = (U oCL U*) oCL Φ ϑ oCL (U oCL U*)›
      by simp
    also have ‹... = U oCL (U*  oCL Φ ϑ oCL U) oCL U*›
```

```
    by (simp add: cblinfun-assoc-left)
  also have ‹... = U o_CL (ϑ ⊗o id-cblinfun) o_CL U*›
    using 1 by simp
  finally show ?thesis
    by −
qed

 with ‹unitary U› show ‹∃ U :: ('a × 'c) ell2 ⇒_CL 'b ell2. unitary U ∧ (∀ϑ. Φ ϑ = sandwich U (ϑ ⊗o
id-cblinfun))›
  by (auto simp: sandwich-apply)
qed

lemma register-bounded-clinear: ‹register F ⟹ bounded-clinear F›
 using preregister-def register-preregister by blast

lemma clinear-register: ‹register F ⟹ clinear F›
 using bounded-clinear.clinear register-bounded-clinear by blast

lemma weak-star-cont-register: ‹register F ⟹ continuous-map weak-star-topology weak-star-topology F›
 using register-def by blast

lemma register-inv-weak-star-continuous:
 assumes ‹register F›
 shows ‹continuous-map (subtopology weak-star-topology (range F)) weak-star-topology (inv F)›
proof (rule continuous-map-iff-preserves-convergence, rename-tac K a)
 fix K a
 assume limit-id: ‹limitin (subtopology weak-star-topology (range F)) id a K›
 from register-decomposition
 have ‹let 'c::type = register-decomposition-basis F in
       limitin weak-star-topology (inv F) (inv F a) K›
 proof with-type-mp
  from assms show ‹register F› by −
 next
  with-type-case
  then obtain U :: ‹('a × 'c) ell2 ⇒_CL 'b ell2› 
    where ‹unitary U› and FU: ‹F ϑ = sandwich U (ϑ ⊗o id-cblinfun)› for ϑ
    by auto
  define δ :: ‹'c ell2 ⇒_CL 'c ell2› where ‹δ = selfbutter (ket (undefined))›
  then have [simp]: ‹trace-class δ›
    by simp
  define u where ‹u t = U o_CL (from-trace-class t ⊗o δ) o_CL U*› for t
  have [simp]: ‹trace-class (u t)› for t
    unfolding u-def
    apply (rule trace-class-comp-left)
    apply (rule trace-class-comp-right)
    by (simp add: trace-class-tensor)
  have uF: ‹trace (from-trace-class t o_CL a) = trace (u t o_CL F a)› for t a
  proof −
    have ‹trace (from-trace-class t o_CL a) = trace (from-trace-class t o_CL a) * trace (δ o_CL id-cblinfun)›
      by (simp add: δ-def trace-butterfly)
    also have ‹... = trace ((from-trace-class t o_CL a) ⊗o (δ o_CL id-cblinfun))›
      by (simp add: trace-class-comp-left trace-tensor)
    also have ‹... = trace ((from-trace-class t ⊗o δ) o_CL (a ⊗o id-cblinfun))›
      by (simp add: comp-tensor-op)
    also have ‹... = trace (U* o_CL u t o_CL U o_CL (a ⊗o id-cblinfun))›
      using ‹unitary U›
      by (simp add: u-def lift-cblinfun-comp[OF unitaryD1] cblinfun-compose-assoc)
    also have ‹... = trace (u t o_CL U o_CL (a ⊗o id-cblinfun) o_CL U*)›
      apply (subst (2) circularity-of-trace)
      by (simp-all add: trace-class-comp-left cblinfun-compose-assoc)
    also have ‹... = trace (u t o_CL F a)›
      by (simp add: sandwich-apply FU cblinfun-compose-assoc)
    finally show ?thesis
```

```
      by −
    qed
    from limit-id
    have ‹a ∈ range F› and KrangeF: ‹∀⇩F a in K. a ∈ range F› and limit-id': ‹limitin weak-star-topology id a
K›
      unfolding limitin-subtopology by auto
    from ‹a ∈ range F› have FiFa: ‹F (inv F a) = a›
      by (simp add: f-inv-into-f)
    from KrangeF
    have ∗: ‹∀⇩F x in K. trace (from-trace-class t o⇩C⇩L F (inv F x)) = trace (from-trace-class t o⇩C⇩L x)› for t
      apply (rule eventually-mono)
      by (simp add: f-inv-into-f)
    from limit-id' have ‹((λa'. trace (from-trace-class t o⇩C⇩L a')) ⟶⇩F trace (from-trace-class t o⇩C⇩L a)) K› for
t
      unfolding limitin-weak-star-topology' by simp
    then have ∗: ‹((λa'. trace (from-trace-class t o⇩C⇩L F (inv F a'))) ⟶⇩F trace (from-trace-class t o⇩C⇩L F (inv
F a))) K› for t
      unfolding FiFa using ∗ by (rule tendsto-cong[THEN iffD2, rotated])
    have ‹((λa'. trace (u t o⇩C⇩L F (inv F a'))) ⟶⇩F trace (u t o⇩C⇩L F (inv F a))) K› for t
      using ∗[of ‹Abs-trace-class (u t)›]
      by (simp add: Abs-trace-class-inverse)
    then have ‹((λa'. trace (from-trace-class t o⇩C⇩L inv F a')) ⟶⇩F trace (from-trace-class t o⇩C⇩L inv F a)) K›
for t
      by (simp add: uF[symmetric])
    then show ‹limitin weak-star-topology (inv F) (inv F a) K›
      by (simp add: limitin-weak-star-topology')
  qed
  note this[cancel-with-type]
  then show ‹limitin weak-star-topology (inv F) (inv F a) K›
    by −
qed

lemma register-inj: ‹inj-on F X› if [simp]: ‹register F›
proof −
  have ‹let 'c::type = register-decomposition-basis F in inj F›
    using register-decomposition[OF ‹register F›]
  proof with-type-mp
    with-type-case
    then obtain U :: ‹('a × 'c) ell2 ⇒⇩C⇩L 'b ell2›
      where ‹unitary U› and F: ‹F a = Complex-Bounded-Linear-Function.sandwich U (a ⊗⇩o id-cblinfun)› for
a
      apply atomize-elim by auto
    have ‹inj (Complex-Bounded-Linear-Function.sandwich U)›
      by (smt (verit, best) ‹unitary U› cblinfun-assoc-left inj-onI sandwich-apply cblinfun-compose-id-right cblin-
fun-compose-id-left unitary-def)
    moreover have ‹inj (λa::'a ell2 ⇒⇩C⇩L -. a ⊗⇩o id-cblinfun)›
      by (rule inj-tensor-left, simp)
    ultimately show ‹inj F›
      unfolding F
      by (smt (z3) inj-def)
  qed
  from this[THEN with-type-prepare-cancel, cancel-type-definition, OF with-type-nonempty, OF this]
  show ‹inj-on F X›
    by (simp add: inj-on-def)
qed

lemma register-norm: ‹norm (F a) = norm a› if ‹register F›
proof −
  from register-decomposition[OF that]
  have ‹let 'c::type = register-decomposition-basis F in
        norm (F a) = norm a›
  proof with-type-mp
    with-type-case
```

```
    then obtain U :: ‹('a × 'c) ell2 ⇒CL 'b ell2› where ‹unitary U›
      and FU: ‹F ϑ = sandwich U (ϑ ⊗o id-cblinfun)› for ϑ
      by metis
    show ‹norm (F a) = norm a›
      using ‹unitary U›
      by (simp add: FU sandwich-apply norm-isometry-compose norm-isometry-compose' norm-isometry tensor-op-norm)
  qed
  note this[cancel-with-type]
  then show ?thesis
    by simp
qed

lemma unitary-sandwich-register: ‹unitary a ⟹ register (sandwich a)›
  by (auto simp: sandwich-apply register-def cblinfun.bounded-clinear-right
      lift-cblinfun-comp[OF unitaryD1] lift-cblinfun-comp[OF unitaryD2]
      cblinfun-assoc-left)

lemma register-adj: ‹register F ⟹ F (a*) = (F a)*›
  using register-def by blast

lemma right-register-tensor-ex: ‹∃ T :: ('a × 'b) update ⇒ ('a × 'c) update.
        register T ∧ (∀ a b. T (a ⊗o b) = a ⊗o F b)› if ‹register F›
proof −
  from register-decomposition[OF ‹register F›]
  have ‹let 'g::type = register-decomposition-basis F in
        ∃ T :: ('a × 'b) update ⇒ ('a × 'c) update.
        register T ∧ (∀ a b. T (a ⊗o b) = a ⊗o F b)›
  proof with-type-mp
    with-type-case
    then obtain U :: ‹('b × 'g) ell2 ⇒CL 'c ell2›
      where [simp]: ‹unitary U› and F: ‹F ϑ = sandwich U *V ϑ ⊗o id-cblinfun› for ϑ
      by auto
    define T :: ‹('a × 'b) update ⇒ ('a × 'c) update›
      where ‹T = sandwich (id-cblinfun ⊗o U) o sandwich assoc-ell2 o (λab. ab ⊗o id-cblinfun)›
    have ‹register T›
      by (auto intro!: register-comp register-tensor-left unitary-sandwich-register unitary-tensor-op simp add: T-def)
    moreover have ‹T (a ⊗o b) = a ⊗o F b› for a b
      by (simp add: T-def F sandwich-tensor-op)
    ultimately show ‹∃ T :: ('a × 'b) update ⇒ ('a × 'c) update. register T ∧ (∀ a b. T (a ⊗o b) = a ⊗o F b)›
      by auto
  qed
  from this[cancel-with-type]
  show ?thesis
    by −
qed




lemma
  fixes F :: ‹'a update ⇒ 'c update› and G
  assumes [simp]: ‹register F› ‹register G›
  assumes FG-comm: ‹⋀a b. F a oCL G b = G b oCL F a›
  shows register-pair-apply: ‹(register-pair F G) (tensor-op a b) = F a oCL G b›
    and register-pair-is-register: ‹register (register-pair F G)›
proof −
  have ∗: ‹register-pair F G = (SOME R. ∀ a b. register R ∧ R (a ⊗o b) = F a oCL G b)›
    using assms unfolding register-pair-def by simp
  from register-decomposition[OF ‹register F›]
  have ‹let 'd::type = register-decomposition-basis F in
```

```
    ∃R. ∀a b. register R ∧ R (a ⊗o b) = F a o_CL G b›
  proof with-type-mp
   with-type-case

   then obtain U :: ‹('a × 'd) ell2 ⇒_CL 'c ell2› where [simp]: ‹unitary U›
     and FU: ‹F ϑ = sandwich U (ϑ ⊗o id-cblinfun)› for ϑ
     by metis
   from register-decomposition[OF ‹register G›]
   have ‹let 'f::type = register-decomposition-basis G in
        ∃R. ∀a b. register R ∧ R (a ⊗o b) = F a o_CL G b›
   proof with-type-mp
     with-type-case
     then obtain V :: ‹('b × 'f) ell2 ⇒_CL 'c ell2› where [simp]: ‹unitary V›
       and GU: ‹G ϑ = sandwich V (ϑ ⊗o id-cblinfun)› for ϑ
       by metis
     show ‹∃FG. ∀a b. register FG ∧ FG (a ⊗o b) = F a o_CL G b›
     proof −
       define G' and ι :: ‹'d update ⇒ ('a × 'd) update› and G₁
         where ‹G' = sandwich (U*) o G› and ‹ι d = id-cblinfun ⊗o d› and ‹G₁ = inv ι o G'› for d
       have [simp]: ‹register G'›
         by (simp add: G'-def register-comp unitary-sandwich-register)
       then have [simp]: ‹bounded-clinear G'›
         by (meson register-bounded-clinear)
       then have [simp]: ‹clinear G'›
         by (simp add: bounded-clinear.axioms(1))


       have ‹range G' = sandwich (U*) ‘ range G›
         by (simp add: GU G'-def image-image)
       also have ‹... ⊆ sandwich (U*) ‘ commutant (range F)›
         by (auto intro!: image-mono simp: commutant-def FG-comm)
       also have ‹... = commutant (sandwich (U*) ‘ range F)›
         by (simp add: sandwich-unitary-commutant)
       also have ‹... = commutant (range (λa. a ⊗o id-cblinfun))›
         apply (rule arg-cong[where f=commutant])
         by (simp add: FU image-image flip: sandwich-compose cblinfun-apply-cblinfun-compose)
       also have ‹... = range (λd. id-cblinfun ⊗o d)›
         by (rule commutant-tensor1)
       also have ‹... = range ι›
         by (simp add: ι-def[abs-def])
       finally have range-G': ‹range G' ⊆ range ι›
         by −

       have ‹continuous-map weak-star-topology weak-star-topology G'›
      by (auto intro!: continuous-map-compose[where X'=weak-star-topology] simp: G'-def weak-star-cont-register)
       then have cont-G': ‹continuous-map weak-star-topology (subtopology weak-star-topology (range ι)) G'›
         using range-G' by (auto intro!: continuous-map-into-subtopology)

       have [simp]: ‹register ι›
         by (simp add: ι-def[abs-def] register-tensor-right)
       then have cont-invι: ‹continuous-map (subtopology weak-star-topology (range ι)) weak-star-topology (inv
ι)›
         by (rule register-inv-weak-star-continuous)
       have ι-inj: ‹x = y› if ‹ι x = ι y› for x y
         by (metis ‹register ι› invI register-inj that)

       have [simp]: ‹register G₁›
       proof (unfold register-def, intro conjI allI)
         from cont-G' cont-invι
         show cont-G₁: ‹continuous-map weak-star-topology weak-star-topology G₁›
           using G₁-def continuous-map-compose by blast
         have ι-cancel: ‹ι (inv ι x) = x› if ‹x ∈ range G'› for x
           by (meson f-inv-into-f range-G' subsetD that)
```

```
      show ‹bounded-clinear G₁›
        using range-G'
        by (auto intro!: bounded-clinearI[where K=1] ι-inj
            simp: G₁-def complex-vector.linear-add[of ι] bounded-clinear.clinear clinear-register
            ι-cancel range-subsetD complex-vector.linear-scale[of ι] register-norm[of G']
            simp flip: complex-vector.linear-add[of G'] complex-vector.linear-scale[of G']
            register-norm[of ι])
      show ‹G₁ id-cblinfun = id-cblinfun›
        by (auto intro!: ι-inj register-of-id[of G'] simp add: G₁-def ι-cancel register-of-id[of ι])
      show adj-G₁: ‹G₁ (a*) = (G₁ a)*› for a
        using range-G'
        by (auto intro!: ι-inj
            simp: G₁-def ι-cancel register-adj[of ι]
            simp flip: register-adj[of G'])
      show mult-G₁: ‹G₁ (a o_CL b) = G₁ a o_CL G₁ b› for a b
        using range-G'
        by (auto intro!: bounded-clinearI[where K=1] ι-inj
            simp: G₁-def ι-cancel register-mult[of G']
            simp flip: register-mult[of ι])
    qed

    obtain T :: ‹('a × 'b) update ⇒ ('a × 'd) update›
      where [simp]: ‹register T› and T-apply: ‹T (a ⊗o b) = a ⊗o G₁ b› for a b
      using ‹register G₁› right-register-tensor-ex by blast

    define FG where ‹FG = sandwich U o T›
    then have [simp]: ‹register FG›
      by (auto intro!: register-comp unitary-sandwich-register simp add: FG-def)

    have ‹FG (a ⊗o b) = F a o_CL G b› for a b
    proof −
      have FG-a: ‹FG (a ⊗o id-cblinfun) = F a›
        by (simp add: FG-def T-apply register-of-id FU)
      have ‹FG (id-cblinfun ⊗o b) = sandwich U (ι (G₁ b))›
        by (simp add: FG-def T-apply ι-def)
      also have ‹... = sandwich U (G' b)›
        apply (rule arg-cong[where f=‹cblinfun-apply -›])
        by (metis G₁-def UNIV-I f-inv-into-f image-subset-iff o-def range-G')
      also have ‹... = G b›
          by (smt (verit) G'-def ‹unitary U› cblinfun-apply-cblinfun-compose cblinfun-compose-id-left cblin-
fun-compose-id-right comp-def id-cblinfun-adjoint sandwich.rep-eq sandwich-compose unitaryD2)
      finally have FG-b: ‹FG (id-cblinfun ⊗o b) = G b›
        by −
      have ‹FG (a ⊗o b) = FG (a ⊗o id-cblinfun) o_CL FG (id-cblinfun ⊗o b)›
        by (simp add: comp-tensor-op register-mult)
      also have ‹... = F a o_CL G b›
        by (simp add: FG-a FG-b)
      finally show ?thesis
        by −
    qed
    with ‹register FG› show ?thesis
      by metis
  qed
  from this[cancel-with-type]
  show ‹∃R. ∀a b. register R ∧ R (a ⊗o b) = F a o_CL G b›
    by −
qed
from this[cancel-with-type]
have ‹∃R. ∀a b. register R ∧ R (a ⊗o b) = F a o_CL G b›
  by −
then have ‹∀a b. register (register-pair F G) ∧ (register-pair F G) (a ⊗o b) = F a o_CL G b›
```

unfolding $*$ **by** (*smt* (*verit*) *someI-ex*)
then show ‹(*register-pair F G*) (*tensor-op a b*) = *F a* o_{CL} *G b*› **and** ‹*register* (*register-pair F G*)›
by *auto*
qed

unbundle *no cblinfun-syntax*

end

10 Generic laws about registers, instantiated quantumly

theory *Laws-Quantum*
imports *Axioms-Quantum*
begin

This notation is only used inside this file

notation *cblinfun-compose* (**infixl** $*_u$ *55*)
notation *tensor-op* (**infixr** $\otimes_u$ *70*)
notation *register-pair* ('(-;-'))

10.1 Elementary facts

declare *id-preregister*[*simp*]
declare *cblinfun-compose-id-left*[*simp*]
declare *cblinfun-compose-id-right*[*simp*]
declare *register-preregister*[*simp*]
declare *register-comp*[*simp*]
declare *register-of-id*[*simp*]
declare *register-tensor-left*[*simp*]
declare *register-tensor-right*[*simp*]
declare *preregister-mult-right*[*simp*]
declare *preregister-mult-left*[*simp*]
declare *register-id*[*simp*]

10.2 Preregisters

lemma *preregister-tensor-left*[*simp*]: ‹*preregister* (λ*b*::'*b*::*type update*. *tensor-op a b*)›
for *a* :: ‹'*a*::*type update*›
proof −
have ‹*preregister* ((λ*b1*::('*a*×'*b*) *update*. (*a* $\otimes_u$ *id-cblinfun*) $*_u$ *b1*) *o* (λ*b*. *tensor-op id-cblinfun b*))›
by (*rule comp-preregister*; *simp*)
then show *?thesis*
by (*simp add*: *o-def comp-tensor-op*)
qed

lemma *preregister-tensor-right*[*simp*]: ‹*preregister* (λ*a*::'*a*::*type update*. *tensor-op a b*)›
for *b* :: ‹'*b*::*type update*›
proof −
have ‹*preregister* ((λ*a1*::('*a*×'*b*) *update*. (*id-cblinfun* $\otimes_u$ *b*) $*_u$ *a1*) *o* (λ*a*. *tensor-op a id-cblinfun*))›
by (*rule comp-preregister*, *simp-all*)
then show *?thesis*
by (*simp add*: *o-def comp-tensor-op*)
qed

10.3 Registers

lemma *id-update-tensor-register*[*simp*]:
assumes ‹*register F*›
shows ‹*register* (λ*a*::'*a*::*type update*. *id-cblinfun* $\otimes_u$ *F a*)›
using *assms* **apply** (*rule register-comp*[*unfolded o-def*])
by *simp*

lemma *register-tensor-id-update*[*simp*]:

assumes ‹*register F*›
shows ‹*register* (λ*a*::′*a*::*type update*. *F a* ⊗*u id-cblinfun*)›
using *assms* **apply** (*rule register-comp*[*unfolded o-def*])
by *simp*

10.4 Tensor product of registers

definition *register-tensor* (**infixr** ⊗*r* *70*) **where**
register-tensor F G = *register-pair* (λ*a*. *tensor-op* (*F a*) *id-cblinfun*) (λ*b*. *tensor-op id-cblinfun* (*G b*))

lemma *register-tensor-is-register*:
fixes *F* :: ′*a*::*type update* ⇒ ′*b*::*type update* **and** *G* :: ′*c*::*type update* ⇒ ′*d*::*type update*
shows *register F* ⟹ *register G* ⟹ *register* (*F* ⊗*r G*)
unfolding *register-tensor-def*
apply (*rule register-pair-is-register*)
by (*simp-all add*: *comp-tensor-op*)

lemma *register-tensor-apply*[*simp*]:
fixes *F* :: ′*a*::*type update* ⇒ ′*b*::*type update* **and** *G* :: ′*c*::*type update* ⇒ ′*d*::*type update*
assumes ‹*register F*› **and** ‹*register G*›
shows (*F* ⊗*r G*) (*a* ⊗*u b*) = *F a* ⊗*u G b*
unfolding *register-tensor-def*
apply (*subst register-pair-apply*)
unfolding *register-tensor-def*
by (*simp-all add*: *assms comp-tensor-op*)

definition *separating* (-::′*b*::*type itself*) *A* ⟷
(∀ *F G* :: ′*a*::*type update* ⇒ ′*b update*. *preregister F* ⟶ *preregister G* ⟶ (∀ *x*∈*A*. *F x* = *G x*) ⟶ *F* = *G*)

lemma *separating-UNIV*[*simp*]: ‹*separating TYPE*(-) *UNIV*›
unfolding *separating-def* **by** *auto*

lemma *separating-mono*: ‹*A* ⊆ *B* ⟹ *separating TYPE*(′*a*::*type*) *A* ⟹ *separating TYPE*(′*a*) *B*›
unfolding *separating-def* **by** (*meson in-mono*)

lemma *register-eqI*: ‹*separating TYPE*(′*b*::*type*) *A* ⟹ *preregister F* ⟹ *preregister G* ⟹ (⋀*x*. *x*∈*A* ⟹ *F x* = *G x*) ⟹ *F* = (*G*::- ⇒ ′*b update*)›
unfolding *separating-def* **by** *auto*

lemma *separating-tensor*:
fixes *A* :: ‹′*a*::*type update set*› **and** *B* :: ‹′*b*::*type update set*›
assumes [*simp*]: ‹*separating TYPE*(′*c*::*type*) *A*›
assumes [*simp*]: ‹*separating TYPE*(′*c*) *B*›
shows ‹*separating TYPE*(′*c*) {*a* ⊗*u b* | *a b*. *a*∈*A* ∧ *b*∈*B*}›
proof (*unfold separating-def*, *intro allI impI*)
fix *F G* :: ‹(′*a*×′*b*) *update* ⇒ ′*c update*›
assume [*simp*]: ‹*preregister F*› ‹*preregister G*›
have [*simp*]: ‹*preregister* (λ*x*. *F* (*a* ⊗*u x*))› **for** *a*
using - ‹*preregister F*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*preregister* (λ*x*. *G* (*a* ⊗*u x*))› **for** *a*
using - ‹*preregister G*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*preregister* (λ*x*. *F* (*x* ⊗*u b*))› **for** *b*
using - ‹*preregister F*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*preregister* (λ*x*. *G* (*x* ⊗*u b*))› **for** *b*
using - ‹*preregister G*› **apply** (*rule comp-preregister*[*unfolded o-def*])
by *simp*

assume ‹∀ *x*∈{*a* ⊗*u b* |*a b*. *a*∈*A* ∧ *b*∈*B*}. *F x* = *G x*›
then have *EQ*: ‹*F* (*a* ⊗*u b*) = *G* (*a* ⊗*u b*)› **if** ‹*a* ∈ *A*› **and** ‹*b* ∈ *B*› **for** *a b*
using *that* **by** *auto*

```
  then have ‹F (a ⊗u b) = G (a ⊗u b)› if ‹a ∈ A› for a b
    apply (rule register-eqI[where A=B, THEN fun-cong, where x=b, rotated −1])
    using that by auto
  then have ‹F (a ⊗u b) = G (a ⊗u b)› for a b
    apply (rule register-eqI[where A=A, THEN fun-cong, where x=a, rotated −1])
    by auto
  then show F = G
    apply (rule tensor-extensionality[rotated −1])
    by auto
qed

lemma register-tensor-distrib:
  assumes [simp]: ‹register F› ‹register G› ‹register H› ‹register L›
  shows ‹(F ⊗r G) o (H ⊗r L) = (F o H) ⊗r (G o L)›
  apply (rule tensor-extensionality)
  by (auto intro!: register-comp register-preregister register-tensor-is-register)
```

The following is easier to apply using the *rule*-method than *separating-tensor*

```
lemma separating-tensor':
  fixes A :: ‹'a::type update set› and B :: ‹'b::type update set›
  assumes ‹separating TYPE('c::type) A›
  assumes ‹separating TYPE('c) B›
  assumes ‹C = {a ⊗u b | a b. a∈A ∧ b∈B}›
  shows ‹separating TYPE('c) C›
  using assms
  by (simp add: separating-tensor)

lemma tensor-extensionality3:
  fixes F G :: ‹('a::type×'b::type×'c::type) update ⇒ 'd::type update›
  assumes [simp]: ‹register F› ‹register G›
  assumes ⋀f g h. F (f ⊗u g ⊗u h) = G (f ⊗u g ⊗u h)
  shows F = G
proof (rule register-eqI[where A=‹{a⊗ub⊗uc| a b c. True}›])
  have ‹separating TYPE('d) {b ⊗u c |b c. True}›
    apply (rule separating-tensor'[where A=UNIV and B=UNIV])
    by auto
  then show ‹separating TYPE('d) {a ⊗u b ⊗u c |a b c. True}›
    apply (rule-tac separating-tensor'[where A=UNIV and B=‹{b⊗uc| b c. True}›])
    by auto
  show ‹preregister F› ‹preregister G› by auto
  show ‹x ∈ {a ⊗u b ⊗u c |a b c. True} ⟹ F x = G x› for x
    using assms(3) by auto
qed

lemma tensor-extensionality3':
  fixes F G :: ‹(('a::type×'b::type)×'c::type) update ⇒ 'd::type update›
  assumes [simp]: ‹register F› ‹register G›
  assumes ⋀f g h. F ((f ⊗u g) ⊗u h) = G ((f ⊗u g) ⊗u h)
  shows F = G
proof (rule register-eqI[where A=‹{(a⊗ub)⊗uc| a b c. True}›])
  have ‹separating TYPE('d) {a ⊗u b | a b. True}›
    apply (rule separating-tensor'[where A=UNIV and B=UNIV])
    by auto
  then show ‹separating TYPE('d) {(a ⊗u b) ⊗u c |a b c. True}›
    apply (rule-tac separating-tensor'[where B=UNIV and A=‹{a⊗ub| a b. True}›])
    by auto
  show ‹preregister F› ‹preregister G› by auto
  show ‹x ∈ {(a ⊗u b) ⊗u c |a b c. True} ⟹ F x = G x› for x
    using assms(3) by auto
qed

lemma register-tensor-id[simp]: ‹id ⊗r id = id›
  apply (rule tensor-extensionality)
```

by (*auto simp add*: *register-tensor-is-register*)

10.5 Pairs and compatibility

definition *compatible* :: ‹('*a*::*type update* ⇒ '*c*::*type update*)
⇒ ('*b*::*type update* ⇒ '*c update*) ⇒ *bool*› **where**
‹*compatible F G* ⟷ *register F* ∧ *register G* ∧ (∀ *a b*. *F a* ∗*u* *G b* = *G b* ∗*u* *F a*)›

lemma *compatibleI*:
assumes *register F* **and** *register G*
assumes ‹⋀*a b*. (*F a*) ∗*u* (*G b*) = (*G b*) ∗*u* (*F a*)›
shows *compatible F G*
using *assms* **unfolding** *compatible-def* **by** *simp*

lemma *swap-registers*:
assumes *compatible R S*
shows *R a* ∗*u* *S b* = *S b* ∗*u* *R a*
using *assms* **unfolding** *compatible-def* **by** *metis*

lemma *compatible-sym*: *compatible x y* ⟹ *compatible y x*
by (*simp add*: *compatible-def*)

lemma *pair-is-register*[*simp*]:
assumes *compatible F G*
shows *register* (*F*; *G*)
by (*metis assms compatible-def register-pair-is-register*)

lemma *register-pair-apply*:
assumes ‹*compatible F G*›
shows ‹(*F*; *G*) (*a* ⊗*u* *b*) = (*F a*) ∗*u* (*G b*)›
apply (*rule register-pair-apply*)
using *assms* **unfolding** *compatible-def* **by** *metis*+

lemma *register-pair-apply*′:
assumes ‹*compatible F G*›
shows ‹(*F*; *G*) (*a* ⊗*u* *b*) = (*G b*) ∗*u* (*F a*)›
apply (*subst register-pair-apply*)
using *assms* **by** (*auto simp*: *compatible-def intro*: *register-preregister*)

lemma *compatible-comp-left*[*simp*]: *compatible F G* ⟹ *register H* ⟹ *compatible* (*F* ∘ *H*) *G*
by (*simp add*: *compatible-def*)

lemma *compatible-comp-right*[*simp*]: *compatible F G* ⟹ *register H* ⟹ *compatible F* (*G* ∘ *H*)
by (*simp add*: *compatible-def*)

lemma *compatible-comp-inner*[*simp*]:
compatible F G ⟹ *register H* ⟹ *compatible* (*H* ∘ *F*) (*H* ∘ *G*)
by (*smt* (*verit*, *best*) *comp-apply compatible-def register-comp register-mult*)

lemma *compatible-register1*: ‹*compatible F G* ⟹ *register F*›
by (*simp add*: *compatible-def*)
lemma *compatible-register2*: ‹*compatible F G* ⟹ *register G*›
by (*simp add*: *compatible-def*)

lemma *pair-o-tensor*:
assumes *compatible A B* **and** [*simp*]: ‹*register C*› **and** [*simp*]: ‹*register D*›
shows (*A*; *B*) *o* (*C* ⊗*r* *D*) = (*A o C*; *B o D*)
apply (*rule tensor-extensionality*)
using *assms* **by** (*simp-all add*: *register-tensor-is-register register-pair-apply comp-preregister*)

lemma *compatible-tensor-id-update-left*[*simp*]:

fixes $F :: 'a::type\ update \Rightarrow 'c::type\ update$ **and** $G :: 'b::type\ update \Rightarrow 'c::type\ update$
assumes *compatible F G*
shows *compatible* $(\lambda a.\ id\text{-}cblinfun \otimes_u F\ a)$ $(\lambda a.\ id\text{-}cblinfun \otimes_u G\ a)$
using *assms* **apply** (*rule compatible-comp-inner*[*unfolded o-def*])
by *simp*

lemma *compatible-tensor-id-update-right*[*simp*]:
fixes $F :: 'a::type\ update \Rightarrow 'c::type\ update$ **and** $G :: 'b::type\ update \Rightarrow 'c::type\ update$
assumes *compatible F G*
shows *compatible* $(\lambda a.\ F\ a \otimes_u id\text{-}cblinfun)$ $(\lambda a.\ G\ a \otimes_u id\text{-}cblinfun)$
using *assms* **apply** (*rule compatible-comp-inner*[*unfolded o-def*])
by *simp*

lemma *compatible-tensor-id-update-rl*[*simp*]:
assumes *register F* **and** *register G*
shows *compatible* $(\lambda a.\ F\ a \otimes_u id\text{-}cblinfun)$ $(\lambda a.\ id\text{-}cblinfun \otimes_u G\ a)$
apply (*rule compatibleI*)
using *assms* **by** (*auto simp*: *comp-tensor-op*)

lemma *compatible-tensor-id-update-lr*[*simp*]:
assumes *register F* **and** *register G*
shows *compatible* $(\lambda a.\ id\text{-}cblinfun \otimes_u F\ a)$ $(\lambda a.\ G\ a \otimes_u id\text{-}cblinfun)$
apply (*rule compatibleI*)
using *assms* **by** (*auto simp*: *comp-tensor-op*)

lemma *register-comp-pair*:
assumes [*simp*]: ‹*register F*› **and** [*simp*]: ‹*compatible G H*›
shows $(F\ o\ G;\ F\ o\ H) = F\ o\ (G;\ H)$
proof (*rule tensor-extensionality*)
show ‹*preregister* $(F \circ G;F \circ H)$› **and** ‹*preregister* $(F \circ (G;H))$›
by *simp-all*

have [*simp*]: ‹*compatible* $(F\ o\ G)$ $(F\ o\ H)$›
apply (*rule compatible-comp-inner*, *simp*)
by *simp*
then have [*simp*]: ‹*register* $(F \circ G)$› ‹*register* $(F \circ H)$›
unfolding *compatible-def* **by** *auto*
from *assms* **have** [*simp*]: ‹*register G*› ‹*register H*›
unfolding *compatible-def* **by** *auto*
fix *a b*
show ‹$(F \circ G;F \circ H)\ (a \otimes_u b) = (F \circ (G;H))\ (a \otimes_u b)$›
by (*auto simp*: *register-pair-apply register-mult comp-tensor-op*)
qed

lemma *swap-registers-left*:
assumes *compatible R S*
shows $R\ a *_u S\ b *_u c = S\ b *_u R\ a *_u c$
using *assms* **unfolding** *compatible-def* **by** *metis*

lemma *swap-registers-right*:
assumes *compatible R S*
shows $c *_u R\ a *_u S\ b = c *_u S\ b *_u R\ a$
by (*metis assms cblinfun-compose-assoc compatible-def*)

lemmas *compatible-ac-rules* = *swap-registers cblinfun-compose-assoc*[*symmetric*] *swap-registers-right*

10.6 Fst and Snd

definition *Fst* **where** ‹$Fst\ a = a \otimes_u id\text{-}cblinfun$›
definition *Snd* **where** ‹$Snd\ a = id\text{-}cblinfun \otimes_u a$›

lemma *register-Fst*[*simp*]: ‹*register Fst*›
unfolding *Fst-def* **by** (*rule register-tensor-left*)

lemma *register-Snd*[*simp*]: ‹*register Snd*›
unfolding *Snd-def* **by** (*rule register-tensor-right*)

lemma *compatible-Fst-Snd*[*simp*]: ‹*compatible Fst Snd*›
apply (*rule compatibleI*, *simp*, *simp*)
by (*simp add*: *Fst-def Snd-def comp-tensor-op*)

lemmas *compatible-Snd-Fst*[*simp*] = *compatible-Fst-Snd*[*THEN compatible-sym*]

definition ‹*swap* = (*Snd*; *Fst*)›

lemma *swap-apply*[*simp*]: *swap* (*a* ⊗$_u$ *b*) = (*b* ⊗$_u$ *a*)
unfolding *swap-def*
by (*simp add*: *Axioms-Quantum.register-pair-apply Fst-def Snd-def comp-tensor-op*)

lemma *swap-o-Fst*: *swap o Fst* = *Snd*
by (*auto simp add*: *Fst-def Snd-def*)
lemma *swap-o-Snd*: *swap o Snd* = *Fst*
by (*auto simp add*: *Fst-def Snd-def*)

lemma *register-swap*[*simp*]: ‹*register swap*›
by (*simp add*: *swap-def*)

lemma *pair-Fst-Snd*: ‹(*Fst*; *Snd*) = *id*›
apply (*rule tensor-extensionality*)
by (*simp-all add*: *register-pair-apply Fst-def Snd-def comp-tensor-op*)

lemma *swap-o-swap*[*simp*]: ‹*swap o swap* = *id*›
by (*metis swap-def compatible-Snd-Fst pair-Fst-Snd register-comp-pair register-swap swap-o-Fst swap-o-Snd*)

lemma *swap-swap*[*simp*]: ‹*swap* (*swap x*) = *x*›
by (*simp add*: *pointfree-idE*)

lemma *inv-swap*[*simp*]: ‹*inv swap* = *swap*›
by (*meson inv-unique-comp swap-o-swap*)

lemma *register-pair-Fst*:
assumes ‹*compatible F G*›
shows ‹(*F*;*G*) *o Fst* = *F*›
using *assms* **by** (*auto intro*!: *ext simp*: *Fst-def register-pair-apply compatible-register2*)

lemma *register-pair-Snd*:
assumes ‹*compatible F G*›
shows ‹(*F*;*G*) *o Snd* = *G*›
using *assms* **by** (*auto intro*!: *ext simp*: *Snd-def register-pair-apply compatible-register1*)

lemma *register-Fst-register-Snd*[*simp*]:
assumes ‹*register F*›
shows ‹(*F o Fst*; *F o Snd*) = *F*›
apply (*rule tensor-extensionality*)
using *assms* **by** (*auto simp*: *register-pair-apply Fst-def Snd-def register-mult comp-tensor-op*)

lemma *register-Snd-register-Fst*[*simp*]:
assumes ‹*register F*›
shows ‹(*F o Snd*; *F o Fst*) = *F o swap*›
apply (*rule tensor-extensionality*)
using *assms* **by** (*auto simp*: *register-pair-apply Fst-def Snd-def register-mult comp-tensor-op*)

lemma *compatible3*[*simp*]:
assumes [*simp*]: *compatible F G* **and** *compatible G H* **and** *compatible F H*
shows *compatible* (*F*; *G*) *H*

proof (*rule compatibleI*)
have [*simp*]: ‹*register F*› ‹*register G*› ‹*register H*›
using *assms compatible-def* **by** *auto*
then have [*simp*]: ‹*preregister F*› ‹*preregister G*› ‹*preregister H*›
using *register-preregister* **by** *blast+*
have [*simp*]: ‹*preregister* (λ*a*. (*F*;*G*) *a* ∗*u z*)› **for** *z*
apply (*rule comp-preregister*[*unfolded o-def*, *of* ‹(*F*;*G*)›])
by *simp-all*
have [*simp*]: ‹*preregister* (λ*a*. *z* ∗*u* (*F*;*G*) *a*)› **for** *z*
apply (*rule comp-preregister*[*unfolded o-def*, *of* ‹(*F*;*G*)›])
by *simp-all*
have (*F*; *G*) (*f* ⊗*u g*) ∗*u H h* = *H h* ∗*u* (*F*; *G*) (*f* ⊗*u g*) **for** *f g h*
proof −
have *FH*: *F f* ∗*u H h* = *H h* ∗*u F f*
using *assms compatible-def* **by** *metis*
have *GH*: *G g* ∗*u H h* = *H h* ∗*u G g*
using *assms compatible-def* **by** *metis*
have ‹(*F*; *G*) (*f* ⊗*u g*) ∗*u* (*H h*) = *F f* ∗*u G g* ∗*u H h*›
using ‹*compatible F G*› **by** (*subst register-pair-apply*, *auto*)
also have ‹. . . = *H h* ∗*u F f* ∗*u G g*›
using *FH GH* **by** (*metis cblinfun-compose-assoc*)
also have ‹. . . = *H h* ∗*u* (*F*; *G*) (*f* ⊗*u g*)›
using ‹*compatible F G*› **by** (*subst register-pair-apply*, *auto simp*: *cblinfun-compose-assoc*)
finally show *?thesis*
by −
qed
then show (*F*; *G*) *fg* ∗*u* (*H h*) = (*H h*) ∗*u* (*F*; *G*) *fg* **for** *fg h*
apply (*rule-tac tensor-extensionality*[*THEN fun-cong*])
by *auto*
show *register H* **and** *register* (*F*; *G*)
by *simp-all*
qed

lemma *compatible3'*[*simp*]:
assumes *compatible F G* **and** *compatible G H* **and** *compatible F H*
shows *compatible F* (*G*; *H*)
apply (*rule compatible-sym*)
apply (*rule compatible3*)
using *assms* **by** (*auto simp*: *compatible-sym*)

lemma *pair-o-swap*[*simp*]:
assumes [*simp*]: *compatible A B*
shows (*A*; *B*) *o swap* = (*B*; *A*)
proof (*rule tensor-extensionality*)
have [*simp*]: *preregister A preregister B*
apply (*metis* (*no-types*, *opaque-lifting*) *assms compatible-register1 register-preregister*)
by (*metis* (*full-types*) *assms compatible-register2 register-preregister*)
then show ‹*preregister* ((*A*; *B*) ∘ *swap*)›
by *simp*
show ‹*preregister* (*B*; *A*)›
by (*metis* (*no-types*, *lifting*) *assms compatible-sym register-preregister pair-is-register*)
show ‹((*A*; *B*) ∘ *swap*) (*a* ⊗*u b*) = (*B*; *A*) (*a* ⊗*u b*)› **for** *a b*

apply (*simp only*: *o-def swap-apply*)
apply (*subst register-pair-apply*, *simp*)
apply (*subst register-pair-apply*, *simp add*: *compatible-sym*)
by (*metis* (*no-types*, *lifting*) *assms compatible-def*)
qed

10.7 Compatibility of register tensor products

lemma *compatible-register-tensor*:
fixes *F* :: ‹'*a*::*type update* ⇒ '*e*::*type update*› **and** *G* :: ‹'*b*::*type update* ⇒ '*f*::*type update*›

```
    and F' :: ‹'c::type update ⇒ 'e update› and G' :: ‹'d::type update ⇒ 'f update›
  assumes [simp]: ‹compatible F F'›
  assumes [simp]: ‹compatible G G'›
  shows ‹compatible (F ⊗r G) (F' ⊗r G')›
proof −
  note [intro!] =
    comp-preregister[OF - preregister-mult-right, unfolded o-def]
    comp-preregister[OF - preregister-mult-left, unfolded o-def]
    comp-preregister
    register-tensor-is-register
  have [simp]: ‹register F› ‹register G› ‹register F'› ‹register G'›
    using assms compatible-def by blast+
  have [simp]: ‹register (F ⊗r G)› ‹register (F' ⊗r G')›
    by (auto simp add: register-tensor-def)
  have [simp]: ‹register (F;F')› ‹register (G;G')›
    by auto
  define reorder :: ‹(('a×'b) × ('c×'d)) update ⇒ (('a×'c) × ('b×'d)) update›
    where ‹reorder = ((Fst o Fst; Snd o Fst); (Fst o Snd; Snd o Snd))›
  have [simp]: ‹preregister reorder›
    by (auto simp: reorder-def)
  have [simp]: ‹reorder ((a ⊗u b) ⊗u (c ⊗u d)) = ((a ⊗u c) ⊗u (b ⊗u d))› for a b c d
    apply (simp add: reorder-def register-pair-apply)
    by (simp add: Fst-def Snd-def comp-tensor-op)
  define Φ where ‹Φ c d = ((F;F') ⊗r (G;G')) o reorder o (λσ. σ ⊗u (c ⊗u d))› for c d
  have [simp]: ‹preregister (Φ c d)› for c d
    unfolding Φ-def
    by (auto intro: register-preregister)
  have ‹Φ c d (a ⊗u b) = (F ⊗r G) (a ⊗u b) *u (F' ⊗r G') (c ⊗u d)› for a b c d
    unfolding Φ-def by (auto simp: register-pair-apply comp-tensor-op)
  then have Φ1: ‹Φ c d σ = (F ⊗r G) σ *u (F' ⊗r G') (c ⊗u d)› for c d σ
    apply (rule-tac fun-cong[of - - σ])
    apply (rule tensor-extensionality)
    by auto
  have ‹Φ c d (a ⊗u b) = (F' ⊗r G') (c ⊗u d) *u (F ⊗r G) (a ⊗u b)› for a b c d
    using assms
    unfolding Φ-def compatible-def by (auto simp: register-pair-apply comp-tensor-op)
  then have Φ2: ‹Φ c d σ = (F' ⊗r G') (c ⊗u d) *u (F ⊗r G) σ› for c d σ
    apply (rule-tac fun-cong[of - - σ])
    apply (rule tensor-extensionality)
    by auto
  from Φ1 Φ2 have ‹(F ⊗r G) σ *u (F' ⊗r G') τ = (F' ⊗r G') τ *u (F ⊗r G) σ› for τ σ
    apply (rule-tac fun-cong[of - - τ])
    apply (rule tensor-extensionality)
    by auto
  then show ?thesis
    apply (rule compatibleI[rotated −1])
    by auto
qed
```

10.8 Associativity of the tensor product

```
definition assoc :: ‹(('a::type×'b::type)×'c::type) update ⇒ ('a×('b×'c)) update› where
  ‹assoc = ((Fst; Snd o Fst); Snd o Snd)›

lemma assoc-is-hom[simp]: ‹preregister assoc›
  by (auto simp: assoc-def)

lemma assoc-apply[simp]: ‹assoc ((a ⊗u b) ⊗u c) = (a ⊗u (b ⊗u c))›
  by (auto simp: assoc-def register-pair-apply Fst-def Snd-def comp-tensor-op)

definition assoc' :: ‹('a×('b×'c)) update ⇒ (('a::type×'b::type)×'c::type) update› where
  ‹assoc' = (Fst o Fst; (Fst o Snd; Snd))›
```

lemma *assoc′-is-hom*[*simp*]: ‹*preregister assoc′*›
by (*auto simp*: *assoc′-def*)

lemma *assoc′-apply*[*simp*]: ‹*assoc′* (*a* ⊗*u* (*b* ⊗*u* *c*)) = ((*a* ⊗*u* *b*) ⊗*u* *c*)›
by (*auto simp*: *assoc′-def register-pair-apply Fst-def Snd-def comp-tensor-op*)

lemma *register-assoc*[*simp*]: ‹*register assoc*›
unfolding *assoc-def*
by *force*

lemma *register-assoc′*[*simp*]: ‹*register assoc′*›
unfolding *assoc′-def*
by *force*

lemma *pair-o-assoc*[*simp*]:
assumes [*simp*]: ‹*compatible F G*› ‹*compatible G H*› ‹*compatible F H*›
shows ‹(*F*; (*G*; *H*)) ∘ *assoc* = ((*F*; *G*); *H*)›
proof (*rule tensor-extensionality3′*)
show ‹*register* ((*F*; (*G*; *H*)) ∘ *assoc*)›
by *simp*
show ‹*register* ((*F*; *G*); *H*)›
by *simp*
show ‹((*F*; (*G*; *H*)) ∘ *assoc*) ((*f* ⊗*u* *g*) ⊗*u* *h*) = ((*F*; *G*); *H*) ((*f* ⊗*u* *g*) ⊗*u* *h*)› **for** *f g h*
by (*simp add*: *register-pair-apply assoc-apply cblinfun-compose-assoc*)
qed

lemma *pair-o-assoc′*[*simp*]:
assumes [*simp*]: ‹*compatible F G*› ‹*compatible G H*› ‹*compatible F H*›
shows ‹((*F*; *G*); *H*) ∘ *assoc′* = (*F*; (*G*; *H*))›
proof (*rule tensor-extensionality3*)
show ‹*register* (((*F*; *G*); *H*) ∘ *assoc′*)›
by *simp*
show ‹*register* (*F*; (*G*; *H*))›
by *simp*
show ‹(((*F*; *G*); *H*) ∘ *assoc′*) (*f* ⊗*u* *g* ⊗*u* *h*) = (*F*; (*G*; *H*)) (*f* ⊗*u* *g* ⊗*u* *h*)› **for** *f g h*
by (*simp add*: *register-pair-apply assoc′-apply cblinfun-compose-assoc*)
qed

lemma *assoc′-o-assoc*[*simp*]: ‹*assoc′ o assoc* = *id*›
apply (*rule tensor-extensionality3′*)
by *auto*

lemma *assoc′-assoc*[*simp*]: ‹*assoc′* (*assoc x*) = *x*›
by (*simp add*: *pointfree-idE*)

lemma *assoc-o-assoc′*[*simp*]: ‹*assoc o assoc′* = *id*›
apply (*rule tensor-extensionality3*)
by *auto*

lemma *assoc-assoc′*[*simp*]: ‹*assoc* (*assoc′ x*) = *x*›
by (*simp add*: *pointfree-idE*)

lemma *inv-assoc*[*simp*]: ‹*inv assoc* = *assoc′*›
using *assoc′-o-assoc assoc-o-assoc′ inv-unique-comp* **by** *blast*

lemma *inv-assoc′*[*simp*]: ‹*inv assoc′* = *assoc*›
by (*simp add*: *inv-equality*)

lemma *bij-assoc*[*simp*]: ‹*bij assoc*›
using *assoc′-o-assoc assoc-o-assoc′ o-bij* **by** *blast*

lemma *bij-assoc′*[*simp*]: ‹*bij assoc′*›
using *assoc′-o-assoc assoc-o-assoc′ o-bij* **by** *blast*

10.9 Iso-registers

definition ‹*iso-register F* ⟷ *register F* ∧ (∃ *G*. *register G* ∧ *F o G* = *id* ∧ *G o F* = *id*)›
for *F* :: ‹-::*type update* ⇒ -::*type update*›

lemma *iso-registerI*:
assumes ‹*register F*› ‹*register G*› ‹*F o G* = *id*› ‹*G o F* = *id*›
shows ‹*iso-register F*›
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *iso-register-def* **by** *blast*

lemma *iso-register-inv*: ‹*iso-register F* ⟹ *iso-register* (*inv F*)›
by (*metis inv-unique-comp iso-register-def*)

lemma *iso-register-inv-comp1*: ‹*iso-register F* ⟹ *inv F o F* = *id*›
using *inv-unique-comp iso-register-def* **by** *blast*

lemma *iso-register-inv-comp2*: ‹*iso-register F* ⟹ *F o inv F* = *id*›
using *inv-unique-comp iso-register-def* **by** *blast*

lemma *iso-register-id*[*simp*]: ‹*iso-register id*›
by (*simp add*: *iso-register-def*)

lemma *iso-register-is-register*: ‹*iso-register F* ⟹ *register F*›
using *iso-register-def* **by** *blast*

lemma *iso-register-comp*[*simp*]:
assumes [*simp*]: ‹*iso-register F*› ‹*iso-register G*›
shows ‹*iso-register* (*F o G*)›
proof −
from *assms* **obtain** *F′ G′* **where** [*simp*]: ‹*register F′*› ‹*register G′*› ‹*F o F′* = *id*› ‹*F′ o F* = *id*›
‹*G o G′* = *id*› ‹*G′ o G* = *id*›
by (*meson iso-register-def*)
have *1*: ‹*F* ∘ *G* ∘ (*G′* ∘ *F′*) = *id*›
by (*metis* ‹*F* ∘ *F′* = *id*› ‹*G* ∘ *G′* = *id*› *fcomp-assoc fcomp-comp id-fcomp*)
have *2*: ‹*G′* ∘ *F′* ∘ (*F* ∘ *G*) = *id*›
by (*metis* (*no-types*, *lifting*) ‹*F* ∘ *F′* = *id*› ‹*F′* ∘ *F* = *id*› ‹*G′* ∘ *G* = *id*› *fun.map-comp inj-iff inv-unique-comp o-inv-o-cancel*)

show *?thesis*
apply (*rule iso-registerI*[**where** *G*=‹*G′ o F′*›])
using *1 2* **by** (*auto simp*: *register-tensor-is-register iso-register-is-register register-tensor-distrib*)
qed

lemma *iso-register-tensor-is-iso-register*[*simp*]:
assumes [*simp*]: ‹*iso-register F*› ‹*iso-register G*›
shows ‹*iso-register* (*F* ⊗*r G*)›
proof −
from *assms* **obtain** *F′ G′* **where** [*simp*]: ‹*register F′*› ‹*register G′*› ‹*F o F′* = *id*› ‹*F′ o F* = *id*›
‹*G o G′* = *id*› ‹*G′ o G* = *id*›
by (*meson iso-register-def*)
show *?thesis*
apply (*rule iso-registerI*[**where** *G*=‹*F′* ⊗*r G′*›])
by (*auto simp*: *register-tensor-is-register iso-register-is-register register-tensor-distrib*)
qed

lemma *iso-register-bij*: ‹*iso-register F* ⟹ *bij F*›
using *iso-register-def o-bij* **by** *auto*

lemma *inv-register-tensor*[*simp*]:
assumes [*simp*]: ‹*iso-register F*› ‹*iso-register G*›
shows ‹*inv* (*F* ⊗*r G*) = *inv F* ⊗*r inv G*›
apply (*auto intro*!: *inj-imp-inv-eq bij-is-inj iso-register-bij*

```
        simp: register-tensor-distrib[unfolded o-def, THEN fun-cong] iso-register-is-register
          iso-register-inv bij-is-surj iso-register-bij surj-f-inv-f)
  by (metis eq-id-iff register-tensor-id)

lemma iso-register-swap[simp]: ‹iso-register swap›
  apply (rule iso-registerI[of - swap])
  by auto

lemma iso-register-assoc[simp]: ‹iso-register assoc›
  apply (rule iso-registerI[of - assoc'])
  by auto

lemma iso-register-assoc'[simp]: ‹iso-register assoc'›
  apply (rule iso-registerI[of - assoc])
  by auto

definition ‹equivalent-registers F G ⟷ (register F ∧ (∃ I. iso-register I ∧ F o I = G))›
  for F G :: ‹-::type update ⇒ -::type update›

lemma iso-register-equivalent-id[simp]: ‹equivalent-registers id F ⟷ iso-register F›
  by (simp add: equivalent-registers-def)

lemma equivalent-registersI:
  assumes ‹register F›
  assumes ‹iso-register I›
  assumes ‹F o I = G›
  shows ‹equivalent-registers F G›
  using assms unfolding equivalent-registers-def by blast

lemma equivalent-registers-refl: ‹equivalent-registers F F› if ‹register F›
  using that by (auto intro!: exI[of - id] simp: equivalent-registers-def)

lemma equivalent-registers-register-left: ‹equivalent-registers F G ⟹ register F›
  using equivalent-registers-def by auto

lemma equivalent-registers-register-right: ‹register G› if ‹equivalent-registers F G›
  by (metis equivalent-registers-def iso-register-def register-comp that)

lemma equivalent-registers-sym:
  assumes ‹equivalent-registers F G›
  shows ‹equivalent-registers G F›
 by (smt (verit) assms comp-id equivalent-registers-def equivalent-registers-register-right fun.map-comp iso-register-def)

lemma equivalent-registers-trans[trans]:
  assumes ‹equivalent-registers F G› and ‹equivalent-registers G H›
  shows ‹equivalent-registers F H›
proof −
  from assms have [simp]: ‹register F› ‹register G›
    by (auto simp: equivalent-registers-def)
  from assms(1) obtain I where [simp]: ‹iso-register I› and ‹F o I = G›
    using equivalent-registers-def by blast
  from assms(2) obtain J where [simp]: ‹iso-register J› and ‹G o J = H›
    using equivalent-registers-def by blast
  have ‹register F›
    by (auto simp: equivalent-registers-def)
  moreover have ‹iso-register (I o J)›
    using ‹iso-register I› ‹iso-register J› iso-register-comp by blast
  moreover have ‹F o (I o J) = H›
    by (simp add: ‹F ∘ I = G› ‹G ∘ J = H› o-assoc)
  ultimately show ?thesis
    by (rule equivalent-registersI)
qed
```

lemma *equivalent-registers-assoc*[*simp*]:
assumes [*simp*]: ‹*compatible F G*› ‹*compatible F H*› ‹*compatible G H*›
shows ‹*equivalent-registers* (*F*;(*G*;*H*)) ((*F*;*G*);*H*)›
apply (*rule equivalent-registersI*[**where** *I*=*assoc*])
by *auto*

lemma *equivalent-registers-pair-right*:
assumes [*simp*]: ‹*compatible F G*›
assumes *eq*: ‹*equivalent-registers G H*›
shows ‹*equivalent-registers* (*F*;*G*) (*F*;*H*)›
proof −
from *eq* **obtain** *I* **where** [*simp*]: ‹*iso-register I*› **and** ‹*G o I* = *H*›
by (*metis equivalent-registers-def*)
then have ∗: ‹(*F*;*G*) ∘ (*id* ⊗*r I*) = (*F*;*H*)›
by (*auto intro*!: *tensor-extensionality register-comp register-preregister register-tensor-is-register*
simp: *register-pair-apply iso-register-is-register*)
show *?thesis*
apply (*rule equivalent-registersI*[**where** *I*=‹*id* ⊗*r I*›])
using ∗ **by** (*auto intro*!: *iso-register-tensor-is-iso-register*)
qed

lemma *equivalent-registers-pair-left*:
assumes [*simp*]: ‹*compatible F G*›
assumes *eq*: ‹*equivalent-registers F H*›
shows ‹*equivalent-registers* (*F*;*G*) (*H*;*G*)›
proof −
from *eq* **obtain** *I* **where** [*simp*]: ‹*iso-register I*› **and** ‹*F o I* = *H*›
by (*metis equivalent-registers-def*)
then have ∗: ‹(*F*;*G*) ∘ (*I* ⊗*r id*) = (*H*;*G*)›
by (*auto intro*!: *tensor-extensionality register-comp register-preregister register-tensor-is-register*
simp: *register-pair-apply iso-register-is-register*)
show *?thesis*
apply (*rule equivalent-registersI*[**where** *I*=‹*I* ⊗*r id*›])
using ∗ **by** (*auto intro*!: *iso-register-tensor-is-iso-register*)
qed

lemma *equivalent-registers-comp*:
assumes ‹*register H*›
assumes ‹*equivalent-registers F G*›
shows ‹*equivalent-registers* (*H o F*) (*H o G*)›
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *comp-assoc equivalent-registers-def register-comp*)

lemma *equivalent-registers-compatible1*:
assumes ‹*compatible F G*›
assumes ‹*equivalent-registers F F'*›
shows ‹*compatible F' G*›
by (*metis assms*(*1*) *assms*(*2*) *compatible-comp-left equivalent-registers-def iso-register-is-register*)

lemma *equivalent-registers-compatible2*:
assumes ‹*compatible F G*›
assumes ‹*equivalent-registers G G'*›
shows ‹*compatible F G'*›
by (*metis assms*(*1*) *assms*(*2*) *compatible-comp-right equivalent-registers-def iso-register-is-register*)

10.10 Compatibility simplification

The simproc *compatibility-warn* produces helpful warnings for subgoals of the form *compatible x y* that are probably unsolvable due to missing declarations of variable compatibility facts. Same for subgoals of the form *register x*.

simproc-setup *compatibility-warn* (*compatible x y* | *register x*) = ‹
let val thy-string = *Markup.markup* (*Theory.get-markup* ***theory***) (*Context.theory-base-name* ***theory***)
in
fn m => fn ctxt => fn ct => let

```
  val (x,y) = case Thm.term-of ct of
                Const(const-name‹compatible›,- ) $ x $ y => (x, SOME y)
              | Const(const-name‹register›,- ) $ x => (x, NONE)
  val str : string lazy = Lazy.lazy (fn () => Syntax.string-of-term ctxt (Thm.term-of ct))
 fun w msg = warning (msg ^\n(Disable these warnings with: using [[simproc del: ^thy-string^.compatibility-warn]]))
  val - = case (x,y) of
       (Free(n,T), SOME (Free(n',T'))) =>
          if String.isPrefix : n orelse String.isPrefix : n' then
                  w (Simplification subgoal  ^ Lazy.force str ^  contains a bound variable.\n ^
                   Try to add some assumptions that makes this goal solvable by the simplifier)
          else if n=n' then (if T=T' then ()
                       else w (In simplification subgoal  ^ Lazy.force str ^
                            , variables have same name and different types.\n ^
                            Probably something is wrong.))
                  else w (Simplification subgoal  ^ Lazy.force str ^
                           occurred but cannot be solved.\n ^
                          Please add assumption/fact  [simp]: ‹ ^ Lazy.force str ^
                          ›  somewhere.)
     | (Free(n,T), NONE) =>
          if String.isPrefix : n then
                  w (Simplification subgoal ' ^ Lazy.force str ^ ' contains a bound variable.\n ^
                   Try to add some assumptions that makes this goal solvable by the simplifier)
          else w (Simplification subgoal  ^ Lazy.force str ^  occurred but cannot be solved.\n ^
                 Please add assumption/fact  [simp]: ‹ ^ Lazy.force str ^ ›  somewhere.)
     | - => ()
  in NONE end
end›


named-theorems register-attribute-rule-immediate
named-theorems register-attribute-rule

lemmas [register-attribute-rule] = conjunct1 conjunct2 iso-register-is-register iso-register-is-register[OF iso-register-inv]
lemmas [register-attribute-rule-immediate] = compatible-sym compatible-register1 compatible-register2
  asm-rl[of ‹compatible - ›] asm-rl[of ‹iso-register ›] asm-rl[of ‹register ›] iso-register-inv
```

The following declares an attribute [*register*]. When the attribute is applied to a fact of the form *register F*, *iso-register F*, *compatible F G* or a conjunction of these, then those facts are added to the simplifier together with some derived theorems (e.g., *compatible F G* also adds *register F*).

In theory *Laws-Complement*, support for *is-unit-register F* and *complements F G* is added to this attribute.

```
setup ‹
let
fun add thm results =
  Net.insert-term (K true) (Thm.concl-of thm, thm) results
  handle Net.INSERT => results
fun try-rule f thm rule state = case SOME (rule OF [thm]) handle THM - => NONE  of
  NONE => state | SOME th => f th state
fun collect (rules,rules-immediate) thm results =
  results |> fold (try-rule add thm) rules-immediate |> fold (try-rule (collect (rules,rules-immediate)) thm) rules
fun declare thm context = let
  val ctxt = Context.proof-of context
  val rules = Named-Theorems.get ctxt @{named-theorems register-attribute-rule}
  val rules-immediate = Named-Theorems.get ctxt @{named-theorems register-attribute-rule-immediate}
  val thms = collect (rules,rules-immediate) thm Net.empty |> Net.entries
  (* val - = print thms *)
  in Simplifier.map-ss (fn ctxt => ctxt addsimps thms) context end
in
Attrib.setup binding‹register›
 (Scan.succeed (Thm.declaration-attribute declare))
  Add register−related rules to the simplifier
end
›
```

10.11 Notation

no-notation *cblinfun-compose* (**infixl** $*_u$ *55*)
no-notation *tensor-op* (**infixr** $\otimes_u$ *70*)

bundle *register-syntax* **begin**
notation *register-tensor* (**infixr** $\otimes_r$ *70*)
notation *register-pair* (‴(-;-‴))
end

end

11 Quantum mechanics basics

theory *Quantum*
imports
Misc
Hilbert-Space-Tensor-Product.Hilbert-Space-Tensor-Product
HOL−Library.Z2
Jordan-Normal-Form.Matrix-Impl
Real-Impl.Real-Impl
HOL−Library.Code-Target-Numeral
begin

unbundle *cblinfun-syntax*

type-synonym ('*a*,'*b*) *matrix* = ‹('*a ell2*, '*b ell2*) *cblinfun*›

11.1 Basic quantum states

11.1.1 EPR pair

definition *vector-β00* = *vec-of-list* [*1*/*sqrt 2*::*complex*, *0*, *0*, *1*/*sqrt 2*]
definition *β00* :: ‹(*bit*×*bit*) *ell2*› **where** [*code del*]: *β00* = *basis-enum-of-vec vector-β00*
lemma *vec-of-basis-enum-β00*[*simp*]: *vec-of-basis-enum β00* = *vector-β00*
by (*auto simp add*: *β00-def vector-β00-def*)
lemma *vec-of-ell2-β00*[*simp*, *code*]: *vec-of-ell2 β00* = *vector-β00*
by (*simp add*: *vec-of-ell2-def*)

lemma *norm-β00*[*simp*]: *norm β00* = *1*
by *eval*

11.1.2 Ket plus

definition *vector-ketplus* = *vec-of-list* [*1*/*sqrt 2*::*complex*, *1*/*sqrt 2*]
definition *ketplus* :: ‹*bit ell2*› (|+⟩) **where** [*code del*]: ‹*ketplus* = *basis-enum-of-vec vector-ketplus*›
lemma *vec-of-basis-enum-ketplus*[*simp*]: *vec-of-basis-enum ketplus* = *vector-ketplus*
by (*auto simp add*: *ketplus-def vector-ketplus-def*)
lemma *vec-of-ell2-ketplus*[*simp*, *code*]: *vec-of-ell2 ketplus* = *vector-ketplus*
by (*simp add*: *vec-of-ell2-def*)

11.2 Basic quantum gates

11.2.1 Pauli X

definition *matrix-pauliX* = *mat-of-rows-list 2* [[*0*::*complex*, *1*], [*1*, *0*]]
definition *pauliX* :: ‹(*bit*, *bit*) *matrix*› **where** [*code del*]: *pauliX* = *cblinfun-of-mat matrix-pauliX*
lemma *mat-of-cblinfun-pauliX*[*simp*, *code*]: *mat-of-cblinfun pauliX* = *matrix-pauliX*
by (*auto simp add*: *pauliX-def matrix-pauliX-def cblinfun-of-mat-inverse*)

derive (*eq*) *ceq bit*

instantiation *bit* :: *ccompare* **begin**

definition *CCOMPARE(bit) = Some (λb1 b2. case (b1, b2) of (0, 0) ⇒ order.Eq | (0, 1) ⇒ order.Lt | (1, 0) ⇒ order.Gt | (1, 1) ⇒ order.Eq)*
instance
by *intro-classes(unfold-locales; auto simp add: ccompare-bit-def split!: bit.splits)*
end

derive (*dlist*) *set-impl bit*

lemma *pauliX-adjoint[simp]: pauliX∗ = pauliX*
by *eval*
lemma *pauliXX[simp]: pauliX o_{CL} pauliX = id-cblinfun*
by *eval*

11.2.2 Pauli Z

definition *matrix-pauliZ = mat-of-rows-list 2 [[1::complex, 0], [0, −1]]*
definition *pauliZ :: ‹(bit, bit) matrix› where [code del]: pauliZ = cblinfun-of-mat matrix-pauliZ*
lemma *mat-of-cblinfun-pauliZ[simp, code]: mat-of-cblinfun pauliZ = matrix-pauliZ*
by (*auto simp add: pauliZ-def matrix-pauliZ-def cblinfun-of-mat-inverse*)
lemma *pauliZ-adjoint[simp]: pauliZ∗ = pauliZ*
by *eval*
lemma *pauliZZ[simp]: pauliZ o_{CL} pauliZ = id-cblinfun*
by *eval*

11.2.3 Hadamard

definition *matrix-hadamard = mat-of-rows-list 2 [[1/sqrt 2::complex, 1/sqrt 2], [1/sqrt 2, −1/sqrt 2]]*
definition *hadamard :: ‹(bit,bit) matrix› where [code del]: hadamard = cblinfun-of-mat matrix-hadamard*

lemma *mat-of-cblinfun-hadamard[simp, code]: mat-of-cblinfun hadamard = matrix-hadamard*
by (*auto simp add: hadamard-def matrix-hadamard-def cblinfun-of-mat-inverse*)

lemma *hada-adj[simp]: hadamard∗ = hadamard*
by *eval*

11.2.4 CNOT

definition *matrix-CNOT = mat-of-rows-list 4 [[1::complex,0,0,0], [0,1,0,0], [0,0,0,1], [0,0,1,0]]*
definition *CNOT :: ‹(bit∗bit, bit∗bit) matrix› where [code del]: CNOT = cblinfun-of-mat matrix-CNOT*

lemma *mat-of-cblinfun-CNOT[simp, code]: mat-of-cblinfun CNOT = matrix-CNOT*
by (*auto simp add: CNOT-def matrix-CNOT-def cblinfun-of-mat-inverse*)

lemma *CNOT-adj[simp]: CNOT∗ = CNOT*
by *eval*

lemma *cnot-apply[simp]: ‹CNOT $*_V$ ket (i,j) = ket (i,j+i)›*
apply (*rule spec*[**where** *x=i*], *rule spec*[**where** *x=j*])
by *eval*

11.2.5 Qubit swap

definition *matrix-Uswap = mat-of-rows-list 4 [[1::complex, 0, 0, 0], [0,0,1,0], [0,1,0,0], [0,0,0,1]]*
definition *Uswap :: ‹(bit×bit, bit×bit) matrix› where*
[*code del*]: *‹Uswap = cblinfun-of-mat matrix-Uswap›*

lemma *mat-of-cblinfun-Uswap[simp, code]: mat-of-cblinfun Uswap = matrix-Uswap*
by (*auto simp add: Uswap-def matrix-Uswap-def cblinfun-of-mat-inverse*)

lemma *dim-col-Uswap[simp]: dim-col matrix-Uswap = 4*
unfolding *matrix-Uswap-def* **by** *simp*
lemma *dim-row-Uswap[simp]: dim-row matrix-Uswap = 4*
unfolding *matrix-Uswap-def* **by** *simp*
lemma *Uswap-adjoint[simp]: Uswap∗ = Uswap*

by *eval*
lemma *Uswap-involution*[*simp*]: *Uswap* o_{CL} *Uswap* = *id-cblinfun*
by *eval*
lemma *unitary-Uswap*[*simp*]: *unitary Uswap*
unfolding *unitary-def* **by** *simp*

lemma *Uswap-apply*[*simp*]: ‹*Uswap* $*_V$ *s* $\otimes_s$ *t* = *t* $\otimes_s$ *s*›
apply (*rule clinear-equal-ket*[**where** *f*=‹λ*s*. *Uswap* $*_V$ *s* $\otimes_s$ *t*›, *THEN fun-cong*])
apply (*auto simp add*: *cblinfun.add-right tensor-ell2-add1 tensor-ell2-scaleC1*
cblinfun.scaleC-right tensor-ell2-add2 tensor-ell2-scaleC2
intro!: *clinearI*)[2]
apply (*rule clinear-equal-ket*[**where** *f*=‹λ*t*. *Uswap* $*_V$ - $\otimes_s$ *t*›, *THEN fun-cong*])
apply (*auto simp add*: *cblinfun.add-right tensor-ell2-add1 tensor-ell2-scaleC1*
cblinfun.scaleC-right tensor-ell2-add2 tensor-ell2-scaleC2
intro!: *clinearI*)[2]
apply (*rule basis-enum-eq-vec-of-basis-enumI*)
apply (*simp add*: *mat-of-cblinfun-cblinfun-apply vec-of-basis-enum-ket tensor-ell2-ket*)
by (*case-tac i*; *case-tac ia*; *hypsubst-thin*; *normalization*)

unbundle *no cblinfun-syntax*

end

12 Derived facts about quantum registers

theory *Quantum-Extra*
imports
Laws-Quantum
Quantum
begin

no-notation *meet* (**infixl** $\sqcap_1$ *70*)
no-notation *Group.mult* (**infixl** $\otimes_1$ *70*)
no-notation *Order.top* ($\top_1$)
unbundle *lattice-syntax*
unbundle *register-syntax*
unbundle *cblinfun-syntax*

lemma *zero-not-register*[*simp*]: ‹$\sim$ *register* (λ-. *0*)›
unfolding *register-def* **by** *simp*

lemma *register-pair-is-register-converse*:
‹*register* (*F*;*G*) $\Longrightarrow$ *register F*› ‹*register* (*F*;*G*) $\Longrightarrow$ *register G*›
using [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
apply (*cases* ‹*register F*›)
apply (*auto simp*: *register-pair-def*)[2]
apply (*cases* ‹*register G*›)
by (*auto simp*: *register-pair-def*)[2]

lemma *register-id'*[*simp*]: ‹*register* (λ*x*. *x*)›
using *register-id* **by** (*simp add*: *id-def*)

lemma *compatible-proj-intersect*:

assumes *compatible R S* **and** *is-Proj a* **and** *is-Proj b*
shows (*R a* $*_S$ $\top$) $\sqcap$ (*S b* $*_S$ $\top$) = ((*R a* o_{CL} *S b*) $*_S$ $\top$)
proof (*rule antisym*)
have ((*R a* o_{CL} *S b*) $*_S$ $\top$) $\leq$ (*S b* $*_S$ $\top$)
apply (*subst swap-registers*[*OF assms*(*1*)])
by (*simp add*: *cblinfun-compose-image cblinfun-image-mono*)
moreover have ((*R a* o_{CL} *S b*) $*_S$ $\top$) $\leq$ (*R a* $*_S$ $\top$)
by (*simp add*: *cblinfun-compose-image cblinfun-image-mono*)
ultimately show ‹((*R a* o_{CL} *S b*) $*_S$ $\top$) $\leq$ (*R a* $*_S$ $\top$) $\sqcap$ (*S b* $*_S$ $\top$)›

by *auto*

have *is-Proj* (*R a*)
using *assms*(*1*) *assms*(*2*) *compatible-register1 register-projector* **by** *blast*
have *is-Proj* (*S b*)
using *assms*(*1*) *assms*(*3*) *compatible-register2 register-projector* **by** *blast*
show ‹(*R a* $*_S$ ⊤) ⊓ (*S b* $*_S$ ⊤) ≤ (*R a* o_{CL} *S b*) $*_S$ ⊤›
proof (*unfold less-eq-ccsubspace.rep-eq*, *rule subsetI*)
fix *ψ*
assume *asm*: ‹*ψ* ∈ *space-as-set* ((*R a* $*_S$ ⊤) ⊓ (*S b* $*_S$ ⊤))›
then have ‹*ψ* ∈ *space-as-set* (*R a* $*_S$ ⊤)›
by *auto*
then have *R*: ‹*R a* $*_V$ *ψ* = *ψ*›
using ‹*is-Proj* (*R a*)› *cblinfun-fixes-range is-Proj-algebraic* **by** *blast*
from *asm* **have** ‹*ψ* ∈ *space-as-set* (*S b* $*_S$ ⊤)›
by *auto*
then have *S*: ‹*S b* $*_V$ *ψ* = *ψ*›
using ‹*is-Proj* (*S b*)› *cblinfun-fixes-range is-Proj-algebraic* **by** *blast*
from *R S* **have** ‹*ψ* = (*R a* o_{CL} *S b*) $*_V$ *ψ*›
by (*simp add*: *cblinfun-apply-cblinfun-compose*)
also have ‹... ∈ *space-as-set* ((*R a* o_{CL} *S b*) $*_S$ ⊤)›
apply *simp* **by** (*metis R S calculation cblinfun-apply-in-image*)
finally show ‹*ψ* ∈ *space-as-set* ((*R a* o_{CL} *S b*) $*_S$ ⊤)›
by −
qed
qed

lemma *compatible-proj-mult*:
assumes *compatible R S* **and** *is-Proj a* **and** *is-Proj b*
shows *is-Proj* (*R a* o_{CL} *S b*)
proof −
have *aux*: ‹∀ *a b*. *R a* o_{CL} *S b* = *S b* o_{CL} *R a* ⟹ *S b* o_{CL} *R a* o_{CL} (*S b* o_{CL} *R a*) = *S b* o_{CL} *R a*›
using *assms*
by (*metis* (*no-types*, *lifting*) *cblinfun-compose-assoc register-mult is-Proj-algebraic compatible-def*)
show *?thesis*
using [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
using *assms* **unfolding** *is-Proj-algebraic compatible-def*
by (*auto simp add*: *assms is-proj-selfadj register-projector aux*)
qed

lemma *sandwich-tensor*:
fixes *a* :: ‹'*a ell2* ⇒$_{CL}$ '*a ell2*› **and** *b* :: ‹'*b ell2* ⇒$_{CL}$ '*b ell2*›
assumes [*simp*]: ‹*unitary a*› ‹*unitary b*›
shows ($*_V$) (*sandwich* (*a* ⊗$_o$ *b*)) = *sandwich a* ⊗$_r$ *sandwich b*
apply (*rule tensor-extensionality*)
by (*auto simp*: *unitary-sandwich-register sandwich-apply register-tensor-is-register*
comp-tensor-op tensor-op-adjoint unitary-tensor-op intro!: *register-preregister unitary-sandwich-register*)

lemma *sandwich-grow-left*:
fixes *a* :: ‹'*a ell2* ⇒$_{CL}$ '*a ell2*›
assumes *unitary a*
shows *sandwich a* ⊗$_r$ *id* = *sandwich* (*a* ⊗$_o$ *id-cblinfun*)
by (*simp add*: *unitary-sandwich-register sandwich-tensor assms id-def*)

lemma *register-sandwich*: ‹*register F* ⟹ *F* (*sandwich a b*) = *sandwich* (*F a*) (*F b*)›
by (*smt* (*verit*, *del-insts*) *register-def sandwich-apply*)

lemma *assoc-ell2-sandwich*: ‹*assoc* = *sandwich assoc-ell2*›
apply (*rule tensor-extensionality3'*)
apply (*simp-all add*: *unitary-sandwich-register*)[2]
apply (*rule equal-ket*)
apply (*case-tac x*)
by (*simp add*: *sandwich-apply assoc-apply cblinfun-apply-cblinfun-compose tensor-op-ell2 assoc-ell2-tensor as-*

soc-ell2'-tensor
flip: *tensor-ell2-ket*)

lemma *assoc-ell2'-sandwich*: ‹*assoc'* = *sandwich* (*assoc-ell2**)›
apply (*rule tensor-extensionality3*)
apply (*simp-all add*: *unitary-sandwich-register*)[2]
apply (*rule equal-ket*)
apply (*case-tac x*)
by (*simp add*: *sandwich-apply assoc'-apply cblinfun-apply-cblinfun-compose tensor-op-ell2 assoc-ell2-tensor assoc-ell2'-tensor*
flip: *tensor-ell2-ket*)

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

lemma *id-tensor-sandwich*:
fixes *a* :: *'a*::*finite ell2* ⇒$_{CL}$ *'b*::*finite ell2*
assumes *unitary a*
shows *id* ⊗$_r$ *sandwich a* = *sandwich* (*id-cblinfun* ⊗$_o$ *a*)
apply (*rule tensor-extensionality*)
using *assms*
by (*auto simp*: *register-tensor-is-register comp-tensor-op sandwich-apply tensor-op-adjoint unitary-sandwich-register*
intro!: *register-preregister unitary-sandwich-register unitary-tensor-op*)

lemma *compatible-selfbutter-join*:
assumes [*register*]: *compatible R S*
shows *R* (*selfbutter* *ψ*) *o*$_{CL}$ *S* (*selfbutter* *φ*) = (*R*; *S*) (*selfbutter* (*ψ* ⊗$_s$ *φ*))
apply (*subst register-pair-apply*[*symmetric*, **where** *F*=*R* **and** *G*=*S*])
using *assms* **by** (*auto simp*: *tensor-butterfly*)

lemma *register-mult'*:
assumes ‹*register F*›
shows ‹*F a* *$_V$ *F b* *$_V$ *c* = *F* (*a* *o*$_{CL}$ *b*) *$_V$ *c*›
by (*simp add*: *assms lift-cblinfun-comp*(4) *register-mult*)

lemma *register-scaleC*:
assumes ‹*register F*› **shows** ‹*F* (*c* *$_C$ *a*) = *c* *$_C$ *F a*›
using *assms* [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
unfolding *register-def*
by (*simp add*: *bounded-clinear.clinear clinear.scaleC*)

lemma *register-adjoint*: *F* (*a**) = (*F a*)* **if** ‹*register F*›
using *register-def that* **by** *blast*

lemma *register-finite-dim*: ‹*register F* ⟷ *clinear F* ∧ *F id-cblinfun* = *id-cblinfun* ∧ (∀ *a b*. *F* (*a* *o*$_{CL}$ *b*) = *F a* *o*$_{CL}$ *F b*) ∧ (∀ *a*. *F* (*a**) = *F a**)›
for *F* :: ‹*'a*::*finite update* ⇒ *'b*::*finite update*›

proof
assume ‹*register F*›
then show ‹*clinear F* ∧ *F id-cblinfun* = *id-cblinfun* ∧ (∀ *a b*. *F* (*a* *o*$_{CL}$ *b*) = *F a* *o*$_{CL}$ *F b*) ∧ (∀ *a*. *F* (*a**) = *F a**)›
unfolding *register-def*
by (*auto simp add*: *bounded-clinear-def*)
next
assume *asm*: ‹*clinear F* ∧ *F id-cblinfun* = *id-cblinfun* ∧ (∀ *a b*. *F* (*a* *o*$_{CL}$ *b*) = *F a* *o*$_{CL}$ *F b*) ∧ (∀ *a*. *F* (*a**) = *F a**)›
then have ‹*clinear F*›
by *simp*
then have ‹*bounded-clinear F*›

by *simp*
then have ‹*continuous-map euclidean euclidean F*›
by (*auto intro*!: *continuous-at-imp-continuous-on clinear-continuous-at*)
then have *wstar*: ‹*continuous-map weak-star-topology weak-star-topology F*›
by *simp*
from *asm* ‹*bounded-clinear F*› *wstar*
show ‹*register F*›
unfolding *register-def* **by** *simp*
qed

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

end

13 Very simple Quantum Hoare logic

theory *QHoare*
imports *Quantum-Extra*
begin

unbundle *register-syntax*
unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
no-notation *Order.top* (⊤₁)

locale *qhoare* =
fixes *memory-type* :: *'mem itself*
begin

definition *apply U R* = *R U* **for** *R* :: ‹*'a update* ⇒ *'mem update*›
definition *ifthen R x* = *R* (*butterfly* (*ket x*) (*ket x*)) **for** *R* :: ‹*'a update* ⇒ *'mem update*›
definition *program S* = *fold* (*o*$_{CL}$) *S id-cblinfun* **for** *S* :: ‹*'mem update list*›

definition *hoare* :: ‹*'mem ell2 ccsubspace* ⇒ (*'mem ell2* ⇒$_{CL}$ *'mem ell2*) *list* ⇒ *'mem ell2 ccsubspace* ⇒ *bool*›
where
hoare C p D ⟷ (∀ *ψ*∈*space-as-set C*. *program p* *$_V$ *ψ* ∈ *space-as-set D*) **for** *C p D*

definition *EQ* :: (*'a update* ⇒ *'mem update*) ⇒ *'a ell2* ⇒ *'mem ell2 ccsubspace* (**infix** =$_q$ *75*) **where**
EQ R ψ = *R* (*selfbutter ψ*) *$_S$ ⊤

lemma *program-skip*[*simp*]: *program* [] = *id-cblinfun*
by (*simp add*: *qhoare.program-def*)

lemma *program-seq*: *program* (*p1*@*p2*) = *program p2 o*$_{CL}$ *program p1*
apply (*induction p2 rule*:*rev-induct*)
apply (*simp-all add*: *program-def*)
by (*meson cblinfun-assoc-left*(*1*))

lemma *hoare-seq*[*trans*]: *hoare C p1 D* ⟹ *hoare D p2 E* ⟹ *hoare C* (*p1*@*p2*) *E*
by (*auto simp*: *program-seq hoare-def*)

lemma *hoare-weaken-left*[*trans*]: ‹*A* ≤ *B* ⟹ *hoare B p C* ⟹ *hoare A p C*›
unfolding *hoare-def*
by (*meson in-mono less-eq-ccsubspace.rep-eq*)

lemma *hoare-weaken-right*[*trans*]: ‹*hoare A p B* ⟹ *B* ≤ *C* ⟹ *hoare A p C*›
unfolding *hoare-def*
by (*meson in-mono less-eq-ccsubspace.rep-eq*)

lemma *hoare-skip*: *C* ≤ *D* ⟹ *hoare C* [] *D*

by (*auto simp*: *program-def hoare-def in-mono less-eq-ccsubspace.rep-eq*)

lemma *hoare-apply*:
assumes $R\ U *_S pre \leq post$
shows *hoare pre* [*apply U R*] *post*
proof −
from *assms* **have** ‹$\psi \in$ *space-as-set pre* $\Longrightarrow R\ U *_V \psi \in$ *space-as-set post*› **for** ψ
by (*metis* (*no-types*, *lifting*) *cblinfun-image.rep-eq closure-subset imageI less-eq-ccsubspace.rep-eq subsetD*)
then show *?thesis*
by (*auto simp*: *hoare-def program-def apply-def*)
qed

lemma *hoare-ifthen*:
fixes $R :: \langle 'a\ update \Rightarrow 'mem\ update\rangle$
assumes R (*selfbutter* (*ket x*)) $*_S$ *pre* $\leq$ *post*
shows *hoare pre* [*ifthen R x*] *post*
proof −
from *assms* **have** ‹$\psi \in$ *space-as-set pre* $\Longrightarrow R$ (*vector-to-cblinfun* (*ket x*) o_{CL} *bra x*) $*_V \psi \in$ *space-as-set post*›
for ψ
by (*metis butterfly-def-one-dim cblinfun-apply-in-image′ less-eq-ccsubspace.rep-eq subsetD*)
then show *?thesis*
by (*auto simp*: *hoare-def program-def ifthen-def butterfly-def*)
qed
end

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

end

14 Quantum teleportation

theory *Teleport*
imports
Real-Impl.Real-Impl
HOL−Library.Code-Target-Numeral
HOL−Library.Word
Hilbert-Space-Tensor-Product.Tensor-Product-Code
QHoare
begin

hide-const (**open**) *Finite-Cartesian-Product.vec*
hide-type (**open**) *Finite-Cartesian-Product.vec*
hide-const (**open**) *Finite-Cartesian-Product.mat*
hide-const (**open**) *Finite-Cartesian-Product.row*
hide-const (**open**) *Finite-Cartesian-Product.column*
no-notation *Group.mult* (**infixl** $\otimes_1$ *70*)
no-notation *Order.top* ($\top_1$)
unbundle *no vec-syntax*
unbundle *no inner-syntax*
unbundle *register-syntax*
unbundle *cblinfun-syntax*

locale *teleport-locale* = *qhoare TYPE*('*mem*) +
fixes $X :: bit\ update \Rightarrow 'mem\ update$
and $\Phi :: (bit*bit)\ update \Rightarrow 'mem\ update$
and $A :: 'atype\ update \Rightarrow 'mem\ update$
and $B :: 'btype\ update \Rightarrow 'mem\ update$
assumes *compat*[*register*]: *mutually compatible* (X,Φ,A,B)
begin

abbreviation $\Phi 1 \equiv \Phi \circ Fst$

abbreviation $\Phi 2 \equiv \Phi \circ Snd$
abbreviation $X\Phi 2 \equiv (X;\Phi 2)$
abbreviation $X\Phi 1 \equiv (X;\Phi 1)$
abbreviation $X\Phi \equiv (X;\Phi)$
abbreviation $XAB \equiv ((X;A);\ B)$
abbreviation $AB \equiv (A;B)$
abbreviation $\Phi 2AB \equiv ((\Phi\ o\ Snd;\ A);\ B)$

definition *teleport a b* = [
apply CNOT $X\Phi 1$,
apply hadamard X,
ifthen $\Phi 1$ *a*,
ifthen X b,
apply (*if a=1 then pauliX else id-cblinfun*) $\Phi 2$,
apply (*if b=1 then pauliZ else id-cblinfun*) $\Phi 2$
]

lemma Φ-$X\Phi$: ‹$\Phi\ a = X\Phi\ (id\text{-}cblinfun \otimes_o a)$›
by (*auto simp*: *register-pair-apply*)
lemma $X\Phi 1$-$X\Phi$: ‹$X\Phi 1\ a = X\Phi\ (assoc\ (a \otimes_o id\text{-}cblinfun))$›
apply (*subst pair-o-assoc*[*unfolded o-def*, *of X* $\Phi 1$ $\Phi 2$, *simplified*, *THEN fun-cong*])
by (*auto simp*: *register-pair-apply*)
lemma $X\Phi 2$-$X\Phi$: ‹$X\Phi 2\ a = X\Phi\ ((id \otimes_r swap)\ (assoc\ (a \otimes_o id\text{-}cblinfun)))$›
apply (*subst pair-o-tensor*[*unfolded o-def*, *THEN fun-cong*], *simp*, *simp*, *simp*)
apply (*subst* (*2*) *register-Fst-register-Snd*[*symmetric*, *of* Φ], *simp*)
using [[*simproc del*: *compatibility-warn*]]
apply (*subst pair-o-swap*[*unfolded o-def*], *simp*)
apply (*subst pair-o-assoc*[*unfolded o-def*, *THEN fun-cong*], *simp*, *simp*, *simp*)
by (*auto simp*: *register-pair-apply*)
lemma $\Phi 2$-$X\Phi$: ‹$\Phi 2\ a = X\Phi\ (id\text{-}cblinfun \otimes_o (id\text{-}cblinfun \otimes_o a))$›
by (*auto simp*: *Snd-def register-pair-apply*)
lemma *X*-$X\Phi$: ‹$X\ a = X\Phi\ (a \otimes_o id\text{-}cblinfun)$›
by (*auto simp*: *register-pair-apply*)
lemma $\Phi 1$-$X\Phi$: ‹$\Phi 1\ a = X\Phi\ (id\text{-}cblinfun \otimes_o (a \otimes_o id\text{-}cblinfun))$›
by (*auto simp*: *Fst-def register-pair-apply*)
lemmas *to-*$X\Phi$ = Φ-$X\Phi$ $X\Phi 1$-$X\Phi$ $X\Phi 2$-$X\Phi$ $\Phi 2$-$X\Phi$ *X*-$X\Phi$ $\Phi 1$-$X\Phi$

lemma *X*-$X\Phi 1$: ‹$X\ a = X\Phi 1\ (a \otimes_o id\text{-}cblinfun)$›
by (*auto simp*: *register-pair-apply*)
lemmas *to-*$X\Phi 1$ = *X*-$X\Phi 1$

lemma *XAB-to-*$X\Phi 2$*-AB*: ‹$XAB\ a = (X\Phi 2;AB)\ ((swap \otimes_r id)\ (assoc'\ (id\text{-}cblinfun \otimes_o assoc\ a)))$›
by (*simp add*: *pair-o-tensor*[*unfolded o-def*, *THEN fun-cong*] *register-pair-apply*
pair-o-swap[*unfolded o-def*, *THEN fun-cong*]
pair-o-assoc'[*unfolded o-def*, *THEN fun-cong*]
pair-o-assoc[*unfolded o-def*, *THEN fun-cong*])

lemma $X\Phi 2$*-to-*$X\Phi 2$*-AB*: ‹$X\Phi 2\ a = (X\Phi 2;AB)\ (a \otimes_o id\text{-}cblinfun)$›
by (*simp add*: *register-pair-apply*)

schematic-goal $\Phi 2AB$*-to-*$X\Phi 2$*-AB*: $\Phi 2AB\ a = (X\Phi 2;AB)\ ?b$
apply (*subst pair-o-assoc'*[*unfolded o-def*, *THEN fun-cong*])
apply *simp-all*[*3*]
apply (*subst register-pair-apply*[**where** *a=id-cblinfun*])
apply *simp-all*[*2*]
apply (*subst pair-o-assoc*[*unfolded o-def*, *THEN fun-cong*])
apply *simp-all*[*3*]
by *simp*

lemmas *to-*$X\Phi 2$*-AB* = *XAB-to-*$X\Phi 2$*-AB* $X\Phi 2$*-to-*$X\Phi 2$*-AB* $\Phi 2AB$*-to-*$X\Phi 2$*-AB*

lemma *teleport*:

```
  assumes [simp]: norm ψ = 1
  shows hoare (XAB =q ψ ⊓ Φ =q β00) (teleport a b) (Φ2AB =q ψ)
proof −
  define XZ :: ‹bit update› where XZ = (if a=1 then (if b=1 then pauliZ oCL pauliX else pauliX) else (if b=1
then pauliZ else id-cblinfun))

  define pre where pre = XAB =q ψ

  define O1 where O1 = Φ (selfbutter β00)
  have ‹(XAB =q ψ ⊓ Φ =q β00) = O1 ∗S pre›
    unfolding pre-def O1-def EQ-def
    apply (subst compatible-proj-intersect[where R=XAB and S=Φ])
      apply (simp-all add: butterfly-is-Proj)
    apply (subst swap-registers[where R=XAB and S=Φ])
    by (simp-all add: cblinfun-assoc-left(2))

  also
  define O2 where O2 = XΦ1 CNOT oCL O1
  have ‹hoare (O1 ∗S pre) [apply CNOT XΦ1] (O2 ∗S pre)›
    apply (rule hoare-apply) by (simp add: O2-def cblinfun-assoc-left(2))

  also
  define O3 where ‹O3 = X hadamard oCL O2›
  have ‹hoare (O2 ∗S pre) [apply hadamard X] (O3 ∗S pre)›
    apply (rule hoare-apply) by (simp add: O3-def cblinfun-assoc-left(2))

  also
  define O4 where ‹O4 = Φ1 (selfbutter (ket a)) oCL O3›
  have ‹hoare (O3 ∗S pre) [ifthen Φ1 a] (O4 ∗S pre)›
    apply (rule hoare-ifthen) by (simp add: O4-def cblinfun-assoc-left(2))

  also
  define O5 where ‹O5 = X (selfbutter (ket b)) oCL O4›
  have ‹hoare (O4 ∗S pre) [ifthen X b] (O5 ∗S pre)›
    apply (rule hoare-ifthen) by (simp add: O5-def cblinfun-assoc-left(2))

  also
  define O6 where ‹O6 = Φ2 (if a=1 then pauliX else id-cblinfun) oCL O5›
  have ‹hoare (O5 ∗S pre) [apply (if a=1 then pauliX else id-cblinfun) (Φ ∘ Snd)] (O6 ∗S pre)›
    apply (rule hoare-apply) by (auto simp add: O6-def cblinfun-assoc-left(2))

  also
  define O7 where ‹O7 = Φ2 (if b = 1 then pauliZ else id-cblinfun) oCL O6›
  have O7: ‹O7 = Φ2 XZ oCL O5›
    by (auto simp add: O6-def O7-def XZ-def register-mult lift-cblinfun-comp[OF register-mult])
  have ‹hoare (O6 ∗S pre) [apply (if b=1 then pauliZ else id-cblinfun) (Φ ∘ Snd)] (O7 ∗S pre)›
    apply (rule hoare-apply)
    by (auto simp add: O7-def cblinfun-assoc-left(2))

  finally have hoare: ‹hoare (XAB =q ψ ⊓ Φ =q β00) (teleport a b) (O7 ∗S pre)›
    by (auto simp add: teleport-def comp-def)

  have O5': O5 = (1/2) ∗C Φ2 (XZ∗) oCL XΦ2 Uswap oCL Φ (butterfly (ket a ⊗s ket b) β00)
    unfolding O7 O5-def O4-def O3-def O2-def O1-def
    apply (simp split del: if-split only: to-XΦ register-mult[of XΦ])
    apply (simp split del: if-split add: register-mult[of XΦ] clinear-register
                flip: complex-vector.linear-scale
                del: comp-apply)
    apply (rule arg-cong[of - - XΦ])
    apply (rule cblinfun-eq-mat-of-cblinfunI)
    apply (simp add: assoc-ell2-sandwich mat-of-cblinfun-tensor-op XZ-def
                     butterfly-def mat-of-cblinfun-compose mat-of-cblinfun-vector-to-cblinfun
                     mat-of-cblinfun-adj vec-of-basis-enum-ket mat-of-cblinfun-id
```

swap-sandwich[*abs-def*] *mat-of-cblinfun-scaleR mat-of-cblinfun-scaleC*
id-tensor-sandwich vec-of-basis-enum-tensor-state mat-of-cblinfun-cblinfun-apply
mat-of-cblinfun-sandwich)
by *normalization*

have [*simp*]: *unitary XZ*
unfolding *unitary-def* **unfolding** *XZ-def*
by (*auto simp*: *cblinfun-assoc-left lift-cblinfun-comp*[*OF pauliZZ*] *lift-cblinfun-comp*[*OF pauliXX*])

have *O7'*: *O7* = (*1/2*) $*_C$ *XΦ2 Uswap* o_{CL} Φ (*butterfly* (*ket a* $\otimes_s$ *ket b*) *β00*)
unfolding *O7 O5'*
by (*simp add*: *cblinfun-compose-assoc*[*symmetric*] *register-mult*[*of Φ2*] *del*: *comp-apply*)

have *O7* $*_S$ *pre* = *XΦ2 Uswap* $*_S$ *XAB* (*selfbutter ψ*) $*_S$ Φ (*butterfly* (*ket* (*a*, *b*)) *β00*) $*_S$ ⊤
apply (*simp add*: *O7' pre-def EQ-def cblinfun-compose-image tensor-ell2-ket*)
apply (*subst lift-cblinfun-comp*[*OF swap-registers*[**where** *R*=Φ **and** *S*=*XAB*]], *simp*)
by (*simp add*: *cblinfun-assoc-left*(*2*))
also have ‹... ≤ *XΦ2 Uswap* $*_S$ *XAB* (*selfbutter ψ*) $*_S$ ⊤›
by (*simp add*: *cblinfun-image-mono*)
also have ‹... = (*XΦ2*;*AB*) (*Uswap* $\otimes_o$ *id-cblinfun*) $*_S$ (*XΦ2*;*AB*)
((*swap* $\otimes_r$ *id*) (*assoc'* (*id-cblinfun* $\otimes_o$ *assoc* (*selfbutter ψ*)))) $*_S$ ⊤›
by (*simp add*: *to-XΦ2-AB*)
also have ‹... = *Φ2AB* (*selfbutter ψ*) $*_S$ *XΦ2 Uswap* $*_S$ ⊤›
apply (*simp add*: *swap-sandwich sandwich-grow-left to-XΦ2-AB*
cblinfun-compose-image[*symmetric*] *register-mult*)
by (*simp add*: *sandwich-apply cblinfun-compose-assoc*[*symmetric*] *comp-tensor-op tensor-op-adjoint*)
also have ‹... ≤ *Φ2AB* $=_q$ *ψ*›
by (*simp add*: *EQ-def cblinfun-image-mono*)
finally have ‹*O7* $*_S$ *pre* ≤ *Φ2AB* $=_q$ *ψ*›
by *simp*

with *hoare*
show *?thesis*
by (*meson basic-trans-rules*(*31*) *hoare-def less-eq-ccsubspace.rep-eq*)
qed

end

locale *concrete-teleport-vars* **begin**

type-synonym *a-state* = *64 word*
type-synonym *b-state* = *1000000 word*
type-synonym *mem* = *a-state* * *bit* * *bit* * *b-state* * *bit*
type-synonym *'a var* = ‹*'a update* ⇒ *mem update*›

definition *A* :: *a-state var* **where** ‹*A a* = *a* $\otimes_o$ *id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *id-cblinfun*›
definition *X* :: ‹*bit var*› **where** ‹*X a* = *id-cblinfun* $\otimes_o$ *a* $\otimes_o$ *id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *id-cblinfun*›
definition *Φ1* :: ‹*bit var*› **where** ‹*Φ1 a* = *id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *a* $\otimes_o$ *id-cblinfun* $\otimes_o$ *id-cblinfun*›
definition *B* :: ‹*b-state var*› **where** ‹*B a* = *id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *a* $\otimes_o$ *id-cblinfun*›
definition *Φ2* :: ‹*bit var*› **where** ‹*Φ2 a* = *id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *id-cblinfun* $\otimes_o$ *a*›

end

interpretation *teleport-concrete*:
concrete-teleport-vars +
teleport-locale concrete-teleport-vars.X
‹(*concrete-teleport-vars.Φ1*; *concrete-teleport-vars.Φ2*)›
concrete-teleport-vars.A
concrete-teleport-vars.B
apply *standard*
using [[*simproc del*: *compatibility-warn*]]

by (*auto simp*: *concrete-teleport-vars.X-def*[*abs-def*]
concrete-teleport-vars.Φ1-def[*abs-def*]
concrete-teleport-vars.Φ2-def[*abs-def*]
concrete-teleport-vars.A-def[*abs-def*]
concrete-teleport-vars.B-def[*abs-def*]
intro!: *compatible3′ compatible3*)

thm *teleport*
thm *teleport-def*

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

end

15 Quantum instantiation of complements

theory *Axioms-Complement-Quantum*
imports
Laws-Quantum
Quantum-Extra
Hilbert-Space-Tensor-Product.Weak-Star-Topology
Hilbert-Space-Tensor-Product.Partial-Trace
With-Type.With-Type
Hilbert-Space-Tensor-Product.Misc-Tensor-Product-TTS
begin

unbundle *no m-inv-syntax*
unbundle *cblinfun-syntax*
unbundle *lattice-syntax*
unbundle *register-syntax*
no-notation *Lattice.join* (**infixl** ⊔ı *65*)
no-notation *elt-set-eq* (**infix** =*o* *50*)
no-notation *eq-closure-of* (*closure′-of*ı)
hide-const (**open**) *Order.top*
declare [[*eta-contract*]]

lemma *register-decomposition-converse*:
assumes ‹*unitary U*›
shows ‹*register* (λ*x*. *sandwich U* (*id-cblinfun* ⊗*o x*))›
using - *unitary-sandwich-register* **apply** (*rule register-comp*[*unfolded o-def*])
using *assms* **by** *auto*

lemma *iso-register-decomposition*:
assumes [*simp*]: ‹*iso-register F*›
shows ‹∃ *U*. *unitary U* ∧ *F* = *sandwich U*›
proof −
from *register-decomposition*
have ‹*let* 'c::*type* = *register-decomposition-basis F in*
∃ *U*. *unitary U* ∧ *F* = *sandwich U*›
proof *with-type-mp*
show [*simp*]: ‹*register F*›
using *assms iso-register-is-register* **by** *blast*
with-type-case

let *?ida* = ‹*id-cblinfun* :: 'c *ell2* ⇒*CL* -›

from *with-type-mp.premise*
obtain *V* :: ‹('a × 'c) *ell2* ⇒*CL* 'b *ell2*› **where** ‹*unitary V*›
and *FV*: ‹*F* ϑ = *sandwich V* (ϑ ⊗*o ?ida*)› **for** ϑ
by *auto*

have *inj-V*: ‹*inj* ((∗*V*) (*sandwich V*))›
by (*meson* ‹*unitary V*› *register-inj unitary-sandwich-register*)
have ‹*surj F*›
by (*meson assms iso-register-inv-comp2 surj-iff*)
have *surj-tensor*: ‹*surj* (λ*a*::'a *ell2* ⇒*CL* 'a *ell2*. *a* ⊗*o ?ida*)›
apply (*rule surj-from-comp*[**where** *g*=‹*sandwich V*›])
using ‹*surj F*› *inj-V* **by** (*auto simp*: *FV*)
then obtain *a* :: ‹'a *ell2* ⇒*CL* 'a *ell2*›
where *a*: ‹*a* ⊗*o ?ida* = *butterfly* (*ket undefined*) (*ket undefined*) ⊗*o butterfly* (*ket undefined*) (*ket undefined*)›
by (*smt* (*verit*, *best*) *surjD*)

have ‹*selfbutter* (*ket* (*undefined*, *undefined*)) ≠ *0*›
by (*metis butterfly-apply cblinfun.zero-left complex-vector.scale-eq-0-iff ket-nonzero orthogonal-ket*)
with *a* **have** ‹*a* ≠ *0*›
by (*auto simp*: *tensor-ell2-ket tensor-butterfly*)

obtain γ **where** γ: ‹*?ida* = γ ∗*C butterfly* (*ket undefined*) (*ket undefined*)›
apply *atomize-elim*
using *a* ‹*a* ≠ *0*› **by** (*rule tensor-op-almost-injective*)
then have ‹*?ida* (*ket undefined*) = γ ∗*C* (*butterfly* (*ket undefined*) (*ket undefined*) ∗*V ket undefined*)›
by (*simp add*: ‹*id-cblinfun* = γ ∗*C butterfly* (*ket undefined*) (*ket undefined*)› *scaleC-cblinfun.rep-eq*)
then have ‹*ket undefined* = γ ∗*C ket undefined*›
by (*metis butterfly-apply cinner-ket-same id-cblinfun-apply ket-nonzero scaleC-cancel-right scaleC-one*)
then have ‹γ = *1*›
by (*smt* (*z3*) γ *butterfly-apply butterfly-scaleC-left cblinfun-id-cblinfun-apply complex-vector.scale-cancel-right cinner-ket-same ket-nonzero*)

define T U **where** ‹T = *CBlinfun* ($\lambda\psi$. ψ $\otimes_s$ *ket undefined*)› **and** ‹U = V o_{CL} T›
have T: ‹T ψ = ψ $\otimes_s$ *ket undefined*› **for** ψ
unfolding *T-def*
apply (*subst bounded-clinear-CBlinfun-apply*)
by (*auto intro*!: *bounded-clinear-tensor-ell22*)
have ‹*sandwich* T (*butterfly* (*ket i*) (*ket j*)) = *butterfly* (*ket i*) (*ket j*) $\otimes_o$ *id-cblinfun*› **for** *i j*
by (*simp add*: T *sandwich-apply cblinfun-comp-butterfly butterfly-comp-cblinfun* γ ‹γ = *1*› *tensor-butterfly*)
then have *sandwich-T*: ‹*sandwich* T *a* = *a* $\otimes_o$ *?ida*› **for** *a*
apply (*rule-tac fun-cong*[**where** *x*=*a*])
apply (*rule weak-star-clinear-eq-butterfly-ketI*[**where** *T*=*weak-star-topology*])
by *auto*

have ‹F (*butterfly x y*) = V o_{CL} (*butterfly x y* $\otimes_o$ *?ida*) o_{CL} $V*$› **for** *x y*
by (*simp add*: *sandwich-apply FV*)
also have ‹... *x y* = V o_{CL} (*butterfly* (T *x*) (T *y*)) o_{CL} $V*$› **for** *x y*
by (*simp add*: T γ ‹γ = *1*› *tensor-butterfly*)
also have ‹... *x y* = U o_{CL} (*butterfly x y*) o_{CL} $U*$› **for** *x y*
by (*simp add*: *U-def butterfly-comp-cblinfun cblinfun-comp-butterfly*)
finally have *F-rep*: ‹F *a* = U o_{CL} *a* o_{CL} $U*$› **for** *a*
apply (*rule-tac fun-cong*[**where** *x*=*a*])
apply (*rule weak-star-clinear-eq-butterfly-ketI*[**where** *T*=*weak-star-topology*])
by (*auto simp*: *clinear-register weak-star-cont-register simp flip*: *sandwich-apply*)

have ‹*isometry* T›
apply (*rule orthogonal-on-basis-is-isometry*[**where** *B*=‹*range ket*›])
by (*auto simp*: T)
moreover have ‹T $*_S$ $\top$ = $\top$›
proof −
have *1*: ‹φ $\otimes_s$ ξ ∈ *range* (($*_V$) T)› **for** φ ξ
proof −
have ‹T $*_V$ (*cinner* (*ket undefined*) ξ $*_C$ φ) = φ $\otimes_s$ (*cinner* (*ket undefined*) ξ $*_C$ *ket undefined*)›
by (*simp add*: T *tensor-ell2-scaleC1 tensor-ell2-scaleC2*)
also have ‹... = φ $\otimes_s$ (*butterfly* (*ket undefined*) (*ket undefined*) $*_V$ ξ)›
by *simp*
also have ‹... = φ $\otimes_s$ (*?ida* $*_V$ ξ)›
by (*simp add*: γ ‹γ = *1*›)
also have ‹... = φ $\otimes_s$ ξ›
by *simp*
finally show *?thesis*
by (*metis range-eqI*)
qed

have ‹$\top$ ≤ *ccspan* {*ket x* | *x*. *True*}›
by (*simp add*: *full-SetCompr-eq*)
also have ‹... ≤ *ccspan* {φ $\otimes_s$ ξ | φ ξ. *True*}›
apply (*rule ccspan-mono*)
by (*auto simp flip*: *tensor-ell2-ket*)
also from *1* **have** ‹... ≤ *ccspan* (*range* (($*_V$) T))›
by (*auto intro*!: *ccspan-mono*)
also have ‹... = T $*_S$ $\top$›
by (*metis* (*mono-tags*, *opaque-lifting*) *calculation cblinfun-image-ccspan cblinfun-image-mono eq-iff top-greatest*)
finally show ‹T $*_S$ $\top$ = $\top$›
using *top.extremum-uniqueI* **by** *blast*
qed

ultimately have ‹*unitary* T›
by (*rule surj-isometry-is-unitary*)
then have ‹*unitary* U›
by (*simp add*: *U-def* ‹*unitary* V›)

from *F-rep* ‹*unitary* U› **show** ‹∃ U. *unitary* U ∧ F = *sandwich* U›

```
    by (auto simp: sandwich-apply[abs-def])
  qed
  from this[THEN with-type-prepare-cancel, cancel-type-definition, OF with-type-nonempty, OF this]
  show ?thesis
    by −
qed

lemma complement-exists:
  fixes F :: ‹'a update ⇒ 'b update›
  assumes ‹register F›
  shows ‹let 'c::type = register-decomposition-basis F in
        ∃ G :: 'c update ⇒ 'b update. compatible F G ∧ iso-register (F;G)›
proof (use register-decomposition[OF ‹register F›] in ‹rule with-type-mp›)
  note [[simproc del: Laws-Quantum.compatibility-warn]]
 assume register-decomposition: ‹∃ U :: ('a × 'c) ell2 ⇒CL 'b ell2. unitary U ∧ (∀ϑ. F ϑ = Complex-Bounded-Linear-Function.san
U (ϑ ⊗o id-cblinfun))›
  then obtain U :: ‹('a × 'c) ell2 ⇒CL 'b ell2›
    where [simp]: unitary U and F: ‹F a = sandwich U (a ⊗o id-cblinfun)› for a
    by auto
  define G :: ‹'c update ⇒ 'b update› where ‹G b = sandwich U (id-cblinfun ⊗o b)› for b
  have [simp]: ‹register G›
    unfolding G-def apply (rule register-decomposition-converse) by simp
  have ‹F a oCL G b = G b oCL F a› for a b
  proof −
    have ‹F a oCL G b = sandwich U (a ⊗o b)›
      by (auto simp: F G-def sandwich-apply cblinfun-assoc-right
          ‹unitary U› lift-cblinfun-comp[OF unitaryD1]
          lift-cblinfun-comp[OF comp-tensor-op])
    moreover have ‹G b oCL F a = sandwich U (a ⊗o b)›
      by (auto simp: F G-def sandwich-apply cblinfun-assoc-right
          ‹unitary U› lift-cblinfun-comp[OF unitaryD1]
          lift-cblinfun-comp[OF comp-tensor-op])
    ultimately show ?thesis by simp
  qed
  then have [simp]: ‹compatible F G›
    by (auto simp: compatible-def ‹register F›)
  moreover have ‹iso-register (F;G)›
  proof −
    have ‹(F;G) (a ⊗o b) = sandwich U (a ⊗o b)› for a b
      by (auto simp: register-pair-apply F G-def sandwich-apply cblinfun-assoc-right
          ‹unitary U› lift-cblinfun-comp[OF unitaryD1]
          lift-cblinfun-comp[OF comp-tensor-op])
    then have FG: ‹(F;G) = sandwich U›
      apply (rule tensor-extensionality[rotated −1])
      by (simp-all add: unitary-sandwich-register)
    define I where ‹I = sandwich (U*)› for x
    have [simp]: ‹register I›
      by (simp add: I-def unitary-sandwich-register)
    have ‹I o (F;G) = id› and FGI: ‹(F;G) o I = id›
      by (auto intro!:ext simp: I-def[abs-def] FG sandwich-apply cblinfun-assoc-right
          ‹unitary U› lift-cblinfun-comp[OF unitaryD1] lift-cblinfun-comp[OF unitaryD2]
          lift-cblinfun-comp[OF comp-tensor-op])
    then show ‹iso-register (F;G)›
      by (auto intro!: iso-registerI)
  qed
  ultimately show ‹∃ G :: 'c update ⇒ 'b update. compatible F G ∧ iso-register (F;G)›
    apply (rule-tac exI[of - G]) by auto
qed

lemma commutant-exchange:
  fixes F :: ‹'a update ⇒ 'b update›
  assumes ‹iso-register F›
  shows ‹commutant (F ‘ X) = F ‘ commutant X›
```

proof (*rule Set.set-eqI*)
fix $x :: \langle 'b\ update \rangle$
from *assms*
obtain G **where** $\langle F\ o\ G = id \rangle$ **and** $\langle G\ o\ F = id \rangle$ **and** [*simp*]: $\langle register\ G \rangle$
using *iso-register-def* **by** *blast*
from *assms* **have** [*simp*]: $\langle register\ F \rangle$
using *iso-register-def* **by** *blast*
have $\langle x \in commutant\ (F\ `\ X) \longleftrightarrow (\forall y \in F\ `\ X.\ x\ o_{CL}\ y = y\ o_{CL}\ x) \rangle$
by (*simp add*: *commutant-def*)
also have $\langle \ldots \longleftrightarrow (\forall y \in F\ `\ X.\ G\ x\ o_{CL}\ G\ y = G\ y\ o_{CL}\ G\ x) \rangle$
by (*metis* (*no-types*, *opaque-lifting*) $\langle F \circ G = id \rangle$ $\langle G\ o\ F = id \rangle$ $\langle register\ G \rangle$ *comp-def eq-id-iff register-def*)
also have $\langle \ldots \longleftrightarrow (\forall y \in X.\ G\ x\ o_{CL}\ y = y\ o_{CL}\ G\ x) \rangle$
by (*simp add*: $\langle G \circ F = id \rangle$ *pointfree-idE*)
also have $\langle \ldots \longleftrightarrow G\ x \in commutant\ X \rangle$
by (*simp add*: *commutant-def*)
also have $\langle \ldots \longleftrightarrow x \in F\ `\ commutant\ X \rangle$
by (*metis* (*no-types*, *opaque-lifting*) $\langle G \circ F = id \rangle$ $\langle F \circ G = id \rangle$ *image-iff pointfree-idE*)
finally show $\langle x \in commutant\ (F\ `\ X) \longleftrightarrow x \in F\ `\ commutant\ X \rangle$
by −
qed

lemma *complement-range*:
assumes [*simp*]: $\langle compatible\ F\ G \rangle$ **and** [*simp*]: $\langle iso\text{-}register\ (F;G) \rangle$
shows $\langle range\ G = commutant\ (range\ F) \rangle$
proof −
have [*simp*]: $\langle register\ F \rangle$ $\langle register\ G \rangle$
using *assms compatible-def* **by** *metis*+
have [*simp*]: $\langle (F;G)\ (a \otimes_o b) = F\ a\ o_{CL}\ G\ b \rangle$ **for** *a b*
using *Laws-Quantum.register-pair-apply assms* **by** *blast*
have [*simp*]: $\langle range\ F = (F;G)\ `\ range\ (\lambda a.\ a \otimes_o id\text{-}cblinfun) \rangle$
by *force*
have [*simp*]: $\langle range\ G = (F;G)\ `\ range\ (\lambda b.\ id\text{-}cblinfun \otimes_o b) \rangle$
by *force*
show $\langle range\ G = commutant\ (range\ F) \rangle$
by (*simp add*: *commutant-exchange commutant-tensor1*)
qed

lemma *register-inv-G-o-F*:
assumes [*simp*]: $\langle register\ F \rangle$ **and** [*simp*]: $\langle register\ G \rangle$ **and** *range-FG*: $\langle range\ F \subseteq range\ G \rangle$
shows $\langle register\ (inv\ G \circ F) \rangle$
proof −
note [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
define *GF* **where** $\langle GF = inv\ G\ o\ F \rangle$
have *F-rangeG*[*simp*]: $\langle F\ x \in range\ G \rangle$ **for** *x*
using *range-FG* **by** *auto*
have [*simp*]: $\langle inj\ F \rangle$ **and** [*simp*]: $\langle inj\ G \rangle$
by (*simp-all add*: *register-inj*)
have [*simp*]: $\langle bounded\text{-}clinear\ F \rangle$ $\langle bounded\text{-}clinear\ G \rangle$
by (*simp-all add*: *register-bounded-clinear*)
have [*simp*]: $\langle clinear\ F \rangle$ $\langle clinear\ G \rangle$
by (*simp-all add*: *bounded-clinear.clinear*)
have *addJ*: $\langle GF\ (x + y) = GF\ x + GF\ y \rangle$ **for** *x y*
unfolding *GF-def o-def*
apply (*rule injD*[*OF* $\langle inj\ G \rangle$])
apply (*subst complex-vector.linear-add*[*OF* $\langle clinear\ G \rangle$])
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*G*], *simp*)+
by (*simp add*: *complex-vector.linear-add*)
have *scaleJ*: $\langle GF\ (r *_C x) = r *_C GF\ x \rangle$ **for** *r x*
unfolding *GF-def o-def*
apply (*rule injD*[*OF* $\langle inj\ G \rangle$])

```
    apply (subst complex-vector.linear-scale[OF ‹clinear G›])
    apply (subst Hilbert-Choice.f-inv-into-f[where f=G], simp)+
    by (simp add: complex-vector.linear-scale)
  have unitalJ: ‹GF id-cblinfun = id-cblinfun›
    unfolding GF-def o-def
    apply (rule injD[OF ‹inj G›])
    apply (subst Hilbert-Choice.f-inv-into-f[where f=G])
    by (auto intro!: range-eqI[of - - id-cblinfun])
  have multJ: ‹GF (a oCL b) = GF a oCL GF b› for a b
    unfolding GF-def o-def
    apply (rule injD[OF ‹inj G›])
    apply (subst register-mult[symmetric, OF ‹register G›])
    apply (subst Hilbert-Choice.f-inv-into-f[where f=G], simp)+
    by (simp add: register-mult)
  have adjJ: ‹GF (a*) = (GF a)*› for a
    unfolding GF-def o-def
    apply (rule injD[OF ‹inj G›])
    apply (subst register-adjoint[OF ‹register G›])
    apply (subst Hilbert-Choice.f-inv-into-f[where f=G], simp)+
    using ‹register F› register-adjoint by blast
  have normJ: ‹norm (GF a) = norm a› for a
    unfolding GF-def
    by (metis F-rangeG ‹register F› ‹register G› f-inv-into-f o-def register-norm)
  have weak-star-J: ‹continuous-map weak-star-topology weak-star-topology GF›
  proof −
    have ‹continuous-map weak-star-topology weak-star-topology F›
      by (simp add: weak-star-cont-register)
    then have ‹continuous-map weak-star-topology (subtopology weak-star-topology (range F)) F›
      by (simp add: continuous-map-into-subtopology)
    moreover have ‹continuous-map (subtopology weak-star-topology (range G)) weak-star-topology (inv G)›
      using ‹register G› register-inv-weak-star-continuous by blast
    ultimately show ‹continuous-map weak-star-topology weak-star-topology GF›
      by (simp add: GF-def range-FG continuous-map-compose continuous-map-from-subtopology-mono)
  qed

  from addJ scaleJ unitalJ multJ adjJ normJ weak-star-J
  show ‹register GF›
    unfolding register-def by (auto intro!: bounded-clinearI[where K=1])
qed


lemma same-range-equivalent:
  fixes F :: ‹'a update ⇒ 'c update› and G :: ‹'b update ⇒ 'c update›
  assumes [simp]: ‹register F› and [simp]: ‹register G›
  assumes range-FG: ‹range F = range G›
  shows ‹equivalent-registers F G›
proof −
  note [[simproc del: Laws-Quantum.compatibility-warn]]
  have regGF: ‹register (inv G o F)›
    using assms by (auto intro!: register-inv-G-o-F)
  have regFG: ‹register (inv F o G)›
    using assms by (auto intro!: register-inv-G-o-F)
  have ‹inj G›
    by (simp add: register-inj)
  with range-FG have GFFG: ‹(inv G o F) o (inv F o G) = id›
    by (smt (verit) assms(1) f-inv-into-f invI isomorphism-expand o-def rangeI register-inj)
  have ‹inj F›
    by (simp add: register-inj)
  with range-FG have FGGF: ‹(inv F o G) o (inv G o F) = id›
    by (metis GFFG fun.set-map image-inv-f-f left-right-inverse-eq surj-iff)
  from regGF regFG GFFG FGGF have iso-FG: ‹iso-register (inv F o G)›
    using iso-registerI by auto
  have FFG: ‹F o (inv F o G) = G›
```

by (*smt* (*verit*) *FGGF GFFG fun.inj-map-strong fun.map-comp fun.set-map image-inv-f-f inj-on-imageI2 inv-id inv-into-injective inv-o-cancel o-inv-o-cancel range-FG surj-id surj-imp-inj-inv*)
from *FFG iso-FG* **show** ‹*equivalent-registers F G*›
by (*simp add*: *equivalent-registersI*)
qed

lemma *complement-unique*:
assumes *compatible F G* **and** ‹*iso-register* (*F*;*G*)›
assumes *compatible F H* **and** ‹*iso-register* (*F*;*H*)›
shows ‹*equivalent-registers G H*›
by (*metis assms compatible-def complement-range same-range-equivalent*)

15.1 Finite dimensional complement

typedef ('*a*, '*b*::*finite*) *complement-domain-simple* = ‹{..< *if CARD*('*b*) *div CARD*('*a*) ≠ *0 then CARD*('*b*) *div CARD*('*a*) *else 1*}›
by *auto*

instance *complement-domain-simple* :: (*type*, *finite*) *finite*
proof *intro-classes*
have ‹*inj Rep-complement-domain-simple*›
by (*simp add*: *Rep-complement-domain-simple-inject inj-on-def*)
moreover have ‹*finite* (*range Rep-complement-domain-simple*)›
by (*metis finite-lessThan type-definition.Rep-range type-definition-complement-domain-simple*)
ultimately show ‹*finite* (*UNIV* :: ('*a*,'*b*) *complement-domain-simple set*)›
using *finite-image-iff* **by** *blast*
qed

lemma *CARD-complement-domain*:
assumes ‹*CARD*('*b*::*finite*) = *n* ∗ *CARD*('*a*)›
shows ‹*CARD*(('*a*,'*b*) *complement-domain-simple*) = *n*›
proof −
from *assms* **have** ‹*n* > *0*›
by (*metis nat-0-less-mult-iff zero-less-card-finite*)
have ∗: ‹*inj Rep-complement-domain-simple*›
by (*simp add*: *Rep-complement-domain-simple-inject inj-on-def*)
moreover have ‹*card* (*range* (*Rep-complement-domain-simple* :: ('*a*,'*b*) *complement-domain-simple* ⇒ -)) = *n*›
apply (*subst type-definition.Rep-range*[*OF type-definition-complement-domain-simple*])
using *assms* ‹*n* > *0*› **apply** *simp*
by *force*
ultimately show *?thesis*
by (*metis card-image*)
qed

lemma *register-decomposition-finite-aux*:
fixes Φ :: ‹'*a*::*finite update* ⇒ '*b*::*finite update*›
assumes [*simp*]: ‹*register* Φ›
shows ‹∃ *U* :: ('*a* × ('*a*, '*b*) *complement-domain-simple*) *ell2* ⇒$_{CL}$ '*b ell2*. *unitary U* ∧
(∀ ϑ. Φ ϑ = *sandwich U* (ϑ ⊗$_o$ *id-cblinfun*))›
— Proof based on [Daw21]
proof −
note [[*simproc del*: *compatibility-warn*]]
fix ξ*0* :: '*a*

have [*simp*]: ‹*clinear* Φ›
by (*simp add*: *clinear-register*)

define *P* **where** ‹*P i* = *Proj* (*ccspan* {*ket i*})› **for** *i* :: '*a*
have *P-butter*: ‹*P i* = *butterfly* (*ket i*) (*ket i*)› **for** *i*
by (*simp add*: *P-def butterfly-eq-proj*)

define P' **where** ‹$P'\ i = \Phi\ (P\ i)$› **for** $i :: 'a$
have *proj-P'*: ‹*is-Proj* $(P'\ i)$› **for** i
by (*simp add*: *P-def P'-def register-projector*)
have ‹$(\sum i{\in}UNIV.\ P\ i) = id\text{-}cblinfun$›
using *sum-butterfly-ket P-butter* **by** *simp*
then have *sumP'id*: ‹$(\sum i{\in}UNIV.\ P'\ i) = id\text{-}cblinfun$›
unfolding *P'-def*
apply (*subst complex-vector.linear-sum*[*OF* ‹*clinear* Φ›, *symmetric*])
by *auto*

define S **where** ‹$S\ i = P'\ i *_S \top$› **for** $i :: 'a$
have *P'id*: ‹$P'\ i *_V \psi = \psi$› **if** ‹$\psi \in$ *space-as-set* $(S\ i)$› **for** $i\ \psi$
using *S-def that proj-P'*
by (*metis cblinfun-fixes-range is-Proj-algebraic*)

obtain *B0* **where** *finiteB0*: ‹*finite* $(B0\ i)$› **and** *cspanB0*: ‹*cspan* $(B0\ i) =$ *space-as-set* $(S\ i)$› **for** i
apply *atomize-elim* **apply** (*simp flip*: *all-conj-distrib*) **apply** (*rule choice*)
by (*meson cfinite-dim-finite-subspace-basis csubspace-space-as-set*)

obtain B **where** *orthoB*: ‹*is-ortho-set* $(B\ i)$›
and *normalB*: ‹$\bigwedge b.\ b \in B\ i \Longrightarrow norm\ b = 1$›
and *cspanB*: ‹*cspan* $(B\ i) =$ *cspan* $(B0\ i)$›
and *finiteB*: ‹*finite* $(B\ i)$› **for** i
apply *atomize-elim* **apply** (*simp flip*: *all-conj-distrib*) **apply** (*rule choice*)
using *orthonormal-basis-of-cspan*[*OF finiteB0*] **by** *blast*

from *cspanB cspanB0* **have** *cspanB*: ‹*cspan* $(B\ i) =$ *space-as-set* $(S\ i)$› **for** i
by *simp*
then have *ccspanB*: ‹*ccspan* $(B\ i) = S\ i$› **for** i
by (*metis ccspan.rep-eq closure-finite-cspan finiteB space-as-set-inject*)
from *orthoB* **have** *indepB*: ‹*cindependent* $(B\ i)$› **for** i
by (*simp add*: *Complex-Inner-Product.is-ortho-set-cindependent*)

have *orthoBiBj*: ‹*is-orthogonal* $x\ y$› **if** ‹$x \in B\ i$› **and** ‹$y \in B\ j$› **and** ‹$i \neq j$› **for** $x\ y\ i\ j$
proof −
from ‹$x \in B\ i$› **obtain** x' **where** x: ‹$x = P'\ i *_V x'$›
by (*metis S-def cblinfun-fixes-range complex-vector.span-base cspanB is-Proj-idempotent proj-P'*)
from ‹$y \in B\ j$› **obtain** y' **where** y: ‹$y = P'\ j *_V y'$›
by (*metis S-def cblinfun-fixes-range complex-vector.span-base cspanB is-Proj-idempotent proj-P'*)
have ‹*cinner* $x\ y =$ *cinner* $(P'\ i *_V x')\ (P'\ j *_V y')$›
using $x\ y$ **by** *simp*
also have ‹$\ldots =$ *cinner* $(P'\ j *_V P'\ i *_V x')\ y'$›
by (*metis cinner-adj-left is-Proj-algebraic proj-P'*)
also have ‹$\ldots =$ *cinner* $(\Phi\ (P\ j\ o_{CL}\ P\ i) *_V x')\ y'$›
unfolding *P'-def register-mult*[*OF* ‹*register* Φ›, *symmetric*] **by** *simp*
also have ‹$\ldots =$ *cinner* $(\Phi\ (butterfly\ (ket\ j)\ (ket\ j)\ o_{CL}\ butterfly\ (ket\ i)\ (ket\ i)) *_V x')\ y'$›
unfolding *P-butter* **by** *simp*
also have ‹$\ldots =$ *cinner* $(\Phi\ 0 *_V x')\ y'$›
by (*metis butterfly-comp-butterfly complex-vector.scale-eq-0-iff orthogonal-ket that*(*3*))
also have ‹$\ldots = 0$›
by (*simp add*: *complex-vector.linear-0*)
finally show *?thesis*
by −
qed

define B' **where** ‹$B' = (\bigcup i{\in}UNIV.\ B\ i)$›

have *P'B*: ‹$P'\ i =$ *Proj* $(ccspan\ (B\ i))$› **for** i
unfolding *ccspanB S-def*
using *proj-P' Proj-on-own-range'*[*symmetric*] *is-Proj-algebraic* **by** *blast*

```
have ‹(∑ i∈UNIV. P' i) = Proj (ccspan B')›
proof (unfold B'-def, use finite[of UNIV] in induction)
  case empty
  show ?case by auto
next
  case (insert j M)
  have ‹(∑ i∈insert j M. P' i) = P' j + (∑ i∈M. P' i)›
    by (meson insert.hyps(1) insert.hyps(2) sum.insert)
  also have ‹... = Proj (ccspan (B j)) + Proj (ccspan (⋃ i∈M. B i))›
    unfolding P'B insert.IH[symmetric] by simp
  also have ‹... = Proj (ccspan (B j ∪ (⋃ i∈M. B i)))›
    apply (rule Proj-orthog-ccspan-union[symmetric])
    using orthoBiBj insert.hyps(2) by auto
  also have ‹... = Proj (ccspan (⋃ i∈insert j M. B i))›
    by auto
  finally show ?case
    by simp
qed

with sumP'id
have ccspanB': ‹ccspan B' = ⊤›
  by (metis Proj-range cblinfun-image-id)
hence cspanB': ‹cspan B' = UNIV›
 by (metis B'-def finiteB ccspan.rep-eq finite-UN-I finite-class.finite-UNIV closure-finite-cspan top-ccsubspace.rep-eq)

from orthoBiBj orthoB have orthoB': ‹is-ortho-set B'›
  unfolding B'-def is-ortho-set-def by blast
then have indepB': ‹cindependent B'›
  using is-ortho-set-cindependent by blast
have cardB': ‹card B' = CARD('b)›
  apply (subst complex-vector.dim-span-eq-card-independent[symmetric])
   apply (rule indepB')
  apply (subst cspanB')
  using cdim-UNIV-ell2 by auto

from orthoBiBj orthoB
have Bdisj: ‹B i ∩ B j = {}› if ‹i ≠ j› for i j
  unfolding is-ortho-set-def
  using that by fastforce

have cardBsame: ‹card (B i) = card (B j)› for i j
proof −
  define Si-to-Sj where ‹Si-to-Sj i j ψ = Φ (butterfly (ket j) (ket i)) *V ψ› for i j ψ
  have S2S2S: ‹Si-to-Sj j i (Si-to-Sj i j ψ) = ψ› if ‹ψ ∈ space-as-set (S i)› for i j ψ
    using that P'id
    by (simp add: Si-to-Sj-def cblinfun-apply-cblinfun-compose[symmetric] register-mult P-butter P'-def)
  also have lin[simp]: ‹clinear (Si-to-Sj i j)› for i j
    unfolding Si-to-Sj-def by simp
  have S2S: ‹Si-to-Sj i j x ∈ space-as-set (S j)› for i j x
  proof −
    have ‹Si-to-Sj i j x = P' j *V Si-to-Sj i j x›
      by (simp add: Si-to-Sj-def cblinfun-apply-cblinfun-compose[symmetric] register-mult P-butter P'-def)
    also have ‹P' j *V Si-to-Sj i j x ∈ space-as-set (S j)›
      by (simp add: S-def)
    finally show ?thesis by −
  qed
  have bij: ‹bij-betw (Si-to-Sj i j) (space-as-set (S i)) (space-as-set (S j))›
    apply (rule bij-betwI[where g=‹Si-to-Sj j i›])
    using S2S S2S2S by (auto intro!: funcsetI)
  have ‹cdim (space-as-set (S i)) = cdim (space-as-set (S j))›
    using lin apply (rule isomorphic-equal-cdim[where f=‹Si-to-Sj i j›])
    using bij by (auto simp: bij-betw-def)
  then show ?thesis
```

```
    by (metis complex-vector.dim-span-eq-card-independent cspanB indepB)
  qed

  have CARD'b: ‹CARD('b) = card (B ξ0) * CARD('a)›
  proof −
    have ‹CARD('b) = card B'›
      using cardB' by simp
    also have ‹... = (∑ i∈UNIV. card (B i))›
      unfolding B'-def apply (rule card-UN-disjoint)
      using finiteB Bdisj by auto
    also have ‹... = (∑ (i::'a)∈UNIV. card (B ξ0))›
      using cardBsame by metis
    also have ‹... = card (B ξ0) * CARD('a)›
      by auto
    finally show ?thesis by −
  qed

  obtain f where bij-f: ‹bij-betw f (UNIV::('a,'b) complement-domain-simple set) (B ξ0)›
    apply atomize-elim apply (rule finite-same-card-bij)
    using finiteB CARD-complement-domain[OF CARD'b] by auto

  define u where ‹u = (λ(ξ,α). Φ (butterfly (ket ξ) (ket ξ0)) *V f α)› for ξ :: 'a and α :: ‹('a,'b) comple-
ment-domain-simple›
  obtain U where Uapply: ‹U *V ket ξα = u ξα› for ξα
    apply atomize-elim
    apply (rule exI[of - ‹cblinfun-extension (range ket) (λk. u (inv ket k))›])
    apply (subst cblinfun-extension-apply)
     apply (rule cblinfun-extension-exists-finite-dim)
    by (auto simp add: inj-ket cindependent-ket)

  define eqa where ‹eqa a b = (if a = b then 1 else 0 :: complex)› for a b :: 'a
  define eqc where ‹eqc a b = (if a = b then 1 else 0 :: complex)› for a b :: ‹('a,'b) complement-domain-simple›
   define eqac where ‹eqac a b = (if a = b then 1 else 0 :: complex)› for a b :: ‹'a * ('a,'b) comple-
ment-domain-simple›

  have ‹cinner (U *V ket ξα) (U *V ket ξ'α') = eqac ξα ξ'α'› for ξα ξ'α'
  proof −
    obtain ξ α ξ' α' where ξα: ‹ξα = (ξ,α)› and ξ'α': ‹ξ'α' = (ξ',α')›
      apply atomize-elim by auto
     have ‹cinner (U *V ket (ξ,α)) (U *V ket (ξ', α')) = cinner (Φ (butterfly (ket ξ) (ket ξ0)) *V f α) (Φ
(butterfly (ket ξ') (ket ξ0)) *V f α')›
      unfolding Uapply u-def by simp
    also have ‹... = cinner ((Φ (butterfly (ket ξ') (ket ξ0)))* *V Φ (butterfly (ket ξ) (ket ξ0)) *V f α) (f α')›
      by (simp add: cinner-adj-left)
    also have ‹... = cinner (Φ (butterfly (ket ξ') (ket ξ0) *) *V Φ (butterfly (ket ξ) (ket ξ0)) *V f α) (f α')›
      by (metis (no-types, lifting) assms register-def)
    also have ‹... = cinner (Φ (butterfly (ket ξ0) (ket ξ') oCL butterfly (ket ξ) (ket ξ0)) *V f α) (f α')›
      by (simp add: register-mult cblinfun-apply-cblinfun-compose[symmetric])
    also have ‹... = cinner (Φ (eqa ξ' ξ *C butterfly (ket ξ0) (ket ξ0)) *V f α) (f α')›
      by (simp add: eqa-def cinner-ket)
    also have ‹... = eqa ξ' ξ * cinner (Φ (butterfly (ket ξ0) (ket ξ0)) *V f α) (f α')›
      by (smt (verit, ccfv-threshold) ‹clinear Φ› eqa-def cblinfun.scaleC-left cinner-commute
            cinner-scaleC-left cinner-zero-right complex-cnj-one complex-vector.linear-scale)
    also have ‹... = eqa ξ' ξ * cinner (P' ξ0 *V f α) (f α')›
      using P-butter P'-def by simp
    also have ‹... = eqa ξ' ξ * cinner (f α) (f α')›
      apply (subst P'id)
       apply (metis bij-betw-imp-surj-on bij-f complex-vector.span-base cspanB rangeI)
      by simp
    also have ‹... = eqa ξ' ξ * eqc α α'›
    proof −
      from normalB bij-f
      have aux1: ‹f α' ·C f α' ≠ 1 ⟹ eqa ξ' ξ = 0›
```

```
      by (metis bij-betw-imp-surj-on cnorm-eq-1 rangeI)
    from orthoB bij-f have aux2: ‹α ≠ α' ⟹ f α ·C f α' ≠ 0 ⟹ eqa ξ' ξ = 0›
      by (smt (z3) bij-betw-iff-bijections iso-tuple-UNIV-I is-ortho-set-def)
    from aux1 aux2 show ?thesis
      unfolding eqc-def by auto
  qed
  finally show ?thesis
    by (simp add: eqa-def eqac-def eqc-def ξ'α' ξα)
 qed
 then have [simp]: ‹isometry U›
  apply (rule-tac orthogonal-on-basis-is-isometry[where B=‹range ket›])
  using eqac-def by auto

 have ‹sandwich (U*) (Φ (butterfly (ket ξ) (ket η))) = butterfly (ket ξ) (ket η) ⊗o id-cblinfun› for ξ η
 proof (rule equal-ket, rename-tac ξ1α)
  fix ξ1α obtain ξ1 :: 'a and α :: ‹('a,'b) complement-domain-simple› where ξ1α: ‹ξ1α = (ξ1,α)›
    apply atomize-elim by auto
  have ‹sandwich (U*) (Φ (butterfly (ket ξ) (ket η))) *V ket ξ1α = U* *V Φ (butterfly (ket ξ) (ket η)) *V Φ
(butterfly (ket ξ1) (ket ξ0)) *V f α›
    by (simp add: sandwich-apply[abs-def] cblinfun-apply-cblinfun-compose ξ1α Uapply u-def)
  also have ‹... = U* *V Φ (butterfly (ket ξ) (ket η) oCL butterfly (ket ξ1) (ket ξ0)) *V f α›
    by (metis (no-types, lifting) assms butterfly-comp-butterfly lift-cblinfun-comp(4) register-mult)
  also have ‹... = U* *V Φ (eqa η ξ1 *C butterfly (ket ξ) (ket ξ0)) *V f α›
    by (simp add: eqa-def cinner-ket)
  also have ‹... = eqa η ξ1 *C U* *V Φ (butterfly (ket ξ) (ket ξ0)) *V f α›
    by (simp add: complex-vector.linear-scale)
  also have ‹... = eqa η ξ1 *C U* *V U *V ket (ξ, α)›
    unfolding Uapply u-def by simp
  also from ‹isometry U› have ‹... = eqa η ξ1 *C ket (ξ, α)›
    unfolding cblinfun-apply-cblinfun-compose[symmetric] by simp
  also have ‹... = (butterfly (ket ξ) (ket η) *V ket ξ1) ⊗s ket α›
    by (simp add: eqa-def tensor-ell2-scaleC1 tensor-ell2-ket)
  also have ‹... = (butterfly (ket ξ) (ket η) ⊗o id-cblinfun) *V ket ξ1α›
    by (simp add: ξ1α tensor-op-ket)
  finally show ‹sandwich (U*) (Φ (butterfly (ket ξ) (ket η))) *V ket ξ1α = (butterfly (ket ξ) (ket η) ⊗o
id-cblinfun) *V ket ξ1α›
    by −
 qed
 then have 1: ‹sandwich (U*) (Φ ϑ) = ϑ ⊗o id-cblinfun› for ϑ
  apply (rule clinear-eq-butterfly-ketI[THEN fun-cong, where x=ϑ, rotated −1])
  by (auto intro: bounded-clinear.clinear bounded-linear-intros register-bounded-clinear ‹register Φ›)

 have [simp]: ‹unitary U›
 proof −
  have ‹Φ (butterfly (ket ξ) (ket ξ1)) *S ⊤ ≤ U *S ⊤› for ξ ξ1
  proof −
   have *: ‹Φ (butterfly (ket ξ) (ket ξ0)) *V b ∈ space-as-set (U *S ⊤)› if ‹b ∈ B ξ0› for b
    apply (subst asm-rl[of ‹Φ (butterfly (ket ξ) (ket ξ0)) *V b = u (ξ, inv f b)›])
     apply (simp add: u-def, metis bij-betw-inv-into-right bij-f that)
    by (metis Uapply cblinfun-apply-in-image)

   have ‹Φ (butterfly (ket ξ) (ket ξ1)) *S ⊤ = Φ (butterfly (ket ξ) (ket ξ0)) *S Φ (butterfly (ket ξ0) (ket ξ0))
*S Φ (butterfly (ket ξ0) (ket ξ1)) *S ⊤›
    unfolding cblinfun-compose-image[symmetric] register-mult[OF assms]
    by simp
   also have ‹... ≤ Φ (butterfly (ket ξ) (ket ξ0)) *S Φ (butterfly (ket ξ0) (ket ξ0)) *S ⊤›
    by (meson cblinfun-image-mono top-greatest)
   also have ‹... = Φ (butterfly (ket ξ) (ket ξ0)) *S S ξ0›
    by (simp add: S-def P'-def P-butter)
   also have ‹... = Φ (butterfly (ket ξ) (ket ξ0)) *S ccspan (B ξ0)›
    by (simp add: ccspanB)
   also have ‹... = ccspan (Φ (butterfly (ket ξ) (ket ξ0)) ‘ B ξ0)›
    by (meson cblinfun-image-ccspan)
```

```
    also have ‹... ≤ U *_S ⊤›
      by (rule ccspan-leqI, use * in auto)
    finally show ?thesis by -
  qed
  moreover have ‹Φ id-cblinfun *_S ⊤ ≤ (SUP ξ∈UNIV. Φ (butterfly (ket ξ) (ket ξ)) *_S ⊤)›
    unfolding sum-butterfly-ket[symmetric]
    apply (subst complex-vector.linear-sum, simp)
    by (rule cblinfun-sum-image-distr)
  ultimately have ‹Φ id-cblinfun *_S ⊤ ≤ U *_S ⊤›
    by (metis (no-types, lifting) Proj-range Proj-top SUP-le-iff assms register-of-id top-le)
  then have ‹U *_S ⊤ = ⊤›
    using top.extremum-unique by auto
  with ‹isometry U› show ‹unitary U›
    by (rule surj-isometry-is-unitary)
qed

have ‹Φ ϑ = sandwich U (ϑ ⊗_o id-cblinfun)› for ϑ
proof -
  from 1 have ‹sandwich U (sandwich (U*) *_V Φ ϑ) = sandwich U (ϑ ⊗_o id-cblinfun)›
    by simp
  then show ?thesis
    by (simp flip: sandwich-compose cblinfun-apply-cblinfun-compose)
qed

then show ?thesis
  apply (rule-tac exI[of - U])
  by simp
qed

lemma register-decomposition-finite:
  fixes Φ :: ‹'a update ⇒ 'b::finite update›
  assumes [simp]: ‹register Φ›
  shows ‹∃ U :: ('a × ('a, 'b) complement-domain-simple) ell2 ⇒_CL 'b ell2. unitary U ∧
            (∀ϑ. Φ ϑ = sandwich U (ϑ ⊗_o id-cblinfun))›
proof -
  have inj-butterket: ‹inj (λa. butterfly (ket a) (ket a) :: 'a update)›
  proof (rule injI)
    fix x y :: 'a
    assume ‹butterfly (ket x) (ket x) = butterfly (ket y) (ket y)›
    then have ‹butterfly (ket x) (ket x) *_V ket x = butterfly (ket y) (ket y)  *_V ket x›
      by simp
    then show ‹x = y›
      apply (cases ‹x = y›)
      by (auto simp: cinner-ket)
  qed

  from cindependent-butterfly-ket
  have ‹cindependent (range (λa. butterfly (ket a) (ket a)) :: 'a update set)›
    apply (subgoal-tac ‹(range (λa. butterfly (ket a) (ket a)) :: 'a update set) ⊆ {butterfly (ket i) (ket j) |i j.
True}›)
    by (auto intro: complex-vector.dependent-mono)
  then have ‹cindependent (Φ ‘ range (λa. butterfly (ket a) (ket a)))›
    apply (rule complex-vector.linear-independent-injective-image[rotated])
    by (simp-all add: register-inj clinear-register)
  then have ‹finite (Φ ‘ range (λa. butterfly (ket a) (ket a)))›
    using cindependent-cfinite-dim-finite by blast
  then have ‹finite (range (λa. butterfly (ket a) (ket a)) :: 'a update set)›
    apply (rule Finite-Set.inj-on-finite[rotated -1, where f=Φ])
    by (simp-all add: register-inj)
  then have ‹finite (UNIV :: 'a set)›
    apply (rule Finite-Set.inj-on-finite[rotated -1, where f=‹λa. butterfly (ket a) (ket a)›])
     apply (rule inj-butterket)
    by simp
```

```
  then have ‹class.finite TYPE('a)›
    by intro-classes
  from register-decomposition-finite-aux[unoverload-type 'a, OF this]
  show ?thesis
    using assms by metis
qed

hide-fact register-decomposition-finite-aux

lemma complement-exists-simple:
  fixes F :: ‹'a update ⇒ 'b::finite update›
  assumes ‹register F›
  shows ‹∃ G :: ('a, 'b) complement-domain-simple update ⇒ 'b update. compatible F G ∧ iso-register (F;G)›
proof −
  note [[simproc del: Laws-Quantum.compatibility-warn]]
  obtain U :: ‹('a × ('a, 'b) complement-domain-simple) ell2 ⇒CL 'b ell2› 
    where [simp]: unitary U and F: ‹F a = sandwich U (a ⊗o id-cblinfun)› for a
    apply atomize-elim using assms by (rule register-decomposition-finite)
  define G :: ‹(('a, 'b) complement-domain-simple) update ⇒ 'b update› where ‹G b = sandwich U (id-cblinfun
⊗o b)› for b
  have [simp]: ‹register G›
    unfolding G-def apply (rule register-decomposition-converse) by simp
  have ‹F a oCL G b = G b oCL F a› for a b
  proof −
    have ‹F a oCL G b = sandwich U (a ⊗o b)›
      by (auto simp: F G-def sandwich-apply cblinfun-assoc-left lift-cblinfun-comp[OF unitaryD1]
          lift-cblinfun-comp[OF comp-tensor-op] ‹unitary U›)
    moreover have ‹G b oCL F a = sandwich U (a ⊗o b)›
      by (auto simp: F G-def sandwich-apply cblinfun-assoc-left lift-cblinfun-comp[OF unitaryD1]
          lift-cblinfun-comp[OF comp-tensor-op] ‹unitary U›)
    ultimately show ?thesis by simp
  qed
  then have [simp]: ‹compatible F G›
    by (auto simp: compatible-def ‹register F› ‹register G›)
  moreover have ‹iso-register (F;G)›
  proof −
    have ‹(F;G) (a ⊗o b) = sandwich U (a ⊗o b)› for a b
      by (auto simp: register-pair-apply F G-def sandwich-apply cblinfun-assoc-left lift-cblinfun-comp[OF uni-
taryD1]
          lift-cblinfun-comp[OF comp-tensor-op] ‹unitary U›)
    then have FG: ‹(F;G) = sandwich U›
      apply (rule tensor-extensionality[rotated −1])
      by (simp-all add: bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose bounded-cbilinear.add-right
clinearI unitary-sandwich-register)

    define I where ‹I = sandwich (U*)› for x
    have [simp]: ‹register I›
      by (simp add: I-def unitary-sandwich-register)
    have IFG: ‹I o (F;G) = id›
      by (auto intro!:ext simp: I-def[abs-def] FG sandwich-apply lift-cblinfun-comp[OF unitaryD1]
          ‹unitary U› cblinfun-assoc-left)
    have FGI: ‹(F;G) o I = id›
      by (auto intro!:ext simp: I-def[abs-def] FG sandwich-apply cblinfun-assoc-left
          lift-cblinfun-comp[OF unitaryD1] lift-cblinfun-comp[OF unitaryD2] ‹unitary U›)
    from IFG FGI show ‹iso-register (F;G)›
      by (auto intro!: iso-registerI)
  qed
  ultimately show ?thesis
    apply (rule-tac exI[of - G]) by (auto)
qed

unbundle no cblinfun-syntax
unbundle no lattice-syntax
```

unbundle *no register-syntax*

end

16 Generic laws about complements, instantiated quantumly

theory *Laws-Complement-Quantum*
imports *Laws-Quantum Axioms-Complement-Quantum*
begin

unbundle *register-syntax*
notation *cblinfun-compose* (**infixl** $*_u$ *55*)
notation *tensor-op* (**infixr** $\otimes_u$ *70*)

definition ‹*complements F G* $\longleftrightarrow$ *compatible F G* $\wedge$ *iso-register* (*F*;*G*)›

lemma *complementsI*: ‹*compatible F G* $\Longrightarrow$ *iso-register* (*F*;*G*) $\Longrightarrow$ *complements F G*›
using *complements-def* **by** *blast*

lemma *complement-exists*:
fixes *F* :: ‹'*a*::*type update* $\Rightarrow$ '*b*::*type update*›
assumes ‹*register F*›
shows ‹*let* '*c*::*type* = *register-decomposition-basis F in*
$\exists$ *G* :: '*c update* $\Rightarrow$ '*b update*. *complements F G*›
by (*simp add*: *assms complement-exists complements-def*)

lemma *complements-sym*: ‹*complements G F*› **if** ‹*complements F G*›
proof (*rule complementsI*)
show [*simp*]: ‹*compatible G F*›
using *compatible-sym complements-def that* **by** *blast*
from *that* **have** ‹*iso-register* (*F*;*G*)›
by (*meson complements-def*)
then obtain *I* **where** [*simp*]: ‹*register I*› **and** ‹(*F*;*G*) *o I* = *id*› **and** ‹*I o* (*F*;*G*) = *id*›
using *iso-register-def* **by** *blast*
have ‹*register* (*swap o I*)›
using ‹*register I*› *register-comp register-swap* **by** *blast*
moreover have ‹(*G*;*F*) *o* (*swap o I*) = *id*›
by (*simp add*: ‹(*F*;*G*) ∘ *I* = *id*› *rewriteL-comp-comp*)
moreover have ‹(*swap o I*) *o* (*G*;*F*) = *id*›
by (*metis* (*no-types*, *opaque-lifting*) *swap-swap* ‹*I* ∘ (*F*;*G*) = *id*› *calculation*(*2*) *comp-def eq-id-iff*)
ultimately show ‹*iso-register* (*G*;*F*)›
using ‹*compatible G F*› *iso-register-def pair-is-register* **by** *blast*
qed

definition *complement* :: ‹('*a*::*type update* $\Rightarrow$ '*b*::*finite update*) $\Rightarrow$ (('*a*,'*b*) *complement-domain-simple update* $\Rightarrow$ '*b update*)› **where**
‹*complement F* = (*SOME G* :: ('*a*, '*b*) *complement-domain-simple update* $\Rightarrow$ '*b update*. *compatible F G* $\wedge$ *iso-register* (*F*;*G*))›

lemma *register-complement*[*simp*]: ‹*register* (*complement F*)› **if** ‹*register F*›
using *complement-exists-simple*[*OF that*]
by (*metis* (*no-types*, *lifting*) *compatible-def complement-def some-eq-imp*)

lemma *complement-is-complement*[*simp*]:
assumes ‹*register F*›
shows ‹*complements F* (*complement F*)›
using *complement-exists-simple*[*OF assms*] **unfolding** *complements-def*
by (*metis* (*mono-tags*, *lifting*) *complement-def some-eq-imp*)

lemma *complement-unique*:
assumes ‹*complements F G*›
assumes ‹*complements F G'*›
shows ‹*equivalent-registers G G'*›

apply (*rule complement-unique*[**where** *F*=*F*])
using *assms* **unfolding** *complements-def* **by** *auto*

lemma *complement-unique'*:
assumes ‹*complements F G*›
shows ‹*equivalent-registers G* (*complement F*)›
apply (*rule complement-unique*[**where** *F*=*F*])
using *assms* **unfolding** *complements-def* **using** *compatible-register1 complement-is-complement complements-def*
by *blast*+

lemma *compatible-complement*[*simp*]: ‹*register F* ⟹ *compatible F* (*complement F*)›
using *complement-is-complement complements-def* **by** *blast*

lemma *complements-register-tensor*:
assumes [*simp*]: ‹*register F*› ‹*register G*›
shows ‹*complements* (*F* ⊗*r G*) (*complement F* ⊗*r complement G*)›
proof (*rule complementsI*)
have [*iff*]: ‹*iso-register* (*F*;*complement F*)› ‹*iso-register* (*G*;*complement G*)›
using *complements-def* **by** *fastforce*+

have *sep4*: ‹*separating TYPE*('*z*::*type*) {(*a* ⊗*u b*) ⊗*u* (*c* ⊗*u d*) |*a b c d*. *True*}›
apply (*rule separating-tensor'*[**where** *A*=‹{(*a* ⊗*u b*) |*a b*. *True*}› **and** *B*=‹{(*c* ⊗*u d*) |*c d*. *True*}›])
apply (*rule separating-tensor'*[**where** *A*=*UNIV* **and** *B*=*UNIV*]) **apply** *auto*[*3*]
apply (*rule separating-tensor'*[**where** *A*=*UNIV* **and** *B*=*UNIV*]) **apply** *auto*[*3*]
by *auto*
show *compat*: ‹*compatible* (*F* ⊗*r G*) (*complement F* ⊗*r complement G*)›
by (*metis assms*(*1*) *assms*(*2*) *compatible-register-tensor complement-is-complement complements-def*)
let *?reorder* = ‹((*Fst o Fst*; *Snd o Fst*); (*Fst o Snd*; *Snd o Snd*))›
have [*simp*]: ‹*register ?reorder*›
by *auto*
have [*simp*]: ‹*?reorder* ((*a* ⊗*u b*) ⊗*u* (*c* ⊗*u d*)) = ((*a* ⊗*u c*) ⊗*u* (*b* ⊗*u d*))›
for *a*::‹'*t*::*type update*› **and** *b*::‹'*u*::*type update*› **and** *c*::‹'*v*::*type update*› **and** *d*::‹'*w*::*type update*›
by (*simp add*: *register-pair-apply Fst-def Snd-def comp-tensor-op*)
have [*simp*]: ‹*iso-register ?reorder*›
apply (*rule iso-registerI*[*of* - *?reorder*]) **apply** *auto*[*2*]
apply (*rule register-eqI*[*OF sep4*]) **apply** *auto*[*3*]
apply (*rule register-eqI*[*OF sep4*]) **by** *auto*
have ‹(*F* ⊗*r G*; *complement F* ⊗*r complement G*) = ((*F*; *complement F*) ⊗*r* (*G*; *complement G*)) *o ?reorder*›
apply (*rule register-eqI*[*OF sep4*])
by (*auto intro*!: *register-preregister register-comp register-tensor-is-register pair-is-register*
simp: *compat register-pair-apply comp-tensor-op*)
moreover have ‹*iso-register* . . . ›
using *assms complement-is-complement complements-def*
by (*auto intro*!: *iso-register-comp iso-register-tensor-is-iso-register*)
ultimately show ‹*iso-register* (*F* ⊗*r G*;*complement F* ⊗*r complement G*)›
by *simp*
qed

definition *is-unit-register* **where**
‹*is-unit-register U* ⟷ *complements U id*›

lemma *register-unit-register*[*simp*]: ‹*is-unit-register U* ⟹ *register U*›
by (*simp add*: *compatible-def complements-def is-unit-register-def*)

lemma *unit-register-compatible*[*simp*]: ‹*compatible U X*› **if** ‹*is-unit-register U*› ‹*register X*›
by (*metis compatible-comp-right complements-def id-comp is-unit-register-def that*(*1*) *that*(*2*))

lemma *unit-register-compatible'*[*simp*]: ‹*compatible X U*› **if** ‹*is-unit-register U*› ‹*register X*›
using *compatible-sym that*(*1*) *that*(*2*) *unit-register-compatible* **by** *blast*

lemma *compatible-complement-left*[*simp*]: ‹*register X* ⟹ *compatible* (*complement X*) *X*›
using *compatible-sym complement-is-complement complements-def* **by** *blast*

lemma *compatible-complement-right*[*simp*]: ‹*register X* ⟹ *compatible X* (*complement X*)›
using *complement-is-complement complements-def* **by** *blast*

lemma *unit-register-pair*[*simp*]: ‹*equivalent-registers X* (*U*; *X*)› **if** [*simp*]: ‹*is-unit-register U*› ‹*register X*›
proof −
from *complement-exists*[*OF* ‹*register X*›]
have ‹*let* 'x::*type* = *register-decomposition-basis X in equivalent-registers X* (*U*; *X*)›
proof *with-type-mp*
note [[*simproc del*: *compatibility-warn*]]

with-type-case
then obtain *compX* :: ‹'*x update* ⇒ '*b update*› **where** *compX*: ‹*complements X compX*›
by *blast*
then have [*simp*]: ‹*register compX*› ‹*compatible X compX*›
by (*auto simp add*: *compatible-def complements-def*)
have [*iff*]: ‹*iso-register* (*X*;*compX*)›
using *compX complements-def* **by** *blast*

have ‹*equivalent-registers id* (*U*; *id*)›
using *complements-def is-unit-register-def iso-register-equivalent-id that*(*1*) **by** *blast*
also have ‹*equivalent-registers* ... (*U*; (*X*; *compX*))›
apply (*rule equivalent-registers-pair-right*)
by (*auto intro*!: *unit-register-compatible*)
also have ‹*equivalent-registers* ... ((*U*; *X*); *compX*)›
apply (*rule equivalent-registers-assoc*)
by *auto*
finally have ‹*complements* (*U*; *X*) *compX*›
by (*auto simp*: *equivalent-registers-def complements-def*)
moreover have ‹*equivalent-registers* (*X*; *compX*) *id*›
using *compX complements-def equivalent-registers-sym iso-register-equivalent-id* **by** *blast*
ultimately show ‹*equivalent-registers X* (*U*; *X*)›
by (*meson complement-unique compX complements-sym*)
qed
from *this*[*cancel-with-type*]
show ‹*equivalent-registers X* (*U*; *X*)›
by −
qed

lemma *unit-register-compose-left*:
assumes [*simp*]: ‹*is-unit-register U*›
assumes [*simp*]: ‹*register A*›
shows ‹*is-unit-register* (*A o U*)›
proof −
from *complement-exists*[*OF* ‹*register A*›]
have ‹*let* 'x::*type* = *register-decomposition-basis A in is-unit-register* (*A o U*)›
proof *with-type-mp*
note [[*simproc del*: *compatibility-warn*]]
with-type-case
then obtain *compA* :: ‹'*x update* ⇒ '*c update*› **where** *compX*: ‹*complements A compA*›
by *blast*
then have [*simp*]: ‹*register compA*› ‹*compatible A compA*›
by (*auto simp add*: *compatible-def complements-def*)
have [*iff*]: ‹*iso-register* (*A*;*compA*)›
using *compX complements-def* **by** *blast*

have ‹*compatible* (*A* ∘ *U*) *A*›
by (*metis assms*(*1*) *assms*(*2*) *comp-id compatible-comp-inner complements-def is-unit-register-def*)
then have *compat'*[*simp*]: ‹*compatible* (*A o U*) (*A*; *compA*)›
by (*auto intro*!: *compatible3'*)
moreover have ‹*equivalent-registers* (*A*; *compA*) *id*›
using *compX complements-def equivalent-registers-sym iso-register-equivalent-id* **by** *blast*
ultimately have *compat*[*simp*]: ‹*compatible* (*A o U*) *id*›

```
    using equivalent-registers-compatible2 by blast

  have aux: ‹equivalent-registers (U;id) id›
    using assms(1) equivalent-registers-sym register-id unit-register-pair by blast

  have ‹equivalent-registers (A o U; id) (A o U; (A; compA))›
    by (auto intro!: equivalent-registers-pair-right)
  also have ‹equivalent-registers ... (A o U; (A o id; compA))›
    by (auto intro!: equivalent-registers-refl pair-is-register)
  also have ‹equivalent-registers ... ((A o U; A o id); compA)›
    apply (intro equivalent-registers-assoc compatible-comp-inner)
    by auto
  also have ‹equivalent-registers ... (A o (U; id); compA)›
   by (metis (no-types, opaque-lifting) assms(1) assms(2) calculation complements-def equivalent-registers-sym
equivalent-registers-trans is-unit-register-def register-comp-pair)
  also have ‹equivalent-registers ... (A o id; compA)›
    apply (intro equivalent-registers-pair-left equivalent-registers-comp)
    using aux by (auto simp: assms)
  also have ‹equivalent-registers ... id›
    by (simp add: ‹equivalent-registers (A;compA) id›)
  finally show ‹is-unit-register (A o U)›
    using compat complementsI equivalent-registers-sym is-unit-register-def iso-register-equivalent-id by blast
qed
from this[cancel-with-type]
show ?thesis
  by -
qed

lemma unit-register-compose-right:
  assumes [simp]: ‹is-unit-register U›
  assumes [simp]: ‹iso-register A›
  shows ‹is-unit-register (U o A)›
proof (unfold is-unit-register-def, rule complementsI)
  show ‹compatible (U ∘ A) id›
    by (simp add: iso-register-is-register)
  have 1: ‹iso-register ((U;id) ∘ A ⊗r id)›
   by (meson assms(1) assms(2) complements-def is-unit-register-def iso-register-comp iso-register-id iso-register-tensor-is-iso-regist
  have 2: ‹id ∘ ((U;id) ∘ A ⊗r id) = (U ∘ A;id)›
   by (metis assms(1) assms(2) complements-def fun.map-id is-unit-register-def iso-register-id iso-register-is-register
pair-o-tensor)
  show ‹iso-register (U ∘ A;id)›
    using 1 2 by auto
qed

lemma unit-register-unique:
  assumes ‹is-unit-register F›
  assumes ‹is-unit-register G›
  shows ‹equivalent-registers F G›
proof -
  have ‹complements F id› ‹complements G id›
    using assms by (metis complements-def equivalent-registers-def id-comp is-unit-register-def)+
  then show ?thesis
    by (meson complement-unique complements-sym equivalent-registers-sym equivalent-registers-trans)
qed

lemma unit-register-domains-isomorphic:
  fixes F :: ‹'a::type update ⇒ 'c::type update›
  fixes G :: ‹'b::type update ⇒ 'd::type update›
  assumes ‹is-unit-register F›
  assumes ‹is-unit-register G›
  shows ‹∃ I :: 'a update ⇒ 'b update. iso-register I›
proof -
  have ‹is-unit-register ((λd. tensor-op id-cblinfun d) o G)›
```

by (*simp add*: *assms*(*2*) *unit-register-compose-left*)
moreover have ‹*is-unit-register* ((λ*c*. *tensor-op c id-cblinfun*) *o F*)›
using *assms*(*1*) *register-tensor-left unit-register-compose-left* **by** *blast*
ultimately have ‹*equivalent-registers* ((λ*d*. *tensor-op id-cblinfun d*) *o G*) ((λ*c*. *tensor-op c id-cblinfun*) *o F*)›
using *unit-register-unique* **by** *blast*
then show *?thesis*
unfolding *equivalent-registers-def* **by** *auto*
qed

lemma *id-complement-is-unit-register*[*simp*]: ‹*is-unit-register* (*complement id*)›
by (*metis is-unit-register-def complement-is-complement complements-def complements-sym equivalent-registers-def id-comp register-id*)

type-synonym *unit-register-domain* = ‹(*unit*, *unit*) *complement-domain-simple*›
definition *unit-register* :: ‹*unit-register-domain update* ⇒ *'a*::*type update*› **where** ‹*unit-register* = (*SOME U*. *is-unit-register U*)›

lemma *unit-register-is-unit-register*[*simp*]: ‹*is-unit-register* (*unit-register* :: *unit-register-domain update* ⇒ *'a*::*type update*)›
proof −
note [[*simproc del*: *compatibility-warn*]]
let *?U* = ‹*unit-register* :: *unit-register-domain update* ⇒ *'a*::*type update*›
let *?U1* = ‹*complement id* :: *unit-register-domain update* ⇒ *unit update*›
from *complement-exists*[*OF register-id*[**where** *'a*=*'a*]]
have ‹*let 'x*::*type* = *register-decomposition-basis* (*id*::*'a update* ⇒ -) *in is-unit-register ?U*›
proof *with-type-mp*
with-type-case
then obtain *U2* :: ‹*'x update* ⇒ *'a update*› **where** *comp1*: ‹*complements id U2*›
by *blast*
then have [*simp*]: ‹*register U2*› ‹*compatible id U2*› ‹*compatible id U2*›
by (*auto simp add*: *compatible-def complements-def*)

have ‹*is-unit-register ?U1*› ‹*is-unit-register U2*›
by (*auto simp*: *comp1 complements-sym is-unit-register-def*)

then obtain *I* :: ‹*unit-register-domain update* ⇒ *'x update*› **where** ‹*iso-register I*›
apply *atomize-elim* **by** (*rule unit-register-domains-isomorphic*)
with ‹*is-unit-register U2*› **have** ‹*is-unit-register* (*U2 o I*)›
by (*rule unit-register-compose-right*)
then show ‹*is-unit-register ?U*›
by (*metis someI-ex unit-register-def*)
qed
from *this*[*cancel-with-type*]
show *?thesis*
by −
qed

lemma *unit-register-domain-tensor-unit*:
fixes *U* :: ‹*'a*::*type update* ⇒ -›
assumes ‹*is-unit-register U*›
shows ‹∃ *I* :: *'b*::*type update* ⇒ (*'a*∗*'b*) *update*. *iso-register I*›

proof −
from *complement-exists*[*OF register-id*[**where** *'a*=*'b*]]
have ‹*let 'x*::*type* = *register-decomposition-basis* (*id* :: *'b update* ⇒ -) *in*
∃ *I* :: *'b*::*type update* ⇒ (*'a*∗*'b*) *update*. *iso-register I*›
proof *with-type-mp*
note [[*simproc del*: *compatibility-warn*]]
with-type-case
assume ‹∃ *G* :: *'x update* ⇒ *'b update*. *complements id G*›
then obtain *U'* :: ‹*'x update* ⇒ *'b update*› **where** *comp*: ‹*complements id U'*›
by *blast*

```
    then have [simp]: ‹register U'› ‹compatible id U'› ‹compatible U' id›
      by (auto simp add: compatible-def complements-def)
    have ‹is-unit-register U'›
      by (simp add: comp complements-sym is-unit-register-def)

    have ‹equivalent-registers (id :: 'b update ⇒ -) (U'; id)›
      using comp complements-def complements-sym iso-register-equivalent-id by blast
    then obtain J :: ‹'b update ⇒ ('x * 'b) update)› where ‹iso-register J›
      using equivalent-registers-def iso-register-inv by blast
    moreover obtain K :: ‹'x update ⇒ 'a update› where ‹iso-register K›
      apply atomize-elim
      using ‹is-unit-register U'› assms
      by (rule unit-register-domains-isomorphic)
    ultimately have ‹iso-register ((K ⊗r id) o J)›
      by auto
    then show ‹∃ I :: 'b::type update ⇒ ('a*'b) update. iso-register I›
      by auto
  qed
  from this[cancel-with-type]
  show ?thesis
    by−
qed

lemma compatible-complement-pair1:
  assumes ‹compatible F G›
  shows ‹compatible F (complement (F;G))›
  by (metis assms compatible-comp-left compatible-complement-right pair-is-register register-Fst register-pair-Fst)

lemma compatible-complement-pair2:
  assumes [simp]: ‹compatible F G›
  shows ‹compatible G (complement (F;G))›
proof −
  have ‹compatible (F;G) (complement (F;G))›
    by simp
  then have ‹compatible ((F;G) o Snd) (complement (F;G))›
    by auto
  then show ?thesis
    by (auto simp: register-pair-Snd)
qed

lemma equivalent-complements:
  assumes ‹complements F G›
  assumes ‹equivalent-registers G G'›
  shows ‹complements F G'›
  apply (rule complementsI)
  apply (metis assms(1) assms(2) compatible-comp-right complements-def equivalent-registers-def iso-register-is-register)
  by (metis assms(1) assms(2) complements-def equivalent-registers-def equivalent-registers-pair-right iso-register-comp)

lemma complements-complement-pair:
  assumes [simp]: ‹compatible F G›
  assumes FG': ‹complements (F;G) FG'›
  shows ‹complements F (G; FG')›
proof (rule complementsI)
  note [[simproc del: compatibility-warn]]
  have ‹compatible (F;G) FG'›
    using FG' complements-def by auto
  then have [simp]: ‹compatible F FG'›
    by (smt (verit) assms(1) compatibleI compatible-register1 compatible-register2 cblinfun-compose-id-right register-of-id register-pair-apply' swap-registers)
  have [simp]: ‹compatible G FG'›
    by (smt (verit) register-pair-apply ‹compatible (F;G) FG'› assms(1) compatibleI compatible-register1 compatible-register2 cblinfun-compose-id-right register-of-id swap-registers)
```

have ‹*equivalent-registers* (*F*; (*G*; *FG′*)) ((*F*;*G*); *FG′*)›
apply (*rule equivalent-registers-assoc*)
apply *simp*
apply (*smt* (*verit*) ‹*compatible* (*F*;*G*) *FG′*› *assms*(*1*) *compatibleI compatible-register1 compatible-register2 cblinfun-compose-id-right register-of-id register-pair-apply′ swap-registers*)
by (*smt* (*verit*) *register-pair-apply* ‹*compatible* (*F*;*G*) *FG′*› *assms*(*1*) *compatibleI compatible-register1 compatible-register2 cblinfun-compose-id-right register-of-id swap-registers*)
also have ‹*equivalent-registers* ... *id*›
by (*meson assms complement-is-complement complements-def equivalent-registers-sym iso-register-equivalent-id pair-is-register*)
finally show ‹*iso-register* (*F*;(*G*;*FG′*))›
using *equivalent-registers-sym iso-register-equivalent-id* **by** *blast*
show ‹*compatible F* (*G*;*FG′*)›
by (*auto intro*!: *compatible3′*)
qed

lemma *equivalent-registers-complement*:
assumes ‹*equivalent-registers F G*›
assumes ‹*complements F F′*›
assumes ‹*complements G G′*›
shows ‹*equivalent-registers F′ G′*›
by (*meson complement-unique assms*(*1*) *assms*(*2*) *assms*(*3*) *complements-sym equivalent-complements*)

lemma *equivalent-registers-complement′*:
assumes ‹*equivalent-registers F G*›
shows ‹*equivalent-registers* (*complement F*) (*complement G*)›
using *assms* **apply** (*rule equivalent-registers-complement*)
using *assms complement-is-complement equivalent-registers-register-left equivalent-registers-register-right*
by *blast*+

lemma *complements-complement-pair′*:
assumes [*simp*]: ‹*compatible F G*›
assumes *FG′*: ‹*complements* (*F*;*G*) *FG′*›
shows ‹*complements G* (*F*; *FG′*)›
proof −
have ‹*equivalent-registers* (*F*;*G*) (*G*;*F*)›
using *assms*(*1*) *equivalent-registers-def iso-register-swap pair-is-register pair-o-swap* **by** *blast*
with *FG′* **have** ∗: ‹*complements* (*G*;*F*) *FG′*›
by (*meson complements-sym equivalent-complements*)
show *?thesis*
apply (*rule complements-complement-pair*)
using ∗ **by** (*simp-all add*: *compatible-sym*)
qed

lemma *complements-chain*:
assumes [*simp*]: ‹*register F*› ‹*register G*›
shows ‹*complements* (*F o G*) (*complement F*; *F o complement G*)›
proof (*rule complementsI*)
show ‹*compatible* (*F o G*) (*complement F*; *F o complement G*)›
by *auto*
have ‹*equivalent-registers* (*F* ∘ *G*;(*complement F*;*F* ∘ *complement G*)) (*F* ∘ *G*;(*F* ∘ *complement G*;*complement F*))›
apply (*rule equivalent-registersI*[**where** *I*=‹*id* ⊗*r swap*›])
by (*auto intro*!: *iso-register-tensor-is-iso-register simp*: *pair-o-tensor*)
also have ‹*equivalent-registers* ... ((*F* ∘ *G*;*F* ∘ *complement G*);*complement F*)›
apply (*rule equivalent-registersI*[**where** *I*=*assoc*])
by (*auto intro*!: *iso-register-tensor-is-iso-register simp*: *pair-o-tensor*)
also have ‹*equivalent-registers* ... (*F o* (*G*; *complement G*);*complement F*)›
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *calculation compatible-complement-right equivalent-registers-sym equivalent-registers-trans register-comp-pair*)
also have ‹*equivalent-registers* ... (*F o id*;*complement F*)›
apply (*rule equivalent-registers-pair-left*, *simp*)
apply (*rule equivalent-registers-comp*, *simp*)

by (*metis assms*(*2*) *complement-is-complement complements-def equivalent-registers-def iso-register-def*)
also have ‹*equivalent-registers* ... *id*›
by (*metis assms*(*1*) *comp-id complement-is-complement complements-def equivalent-registers-def iso-register-def*)
finally show ‹*iso-register* (*F* ∘ *G*;(*complement F*;*F* ∘ *complement G*))›
using *equivalent-registers-sym iso-register-equivalent-id* **by** *blast*
qed

lemma *complements-Fst-Snd*[*simp*]: ‹*complements Fst Snd*›
by (*auto intro*!: *complementsI simp*: *pair-Fst-Snd*)

lemma *complements-Snd-Fst*[*simp*]: ‹*complements Snd Fst*›
by (*auto intro*!: *complementsI simp flip*: *swap-def*)

lemma *compatible-unit-register*[*simp*]: ‹*register F* ⟹ *compatible F unit-register*›
using *compatible-sym unit-register-compatible unit-register-is-unit-register* **by** *blast*

lemma *complements-id-unit-register*[*simp*]: ‹*complements id unit-register*›
using *complements-sym is-unit-register-def unit-register-is-unit-register* **by** *blast*

lemma *complements-iso-unit-register*: ‹*iso-register I* ⟹ *is-unit-register U* ⟹ *complements I U*›
using *complements-sym equivalent-complements is-unit-register-def iso-register-equivalent-id* **by** *blast*

lemma *iso-register-complement-is-unit-register*[*simp*]:
assumes ‹*iso-register F*›
shows ‹*is-unit-register* (*complement F*)›
by (*meson assms complement-is-complement complements-sym equivalent-complements equivalent-registers-sym is-unit-register-def iso-register-equivalent-id iso-register-is-register*)

Adding support for *is-unit-register F* and *complements F G* to the [*register*] attribute

lemmas [*register-attribute-rule*] = *is-unit-register-def*[*THEN iffD1*] *complements-def*[*THEN iffD1*]
lemmas [*register-attribute-rule-immediate*] = *asm-rl*[*of* ‹*is-unit-register* -›]

no-notation *cblinfun-compose* (**infixl** ∗*u* *55*)
no-notation *tensor-op* (**infixr** ⊗*u* *70*)
unbundle *no register-syntax*

end

17 More derived facts about quantum registers

This theory contains some derived facts that cannot be placed in theory *Quantum-Extra* because they depend on *Laws-Complement-Quantum*.

theory *Quantum-Extra2*
imports
Laws-Complement-Quantum
Quantum
begin

unbundle *register-syntax*

definition *empty-var* :: ‹'*a*::{*CARD-1*,*enum*} *update* ⇒ '*b update*› **where**
empty-var a = *one-dim-iso a* ∗*C id-cblinfun*

lemma *is-unit-register-empty-var*[*register*]: ‹*is-unit-register empty-var*› (**is** ‹*is-unit-register ?e*›)
proof −
note *continuous-map-compose*[*trans*]
have ‹*continuous-map weak-star-topology euclidean* (*id* :: '*a update* ⇒ -)›
by *simp*
also have ‹*continuous-map euclidean euclidean one-dim-iso*›
by (*simp add*: *clinear-continuous-at continuous-at-imp-continuous-on*)
also have ‹*continuous-map euclidean euclidean* (λ*a*. *a* ∗*C id-cblinfun*)›

```
    by (simp add: continuous-at-imp-continuous-on)
  also have ‹continuous-map euclidean weak-star-topology (id :: 'b update ⇒ -)›
    by (metis comp-id fun.map-ident weak-star-topology-weaker-than-euclidean)
  finally have ‹continuous-map weak-star-topology weak-star-topology (λa :: 'a update. one-dim-iso a *C id-cblinfun
:: 'b update)›
    by (simp add: o-def)
  then have [simp]: ‹register ?e›
    unfolding register-def empty-var-def
    by (simp add: clinearI scaleC-left.add)
  have [simp]: ‹compatible ?e id›
    by (auto intro!: compatibleI simp: empty-var-def)
  have [simp]: ‹iso-register (?e; id)›
    by (auto intro!: same-range-equivalent range-eqI[where x=‹id-cblinfun ⊗o -›]
        simp del: id-cblinfun-eq-1 simp flip: iso-register-equivalent-id simp: register-pair-apply)
  show ?thesis
    by (auto intro!: complementsI simp: is-unit-register-def)
qed

instance complement-domain-simple :: (type, type) default ..

unbundle no register-syntax

end
theory Pure-States
  imports Quantum-Extra2 HOL−Eisbach.Eisbach
begin

unbundle cblinfun-syntax
unbundle register-syntax

definition ‹pure-state-target-vector F ηF = (if ket default ∈ range (cblinfun-apply (F (butterfly ηF ηF)))
   then ket default
   else (SOME η'. norm η' = 1 ∧ η' ∈ range (cblinfun-apply (F (butterfly ηF ηF)))))›

lemma pure-state-target-vector-eqI:
  assumes ‹F (butterfly ηF ηF) = G (butterfly ηG ηG)›
  shows ‹pure-state-target-vector F ηF = pure-state-target-vector G ηG›
  by (simp add: assms pure-state-target-vector-def)

lemma pure-state-target-vector-ket-default: ‹pure-state-target-vector F ηF = ket default› if ‹ket default ∈ range
(cblinfun-apply (F (butterfly ηF ηF)))›
  by (simp add: pure-state-target-vector-def that)

lemma
  assumes [simp]: ‹ηF ≠ 0› ‹register F›
  shows pure-state-target-vector-in-range: ‹pure-state-target-vector F ηF ∈ range ((*V) (F (selfbutter ηF)))› (is
?range)
    and pure-state-target-vector-norm: ‹norm (pure-state-target-vector F ηF) = 1› (is ?norm)
proof −
  from assms have ‹selfbutter ηF ≠ 0›
    by (metis butterfly-0-right complex-vector.scale-zero-right inj-selfbutter-upto-phase)
  then have ‹F (selfbutter ηF) ≠ 0›
    using register-inj[OF ‹register F›, THEN injD, where y=0]
    by (auto simp: complex-vector.linear-0 clinear-register)
  then obtain ψ' where ψ': ‹F (selfbutter ηF) *V ψ' ≠ 0›
    by (meson cblinfun-eq-0-on-UNIV-span complex-vector.span-UNIV)
  have ex: ‹∃ψ. norm ψ = 1 ∧ ψ ∈ range ((*V) (F (selfbutter ηF)))›
    apply (rule exI[of - ‹(F (selfbutter ηF) *V ψ') /C norm (F (selfbutter ηF) *V ψ')›])
    using ψ'
    apply (simp add: norm-inverse cblinfun.scaleC-right)
    by (simp flip: cblinfun.scaleC-right)
  then show ?range
    by (metis (mono-tags, lifting) pure-state-target-vector-def verit-sko-ex')
```

show *?norm*
apply (*simp add*: *pure-state-target-vector-def*)
using *ex* **by** (*metis* (*mono-tags*, *lifting*) *verit-sko-ex'*)
qed

lemma *pure-state-target-vector-correct*:
assumes [*simp*]: ‹$\eta \neq 0$› ‹*register F*›
shows ‹F (*selfbutter* η) $*_V$ *pure-state-target-vector F* $\eta \neq 0$›
proof −
obtain ψ **where** ψ: ‹F (*selfbutter* η) ψ = *pure-state-target-vector F* η›
apply *atomize-elim*
using *pure-state-target-vector-in-range*[*OF assms*]
by (*smt* (*verit*, *best*) *image-iff top-ccsubspace.rep-eq top-set-def*)

define *n* **where** ‹n = *cinner* η η›
then have ‹$n \neq 0$›
by *auto*

have *pure-state-target-vector-neq0*: ‹*pure-state-target-vector F* $\eta \neq 0$›
using *pure-state-target-vector-norm*[*OF assms*]
by *auto*

have ‹F (*selfbutter* η) $*_V$ *pure-state-target-vector F* η = F (*selfbutter* η) $*_V$ F (*selfbutter* η) $*_V$ ψ›
by (*simp add*: ψ)
also have ‹... = n $*_C$ F (*selfbutter* η) $*_V$ ψ›
by (*simp flip*: *cblinfun-apply-cblinfun-compose add*: *register-mult register-scaleC n-def*)
also have ‹... = n $*_C$ *pure-state-target-vector F* η›
by (*simp add*: ψ)
also have ‹... ≠ 0›
using *pure-state-target-vector-neq0* ‹$n \neq 0$›
by *auto*
finally show *?thesis*
by −
qed

definition ‹*pure-state'* F ψ η_F = F (*butterfly* ψ η_F) $*_V$ *pure-state-target-vector F* η_F›

abbreviation ‹*pure-state* F ψ ≡ *pure-state'* F ψ (*ket default*)›

nonterminal *pure-tensor*
syntax *-pure-tensor* :: ‹'a ⇒ 'b ⇒ *pure-tensor* ⇒ *pure-tensor*› (- - $\otimes_p$ - [*1000*, *0*, *0*] *1000*)
syntax *-pure-tensor2* :: ‹'a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ *pure-tensor*› (- - $\otimes_p$ - - [*1000*, *0*, *1000*, *0*] *1000*)
syntax *-pure-tensor1* :: ‹'a ⇒ 'b ⇒ *pure-tensor*›
syntax *-pure-tensor-start* :: ‹*pure-tensor* ⇒ 'a› ('(-'))

translations
-pure-tensor2 F ψ *G* φ ⇀ *CONST pure-state* (*F*; *G*) (ψ $\otimes_s$ φ)
-pure-tensor F ψ (*CONST pure-state G* φ) ⇀ *CONST pure-state* (*F*; *G*) (ψ $\otimes_s$ φ)
-pure-tensor-start x ⇀ *x*

-pure-tensor-start (*-pure-tensor2 F* ψ *G* φ) ↽ *CONST pure-state* (*F*; *G*) (ψ $\otimes_s$ φ)
-pure-tensor F ψ (*-pure-tensor2 G* φ *H* η) ↽ *-pure-tensor2 F* ψ (*G*;*H*) (φ $\otimes_s$ η)

term ‹(*F* ψ $\otimes_p$ *G* φ $\otimes_p$ *H z*)›
term ‹*pure-state* (*F*; *G*) (*a* $\otimes_s$ *b*)›

lemma *register-pair-butterfly-tensor*: ‹(*F*; *G*) (*butterfly* (*a* $\otimes_s$ *b*) (*c* $\otimes_s$ *d*)) = *F* (*butterfly a c*) o_{CL} *G* (*butterfly b d*)›
if [*simp*]: ‹*compatible F G*›
by (*auto simp*: *default-prod-def simp flip*: *tensor-ell2-ket tensor-butterfly register-pair-apply*)

lemma *pure-state-eqI*:

assumes ‹F (*selfbutter* η_F) = G (*selfbutter* η_G)›
assumes ‹F (*butterfly* ψ η_F) = G (*butterfly* φ η_G)›
shows ‹*pure-state'* F ψ η_F = *pure-state'* G φ η_G›
proof −
from *assms(1)* **have** ‹*pure-state-target-vector* F η_F = *pure-state-target-vector* G η_G›
by (*rule pure-state-target-vector-eqI*)
with *assms(2)*
show *?thesis*
unfolding *pure-state'-def*
by *simp*
qed

definition ‹*regular-register* F ⟷ *register* F ∧ (∃ *a*. (F; *complement* F) (*selfbutter* (*ket default*) ⊗$_o$ *a*) = *selfbutter* (*ket default*))›

lemma *regular-registerI*:
assumes [*simp*]: ‹*register* F›
assumes [*simp*]: ‹*complements* F G›
assumes *eq*: ‹(F; G) (*selfbutter* (*ket default*) ⊗$_o$ *a*) = *selfbutter* (*ket default*)›
shows ‹*regular-register* F›
proof −
have [*simp*]: ‹*compatible* F G›
using *assms* **by** (*simp add*: *complements-def*)
from ‹*complements* F G›
obtain I **where** *cFI*: ‹*complement* F *o* I = G› **and** ‹*iso-register* I›
apply *atomize-elim*
using *complement-unique'* *equivalent-registers-def* *equivalent-registers-sym* **by** *blast*
have ‹(F; *complement* F) (*selfbutter* (*ket default*) ⊗$_o$ I *a*) = (F; G) (*selfbutter* (*ket default*) ⊗$_o$ *a*)›
using *cFI* **by** (*auto simp*: *register-pair-apply*)
also have ‹... = *selfbutter* (*ket default*)›
by (*rule eq*)
finally show *?thesis*
unfolding *regular-register-def* **by** *auto*
qed

lemma *regular-register-pair*:
assumes [*simp*]: ‹*compatible* F G›
assumes ‹*regular-register* F› **and** ‹*regular-register* G›
shows ‹*regular-register* (F;G)›
proof −
have [*simp*]: ‹*bij* (F;*complement* F)› ‹*bij* (G;*complement* G)›
using *assms(1)* *compatible-def* *complement-is-complement* *complements-def* *iso-register-bij* **by** *blast+*
have *bij-FGcFG*[*simp*]: ‹*bij* ((F;G);*complement* (F;G))›
using *assms(1)* *complement-is-complement* *complements-def* *iso-register-bij* *pair-is-register* **by** *blast*
have [*simp*]: ‹*register* F› ‹*register* G›
using *assms(1)* **unfolding** *compatible-def* **by** *auto*

obtain *aF* **where** [*simp*]: ‹*inv* (F;*complement* F) (*selfbutter* (*ket default*)) = *selfbutter* (*ket default*) ⊗$_o$ *aF*›
by (*metis assms(2)* *compatible-complement-right* *invI* *pair-is-register* *register-inj* *regular-register-def*)
obtain *aG* **where** [*simp*]: ‹*inv* (G;*complement* G) (*selfbutter* (*ket default*)) = *selfbutter* (*ket default*) ⊗$_o$ *aG*›
by (*metis assms(3)* *complement-is-complement* *complements-def* *inj-iff* *inv-f-f* *iso-register-inv-comp1* *regular-register-def*)
define *t1* **where** ‹*t1* = *inv* ((F;G); *complement* (F;G)) (*selfbutter* (*ket default*))›
define *t2* **where** ‹*t2* = *inv* (F; (G; *complement* (F;G))) (*selfbutter* (*ket default*))›
define *t3* **where** ‹*t3* = *inv* (G; (F; *complement* (F;G))) (*selfbutter* (*ket default*))›

have ‹*complements* F (G; *complement* (F;G))›
apply (*rule complements-complement-pair*)
by *simp-all*
then have ‹*equivalent-registers* (*complement* F) (G; *complement* (F;G))›
using *complement-unique'* *equivalent-registers-sym* **by** *blast*

then obtain *I* **where** [*simp*]: ‹*iso-register I*› **and** *I*: ‹(*G*; *complement* (*F*;*G*)) = *complement F o I*›
by (*metis equivalent-registers-def*)
then have [*simp*]: ‹*register I*›
by (*meson iso-register-is-register*)
have [*simp*]: ‹*bij* (*id* ⊗*r I*)›
by (*rule iso-register-bij*, *simp*)
have [*simp*]: ‹*inv* (*id* ⊗*r I*) = *id* ⊗*r inv I*›
by *auto*

have ‹*t2* = (*inv* (*id* ⊗*r I*) *o inv* (*F*;*complement F*)) (*selfbutter* (*ket default*))›
unfolding *t2-def I*
apply (*subst o-inv-distrib*[*symmetric*])
by (*auto simp*: *pair-o-tensor*)
also have ‹. . . = (*selfbutter* (*ket default*) ⊗*o inv I aF*)›
apply *simp*
by (*metis* ‹*iso-register I*› *id-def iso-register-def iso-register-inv register-id register-tensor-apply*)
finally have *t2'*: ‹*t2* = *selfbutter* (*ket default*) ⊗*o inv I aF*›
by *simp*

have ∗: ‹*complements G* (*F*; *complement* (*F*;*G*))›
apply (*rule complements-complement-pair'*)
by *simp-all*
then have [*simp*]: ‹*compatible G* (*F*; *complement* (*F*;*G*))›
using *complements-def* **by** *blast*
from ∗ **have** ‹*equivalent-registers* (*complement G*) (*F*; *complement* (*F*;*G*))›
using *complement-unique' equivalent-registers-sym* **by** *blast*
then obtain *J* **where** [*simp*]: ‹*iso-register J*› **and** *I*: ‹(*F*; *complement* (*F*;*G*)) = *complement G o J*›
by (*metis equivalent-registers-def*)
then have [*simp*]: ‹*register J*›
by (*meson iso-register-is-register*)
have [*simp*]: ‹*bij* (*id* ⊗*r J*)›
by (*rule iso-register-bij*, *simp*)
have [*simp*]: ‹*inv* (*id* ⊗*r J*) = *id* ⊗*r inv J*›
by *auto*

have ‹*t3* = (*inv* (*id* ⊗*r J*) *o inv* (*G*;*complement G*)) (*selfbutter* (*ket default*))›
unfolding *t3-def I*
apply (*subst o-inv-distrib*[*symmetric*])
by (*auto simp*: *pair-o-tensor*)
also have ‹. . . = (*selfbutter* (*ket default*) ⊗*o inv J aG*)›
apply *simp*
by (*metis* ‹*iso-register J*› *id-def iso-register-def iso-register-inv register-id register-tensor-apply*)
finally have *t3'*: ‹*t3* = *selfbutter* (*ket default*) ⊗*o inv J aG*›
by *simp*

have ∗: ‹((*F*;*G*); *complement* (*F*;*G*)) *o assoc'* = (*F*; (*G*; *complement* (*F*;*G*)))›
apply (*rule tensor-extensionality3*)
by (*auto simp*: *register-pair-apply compatible-complement-pair1 compatible-complement-pair2*)
have *t2-t1*: ‹*t2* = *assoc t1*›
unfolding *t1-def t2-def* ∗[*symmetric*] **apply** (*subst o-inv-distrib*)
by *auto*

have ∗: ‹((*F*;*G*); *complement* (*F*;*G*)) *o* (*swap* ⊗*r id*) *o assoc'* = (*G*; (*F*; *complement* (*F*;*G*)))›
apply (*rule tensor-extensionality3*)
by (*auto intro*!: *register-comp register-tensor-is-register pair-is-register complements-complement-pair*
simp: *register-pair-apply compatible-complement-pair1*
lift-cblinfun-comp[*OF swap-registers-left*, *of F G*] *cblinfun-assoc-left*)
have *t3-t1*: ‹*t3* = *assoc* ((*swap* ⊗*r id*) *t1*)›
unfolding *t1-def t3-def* ∗[*symmetric*] **apply** (*subst o-inv-distrib*)
by (*auto intro*!: *bij-comp simp*: *iso-register-bij o-inv-distrib*)

from ‹*t2* = *assoc t1*› ‹*t3* = *assoc* ((*swap* ⊗*r id*) *t1*)›
have ∗: ‹*selfbutter* (*ket default*) ⊗*o inv J aG* = *assoc* ((*swap* ⊗*r id*) (*assoc'* (*selfbutter* (*ket default*) ⊗*o inv I*

```
aF)))›
    by (simp add: t2' t3')

  have ‹selfbutter (ket default) ⊗o swap (inv J aG) = (id ⊗r swap) (selfbutter (ket default) ⊗o inv J aG)›
    by auto
  also have ‹... = ((id ⊗r swap) o assoc o (swap ⊗r id) o assoc') (selfbutter (ket default) ⊗o inv I aF)›
    by (simp add: ∗)
  also have ‹... = (assoc o swap) (selfbutter (ket default) ⊗o inv I aF)›
    apply (rule fun-cong[where g=‹assoc o swap›])
    apply (intro tensor-extensionality3 register-comp register-tensor-is-register)
    by auto
  also have ‹... = assoc (inv I aF ⊗o selfbutter (ket default))›
    by auto
  finally have ∗: ‹selfbutter (ket default) ⊗o swap (inv J aG) = assoc (inv I aF ⊗o selfbutter (ket default))›
    by −

  obtain c where ∗: ‹butterfly (ket default :: 'c ell2) (ket default :: 'c ell2) ⊗o swap (inv J aG) = selfbutter
(ket default) ⊗o c ⊗o selfbutter (ket default)›
    apply atomize-elim
    apply (rule overlapping-tensor)
    using ∗ unfolding assoc-ell2-sandwich sandwich-apply
    by auto

  have ‹t1 = ((swap ⊗r id) o assoc') t3›
    by (simp add: t3-t1 register-tensor-distrib[unfolded o-def, THEN fun-cong] flip: id-def)
  also have ‹... = ((swap ⊗r id) o assoc' o (id ⊗r swap)) (butterfly (ket default :: 'c ell2) (ket default :: 'c
ell2) ⊗o swap (inv J aG))›
    unfolding t3' by auto
  also have ‹... = ((swap ⊗r id) o assoc' o (id ⊗r swap)) (selfbutter (ket default) ⊗o c ⊗o selfbutter (ket
default))›
    unfolding ∗ by simp
  also have ‹... = selfbutter (ket default) ⊗o c›
    apply simp
    by (simp add: default-prod-def tensor-butterfly tensor-ell2-ket)
  finally have ‹t1 = selfbutter (ket default) ⊗o c›
    by −

  then show ?thesis
    by (auto intro!: exI[of - c] simp: regular-register-def t1-def
        simp flip: bij-inv-eq-iff[OF bij-FGcFG])
qed

lemma regular-register-comp: ‹regular-register (F o G)› if ‹regular-register F› ‹regular-register G›
proof −
  have [simp]: ‹register F› ‹register G›
    using regular-register-def that by blast+
  from that obtain a where a: ‹(F; complement F) (selfbutter (ket default) ⊗o a) = selfbutter (ket default)›
    unfolding regular-register-def by metis
  from that obtain b where b: ‹(G; complement G) (selfbutter (ket default) ⊗o b) = selfbutter (ket default)›
    unfolding regular-register-def by metis
  have ‹complements (F o G) (complement F; F o complement G)›
    by (simp add: complements-chain)
  then have ‹equivalent-registers (complement F; F o complement G) (complement (F o G))›
    using complement-unique' by blast
  then obtain J where [simp]: ‹iso-register J› and 1: ‹(complement F; F o complement G) o J = (complement
(F o G))›
    using equivalent-registers-def by blast
  have [simp]: ‹register J›
    by (simp add: iso-register-is-register)

  define c where ‹c = inv J (a ⊗o b)›

  have ‹((F o G); complement (F o G)) (selfbutter (ket default) ⊗o c) = ((F o G); (complement F; F o
```

complement G)) (*selfbutter* (*ket default*) ⊗*o J c*)›
by (*auto simp flip*: *1 simp*: *register-pair-apply*)
also have ‹... = ((*F o* (*G*; *complement G*); *complement F*) *o assoc′ o* (*id* ⊗*r swap*)) (*selfbutter* (*ket default*) ⊗*o J c*)›
apply (*subst register-comp-pair*[*symmetric*])
apply *auto*[*2*]
apply (*subst pair-o-assoc′*)
apply *auto*[*3*]
apply (*subst pair-o-tensor*)
by *auto*
also have ‹... = ((*F o* (*G*; *complement G*); *complement F*) *o assoc′*) (*selfbutter* (*ket default*) ⊗*o swap* (*J c*))›
by *auto*
also have ‹... = ((*F o* (*G*; *complement G*); *complement F*) *o assoc′*) (*selfbutter* (*ket default*) ⊗*o* (*b* ⊗*o a*))›
unfolding *c-def* **apply** (*subst surj-f-inv-f*[**where** *f*=*J*])
apply (*meson* ‹*iso-register J*› *bij-betw-inv-into-right iso-register-inv-comp1 iso-register-inv-comp2 iso-tuple-UNIV-I o-bij surj-iff-all*)
by *auto*
also have ‹... = (*F* ∘ (*G*;*complement G*);*complement F*) ((*selfbutter* (*ket default*) ⊗*o b*) ⊗*o a*)›
by (*simp add*: *assoc′-apply*)
also have ‹... = (*F*; *complement F*) ((*G*;*complement G*) (*selfbutter* (*ket default*) ⊗*o b*) ⊗*o a*)›
by (*simp add*: *register-pair-apply′*)
also have ‹... = *selfbutter* (*ket default*)›
by (*auto simp*: *a b*)
finally have ‹(*F* ∘ *G*;*complement* (*F* ∘ *G*)) (*selfbutter* (*ket default*) ⊗*o c*) = *selfbutter* (*ket default*)›
by −
then show *?thesis*
using ‹*register F*› ‹*register G*› *register-comp regular-register-def* **by** *blast*
qed

lemma *regular-iso-register*:
assumes ‹*regular-register F*›
assumes [*register*]: ‹*iso-register F*›
shows ‹*F* (*selfbutter* (*ket default*)) = *selfbutter* (*ket default*)›
proof −
from *assms*(*1*) **obtain** *a* **where** *a*: ‹(*F*;*complement F*) (*selfbutter* (*ket default*) ⊗*o a*) = *selfbutter* (*ket default*)›
using *regular-register-def* **by** *blast*

let *?u* = ‹*empty-var* :: (*unit ell2* ⇒*CL unit ell2*) ⇒ -›
have ‹*is-unit-register ?u*› **and** ‹*is-unit-register* (*complement F*)›
by *auto*
then have ‹*equivalent-registers* (*complement F*) *?u*›
using *unit-register-unique* **by** *blast*
then obtain *I* **where** ‹*iso-register I*› **and** ‹*complement F* = *?u o I*›
by (*metis* ‹*is-unit-register* (*complement F*)› *equivalent-registers-def is-unit-register-empty-var unit-register-unique*)
have ‹*selfbutter* (*ket default*) = (*F*; *?u o I*) (*selfbutter* (*ket default*) ⊗*o a*)›
using ‹*complement F* = *empty-var* ∘ *I*› *a* **by** *presburger*
also have ‹... = (*F*; *?u*) (*selfbutter* (*ket default*) ⊗*o I a*)›
by (*metis Laws-Quantum.register-pair-apply* ‹*complement F* = *empty-var* ∘ *I*› ‹*equivalent-registers* (*complement F*) *empty-var*› *assms*(*2*) *comp-apply complement-is-complement complements-def equivalent-complements iso-register-is-register*)
also have ‹... = (*F*; *?u*) (*selfbutter* (*ket default*) ⊗*o* (*one-dim-iso* (*I a*) ∗*C id-cblinfun*))›
by *simp*
also have ‹... = *one-dim-iso* (*I a*) ∗*C* (*F*; *?u*) (*selfbutter* (*ket default*) ⊗*o id-cblinfun*)›
by (*simp add*: *Axioms-Quantum.register-pair-apply empty-var-def iso-register-is-register*)
also have ‹... = *one-dim-iso* (*I a*) ∗*C F* (*selfbutter* (*ket default*))›
by (*auto simp*: *register-pair-apply iso-register-is-register simp del*: *id-cblinfun-eq-1*)
finally have *F*: ‹*one-dim-iso* (*I a*) ∗*C F* (*selfbutter* (*ket default*)) = *selfbutter* (*ket default*)›
by *simp*

from *F* **have** ‹*one-dim-iso* (*I a*) ≠ (*0*::*complex*)›
by (*metis butterfly-apply butterfly-scaleC-left complex-vector.scale-eq-0-iff id-cblinfun-eq-1 id-cblinfun-not-0 cinner-ket-same ket-nonzero one-dim-iso-of-one one-dim-iso-of-zero′*)

have ‹*selfbutter* (*ket default*) = *one-dim-iso* (*I a*) ∗*C F* (*selfbutter* (*ket default*))›

```
    using F by simp
  also have ‹... = one-dim-iso (I a) *C F (selfbutter (ket default) oCL selfbutter (ket default))›
    by auto
  also have ‹... = one-dim-iso (I a) *C (F (selfbutter (ket default)) oCL F (selfbutter (ket default)))›
    by (simp add: assms(2) iso-register-is-register register-mult)
  also have ‹... = one-dim-iso (I a) *C ((selfbutter (ket default) /C one-dim-iso (I a)) oCL (selfbutter (ket default) /C one-dim-iso (I a)))›
    by (metis (no-types, lifting) F ‹one-dim-iso (I a) ≠ 0› complex-vector.scale-left-imp-eq inverse-1 left-inverse scaleC-scaleC zero-neq-one)
  also have ‹... = one-dim-iso (I a) *C selfbutter (ket default)›
   by (smt (verit, best) butterfly-comp-butterfly calculation cblinfun-compose-scaleC-left cblinfun-compose-scaleC-right complex-vector.scale-cancel-left cinner-ket-same left-inverse scaleC-one scaleC-scaleC)
  finally have ‹one-dim-iso (I a) = (1::complex)›
   by (metis butterfly-0-left butterfly-apply complex-vector.scale-cancel-right cinner-ket-same ket-nonzero scaleC-one)
  with F show ‹F (selfbutter (ket default)) = selfbutter (ket default)›
    by simp
qed

lemma pure-state-nested:
  assumes [simp]: ‹compatible F G›
  assumes ‹regular-register H›
  assumes ‹iso-register H›
  shows ‹pure-state (F;G) (pure-state H h ⊗s g) = pure-state ((F o H);G) (h ⊗s g)›
proof −
  note [[simproc del: Laws-Quantum.compatibility-warn]]
  have [simp]: ‹register H›
    by (meson assms(3) iso-register-is-register)
  have [simp]: ‹H (selfbutter (ket default)) = selfbutter (ket default)›
    apply (rule regular-iso-register)
    using assms by auto
  have 1: ‹pure-state-target-vector H (ket default) = ket default›
    apply (rule pure-state-target-vector-ket-default)
    apply simp
    by (metis (no-types, lifting) cinner-ket-same rangeI scaleC-one)

  have ‹butterfly (pure-state H h) (ket default) = butterfly (H (butterfly h (ket default)) *V ket default) (ket default)›
    by (simp add: pure-state'-def 1)
  also have ‹... = H (butterfly h (ket default)) oCL selfbutter (ket default)›
   by (metis (no-types, opaque-lifting) adj-cblinfun-compose butterfly-adjoint butterfly-comp-cblinfun double-adj)
  also have ‹... = H (butterfly h (ket default)) oCL H (selfbutter (ket default))›
    by simp
  also have ‹... = H (butterfly h (ket default) oCL selfbutter (ket default))›
    by (meson ‹register H› register-mult)
  also have ‹... = H (butterfly h (ket default))›
    by auto
  finally have 2: ‹butterfly (pure-state H h) (ket default) = H (butterfly h (ket default))›
    by simp

  show ?thesis
    apply (rule pure-state-eqI)
    using 1 2
    by (auto simp: register-pair-butterfly-tensor compatible-ac-rules default-prod-def simp flip: tensor-ell2-ket)
qed

lemma state-apply1:
  assumes [register]: ‹compatible F G›
  shows ‹F U *V (F ψ ⊗p G φ) = (F (U ψ) ⊗p G φ)›
proof −
  have [register]: ‹compatible F G›
    using assms(1) complements-def by blast
  have ‹F U *V (F ψ ⊗p G φ) = (F;G) (U ⊗o id-cblinfun) *V (F ψ ⊗p G φ)›
    apply (subst register-pair-apply)
```

```
    by auto
  also have ‹... = (F (U ψ) ⊗p G φ)›
    unfolding pure-state'-def
    by (auto simp: register-mult' cblinfun-comp-butterfly tensor-op-ell2)
  finally show ?thesis
    by −
qed

lemma Fst-regular[simp]: ‹regular-register Fst›
  apply (rule regular-registerI[where a=‹selfbutter (ket default)› and G=Snd])
  by (auto simp: pair-Fst-Snd default-prod-def tensor-ell2-ket tensor-butterfly)

lemma Snd-regular[simp]: ‹regular-register Snd›
  apply (rule regular-registerI[where a=‹selfbutter (ket default)› and G=Fst])
    apply auto[2]
  apply (simp only: default-prod-def swap-apply flip: swap-def tensor-ell2-ket)
  by (auto simp: tensor-butterfly)

lemma id-regular[simp]: ‹regular-register id›
  apply (rule regular-registerI[where G=unit-register and a=id-cblinfun])
  by (auto simp: register-pair-apply)

lemma swap-regular[simp]: ‹regular-register swap›
  by (auto intro!: regular-register-pair simp: swap-def)

lemma assoc-regular[simp]: ‹regular-register assoc›
  by (auto intro!: regular-register-pair regular-register-comp simp: assoc-def)

lemma assoc'-regular[simp]: ‹regular-register assoc'›
  by (auto intro!: regular-register-pair regular-register-comp simp: assoc'-def)

lemma cspan-pure-state':
  assumes ‹iso-register F›
  assumes ‹cspan (g ‘ X) = UNIV›
  assumes η-cond: ‹F (selfbutter η) *V pure-state-target-vector F η ≠ 0›
  shows ‹cspan ((λz. pure-state' F (g z) η) ‘ X) = UNIV›
proof −
  from iso-register-decomposition[of F]
  obtain U where [simp]: ‹unitary U› and F: ‹F = sandwich U›
    using assms(1) by blast

  define η' c where ‹η' = pure-state-target-vector F η› and ‹c = cinner (U *V η) η'›

  from η-cond
  have ‹c ≠ 0›
    by (simp add: η'-def F sandwich-apply c-def cinner-adj-right cblinfun.scaleC-right)

  have ‹cspan ((λz. pure-state' F (g z) η) ‘ X) = cspan ((λz. F (butterfly (g z) η) *V η') ‘ X)›
    by (simp add: η'-def pure-state'-def)
  also have ‹... = cspan ((λz. (butterfly (U *V g z) (U *V η)) *V η') ‘ X)›
    by (simp add: F sandwich-apply cinner-adj-right cblinfun.scaleC-right)
  also have ‹... = cspan ((λz. c *C U *V g z) ‘ X)›
    by (simp add: c-def)
  also have ‹... = (λz. c *C U *V z) ‘ cspan (g ‘ X)›
    apply (subst complex-vector.linear-span-image[symmetric])
    by (auto simp: image-image)
  also have ‹... = (λz. c *C U *V z) ‘ UNIV›
    using assms(2) by presburger
  also have ‹... = UNIV›
    apply (rule surjI[where f=‹λz. (U* *V z) /C c›])
    using ‹c ≠ 0› by (auto simp flip: cblinfun-apply-cblinfun-compose cblinfun.scaleC-right)
  finally show ?thesis
    by −
```

qed

lemma *cspan-pure-state*:
assumes [*simp*]: ‹*iso-register F*›
assumes ‹*cspan* (*g* ‘ *X*) = *UNIV*›
shows ‹*cspan* ((λ*z*. *pure-state F* (*g z*)) ‘ *X*) = *UNIV*›
apply (*rule cspan-pure-state′*)
using *assms* **apply** *auto*[*2*]
apply (*rule pure-state-target-vector-correct*)
by (*auto simp*: *iso-register-is-register*)

lemma *pure-state-bounded-clinear*:
assumes [*register*]: ‹*compatible F G*›
shows ‹*bounded-clinear* (λψ. (*F* ψ ⊗*p* *G* φ))›
proof −
have [*bounded-clinear*]: ‹*bounded-clinear* (*F*;*G*)›
using *assms pair-is-register register-bounded-clinear* **by** *blast*
show *?thesis*
unfolding *pure-state′-def*
by (*auto intro*!: *bounded-linear-intros*)
qed

lemma *pure-state-bounded-clinear-right*:
assumes [*register*]: ‹*compatible F G*›
shows ‹*bounded-clinear* (λφ. (*F* ψ ⊗*p* *G* φ))›
proof −
have [*bounded-clinear*]: ‹*bounded-clinear* (*F*;*G*)›
using *assms pair-is-register register-bounded-clinear* **by** *blast*
show *?thesis*
unfolding *pure-state′-def*
by (*auto intro*!: *bounded-linear-intros*)
qed

lemma *pure-state-clinear*:
assumes [*register*]: ‹*compatible F G*›
shows ‹*clinear* (λψ. (*F* ψ ⊗*p* *G* φ))›
using *assms bounded-clinear.clinear pure-state-bounded-clinear* **by** *blast*

method *pure-state-flatten-nested* =
(*subst pure-state-nested*, (*auto*; *fail*)[*3*])+

The following method *pure-state-eq* tries to solve a equality where both sides are of the form $F_1(\psi_1) \otimes_p F_2(\psi_2) \otimes_p \ldots \otimes_p F_n(\psi_n)$ by reordering the registers and unfolding nested register pairs. (For the unfolding of nested pairs, it is necessary that the corresponding *compatible F G* facts are provable by the simplifier.)

If the some of the pure states ψ_i themselves are $\otimes_p$-tensors, they will be flattened if possible. (If all necessary conditions can be proven, such as *regular-register* etc.)

The method may either succeed, fail, or reduce the equality to a hopefully simpler one.

method *pure-state-eq* =
(*pure-state-flatten-nested?*,
rule pure-state-eqI;
auto simp: *register-pair-butterfly-tensor compatible-ac-rules default-prod-def*
simp flip: *tensor-ell2-ket*)

lemma *example*:
fixes *F* :: ‹*bit update* ⇒ *′c*::{*finite*,*default*} *update*›
and *G* :: ‹*bit update* ⇒ *′c update*›
assumes [*register*]: ‹*compatible F G*›
shows ‹(*F*;*G*) *CNOT* ∘*CL* (*G*;*F*) *CNOT* ∘*CL* (*F*;*G*) *CNOT* = (*F*;*G*) *swap-ell2*›
proof −
define *Z* **where** ‹*Z* = *complement* (*F*;*G*)›
then have [*register*]: ‹*compatible Z F*› ‹*compatible Z G*›

using *assms compatible-complement-pair1 compatible-complement-pair2 compatible-sym* **by** *blast+*

have [*simp*]: ‹*iso-register* (*F*;(*G*;*Z*))›
using *Z-def assms complement-is-complement complements-complement-pair complements-def pair-is-register*
by *blast*

have *eq1*: ‹((*F*;*G*) *CNOT* o_{CL} (*G*;*F*) *CNOT* o_{CL} (*F*;*G*) *CNOT*) $*_V$ (*F*(*ket f*) $\otimes_p$ *G*(*ket g*) $\otimes_p$ *Z*(*ket z*))
= (*F*;*G*) *swap-ell2* $*_V$ (*F*(*ket f*) $\otimes_p$ *G*(*ket g*) $\otimes_p$ *Z*(*ket z*))› **for** *f g z*
proof −
have ‹(*F*(*ket f*) $\otimes_p$ *G*(*ket g*) $\otimes_p$ *Z*(*ket z*)) = ((*F*;*G*) (*ket f* $\otimes_s$ *ket g*) $\otimes_p$ *Z*(*ket z*))›
by *pure-state-eq*
also have ‹(*F*;*G*) *CNOT* $*_V$ ... = ((*F*;*G*) (*ket f* $\otimes_s$ *ket* (*g*+*f*)) $\otimes_p$ *Z*(*ket z*))›
apply (*subst state-apply1*) **by** (*auto simp*: *tensor-ell2-ket*)
also have ‹... = ((*G*;*F*) (*ket* (*g*+*f*) $\otimes_s$ *ket f*) $\otimes_p$ *Z*(*ket z*))›
by *pure-state-eq*
also have ‹(*G*;*F*) *CNOT* $*_V$ ... = ((*G*;*F*) (*ket* (*g*+*f*) $\otimes_s$ *ket g*) $\otimes_p$ *Z ket z*)›
apply (*subst state-apply1*) **by** (*auto simp*: *tensor-ell2-ket*)
also have ‹... = ((*F*;*G*) (*ket g* $\otimes_s$ *ket* (*g*+*f*)) $\otimes_p$ *Z ket z*)›
by *pure-state-eq*
also have ‹(*F*;*G*) *CNOT* $*_V$ ... = ((*F*;*G*) *ket g* $\otimes_s$ *ket f* $\otimes_p$ *Z ket z*)›
apply (*subst state-apply1*)
apply *simp*
by (*metis add-diff-cancel-left′ cnot-apply minus-bit-def tensor-ell2-ket*)
also have ‹... = (*F*(*ket g*) $\otimes_p$ *G*(*ket f*) $\otimes_p$ *Z*(*ket z*))›
by *pure-state-eq*
finally have *1*: ‹((*F*;*G*) *CNOT* o_{CL} (*G*;*F*) *CNOT* o_{CL} (*F*;*G*) *CNOT*) $*_V$ (*F*(*ket f*) $\otimes_p$ *G*(*ket g*) $\otimes_p$ *Z*(*ket z*)) = (*F*(*ket g*) $\otimes_p$ *G*(*ket f*) $\otimes_p$ *Z*(*ket z*))›
by *auto*

have ‹(*F*(*ket f*) $\otimes_p$ *G*(*ket g*) $\otimes_p$ *Z*(*ket z*)) = ((*F*;*G*) (*ket f* $\otimes_s$ *ket g*) $\otimes_p$ *Z*(*ket z*))›
by *pure-state-eq*
also have ‹(*F*;*G*) *swap-ell2* $*_V$ ... = ((*F*;*G*) (*ket g* $\otimes_s$ *ket f*) $\otimes_p$ *Z*(*ket z*))›
by (*auto simp*: *state-apply1 swap-ell2-tensor simp del*: *tensor-ell2-ket*)
also have ‹... = (*F*(*ket g*) $\otimes_p$ *G*(*ket f*) $\otimes_p$ *Z*(*ket z*))›
by *pure-state-eq*
finally have *2*: ‹(*F*;*G*) *swap-ell2* $*_V$ (*F*(*ket f*) $\otimes_p$ *G*(*ket g*) $\otimes_p$ *Z*(*ket z*)) = (*F*(*ket g*) $\otimes_p$ *G*(*ket f*) $\otimes_p$ *Z*(*ket z*))›
by −

from *1 2* **show** *?thesis*
by *simp*
qed

then have *eq1*: ‹((*F*;*G*) *CNOT* o_{CL} (*G*;*F*) *CNOT* o_{CL} (*F*;*G*) *CNOT*) $*_V$ ψ
= (*F*;*G*) *swap-ell2* $*_V$ ψ› **if** ‹$\psi \in \{(F(ket\ f) \otimes_p G(ket\ g) \otimes_p Z(ket\ z))|\ f\ g\ z.\ True\}$› **for** ψ
using *that* **by** *auto*

moreover have ‹*cspan* $\{(F(ket\ f) \otimes_p G(ket\ g) \otimes_p Z(ket\ z))|\ f\ g\ z.\ True\} = UNIV$›
apply (*simp only*: *double-exists setcompr-eq-image full-SetCompr-eq*)
apply *simp*
apply (*rule cspan-pure-state*)
by (*auto simp*: *tensor-ell2-ket*)

ultimately show *?thesis*
using *cblinfun-eq-on-UNIV-span* **by** *blast*
qed

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

end

theory *Check-Autogenerated-Files*

imports *Laws-Classical Laws-Quantum Laws-Complement-Quantum*
begin

ML ‹
let
fun check kind file expected = let
val content = File.read (Path.append (Resources.master-directory ***theory****) (Path.basic file))*
val hash = SHA1.digest content |> SHA1.rep
in
if hash = expected then () else
error (kind ^ file ^file ^ has changed.\nPlease run \python3 instantiate-laws.py\ to recreated autogenerated files.\nExpected SHA1 hash ^ expected ^, got ^ hash)
end
in
check Source Axioms-Classical.thy e3c6ca3fa815506631517ea5d682c23c08f4ed4f;
check Source Axioms-Quantum.thy eaa50706b1a7d09ca9ce4530c86f81e501937589;
check Source Laws.thy 65ab3f7e71975cd14617f501b65a6acadceb1ef3;
check Source Laws-Complement.thy 70ce14ff0247460eb56e09b8a5ed7b2715abebe0;
check Generated Laws-Classical.thy 8a50d01926a3dabda71b8992451bf4acfb420654;
check Generated Laws-Complement-Quantum.thy fdc4022c54b9a5f2a95de15c6c391727e1b3bcc1;
check Generated Laws-Quantum.thy 38a1ec5c464069098cb5855ace1d69fc49aea29b
end
›

end

References

[Daw21] Matthew Daws. Answer to mathoverlow question “unital *-homomorphisms between matrices”. https://mathoverflow.net/a/390180, 2021. Last accessed 2021-10-12.

[Unr24] Dominique Unruh. Quantum references. arXiv:2105.10914 [cs.LO], 2021–2024.