Quantum and Classical Registers

Dominique Unruh

January 21, 2022

Abstract

A formalization of the theory of quantum and classical registers as developed by Unruh [Unr21]. 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 [Unr21]. 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 [Unr21]. [Unr21] 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 [Unr21] for an explanation of this mechanism and the reasons for it.

Contents

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** ⊗$_u$ *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* (λ*b*::'*b*::*domain update*. *tensor-update a b*)›
  for *a* :: ‹'*a*::*domain update*›
proof −
  have ‹*preregister* ((λ*b1*::('*a*×'*b*) *update*. (*a* ⊗*u* *id-update*) ∗*u* *b1*) *o* (λ*b*. *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* (λ*a*::'*a*::*domain update*. *tensor-update a b*)›
  for *b* :: ‹'*b*::*domain update*›
proof −
  have ‹*preregister* ((λ*a1*::('*a*×'*b*) *update*. (*id-update* ⊗*u* *b*) ∗*u* *a1*) *o* (λ*a*. *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* (λ*a*::'*a*::*domain update*. *id-update* ⊗*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*::*domain update*. *F a* ⊗*u* *id-update*)›
  using *assms* **apply** (*rule register-comp*[*unfolded o-def*])
  by *simp*

2.4 Tensor product of registers

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

lemma *register-tensor-is-register*:
  fixes *F* :: '*a*::*domain update* ⇒ '*b*::*domain update* **and** *G* :: '*c*::*domain update* ⇒ '*d*::*domain 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*::*domain update* ⇒ '*b*::*domain update* **and** *G* :: '*c*::*domain update* ⇒ '*d*::*domain 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*::*domain itself*) *A* ⟷
(∀ *F G* :: '*a*::*domain 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*::*domain*) *A* ⟹ *separating TYPE*('*a*) *B*›
unfolding *separating-def* **by** (*meson in-mono*)

lemma *register-eqI*: ‹*separating TYPE*('*b*::*domain*) *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*::*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* ⊗*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*::*domain update set*› **and** *B* :: ‹'*b*::*domain update set*›
assumes ‹*separating TYPE*('*c*::*domain*) *A*›

assumes ‹*separating TYPE('c) B*›
assumes ‹$C = \{a \otimes_u b \mid a\ b.\ a{\in}A \wedge b{\in}B\}$›
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 $\bigwedge f\ g\ h.\ F\ (f \otimes_u g \otimes_u h) = G\ (f \otimes_u g \otimes_u h)$
shows $F = G$
proof (*rule register-eqI*[**where** A=‹$\{a \otimes_u b \otimes_u c \mid a\ b\ c.\ True\}$›])
have ‹*separating TYPE('d)* $\{b \otimes_u c \mid b\ c.\ True\}$›
apply (*rule separating-tensor'*[**where** A=*UNIV* **and** B=*UNIV*])
by *auto*
then show ‹*separating TYPE('d)* $\{a \otimes_u b \otimes_u c \mid a\ b\ c.\ True\}$›
apply (*rule-tac separating-tensor'*[**where** A=*UNIV* **and** B=‹$\{b \otimes_u c \mid b\ c.\ True\}$›])
by *auto*
show ‹*preregister F*› ‹*preregister G*› **by** *auto*
show ‹$x \in \{a \otimes_u b \otimes_u c \mid a\ b\ c.\ True\} \Longrightarrow F\ x = G\ x$› **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 $\bigwedge f\ g\ h.\ F\ ((f \otimes_u g) \otimes_u h) = G\ ((f \otimes_u g) \otimes_u h)$
shows $F = G$
proof (*rule register-eqI*[**where** A=‹$\{(a \otimes_u b) \otimes_u c \mid a\ b\ c.\ True\}$›])
have ‹*separating TYPE('d)* $\{a \otimes_u b \mid a\ b.\ True\}$›
apply (*rule separating-tensor'*[**where** A=*UNIV* **and** B=*UNIV*])
by *auto*
then show ‹*separating TYPE('d)* $\{(a \otimes_u b) \otimes_u c \mid a\ b\ c.\ True\}$›
apply (*rule-tac separating-tensor'*[**where** B=*UNIV* **and** A=‹$\{a \otimes_u b \mid a\ b.\ True\}$›])
by *auto*
show ‹*preregister F*› ‹*preregister G*› **by** *auto*
show ‹$x \in \{(a \otimes_u b) \otimes_u c \mid a\ b\ c.\ True\} \Longrightarrow F\ x = G\ x$› **for** *x*
using *assms*(*3*) **by** *auto*
qed

lemma *register-tensor-id*[*simp*]: ‹$id \otimes_r id = id$›
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* ⟹ *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*::*domain update* ⇒ *′c*::*domain update* **and** *G* :: *′b*::*domain update* ⇒ *′c*::*domain update*
assumes *compatible F G*
shows *compatible* (λ*a*. *id-update* ⊗*u* *F a*) (λ*a*. *id-update* ⊗*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* ⇒ *′c*::*domain update* **and** *G* :: *′b*::*domain update* ⇒ *′c*::*domain update*
assumes *compatible F G*
shows *compatible* (λ*a*. *F a* ⊗*u* *id-update*) (λ*a*. *G a* ⊗*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* ($\lambda a.$ *id-update* $\otimes_u$ *F a*) ($\lambda a.$ *G a* $\otimes_u$ *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* $\otimes_u$ *b*) = (*F* ∘ (*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*

2.6 Fst and Snd

definition *Fst* **where** ‹*Fst a* = *a* $\otimes_u$ *id-update*›
definition *Snd* **where** ‹*Snd a* = *id-update* $\otimes_u$ *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* $\otimes_u$ *b*) = (*b* $\otimes_u$ *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*
unfolding Φ-*def* **apply** (*auto simp*: *register-pair-apply*)
by (*metis assms*(*1*) *assms*(*2*) *compatible-def 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 [*simp*]: ‹*bij assoc*›
using *assoc'-o-assoc assoc-o-assoc' o-bij* **by** *blast*

lemma [*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*)
show *?thesis*
apply (*rule iso-registerI*[**where** *G*=‹*G′ o F′*›])
apply (*auto simp*: *register-tensor-is-register iso-register-is-register register-tensor-distrib*)
apply (*metis* ‹*F* ∘ *F′* = *id*› ‹*G* ∘ *G′* = *id*› *fcomp-assoc fcomp-comp id-fcomp*)
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*)
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-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)
```

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-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 ⊗u 70)

bundle register-notation begin
notation register-tensor (infixr ⊗r 70)
notation register-pair ('(-;-'))
end

bundle no-register-notation begin
no-notation register-tensor (infixr ⊗r 70)
no-notation register-pair ('(-;-'))
end

end
```

3 Axioms of complements

```
theory Axioms-Complement
  imports Laws
begin

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

axiomatization where
  complement-exists: ‹register F ⟹ ∃ G :: ('a, 'b) complement-domain 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

notation *comp-update* (**infixl** $*_u$ *55*)
notation *tensor-update* (**infixr** $\otimes_u$ *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 *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 update*) ⇒ (('*a*,'*b*) *complement-domain update* ⇒ '*b update*)› **where**
‹*complement F* = (*SOME G* :: ('*a*, '*b*) *complement-domain update* ⇒ '*b update*. *compatible F G* ∧ *iso-register* (*F*;*G*))›

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

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

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 *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* ...›
apply (*auto intro*!: *iso-register-comp iso-register-tensor-is-iso-register*)
using *assms complement-is-complement complements-def* **by** *blast+*
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 −
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*; *complement X*))›
apply (*rule equivalent-registers-pair-right*)
apply (*auto intro*!: *unit-register-compatible*)
using *complement-is-complement complements-def equivalent-registersI id-comp register-id that*(*2*) **by** *blast*
also have ‹*equivalent-registers* ... ((*U*; *X*); *complement X*)›
apply (*rule equivalent-registers-assoc*)
by *auto*

finally have ‹*complements* (*U*; *X*) (*complement X*)›
by (*auto simp*: *equivalent-registers-def complements-def*)
moreover have ‹*equivalent-registers* (*X*; *complement X*) *id*›
by (*metis complement-is-complement complements-def equivalent-registers-def iso-register-def that*)
ultimately show *?thesis*
by (*meson complement-unique complement-is-complement complements-sym equivalent-registers-sym equivalent-registers-trans that*)
qed

lemma *unit-register-compose-left*:
assumes [*simp*]: ‹*is-unit-register U*›
assumes [*simp*]: ‹*register A*›
shows ‹*is-unit-register* (*A o U*)›
proof −
have ‹*compatible* (*A o U*) (*A*; *complement A*)›
apply (*auto intro*!: *compatible3′*)
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*) *id*›
by (*metis assms*(*2*) *compatible-comp-right complement-is-complement complements-def iso-register-def*)
have ‹*equivalent-registers* (*A o U*; *id*) (*A o U*; (*A*; *complement A*))›
apply (*auto intro*!: *equivalent-registers-pair-right*)
using *assms*(*2*) *complement-is-complement complements-def equivalent-registers-def id-comp register-id* **by** *blast*
also have ‹*equivalent-registers* ... ((*A o U*; *A o id*); *complement A*)›
apply *auto*
by (*metis* (*no-types*, *opaque-lifting*) *compat assms*(*1*) *assms*(*2*) *compatible-comp-left compatible-def compatible-register1 complement-is-complement complements-def equivalent-registers-assoc id-apply register-unit-register*)
also have ‹*equivalent-registers* ... (*A o* (*U*; *id*); *complement A*)›
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*; *complement A*)›
apply (*intro equivalent-registers-pair-left equivalent-registers-comp*)
apply (*auto simp*: *assms*)
using *assms*(*1*) *equivalent-registers-sym register-id unit-register-pair* **by** *blast*
also have ‹*equivalent-registers* ... *id*›
by (*metis assms*(*2*) *comp-id complement-is-complement complements-def equivalent-registers-def iso-register-inv iso-register-inv-comp2 pair-is-register*)
finally show *?thesis*
using *compat complementsI equivalent-registers-sym is-unit-register-def iso-register-equivalent-id* **by** *blast*
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-registe*
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›*
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 −
let *?U0* = ‹*complement id* :: *unit-register-domain update* ⇒ *some-domain update›*
let *?U1* = ‹*complement id* :: (*'a*, *'a*) *complement-domain update* ⇒ *'a update›*
have ‹*is-unit-register ?U0*› ‹*is-unit-register ?U1›*
by *auto*
then obtain *I* :: ‹*unit-register-domain update* ⇒ (*'a*, *'a*) *complement-domain update›* **where** ‹*iso-register I›*
apply *atomize-elim* **by** (*rule unit-register-domains-isomorphic*)
with ‹*is-unit-register ?U1›* **have** ‹*is-unit-register* (*?U1 o I*)›
by (*rule unit-register-compose-right*)
then show *?thesis*
by (*metis someI-ex unit-register-def*)
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 −
have ‹*equivalent-registers* (*id* :: *'b update* ⇒ -) (*complement id*; *id*)›
using *id-complement-is-unit-register iso-register-equivalent-id register-id unit-register-pair* **by** *blast*
then obtain *J* :: ‹*'b update* ⇒ (((*'b*, *'b*) *complement-domain* ∗ *'b*) *update*)› **where** ‹*iso-register J›*
using *equivalent-registers-def iso-register-inv* **by** *blast*
moreover obtain *K* :: ‹(*'b*, *'b*) *complement-domain update* ⇒ *'a update›* **where** ‹*iso-register K›*
using *assms id-complement-is-unit-register unit-register-domains-isomorphic* **by** *blast*
ultimately have ‹*iso-register* ((*K* ⊗*r id*) *o J*)›
by *auto*
then show *?thesis*
by *auto*

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*›
shows ‹*complements F* (*G*; *complement* (*F*;*G*))›
proof (*rule complementsI*)
have ‹*equivalent-registers* (*F*; (*G*; *complement* (*F*;*G*))) ((*F*;*G*); *complement* (*F*;*G*))›
apply (*rule equivalent-registers-assoc*)
by (*auto simp add*: *compatible-complement-pair1 compatible-complement-pair2*)
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*;*complement* (*F*;*G*)))›
using *equivalent-registers-sym iso-register-equivalent-id* **by** *blast*
show ‹*compatible F* (*G*;*complement* (*F*;*G*))›
using *assms compatible3′ compatible-complement-pair1 compatible-complement-pair2* **by** *blast*
qed

lemma *equivalent-registers-complement*:
assumes ‹*equivalent-registers F G*›
shows ‹*equivalent-registers* (*complement F*) (*complement G*)›
proof −
have ‹*complements F* (*complement F*)›
using *assms complement-is-complement equivalent-registers-register-left* **by** *blast*
with *assms* **have** ‹*complements G* (*complement F*)›
by (*meson complements-sym equivalent-complements*)
then show *?thesis*
by (*rule complement-unique*)
qed

lemma *complements-complement-pair′*:
assumes [*simp*]: ‹*compatible F G*›
shows ‹*complements G* (*F*; *complement* (*F*;*G*))›
proof −
have ‹*equivalent-registers* (*F*;*G*) (*G*;*F*)›
apply (*rule equivalent-registersI*[**where** *I*=*swap*])
by *auto*

then have ‹*equivalent-registers* (*complement* (*F*;*G*)) (*complement* (*G*;*F*))›
by (*rule equivalent-registers-complement*)
then have ‹*equivalent-registers* (*F*; (*complement* (*F*;*G*))) (*F*; (*complement* (*G*;*F*)))›
apply (*rule equivalent-registers-pair-right*[*rotated*])
using *assms compatible-complement-pair1* **by** *blast*
moreover have ‹*complements G* (*F*; *complement* (*G*;*F*))›
apply (*rule complements-complement-pair*)
using *assms compatible-sym* **by** *blast*
ultimately show *?thesis*
by (*meson equivalent-complements equivalent-registers-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*)

end

5 Classical instantiation of registerss

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* ∘ *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* ⇒ *None* | *Some x* ⇒ *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* ⇒ *None* | *Some x* ⇒ *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* ⇒ *None* | *Some x* ⇒ *sG x m*)›
and ‹*g m* = *gF* (*gG m*)› **for** *a m*
have ‹(*G* ∘ *F*) *a m* = (*case a* (*g m*) *of None* ⇒ *None* | *Some x* ⇒ *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* ∘ *F*)
using *preregister-def* **by** *blast*
qed

definition *tensor-update* :: ‹*'a update* ⇒ *'b update* ⇒ (*'a*×*'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* ⇒ '*a* ⇒ '*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* ≠ *G*›
then obtain *z m* **where** *neq'*: ‹*F z m* ≠ *G z m*›
apply *atomize-elim* **by** *auto*
obtain *gF sF* **where** *gsF*: ‹*F z m* = (*case z* (*gF m*) *of None* ⇒ *None* | *Some x* ⇒ *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* ⇒ *None* | *Some x* ⇒ *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* ≠ *gG m*› ‹*F z m* = *Some x*›
| (*neqG*) *y* **where** ‹*gF m* ≠ *gG m*› ‹*G z m* = *Some y*›
| (*neqNone*) ‹*gF m* ≠ *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 ‹F (update1 x y) = G (update1 x y)› for x y
    using FGeq[of ‹update1 (fst x) (fst y)› ‹update1 (snd x) (snd y)›]
    apply (auto intro!:ext simp: tensor-update-def[abs-def] update1-def[abs-def])
    by (smt (z3) assms(1) assms(2) option.case(2) option.case-eq-if preregister-def prod.collapse)
  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)›
  apply (auto simp: register-def)
  apply (rule exI[of - fst])
  apply (rule exI[of - ‹λx' (x,y). (x',y)›])
```

by (*auto 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* (λ*a*. *tensor-update Some a*)›
apply (*auto simp*: *register-def*)
apply (*rule exI*[*of* - *snd*])
apply (*rule exI*[*of* - ‹λ*y*′ (*x*,*y*). (*x*,*y*′)›])
by (*auto 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* - ‹λ*x 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* ∘ *F*)
unfolding *register-def register-from-getter-setter-def* **by** *blast*
qed

lemma *register-mult*: *register F* ⟹ *F a* ∘$_m$ *F b* = *F* (*a* ∘$_m$ *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*›
using *compat* **apply** (*auto intro*!: *ext simp*: *setter-def register-apply-def o-def map-comp-def*)
by (*smt* (*verit*, *best*) *assms*(*1*) *assms*(*2*) *option.case-eq-if option.distinct*(*1*) *register-def register-from-getter-setter-def*)

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*: ‹$\bigwedge$*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** ⊗$_u$ *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* ⊗*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* ⊗*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* ⊗*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 Some*)›
using *assms* **apply** (*rule register-comp*[*unfolded o-def*])
by *simp*

6.4 Tensor product of registers

definition *register-tensor* (**infixr** ⊗*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*⊗*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*::*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⊗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*)

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

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

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

lemma *compatible-comp-inner*[*simp*]:
compatible F G ⟹ *register H* ⟹ *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* ⟹ *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 $(A;\ B)\ o\ (C \otimes_r D) = (A\ o\ C;\ B\ o\ D)$
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 $F :: 'a::type\ update \Rightarrow 'c::type\ update$ **and** $G :: 'b::type\ update \Rightarrow 'c::type\ update$
assumes *compatible F G*
shows *compatible* $(\lambda a.\ Some \otimes_u F\ a)\ (\lambda a.\ Some \otimes_u G\ a)$
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* $(\lambda a.\ F\ a \otimes_u Some)\ (\lambda a.\ G\ a \otimes_u Some)$
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* ∘ *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*

6.6 Fst and Snd

definition *Fst* **where** ‹*Fst a* = *a* ⊗*u* *Some*›
definition *Snd* **where** ‹*Snd a* = *Some* ⊗*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* (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

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*
unfolding Φ-*def* **apply** (*auto simp*: *register-pair-apply*)

by (*metis assms(1) assms(2) compatible-def tensor-update-mult*)
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

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

lemma [*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*)
show *?thesis*

apply (*rule iso-registerI*[**where** *G*=‹*G' o F'*›])
apply (*auto simp*: *register-tensor-is-register iso-register-is-register register-tensor-distrib*)
apply (*metis* ‹*F* ∘ *F'* = *id*› ‹*G* ∘ *G'* = *id*› *fcomp-assoc fcomp-comp id-fcomp*)
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*)
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-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*)

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

bundle *no-register-notation* **begin**
no-notation *register-tensor* (**infixr** $\otimes_r$ *70*)
no-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
begin

— Remove notation that collides with the notation we use
no-notation *Order.top* ($\top_1$)
no-notation *m-inv* (*inv*₁ - [*81*] *80*)
unbundle *no-vec-syntax*
unbundle *no-inner-syntax*

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

abbreviation *butterket i j ≡ butterfly* (*ket i*) (*ket j*)
abbreviation *selfbutterket i ≡ butterfly* (*ket i*) (*ket i*)

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
apply *intro-classes*
apply (*auto simp*: *enum-bit-def enum-all-bit-def enum-ex-bit-def*)
apply (*metis bit-not-one-iff*)
by (*metis bit-not-zero-iff*)
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 *apply-id-cblinfun*[*simp*]: ‹(∗$_V$) *id-cblinfun* = *id*›
by *auto*

Overriding
isaComplex-Bounded-Operators.Complex-Bounded-Linear-Function.*sandwich*. The latter is the same function but defined as a - ⇒$_{CL}$ - which is less convenient for us.

definition *sandwich* **where** ‹*sandwich a b* = *a* o$_{CL}$ *b* o$_{CL}$ *a*∗›

lemma *clinear-sandwich*[*simp*]: ‹*clinear* (*sandwich a*)›
apply (*rule clinearI*)
apply (*simp add*: *bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose bounded-cbilinear.add-right sandwich-def*)
by (*simp add*: *sandwich-def*)

lemma *sandwich-id*[*simp*]: ‹*sandwich id-cblinfun* = *id*›
by (*auto simp*: *sandwich-def*)

lemma *mat-of-cblinfun-sandwich*:
fixes *a* :: (-::*onb-enum*, -::*onb-enum*) *cblinfun*
shows ‹*mat-of-cblinfun* (*sandwich a b*) = (*let a′* = *mat-of-cblinfun a in a′* ∗ *mat-of-cblinfun b* ∗ *mat-adjoint a′*)›
by (*simp add*: *mat-of-cblinfun-compose sandwich-def Let-def mat-of-cblinfun-adj*)

lemma *prod-cases3′* [*cases type*]:

obtains (*fields*) *a b c* **where** $y = ((a, b), c)$
by (*cases y*, *case-tac a*) *blast*

lemma *lift-cblinfun-comp*:
assumes ‹$a\ o_{CL}\ b = c$›
shows ‹$a\ o_{CL}\ b = c$›
and ‹$a\ o_{CL}\ (b\ o_{CL}\ d) = c\ o_{CL}\ d$›
and ‹$a *_S (b *_S S) = c *_S S$›
and ‹$a *_V (b *_V x) = c *_V x$›
apply (*fact assms*)
apply (*simp add*: *assms cblinfun-assoc-left*(*1*))
using *assms cblinfun-assoc-left*(*2*) **apply** *force*
using *assms* **by** *force*

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 *enum-inj*:
assumes $i < CARD('a)$ **and** $j < CARD('a)$
shows (*Enum.enum* ! *i* :: *'a*::*enum*) = *Enum.enum* ! *j* ⟷ *i* = *j*
using *inj-on-nth*[*OF enum-distinct*, **where** *I*=‹{..<*CARD*(*'a*)}›]
using *assms* **by** (*auto dest*: *inj-onD simp flip*: *card-UNIV-length-enum*)

lemma [*simp*]: *dim-col* (*mat-adjoint m*) = *dim-row m*
unfolding *mat-adjoint-def* **by** *simp*
lemma [*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*: ‹$(\exists x\ y.\ Q\ x\ y) \longleftrightarrow (\exists z.\ Q\ (fst\ z)\ (snd\ z))$›
by *simp*

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* = $(\forall x.\ f\ x \neq None)$›

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* $\longleftrightarrow$ $(\forall a\ b.$ *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*]: ‹$\forall a\ b.$ *setter F a* ∘ *setter G b* = *setter G b* ∘ *setter F a*›
have ‹*valid-getter-setter* (*getter F*) (*setter F*)›
by *auto*

then have ‹$F\ a \circ_m\ G\ b = G\ b \circ_m\ F\ a$› **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*)
unfolding *register-from-getter-setter-def valid-getter-setter-def*
apply (*auto intro*!: *ext simp*: *option.case-eq-if map-comp-def*)
apply ((*metis commute option.distinct option.simps*)+)[*4*]
apply (*smt* (*verit*, *ccfv-threshold*) *assms*(*2*) *commute valid-getter-setter-def valid-getter-setter-getter-setter*)
apply (*smt* (*verit*, *ccfv-threshold*) *assms*(*2*) *commute valid-getter-setter-def valid-getter-setter-getter-setter*)
by (*smt* (*verit*, *del-insts*) *assms*(*2*) *commute option.inject valid-getter-setter-def valid-getter-setter-getter-setter*)
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 ‹$\bigwedge x\ y\ m.\ s\ x\ (s'\ y\ m) = s'\ y\ (s\ x\ m)$›
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 $p :: 'b \Rightarrow 'a$
assumes [*simp*]: *bij p*
shows *register* (*permutation-register p*)
apply (*auto intro*!: *register-register-from-getter-setter simp*: *permutation-register-def valid-getter-setter-def bij-inv-eq-iff*)
by (*meson assms 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-setter*

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 Tensor products (finite dimensional)

theory *Finite-Tensor-Product*
imports *Complex-Bounded-Operators.Complex-L2 Misc*
begin

declare *cblinfun.scaleC-right*[*simp*]

unbundle *cblinfun-notation*
no-notation *m-inv* (*inv*ı - [*81*] *80*)

lift-definition *tensor-ell2* :: ‹'*a*::*finite ell2* ⇒ '*b*::*finite ell2* ⇒ ('*a*×'*b*) *ell2*› (**infixr** ⊗$_s$ *70*) **is**
‹λψ φ (*i*,*j*). ψ *i* ∗ φ *j*›
by *simp*

lemma *tensor-ell2-add2*: ‹*tensor-ell2 a* (*b* + *c*) = *tensor-ell2 a b* + *tensor-ell2 a c*›
apply *transfer* **apply** (*rule ext*) **apply** (*auto simp*: *case-prod-beta*)
by (*meson algebra-simps*)

lemma *tensor-ell2-add1*: ‹*tensor-ell2* (*a* + *b*) *c* = *tensor-ell2 a c* + *tensor-ell2 b c*›
apply *transfer* **apply** (*rule ext*) **apply** (*auto simp*: *case-prod-beta*)
by (*simp add*: *vector-space-over-itself.scale-left-distrib*)

lemma *tensor-ell2-scaleC2*: ‹*tensor-ell2 a* (*c* ∗$_C$ *b*) = *c* ∗$_C$ *tensor-ell2 a b*›
apply *transfer* **apply** (*rule ext*) **by** (*auto simp*: *case-prod-beta*)

lemma *tensor-ell2-scaleC1*: ‹*tensor-ell2* (*c* ∗$_C$ *a*) *b* = *c* ∗$_C$ *tensor-ell2 a b*›
apply *transfer* **apply** (*rule ext*) **by** (*auto simp*: *case-prod-beta*)

lemma *tensor-ell2-inner-prod*[*simp*]: ‹⟨*tensor-ell2 a b*, *tensor-ell2 c d*⟩ = ⟨*a*,*c*⟩ ∗ ⟨*b*,*d*⟩›
apply *transfer*
by (*auto simp*: *case-prod-beta sum-product sum.cartesian-product mult.assoc mult.left-commute*)

lemma *clinear-tensor-ell21*: *clinear* (λ*b*. *tensor-ell2 a b*)
apply (*rule clinearI*; *transfer*)
apply (*auto simp*: *case-prod-beta*)

by (*simp add*: *cond-case-prod-eta algebra-simps*)

lemma *clinear-tensor-ell22*: *clinear* (λ*a*. *tensor-ell2 a b*)
apply (*rule clinearI*; *transfer*)
apply (*auto simp*: *case-prod-beta*)
by (*simp add*: *case-prod-beta′ algebra-simps*)

lemma *tensor-ell2-ket*[*simp*]: *tensor-ell2* (*ket i*) (*ket j*) = *ket* (*i*,*j*)
apply *transfer* **by** *auto*

definition *tensor-op* :: ‹('*a ell2*, '*b*::*finite ell2*) *cblinfun* ⇒ ('*c ell2*, '*d*::*finite ell2*) *cblinfun*
⇒ (('*a*×'*c*) *ell2*, ('*b*×'*d*) *ell2*) *cblinfun*› (**infixr** ⊗*o* *70*) **where**
‹*tensor-op M N* = (*SOME P*. ∀*a c*. *P* ∗*V* (*ket* (*a*,*c*))
= *tensor-ell2* (*M* ∗*V* *ket a*) (*N* ∗*V* *ket c*))›

lemma *tensor-op-ket*:
fixes *a* :: ‹'*a*::*finite*› **and** *b* :: ‹'*b*› **and** *c* :: ‹'*c*::*finite*› **and** *d* :: ‹'*d*›
shows ‹*tensor-op M N* ∗*V* (*ket* (*a*,*c*)) = *tensor-ell2* (*M* ∗*V* *ket a*) (*N* ∗*V* *ket c*)›
proof −
define *S* :: ‹('*a*×'*c*) *ell2 set*› **where** *S* = *ket* ‘ *UNIV*
define φ **where** ‹φ = (λ(*a*,*c*). *tensor-ell2* (*M* ∗*V* *ket a*) (*N* ∗*V* *ket c*))›
define φ′ **where** ‹φ′ = φ ∘ *inv ket*›

have *def*: ‹*tensor-op M N* = (*SOME P*. ∀*a c*. *P* ∗*V* (*ket* (*a*,*c*)) = φ (*a*,*c*))›
unfolding *tensor-op-def* φ-*def* **by** *auto*

have ‹*cindependent S*›
using *S-def cindependent-ket* **by** *blast*
moreover have ‹*cspan S* = *UNIV*›
using *S-def cspan-range-ket-finite* **by** *blast*
ultimately have *cblinfun-extension-exists S* φ′
by (*rule cblinfun-extension-exists-finite-dim*)
then have ∃ *P*. ∀*x*∈*S*. *P* ∗*V* *x* = φ′ *x*
unfolding *cblinfun-extension-exists-def* **by** *auto*
then have *ex*: ‹∃ *P*. ∀*a c*. *P* ∗*V* *ket* (*a*,*c*) = φ (*a*,*c*)›
by (*metis S-def* φ′-*def comp-eq-dest-lhs inj-ket inv-f-f rangeI*)

then have ‹*tensor-op M N* ∗*V* (*ket* (*a*,*c*)) = φ (*a*,*c*)›
unfolding *def* **apply** (*rule someI2-ex*[**where** *P*=‹λ*P*. ∀*a c*. *P* ∗*V* (*ket* (*a*,*c*)) = φ (*a*,*c*)›])
by *auto*
then show *?thesis*
unfolding φ-*def* **by** *auto*
qed

lemma *tensor-op-ell2*: *tensor-op A B* ∗*V* *tensor-ell2* ψ φ = *tensor-ell2* (*A* ∗*V* ψ) (*B* ∗*V* φ)
proof −
have *1*: ‹*clinear* (λ*a*. *tensor-op A B* ∗*V* *tensor-ell2 a* (*ket b*))› **for** *b*
by (*auto intro*!: *clinearI simp*: *tensor-ell2-add1 tensor-ell2-scaleC1 cblinfun.add-right*)
have *2*: ‹*clinear* (λ*a*. *tensor-ell2* (*A* ∗*V* *a*) (*B* ∗*V* *ket b*))› **for** *b*
by (*auto intro*!: *clinearI simp*: *tensor-ell2-add1 tensor-ell2-scaleC1 cblinfun.add-right*)
have *3*: ‹*clinear* (λ*a*. *tensor-op A B* ∗*V* *tensor-ell2* ψ *a*)›
by (*auto intro*!: *clinearI simp*: *tensor-ell2-add2 tensor-ell2-scaleC2 cblinfun.add-right*)
have *4*: ‹*clinear* (λ*a*. *tensor-ell2* (*A* ∗*V* ψ) (*B* ∗*V* *a*))›
by (*auto intro*!: *clinearI simp*: *tensor-ell2-add2 tensor-ell2-scaleC2 cblinfun.add-right*)

have *eq-ket-ket*: ‹*tensor-op A B* ∗*V* *tensor-ell2* (*ket a*) (*ket b*) = *tensor-ell2* (*A* ∗*V* *ket a*) (*B* ∗*V* *ket b*)› **for** *a b*
by (*simp add*: *tensor-op-ket*)
have *eq-ket*: ‹*tensor-op A B* ∗*V* *tensor-ell2* ψ (*ket b*) = *tensor-ell2* (*A* ∗*V* ψ) (*B* ∗*V* *ket b*)› **for** *b*
apply (*rule fun-cong*[**where** *x*=ψ])
using *1 2 eq-ket-ket* **by** (*rule clinear-equal-ket*)

show *?thesis*
apply (*rule fun-cong*[**where** $x=\varphi$])
using *3 4 eq-ket* **by** (*rule clinear-equal-ket*)
qed

lemma *comp-tensor-op*: (*tensor-op a b*) o_{CL} (*tensor-op c d*) = *tensor-op* (*a* o_{CL} *c*) (*b* o_{CL} *d*)
for *a* :: *'e*::*finite ell2* $\Rightarrow_{CL}$ *'c*::*finite ell2* **and** *b* :: *'f*::*finite ell2* $\Rightarrow_{CL}$ *'d*::*finite ell2* **and**
c :: *'a*::*finite ell2* $\Rightarrow_{CL}$ *'e ell2* **and** *d* :: *'b*::*finite ell2* $\Rightarrow_{CL}$ *'f ell2*
apply (*rule equal-ket*)
apply (*rename-tac ij*, *case-tac ij*, *rename-tac i j*, *hypsubst-thin*)
by (*simp flip*: *tensor-ell2-ket add*: *tensor-op-ell2 cblinfun-apply-cblinfun-compose*)

lemma *tensor-op-cbilinear*: ‹*cbilinear* (*tensor-op* :: *'a*::*finite ell2* $\Rightarrow_{CL}$ *'b*::*finite ell2*
$\Rightarrow$ *'c*::*finite ell2* $\Rightarrow_{CL}$ *'d*::*finite ell2* $\Rightarrow$ -)›
proof −
have ‹*clinear* (*λb*::*'c ell2* $\Rightarrow_{CL}$ *'d ell2*. *tensor-op a b*)› **for** *a* :: ‹*'a ell2* $\Rightarrow_{CL}$ *'b ell2*›
apply (*rule clinearI*)
apply (*rule equal-ket*, *rename-tac ij*, *case-tac ij*, *rename-tac i j*, *hypsubst-thin*)
apply (*simp flip*: *tensor-ell2-ket add*: *tensor-op-ell2 cblinfun.add-left tensor-ell2-add2*)
apply (*rule equal-ket*, *rename-tac ij*, *case-tac ij*, *rename-tac i j*, *hypsubst-thin*)
by (*simp add*: *scaleC-cblinfun.rep-eq tensor-ell2-scaleC2 tensor-op-ket*)

moreover have ‹*clinear* (*λa*::*'a*::*finite ell2* $\Rightarrow_{CL}$ *'b*::*finite ell2*. *tensor-op a b*)› **for** *b* :: ‹*'c ell2* $\Rightarrow_{CL}$ *'d ell2*›
apply (*rule clinearI*)
apply (*rule equal-ket*, *rename-tac ij*, *case-tac ij*, *rename-tac i j*, *hypsubst-thin*)
apply (*simp flip*: *tensor-ell2-ket add*: *tensor-op-ell2 cblinfun.add-left tensor-ell2-add1*)
apply (*rule equal-ket*, *rename-tac ij*, *case-tac ij*, *rename-tac i j*, *hypsubst-thin*)
by (*simp add*: *scaleC-cblinfun.rep-eq tensor-ell2-scaleC1 tensor-op-ket*)

ultimately show *?thesis*
unfolding *cbilinear-def* **by** *auto*
qed

lemma *tensor-butter*: ‹*tensor-op* (*butterket i j*) (*butterket k l*) = *butterket* (*i*,*k*) (*j*,*l*)›
for *i* :: - **and** *j* :: -::*finite* **and** *k* :: - **and** *l* :: -::*finite*
apply (*rule equal-ket*, *rename-tac x*, *case-tac x*)
apply (*auto simp flip*: *tensor-ell2-ket simp*: *cblinfun-apply-cblinfun-compose tensor-op-ell2 butterfly-def*)
by (*auto simp*: *tensor-ell2-scaleC1 tensor-ell2-scaleC2*)

lemma *cspan-tensor-op*: ‹*cspan* {*tensor-op* (*butterket i j*) (*butterket k l*)| *i* (*j*::-::*finite*) *k* (*l*::-::*finite*). *True*} = *UNIV*›
unfolding *tensor-butter*
apply (*subst cspan-butterfly-ket*[*symmetric*])
by (*metis surj-pair*)

lemma *cindependent-tensor-op*: ‹*cindependent* {*tensor-op* (*butterket i j*) (*butterket k l*)| *i* (*j*::-::*finite*) *k* (*l*::-::*finite*). *True*}›
unfolding *tensor-butter*
using *cindependent-butterfly-ket*
by (*smt* (*z3*) *Collect-mono-iff complex-vector.independent-mono*)

lemma *tensor-extensionality*:
fixes *F G* :: ‹((*'a*::*finite* × *'b*::*finite*) *ell2*) $\Rightarrow_{CL}$ ((*'c*::*finite* × *'d*::*finite*) *ell2*)) $\Rightarrow$ *'e*::*complex-vector*›
assumes [*simp*]: *clinear F clinear G*
assumes *tensor-eq*: ($\bigwedge$*a b*. *F* (*tensor-op a b*) = *G* (*tensor-op a b*))
shows *F* = *G*
proof (*rule ext*, *rule complex-vector.linear-eq-on-span*[**where** *f*=*F* **and** *g*=*G*])
show ‹*clinear F*› **and** ‹*clinear G*›
using *assms* **by** (*simp-all add*: *cbilinear-def*)
show ‹*x* ∈ *cspan* {*tensor-op* (*butterket i j*) (*butterket k l*)| *i j k l*. *True*}›

```
    for x :: ‹('a × 'b) ell2 ⇒CL ('c × 'd) ell2›
    using cspan-tensor-op by auto
  show ‹F x = G x› if ‹x ∈ {tensor-op (butterket i j) (butterket k l) |i j k l. True}› for x
    using that by (auto simp: tensor-eq)
qed

lemma tensor-id[simp]: ‹tensor-op id-cblinfun id-cblinfun = id-cblinfun›
  apply (rule equal-ket, rename-tac x, case-tac x)
  by (simp flip: tensor-ell2-ket add: tensor-op-ell2)

lemma tensor-op-adjoint: ‹(tensor-op a b)* = tensor-op (a*) (b*)›
  apply (rule cinner-ket-adjointI[symmetric])
  apply (auto simp flip: tensor-ell2-ket simp: tensor-op-ell2)
  by (simp add: cinner-adj-left)

lemma tensor-butterfly[simp]: tensor-op (butterfly ψ ψ') (butterfly φ φ') = butterfly (tensor-ell2 ψ φ) (tensor-ell2
ψ' φ')
  apply (rule equal-ket, rename-tac x, case-tac x)
  by (simp flip: tensor-ell2-ket add: tensor-op-ell2 butterfly-def
      cblinfun-apply-cblinfun-compose tensor-ell2-scaleC1 tensor-ell2-scaleC2)

definition tensor-lift :: ‹(('a1::finite ell2 ⇒CL 'a2::finite ell2) ⇒ ('b1::finite ell2 ⇒CL 'b2::finite ell2) ⇒ 'c)
                    ⇒ ((('a1×'b1) ell2 ⇒CL ('a2×'b2) ell2) ⇒ 'c::complex-vector)› where
  tensor-lift F2 = (SOME G. clinear G ∧ (∀ a b. G (tensor-op a b) = F2 a b))

lemma
  fixes F2 :: 'a::finite ell2 ⇒CL 'b::finite ell2
          ⇒ 'c::finite ell2 ⇒CL 'd::finite ell2
          ⇒ 'e::complex-normed-vector
  assumes cbilinear F2
  shows tensor-lift-clinear: clinear (tensor-lift F2)
    and tensor-lift-correct: ‹(λa b. tensor-lift F2 (tensor-op a b)) = F2›
proof −
  define F2' t4 φ where
    ‹F2' = tensor-lift F2› and
    ‹t4 = (λ(i,j,k,l). tensor-op (butterket i j) (butterket k l))› and
    ‹φ m = (let (i,j,k,l) = inv t4 m in F2 (butterket i j) (butterket k l))› for m
  have t4inj: x = y if t4 x = t4 y for x y
  proof (rule ccontr)
    obtain i j k l where x: x = (i,j,k,l) by (meson prod-cases4)
    obtain i' j' k' l' where y: y = (i',j',k',l') by (meson prod-cases4)
    have 1: bra (i,k) *V t4 x *V ket (j,l) = 1
      by (auto simp: t4-def x tensor-op-ell2 butterfly-def cinner-ket simp flip: tensor-ell2-ket)
    assume ‹x ≠ y›
    then have 2: bra (i,k) *V t4 y *V ket (j,l) = 0
      by (auto simp: t4-def x y tensor-op-ell2 butterfly-def cblinfun-apply-cblinfun-compose cinner-ket
          simp flip: tensor-ell2-ket)
    from 1 2 that
    show False
      by auto
  qed
  have ‹φ (tensor-op (butterket i j) (butterket k l)) = F2 (butterket i j) (butterket k l)› for i j k l
    apply (subst asm-rl[of ‹tensor-op (butterket i j) (butterket k l) = t4 (i,j,k,l)›])
     apply (simp add: t4-def)
    by (auto simp add: injI t4inj inv-f-f φ-def)

  have *: ‹range t4 = {tensor-op (butterket i j) (butterket k l) |i j k l. True}›
    apply (auto simp: case-prod-beta t4-def)
    using image-iff by fastforce

  have cblinfun-extension-exists (range t4) φ
    thm cblinfun-extension-exists-finite-dim[where φ=φ]
    apply (rule cblinfun-extension-exists-finite-dim)
```

```
    apply auto unfolding ∗
   using cindependent-tensor-op
   using cspan-tensor-op
   by auto

  then obtain G where G: ‹G ∗V (t4 (i,j,k,l)) = F2 (butterket i j) (butterket k l)› for i j k l
   apply atomize-elim
   unfolding cblinfun-extension-exists-def
   apply auto
   by (metis (no-types, lifting) t4inj φ-def f-inv-into-f rangeI split-conv)

  have ∗: ‹G ∗V tensor-op (butterket i j) (butterket k l) = F2 (butterket i j) (butterket k l)› for i j k l
   using G by (auto simp: t4-def)
  have ∗: ‹G ∗V tensor-op a (butterket k l) = F2 a (butterket k l)› for a k l
   apply (rule complex-vector.linear-eq-on-span[where g=‹λa. F2 a -› and B=‹{butterket k l|k l. True}›])
   unfolding cspan-butterfly-ket
   using ∗ apply (auto intro!: clinear-compose[unfolded o-def, where f=‹λa. tensor-op a -› and g=‹(∗V) G›])
    apply (metis cbilinear-def tensor-op-cbilinear)
   using assms unfolding cbilinear-def by blast
  have G-F2: ‹G ∗V tensor-op a b = F2 a b› for a b
   apply (rule complex-vector.linear-eq-on-span[where g=‹F2 a› and B=‹{butterket k l|k l. True}›])
   unfolding cspan-butterfly-ket
   using ∗ apply (auto simp: cblinfun.add-right clinearI
                    intro!: clinear-compose[unfolded o-def, where f=‹tensor-op a› and g=‹(∗V) G›])
   apply (meson cbilinear-def tensor-op-cbilinear)
   using assms unfolding cbilinear-def by blast

  have ‹clinear F2' ∧ (∀ a b. F2' (tensor-op a b) = F2 a b)›
   unfolding F2'-def tensor-lift-def
   apply (rule someI[where x=‹(∗V) G› and P=‹λG. clinear G ∧ (∀ a b. G (tensor-op a b) = F2 a b)›])
   using G-F2 by (simp add: cblinfun.add-right clinearI)

  then show ‹clinear F2'› and ‹(λa b. tensor-lift F2 (tensor-op a b)) = F2›
   unfolding F2'-def by auto
qed

lift-definition assoc-ell20 :: ‹(('a::finite×'b::finite)×'c::finite) ell2 ⇒ ('a×('b×'c)) ell2› is
  ‹λf (a,(b,c)). f ((a,b),c)›
  by auto

lift-definition assoc-ell20' :: ‹('a::finite×('b::finite×'c::finite)) ell2 ⇒ (('a×'b)×'c) ell2› is
  ‹λf ((a,b),c). f (a,(b,c))›
  by auto

lift-definition assoc-ell2 :: ‹(('a::finite×'b::finite)×'c::finite) ell2 ⇒CL ('a×('b×'c)) ell2›
  is assoc-ell20
  apply (subst bounded-clinear-finite-dim)
   apply (rule clinearI; transfer)
  by auto

lift-definition assoc-ell2' :: ‹('a::finite×('b::finite×'c::finite)) ell2 ⇒CL (('a×'b)×'c) ell2› is
  assoc-ell20'
  apply (subst bounded-clinear-finite-dim)
   apply (rule clinearI; transfer)
  by auto

lemma assoc-ell2-tensor: ‹assoc-ell2 ∗V tensor-ell2 (tensor-ell2 a b) c = tensor-ell2 a (tensor-ell2 b c)›
  apply (rule clinear-equal-ket[THEN fun-cong, where x=a])
    apply (simp add: cblinfun.add-right clinearI tensor-ell2-add1 tensor-ell2-scaleC1)
   apply (simp add: clinear-tensor-ell22)
  apply (rule clinear-equal-ket[THEN fun-cong, where x=b])
     apply (simp add: cblinfun.add-right clinearI tensor-ell2-add1 tensor-ell2-add2 tensor-ell2-scaleC1 tensor-ell2-scaleC2)
```

apply (*simp add*: *clinearI tensor-ell2-add1 tensor-ell2-add2 tensor-ell2-scaleC1 tensor-ell2-scaleC2*)
apply (*rule clinear-equal-ket*[*THEN fun-cong*, **where** *x*=*c*])
apply (*simp add*: *cblinfun.add-right clinearI tensor-ell2-add2 tensor-ell2-scaleC2*)
apply (*simp add*: *clinearI tensor-ell2-add2 tensor-ell2-scaleC2*)
unfolding *assoc-ell2.rep-eq*
apply *transfer*
by *auto*

lemma *assoc-ell2′-tensor*: ‹*assoc-ell2′* $*_V$ *tensor-ell2 a* (*tensor-ell2 b c*) = *tensor-ell2* (*tensor-ell2 a b*) *c*›
apply (*rule clinear-equal-ket*[*THEN fun-cong*, **where** *x*=*a*])
apply (*simp add*: *cblinfun.add-right clinearI tensor-ell2-add1 tensor-ell2-scaleC1*)
apply (*simp add*: *clinearI tensor-ell2-add1 tensor-ell2-scaleC1*)
apply (*rule clinear-equal-ket*[*THEN fun-cong*, **where** *x*=*b*])
apply (*simp add*: *cblinfun.add-right clinearI tensor-ell2-add1 tensor-ell2-add2 tensor-ell2-scaleC1 tensor-ell2-scaleC2*)
apply (*simp add*: *clinearI tensor-ell2-add1 tensor-ell2-add2 tensor-ell2-scaleC1 tensor-ell2-scaleC2*)
apply (*rule clinear-equal-ket*[*THEN fun-cong*, **where** *x*=*c*])
apply (*simp add*: *cblinfun.add-right clinearI tensor-ell2-add2 tensor-ell2-scaleC2*)
apply (*simp add*: *clinearI tensor-ell2-add2 tensor-ell2-scaleC2*)
unfolding *assoc-ell2′.rep-eq*
apply *transfer*
by *auto*

lemma *adjoint-assoc-ell2*[*simp*]: ‹*assoc-ell2** = *assoc-ell2′*›
proof (*rule adjoint-eqI*[*symmetric*])
have [*simp*]: ‹*clinear* (*cinner* (*assoc-ell2′* $*_V$ *x*))› **for** *x* :: ‹(′*a* × ′*b* × ′*c*) *ell2*›
by (*metis* (*no-types*, *lifting*) *cblinfun.add-right cinner-scaleC-right clinearI complex-scaleC-def mult.comm-neutral of-complex-def vector-to-cblinfun-adj-apply*)
have [*simp*]: ‹*clinear* (λ*a*. ⟨*x*, *assoc-ell2* $*_V$ *a*⟩)› **for** *x* :: ‹(′*a* × ′*b* × ′*c*) *ell2*›
by (*simp add*: *cblinfun.add-right cinner-add-right clinearI*)
have [*simp*]: ‹*antilinear* (λ*a*. ⟨*a*, *y*⟩)› **for** *y* :: ‹(′*a* × ′*b* × ′*c*) *ell2*›
using *bounded-antilinear-cinner-left bounded-antilinear-def* **by** *blast*
have [*simp*]: ‹*antilinear* (λ*a*. ⟨*assoc-ell2′* $*_V$ *a*, *y*⟩)› **for** *y* :: ‹((′*a* × ′*b*) × ′*c*) *ell2*›
by (*simp add*: *cblinfun.add-right cinner-add-left antilinearI*)
have ‹⟨*assoc-ell2′* $*_V$ (*ket x*), *ket y*⟩ = ⟨*ket x*, *assoc-ell2* $*_V$ *ket y*⟩› **for** *x* :: ‹′*a* × ′*b* × ′*c*› **and** *y*
apply (*cases x*, *cases y*)
by (*simp flip*: *tensor-ell2-ket add*: *assoc-ell2′-tensor assoc-ell2-tensor*)
then have ‹⟨*assoc-ell2′* $*_V$ (*ket x*), *y*⟩ = ⟨*ket x*, *assoc-ell2* $*_V$ *y*⟩› **for** *x* :: ‹′*a* × ′*b* × ′*c*› **and** *y*
by (*rule clinear-equal-ket*[*THEN fun-cong*, *rotated 2*], *simp-all*)
then show ‹⟨*assoc-ell2′* $*_V$ *x*, *y*⟩ = ⟨*x*, *assoc-ell2* $*_V$ *y*⟩› **for** *x* :: ‹(′*a* × ′*b* × ′*c*) *ell2*› **and** *y*
by (*rule antilinear-equal-ket*[*THEN fun-cong*, *rotated 2*], *simp-all*)
qed

lemma *adjoint-assoc-ell2′*[*simp*]: ‹*assoc-ell2′** = *assoc-ell2*›
by (*simp flip*: *adjoint-assoc-ell2*)

lift-definition *swap-ell20* :: ‹(′*a*::*finite*×′*b*::*finite*) *ell2* ⇒ (′*b*×′*a*) *ell2*› **is**
‹λ*f* (*a*,*b*). *f* (*b*,*a*)›
by *auto*

lift-definition *swap-ell2* :: ‹(′*a*::*finite*×′*b*::*finite*) *ell2* ⇒$_{CL}$ (′*b*×′*a*) *ell2*›
is *swap-ell20*
apply (*subst bounded-clinear-finite-dim*)
apply (*rule clinearI*; *transfer*)
by *auto*

lemma *swap-ell2-tensor*[*simp*]: ‹*swap-ell2* $*_V$ *tensor-ell2 a b* = *tensor-ell2 b a*›
apply (*rule clinear-equal-ket*[*THEN fun-cong*, **where** *x*=*a*])
apply (*simp add*: *cblinfun.add-right clinearI tensor-ell2-add1 tensor-ell2-scaleC1*)
apply (*simp add*: *clinear-tensor-ell21*)
apply (*rule clinear-equal-ket*[*THEN fun-cong*, **where** *x*=*b*])
apply (*simp add*: *cblinfun.add-right clinearI tensor-ell2-add1 tensor-ell2-add2 tensor-ell2-scaleC1 ten-*

sor-ell2-scaleC2)
apply (*simp add*: *clinearI tensor-ell2-add1 tensor-ell2-add2 tensor-ell2-scaleC1 tensor-ell2-scaleC2*)
unfolding *swap-ell2.rep-eq*
apply *transfer*
by *auto*

lemma *adjoint-swap-ell2*[*simp*]: ‹*swap-ell2** = *swap-ell2*›
proof (*rule adjoint-eqI*[*symmetric*])
have [*simp*]: ‹*clinear* (*cinner* (*swap-ell2* $*_V$ *x*))› **for** *x* :: ‹('*a* × '*b*) *ell2*›
by (*metis* (*no-types*, *lifting*) *cblinfun.add-right cinner-scaleC-right clinearI complex-scaleC-def mult.comm-neutral of-complex-def vector-to-cblinfun-adj-apply*)
have [*simp*]: ‹*clinear* (λ*a*. ⟨*x*, *swap-ell2* $*_V$ *a*⟩)› **for** *x* :: ‹('*a* × '*b*) *ell2*›
by (*simp add*: *cblinfun.add-right cinner-add-right clinearI*)
have [*simp*]: ‹*antilinear* (λ*a*. ⟨*a*, *y*⟩)› **for** *y* :: ‹('*a* × '*b*) *ell2*›
using *bounded-antilinear-cinner-left bounded-antilinear-def* **by** *blast*
have [*simp*]: ‹*antilinear* (λ*a*. ⟨*swap-ell2* $*_V$ *a*, *y*⟩)› **for** *y* :: ‹('*b* × '*a*) *ell2*›
by (*simp add*: *cblinfun.add-right cinner-add-left antilinearI*)
have ‹⟨*swap-ell2* $*_V$ (*ket x*), *ket y*⟩ = ⟨*ket x*, *swap-ell2* $*_V$ *ket y*⟩› **for** *x* :: ‹'*a* × '*b*› **and** *y*
apply (*cases x*, *cases y*)
by (*simp flip*: *tensor-ell2-ket add*: *swap-ell2-tensor*)
then have ‹⟨*swap-ell2* $*_V$ (*ket x*), *y*⟩ = ⟨*ket x*, *swap-ell2* $*_V$ *y*⟩› **for** *x* :: ‹'*a* × '*b*› **and** *y*
by (*rule clinear-equal-ket*[*THEN fun-cong*, *rotated 2*], *simp-all*)
then show ‹⟨*swap-ell2* $*_V$ *x*, *y*⟩ = ⟨*x*, *swap-ell2* $*_V$ *y*⟩› **for** *x* :: ‹('*a* × '*b*) *ell2*› **and** *y*
apply (*rule antilinear-equal-ket*[*THEN fun-cong*, *rotated 2*])
by *simp-all*
qed

lemma *tensor-ell2-extensionality*:
assumes (⋀*s t*. *a* $*_V$ (*s* ⊗$_s$ *t*) = *b* $*_V$ (*s* ⊗$_s$ *t*))
shows *a* = *b*
apply (*rule equal-ket*, *case-tac x*, *hypsubst-thin*)
by (*simp add*: *assms flip*: *tensor-ell2-ket*)

lemma *assoc-ell2'-assoc-ell2*[*simp*]: ‹*assoc-ell2'* o_{CL} *assoc-ell2* = *id-cblinfun*›
by (*auto intro*!: *equal-ket simp*: *cblinfun-apply-cblinfun-compose assoc-ell2'-tensor assoc-ell2-tensor simp flip*: *tensor-ell2-ket*)

lemma *assoc-ell2-assoc-ell2'*[*simp*]: ‹*assoc-ell2* o_{CL} *assoc-ell2'* = *id-cblinfun*›
by (*auto intro*!: *equal-ket simp*: *cblinfun-apply-cblinfun-compose assoc-ell2'-tensor assoc-ell2-tensor simp flip*: *tensor-ell2-ket*)

lemma *unitary-assoc-ell2*[*simp*]: *unitary assoc-ell2*
unfolding *unitary-def* **by** *auto*

lemma *unitary-assoc-ell2'*[*simp*]: *unitary assoc-ell2'*
unfolding *unitary-def* **by** *auto*

lemma *tensor-op-left-add*: ‹(*x* + *y*) ⊗$_o$ *b* = *x* ⊗$_o$ *b* + *y* ⊗$_o$ *b*›
for *x y* :: ‹'*a*::*finite ell2* ⇒$_{CL}$ '*c*::*finite ell2*› **and** *b* :: ‹'*b*::*finite ell2* ⇒$_{CL}$ '*d*::*finite ell2*›
apply (*auto intro*!: *equal-ket simp*: *tensor-op-ket*)
by (*simp add*: *plus-cblinfun.rep-eq tensor-ell2-add1 tensor-op-ket*)

lemma *tensor-op-right-add*: ‹*b* ⊗$_o$ (*x* + *y*) = *b* ⊗$_o$ *x* + *b* ⊗$_o$ *y*›
for *x y* :: ‹'*a*::*finite ell2* ⇒$_{CL}$ '*c*::*finite ell2*› **and** *b* :: ‹'*b*::*finite ell2* ⇒$_{CL}$ '*d*::*finite ell2*›
apply (*auto intro*!: *equal-ket simp*: *tensor-op-ket*)
by (*simp add*: *plus-cblinfun.rep-eq tensor-ell2-add2 tensor-op-ket*)

lemma *tensor-op-scaleC-left*: ‹(*c* $*_C$ *x*) ⊗$_o$ *b* = *c* $*_C$ (*x* ⊗$_o$ *b*)›
for *x* :: ‹'*a*::*finite ell2* ⇒$_{CL}$ '*c*::*finite ell2*› **and** *b* :: ‹'*b*::*finite ell2* ⇒$_{CL}$ '*d*::*finite ell2*›
apply (*auto intro*!: *equal-ket simp*: *tensor-op-ket*)
by (*metis scaleC-cblinfun.rep-eq tensor-ell2-ket tensor-ell2-scaleC1 tensor-op-ell2*)

lemma *tensor-op-scaleC-right*: ‹$b \otimes_o (c *_C x) = c *_C (b \otimes_o x)$›
  for x :: ‹$'a$::*finite ell2* $\Rightarrow_{CL}$ $'c$::*finite ell2*› **and** b :: ‹$'b$::*finite ell2* $\Rightarrow_{CL}$ $'d$::*finite ell2*›
  apply (*auto intro*!: *equal-ket simp*: *tensor-op-ket*)
  by (*metis scaleC-cblinfun.rep-eq tensor-ell2-ket tensor-ell2-scaleC2 tensor-op-ell2*)

lemma *clinear-tensor-left*[*simp*]: ‹*clinear* ($\lambda a.\ a \otimes_o b$ :: -::*finite ell2* $\Rightarrow_{CL}$ -::*finite ell2*)›
  apply (*rule clinearI*)
   apply (*rule tensor-op-left-add*)
  by (*rule tensor-op-scaleC-left*)

lemma *clinear-tensor-right*[*simp*]: ‹*clinear* ($\lambda b.\ a \otimes_o b$ :: -::*finite ell2* $\Rightarrow_{CL}$ -::*finite ell2*)›
  apply (*rule clinearI*)
   apply (*rule tensor-op-right-add*)
  by (*rule tensor-op-scaleC-right*)

lemma *tensor-ell2-nonzero*: ‹$a \otimes_s b \neq 0$› **if** ‹$a \neq 0$› **and** ‹$b \neq 0$›
  apply (*use that* **in** *transfer*)
  apply *auto*
  by (*metis mult-eq-0-iff old.prod.case*)

lemma *tensor-op-nonzero*:
  fixes a :: ‹$'a$::*finite ell2* $\Rightarrow_{CL}$ $'c$::*finite ell2*› **and** b :: ‹$'b$::*finite ell2* $\Rightarrow_{CL}$ $'d$::*finite ell2*›
  assumes ‹$a \neq 0$› **and** ‹$b \neq 0$›
  shows ‹$a \otimes_o b \neq 0$›
proof −
  from ‹$a \neq 0$› **obtain** i **where** i: ‹$a *_V$ *ket* $i \neq 0$›
    by (*metis cblinfun.zero-left equal-ket*)
  from ‹$b \neq 0$› **obtain** j **where** j: ‹$b *_V$ *ket* $j \neq 0$›
    by (*metis cblinfun.zero-left equal-ket*)
  from i j **have** *ijneq0*: ‹$(a *_V$ *ket* $i) \otimes_s (b *_V$ *ket* $j) \neq 0$›
    by (*simp add*: *tensor-ell2-nonzero*)
  have ‹$(a *_V$ *ket* $i) \otimes_s (b *_V$ *ket* $j) = (a \otimes_o b) *_V$ *ket* (i,j)›
    by (*simp add*: *tensor-op-ket*)
  with *ijneq0* **show** ‹$a \otimes_o b \neq 0$›
    by *force*
qed

lemma *inj-tensor-ell2-left*: ‹*inj* (λa::$'a$::*finite ell2*. $a \otimes_s b$)› **if** ‹$b \neq 0$› **for** b :: ‹$'b$::*finite ell2*›
proof (*rule injI*, *rule ccontr*)
  fix x y :: ‹$'a$ *ell2*›
  assume *eq*: ‹$x \otimes_s b = y \otimes_s b$›
  assume *neq*: ‹$x \neq y$›
  define a **where** ‹$a = x - y$›
  from *neq a-def* **have** *neq0*: ‹$a \neq 0$›
    by *auto*
  with ‹$b \neq 0$› **have** ‹$a \otimes_s b \neq 0$›
    by (*simp add*: *tensor-ell2-nonzero*)
  then have ‹$x \otimes_s b \neq y \otimes_s b$›
    unfolding *a-def*
    by (*metis add-cancel-left-left diff-add-cancel tensor-ell2-add1*)
  with *eq* **show** *False*
    by *auto*
qed

lemma *inj-tensor-ell2-right*: ‹*inj* (λb::$'b$::*finite ell2*. $a \otimes_s b$)› **if** ‹$a \neq 0$› **for** a :: ‹$'a$::*finite ell2*›
proof (*rule injI*, *rule ccontr*)
  fix x y :: ‹$'b$ *ell2*›
  assume *eq*: ‹$a \otimes_s x = a \otimes_s y$›
  assume *neq*: ‹$x \neq y$›
  define b **where** ‹$b = x - y$›
  from *neq b-def* **have** *neq0*: ‹$b \neq 0$›
    by *auto*
  with ‹$a \neq 0$› **have** ‹$a \otimes_s b \neq 0$›

```
    by (simp add: tensor-ell2-nonzero)
  then have ‹a ⊗s x ≠ a ⊗s y›
    unfolding b-def
    by (metis add-cancel-left-left diff-add-cancel tensor-ell2-add2)
  with eq show False
    by auto
qed



lemma inj-tensor-left: ‹inj (λa::'a::finite ell2 ⇒CL 'c::finite ell2. a ⊗o b)› if ‹b ≠ 0› for b :: ‹'b::finite ell2 ⇒CL
'd::finite ell2›
proof (rule injI, rule ccontr)
  fix x y :: ‹'a ell2 ⇒CL 'c ell2›
  assume eq: ‹x ⊗o b = y ⊗o b›
  assume neq: ‹x ≠ y›
  define a where ‹a = x − y›
  from neq a-def have neq0: ‹a ≠ 0›
    by auto
  with ‹b ≠ 0› have ‹a ⊗o b ≠ 0›
    by (simp add: tensor-op-nonzero)
  then have ‹x ⊗o b ≠ y ⊗o b›
    unfolding a-def
    by (metis add-cancel-left-left diff-add-cancel tensor-op-left-add)
  with eq show False
    by auto
qed

lemma inj-tensor-right: ‹inj (λb::'b::finite ell2 ⇒CL 'c::finite ell2. a ⊗o b)› if ‹a ≠ 0› for a :: ‹'a::finite ell2
⇒CL 'd::finite ell2›
proof (rule injI, rule ccontr)
  fix x y :: ‹'b ell2 ⇒CL 'c ell2›
  assume eq: ‹a ⊗o x = a ⊗o y›
  assume neq: ‹x ≠ y›
  define b where ‹b = x − y›
  from neq b-def have neq0: ‹b ≠ 0›
    by auto
  with ‹a ≠ 0› have ‹a ⊗o b ≠ 0›
    by (simp add: tensor-op-nonzero)
  then have ‹a ⊗o x ≠ a ⊗o y›
    unfolding b-def
    by (metis add-cancel-left-left diff-add-cancel tensor-op-right-add)
  with eq show False
    by auto
qed

lemma tensor-ell2-almost-injective:
  assumes ‹tensor-ell2 a b = tensor-ell2 c d›
  assumes ‹a ≠ 0›
  shows ‹∃γ. b = γ *C d›
proof −
  from ‹a ≠ 0› obtain i where i: ‹cinner (ket i) a ≠ 0›
    by (metis cinner-eq-zero-iff cinner-ket-left ell2-pointwise-ortho)
  have ‹cinner (ket i ⊗s ket j) (a ⊗s b) = cinner (ket i ⊗s ket j) (c ⊗s d)› for j
    using assms by simp
  then have eq2: ‹(cinner (ket i) a) * (cinner (ket j) b) = (cinner (ket i) c) * (cinner (ket j) d)› for j
    by (metis tensor-ell2-inner-prod)
  then obtain γ where ‹cinner (ket i) c = γ * cinner (ket i) a›
    by (metis i eq-divide-eq)
  with eq2 have ‹(cinner (ket i) a) * (cinner (ket j) b) = (cinner (ket i) a) * (γ * cinner (ket j) d)› for j
    by simp
  then have ‹cinner (ket j) b = cinner (ket j) (γ *C d)› for j
    using i by force
```

then have ‹$b = \gamma *_C d$›
by (*simp add*: *cinner-ket-eqI*)
then show *?thesis*
by *auto*
qed

lemma *tensor-op-almost-injective*:
fixes $a\ c$ :: ‹$'a$::*finite ell2* $\Rightarrow_{CL}$ $'b$::*finite ell2*›
and $b\ d$ :: ‹$'c$::*finite ell2* $\Rightarrow_{CL}$ $'d$::*finite ell2*›
assumes ‹*tensor-op a b* = *tensor-op c d*›
assumes ‹$a \neq 0$›
shows ‹$\exists \gamma.\ b = \gamma *_C d$›
proof (*cases* ‹$d = 0$›)
case *False*
from ‹$a \neq 0$› **obtain** ψ **where** ψ: ‹$a *_V \psi \neq 0$›
by (*metis cblinfun.zero-left cblinfun-eqI*)
have ‹$(a \otimes_o b)\ (\psi \otimes_s \varphi) = (c \otimes_o d)\ (\psi \otimes_s \varphi)$› **for** φ
using *assms* **by** *simp*
then have *eq2*: ‹$(a\ \psi) \otimes_s (b\ \varphi) = (c\ \psi) \otimes_s (d\ \varphi)$› **for** φ
by (*simp add*: *tensor-op-ell2*)
then have *eq2'*: ‹$(d\ \varphi) \otimes_s (c\ \psi) = (b\ \varphi) \otimes_s (a\ \psi)$› **for** φ
by (*metis swap-ell2-tensor*)
from *False* **obtain** *φ0* **where** *φ0*: ‹$d\ \varphi 0 \neq 0$›
by (*metis cblinfun.zero-left cblinfun-eqI*)
obtain γ **where** ‹$c\ \psi = \gamma *_C a\ \psi$›
apply *atomize-elim*
using *eq2' φ0* **by** (*rule tensor-ell2-almost-injective*)
with *eq2* **have** ‹$(a\ \psi) \otimes_s (b\ \varphi) = (a\ \psi) \otimes_s (\gamma *_C d\ \varphi)$› **for** φ
by (*simp add*: *tensor-ell2-scaleC1 tensor-ell2-scaleC2*)
then have ‹$b\ \varphi = \gamma *_C d\ \varphi$› **for** φ
by (*smt* (*verit*, *best*) ψ *complex-vector.scale-cancel-right tensor-ell2-almost-injective tensor-ell2-nonzero tensor-ell2-scaleC2*)
then have ‹$b = \gamma *_C d$›
by (*simp add*: *cblinfun-eqI*)
then show *?thesis*
by *auto*
next
case *True*
then have ‹$c \otimes_o d = 0$›
by (*metis add-cancel-right-left tensor-op-right-add*)
then have ‹$a \otimes_o b = 0$›
using *assms*(*1*) **by** *presburger*
with ‹$a \neq 0$› **have** ‹$b = 0$›
by (*meson tensor-op-nonzero*)
then show *?thesis*
by *auto*
qed

lemma *tensor-ell2-0-left*[*simp*]: ‹*tensor-ell2 0 x* = *0*›
apply *transfer* **by** *auto*

lemma *tensor-ell2-0-right*[*simp*]: ‹*tensor-ell2 x 0* = *0*›
apply *transfer* **by** *auto*

lemma *tensor-op-0-left*[*simp*]: ‹*tensor-op 0 x* = (*0* :: ($'a$::*finite*∗$'b$::*finite*) *ell2* $\Rightarrow_{CL}$ ($'c$::*finite*∗$'d$::*finite*) *ell2*)›
apply (*rule equal-ket*)
by (*auto simp flip*: *tensor-ell2-ket simp*: *tensor-op-ell2*)

lemma *tensor-op-0-right*[*simp*]: ‹*tensor-op x 0* = (*0* :: ($'a$::*finite*∗$'b$::*finite*) *ell2* $\Rightarrow_{CL}$ ($'c$::*finite*∗$'d$::*finite*) *ell2*)›
apply (*rule equal-ket*)
by (*auto simp flip*: *tensor-ell2-ket simp*: *tensor-op-ell2*)

```
lemma bij-tensor-ell2-one-dim-left:
  assumes ‹ψ ≠ 0›
  shows ‹bij (λx::'b::finite ell2. (ψ :: 'a::CARD-1 ell2) ⊗s x)›
proof (rule bijI)
  show ‹inj (λx::'b::finite ell2. (ψ :: 'a::CARD-1 ell2) ⊗s x)›
    using assms by (rule inj-tensor-ell2-right)
  have ‹∃x. ψ ⊗s x = φ› for φ :: ‹('a*'b) ell2›
  proof (use assms in transfer)
    fix ψ :: ‹'a ⇒ complex› and φ :: ‹'a*'b ⇒ complex›
    assume ‹has-ell2-norm φ› and ‹ψ ≠ (λ-. 0)›
    define c where ‹c = ψ undefined›
    then have ‹ψ a = c› for a
      apply (subst everything-the-same[of - undefined])
      by simp
    with ‹ψ ≠ (λ-. 0)› have ‹c ≠ 0›
      by auto

    define x where ‹x j = φ (undefined, j) / c› for j
    have ‹(λ(i, j). ψ i * x j) = φ›
      apply (auto intro!: ext simp: x-def ‹ψ - = c› ‹c ≠ 0›)
      apply (subst (2) everything-the-same[of - undefined])
      by simp
    then show ‹∃x∈Collect has-ell2-norm. (λ(i, j). ψ i * x j) = φ›
      apply (rule bexI[where x=x])
      by simp
  qed

  then show ‹surj (λx::'b::finite ell2. (ψ :: 'a::CARD-1 ell2) ⊗s x)›
    by (metis surj-def)
qed

lemma bij-tensor-op-one-dim-left:
  assumes ‹a ≠ 0›
 shows ‹bij (λx::'c::finite ell2 ⇒CL 'd::finite ell2. (a :: 'a::{CARD-1,enum} ell2 ⇒CL 'b::{CARD-1,enum} ell2)
⊗o x)›
proof (rule bijI)
  define t where ‹t = (λx::'c ell2 ⇒CL 'd ell2. (a :: 'a ell2 ⇒CL 'b ell2) ⊗o x)›
  define i where
    ‹i = tensor-lift (λ(x::'a ell2 ⇒CL 'b ell2) (y::'c ell2 ⇒CL 'd ell2). (one-dim-iso x / one-dim-iso a) *C y)›

  have [simp]: ‹clinear i›
    by (auto intro!: tensor-lift-clinear simp: i-def cbilinear-def clinearI scaleC-add-left add-divide-distrib)
  have [simp]: ‹clinear t›
    by (simp add: t-def)
  have ‹i (x ⊗o y) = (one-dim-iso x / one-dim-iso a) *C y› for x y
    by (auto intro!: clinearI tensor-lift-correct[THEN fun-cong, THEN fun-cong] simp: t-def i-def cbilinear-def
scaleC-add-left add-divide-distrib)
  then have ‹t (i (x ⊗o y)) = x ⊗o y› for x y
    apply (simp add: t-def)
   by (smt (z3) assms complex-vector.scale-eq-0-iff nonzero-mult-div-cancel-right one-dim-scaleC-1 scaleC-scaleC
tensor-op-scaleC-left tensor-op-scaleC-right times-divide-eq-left)
  then have ‹t (i x) = x› for x
    apply (rule-tac fun-cong[where x=x])
    apply (rule tensor-extensionality)
    by (auto intro: clinear-compose complex-vector.module-hom-ident simp flip: o-def[of t i])
  then show ‹surj t›
    by (rule surjI)

  show ‹inj t›
    unfolding t-def using assms by (rule inj-tensor-right)
qed
```

lemma *swap-ell2-selfinv*[*simp*]: ‹*swap-ell2* o_{CL} *swap-ell2* = *id-cblinfun*›
apply (*rule tensor-ell2-extensionality*)
by *auto*

lemma *bij-tensor-op-one-dim-right*:
assumes ‹$b \neq 0$›
shows ‹*bij* (λ*x*::'*c*::*finite ell2* ⇒$_{CL}$ '*d*::*finite ell2*. *x* ⊗$_o$ (*b* :: '*a*::{*CARD-1*,*enum*} *ell2* ⇒$_{CL}$ '*b*::{*CARD-1*,*enum*} *ell2*))›
(**is** ‹*bij ?f*›)
proof −
let *?sf* = ‹(λ*x*. *swap-ell2* o_{CL} (*?f x*) o_{CL} *swap-ell2*)›
let *?s* = ‹(λ*x*. *swap-ell2* o_{CL} *x* o_{CL} *swap-ell2*)›
let *?g* = ‹(λ*x*::'*c*::*finite ell2* ⇒$_{CL}$ '*d*::*finite ell2*. (*b* :: '*a*::{*CARD-1*,*enum*} *ell2* ⇒$_{CL}$ '*b*::{*CARD-1*,*enum*} *ell2*) ⊗$_o$ *x*)›
have ‹*?sf* = *?g*›
by (*auto intro*!: *ext tensor-ell2-extensionality simp add*: *swap-ell2-tensor tensor-op-ell2*)
have ‹*bij ?g*›
using *assms* **by** (*rule bij-tensor-op-one-dim-left*)
have ‹*?s o ?sf* = *?f*›
apply (*auto intro*!: *ext simp*: *cblinfun-assoc-left*)
by (*auto simp*: *cblinfun-assoc-right*)
also have ‹*bij ?s*›
apply (*rule o-bij*[**where** *g*=‹(λ*x*. *swap-ell2* o_{CL} *x* o_{CL} *swap-ell2*)›])
apply (*auto intro*!: *ext simp*: *cblinfun-assoc-left*)
by (*auto simp*: *cblinfun-assoc-right*)
show ‹*bij ?f*›
apply (*subst* ‹*?s o ?sf* = *?f*›[*symmetric*], *subst* ‹*?sf* = *?g*›)
using ‹*bij ?g*› ‹*bij ?s*› **by** (*rule bij-comp*)
qed

lemma *overlapping-tensor*:
fixes *a23* :: ‹('*a2*::*finite*∗'*a3*::*finite*) *ell2* ⇒$_{CL}$ ('*b2*::*finite*∗'*b3*::*finite*) *ell2*›
and *b12* :: ‹('*a1*::*finite*∗'*a2*) *ell2* ⇒$_{CL}$ ('*b1*::*finite*∗'*b2*) *ell2*›
assumes *eq*: ‹*butterfly* ψ ψ' ⊗$_o$ *a23* = *assoc-ell2* o_{CL} (*b12* ⊗$_o$ *butterfly* φ φ') o_{CL} *assoc-ell2'*›
assumes ‹ψ ≠ 0› ‹ψ' ≠ 0› ‹φ ≠ 0› ‹φ' ≠ 0›
shows ‹∃ *c*. *butterfly* ψ ψ' ⊗$_o$ *a23* = *butterfly* ψ ψ' ⊗$_o$ *c* ⊗$_o$ *butterfly* φ φ'›
proof −
note [[*show-types*]]
let *?id1* = ‹*id-cblinfun* :: *unit ell2* ⇒$_{CL}$ *unit ell2*›
note *id-cblinfun-eq-1*[*simp del*]
define *d* **where** ‹*d* = *butterfly* ψ ψ' ⊗$_o$ *a23*›

define ψ_n ψ_n' $a23_n$ **where** ‹ψ_n = ψ /$_C$ *norm* ψ› **and** ‹ψ_n' = ψ' /$_C$ *norm* ψ'› **and** ‹$a23_n$ = *norm* ψ ∗$_C$ *norm* ψ' ∗$_C$ *a23*›
have [*simp*]: ‹*norm* ψ_n = 1› ‹*norm* ψ_n' = 1›
using ‹ψ ≠ 0› ‹ψ' ≠ 0› **by** (*auto simp*: ψ_n-*def* ψ_n'-*def norm-inverse*)
have *n1*: ‹*butterfly* ψ_n ψ_n' ⊗$_o$ $a23_n$ = *butterfly* ψ ψ' ⊗$_o$ *a23*›
apply (*auto simp*: ψ_n-*def* ψ_n'-*def* $a23_n$-*def tensor-op-scaleC-left tensor-op-scaleC-right*)
by (*metis* (*no-types*, *lifting*) *assms*(*2*) *assms*(*3*) *inverse-mult-distrib mult.commute no-zero-divisors norm-eq-zero of-real-eq-0-iff right-inverse scaleC-one*)

define φ_n φ_n' $b12_n$ **where** ‹φ_n = φ /$_C$ *norm* φ› **and** ‹φ_n' = φ' /$_C$ *norm* φ'› **and** ‹$b12_n$ = *norm* φ ∗$_C$ *norm* φ' ∗$_C$ *b12*›
have [*simp*]: ‹*norm* φ_n = 1› ‹*norm* φ_n' = 1›
using ‹φ ≠ 0› ‹φ' ≠ 0› **by** (*auto simp*: φ_n-*def* φ_n'-*def norm-inverse*)
have *n2*: ‹$b12_n$ ⊗$_o$ *butterfly* φ_n φ_n' = *b12* ⊗$_o$ *butterfly* φ φ'›
apply (*auto simp*: φ_n-*def* φ_n'-*def* $b12_n$-*def tensor-op-scaleC-left tensor-op-scaleC-right*)
by (*metis* (*no-types*, *lifting*) *assms*(*4*) *assms*(*5*) *field-class.field-inverse inverse-mult-distrib mult.commute no-zero-divisors norm-eq-zero of-real-hom.hom-0 scaleC-one*)

define *c'* :: ‹(*unit*∗'*a2*∗*unit*) *ell2* ⇒$_{CL}$ (*unit*∗'*b2*∗*unit*) *ell2*›
where ‹*c'* = (*vector-to-cblinfun* ψ_n ⊗$_o$ *id-cblinfun* ⊗$_o$ *vector-to-cblinfun* φ_n)∗ o_{CL} *d*
o_{CL} (*vector-to-cblinfun* ψ_n' ⊗$_o$ *id-cblinfun* ⊗$_o$ *vector-to-cblinfun* φ_n')›

define c'' :: ‹$'a2\ ell2 \Rightarrow_{CL}\ 'b2\ ell2$›
where ‹$c'' = inv\ (\lambda c''.\ id\text{-}cblinfun \otimes_o c'' \otimes_o id\text{-}cblinfun)\ c'$›

have $*$: ‹$bij\ (\lambda c''::'a2\ ell2 \Rightarrow_{CL}\ 'b2\ ell2.\ ?id1 \otimes_o c'' \otimes_o ?id1)$›
apply ($subst\ asm\text{-}rl[of$ ‹$- = (\lambda x.\ id\text{-}cblinfun \otimes_o x)\ o\ (\lambda c''.\ c'' \otimes_o id\text{-}cblinfun)$›$]$)
using $[[show\text{-}consts]]$
by ($auto\ intro!$: $bij\text{-}comp\ bij\text{-}tensor\text{-}op\text{-}one\text{-}dim\text{-}left\ bij\text{-}tensor\text{-}op\text{-}one\text{-}dim\text{-}right$)

have $c'\text{-}c''$: ‹$c' = ?id1 \otimes_o c'' \otimes_o ?id1$›
unfolding $c''\text{-}def$
apply ($rule\ surj\text{-}f\text{-}inv\text{-}f[$**where** $y{=}c'$, $symmetric]$)
using $*$ **by** ($rule\ bij\text{-}is\text{-}surj$)

define c :: ‹$'a2\ ell2 \Rightarrow_{CL}\ 'b2\ ell2$›
where ‹$c = c'' /_C norm\ \psi /_C norm\ \psi' /_C norm\ \varphi /_C norm\ \varphi'$›

have aux: ‹$assoc\text{-}ell2'\ o_{CL}\ (assoc\text{-}ell2\ o_{CL}\ x\ o_{CL}\ assoc\text{-}ell2')\ o_{CL}\ assoc\text{-}ell2 = x$› **for** x
apply ($simp\ add$: $cblinfun\text{-}assoc\text{-}left$)
by ($simp\ add$: $cblinfun\text{-}assoc\text{-}right$)
have $aux2$: ‹$(assoc\text{-}ell2\ o_{CL}\ ((x \otimes_o y) \otimes_o z)\ o_{CL}\ assoc\text{-}ell2') = x \otimes_o (y \otimes_o z)$› **for** $x\ y\ z$
apply ($rule\ equal\text{-}ket$, $rename\text{-}tac\ xyz$)
apply ($case\text{-}tac\ xyz$, $hypsubst\text{-}thin$)
by ($simp\ flip$: $tensor\text{-}ell2\text{-}ket\ add$: $assoc\text{-}ell2'\text{-}tensor\ assoc\text{-}ell2\text{-}tensor\ tensor\text{-}op\text{-}ell2$)

have ‹$d = (butterfly\ \psi_n\ \psi_n \otimes_o id\text{-}cblinfun)\ o_{CL}\ d\ o_{CL}\ (butterfly\ \psi_n{}'\ \psi_n{}' \otimes_o id\text{-}cblinfun)$›
by ($auto\ simp$: $d\text{-}def\ n1[symmetric]\ comp\text{-}tensor\text{-}op\ cnorm\text{-}eq\text{-}1[THEN\ iffD1]$)
also have ‹$\ldots = (butterfly\ \psi_n\ \psi_n \otimes_o id\text{-}cblinfun)\ o_{CL}\ assoc\text{-}ell2\ o_{CL}\ (b12_n \otimes_o butterfly\ \varphi_n\ \varphi_n{}')$
$o_{CL}\ assoc\text{-}ell2'\ o_{CL}\ (butterfly\ \psi_n{}'\ \psi_n{}' \otimes_o id\text{-}cblinfun)$›
by ($auto\ simp$: $d\text{-}def\ eq\ n2\ cblinfun\text{-}assoc\text{-}left$)
also have ‹$\ldots = (butterfly\ \psi_n\ \psi_n \otimes_o id\text{-}cblinfun)\ o_{CL}\ assoc\text{-}ell2\ o_{CL}$
$((id\text{-}cblinfun \otimes_o butterfly\ \varphi_n\ \varphi_n)\ o_{CL}\ (b12_n \otimes_o butterfly\ \varphi_n\ \varphi_n{}')\ o_{CL}\ (id\text{-}cblinfun \otimes_o butterfly\ \varphi_n{}'\ \varphi_n{}'))$
$o_{CL}\ assoc\text{-}ell2'\ o_{CL}\ (butterfly\ \psi_n{}'\ \psi_n{}' \otimes_o id\text{-}cblinfun)$›
by ($auto\ simp$: $comp\text{-}tensor\text{-}op\ cnorm\text{-}eq\text{-}1[THEN\ iffD1]$)
also have ‹$\ldots = (butterfly\ \psi_n\ \psi_n \otimes_o id\text{-}cblinfun)\ o_{CL}\ assoc\text{-}ell2\ o_{CL}$
$((id\text{-}cblinfun \otimes_o butterfly\ \varphi_n\ \varphi_n)\ o_{CL}\ (assoc\text{-}ell2'\ o_{CL}\ d\ o_{CL}\ assoc\text{-}ell2)\ o_{CL}\ (id\text{-}cblinfun \otimes_o butterfly\ \varphi_n{}'\ \varphi_n{}'))$
$o_{CL}\ assoc\text{-}ell2'\ o_{CL}\ (butterfly\ \psi_n{}'\ \psi_n{}' \otimes_o id\text{-}cblinfun)$›
by ($auto\ simp$: $d\text{-}def\ n2\ eq\ aux$)
also have ‹$\ldots = ((butterfly\ \psi_n\ \psi_n \otimes_o id\text{-}cblinfun)\ o_{CL}\ (assoc\text{-}ell2\ o_{CL}\ (id\text{-}cblinfun \otimes_o butterfly\ \varphi_n\ \varphi_n)\ o_{CL}\ assoc\text{-}ell2'))$
$o_{CL}\ d\ o_{CL}\ ((assoc\text{-}ell2\ o_{CL}\ (id\text{-}cblinfun \otimes_o butterfly\ \varphi_n{}'\ \varphi_n{}')\ o_{CL}\ assoc\text{-}ell2')\ o_{CL}\ (butterfly\ \psi_n{}'\ \psi_n{}' \otimes_o id\text{-}cblinfun))$›
by ($auto\ simp$: $sandwich\text{-}def\ cblinfun\text{-}assoc\text{-}left$)
also have ‹$\ldots = (butterfly\ \psi_n\ \psi_n \otimes_o id\text{-}cblinfun \otimes_o butterfly\ \varphi_n\ \varphi_n)$
$o_{CL}\ d\ o_{CL}\ (butterfly\ \psi_n{}'\ \psi_n{}' \otimes_o id\text{-}cblinfun \otimes_o butterfly\ \varphi_n{}'\ \varphi_n{}')$›
apply ($simp\ only$: $tensor\text{-}id[symmetric]\ comp\text{-}tensor\text{-}op\ aux2$)
by ($simp\ add$: $cnorm\text{-}eq\text{-}1[THEN\ iffD1]$)
also have ‹$\ldots = (vector\text{-}to\text{-}cblinfun\ \psi_n \otimes_o id\text{-}cblinfun \otimes_o vector\text{-}to\text{-}cblinfun\ \varphi_n)$
$o_{CL}\ c'\ o_{CL}\ (vector\text{-}to\text{-}cblinfun\ \psi_n{}' \otimes_o id\text{-}cblinfun \otimes_o vector\text{-}to\text{-}cblinfun\ \varphi_n{}')*$›
apply ($simp\ add$: $c'\text{-}def\ butterfly\text{-}def\text{-}one\text{-}dim[$**where** $'c{=}unit\ ell2]\ cblinfun\text{-}assoc\text{-}left\ comp\text{-}tensor\text{-}op$
$tensor\text{-}op\text{-}adjoint\ cnorm\text{-}eq\text{-}1[THEN\ iffD1]$)
by ($simp\ add$: $cblinfun\text{-}assoc\text{-}right\ comp\text{-}tensor\text{-}op$)
also have ‹$\ldots = butterfly\ \psi_n\ \psi_n{}' \otimes_o c'' \otimes_o butterfly\ \varphi_n\ \varphi_n{}'$›
by ($simp\ add$: $c'\text{-}c''\ comp\text{-}tensor\text{-}op\ tensor\text{-}op\text{-}adjoint\ butterfly\text{-}def\text{-}one\text{-}dim[symmetric]$)
also have ‹$\ldots = butterfly\ \psi\ \psi' \otimes_o c \otimes_o butterfly\ \varphi\ \varphi'$›
by ($simp\ add$: $\psi_n\text{-}def\ \psi_n{}'\text{-}def\ \varphi_n\text{-}def\ \varphi_n{}'\text{-}def\ c\text{-}def\ tensor\text{-}op\text{-}scaleC\text{-}left\ tensor\text{-}op\text{-}scaleC\text{-}right$)
finally have $d\text{-}c$: ‹$d = butterfly\ \psi\ \psi' \otimes_o c \otimes_o butterfly\ \varphi\ \varphi'$›
by $-$
then show $?thesis$
by ($auto\ simp$: $d\text{-}def$)
qed

lemma *norm-tensor-ell2*: ‹*norm* $(a \otimes_s b) = norm\ a * norm\ b$›
apply *transfer*
by (*simp add*: *ell2-norm-finite sum-product sum.cartesian-product case-prod-beta norm-mult power-mult-distrib flip*: *real-sqrt-mult*)

lemma *bounded-cbilinear-tensor-ell2*[*bounded-cbilinear*]: ‹*bounded-cbilinear* $(\otimes_s)$›
proof *standard*
fix $a\ a'$ :: $'a$ *ell2* **and** $b\ b'$ :: $'b$ *ell2* **and** r :: *complex*
show ‹*tensor-ell2* $(a + a')\ b = tensor\text{-}ell2\ a\ b + tensor\text{-}ell2\ a'\ b$›
by (*meson tensor-ell2-add1*)
show ‹*tensor-ell2* $a\ (b + b') = tensor\text{-}ell2\ a\ b + tensor\text{-}ell2\ a\ b'$›
by (*simp add*: *tensor-ell2-add2*)
show ‹*tensor-ell2* $(r *_C a)\ b = r *_C tensor\text{-}ell2\ a\ b$›
by (*simp add*: *tensor-ell2-scaleC1*)
show ‹*tensor-ell2* $a\ (r *_C b) = r *_C tensor\text{-}ell2\ a\ b$›
by (*simp add*: *tensor-ell2-scaleC2*)
show ‹$\exists K.\ \forall a\ b.\ norm\ (tensor\text{-}ell2\ a\ b) \leq norm\ a * norm\ b * K$ ›
apply (*rule exI*[*of* - *1*])
by (*simp add*: *norm-tensor-ell2*)
qed

end

10 Quantum instantiation of registers

theory *Axioms-Quantum*
imports *Jordan-Normal-Form.Matrix-Impl HOL−Library.Rewrite*
Complex-Bounded-Operators.Complex-L2
Finite-Tensor-Product
begin

unbundle *cblinfun-notation*
no-notation *m-inv* (*inv*ı - [*81*] *80*)

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

lemma *preregister-mult-right*: ‹*clinear* $(\lambda a.\ a\ o_{CL}\ z)$›
by (*simp add*: *bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose clinearI*)
lemma *preregister-mult-left*: ‹*clinear* $(\lambda a.\ z\ o_{CL}\ a)$›
by (*meson cbilinear-cblinfun-compose cbilinear-def*)

definition *register* :: ‹($'a$::*finite update* ⇒ $'b$::*finite update*) ⇒ *bool*› **where**
register $F \longleftrightarrow$
clinear F
$\wedge$ F *id-cblinfun* = *id-cblinfun*
$\wedge$ $(\forall a\ b.\ F(a\ o_{CL}\ b) = F\ a\ o_{CL}\ F\ b)$
$\wedge$ $(\forall a.\ F\ (a*) = (F\ a)*)$

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* $F \Longrightarrow$ *clinear* F
unfolding *register-def* **by** *simp*

lemma *register-comp*: *register* $F \Longrightarrow$ *register* $G \Longrightarrow$ *register* $(G \circ F)$
unfolding *register-def*
apply *auto*

using *clinear-compose* **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* (λ*a*. *tensor-op a id-cblinfun*)›
by (*simp add*: *comp-tensor-op register-def tensor-op-cbilinear tensor-op-adjoint*)

lemma *register-tensor-right*: ‹*register* (λ*a*. *tensor-op id-cblinfun a*)›
by (*simp add*: *comp-tensor-op register-def tensor-op-cbilinear tensor-op-adjoint*)

definition *register-pair* ::
‹('*a*::*finite update* ⇒ '*c*::*finite update*) ⇒ ('*b*::*finite update* ⇒ '*c update*)
⇒ (('*a*×'*b*) *update* ⇒ '*c update*)› **where**
‹*register-pair F G* = (*if register F* ∧ *register G* ∧ (∀ *a b*. *F a* o_{CL} *G b* = *G b* o_{CL} *F a*) *then*
tensor-lift (λ*a b*. *F a* o_{CL} *G b*) *else* (λ-. *0*))›

lemma *cbilinear-F-comp-G*[*simp*]: ‹*clinear F* $\Longrightarrow$ *clinear G* $\Longrightarrow$ *cbilinear* (λ*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-pair-apply*:
assumes [*simp*]: ‹*register F*› ‹*register G*›
assumes ‹⋀*a b*. *F a* o_{CL} *G b* = *G b* o_{CL} *F a*›
shows ‹(*register-pair F G*) (*tensor-op a b*) = *F a* o_{CL} *G b*›
unfolding *register-pair-def*
apply (*simp flip*: *assms*(*3*))
by (*metis assms*(*1*) *assms*(*2*) *cbilinear-F-comp-G register-preregister tensor-lift-correct*)

lemma *register-pair-is-register*:
fixes *F* :: ‹'*a*::*finite update* ⇒ '*c*::*finite update*› **and** *G*
assumes [*simp*]: ‹*register F*› **and** [*simp*]: ‹*register G*›
assumes ‹⋀*a b*. *F a* o_{CL} *G b* = *G b* o_{CL} *F a*›
shows ‹*register* (*register-pair F G*)›
proof (*unfold register-def*, *intro conjI allI*)
have [*simp*]: ‹*clinear F*› ‹*clinear G*›
using *assms register-def* **by** *blast*+
have [*simp*]: ‹*F id-cblinfun* = *id-cblinfun*› ‹*G id-cblinfun* = *id-cblinfun*›
using *assms*(*1*,*2*) *register-def* **by** *blast*+
show [*simp*]: ‹*clinear* (*register-pair F G*)›
unfolding *register-pair-def*
using *assms* **apply** *auto*
apply (*rule tensor-lift-clinear*)
by (*simp flip*: *assms*(*3*))
show ‹*register-pair F G id-cblinfun* = *id-cblinfun*›
apply (*simp flip*: *tensor-id*)
apply (*subst register-pair-apply*)
using *assms* **by** *simp-all*
have [*simp*]: ‹*clinear* (λ*y*. *register-pair F G* (*x* o_{CL} *y*))› **for** *x* :: ‹('*a*×'*b*) *update*›
apply (*rule clinear-compose*[*unfolded o-def*, **where** *g*=‹*register-pair F G*›])
by (*simp-all add*: *preregister-mult-left bounded-cbilinear.add-right clinearI*)
have [*simp*]: ‹*clinear* (λ*y*. *x* o_{CL} *register-pair F G y*)› **for** *x* :: ‹'*c update*›
apply (*rule clinear-compose*[*unfolded o-def*, **where** *f*=‹*register-pair F G*›])
by (*simp-all add*: *preregister-mult-left bounded-cbilinear.add-right clinearI*)
have [*simp*]: ‹*clinear* (λ*x*. *register-pair F G* (*x* o_{CL} *y*))› **for** *y* :: ‹('*a*×'*b*) *update*›
apply (*rule clinear-compose*[*unfolded o-def*, **where** *g*=‹*register-pair F G*›])
by (*simp-all add*: *bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose clinearI*)
have [*simp*]: ‹*clinear* (λ*x*. *register-pair F G x* o_{CL} *y*)› **for** *y* :: ‹'*c update*›
apply (*rule clinear-compose*[*unfolded o-def*, **where** *f*=‹*register-pair F G*›])
by (*simp-all add*: *bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose clinearI*)
have [*simp*]: ‹*F* (*x* o_{CL} *y*) = *F x* o_{CL} *F y*› **for** *x y*
by (*simp add*: *register-mult*)

have [*simp*]: ‹*G* (*x* o_{CL} *y*) = *G x* o_{CL} *G y*› **for** *x y*
by (*simp add*: *register-mult*)
have [*simp*]: ‹*clinear* (λ*a*. (*register-pair F G* (*a*∗))∗)›
apply (*rule antilinear-o-antilinear*[*unfolded o-def*, **where** *f*=‹*adj*›])
apply *simp*
apply (*rule antilinear-o-clinear*[*unfolded o-def*, **where** *g*=‹*adj*›])
by (*simp-all*)
have [*simp*]: ‹*F* (*a*∗) = (*F a*)∗› **for** *a*
using *assms*(*1*) *register-def* **by** *blast*
have [*simp*]: ‹*G* (*b*∗) = (*G b*)∗› **for** *b*
using *assms*(*2*) *register-def* **by** *blast*

fix *a b*
show ‹*register-pair F G* (*a* o_{CL} *b*) = *register-pair F G a* o_{CL} *register-pair F G b*›
apply (*rule tensor-extensionality*[*THEN fun-cong*, **where** *x*=*b*], *simp-all*)
apply (*rule tensor-extensionality*[*THEN fun-cong*, **where** *x*=*a*], *simp-all*)
apply (*simp-all add*: *comp-tensor-op register-pair-apply assms*(*3*))
using *assms*(*3*)
by (*metis cblinfun-compose-assoc*)
have ‹(*register-pair F G* (*a*∗))∗ = *register-pair F G a*›
apply (*rule tensor-extensionality*[*THEN fun-cong*, **where** *x*=*a*])
by (*simp-all add*: *tensor-op-adjoint register-pair-apply assms*(*3*))
then show ‹*register-pair F G* (*a*∗) = *register-pair F G a*∗›
by (*metis double-adj*)
qed

end

11 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** ⊗$_u$ *70*)
notation *register-pair* (′(-;-′))

11.1 Elementary facts

declare *complex-vector.linear-id*[*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*]

11.2 Preregisters

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

lemma *preregister-tensor-right*[*simp*]: ‹*clinear* (λa::'a::*finite update*. *tensor-op a b*)›
for *b* :: ‹'b::*finite update*›
proof −
have ‹*clinear* ((λa1::('a×'b) *update*. (*id-cblinfun* ⊗$_u$ *b*) ∗$_u$ *a1*) *o* (λa. *tensor-op a id-cblinfun*))›
by (*rule clinear-compose*, *simp-all*)
then show *?thesis*
by (*simp add*: *o-def comp-tensor-op*)
qed

11.3 Registers

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

11.4 Tensor product of registers

definition *register-tensor* (**infixr** ⊗$_r$ *70*) **where**
register-tensor F G = *register-pair* (λa. *tensor-op* (*F a*) *id-cblinfun*) (λb. *tensor-op id-cblinfun* (*G b*))

lemma *register-tensor-is-register*:
fixes *F* :: 'a::*finite update* ⇒ 'b::*finite update* **and** *G* :: 'c::*finite update* ⇒ 'd::*finite 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::*finite update* ⇒ 'b::*finite update* **and** *G* :: 'c::*finite update* ⇒ 'd::*finite 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::*finite itself*) *A* ⟷
(∀ *F G* :: 'a::*finite update* ⇒ 'b *update*. *clinear F* ⟶ *clinear 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::*finite*) *A* ⟹ *separating TYPE*('a) *B*›
unfolding *separating-def* **by** (*meson in-mono*)

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

lemma *separating-tensor*:
fixes *A* :: ‹'a::*finite update set*› **and** *B* :: ‹'b::*finite update set*›
assumes [*simp*]: ‹*separating TYPE*('c::*finite*) *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 $F\ G :: \langle('a \times 'b)\ update \Rightarrow 'c\ update\rangle$
assume [*simp*]: ‹*clinear F*› ‹*clinear G*›
have [*simp*]: ‹*clinear* $(\lambda x.\ F\ (a \otimes_u x))$› **for** *a*
using - ‹*clinear F*› **apply** (*rule clinear-compose*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*clinear* $(\lambda x.\ G\ (a \otimes_u x))$› **for** *a*
using - ‹*clinear G*› **apply** (*rule clinear-compose*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*clinear* $(\lambda x.\ F\ (x \otimes_u b))$› **for** *b*
using - ‹*clinear F*› **apply** (*rule clinear-compose*[*unfolded o-def*])
by *simp*
have [*simp*]: ‹*clinear* $(\lambda x.\ G\ (x \otimes_u b))$› **for** *b*
using - ‹*clinear G*› **apply** (*rule clinear-compose*[*unfolded o-def*])
by *simp*

assume $\langle\forall x{\in}\{a \otimes_u b \mid a\ b.\ a{\in}A \wedge b{\in}B\}.\ F\ x = G\ x\rangle$
then have *EQ*: $\langle F\ (a \otimes_u b) = G\ (a \otimes_u b)\rangle$ **if** ‹$a \in A$› **and** ‹$b \in B$› **for** *a b*
using *that* **by** *auto*
then have $\langle F\ (a \otimes_u b) = G\ (a \otimes_u b)\rangle$ **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 $\langle F\ (a \otimes_u b) = G\ (a \otimes_u b)\rangle$ **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 $\langle(F \otimes_r G)\ o\ (H \otimes_r L) = (F\ o\ H) \otimes_r (G\ o\ L)\rangle$
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 $A :: \langle'a{::}finite\ update\ set\rangle$ **and** $B :: \langle'b{::}finite\ update\ set\rangle$
assumes ‹*separating TYPE*('*c*::*finite*) *A*›
assumes ‹*separating TYPE*('*c*) *B*›
assumes $\langle C = \{a \otimes_u b \mid a\ b.\ a{\in}A \wedge b{\in}B\}\rangle$
shows ‹*separating TYPE*('*c*) *C*›
using *assms*
by (*simp add*: *separating-tensor*)

lemma *tensor-extensionality3*:
fixes $F\ G :: \langle('a{::}finite \times 'b{::}finite \times 'c{::}finite)\ update \Rightarrow 'd{::}finite\ update\rangle$
assumes [*simp*]: ‹*register F*› ‹*register G*›
assumes $\bigwedge f\ g\ h.\ F\ (f \otimes_u g \otimes_u h) = G\ (f \otimes_u g \otimes_u h)$
shows *F* = *G*
proof (*rule register-eqI*[**where** *A*=$\langle\{a{\otimes_u}b{\otimes_u}c|\ a\ b\ c.\ True\}\rangle$])
have ‹*separating TYPE*('*d*) $\{b \otimes_u c \mid b\ c.\ True\}$›
apply (*rule separating-tensor′*[**where** *A*=*UNIV* **and** *B*=*UNIV*])
by *auto*
then show ‹*separating TYPE*('*d*) $\{a \otimes_u b \otimes_u c \mid a\ b\ c.\ True\}$›
apply (*rule-tac separating-tensor′*[**where** *A*=*UNIV* **and** *B*=$\langle\{b{\otimes_u}c|\ b\ c.\ True\}\rangle$])
by *auto*
show ‹*clinear F*› ‹*clinear G*› **by** *auto*
show $\langle x \in \{a \otimes_u b \otimes_u c \mid a\ b\ c.\ True\} \Longrightarrow F\ x = G\ x\rangle$ **for** *x*
using *assms*(*3*) **by** *auto*
qed

lemma *tensor-extensionality3′*:

fixes $F\ G :: \langle(('a::finite \times 'b::finite) \times 'c::finite)\ update \Rightarrow 'd::finite\ update\rangle$
assumes [*simp*]: ‹*register F*› ‹*register G*›
assumes $\bigwedge f\ g\ h.\ F\ ((f \otimes_u g) \otimes_u h) = G\ ((f \otimes_u g) \otimes_u h)$
shows $F = G$
proof (*rule register-eqI*[**where** $A=\langle\{(a \otimes_u b) \otimes_u c|\ a\ b\ c.\ True\}\rangle$])
have ‹*separating TYPE*('*d*) $\{a \otimes_u b \mid a\ b.\ True\}$›
apply (*rule separating-tensor'*[**where** *A*=*UNIV* **and** *B*=*UNIV*])
by *auto*
then show ‹*separating TYPE*('*d*) $\{(a \otimes_u b) \otimes_u c \mid a\ b\ c.\ True\}$›
apply (*rule-tac separating-tensor'*[**where** *B*=*UNIV* **and** $A=\langle\{a \otimes_u b|\ a\ b.\ True\}\rangle$])
by *auto*
show ‹*clinear F*› ‹*clinear G*› **by** *auto*
show $\langle x \in \{(a \otimes_u b) \otimes_u c \mid a\ b\ c.\ True\} \Longrightarrow F\ x = G\ x\rangle$ **for** *x*
using *assms*(*3*) **by** *auto*
qed

lemma *register-tensor-id*[*simp*]: $\langle id \otimes_r id = id\rangle$
apply (*rule tensor-extensionality*)
by (*auto simp add*: *register-tensor-is-register*)

11.5 Pairs and compatibility

definition *compatible* :: $\langle('a::finite\ update \Rightarrow 'c::finite\ update)$
$\Rightarrow ('b::finite\ update \Rightarrow 'c\ update) \Rightarrow bool\rangle$ **where**
$\langle compatible\ F\ G \longleftrightarrow register\ F \wedge register\ G \wedge (\forall a\ b.\ F\ a *_u G\ b = G\ b *_u F\ a)\rangle$

lemma *compatibleI*:
assumes *register F* **and** *register G*
assumes $\langle\bigwedge a\ b.\ (F\ a) *_u (G\ b) = (G\ b) *_u (F\ a)\rangle$
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 $\langle(F;\ G)\ (a \otimes_u b) = (F\ a) *_u (G\ b)\rangle$
apply (*rule register-pair-apply*)
using *assms* **unfolding** *compatible-def* **by** *metis+*

lemma *register-pair-apply'*:
assumes ‹*compatible F G*›
shows $\langle(F;\ G)\ (a \otimes_u b) = (G\ b) *_u (F\ a)\rangle$
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 clinear-compose*)

lemma *compatible-tensor-id-update-left*[*simp*]:
fixes *F* :: *'a*::*finite update* $\Rightarrow$ *'c*::*finite update* **and** *G* :: *'b*::*finite update* $\Rightarrow$ *'c*::*finite update*
assumes *compatible F G*
shows *compatible* (λa. *id-cblinfun* $\otimes_u$ *F a*) (λa. *id-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*::*finite update* $\Rightarrow$ *'c*::*finite update* **and** *G* :: *'b*::*finite update* $\Rightarrow$ *'c*::*finite update*
assumes *compatible F G*
shows *compatible* (λa. *F a* $\otimes_u$ *id-cblinfun*) (λa. *G a* $\otimes_u$ *id-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* (λa. *F a* $\otimes_u$ *id-cblinfun*) (λa. *id-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* (λa. *id-cblinfun* $\otimes_u$ *F a*) (λa. *G a* $\otimes_u$ *id-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 ‹*clinear* (*F* $\circ$ *G*;*F* $\circ$ *H*)› **and** ‹*clinear* (*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*

11.6 Fst and Snd

definition *Fst* **where** ‹*Fst a = a ⊗u id-cblinfun*›
definition *Snd* **where** ‹*Snd a = id-cblinfun ⊗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*]: ‹*clinear F*› ‹*clinear G*› ‹*clinear H*›
using *register-preregister* **by** *blast+*
have [*simp*]: ‹*clinear* (λ*a*. (*F*;*G*) *a* ∗*u z*)› **for** *z*
apply (*rule clinear-compose*[*unfolded o-def*, *of* ‹(*F*;*G*)›])
by *simp-all*
have [*simp*]: ‹*clinear* (λ*a*. *z* ∗*u* (*F*;*G*) *a*)› **for** *z*
apply (*rule clinear-compose*[*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*]: *clinear A clinear B*

apply (*metis* (*no-types*, *opaque-lifting*) *assms compatible-register1 register-preregister*)
by (*metis* (*full-types*) *assms compatible-register2 register-preregister*)
then show ‹*clinear* ((*A*; *B*) ∘ *swap*)›
by *simp*
show ‹*clinear* (*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

11.7 Compatibility of register tensor products

lemma *compatible-register-tensor*:
fixes *F* :: ‹'*a*::*finite update* ⇒ '*e*::*finite update*› **and** *G* :: ‹'*b*::*finite update* ⇒ '*f*::*finite update*›
and *F'* :: ‹'*c*::*finite update* ⇒ '*e update*› **and** *G'* :: ‹'*d*::*finite update* ⇒ '*f update*›
assumes [*simp*]: ‹*compatible F F'*›
assumes [*simp*]: ‹*compatible G G'*›
shows ‹*compatible* (*F* ⊗*r* *G*) (*F'* ⊗*r* *G'*)›
proof −
note [*intro*!] =
clinear-compose[*OF - preregister-mult-right*, *unfolded o-def*]
clinear-compose[*OF - preregister-mult-left*, *unfolded o-def*]
clinear-compose
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*]: ‹*clinear 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*]: ‹*clinear* (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 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*
unfolding Φ-*def* **apply** (*auto simp*: *register-pair-apply*)
by (*metis assms*(*1*) *assms*(*2*) *compatible-def comp-tensor-op*)
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

11.8 Associativity of the tensor product

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

lemma *assoc-is-hom*[*simp*]: ‹*clinear 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*::*finite*×′*b*::*finite*)×′*c*::*finite*) *update*› **where**
‹*assoc*′ = (*Fst o Fst*; (*Fst o Snd*; *Snd*))›

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

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

11.9 Iso-registers

definition ‹*iso-register F* ⟷ *register F* ∧ (∃ *G*. *register G* ∧ *F o G* = *id* ∧ *G o F* = *id*)›
for *F* :: ‹-::*finite update* ⇒ -::*finite 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*)
show *?thesis*
apply (*rule iso-registerI*[**where** *G*=‹*G' o F'*›])
apply (*auto simp*: *register-tensor-is-register iso-register-is-register register-tensor-distrib*)
apply (*metis* ‹*F* ∘ *F'* = *id*› ‹*G* ∘ *G'* = *id*› *fcomp-assoc fcomp-comp id-fcomp*)
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*)
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* :: ‹-::*finite update* ⇒ -::*finite 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-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*)

11.10 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-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
›

11.11 Notation

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

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

bundle *no-register-notation* **begin**
no-notation *register-tensor* (**infixr** $\otimes_r$ *70*)
no-notation *register-pair* (‴(-;-)‴)
end

end

12 Quantum mechanics basics

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

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

12.1 Basic quantum states

12.1.1 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*

12.1.2 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*)

12.2 Basic quantum gates

12.2.1 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 [*simp*, *code*]: *mat-of-cblinfun pauliX* = *matrix-pauliX*
  apply (*auto simp add*: *pauliX-def matrix-pauliX-def*)

apply (*subst cblinfun-of-mat-inverse*)
by (*auto*)

derive (*eq*) *ceq bit*

instantiation *bit* :: *ccompare* **begin**
definition *CCOMPARE*(*bit*) = *Some* (λ*b1 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*

12.2.2 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 [*simp*, *code*]: *mat-of-cblinfun pauliZ* = *matrix-pauliZ*
apply (*auto simp add*: *pauliZ-def matrix-pauliZ-def*)
apply (*subst cblinfun-of-mat-inverse*)
by (*auto*)
lemma *pauliZ-adjoint*[*simp*]: *pauliZ*∗ = *pauliZ*
by *eval*
lemma *pauliZZ*[*simp*]: *pauliZ* o_{CL} *pauliZ* = *id-cblinfun*
by *eval*

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

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

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

lemma [*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*

12.2.5 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*
apply (*auto simp add*: *Uswap-def matrix-Uswap-def*)
apply (*subst cblinfun-of-mat-inverse*)
by (*auto*)

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* ⊗$_s$ *t* = *t* ⊗$_s$ *s*›
apply (*rule clinear-equal-ket*[**where** *f*=‹λ*s*. *Uswap* ∗$_V$ *s* ⊗$_s$ *t*›, *THEN fun-cong*])
apply (*simp add*: *cblinfun.add-right clinearI tensor-ell2-add1 tensor-ell2-scaleC1*)
apply (*simp add*: *clinear-tensor-ell21*)
apply (*rule clinear-equal-ket*[**where** *f*=‹λ*t*. *Uswap* ∗$_V$ - ⊗$_s$ *t*›, *THEN fun-cong*])
apply (*simp add*: *cblinfun.add-right clinearI tensor-ell2-add2 tensor-ell2-scaleC2*)
apply (*simp add*: *clinear-tensor-ell22*)
apply (*rule basis-enum-eq-vec-of-basis-enumI*)
apply (*simp add*: *mat-of-cblinfun-cblinfun-apply vec-of-basis-enum-ket*)
by (*case-tac i*; *case-tac ia*; *hypsubst-thin*; *normalization*)

end

13 Derived facts about quantum registers

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

no-notation *meet* (**infixl** ⊓$_1$ *70*)
no-notation *Group.mult* (**infixl** ⊗$_1$ *70*)
no-notation *Order.top* (⊤$_1$)
unbundle *register-notation*
unbundle *cblinfun-notation*

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

lemma *register-pair-is-register-converse*:
‹*register* (*F*;*G*) ⟹ *register F*› ‹*register* (*F*;*G*) ⟹ *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 *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*)

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 $\langle((R\ a\ o_{CL}\ S\ b) *_S \top) \leq (R\ a *_S \top) \sqcap (S\ b *_S \top)\rangle$
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 $\langle(R\ a *_S \top) \sqcap (S\ b *_S \top) \leq (R\ a\ o_{CL}\ S\ b) *_S \top\rangle$
proof (*unfold less-eq-ccsubspace.rep-eq*, *rule*)
fix ψ
assume *asm*: $\langle\psi \in$ *space-as-set* $((R\ a *_S \top) \sqcap (S\ b *_S \top))\rangle$
then have $\langle\psi \in$ *space-as-set* $(R\ a *_S \top)\rangle$
by *auto*
then have *R*: $\langle R\ a *_V \psi = \psi\rangle$
using $\langle$*is-Proj* (*R a*)$\rangle$ *cblinfun-fixes-range is-Proj-algebraic* **by** *blast*
from *asm* **have** $\langle\psi \in$ *space-as-set* $(S\ b *_S \top)\rangle$
by *auto*
then have *S*: $\langle S\ b *_V \psi = \psi\rangle$
using $\langle$*is-Proj* (*S b*)$\rangle$ *cblinfun-fixes-range is-Proj-algebraic* **by** *blast*
from *R S* **have** $\langle\psi = (R\ a\ o_{CL}\ S\ b) *_V \psi\rangle$
by (*simp add*: *cblinfun-apply-cblinfun-compose*)
also have $\langle\ldots \in$ *space-as-set* $((R\ a\ o_{CL}\ S\ b) *_S \top)\rangle$
apply *simp* **by** (*metis R S calculation cblinfun-apply-in-image*)
finally show $\langle\psi \in$ *space-as-set* $((R\ a\ o_{CL}\ S\ b) *_S \top)\rangle$
by −
qed
qed

lemma *compatible-proj-mult*:
assumes *compatible R S* **and** *is-Proj a* **and** *is-Proj b*
shows *is-Proj* $(R\ a\ o_{CL}\ S\ b)$
using [[*simproc del*: *Laws-Quantum.compatibility-warn*]]
using *assms* **unfolding** *is-Proj-algebraic compatible-def*
apply *auto*
apply (*metis* (*no-types*, *lifting*) *cblinfun-compose-assoc register-mult*)
by (*simp add*: *assms*(*2*) *assms*(*3*) *is-proj-selfadj register-projector*)

lemma *unitary-sandwich-register*: $\langle$*unitary a* $\Longrightarrow$ *register* (*sandwich a*)$\rangle$
unfolding *register-def*
apply (*auto simp*: *sandwich-def*)
apply (*metis* (*no-types*, *lifting*) *cblinfun-assoc-left*(*1*) *cblinfun-compose-id-right unitaryD1*)

by (*simp add*: *lift-cblinfun-comp*(*2*))

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

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

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

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-def assoc-apply cblinfun-apply-cblinfun-compose tensor-op-ell2 assoc-ell2-tensor assoc-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-def 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-def*)[*2*]
apply (*rule tensor-ell2-extensionality*)
by (*simp add*: *sandwich-def 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-def tensor-op-adjoint unitary-sandwich-register*)

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*

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*›
by (*simp add*: *assms complex-vector.linear-scale*)

lemma *register-bounded-clinear*: ‹*register F* $\Longrightarrow$ *bounded-clinear F*›
using *bounded-clinear-finite-dim register-def* **by** *blast*

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

end

14 Very simple Quantum Hoare logic

theory *QHoare*
imports *Quantum-Extra*
begin

no-notation *Order.top* ($\top_1$)

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

definition *apply U R* = *R U* **for** *R* :: ‹*'a update* ⇒ *'mem update*›
definition *ifthen R x* = *R* (*butterket x 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 $\longleftrightarrow$ ($\forall \psi \in$*space-as-set C*. *program p* $*_V$ ψ $\in$ *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) $*_S$ $\top$

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* $\Longrightarrow$ *hoare D p2 E* $\Longrightarrow$ *hoare C* (*p1*@*p2*) *E*
by (*auto simp*: *program-seq hoare-def*)

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

lemma *hoare-skip*: *C* ≤ *D* $\Longrightarrow$ *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* ≤ *post*
shows *hoare pre* [*apply U R*] *post*
using *assms*
apply (*auto simp*: *hoare-def program-def apply-def*)
by (*metis* (*no-types*, *lifting*) *cblinfun-image.rep-eq closure-subset imageI less-eq-ccsubspace.rep-eq subsetD*)

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*
using *assms*
apply (*auto simp*: *hoare-def program-def ifthen-def butterfly-def*)
by (*metis* (*no-types*, *lifting*) *cblinfun-image.rep-eq closure-subset imageI less-eq-ccsubspace.rep-eq subsetD*)

end

end

15 Tensor products as matrices

theory *Finite-Tensor-Product-Matrices*
imports *Finite-Tensor-Product*
begin

definition *tensor-pack* :: $nat \Rightarrow nat \Rightarrow (nat \times nat) \Rightarrow nat$
where *tensor-pack X Y* = $(\lambda(x, y).\ x * Y + y)$

definition *tensor-unpack* :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow (nat \times nat)$
where *tensor-unpack X Y xy* = $(xy\ div\ Y,\ xy\ mod\ Y)$

lemma *tensor-unpack-inj*:
assumes $i < A * B$ **and** $j < A * B$
shows *tensor-unpack A B i* = *tensor-unpack A B j* $\longleftrightarrow$ $i = j$
by (*metis div-mult-mod-eq prod.sel*(*1*) *prod.sel*(*2*) *tensor-unpack-def*)

lemma *tensor-unpack-bound1*[*simp*]: $i < A * B \Longrightarrow fst\ (tensor\text{-}unpack\ A\ B\ i) < A$
unfolding *tensor-unpack-def*
apply *auto*
using *less-mult-imp-div-less* **by** *blast*
lemma *tensor-unpack-bound2*[*simp*]: $i < A * B \Longrightarrow snd\ (tensor\text{-}unpack\ A\ B\ i) < B$
unfolding *tensor-unpack-def*
apply *auto*
by (*metis mod-less-divisor mult.commute mult-zero-left nat-neq-iff not-less0*)

lemma *tensor-unpack-fstfst*: $\langle fst\ (tensor\text{-}unpack\ A\ B\ (fst\ (tensor\text{-}unpack\ (A * B)\ C\ i)))$
$= fst\ (tensor\text{-}unpack\ A\ (B * C)\ i)\rangle$
unfolding *tensor-unpack-def* **apply** *auto*
by (*metis div-mult2-eq mult.commute*)
lemma *tensor-unpack-sndsnd*: $\langle snd\ (tensor\text{-}unpack\ B\ C\ (snd\ (tensor\text{-}unpack\ A\ (B * C)\ i)))$
$= snd\ (tensor\text{-}unpack\ (A * B)\ C\ i)\rangle$
unfolding *tensor-unpack-def* **apply** *auto*
by (*meson dvd-triv-right mod-mod-cancel*)
lemma *tensor-unpack-fstsnd*: $\langle fst\ (tensor\text{-}unpack\ B\ C\ (snd\ (tensor\text{-}unpack\ A\ (B * C)\ i)))$
$= snd\ (tensor\text{-}unpack\ A\ B\ (fst\ (tensor\text{-}unpack\ (A * B)\ C\ i)))\rangle$
unfolding *tensor-unpack-def* **apply** *auto*
by (*metis* (*no-types*, *lifting*) *Euclidean-Division.div-eq-0-iff add-0-iff bits-mod-div-trivial div-mult-self4 mod-mult2-eq mod-mult-self1-is-0 mult.commute*)

definition *tensor-state-jnf* $\psi\ \varphi$ = (*let d1* = *dim-vec* ψ *in let d2* = *dim-vec* φ *in*
vec (*d1*$*$*d2*) (λ*i*. *let* (*i1*,*i2*) = *tensor-unpack d1 d2 i in* (*vec-index* ψ *i1*) $*$ (*vec-index* φ *i2*)))

lemma *tensor-state-jnf-dim*[*simp*]: $\langle dim\text{-}vec\ (tensor\text{-}state\text{-}jnf\ \psi\ \varphi) = dim\text{-}vec\ \psi * dim\text{-}vec\ \varphi\rangle$
unfolding *tensor-state-jnf-def Let-def* **by** *simp*

lemma *enum-prod-nth-tensor-unpack*:
assumes $\langle i < CARD('a) * CARD('b)\rangle$
shows (*Enum.enum* ! *i* :: *'a*::*enum*$\times$*'b*::*enum*) =

```
      (let (i1,i2) = tensor-unpack CARD('a) CARD('b) i in
          (Enum.enum ! i1, Enum.enum ! i2))
  using assms
  by (simp add: enum-prod-def card-UNIV-length-enum product-nth tensor-unpack-def)

lemma vec-of-basis-enum-tensor-state-index:
  fixes ψ :: ‹'a::enum ell2› and φ :: ‹'b::enum ell2›
  assumes [simp]: ‹i < CARD('a) * CARD('b)›
  shows ‹vec-of-basis-enum (ψ ⊗s φ) $ i = (let (i1,i2) = tensor-unpack CARD('a) CARD('b) i in
    vec-of-basis-enum ψ $ i1 * vec-of-basis-enum φ $ i2)›
proof −
  define i1 i2 where i1 = fst (tensor-unpack CARD('a) CARD('b) i)
    and i2 = snd (tensor-unpack CARD('a) CARD('b) i)
  have [simp]: i1 < CARD('a) i2 < CARD('b)
    using assms i1-def tensor-unpack-bound1 apply presburger
    using assms i2-def tensor-unpack-bound2 by presburger

  have ‹vec-of-basis-enum (ψ ⊗s φ) $ i = Rep-ell2 (ψ ⊗s φ) (enum-class.enum ! i)›
    by (simp add: vec-of-basis-enum-ell2-component)
  also have ‹... = Rep-ell2 ψ (Enum.enum!i1) * Rep-ell2 φ (Enum.enum!i2)›
    apply (transfer fixing: i i1 i2)
    by (simp add: enum-prod-nth-tensor-unpack case-prod-beta i1-def i2-def)
  also have ‹... = vec-of-basis-enum ψ $ i1 * vec-of-basis-enum φ $ i2›
    by (simp add: vec-of-basis-enum-ell2-component)
  finally show ?thesis
    by (simp add: case-prod-beta i1-def i2-def)
qed

lemma vec-of-basis-enum-tensor-state:
  fixes ψ :: ‹'a::enum ell2› and φ :: ‹'b::enum ell2›
  shows ‹vec-of-basis-enum (ψ ⊗s φ) = tensor-state-jnf (vec-of-basis-enum ψ) (vec-of-basis-enum φ)›
  apply (rule eq-vecI, simp-all)
  apply (subst vec-of-basis-enum-tensor-state-index, simp-all)
  by (simp add: tensor-state-jnf-def case-prod-beta Let-def)


lemma mat-of-cblinfun-tensor-op-index:
  fixes a :: ‹'a::enum ell2 ⇒CL 'b::enum ell2› and b :: ‹'c::enum ell2 ⇒CL 'd::enum ell2›
  assumes [simp]: ‹i < CARD('b) * CARD('d)›
  assumes [simp]: ‹j < CARD('a) * CARD('c)›
  shows ‹mat-of-cblinfun (tensor-op a b) $$ (i,j) =
          (let (i1,i2) = tensor-unpack CARD('b) CARD('d) i in
           let (j1,j2) = tensor-unpack CARD('a) CARD('c) j in
               mat-of-cblinfun a $$ (i1,j1) * mat-of-cblinfun b $$ (i2,j2))›
proof −
  define i1 i2 j1 j2
    where i1 = fst (tensor-unpack CARD('b) CARD('d) i)
      and i2 = snd (tensor-unpack CARD('b) CARD('d) i)
      and j1 = fst (tensor-unpack CARD('a) CARD('c) j)
      and j2 = snd (tensor-unpack CARD('a) CARD('c) j)
  have [simp]: i1 < CARD('b) i2 < CARD('d) j1 < CARD('a) j2 < CARD('c)
    using assms i1-def tensor-unpack-bound1 apply presburger
    using assms i2-def tensor-unpack-bound2 apply blast
    using assms(2) j1-def tensor-unpack-bound1 apply blast
    using assms(2) j2-def tensor-unpack-bound2 by presburger

  have ‹mat-of-cblinfun (tensor-op a b) $$ (i,j)
      = Rep-ell2 (tensor-op a b *V ket (Enum.enum!j)) (Enum.enum ! i)›
    by (simp add: mat-of-cblinfun-ell2-component)
  also have ‹... = Rep-ell2 ((a *V ket (Enum.enum!j1)) ⊗s (b *V ket (Enum.enum!j2))) (Enum.enum!i)›
   by (simp add: tensor-op-ell2 enum-prod-nth-tensor-unpack[where i=j] Let-def case-prod-beta j1-def[symmetric]
j2-def[symmetric] flip: tensor-ell2-ket)
  also have ‹... = vec-of-basis-enum ((a *V ket (Enum.enum!j1)) ⊗s b *V ket (Enum.enum!j2)) $ i›
```

```
    by (simp add: vec-of-basis-enum-ell2-component)
  also have ‹... = vec-of-basis-enum (a *V ket (enum-class.enum ! j1)) $ i1 *
                vec-of-basis-enum (b *V ket (enum-class.enum ! j2)) $ i2›
    by (simp add: case-prod-beta vec-of-basis-enum-tensor-state-index i1-def[symmetric] i2-def[symmetric])
  also have ‹... = Rep-ell2 (a *V ket (enum-class.enum ! j1)) (enum-class.enum ! i1) *
                Rep-ell2 (b *V ket (enum-class.enum ! j2)) (enum-class.enum ! i2)›
    by (simp add: vec-of-basis-enum-ell2-component)
  also have ‹... = mat-of-cblinfun a $$ (i1, j1) * mat-of-cblinfun b $$ (i2, j2)›
    by (simp add: mat-of-cblinfun-ell2-component)
  finally show ?thesis
    by (simp add: i1-def[symmetric] i2-def[symmetric] j1-def[symmetric] j2-def[symmetric] case-prod-beta)
qed


definition tensor-op-jnf A B =
  (let r1 = dim-row A in
   let c1 = dim-col A in
   let r2 = dim-row B in
   let c2 = dim-col B in
   mat (r1*r2) (c1*c2)
   (λ(i,j). let (i1,i2) = tensor-unpack r1 r2 i in
            let (j1,j2) = tensor-unpack c1 c2 j in
              (A $$ (i1,j1)) * (B $$ (i2,j2))))

lemma tensor-op-jnf-dim[simp]:
  ‹dim-row (tensor-op-jnf a b) = dim-row a * dim-row b›
  ‹dim-col (tensor-op-jnf a b) = dim-col a * dim-col b›
  unfolding tensor-op-jnf-def Let-def by simp-all


lemma mat-of-cblinfun-tensor-op:
  fixes a :: ‹'a::enum ell2 ⇒CL 'b::enum ell2› and b :: ‹'c::enum ell2 ⇒CL 'd::enum ell2›
  shows ‹mat-of-cblinfun (tensor-op a b) = tensor-op-jnf (mat-of-cblinfun a) (mat-of-cblinfun b)›
  apply (rule eq-matI, simp-all add: )
  apply (subst mat-of-cblinfun-tensor-op-index, simp-all)
  by (simp add: tensor-op-jnf-def case-prod-beta Let-def)


lemma mat-of-cblinfun-assoc-ell2'[simp]:
 ‹mat-of-cblinfun (assoc-ell2' :: (('a::enum×('b::enum×'c::enum)) ell2 ⇒CL -)) = one-mat (CARD('a)*CARD('b)*CARD('c))›
  (is mat-of-cblinfun ?assoc = -)
proof (rule mat-eq-iff[THEN iffD2], intro conjI allI impI)

  show ‹dim-row (mat-of-cblinfun ?assoc) =
    dim-row (1m (CARD('a) * CARD('b) * CARD('c)))›
    by (simp)
  show ‹dim-col (mat-of-cblinfun ?assoc) =
    dim-col (1m (CARD('a) * CARD('b) * CARD('c)))›
    by (simp)

  fix i j
  let ?i = Enum.enum ! i :: (('a×'b)×'c) and ?j = Enum.enum ! j :: ('a×('b×'c))

  assume ‹i < dim-row (1m (CARD('a) * CARD('b) * CARD('c)))›
  then have iB[simp]: ‹i < CARD('a) * CARD('b) * CARD('c)› by simp
  then have iB'[simp]: ‹i < CARD('a) * (CARD('b) * CARD('c))› by linarith
  assume ‹j < dim-col (1m (CARD('a) * CARD('b) * CARD('c)))›
  then have jB[simp]: ‹j < CARD('a) * CARD('b) * CARD('c)› by simp
  then have jB'[simp]: ‹j < CARD('a) * (CARD('b) * CARD('c))› by linarith

  define i1 i23 i2 i3
    where i1 = fst (tensor-unpack CARD('a) (CARD('b)*CARD('c)) i)
      and i23 = snd (tensor-unpack CARD('a) (CARD('b)*CARD('c)) i)
```

```
    and i2 = fst (tensor-unpack CARD('b) CARD('c) i23)
    and i3 = snd (tensor-unpack CARD('b) CARD('c) i23)
  define j12 j1 j2 j3
   where j12 = fst (tensor-unpack (CARD('a)*CARD('b)) CARD('c) j)
    and j1 = fst (tensor-unpack CARD('a) CARD('b) j12)
    and j2 = snd (tensor-unpack CARD('a) CARD('b) j12)
    and j3 = snd (tensor-unpack (CARD('a)*CARD('b)) CARD('c) j)

  have [simp]: j12 < CARD('a)*CARD('b) i23 < CARD('b)*CARD('c)
   using j12-def jB tensor-unpack-bound1 apply presburger
   using i23-def iB' tensor-unpack-bound2 by blast

  have j1': ‹fst (tensor-unpack CARD('a) (CARD('b) * CARD('c)) j) = j1›
   by (simp add: j1-def j12-def tensor-unpack-fstfst)

  let ?i1 = Enum.enum ! i1 :: 'a and ?i2 = Enum.enum ! i2 :: 'b and ?i3 = Enum.enum ! i3 :: 'c
  let ?j1 = Enum.enum ! j1 :: 'a and ?j2 = Enum.enum ! j2 :: 'b and ?j3 = Enum.enum ! j3 :: 'c

  have i: ‹?i = ((?i1,?i2),?i3)›
   by (auto simp add: enum-prod-nth-tensor-unpack case-prod-beta
       tensor-unpack-fstfst tensor-unpack-fstsnd tensor-unpack-sndsnd i1-def i2-def i23-def i3-def)
  have j: ‹?j = (?j1,(?j2,?j3))›
   by (auto simp add: enum-prod-nth-tensor-unpack case-prod-beta
      tensor-unpack-fstfst tensor-unpack-fstsnd tensor-unpack-sndsnd j1-def j2-def j12-def j3-def)
  have ijeq: ‹(?i1,?i2,?i3) = (?j1,?j2,?j3) ⟷ i = j›
   unfolding i1-def i2-def i3-def j1-def j2-def j3-def apply simp
   apply (subst enum-inj, simp, simp)
   apply (subst enum-inj, simp, simp)
   apply (subst enum-inj, simp, simp)
  apply (subst tensor-unpack-inj[symmetric, where i=i and j=j and A=CARD('a) and B=CARD('b)*CARD('c)],
simp, simp)
   unfolding prod-eq-iff
  apply (subst tensor-unpack-inj[symmetric, where i=‹snd (tensor-unpack CARD('a) (CARD('b) * CARD('c))
i)› and A=CARD('b) and B=CARD('c)], simp, simp)
    by (simp add: i1-def[symmetric] j1-def[symmetric] i2-def[symmetric] j2-def[symmetric] i3-def[symmetric]
j3-def[symmetric]
       i23-def[symmetric] j12-def[symmetric] j1'
       prod-eq-iff tensor-unpack-fstsnd tensor-unpack-sndsnd)

  have ‹mat-of-cblinfun ?assoc $$ (i, j) = Rep-ell2 (assoc-ell2' *V ket ?j) ?i›
   by (subst mat-of-cblinfun-ell2-component, auto)
  also have ‹... = Rep-ell2 ((ket ?j1 ⊗s ket ?j2) ⊗s ket ?j3) ?i›
   by (simp add: j assoc-ell2'-tensor flip: tensor-ell2-ket)
  also have ‹... = (if (?i1,?i2,?i3) = (?j1,?j2,?j3) then 1 else 0)›
   by (auto simp add: ket.rep-eq i)
  also have ‹... = (if i=j then 1 else 0)›
   using ijeq by simp
  finally
  show ‹mat-of-cblinfun ?assoc $$ (i, j) =
        1m (CARD('a) * CARD('b) * CARD('c)) $$ (i, j)›
   by auto
qed

lemma assoc-ell2'-inv: assoc-ell2 oCL assoc-ell2' = id-cblinfun
  apply (rule equal-ket, case-tac x, hypsubst)
  by (simp flip: tensor-ell2-ket add: cblinfun-apply-cblinfun-compose assoc-ell2'-tensor assoc-ell2-tensor)

lemma assoc-ell2-inv: assoc-ell2' oCL assoc-ell2 = id-cblinfun
  apply (rule equal-ket, case-tac x, hypsubst)
  by (simp flip: tensor-ell2-ket add: cblinfun-apply-cblinfun-compose assoc-ell2'-tensor assoc-ell2-tensor)

lemma mat-of-cblinfun-assoc-ell2[simp]:
 ‹mat-of-cblinfun (assoc-ell2 :: (('a::enum×'b::enum)×'c::enum) ell2 ⇒CL -)) = one-mat (CARD('a)*CARD('b)*CARD('c))›
```

(**is** *mat-of-cblinfun ?assoc* = -)
proof −
let *?assoc′* = *assoc-ell2′* :: ((′*a*::*enum*×(′*b*::*enum*×′*c*::*enum*)) *ell2* ⇒$_{CL}$ -)
have *one-mat* (*CARD*(′*a*)∗*CARD*(′*b*)∗*CARD*(′*c*)) = *mat-of-cblinfun* (*?assoc* o_{CL} *?assoc′*)
by (*simp add*: *mult.assoc mat-of-cblinfun-id*)
also have ‹. . . = *mat-of-cblinfun ?assoc* ∗ *mat-of-cblinfun ?assoc′*›
using *mat-of-cblinfun-compose* **by** *blast*
also have ‹. . . = *mat-of-cblinfun ?assoc* ∗ *one-mat* (*CARD*(′*a*)∗*CARD*(′*b*)∗*CARD*(′*c*))›
by *simp*
also have ‹. . . = *mat-of-cblinfun ?assoc*›
apply (*rule right-mult-one-mat′*)
by (*simp*)
finally show *?thesis*
by *simp*
qed

end

16 Quantum teleportation

theory *Teleport*
imports
QHoare
Real-Impl.Real-Impl
HOL−Library.Code-Target-Numeral
Finite-Tensor-Product-Matrices
HOL−Library.Word
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** ⊗$_1$ *70*)
no-notation *Order.top* (⊤$_1$)
unbundle *no-vec-syntax*
unbundle *no-inner-syntax*

locale *teleport-locale* = *qhoare TYPE*(′*mem*::*finite*) +
fixes *X* :: *bit update* ⇒ ′*mem*::*finite update*
and Φ :: (*bit*∗*bit*) *update* ⇒ ′*mem update*
and *A* :: ′*atype*::*finite update* ⇒ ′*mem update*
and *B* :: ′*btype*::*finite update* ⇒ ′*mem update*
assumes *compat*[*register*]: *mutually compatible* (*X*,Φ,*A*,*B*)
begin

abbreviation Φ*1* ≡ Φ ∘ *Fst*
abbreviation Φ*2* ≡ Φ ∘ *Snd*
abbreviation *X*Φ*2* ≡ (*X*;Φ*2*)
abbreviation *X*Φ*1* ≡ (*X*;Φ*1*)
abbreviation *X*Φ ≡ (*X*;Φ)
abbreviation *XAB* ≡ ((*X*;*A*); *B*)
abbreviation *AB* ≡ (*A*;*B*)
abbreviation Φ*2AB* ≡ ((Φ *o Snd*; *A*); *B*)

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

```
    apply (if b=1 then pauliZ else id-cblinfun) Φ2
  ]


lemma Φ-XΦ: ‹Φ a = XΦ (id-cblinfun ⊗o a)›
  by (auto simp: register-pair-apply)
lemma XΦ1-XΦ: ‹XΦ1 a = XΦ (assoc (a ⊗o id-cblinfun))›
  apply (subst pair-o-assoc[unfolded o-def, of X Φ1 Φ2, simplified, THEN fun-cong])
  by (auto simp: register-pair-apply)
lemma XΦ2-XΦ: ‹XΦ2 a = XΦ ((id ⊗r swap) (assoc (a ⊗o id-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 Φ2-XΦ: ‹Φ2 a = XΦ (id-cblinfun ⊗o (id-cblinfun ⊗o a))›
  by (auto simp: Snd-def register-pair-apply)
lemma X-XΦ: ‹X a = XΦ (a ⊗o id-cblinfun)›
  by (auto simp: register-pair-apply)
lemma Φ1-XΦ: ‹Φ1 a = XΦ (id-cblinfun ⊗o (a ⊗o id-cblinfun))›
  by (auto simp: Fst-def register-pair-apply)
lemmas to-XΦ = Φ-XΦ XΦ1-XΦ XΦ2-XΦ Φ2-XΦ X-XΦ Φ1-XΦ

lemma X-XΦ1: ‹X a = XΦ1 (a ⊗o id-cblinfun)›
  by (auto simp: register-pair-apply)
lemmas to-XΦ1 = X-XΦ1

lemma XAB-to-XΦ2-AB: ‹XAB a = (XΦ2;AB) ((swap ⊗r id) (assoc' (id-cblinfun ⊗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 XΦ2-to-XΦ2-AB: ‹XΦ2 a = (XΦ2;AB) (a ⊗o id-cblinfun)›
  by (simp add: register-pair-apply)

schematic-goal Φ2AB-to-XΦ2-AB: Φ2AB a = (XΦ2;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-XΦ2-AB = XAB-to-XΦ2-AB XΦ2-to-XΦ2-AB Φ2AB-to-XΦ2-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* o_{CL} *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* o_{CL} *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* (*selfbutterket a*) o_{CL} *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* (*selfbutterket b*) o_{CL} *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*) o_{CL} *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*) o_{CL} *O6*›
have *O7*: ‹*O7* = *Φ2 XZ* o_{CL} *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**) o_{CL} *XΦ2 Uswap* o_{CL} Φ (*butterfly* (*ket a* $\otimes_s$ *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*]
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* **apply** *auto*
apply (*metis cblinfun-assoc-left*(*1*) *pauliXX pauliZZ cblinfun-compose-id-left*)
by (*metis cblinfun-assoc-left*(*1*) *pauliXX pauliZZ cblinfun-compose-id-left*)

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)
    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 ⊗_o id-cblinfun) *_S (XΦ2;AB)
                   ((swap ⊗_r id) (assoc' (id-cblinfun ⊗_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-def 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 ⊗_o id-cblinfun ⊗_o id-cblinfun ⊗_o id-cblinfun ⊗_o id-cblinfun›
definition X :: ‹bit var› where ‹X a = id-cblinfun ⊗_o a ⊗_o id-cblinfun ⊗_o id-cblinfun ⊗_o id-cblinfun›
definition Φ1 :: ‹bit var› where ‹Φ1 a = id-cblinfun ⊗_o id-cblinfun ⊗_o a ⊗_o id-cblinfun ⊗_o id-cblinfun›
definition B :: ‹b-state var› where ‹B a = id-cblinfun ⊗_o id-cblinfun ⊗_o id-cblinfun ⊗_o a ⊗_o id-cblinfun›
definition Φ2 :: ‹bit var› where ‹Φ2 a = id-cblinfun ⊗_o id-cblinfun ⊗_o id-cblinfun ⊗_o id-cblinfun ⊗_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

end
```

17 Quantum instantiation of complements

theory *Axioms-Complement-Quantum*
imports *Laws-Quantum Finite-Tensor-Product Quantum-Extra*
begin

no-notation *m-inv* (*inv*ı - [*81*] *80*)
no-notation *Lattice.join* (**infixl** ⊔ı *65*)

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

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

lemma *CARD-complement-domain*:
assumes ‹*CARD*('*b*::*finite*) = *n* ∗ *CARD*('*a*::*finite*)›
shows ‹*CARD*(('*a*,'*b*) *complement-domain*) = *n*›
proof −
from *assms* **have** ‹*n* > *0*›
by (*metis zero-less-card-finite zero-less-mult-pos2*)
have ∗: ‹*inj Rep-complement-domain*›
by (*simp add*: *Rep-complement-domain-inject inj-on-def*)
moreover have ‹*card* (*range* (*Rep-complement-domain* :: ('*a*,'*b*) *complement-domain* ⇒ -)) = *n*›
apply (*subst type-definition.Rep-range*[*OF type-definition-complement-domain*])
using *assms* ‹*n* > *0*› **by** *simp*
ultimately show *?thesis*
by (*metis card-image*)
qed

lemma *register-decomposition*:
fixes Φ :: ‹'*a*::*finite update* ⇒ '*b*::*finite update*›
assumes [*simp*]: ‹*register* Φ›
shows ‹∃ *U* :: ('*a* × ('*a*, '*b*) *complement-domain*) *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 *ξ0* :: '*a*

have [*simp*]: ‹*clinear* Φ›
by *simp*

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

define *P'* **where** ‹*P' i* = Φ (*P i*)› **for** *i* :: '*a*
have *proj-P'*: ‹*is-Proj* (*P' i*)› **for** *i*
by (*simp add*: *P-def P'-def register-projector*)
have ‹(∑ *i*∈*UNIV*. *P i*) = *id-cblinfun*›
using *sum-butterfly-ket P-butter* **by** *simp*
then have *sumP'id*: ‹(∑ *i*∈*UNIV*. *P' i*) = *id-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 ‹... = *cinner* ($P'\ j *_V P'\ i *_V x'$) y'›
by (*metis cinner-adj-left is-Proj-algebraic proj-P'*)
also have ‹... = *cinner* (Φ ($P\ j\ o_{CL}\ P\ i$) $*_V x'$) y'›
unfolding *P'-def register-mult*[*OF* ‹*register* Φ›, *symmetric*] **by** *simp*
also have ‹... = *cinner* (Φ (*selfbutterket j* o_{CL} *selfbutterket i*) $*_V x'$) y'›
unfolding *P-butter* **by** *simp*
also have ‹... = *cinner* ($\Phi\ 0 *_V x'$) y'›
by (*metis butterfly-comp-butterfly complex-vector.scale-eq-0-iff orthogonal-ket that*(*3*))
also have ‹... = *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 ‹$(\sum i{\in}UNIV.\ P'\ i)$ = *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 ‹$(\sum i{\in}insert\ j\ M.\ P'\ i) = P'\ j + (\sum i{\in}M.\ P'\ i)$›
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
  apply auto by (metis cinner-eq-zero-iff that)

have cardBsame: ‹card (B i) = card (B j)› for i j
proof −
  define Si-to-Sj where ‹Si-to-Sj i j ψ = Φ (butterket j 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 apply (auto simp: bij-betw-def)
    by (metis complex-vector.span-span cspanB)
  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 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 (butterket ξ ξ0) *V f α)› for ξ :: 'a and α :: ‹('a,'b) complement-domain›
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›
define eqac where ‹eqac a b = (if a = b then 1 else 0 :: complex)› for a b :: ‹'a * ('a,'b) complement-domain›

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 (butterket ξ ξ0) *V f α) (Φ (butterket ξ' ξ0)
*V f α')›
    unfolding Uapply u-def by simp
  also have ‹... = cinner ((Φ (butterket ξ' ξ0))* *V Φ (butterket ξ ξ0) *V f α) (f α')›
    by (simp add: cinner-adj-left)
  also have ‹... = cinner (Φ (butterket ξ' ξ0 *) *V Φ (butterket ξ ξ0) *V f α) (f α')›
    by (metis (no-types, lifting) assms register-def)
  also have ‹... = cinner (Φ (butterket ξ0 ξ' oCL butterket ξ ξ0) *V f α) (f α')›
    by (simp add: register-mult cblinfun-apply-cblinfun-compose[symmetric])
  also have ‹... = cinner (Φ (eqa ξ' ξ *C selfbutterket ξ0) *V f α) (f α')›
    apply simp by (metis eqa-def cinner-ket)
  also have ‹... = eqa ξ' ξ * cinner (Φ (selfbutterket 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'›
    using bij-f orthoB normalB unfolding is-ortho-set-def eqc-def apply auto
     apply (metis bij-betw-imp-surj-on cnorm-eq-1 rangeI)
    by (smt (z3) bij-betw-iff-bijections iso-tuple-UNIV-I)
  finally show ?thesis
    by (simp add: eqa-def eqac-def eqc-def ξ'α' ξα)
qed
then have ‹isometry U›
  apply (rule-tac orthogonal-on-basis-is-isometry[where B=‹range ket›])
  using eqac-def by auto

have ‹U* oCL Φ (butterket ξ η) oCL U = butterket ξ 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⟩ where ξ1α: ⟨ξ1α = (ξ1,α)⟩
      apply atomize-elim by auto
    have ⟨(U* o_CL Φ (butterket ξ η) o_CL U) *_V ket ξ1α = U* *_V Φ (butterket ξ η) *_V Φ (butterket ξ1 ξ0) *_V
f α⟩
      unfolding cblinfun-apply-cblinfun-compose ξ1α Uapply u-def by simp
    also have ⟨... = U* *_V Φ (butterket ξ η o_CL butterket ξ1 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 butterket ξ ξ0) *_V f α⟩
      by (simp add: eqa-def cinner-ket)
    also have ⟨... = eqa η ξ1 *_C U* *_V Φ (butterket ξ 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 ⟨... = (butterket ξ η *_V ket ξ1) ⊗_s ket α⟩
      by (simp add: eqa-def tensor-ell2-scaleC1)
    also have ⟨... = (butterket ξ η ⊗_o id-cblinfun) *_V ket ξ1α⟩
      by (simp add: ξ1α tensor-op-ket)
    finally show ⟨(U* o_CL Φ (butterket ξ η) o_CL U) *_V ket ξ1α = (butterket ξ η ⊗_o id-cblinfun) *_V ket ξ1α⟩
      by −
  qed
  then have 1: ⟨U* o_CL Φ ϑ o_CL U = ϑ ⊗_o id-cblinfun⟩ for ϑ
    apply (rule-tac clinear-eq-butterfly-ketI[THEN fun-cong, where x=ϑ])
   by (auto intro!: clinearI simp add: bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose complex-vector.linear-add
complex-vector.linear-scale)

  have ⟨unitary U⟩
  proof −
    have ⟨Φ (butterket ξ ξ1) *_S ⊤ ≤ U *_S ⊤⟩ for ξ ξ1
    proof −
      have *: ⟨Φ (butterket ξ ξ0) *_V b ∈ space-as-set (U *_S ⊤)⟩ if ⟨b ∈ B ξ0⟩ for b
        apply (subst asm-rl[of ⟨Φ (butterket ξ 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 (butterket ξ ξ1) *_S ⊤ = Φ (butterket ξ ξ0) *_S Φ (butterket ξ0 ξ0) *_S Φ (butterket ξ0 ξ1) *_S ⊤⟩
        unfolding cblinfun-compose-image[symmetric] register-mult[OF assms]
        by simp
      also have ⟨... ≤ Φ (butterket ξ ξ0) *_S Φ (butterket ξ0 ξ0) *_S ⊤⟩
        by (meson cblinfun-image-mono top-greatest)
      also have ⟨... = Φ (butterket ξ ξ0) *_S S ξ0⟩
        by (simp add: S-def P'-def P-butter)
      also have ⟨... = Φ (butterket ξ ξ0) *_S ccspan (B ξ0)⟩
        by (simp add: ccspanB)
      also have ⟨... = ccspan (Φ (butterket ξ 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 (selfbutterket 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 ⊤⟩
      apply auto by (meson SUP-le-iff order.trans)
    then have ⟨U *_S ⊤ = ⊤⟩
      apply auto
      using top.extremum-unique by blast
    with ⟨isometry U⟩ show ⟨unitary U⟩
      by (rule surj-isometry-is-unitary)
  qed
```

```
  have ‹Φ ϑ = U o_CL (ϑ ⊗o id-cblinfun) o_CL U*› for ϑ
  proof −
    from ‹unitary U›
    have ‹Φ ϑ = (U o_CL U*) o_CL Φ ϑ o_CL (U o_CL U*)›
      by simp
    also have ‹... = U o_CL (U*  o_CL Φ ϑ o_CL U) o_CL 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 ?thesis
    by (auto simp: sandwich-def)
qed

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


lemma register-inj: ‹inj F› if ‹register F›
proof −
  obtain U :: ‹('a × ('a, 'b) complement-domain) ell2 ⇒_CL 'b ell2›
    where ‹unitary U› and F: ‹F a = sandwich U (a ⊗o id-cblinfun)› for a
    apply atomize-elim using ‹register F› by (rule register-decomposition)
  have ‹inj (sandwich U)›
     by (smt (verit, best) ‹unitary U› cblinfun-assoc-left inj-onI sandwich-def cblinfun-compose-id-right cblin-
fun-compose-id-left unitary-def)
  moreover have ‹inj (λa::'a::finite ell2 ⇒_CL -. a ⊗o id-cblinfun)›
    by (rule inj-tensor-left, simp)
  ultimately show ‹inj F›
    unfolding F
    by (smt (z3) inj-def)
qed

lemma iso-register-decomposition:
  assumes [simp]: ‹iso-register F›
  shows ‹∃ U. unitary U ∧ F = sandwich U›
proof −
  have [simp]: ‹register F›
    using assms iso-register-is-register by blast

  let ?ida = ‹id-cblinfun :: ('a, 'b) complement-domain ell2 ⇒_CL -›

  from register-decomposition[OF ‹register F›]
  obtain V :: ‹('a × ('a, 'b) complement-domain) ell2 ⇒_CL 'b ell2› where ‹unitary V›
    and FV: ‹F ϑ = sandwich V (ϑ ⊗o ?ida)› for ϑ
    by auto

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

then have ‹$a \neq 0$›
apply *auto*
by (*metis butterfly-apply cblinfun.zero-left complex-vector.scale-eq-0-iff ket-nonzero orthogonal-ket*)

obtain γ **where** γ: ‹*?ida* = γ $*_C$ *selfbutterket undefined*›
apply *atomize-elim*
using *a* ‹$a \neq 0$› **by** (*rule tensor-op-almost-injective*)
then have ‹*?ida* (*ket undefined*) = γ $*_C$ (*selfbutterket undefined* $*_V$ *ket undefined*)›
by (*simp add*: ‹*id-cblinfun* = γ $*_C$ *selfbutterket undefined*› *scaleC-cblinfun.rep-eq*)
then have ‹*ket undefined* = γ $*_C$ *ket undefined*›
by (*metis butterfly-apply cinner-scaleC-right id-cblinfun-apply cinner-ket-same mult.right-neutral 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-finite-dim clinear-tensor-ell22*)
have *sandwich-T*: ‹*sandwich T a* = *a* $\otimes_o$ *?ida*› **for** *a*
apply (*rule fun-cong*[**where** *x*=*a*])
apply (*rule clinear-eq-butterfly-ketI*)
apply *auto*
by (*metis* (*no-types*, *opaque-lifting*) *Misc.sandwich-def T* γ ‹γ = *1*› *adj-cblinfun-compose butterfly-adjoint cblinfun-comp-butterfly scaleC-one tensor-butterfly*)

have ‹*F* (*butterfly x y*) = *V* o_{CL} (*butterfly x y* $\otimes_o$ *?ida*) o_{CL} *V**› **for** *x y*
by (*simp add*: *Misc.sandwich-def 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*›)
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-tac clinear-eq-butterfly-ketI*)
apply *auto*
by (*metis* (*no-types*, *lifting*) *cblinfun-apply-clinear clinear-iff sandwich-apply*)

have ‹*isometry T*›
apply (*rule orthogonal-on-basis-is-isometry*[**where** *B*=‹*range ket*›])
by (*auto simp*: *T*)
moreover have ‹*T* $*_S$ ⊤ = ⊤›
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-scaleC2*)
also have ‹... = φ $\otimes_s$ (*selfbutterket 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 ‹⊤ ≤ *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 ⊤›
  by (metis (mono-tags, opaque-lifting) calculation cblinfun-image-ccspan cblinfun-image-mono eq-iff top-greatest)
  finally show ‹T *S ⊤ = ⊤›
    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 ?thesis
  by (auto simp: sandwich-def[abs-def])
qed

lemma complement-exists:
  fixes F :: ‹'a::finite update ⇒ 'b::finite update›
  assumes ‹register F›
  shows ‹∃ G :: ('a, 'b) complement-domain 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) 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)
  define G :: ‹(('a, 'b) complement-domain) 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)›
      apply (auto simp: F G-def sandwich-def)
    by (metis (no-types, lifting) ‹unitary U› isometryD cblinfun-assoc-left(1) comp-tensor-op cblinfun-compose-id-right
cblinfun-compose-id-left unitary-isometry)
    moreover have ‹G b oCL F a = sandwich U (a ⊗o b)›
      apply (auto simp: F G-def sandwich-def)
    by (metis (no-types, lifting) ‹unitary U› isometryD cblinfun-assoc-left(1) comp-tensor-op cblinfun-compose-id-right
cblinfun-compose-id-left unitary-isometry)
    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
      apply (auto simp: register-pair-apply F G-def sandwich-def)
    by (metis (no-types, lifting) ‹unitary U› isometryD cblinfun-assoc-left(1) comp-tensor-op cblinfun-compose-id-right
cblinfun-compose-id-left unitary-isometry)
    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)
    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›
      apply (auto intro!:ext simp: I-def[abs-def] FG sandwich-def)
     apply (metis (no-types, opaque-lifting) ‹unitary U› isometryD cblinfun-assoc-left(1) cblinfun-compose-id-right
cblinfun-compose-id-left unitary-isometry)
    by (metis (no-types, lifting) ‹unitary U› cblinfun-assoc-left(1) cblinfun-compose-id-left cblinfun-compose-id-right
```

unitaryD2)
  then show ‹*iso-register* (*F*;*G*)›
    by (*auto intro*!: *iso-registerI*)
 qed
 ultimately show *?thesis*
  apply (*rule-tac exI*[*of* - *G*]) **by** (*auto*)
qed

definition ‹*commutant F* = {*x*. ∀ *y*∈*F*. *x* o_{CL} *y* = *y* o_{CL} *x*}›

lemma *commutant-exchange*:
 fixes *F* :: ‹'*a*::*finite update* ⇒ '*b*::*finite update*›
 assumes ‹*iso-register F*›
 shows ‹*commutant* (*F* ‘ *X*) = *F* ‘ *commutant X*›
proof (*rule Set.set-eqI*)
 fix *x* :: ‹'*b update*›
 from *assms*
 obtain *G* **where** ‹*F o G* = *id*› **and** ‹*G o F* = *id*› **and** [*simp*]: ‹*register G*›
  using *iso-register-def* **by** *blast*
 from *assms* **have** [*simp*]: ‹*register F*›
  using *iso-register-def* **by** *blast*
 have ‹*x* ∈ *commutant* (*F* ‘ *X*) ⟷ (∀ *y* ∈ *F* ‘ *X*. *x* o_{CL} *y* = *y* o_{CL} *x*)›
  by (*simp add*: *commutant-def*)
 also have ‹... ⟷ (∀ *y* ∈ *F* ‘ *X*. *G x* o_{CL} *G y* = *G y* o_{CL} *G x*)›
  by (*metis* (*no-types*, *opaque-lifting*) ‹*F* ∘ *G* = *id*› ‹*G o F* = *id*› ‹*register G*› *comp-def eq-id-iff register-def*)
 also have ‹... ⟷ (∀ *y* ∈ *X*. *G x* o_{CL} *y* = *y* o_{CL} *G x*)›
  by (*simp add*: ‹*G* ∘ *F* = *id*› *pointfree-idE*)
 also have ‹... ⟷ *G x* ∈ *commutant X*›
  by (*simp add*: *commutant-def*)
 also have ‹... ⟷ *x* ∈ *F* ‘ *commutant X*›
  by (*metis* (*no-types*, *opaque-lifting*) ‹*G* ∘ *F* = *id*› ‹*F* ∘ *G* = *id*› *image-iff pointfree-idE*)
 finally show ‹*x* ∈ *commutant* (*F* ‘ *X*) ⟷ *x* ∈ *F* ‘ *commutant X*›
  by −
qed

lemma *commutant-tensor1*: ‹*commutant* (*range* (λ*a*. *a* ⊗$_o$ *id-cblinfun*)) = *range* (λ*b*. *id-cblinfun* ⊗$_o$ *b*)›
proof (*rule Set.set-eqI*, *rule iffI*)
 fix *x* :: ‹('*a* × '*b*) *ell2* ⇒$_{CL}$ ('*a* × '*b*) *ell2*›
 fix γ :: '*a*
 assume ‹*x* ∈ *commutant* (*range* (λ*a*. *a* ⊗$_o$ *id-cblinfun*))›
 then have *comm*: ‹(*a* ⊗$_o$ *id-cblinfun*) *∗$_V$ x ∗$_V$* ψ = *x ∗$_V$* (*a* ⊗$_o$ *id-cblinfun*) *∗$_V$* ψ› **for** *a* ψ
  by (*metis* (*mono-tags*, *lifting*) *commutant-def mem-Collect-eq rangeI cblinfun-apply-cblinfun-compose*)

 obtain *x'* **where** *x'*: ‹*cinner* (*ket j*) (*x' ∗$_V$ ket l*) = *cinner* (*ket* (γ,*j*)) (*x ∗$_V$ ket* (γ,*l*))› **for** *j l*
 proof *atomize-elim*
  obtain ψ **where** ψ: ‹*cinner* (*ket j*) (ψ *l*) = *cinner* (*ket* (γ, *j*)) (*x ∗$_V$ ket* (γ, *l*))› **for** *l j*
   apply (*atomize-elim*, *rule choice*, *rule allI*)
   apply (*rule-tac x*=‹*Abs-ell2* (λ*j*. *cinner* (*ket* (γ, *j*)) (*x ∗$_V$ ket* (γ, *l*)))› **in** *exI*)
   by (*simp add*: *cinner-ket-left Abs-ell2-inverse*)
  obtain *x'* **where** ‹*x' ∗$_V$ ket l* = ψ *l*› **for** *l*
   apply *atomize-elim*
   apply (*rule exI*[*of* - ‹*cblinfun-extension* (*range ket*) (λ*l*. ψ (*inv ket l*))›])
   apply (*subst cblinfun-extension-apply*)
    apply (*rule cblinfun-extension-exists-finite-dim*)
   by (*auto simp add*: *inj-ket cindependent-ket*)
  with ψ **have** ‹*cinner* (*ket j*) (*x' ∗$_V$ ket l*) = *cinner* (*ket* (γ, *j*)) (*x ∗$_V$ ket* (γ, *l*))› **for** *j l*
   by *auto*
  then show ‹∃ *x'*. ∀ *j l*. *cinner* (*ket j*) (*x' ∗$_V$ ket l*) = *cinner* (*ket* (γ, *j*)) (*x ∗$_V$ ket* (γ, *l*))›
   by *auto*
 qed

 have ‹*cinner* (*ket* (*i*,*j*)) (*x ∗$_V$ ket* (*k*,*l*)) = *cinner* (*ket* (*i*,*j*)) ((*id-cblinfun* ⊗$_o$ *x'*) *∗$_V$ ket* (*k*,*l*))› **for** *i j k l*
 proof −

```
    have ‹cinner (ket (i,j)) (x *V ket (k,l))
        = cinner ((butterket i γ ⊗o id-cblinfun) *V ket (γ,j)) (x *V (butterket k γ ⊗o id-cblinfun) *V ket (γ,l))›
      by (auto simp: tensor-op-ket)
    also have ‹... = cinner (ket (γ,j)) ((butterket γ i ⊗o id-cblinfun) *V x *V (butterket k γ ⊗o id-cblinfun) *V
ket (γ,l))›
      by (metis (no-types, lifting) cinner-adj-left butterfly-adjoint id-cblinfun-adjoint tensor-op-adjoint)
    also have ‹... = cinner (ket (γ,j)) (x *V (butterket γ i ⊗o id-cblinfun oCL butterket k γ ⊗o id-cblinfun) *V
ket (γ,l))›
      unfolding comm by (simp add: cblinfun-apply-cblinfun-compose)
    also have ‹... = cinner (ket i) (ket k) * cinner (ket (γ,j)) (x *V ket (γ,l))›
      by (simp add: comp-tensor-op tensor-op-ket tensor-op-scaleC-left)
    also have ‹... = cinner (ket i) (ket k) * cinner (ket j) (x' *V ket l)›
      by (simp add: x')
    also have ‹... = cinner (ket (i,j)) ((id-cblinfun ⊗o x') *V ket (k,l))›
      apply (simp add: tensor-op-ket)
      by (simp flip: tensor-ell2-ket)
    finally show ?thesis by −
  qed
  then have ‹x = (id-cblinfun ⊗o x')›
    by (auto intro!: equal-ket cinner-ket-eqI)
  then show ‹x ∈ range (λb. id-cblinfun ⊗o b)›
    by auto
next
  fix x :: ‹('a × 'b) ell2 ⇒CL ('a × 'b) ell2›
  assume ‹x ∈ range (λb. id-cblinfun ⊗o b)›
  then obtain b where x: ‹x = id-cblinfun ⊗o b›
    by auto
  then show ‹x ∈ commutant (range (λa. a ⊗o id-cblinfun))›
    by (auto simp: x commutant-def comp-tensor-op)
qed

lemma complement-range:
  assumes [simp]: ‹compatible F G› and [simp]: ‹iso-register (F;G)›
  shows ‹range G = commutant (range F)›
proof −
  have [simp]: ‹register F› ‹register G›
    using assms compatible-def by metis+
  have [simp]: ‹(F;G) (a ⊗o b) = F a oCL G b› for a b
    using Laws-Quantum.register-pair-apply assms by blast
  have [simp]: ‹range F = (F;G) ‘ range (λa. a ⊗o id-cblinfun)›
    by force
  have [simp]: ‹range G = (F;G) ‘ range (λb. id-cblinfun ⊗o b)›
    by force
  show ‹range G = commutant (range F)›
    by (simp add: commutant-exchange commutant-tensor1)
qed

lemma same-range-equivalent:
  fixes F :: ‹'a::finite update ⇒ 'c::finite update› and G :: ‹'b::finite update ⇒ 'c::finite update›
  assumes [simp]: ‹register F› and [simp]: ‹register G›
  assumes ‹range F = range G›
  shows ‹equivalent-registers F G›
proof −
  have G-rangeF[simp]: ‹G x ∈ range F› for x
    by (simp add: assms)
  have F-rangeG[simp]: ‹F x ∈ range G› for x
    by (simp add: assms(3)[symmetric])
  have [simp]: ‹inj F› and [simp]: ‹inj G›
    by (simp-all add: register-inj)
  have [simp]: ‹clinear F› ‹clinear G›
    by simp-all
  define I J where ‹I x = inv F (G x)› and ‹J y = inv G (F y)› for x y
  have addI: ‹I (x + y) = I x + I y› for x y
```

unfolding *I-def*
apply (*rule injD*[*OF* ‹*inj F*›])
apply (*subst complex-vector.linear-add*[*OF* ‹*clinear F*›])
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*F*], *simp*)+
by (*simp add*: *complex-vector.linear-add*)
have *addJ*: ‹*J* (*x* + *y*) = *J x* + *J y*› **for** *x y*
unfolding *J-def*
apply (*rule injD*[*OF* ‹*inj G*›])
apply (*subst complex-vector.linear-add*[*OF* ‹*clinear G*›])
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*G*], *simp*)+
by (*simp add*: *complex-vector.linear-add*)
have *scaleI*: ‹*I* (*r* *$_C$ *x*) = *r* *$_C$ *I x*› **for** *r x*
unfolding *I-def*
apply (*rule injD*[*OF* ‹*inj F*›])
apply (*subst complex-vector.linear-scale*[*OF* ‹*clinear F*›])
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*F*], *simp*)+
by (*simp add*: *complex-vector.linear-scale*)
have *scaleJ*: ‹*J* (*r* *$_C$ *x*) = *r* *$_C$ *J x*› **for** *r x*
unfolding *J-def*
apply (*rule injD*[*OF* ‹*inj G*›])
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 *unitalI*: ‹*I id-cblinfun* = *id-cblinfun*›
unfolding *I-def*
apply (*rule injD*[*OF* ‹*inj F*›])
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*F*])
apply *auto*
by (*metis register-of-id G-rangeF assms*(*2*))
have *unitalJ*: ‹*J id-cblinfun* = *id-cblinfun*›
unfolding *J-def*
apply (*rule injD*[*OF* ‹*inj G*›])
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*G*])
apply *auto*
by (*metis register-of-id F-rangeG assms*(*1*))
have *multI*: ‹*I* (*a* *$_{CL}$ *b*) = *I a* *$_{CL}$ *I b*› **for** *a b*
unfolding *I-def*
apply (*rule injD*[*OF* ‹*inj F*›])
apply (*subst register-mult*[*symmetric*, *OF* ‹*register F*›])
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*F*], *simp*)+
by (*simp add*: *register-mult*)
have *multJ*: ‹*J* (*a* *$_{CL}$ *b*) = *J a* *$_{CL}$ *J b*› **for** *a b*
unfolding *J-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 *adjI*: ‹*I* (*a**) = (*I a*)*› **for** *a*
unfolding *I-def*
apply (*rule injD*[*OF* ‹*inj F*›])
apply (*subst register-adjoint*[*OF* ‹*register F*›])
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*F*], *simp*)+
using *assms*(*2*) *register-adjoint* **by** *blast*
have *adjJ*: ‹*J* (*a**) = (*J a*)*› **for** *a*
unfolding *J-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 *assms*(*1*) *register-adjoint* **by** *blast*

from *addI scaleI unitalI multI adjI*
have ‹*register I*›
unfolding *register-def* **by** (*auto intro*!: *clinearI*)

from *addJ scaleJ unitalJ multJ adjJ*
have ‹*register J*›
unfolding *register-def* **by** (*auto intro*!: *clinearI*)

have ‹*I o J = id*›
unfolding *I-def J-def o-def*
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*G*], *simp*)
apply (*subst Hilbert-Choice.inv-f-f*[*OF* ‹*inj F*›])
by *auto*
have ‹*J o I = id*›
unfolding *I-def J-def o-def*
apply (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*F*], *simp*)
apply (*subst Hilbert-Choice.inv-f-f*[*OF* ‹*inj G*›])
by *auto*

from ‹*I o J = id*› ‹*J o I = id*› ‹*register I*› ‹*register J*›
have ‹*iso-register I*›
using *iso-register-def* **by** *blast*

have ‹*F o I = G*›
unfolding *I-def o-def*
by (*subst Hilbert-Choice.f-inv-into-f*[**where** *f*=*F*], *auto*)

with ‹*iso-register I*› **show** *?thesis*
unfolding *equivalent-registers-def* **by** *auto*
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*)

end

18 Generic laws about complements, instantiated quantumly

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

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

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

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

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 *sep4*: ‹*separating TYPE*('*z*::*finite*) {(*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*::*finite update*› **and** *b*::‹'*u*::*finite update*› **and** *c*::‹'*v*::*finite update*› **and** *d*::‹'*w*::*finite 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* ...›
apply (*auto intro*!: *iso-register-comp iso-register-tensor-is-iso-register*)
using *assms complement-is-complement complements-def* **by** *blast*+
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 −
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*; *complement X*))›
apply (*rule equivalent-registers-pair-right*)
apply (*auto intro*!: *unit-register-compatible*)
using *complement-is-complement complements-def equivalent-registersI id-comp register-id that*(*2*) **by** *blast*
also have ‹*equivalent-registers* ... ((*U*; *X*); *complement X*)›
apply (*rule equivalent-registers-assoc*)
by *auto*
finally have ‹*complements* (*U*; *X*) (*complement X*)›
by (*auto simp*: *equivalent-registers-def complements-def*)
moreover have ‹*equivalent-registers* (*X*; *complement X*) *id*›
by (*metis complement-is-complement complements-def equivalent-registers-def iso-register-def that*)
ultimately show *?thesis*
by (*meson complement-unique complement-is-complement complements-sym equivalent-registers-sym equivalent-registers-trans that*)
qed

lemma *unit-register-compose-left*:
assumes [*simp*]: ‹*is-unit-register U*›
assumes [*simp*]: ‹*register A*›
shows ‹*is-unit-register* (*A o U*)›
proof −
have ‹*compatible* (*A o U*) (*A*; *complement A*)›
apply (*auto intro*!: *compatible3'*)
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*) *id*›
by (*metis assms*(*2*) *compatible-comp-right complement-is-complement complements-def iso-register-def*)
have ‹*equivalent-registers* (*A o U*; *id*) (*A o U*; (*A*; *complement A*))›
apply (*auto intro*!: *equivalent-registers-pair-right*)
using *assms*(*2*) *complement-is-complement complements-def equivalent-registers-def id-comp register-id* **by** *blast*
also have ‹*equivalent-registers* ... ((*A o U*; *A o id*); *complement A*)›
apply *auto*
by (*metis* (*no-types*, *opaque-lifting*) *compat assms*(*1*) *assms*(*2*) *compatible-comp-left compatible-def compatible-register1 complement-is-complement complements-def equivalent-registers-assoc id-apply register-unit-register*)
also have ‹*equivalent-registers* ... (*A o* (*U*; *id*); *complement A*)›
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*; *complement A*)›
apply (*intro equivalent-registers-pair-left equivalent-registers-comp*)
apply (*auto simp*: *assms*)
using *assms*(*1*) *equivalent-registers-sym register-id unit-register-pair* **by** *blast*
also have ‹*equivalent-registers* ... *id*›
by (*metis assms*(*2*) *comp-id complement-is-complement complements-def equivalent-registers-def iso-register-inv*

iso-register-inv-comp2 pair-is-register)
finally show *?thesis*
using *compat complementsI equivalent-registers-sym is-unit-register-def iso-register-equivalent-id* **by** *blast*
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-registe*
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*::*finite update* ⇒ '*c*::*finite update*›
fixes *G* :: ‹'*b*::*finite update* ⇒ '*d*::*finite 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-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*›
definition *unit-register* :: ‹*unit-register-domain update* ⇒ '*a*::*finite 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*::*finite update*)›
proof −
let *?U0* = ‹*complement id* :: *unit-register-domain update* ⇒ *unit update*›
let *?U1* = ‹*complement id* :: ('*a*, '*a*) *complement-domain update* ⇒ '*a update*›

have ‹*is-unit-register ?U0*› ‹*is-unit-register ?U1*›
by *auto*
then obtain *I* :: ‹*unit-register-domain update* ⇒ (*'a*, *'a*) *complement-domain update*› **where** ‹*iso-register I*›
apply *atomize-elim* **by** (*rule unit-register-domains-isomorphic*)
with ‹*is-unit-register ?U1*› **have** ‹*is-unit-register* (*?U1 o I*)›
by (*rule unit-register-compose-right*)
then show *?thesis*
by (*metis someI-ex unit-register-def*)
qed

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

proof −
have ‹*equivalent-registers* (*id* :: *'b update* ⇒ -) (*complement id*; *id*)›
using *id-complement-is-unit-register iso-register-equivalent-id register-id unit-register-pair* **by** *blast*
then obtain *J* :: ‹*'b update* ⇒ (((*'b*, *'b*) *complement-domain* ∗ *'b*) *update*)› **where** ‹*iso-register J*›
using *equivalent-registers-def iso-register-inv* **by** *blast*
moreover obtain *K* :: ‹(*'b*, *'b*) *complement-domain update* ⇒ *'a update*› **where** ‹*iso-register K*›
using *assms id-complement-is-unit-register unit-register-domains-isomorphic* **by** *blast*
ultimately have ‹*iso-register* ((*K* ⊗*r id*) *o J*)›
by *auto*
then show *?thesis*
by *auto*
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*›
shows ‹*complements F* (*G*; *complement* (*F*;*G*))›
proof (*rule complementsI*)
have ‹*equivalent-registers* (*F*; (*G*; *complement* (*F*;*G*))) ((*F*;*G*); *complement* (*F*;*G*))›
apply (*rule equivalent-registers-assoc*)
by (*auto simp add*: *compatible-complement-pair1 compatible-complement-pair2*)
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*;*complement* (*F*;*G*)))›
using *equivalent-registers-sym iso-register-equivalent-id* **by** *blast*
show ‹*compatible F* (*G*;*complement* (*F*;*G*))›
using *assms compatible3′ compatible-complement-pair1 compatible-complement-pair2* **by** *blast*
qed

lemma *equivalent-registers-complement*:
assumes ‹*equivalent-registers F G*›
shows ‹*equivalent-registers* (*complement F*) (*complement G*)›
proof −
have ‹*complements F* (*complement F*)›
using *assms complement-is-complement equivalent-registers-register-left* **by** *blast*
with *assms* **have** ‹*complements G* (*complement F*)›
by (*meson complements-sym equivalent-complements*)
then show *?thesis*
by (*rule complement-unique*)
qed

lemma *complements-complement-pair′*:
assumes [*simp*]: ‹*compatible F G*›
shows ‹*complements G* (*F*; *complement* (*F*;*G*))›
proof −
have ‹*equivalent-registers* (*F*;*G*) (*G*;*F*)›
apply (*rule equivalent-registersI*[**where** *I*=*swap*])
by *auto*
then have ‹*equivalent-registers* (*complement* (*F*;*G*)) (*complement* (*G*;*F*))›
by (*rule equivalent-registers-complement*)
then have ‹*equivalent-registers* (*F*; (*complement* (*F*;*G*))) (*F*; (*complement* (*G*;*F*)))›
apply (*rule equivalent-registers-pair-right*[*rotated*])
using *assms compatible-complement-pair1* **by** *blast*
moreover have ‹*complements G* (*F*; *complement* (*G*;*F*))›
apply (*rule complements-complement-pair*)
using *assms compatible-sym* **by** *blast*
ultimately show *?thesis*
by (*meson equivalent-complements equivalent-registers-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** $\otimes_u$ *70*)

end

19 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

definition *empty-var* :: ‹'*a*::{*CARD-1*,*enum*} *update* ⇒ '*b*::*finite update*› **where**
empty-var a = *one-dim-iso a* $*_C$ *id-cblinfun*

lemma *is-unit-register-empty-var*[*register*]: ‹*is-unit-register empty-var*›
proof −
have [*simp*]: ‹*register empty-var*›
unfolding *register-def empty-var-def*
by (*simp add*: *clinearI scaleC-left.add*)
have [*simp*]: ‹*compatible empty-var id*›
by (*auto intro*!: *compatibleI simp*: *empty-var-def*)
have [*simp*]: ‹*iso-register* (*empty-var*;*id*)›
by (*auto intro*!: *same-range-equivalent range-eqI*[**where** *x*=‹*id-cblinfun* $\otimes_o$ -›]
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* :: (*type*, *type*) *default* **..**

end
theory *Pure-States*

imports *Quantum-Extra2 HOL−Eisbach.Eisbach*
begin

definition ‹*pure-state-target-vector F* η_F = (*if ket default* ∈ *range* (*cblinfun-apply* (*F* (*butterfly* η_F η_F)))
then ket default
else (*SOME* η'. *norm* η' = 1 ∧ η' ∈ *range* (*cblinfun-apply* (*F* (*butterfly* η_F η_F)))))›

lemma *pure-state-target-vector-eqI*:
assumes ‹*F* (*butterfly* η_F η_F) = *G* (*butterfly* η_G η_G)›
shows ‹*pure-state-target-vector F* η_F = *pure-state-target-vector G* η_G›
by (*simp add*: *assms pure-state-target-vector-def*)

lemma *pure-state-target-vector-ket-default*: ‹*pure-state-target-vector F* η_F = *ket default*› **if** ‹*ket default* ∈ *range* (*cblinfun-apply* (*F* (*butterfly* η_F η_F)))›
by (*simp add*: *pure-state-target-vector-def that*)

lemma
assumes [*simp*]: ‹$\eta_F \neq 0$› ‹*register F*›
shows *pure-state-target-vector-in-range*: ‹*pure-state-target-vector F* η_F ∈ *range* ((∗$_V$) (*F* (*selfbutter* η_F)))› (**is** *?range*)
and *pure-state-target-vector-norm*: ‹*norm* (*pure-state-target-vector F* η_F) = 1› (**is** *?norm*)
proof −
from *assms* **have** ‹*selfbutter* $\eta_F \neq 0$›
by (*metis butterfly-0-right complex-vector.scale-zero-right inj-selfbutter-upto-phase*)
then have ‹*F* (*selfbutter* η_F) ≠ 0›
using *register-inj*[*OF* ‹*register F*›, *THEN injD*, **where** *y*=*0*]
by (*auto simp*: *complex-vector.linear-0*)
then obtain ψ' **where** ψ': ‹*F* (*selfbutter* η_F) ∗$_V$ 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* η_F)))›
apply (*rule exI*[*of* - ‹(*F* (*selfbutter* η_F) ∗$_V$ ψ') /$_C$ *norm* (*F* (*selfbutter* η_F) ∗$_V$ ψ')›])
using ψ' **apply** (*auto simp add*: *norm-inverse*)
by (*metis cblinfun.scaleC-right rangeI*)
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* ≠ 0›
by *auto*

have *pure-state-target-vector-neq0*: ‹*pure-state-target-vector F* η ≠ 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 ≠ 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› (- - ⊗_p - [1000, 0, 0] 1000)
syntax -pure-tensor2 :: ‹'a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ pure-tensor› (- - ⊗_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 ⊗_s φ)
  -pure-tensor F ψ (CONST pure-state G φ) ⇀ CONST pure-state (F; G) (ψ ⊗_s φ)
  -pure-tensor-start x ⇀ x

  -pure-tensor-start (-pure-tensor2 F ψ G φ) ↽ CONST pure-state (F; G) (ψ ⊗_s φ)
  -pure-tensor F ψ (-pure-tensor2 G φ H η) ↽ -pure-tensor2 F ψ (G;H) (φ ⊗_s η)

term ‹(F ψ ⊗_p G φ ⊗_p H z)›
term ‹pure-state (F; G) (a ⊗_s b)›

lemma register-pair-butterfly-tensor: ‹(F; G) (butterfly (a ⊗_s b) (c ⊗_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) (selfbutterket default ⊗_o a) = selfbutterket
default)›

lemma regular-registerI:
  assumes [simp]: ‹register F›
  assumes [simp]: ‹complements F G›
  assumes eq: ‹(F; G) (selfbutterket default ⊗_o a) = selfbutterket 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
    by (meson Laws-Complement-Quantum.complement-unique equivalent-registers-def equivalent-registers-sym)
```

```
  have ‹(F; complement F) (selfbutterket default ⊗o I a) = (F; G) (selfbutterket default ⊗o a)›
    using cFI by (auto simp: register-pair-apply)
  also have ‹... = selfbutterket 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 [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) (selfbutterket default) = selfbutterket 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) (selfbutterket default) = selfbutterket default ⊗o aG›
    by (metis assms(3) complement-is-complement complements-def inj-iff inv-f-f iso-register-inv-comp1 regu-
lar-register-def)
  define t1 where ‹t1 = inv ((F;G); complement (F;G)) (selfbutterket default)›
  define t2 where ‹t2 = inv (F; (G; complement (F;G))) (selfbutterket default)›
  define t3 where ‹t3 = inv (G; (F; complement (F;G))) (selfbutterket default)›


  have ‹complements F (G; complement (F;G))›
    apply (rule complements-complement-pair)
    by simp
  then have ‹equivalent-registers (complement F) (G; complement (F;G))›
    using Laws-Complement-Quantum.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)) (selfbutterket default)›
    unfolding t2-def I
    apply (subst o-inv-distrib[symmetric])
    by (auto simp: pair-o-tensor)
  also have ‹... = (selfbutterket default ⊗o inv I aF)›
    apply auto
    by (metis ‹iso-register I› id-def iso-register-def iso-register-inv register-id register-tensor-apply)
  finally have t2': ‹t2 = selfbutterket default ⊗o inv I aF›
    by simp

  have ∗: ‹complements G (F; complement (F;G))›
    apply (rule complements-complement-pair')
    by simp
  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* $\otimes_r$ *J*)›
by (*rule iso-register-bij*, *simp*)
have [*simp*]: ‹*inv* (*id* $\otimes_r$ *J*) = *id* $\otimes_r$ *inv J*›
by *auto*

have ‹*t3* = (*inv* (*id* $\otimes_r$ *J*) *o inv* (*G*;*complement G*)) (*selfbutterket default*)›
unfolding *t3-def I*
apply (*subst o-inv-distrib*[*symmetric*])
by (*auto simp*: *pair-o-tensor*)
also have ‹. . . = (*selfbutterket default* $\otimes_o$ *inv J aG*)›
apply *auto*
by (*metis* ‹*iso-register J*› *id-def iso-register-def iso-register-inv register-id register-tensor-apply*)
finally have *t3'*: ‹*t3* = *selfbutterket default* $\otimes_o$ *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* $\otimes_r$ *id*) *o assoc'* = (*G*; (*F*; *complement* (*F*;*G*)))›
apply (*rule tensor-extensionality3*)
apply (*auto intro*!: *register-comp register-tensor-is-register pair-is-register complements-complement-pair simp*: *register-pair-apply compatible-complement-pair1*)
by (*metis assms*(*1*) *cblinfun-assoc-left*(*1*) *swap-registers-left*)
have *t3-t1*: ‹*t3* = *assoc* ((*swap* $\otimes_r$ *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* $\otimes_r$ *id*) *t1*)›
have ∗: ‹*selfbutterket default* $\otimes_o$ *inv J aG* = *assoc* ((*swap* $\otimes_r$ *id*) (*assoc'* (*selfbutterket default* $\otimes_o$ *inv I aF*)))›
by (*simp add*: *t2' t3'*)

have ‹*selfbutterket default* $\otimes_o$ *swap* (*inv J aG*) = (*id* $\otimes_r$ *swap*) (*selfbutterket default* $\otimes_o$ *inv J aG*)›
by *auto*
also have ‹. . . = ((*id* $\otimes_r$ *swap*) *o assoc o* (*swap* $\otimes_r$ *id*) *o assoc'*) (*selfbutterket default* $\otimes_o$ *inv I aF*)›
by (*simp add*: ∗)
also have ‹. . . = (*assoc o swap*) (*selfbutterket default* $\otimes_o$ *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* $\otimes_o$ *selfbutterket default*)›
by *auto*
finally have ∗: ‹*selfbutterket default* $\otimes_o$ *swap* (*inv J aG*) = *assoc* (*inv I aF* $\otimes_o$ *selfbutterket default*)›
by −

obtain *c* **where** ∗: ‹*selfbutterket* (*default*::′*c*) $\otimes_o$ *swap* (*inv J aG*) = *selfbutterket default* $\otimes_o$ *c* $\otimes_o$ *selfbutterket default*›
apply *atomize-elim*
apply (*rule overlapping-tensor*)
using ∗ **unfolding** *assoc-ell2-sandwich sandwich-def*
by *auto*

have ‹*t1* = ((*swap* $\otimes_r$ *id*) *o assoc'*) *t3*›
by (*simp add*: *t3-t1 register-tensor-distrib*[*unfolded o-def*, *THEN fun-cong*] *flip*: *id-def*)
also have ‹. . . = ((*swap* $\otimes_r$ *id*) *o assoc'* *o* (*id* $\otimes_r$ *swap*)) (*selfbutterket* (*default*::′*c*) $\otimes_o$ *swap* (*inv J aG*))›
unfolding *t3'* **by** *auto*
also have ‹. . . = ((*swap* $\otimes_r$ *id*) *o assoc'* *o* (*id* $\otimes_r$ *swap*)) (*selfbutterket default* $\otimes_o$ *c* $\otimes_o$ *selfbutterket default*)›
unfolding ∗ **by** *simp*

```
  also have ‹... = selfbutterket default ⊗o c›
    apply (simp del: tensor-butterfly)
    by (simp add: default-prod-def)
  finally have ‹t1 = selfbutterket default ⊗o c›
    by −

  then show ?thesis
    apply (auto intro!: exI[of - c] simp: regular-register-def t1-def)
    by (metis ‹bij ((F;G);complement (F;G))› bij-inv-eq-iff)
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) (selfbutterket default ⊗o a) = selfbutterket default›
    unfolding regular-register-def by metis
  from that obtain b where b: ‹(G; complement G) (selfbutterket default ⊗o b) = selfbutterket 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)) (selfbutterket default ⊗o c) = ((F o G); (complement F; F o complement
G)) (selfbutterket 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)) (selfbutterket 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') (selfbutterket default ⊗o swap (J c))›
    by auto
  also have ‹... = ((F o (G; complement G); complement F) o assoc') (selfbutterket 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) ((selfbutterket default ⊗o b) ⊗o a)›
    by (simp add: assoc'-apply)
  also have ‹... = (F; complement F) ((G;complement G) (selfbutterket default ⊗o b) ⊗o a)›
    by (simp add: register-pair-apply')
  also have ‹... = selfbutterket default›
    by (auto simp: a b)
  finally have ‹(F ∘ G;complement (F ∘ G)) (selfbutterket default ⊗o c) = selfbutterket 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* (*selfbutterket default*) = *selfbutterket default*›
proof −
from *assms*(*1*) **obtain** *a* **where** *a*: ‹(*F*;*complement F*) (*selfbutterket default* ⊗*o* *a*) = *selfbutterket 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 ‹*selfbutterket default* = (*F*; *?u o I*) (*selfbutterket default* ⊗*o* *a*)›
using ‹*complement F* = *empty-var* ∘ *I*› *a* **by** *presburger*
also have ‹. . . = (*F*; *?u*) (*selfbutterket 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*) (*selfbutterket default* ⊗*o* (*one-dim-iso* (*I a*) ∗*C* *id-cblinfun*))›
by *simp*
also have ‹. . . = *one-dim-iso* (*I a*) ∗*C* (*F*; *?u*) (*selfbutterket 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* (*selfbutterket 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* (*selfbutterket default*) = *selfbutterket 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 ‹*selfbutterket default* = *one-dim-iso* (*I a*) ∗*C* *F* (*selfbutterket default*)›
using *F* **by** *simp*
also have ‹. . . = *one-dim-iso* (*I a*) ∗*C* *F* (*selfbutterket default* o*CL* *selfbutterket default*)›
by *auto*
also have ‹. . . = *one-dim-iso* (*I a*) ∗*C* (*F* (*selfbutterket default*) o*CL* *F* (*selfbutterket default*))›
by (*simp add*: *assms*(*2*) *iso-register-is-register register-mult*)
also have ‹. . . = *one-dim-iso* (*I a*) ∗*C* ((*selfbutterket default* /*C* *one-dim-iso* (*I a*)) o*CL* (*selfbutterket 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* *selfbutterket 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* (*selfbutterket default*) = *selfbutterket 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* (*selfbutterket default*) = *selfbutterket 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 auto
    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 selfbutterket 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 (selfbutterket default)›
    by simp
  also have ‹... = H (butterfly h (ket default) oCL selfbutterket 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=‹selfbutterket default› and G=Snd])
  by (auto simp: pair-Fst-Snd default-prod-def)

lemma Snd-regular[simp]: ‹regular-register Snd›
  apply (rule regular-registerI[where a=‹selfbutterket default› and G=Fst])
    apply auto[2]
  apply (auto simp only: default-prod-def swap-apply simp flip: swap-def)
  by auto

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-def c-def cinner-adj-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-def cinner-adj-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*)
finally show *?thesis*
by −
qed

lemma *cspan-pure-state*:
assumes [*simp*]: ‹*iso-register F*›
assumes ‹*cspan* (*g* ‘ *X*) = *UNIV*›
shows ‹*cspan* ((*λz. 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* *o*$_{CL}$ (*G*;*F*) *CNOT* *o*$_{CL}$ (*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 complements-complement-pair complements-def* **by** *blast*

have *eq1*: ‹((*F*;*G*) *CNOT* *o*$_{CL}$ (*G*;*F*) *CNOT* *o*$_{CL}$ (*F*;*G*) *CNOT*) ∗$_V$ (*F*(*ket f*) ⊗*p* *G*(*ket g*) ⊗*p* *Z*(*ket z*))
= (*F*;*G*) *swap-ell2* ∗$_V$ (*F*(*ket f*) ⊗*p* *G*(*ket g*) ⊗*p* *Z*(*ket z*))› **for** *f g z*
proof −
have ‹(*F*(*ket f*) ⊗*p* *G*(*ket g*) ⊗*p* *Z*(*ket z*)) = ((*F*;*G*) (*ket f* ⊗*s* *ket g*) ⊗*p* *Z*(*ket z*))›
by *pure-state-eq*
also have ‹(*F*;*G*) *CNOT* ∗$_V$ ... = ((*F*;*G*) (*ket f* ⊗*s* *ket* (*g*+*f*)) ⊗*p* *Z*(*ket z*))›
apply (*subst state-apply1*) **by** *auto*
also have ‹... = ((*G*;*F*) (*ket* (*g*+*f*) ⊗*s* *ket f*) ⊗*p* *Z*(*ket z*))›
by *pure-state-eq*
also have ‹(*G*;*F*) *CNOT* ∗$_V$ ... = ((*G*;*F*) (*ket* (*g*+*f*) ⊗*s* *ket g*) ⊗*p* *Z ket z*)›
apply (*subst state-apply1*) **by** *auto*
also have ‹... = ((*F*;*G*) (*ket g* ⊗*s* *ket* (*g*+*f*)) ⊗*p* *Z ket z*)›
by *pure-state-eq*
also have ‹(*F*;*G*) *CNOT* ∗$_V$ ... = ((*F*;*G*) *ket g* ⊗*s* *ket f* ⊗*p* *Z ket z*)›
apply (*subst state-apply1*)
apply *simp*
using *add-right-imp-eq* **by** *fastforce*
also have ‹... = (*F*(*ket g*) ⊗*p* *G*(*ket f*) ⊗*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*) ⊗*p* *G*(*ket g*) ⊗*p* *Z*(*ket z*)) = (*F*(*ket g*) ⊗*p* *G*(*ket f*) ⊗*p* *Z*(*ket z*))›
by *auto*

have ‹(*F*(*ket f*) ⊗*p* *G*(*ket g*) ⊗*p* *Z*(*ket z*)) = ((*F*;*G*) (*ket f* ⊗*s* *ket g*) ⊗*p* *Z*(*ket z*))›
by *pure-state-eq*
also have ‹(*F*;*G*) *swap-ell2* ∗$_V$ ... = ((*F*;*G*) (*ket g* ⊗*s* *ket f*) ⊗*p* *Z*(*ket z*))›

```
    by (auto simp: state-apply1 swap-ell2-tensor simp del: tensor-ell2-ket)
  also have ‹... = (F(ket g) ⊗p G(ket f) ⊗p Z(ket z))›
    by pure-state-eq
  finally have 2: ‹(F;G) swap-ell2 *V (F(ket f) ⊗p G(ket g) ⊗p Z(ket z)) = (F(ket g) ⊗p G(ket f) ⊗p Z(ket z))›
    by −

  from 1 2 show ?thesis
    by simp
qed

then have eq1: ‹((F;G) CNOT oCL (G;F) CNOT oCL (F;G) CNOT) *V ψ
      = (F;G) swap-ell2 *V ψ› if ‹ψ ∈ {(F(ket f) ⊗p G(ket g) ⊗p Z(ket z))| f g z. True}› for ψ
  using that by auto

moreover have ‹cspan {(F(ket f) ⊗p G(ket g) ⊗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

ultimately show ?thesis
  using cblinfun-eq-on-UNIV-span by blast
qed

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 f4a0dac97bed23ec5b7c4cbf779f8eb2a12aa488;
  check Source Axioms-Quantum.thy b1ac4a827c2b943202c03611176d3c723119d8e1;
  check Source Laws.thy 37803f67bcda2df6bf7abe2417d3bf49e6317dcd;
  check Source Laws-Complement.thy 6065101853fa432ca4bd6fd3113315e856e21ecb;
  check Generated Laws-Classical.thy 051bc83ae9bd061fba08bfa29468a3421817068b;
  check Generated Laws-Complement-Quantum.thy c58ba1680287643d1c3f9b2b97d9784db8e1dd84;
  check Generated Laws-Quantum.thy 36d05e686993e4f9b7c5bf57639d840b3e9b2e47
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.

[Unr21] Dominique Unruh. Quantum and classical registers. arXiv:2105.10914 [cs.LO], 2021.